sapientia
textbooks



Dávid Iclănzan
Zoltán Kátai
Emil Horobeţ

ECN Problems: A Journey into Competitive Programming

Scientia Publishing House

ECN PROBLEMS: A JOURNEY INTO COMPETITIVE PROGRAMMING

EDITED BY:
DÁVID ICLĂNZAN
ZOLTÁN KÁTAI
EMIL HOROBEŢ



SAPIENTIA HUNGARIAN UNIVERSITY OF TRANSYLVANIA
FACULTY OF TÂRGU MUREȘ
DEPARTMENT OF MATHEMATICS–INFORMATICS

ECN PROBLEMS: A JOURNEY INTO COMPETITIVE PROGRAMMING

EDITED BY:
DÁVID ICLĂNZAN
ZOLTÁN KÁTAI
EMIL HOROBEŢ

Scientia Publishing House
Cluj-Napoca · 2026

sapientia textbooks **computer sciences**

Készült
a Magyar Kormány
Támogatásával

Társadalmi Kapcsolatokért
és Kultúráért Felelős Minisztérium

Bethlen Gábor
Alap

DOI: 10.47745/SAPVOL.2026.32

Publisher in charge:
Angella Sorbán

Consultant:
Lajos Losonczi (Târgu-Mureș)

Publishing coordinator:
Beáta Szabó

First edition: 2026
© Scientia 2026
All rights reserved, including reproduction, public performance, radio and television broadcasting, or translation. This applies also to individual chapters.

Descrierea CIP a Bibliotecii Naţionale a României
ICLĂNZAN, DÁVID
ECN problems : A journey into competitive programming / Dávid Iclănzan, Zoltán Kátai, Emil Horobeţ. - Cluj-Napoca : Scientia, 2026
ISBN 978-606-975-120-6

I. Kátai, Zoltán
II. Horobeţ, Emil
004

CONTENTS

Preface **23**

I INFORMATICS

1 The Past and Present of Competitive Programming **27**

1.1 Competitive Programming in Romania 27
1.2 Problem Types and Scoring Systems 29
1.3 Contest Types 31
1.4 ICPC Tactics and Strategies 32
1.5 Preparation for Competitive Programming Contests 37

2 Dynamic Programming **44**

2.1 Introduction 44
2.2 DP Diversity 45
2.2.1 "Value – to – Value" DP Problems 45
2.2.2 "Subsequence – to – Subsequence" DP Problems 46
2.2.3 "Subsequence Pair – to – Subsequence Pair" DP Problems 48
2.2.4 "Subsegment – to – Subsegment" DP Problems 50
2.2.5 "Subset – to – Subset" DP Problems 52
2.2.6 "Subtree – to – Subtree" DP Problems 54
2.3 Recipe to Address DP Problems 56
2.4 The Pyramid (ECN-2017) 57
2.4.1 Rationale and Solution 58
2.5 Fixing Bugs (ECN-2016) 64
2.6 The Mosaic (ECN-2011) 65
2.6.1 Rationale and Solution 67

3 Divide et Impera **73**

3.1 The Divide et Impera Approach 73
3.2 The Towers of Hanoi Problem 75
3.2.1 The Legend 75
3.2.2 The Divide et Impera Solution 76
3.3 ECN-2009, Problem D 78

3.3.1 Problem Statement 78
3.3.2 The Solution 79
3.4 ECN-2010, Problem A 80
3.4.1 Problem Statement 80
3.4.2 The Solution 82
3.4.3 The Algorithm 82
3.5 ECN-2013, Problem H 83
3.5.1 Problem Statement 83
3.5.2 The Solution 84
3.5.3 Program Parts 86
3.6 Divide et Impera – Advantages and Disadvantages 89

4 Sorting and Searching 93

5 Algorithmic Explorations in Combinatorial Challenges: Unraveling Complexity 116
5.1 Introduction 116
5.2 Mountain Permutation 117
5.3 Mountain String 124

6 Number Theory and Combinatorics 132
6.1 Farey Series – 2006, Round 1, Problem G 137
6.2 Arrays – 2007, Round 1, Problem G 138
6.3 Sums of Consecutive Integers – 2007, Round 2, Problem G 140
6.4 Decomposition of Factorial – 2007, Round 3, Problem E 140
6.5 Maximum Border – 2007, Round 3, Problem G 141
6.6 Regular Numbers – 2008, Problem C 142
6.7 Divisibility – 2010, Problem H 142
6.8 Primitive Roots – 2011, Problem H 143
6.9 Simple Encryption – 2011, Problem I 144
6.10 Tenders's Invitations – 2012, Problem G 145
6.11 Decryptions – 2013, Problem I 147
6.12 Hash Values – 2015, Problem K 149
6.13 Addition of Points – 2018, Problem G 150

7 ECN-2023 Problems 155
7.1 Problem A 155
7.1.1 Problem Statement 155

7.1.2 The Key Idea 157
7.1.3 Background 157
7.1.4 Representing Graphs 158
7.1.5 Traversing Graphs 159
7.1.6 Lowest Common Ancestors 161
7.1.7 Algorithmic Approach 162
7.2 Problem B 167
7.2.1 Problem Statement 167
7.2.2 The Key Idea 169
7.2.3 Background 170
7.2.3.1 The Original Problem 170
7.2.4 Algorithmic Approach 173
7.3 Problem C 176
7.3.1 Problem Statement 176
7.3.2 The Key Idea 176
7.3.3 Background 177
7.3.4 Algorithmic Approach 178
7.4 Problem D 179
7.4.1 Problem Statement 179
7.4.2 The Key Idea 181
7.4.3 Background 182
7.4.4 Algorithmic Approach 183
7.4.5 Implementation(s) 184
7.4.5.1 The Mathematica Code 185
7.4.5.2 The C++ Code 186
7.5 Problem E 190
7.5.1 Problem Statement 190
7.5.2 The Key Idea 192
7.5.3 Background 192
7.5.3.1 Range Trees 192
7.5.3.2 Applications 192
7.5.4 Algorithmic Approach 194
7.6 Problem F 199
7.6.1 Problem Statement 199
7.6.2 The Key Idea 200
7.6.3 Code Implementation 201
7.6.4 A Simplified Version with Extra Explanations 206
7.7 Problem G 211
7.7.1 Problem Statement 211

7.7.2 The Key Idea 213
7.7.3 Background 214
7.7.4 Algorithmic Approach 214
7.8 Problem H 216
7.8.1 Problem Statement 216
7.8.2 The Key Idea 218
7.8.3 Background 218
7.8.4 Algorithmic Approach 221
7.8.5 Code Implementation 222
7.9 Problem I 225
7.9.1 Problem Statement 225
7.9.2 The Key Idea 226
7.9.3 Background 227
7.9.3.1 Applications 227
7.9.4 Algorithmic Approach 228
7.9.5 Supplementary Explanations 234
7.10 Problem J 237
7.10.1 Problem Statement 237
7.10.2 The Key Idea 239
7.10.3 Background 240
7.10.4 Algorithmic Approach 240
7.11 Problem K 243
7.11.1 Problem Statement 243
7.11.2 The Key Idea 247
7.11.3 Background 247
7.11.4 Algorithmic Approach 248
7.12 Problem L 255
7.12.1 Problem Statement 255
7.12.2 The Key Idea 257
7.12.3 Background 258
7.12.4 Algorithmic Approach 260
7.13 Problem M 271
7.13.1 Problem Statement 271
7.13.2 The Key Idea 273
7.13.3 Background 273
7.13.4 Algorithmic Approach 275
7.13.5 Subsequent Lessons 278
7.14 Problem N 283
7.14.1 Problem Statement 283

7.14.2 The Key Idea 286
7.14.3 Background 286
7.14.4 Algorithmic Approach 287

II MATHEMATICS

2 Selected Problems from "Bege Antal" Memorial Mathematics Contest 298

2.1 Combinatorics 298
2.2 Number Theory Problems 307
2.3 Geometry Problems 312

Kivonat 326

Rezumat 327

About the Authors 328

TARTALOMJEGYZÉK

Előszó **23**

I. Informatika

1. A versenyprogramozás múltja és jelene **27**

1.1. Versenyprogramozás Romániában 27
1.2. Feladattípusok és pontozási rendszerek 29
1.3. Versenytípusok 31
1.4. ICPC-taktikák és -stratégiák 32
1.5. Felkészülés versenyprogramozási versenyekre 37

2. Dinamikus programozás **44**

2.1. Bevezetés 44
2.2. A DP sokfélesége 45
2.2.1. „Értéktől értékig” DP-feladatok 45
2.2.2. „Részsorozattól részsorozatig” 46
2.2.3. „Részsorozatpártól részsorozatpárig” DP-feladatok 48
2.2.4. „Részszakasztól részszakaszig” DP-feladatok 50
2.2.5. „Részhalmaztól részhalmazig” DP-feladatok 52
2.2.6. „Részfától részfáig” DP-feladatok 54
2.3. Recept DP-feladatok megoldásához 56
2.4. A piramis (ECN-2017) 57
2.4.1. Indoklás és megoldás 58
2.5. Hibák javítása (ECN-2016) 64
2.6. A mozaik (ECN-2011) 65
2.6.1. Indoklás és megoldás 67

3. Oszd meg és uralkodj **73**

3.1. Az oszd meg és uralkodj megközelítés 73
3.2. A Hanoi tornyai feladat 75
3.2.1. A legenda 75
3.2.2. Az oszd meg és uralkodj megoldás 76
3.3. ECN-2009, D feladat 78
3.3.1. Feladatleírás 78

3.3.2. A megoldás 79
3.4. ECN-2010, A feladat 80
3.4.1. Feladatleírás 80
3.4.2. A megoldás 82
3.4.3. Az algoritmus 82
3.5. ECN-2013, H feladat 83
3.5.1. Feladatleírás 83
3.5.2. A megoldás 84
3.5.3. Programrészek 86
3.6. Oszd meg és uralkodj – előnyök és hátrányok 89

4. Rendezés és keresés 93

5. Algoritmikus feltárások kombinatorikai kihívásokban: a komplexitás feltárása 116
5.1. Bevezetés 116
5.2. Hegy-permutáció 117
5.3. Hegy-karaktersorozat 124

6. Számelmélet és kombinatorika 132
6.1. Farey-sorozat – 2006, 1. forduló, G feladat 137
6.2. Tömbök – 2007, 1. forduló, G feladat 138
6.3. Egymást követő egész számok összegei – 2007, 2. forduló, G feladat 140
6.4. Faktoriális felbontása – 2007, 3. forduló, E feladat 140
6.5. Maximális határ – 2007, 3. forduló, G feladat 141
6.6. Reguláris számok – 2008, C feladat 142
6.7. Oszthatóság – 2010, H feladat 142
6.8. Primitív gyökök – 2011, H feladat 143
6.9. Egyszerű titkosítás – 2011, I feladat 144
6.10. Az ajánlattevők meghívásai – 2012, G feladat 145
6.11. Visszafejtések – 2013, I feladat 147
6.12. Hash-értékek – 2015, K feladat 149
6.13. Pontok összeadása – 2018, G feladat 150

7. ECN-2023 feladatok 155
7.1. A feladat 155
7.1.1. Feladatleírás 155

7.1.2. Kulcsötlet 157
7.1.3. Háttér 157
7.1.4. Gráfok ábrázolása 158
7.1.5. Gráfok bejárása 159
7.1.6. Legközelebbi közös ősök 161
7.1.7. Algoritmikus megközelítés 162
7.2. B feladat 167
7.2.1. Feladatleírás 167
7.2.2. Kulcsötlet 169
7.2.3. Háttér 170
7.2.3.1. Az eredeti feladat 170
7.2.4. Algoritmikus megközelítés 173
7.3. C feladat 176
7.3.1. Feladatleírás 176
7.3.2. Kulcsötlet 176
7.3.3. Háttér 177
7.3.4. Algoritmikus megközelítés 178
7.4. D feladat 179
7.4.1. Feladatleírás 179
7.4.2. Kulcsötlet 181
7.4.3. Háttér 182
7.4.4. Algoritmikus megközelítés 183
7.4.5. Implementáció(k) 184
7.4.5.1. Mathematica-kód 185
7.4.5.2. C++-kód 186
7.5. E feladat 190
7.5.1. Feladatleírás 190
7.5.2. Kulcsötlet 192
7.5.3. Háttér 192
7.5.3.1. Tartományfák 192
7.5.3.2. Alkalmazások 192
7.5.4. Algoritmikus megközelítés 194
7.6. F feladat 199
7.6.1. Feladatleírás 199
7.6.2. Kulcsötlet 200
7.6.3. Kódimplementáció 201
7.6.4. Egyszerűsített változat további magyarázatokkal 206
7.7. G feladat 211
7.7.1. Feladatleírás 211

7.7.2. Kulcsötlet 213
7.7.3. Háttér 214
7.7.4. Algoritmikus megközelítés 214
7.8. H feladat 216
7.8.1. Feladatleírás 216
7.8.2. Kulcsötlet 218
7.8.3. Háttér 218
7.8.4. Algoritmikus megközelítés 221
7.8.5. Kódimplementáció 222
7.9. I feladat 225
7.9.1. Feladatleírás 225
7.9.2. Kulcsötlet 226
7.9.3. Háttér 227
7.9.3.1. Alkalmazások 227
7.9.4. Algoritmikus megközelítés 228
7.9.5. Kiegészítő magyarázatok 234
7.10. J feladat 237
7.10.1. Feladatleírás 237
7.10.2. Kulcsötlet 239
7.10.3. Háttér 240
7.10.4. Algoritmikus megközelítés 240
7.11. K feladat 243
7.11.1. Feladatleírás 243
7.11.2. Kulcsötlet 247
7.11.3. Háttér 247
7.11.4. Algoritmikus megközelítés 248
7.12. L feladat 255
7.12.1. Feladatleírás 255
7.12.2. Kulcsötlet 257
7.12.3. Háttér 258
7.12.4. Algoritmikus megközelítés 260
7.13. M feladat 271
7.13.1. Feladatleírás 271
7.13.2. Kulcsötlet 273
7.13.3. Háttér 273
7.13.4. Algoritmikus megközelítés 275
7.13.5. További tanulságok 278
7.14. N feladat 283
7.14.1. Feladatleírás 283

7.14.2. Kulcsötlet 286
7.14.3. Háttér 286
7.14.4. Algoritmikus megközelítés 287

II. Matematika

2. Válogatott feladatok a Bege Antal emlék-matematikaversenyről 298

2.1. Kombinatorika 298
2.2. Számelméleti feladatok 307
2.3. Geometriai feladatok 312

Kivonat 326

Szerzökről 328

CUPRINS

Prefață **23**

I. Informatică

1. Trecutul și prezentul programării competitive **27**

1.1. Programarea competitivă în România 27
1.2. Tipuri de probleme și sisteme de punctare 29
1.3. Tipuri de concursuri 31
1.4. Tactici și strategii ICPC 32
1.5. Pregătirea pentru concursurile de programare competitivă 37

2. Programare dinamică **44**

2.1. Introducere 44
2.2. Diversitatea programării dinamice 45
2.2.1. Probleme DP de tip „Valoare – la – valoare” 45
2.2.2. Probleme DP de tip „Subsecvență – la – subsecvență” 46
2.2.3. Probleme DP de tip „Pereche de subsecvențe – la – pereche de subsecvențe” 48
2.2.4. Probleme DP de tip „Subsegment – la – subsegment” 50
2.2.5. Probleme DP de tip „Submulțime – la – submulțime” 52
2.2.6. Probleme DP de tip „Subarbore – la – subarbore” 54
2.3. Rețetă pentru abordarea problemelor 56
2.4. Piramida (ECN-2017) 57
2.4.1. Raționament și soluție 58
2.5. Corectarea erorilor (ECN-2016) 64
2.6. Mozaicul (ECN-2011) 65
2.6.1. Raționament și soluție 67

3. Divide et Impera **73**

3.1. Abordarea Divide et Impera 73
3.2. Problema Turnurilor din Hanoi 75
3.2.1. Legenda 75
3.2.2. Soluția Divide et Impera 76
3.3. ECN-2009, Problema D 78

3.3.1. Enunțul problemei 78
3.3.2. Soluția 79
3.4. ECN-2010, Problema A 80
3.4.1. Enunțul problemei 80
3.4.2. Soluția 82
3.4.3. Algoritmul 82
3.5. ECN-2013, Problema H 83
3.5.1. Enunțul problemei 83
3.5.2. Soluția 84
3.5.3. Părți ale programului 86
3.6. Divide et Impera – avantaje și dezavantaje 89

4. Sortare și căutare 93

5. Explorări algoritmice în provocări combinatoriale: Descifrarea complexității 116
5.1. Introducere 116
5.2. Permutare de tip munte 117
5.3. Șir de tip munte 124

6. Teoria numerelor și combinatorică 132
6.1. Șirul lui Farey – 2006, Runda 1, Problema G 137
6.2. Tablouri – 2007, Runda 1, Problema G 138
6.3. Sume de numere întregi consecutive – 2007, Runda 2, Problema G 140
6.4. Descompunerea factorialului – 2007, Runda 3, Problema E 140
6.5. Bordură maximă – 2007, Runda 3, Problema G 141
6.6. Numere regulate – 2008, Problema C 142
6.7. Divizibilitate – 2010, Problema H 142
6.8. Rădăcini primitive – 2011, Problema H 143
6.9. Criptare simplă – 2011, Problema I 144
6.10. Invitațiile ofertanților – 2012, Problema G 145
6.11. Decriptări – 2013, Problema I 147
6.12. Valori hash – 2015, Problema K 149
6.13. Adunarea punctelor – 2018, Problema G 150

7. Probleme ECN-2023 155
7.1. Problema A 155

7.1.1. Enunțul problemei 155
7.1.2. Ideea-cheie 157
7.1.3. Fundamente 157
7.1.4. Reprezentarea grafurilor 158
7.1.5. Parcurgerea grafurilor 159
7.1.6. Cei mai apropiați strămoși comuni 161
7.1.7. Abordare algoritmică 162
7.2. Problema B 167
7.2.1. Enunțul problemei 167
7.2.2. Ideea-cheie 169
7.2.3. Fundamente 170
7.2.3.1. Problema originală 170
7.2.4. Abordare algoritmică 173
7.3. Problema C 176
7.3.1. Enunțul problemei 176
7.3.2. Ideea-cheie 176
7.3.3. Fundamente 177
7.3.4. Abordare algoritmică 178
7.4. Problema D 179
7.4.1. Enunțul problemei 179
7.4.2. Ideea-cheie 181
7.4.3. Fundamente 182
7.4.4. Abordare algoritmică 183
7.4.5. Implementare/implementări 184
7.4.5.1. Codul Mathematica 185
7.4.5.2. Codul C++ 186
7.5. Problema E 190
7.5.1. Enunțul problemei 190
7.5.2. Ideea-cheie 192
7.5.3. Fundamente 192
7.5.3.1. Arbori de intervale 192
7.5.3.2. Aplicații 192
7.5.4. Abordare algoritmică 194
7.6. Problema F 199
7.6.1. Enunțul problemei 199
7.6.2. Ideea-cheie 200
7.6.3. Implementarea codului 201
7.6.4. O versiune simplificată cu explicații suplimentare 206
7.7. Problema G 211

7.7.1. Enunțul problemei 211
7.7.2. Ideea-cheie 213
7.7.3. Fundamente 214
7.7.4. Abordare algoritmică 214
7.8. Problema H 216
7.8.1. Enunțul problemei 216
7.8.2. Ideea-cheie 218
7.8.3. Fundamente 218
7.8.4. Abordare algoritmică 221
7.8.5. Implementarea codului 222
7.9. Problema I 225
7.9.1. Enunțul problemei 225
7.9.2. Ideea-cheie 226
7.9.3. Fundamente 227
7.9.3.1. Aplicații 227
7.9.4. Abordare algoritmică 228
7.9.5. Explicații suplimentare 234
7.10. Problema J 237
7.10.1. Enunțul problemei 237
7.10.2. Ideea-cheie 239
7.10.3. Fundamente 240
7.10.4. Abordare algoritmică 240
7.11. Problema K 243
7.11.1. Enunțul problemei 243
7.11.2. Ideea-cheie 247
7.11.3. Fundamente 247
7.11.4. Abordare algoritmică 248
7.12. Problema L 255
7.12.1. Enunțul problemei 255
7.12.2. Ideea-cheie 257
7.12.3. Fundamente 258
7.12.4. Abordare algoritmică 260
7.13. Problema M 271
7.13.1. Enunțul problemei 271
7.13.2. Ideea-cheie 273
7.13.3. Fundamente 273
7.13.4. Abordare algoritmică 275
7.13.5. Lecții ulterioare 278
7.14. Problema N 283

7.14.1. Enunțul problemei 283
7.14.2. Ideea-cheie 286
7.14.3. Fundamente 286
7.14.4. Abordare algoritmică 287

II. Matematică

2. Probleme selectate de la Concursul Memorial de Matematică „Bege Antal” 298

2.1. Combinatorică 298
2.2. Probleme de teoria numerelor 307
2.3. Probleme de geometrie 312

Rezumat 326

Despre autori 328

PREFACE

"Change is the law of life, and those who look only to the past or present are certain to miss the future."

— John F. Kennedy

We are living through a profound transformation that is mainly caused by the rapid progress of artificial intelligence technologies. AI-powered tools become increasingly adept at handling routine coding tasks, altering the professional landscape, career paths for aspiring and established programmers.

You might have encountered headlines and editorials about shrinking or changing job markets, but this transformation also brings substantial new opportunities. The fundamental abilities that make programmers outstanding, namely critical thinking along with complex problem-solving abilities and elegant, efficient algorithm design, are becoming even more valuable. AI provides assistance in code generation, but human creativity combined with analytical depth and innovative problem-solving capabilities still remains irreplaceable. It is these very skills that competitive programming develops and cultivates with high efficacy.

Practicing competitive programming brings essential benefits. You develop algorithmic thinking, which means decomposing complex problems into manageable segments and devising step-by-step strategies for solutions. It also helps you master data structures, understanding how to organize data effectively for optimal performance. You will also develop the skill set to write efficient codes, to solve problems in a resource-conscious way, an ability highly appreciated in any software engineering role. Finally, competitive programming builds your ability to perform under pressure, to work efficiently under time constraints, which prepares you for real development sprints and critical debugging sessions.

This book, *ECN Problems: A Journey into Competitive Programming*, is designed as both an educational resource and a motivational support

for your competitive programming development. The Sapientia ECN (Efficiency Challenge Networking) Programming and Mathematics Contest has challenged and inspired young minds since 2006, and the selected problems presented here reflect that tradition. Going beyond simply compiling tasks and solutions, the book provides a structured approach to building the mindset of a competitive programmer.

The opening chapter presents invaluable content and advice from a distinguished expert who brings extensive competitive programming experience to the table. This section provides readers with essential guidance together with fundamental preparation methods in company with contest strategies and mindsets required to achieve excellence. Every chapter following this introduction features content from different authors who have experience working with the Sapientia ECN contest as either problem proponents or mentors, coaches. The chapters present problems from various years of the ECN contest along with an in-depth analysis, strategies, mindsets, and lessons learned along the way. You will begin by studying essential theoretical concepts which include algorithms and data structures needed for the problem solution. The theoretical framework becomes practical through detailed explanations that demonstrate how to analyze and solve ECN problems. The learning approach combines general instruction with contest-specific examples to create an effective, practical educational content. A final section extends this learning experience to mathematical problem-solving through solved problems from the "Bege Antal Memorial Contest", which serves as the ECN contest's mathematics section. We hope this chapter reinforces the interplay between algorithmic and mathematical reasoning.

The journey begins with a single problem. Progressing through the material step by step, contest by contest, you will learn and grow to become a problem-solver who can architect robust, elegant, and optimized solutions. Competitive programming serves as systematic training in adaptability, in confronting unfamiliar challenges with confidence and rigor, and in transforming challenges that seem insurmountable at first into structured solutions.

The future of software development remains bright for those who can think deeply and solve creatively. Let this book be a stepping stone on your path to becoming one of them.

Dávid Iclănzan
Editor

PART I
INFORMATICS

CHAPTER 1

THE PAST AND PRESENT OF COMPETITIVE PROGRAMMING

Author: Csaba György Pătcaș

Since the first competitive programming contests in the 1970s, the field has gone through important changes and is now more popular than ever. There are numerous offline and online contests for every age and experience level, each having its unique set of rules, problem types, and scoring systems. Some of them are competitions among teams, such as *Sapientia ECN*, a contest organized in Romania that closely follows the rules of the *International Collegiate Programming Contest*. Having three team members work on a single computer gives a unique dynamic to the contest, and in this setting, optimal team strategy is not trivial. We present tactics and strategies from the published literature, adding our own remarks based on decades of experience in competitive programming, where necessary. At the end of the chapter, we give a comprehensive list of resources that can help in preparing for competitive programming contests and offer some advice on how to use them to practice efficiently.

1.1 Competitive Programming in Romania

Romania has an important tradition in science contests. Even before informatics became a subject taught in schools, the first International Mathematical Olympiad was organized in Romania in 1959 and has since been won five times by the Romanian team.

In 1971, the first informatics high schools were founded in the major cities of the country, and the first Olympiad in Informatics was organized in 1978 [1].

Two of the most important international computer science contests for high school students were held for the first time in the country. The first edition of the *Balkan Olympiad in Informatics (BOI)* was held in 1993 on

the shores of the Black Sea, in Constanța, then in Iași in 2003, in 2011 in Bistrița, and in Timișoara in 2018. The *Central European Olympiad in Informatics (CEOI)* took place in Cluj-Napoca in 1994, then again in 2000, two additional editions being held in Târgu-Mureș in 2009 and Piatra Neamț in 2016.

Looking at the historical results of the most important international competition for high school students, the *International Olympiad in Informatics (IOI)*, Romania is the 6$^{\text{th}}$ strongest nation worldwide, with 32 gold, 53 silver, and 34 bronze medals as of 2021. Two times has a Romanian student managed the rare performance of achieving a perfect score: Radu Lupșa in 1993 and Mihai Stroe in 1998. Also remarkable are the four gold medals won by Rareș Buhai between 2012 and 2015 and the second place won by Mihai Pătrașcu in 2001 [2].

One of the reasons for this success is most probably the journal called *Gazeta de Informatică* (later *GInfo*), which was unique worldwide between 1991 and 2007. It published articles about algorithms and data structures, problems, and solutions of national and international competitive programming contests [3].

As the Internet became widely available, new convenient online resources were created for the Romanian competitive programming community. In those early years, probably the most important one was the email list started by IOI silver medalist Cătălin Frâncu in 1998 [4]. In 2004, probably the most influential resource since then, *infoarena* was born, which is a unique online judge system able to give much more detailed feedback on submissions compared to the well-known international ones. *Algoritmiada*, one of the biggest online contests in the country, is also organized on this site. Other important online contests for the Romanian community are *Romanian Master of Informatics* [5] (since 2014), *International Informatics Olympiad in Teams* [6] and *inf0(1) cup* [7] (since 2017). Also worth mentioning are the educational resources at *.campion*, and *pbinfo* [8, 9].

For university-aged students, arguably the most important international contest in competitive programming is the *International Collegiate Programming Contest* (*ICPC*, until 2017 *ACM-ICPC*). Romania is part of the Southeastern European region, along with Albania, Bulgaria, Croatia, Greece, Moldova, Turkey, Lebanon, Israel, Macedonia, Cyprus, Ukraine, and Serbia [10]. Since 1995, teams from Romanian universities qualified 22 times for the World Finals and managed to place in top 10 four times (in 1998 and 2005 the team of the Polytechnic University of Bucharest and in 1999 and 2020 the team of the University of Bucharest). In 2019, a regular

participant at *Sapientia ECN*, the team representing Alexandru Ioan Cuza University Iași qualified for the first time for the World Finals [11].

1.2 Problem Types and Scoring Systems

The scoring of programming contests went through big changes during their history. In the early days, judges had to score tests written on paper, which inadvertently introduced the problem of subjectivity, scoring took a long time, and the computer did not help the contestants, or the judges.

In the next phase, the problems were already solved on a computer, but scoring was still difficult. At the first informatics olympiads, students wrote programs on a special sheet using programming languages such as Fortran, Cobol, or ASSIRIS, then this programming form was given to computer operators who punched the cards. The program was run two times; after the first run, the errors made by the operators were corrected [1].

Later, test cases were introduced manually, and the programs were run from the respective IDE.[1] This way, it was impossible to precisely measure running time and to test on extremely large test cases.

Today, scoring is done automatically using special software developed for this purpose.

In modern competitive programming, we can see different problem types.

1. *Standard or batch problems*

   The contestants must submit a source code that reads the input and writes out the result. Scoring is done by compiling these source codes into executables and running them against a set of test cases. A test case is considered correctly solved if the contestant's program could be successfully compiled, it ran on the test case by respecting the memory limit, without any runtime errors, and under the time limit, and it produced the correct output for that particular test case.

2. *Interactive problems*

1 Integrated Development Environment, which back in the day was usually some version of Turbo Pascal, or Turbo C, later Borland Pascal and Borland C.

The programs of the contestants must interact in some way with the program of the judges and produce a solution on the fly based on that interaction. Interactive problems appeared at IOI in the early 2000s, and nowadays we can see them in the later stages of national contests and on *Codeforces* [15] and *Google Code Jam* [13] as well.

3. *Output-only problems*

   Only the input files are given, and the contestants must generate and submit the correct output in any way they can (usually by writing a program to assist them). This problem type was popularized by the *Internet Problem Solving Contest* first organized in 1999 [12], then later used at *Google Code Jam* (until 2017) and the *Cloudflight Coding Contest* (formerly known as *Catalysts Coding Contest*) [14].

Each problem type can be scored by different rules based on the contest and the particular problem.

1. *All or nothing*

   Score for a problem is obtained only if it correctly solves all the test cases for that particular problem. This system is the most widely used, and we can observe it at *ICPC* and also in the most important online contests such as the ones organized by *Codeforces* or *Topcoder* [16].

2. *Partial scores*

   The test cases are treated independently, and each test case is worth a certain amount of points. Contestants get points for the test cases they solve correctly. This system is generally used in middle- and high-school-level olympiads. In some cases, multiple inputs are grouped together, and all tests in a group must be solved correctly to obtain the points associated with the group.

3. *Scoring by formula*

   For particularly difficult problems, for which the optimal solution is not known or is very difficult to obtain, the score for each test case may be calculated using a formula specific to the problem. The formula may include the optimal solution and/or the best solution obtained by any contestant during the contest.

1.3 Contest Types

Competitive programming contests can be categorized based on several questions:

- Which problem types can appear?
- How many times can contestants submit solutions to a specific problem? Is there a penalty for multiple submissions?
- Do contestants receive immediate feedback on their submissions? Is this a full feedback or just a partial one (e.g. just for a small subset of all test cases)?
- Is solving speed a factor in the scoring system?
- Is the contest organized online, offline, or both?
- Is the contest individual, for teams, or both? In a team setting, how many computers can a team use?
- What resources can be used during the contest?
- How long does the contest last?

Some contests also include a so-called challenge phase or hacking, when contestants can see the source code submitted by others, and they may try to come up for additional points with a test case which breaks the solution of a competitor. This can be a good practice to improve one's code reading skills, which can be helpful not only when competing individually but also in certain team strategies, as we will discuss later.

Of particular interest to us are the *ICPC* rules, which are used at the *Sapientia ECN* contest. In this case, the answers to the above questions are the following:

- Only standard problems can appear.
- There is no limit on the number of submissions per problem. A team gets a 20-minute penalty for each failed submission for every problem that is later solved by that particular team.
- For every submission, there is full and immediate feedback, which can be one of the following:[2] *Compilation Error*, *Runtime Error*, *Time Limit Exceeded*, *Wrong Answer*, and *Accepted*.
- The primary scoring criterion is the number of solved problems. The secondary criterion is the sum of minutes needed to solve each problem (counted from the start of the contest), with the penalty minutes for wrong submissions added.

2 We mention that mirroring *ICPC* rules, *Presentation Error* is also a possible feedback message, but the judges are trying to avoid it most of the time.

- Most of the time, *ICPC* contests are organized offline, a notable exception being the special conditions caused by the COVID-19 pandemic.
- Generally, teams of three contestants from the same institution are competing, and every team can use a single computer.[3]
- It varies from region to region what resources can be used at *ICPC* contests. In Romania, usually no other resources other than the official documentation of the programming languages used are allowed. In some cases, a limited amount of printed materials is allowed.[4]
- Traditionally, the contest runs for five hours.

1.4 ICPC Tactics and Strategies

In a traditional *ICPC* setting, with teams of three members and a single computer, there is a special dynamic, because the team has five "computer hours" and ten "non-computer hours." This means that time spent at the computer is very sensitive and needs to be optimized, while, on the other hand, the non-computer activities need to be taken as much advantage of as possible.

To avoid the terminal becoming a bottleneck, generally it is good advice to try to use the keyboard as much as possible, even if just for writing basic I/O functions, to use the seat next to the computer only for typing, not thinking, and to try to avoid real-time debugging on the computer as much as possible, by writing program code on paper while preparing a problem and by trying to find bugs on paper. On the other hand, it might be smart not to start new problems in the final phase of the contest and rather try to find errors in existing implementations.

Some authors suggest that "once the algorithm is clear, more than one member working on a single program should be avoided (...) the most efficient way to write a program is to write it alone" [17]. But in our experience, in some cases a *coder-observer* type of pair programming arrangement

3 During the COVID-19 pandemic, every team member could use their own computer.

4 During the COVID-19 pandemic, teams could use any offline or online resource that was available before the start of the contest. Of course, communication with persons outside the team was forbidden.

can work exceptionally well and reduce the number of bugs to almost zero. Beware, the job of the *observer* is the more difficult one in this case!

Other proposed tactics include completely rewriting an easy problem if another team member is struggling with it [18] and tackling hard problems early in the contest so that there is time to solve them [17].

We also note that wrong submissions are usually very expensive and should be avoided if possible. By spending 5 minutes on testing after a problem is done, a team will still be 15 minutes ahead compared to submitting a program with a silly bug, which can then be immediately corrected.

By considering the way penalty minutes are calculated, it is easy to see that the *order* in which a team solves the problems can make a big change in their final ranking. For instance, let us take two problems, where problem A can be solved in 10 minutes and problem B in 20 minutes. If a team solves A first and B second without wrong submissions, its total penalty minutes will be $10 + (10 + 20) = 40$. On the other hand, if another team solves problem B first and problem A second without wrong submissions, its total penalty minutes will be $20 + (20 + 10) = 50$. It is easy to see that the optimal strategy is to solve the problems in their increasing order of difficulty, assuming that a problem's difficulty is directly correlated with the time needed to solve it [19].

This is, of course, a much simplified approach, because it assumes that the total number of problems solved by the end of the contest (first scoring criteria) will be maximal (that is, the team will solve the problems they are able to in the given time), and it does not take into consideration possible wrong submissions and how the team uses the resources of their members to parallelize the work needed to solve the problems. Furthermore, the importance of psychological factors cannot be underestimated, as stress level, motivation, and morale can make a big difference in the final results of a contest [20]. In [21], the importance of avoiding tunnel vision (working on a single problem and continuously submitting solutions for it) is rightfully stressed.

In fact, when two-way or three-way parallelization is introduced, the optimal number of solved problems is not always achieved if the problems are tackled in order of increasing complexity, as proven in [19]. Actually, all the problems are solved in the shortest amount of time when tackling them hardest to easiest. For instance, let us consider two-way parallelization and problem A needing 20 minutes to be solved, problem B 30 minutes, and problem C 40 minutes. If solving easiest to hardest, the first part of the team will first solve problem A, while the other part of the team works

on problem B. When problem A is finished after 20 minutes, problem C is started, leading to a total time of 60 minutes. On the other hand, if solving from hardest to easiest, the team will finish in just 50 minutes, the first part of the team solving problem C while the second part solves problems B and A.

However, solving the problems in decreasing order of difficulty results in the highest penalty time and poses other disadvantages. A team may start to work on a problem which they ultimately are not able to solve, blocking themselves altogether at worst, losing a lot of precious time at best. Also, it may be difficult psychologically to have a smaller number of problems solved while other teams are crushing the easy problems one by one. But often the teams that are in the first positions after the first 2-3 hours finish much lower by the end of the contest. It further complicates things that the difficulty of problems is not always determined correctly. By looking at which problems have been solved by other teams, the difficulties can be reevaluated, but sometimes this can also be misleading.

As we can see, optimal team strategy should be the result of a careful balancing act between the goal of a maximal number of problems solved by the end of the contest with a minimal penalty time, the order the problems are tackled in, and the duties of each team member at any given time. Team strategy can make a big difference in how well a team performs during the contest. It must be shaped after the specific strengths and weaknesses of the team members in order to maximize performance. Some authors suggest that it is beneficial for each member of the team to specialize on certain topics (e.g. graph theory, dynamic programming, math) and work on problems related to their speciality during the contest. But it is also possible to identify other skills, such as problem reading, theoretical algorithm design, coding, bug finding on paper, or crafting test cases, and mold the team strategy based on those [22]. The authors of [21] propose dividing the contest into four phases (opening, early game, midgame, and end game) and adjusting the strategy accordingly.

We always recommend choosing a *team leader* or *team captain* from the team members, who is ultimately responsible for taking decisions during the contest. It is not always the strongest competitor who is also the best team leader. A good coach should be able to observe these subtle differences in the behavior of the team members and suggest a *team leader* accordingly. For instance, the method we frequently use for assigning a team leader is by voting in such a way that each team member gets one vote (they can vote on themselves as well) and the coach gets one vote and a half.

Let us see a few examples of possible team strategies [17, 19, 22, 20, 23]:

- *Simple or individual strategy*
  When every team member is a particularly strong and well-rounded competitor, they can all work on their own. One member solves a problem on the computer, while the other two prepare their problems on paper, in order to minimize their time spent on the computer. Surprisingly, there were cases where this strategy gave extremely good results even at superior stages of the *ICPC*. The advantage of such a strategy is that it maximizes parallelization and reduces problems caused by miscommunication and different programming styles.
- *Rotation strategy*
  This strategy is based on the fact that by assisting the person on the computer, implementation errors can be mitigated, and debugging time can be minimized. Thus, the team members rotate among themselves three roles: the *coder* who works on the current problem at the computer, the *observer* who follows very closely the work of the *coder*, and the *algorist* who thinks about the next problem and prepares the solution on paper. After the current problem is solved, the *algorist* becomes the *coder*, the *coder* becomes the *observer*, and the *observer* becomes the *algorist*.
  Another version of this strategy is to have a *coder*, an *algorist*, and a *debugger*, who tries to find errors on paper. The duties can change during the contest according to the decisions of the *team leader*. However, based on the argument that the best person to code a solution is the person who came up with it, in some teams the *coder* is necessarily the *algorist*, which frees up a team member for other duties.
- *Manager-based method*
  Derived from the previously presented duty-based method, in this case, the team is composed of an *algorist*, a *coder*, and a *manager*, and the goal is to produce error-free code. The *manager* is the one who makes decisions in difficult situations, but is also responsible of debugging and preparing test cases. The *manager* helps the *algorist* as well; this way, each part of the job is done by two people, reducing the probability of bugs. The *coder* and the *algorist* can change duties among themselves when the *coder* becomes exhausted as the contest progresses. A possible disadvantage of this strategy is that it may

require a lot of practice in order to learn the correct communication and collaboration techniques among team members.

- *Terminal man*
  In some teams, one of the members is particularly fast and precise in implementing solutions. It makes sense to keep this person at all times at the computer, while the other two members do their best to discuss solutions among themselves and prepare them for the terminal man to implement as fast as possible.
- *Think tank*
  For teams without fast implementers but with members having a strong theoretical background, it may make sense to sacrifice the first part of the contest to analyze and discuss all the problems, this way trying to maximize the total number of problems solved by the end of the contest.
- *Burn the candle at both ends*
  This is a version of the previous strategy, where the think tank is formed by the two members who are the best problem analyzers, while the third member is working on the easiest problems at the computer. This strategy provides a nice balance between the potential number of problems solved by the end of the contest and total penalty time.
- *Ad hoc strategy*
  Probably most teams apply some form of ad hoc strategy, by always dynamically adapting to the current situation. This may include a mix of tactics such as individual problem solving, preparing problems on paper, pair programming, debugging source code listed on paper, preparing test cases for the problem that is being solved, discussing solutions, and so on.
- *Rule-based strategy*
  A more structured version of the previous strategy is a mixed one, with the team having a set of simple rules with priorities which are followed during the contest. The rules are continuously improved during practice.

For a theoretical study of team strategies, we refer the reader to [19], where 15 different strategies are compared under four possible problem complexity patterns. Under the assumptions made by the authors, they conclude that the more problems are solved concurrently, the quicker the team will finish the problems and the lower the penalty time will be. Although penalty time can be minimized by solving problems in order of ease,

minimizing total solving time is shown to be NP-complete. The authors suggest a hybrid strategy, which starts by the team estimating if it can win by the number of problems solved, in which case it must start to solve problems in decreasing order of difficulty. But if the team estimates that there will be a tie on solved problems, then it tries to minimize penalty time by solving the problems in increasing order of difficulty.

1.5 Preparation for Competitive Programming Contests

To do well in competitive programming, it is almost obligatory to have a strong grasp on algorithms and data structures. Of course, formal education can help with this, but there are also classical books that have been proven to be great resources during the years. The most popular one is probably *Introduction to Algorithms*, frequently abbreviated as *CLRS* by the initials of the authors [24].

Being familiar with the complexity analysis of algorithms and asymptotic notations such as O or Θ is crucial to doing well in programming contests (and arguably in programming in general; no wonder it comes up frequently in technical interviews at serious companies). Having this knowledge, the maximum input size of every problem becomes a hint for the solution. On modern computers, around 10^8 operations can be processed in a few seconds [25], so for most competitive programming problems with time limits expressed in milliseconds, it is safe to assume that we can make around one million to hundred million operations, depending on the type of the operations. Therefore, if the upper limit for n is 10 000, it does not make sense to implement an algorithm with $\Theta(n^3)$ time complexity, as it will surely receive a *Time Limit Exceeded* feedback. Rather, we should probably look for a $\Theta(n^2)$ algorithm, or perhaps a $\Theta(n \log n)$ or a $\Theta(n)$ one. It is important to note that as the size of the input or output approaches 1 Megabyte (around 1 million characters), the input/output routine will take so much time that it will become a bottleneck. In this case, it may make sense to optimize this part of the program as well by using language-specific features (such as `cin.tie` or `sync_with_stdio` in C++) and/or buffering techniques.

Some books directly tackle the field of competitive programming, one of the early ones being [23]. The field became more and more popular, as working in the IT sector became a desirable carrier path for many, and more and more people realized that the skill set which can be acquired

in programming contests is often useful in technical interviews, especially those at large international companies. This increase in popularity led to the creation of more contests and the publication of new books such as [25, 26]. In early 2022, Google announced an AI that will be able to compete in competitive programming contests, which should further increase the appeal and popularity of the field [27].

In the age of multimedia and the Internet, other types of resources could be preferred to books by some or used to supplement them by others, especially after online teaching became more prevalent because of the COVID-19 pandemic. [28] is an excellent collection of competitive-programming-specific algorithms with clean implementations in C++. There are also YouTube channels [29], and also free [30] and paid online courses specifically targeting competitive programming [31, 32, 33].

But theory is useless without practice in this context. Sites called online judges allow users to submit solutions to problems from vast problem sets. There is no limit on the number of submissions, and solutions can be sent in at any time. Some of the most well-known ones are probably [34, 1, 36, 37, 38]. Some of them have educational resources and problem sets, and [40] has a training program with problems of increasing difficulty. [41] is a complementing tool to [34], which helps users keep track of the problems they have solved and picking the next problem to attempt [25].

While trying to solve problems on online judges, it is impossible to avoid getting stuck on a particular problem. Sometimes it is very difficult to figure out what the problem with our solution is, why we are getting *Wrong Answer*, or how we could implement a more efficient solution to avoid *Time Limit Exceeded* or *Memory Limit Exceeded*. A typical mistake that we see time and time again by beginner programmers is making assumptions about the structure of the input. The general rule of thumb is that anything can happen, unless explicitly stated otherwise in the problem statement, so there is absolutely no reason to assume that the given graph is connected, or the given integers are all positive and relatively small [23]. The collection of typical mistakes found in [18] can be helpful, especially for novice programmers.

As we mentioned earlier, having good *testing* skills is very important to avoid wrong submissions (and thus greater penalty times in a contest), but also to do well in challenge and hacking phases and to find mistakes when trying to solve a practice problem. Test cases against which a problem is tested should include the sample input given in the problem statement, small cases crafted on paper, tricky corner cases, extremely small AND large

cases with easily verifiable answers, and also random cases. Extreme care should be taken for problems with multiple test cases in a single input. For these problems, [25] suggests including two identical test cases consecutively to detect initialization errors, but this may not be enough, and you should try some cases in increasing and decreasing order of their sizes as well.

Some online judge sites also have dedicated discussion forums that try to offer a solution to these problems. Another great resource is *uDebug* [39], which offers hints for an ever-growing number of problems from the most popular online judges, giving the possibility to check what output a correct solution gives for a specific input. The community can also add critical test cases for every problem, this way helping those who try to solve the problem later.

Some of the online judges also allow the creation of virtual contests, so the users can try themselves in conditions more similar to a real contest, with a predefined problem set and in a limited timeframe. For organizing virtual contests, the easiest and most popular solution in the past years is probably *vjudge*, a special online judge which aggregates the problems of all the other important online judges [42].

A very useful feature of some of these sites is the ability to read the source codes of others, which can help tremendously in improving one's coding ability. It is usually a good idea to solve a problem multiple times, until the coder is satisfied with the time it took to implement and the cleanness of the code.

But arguably nothing will prepare one better for competing than... competing. *Codeforces* and *Topcoder* organize algorithmic contests regularly, and every user has a rating similar to the Elo rating used in chess, which is recalculated after each contest based on how the user did in that particular contest compared to others.

Other sites organize contests where team participation is also allowed, which is valuable for teams preparing for the *ICPC*. Trying out and fine-tuning team strategies during these preparation contests is crucial.

As most of these contests are time-sensitive, general best practices usually observed in software development (such as avoiding global variables or designing a robust object-oriented class structure) are NOT followed, to decrease the time needed for implementation. Still, it is a good idea to make the programs clean and easily understandable, especially in a team setting, where agreeing upon some style and naming conventions is advised.

In a contest scenario, one probably tries extra hard to solve a problem, giving their best, compared to a more relaxed practice scenario. Thinking

so much and so intensely about the solution of a problem and still not being able to solve it can be, of course, frustrating, but also an excellent learning experience. Due to this, it is highly recommended to solve the remaining unsolved problems after the contest has been concluded. This can be done based on the published editorials explaining the solution of every problem, the source code of other contestants, discussion with others, and further thinking without the stress and time constraints of a contest. In competitive programming, this strategy is called *upsolving*.

BIBLIOGRAPHY

[1] E. Cerchez and M. I. Andreica. 2008. Romanian national olympiads in informatics and training. *Olympiads in Informatics: An International Journal 2*, 37–47.

[2] International Olympiad in Informatics – Statistics
`http://stats.ioinformatics.org/countries`

[3] *Gazeta de Informatică – Istoria GInfo*

[4] Everyone Needs Problems! (Lista lu' Frâncu)

[5] Romanian Master of Informatics
`https://rmi.lbi.ro/rmi_2021`

[6] International Informatics Olympiad in Teams
`http://cni.nt.edu.ro/ioit/?page_id=565`

[7] InfO(1) cup
`https://www.info1cup.com`

[8] .campion

[9] pbinfo
`https://www.pbinfo.ro`

[10] ICPC SouthEastern European Regional Contest

[11] ICPC World Champion Hall of Fame
`https://icpc.global/community/world-finals-champions`

[12] Internet Problem Solving Contest
`https://ipsc.ksp.sk`

[13] Google Code Jam

[14] Cloudlfight Coding Contest
`https://register.codingcontest.org`

[15] Codeforces
https://codeforces.com

[16] Topcoder Competitive Programming
https://www.topcoder.com/thrive/tracks?track=Competitive%20Programming&ref=nav

[17] F. Ernst, J. Moelands, and S. Pieterse. 1996. Teamwork in programming contests: 3* 1 = 4. *ACM Crossroads Student Magazine* 3, 2.

[18] S. Manzoor. 2008. Common mistakes in online and real-time contests. *ACM Crossroads Student Magazine* 14, 4, 10–16.

[19] A. Trotman and C. Handley. 2008. Programming contest strategy. *Computers & Education* 50, 3, 821–837.

[20] S. A. Amraii. 2007. Observations on teamwork strategies in the ACM international collegiate programming contest. *ACM Crossroads Student Magazine* 14, 1, 1–9.

[21] A. Bloomfield and B. Sotomayor. 2016. A programming contest strategy guide. In *Proceedings of the* 47^{th} *ACM Technical Symposium on Computing Science Education*, 609–614.

[22] R. Liu. 2008. *Training ICPC Teams: A Technical Guide. Competitive Learning Institute Symposium, Banff.*

[23] S. S. Skiena and M. A. Revilla. 2003. *Programming Challenges*: *The Programming Contest Training Manual*. Springer.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms*. MIT Press.

[25] S. Halim, F. Halim, S. S. Skiena, and M. A. Revilla. 2013. *Competitive Programming 3*. Lulu Independent Publish.

[26] A. Laaksonen. 2020. *Guide to Competitive Programming*. Springer.

[27] Competitive Programming with AlphaCode
https://deepmind.com/blog/article/Competitive-programming-with-AlphaCode

[28] Algorithms for Competitive Programming
https://cp-algorithms.com

[29] Errichto Youtube Channel
https://www.youtube.com/channel/UCBr_Fu6q9iHYQCh13jmpbrg

[30] Codeforces Courses
https://codeforces.com/edu/courses

[31] The Bible of Competitive Programming & Coding Interviews
https://www.udemy.com/course/the-bible-of-algorithms-and-interview-questions

[32] Graph Theory Algorithms for Competitive Programming
https://www.udemy.com/course/graph-theory-algorithms-for-competitive-programming

[33] Competitive Programming
https://www.udemy.com/course/competitive-programming

[34] Online Judge (formerly Universidad de Valladolid online judge, UVA)
https://onlinejudge.org

[35] Peking University JudgeOnline (PKU)
http://poj.org

[36] AtCoder
https://atcoder.jp

[37] Kattis Problem Archive
https://open.kattis.com

[38] Timus Online Judge
https://acm.timus.ru

[39] uDebug
https://www.udebug.com

[40] USACO Training Gateway
https://train.usaco.org/

[41] uHunt
https://uhunt.onlinejudge.org

[42] Virtual Judge
https://vjudge.net

CHAPTER 2

DYNAMIC PROGRAMMING

Author: Zoltán Kátai

Abstract. This chapter presents two different approaches (recipes) to address dynamic programming problems. The suggested methods are illustrated on two "ECN-problems."

2.1 Introduction

Dynamic programming (DP) as optimization method was proposed by Richard Bellman. DP can be used to solve optimization problems that satisfy the principle of optimality: The optimal solution of the original problem is built on optimal sub-solutions with respect to the corresponding subproblems. The principle of the optimality implicitly suggests that the problem can be decomposed into (or reduced to) similar subproblems. Usually, this operation can be performed in several ways. The goal is to build up the optimal solution of the original problem from the optimal solutions of its smaller subproblems. Consequently, DP is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

The bottom-up solution-building process starts from the level of trivial subproblems, and the optimal solutions of the larger subproblems are computed from the optimal solution of the already solved (and stored; usually, in the cells of an array) smaller subproblems. At the core of the DP problem-solving process, there is a recursive formula that describes how to calculate the optimal value of the current subproblem from the optimal values of which direct sub-subproblems. The optimal decision sequence can be extracted, in a top-down way, from the array that stores the optimum values.

2.2 DP Diversity

There are several ways to classify DP problems. One of them is based on the "nature" of the bottom-up building process (or on the type of "subproblem-dependencies") [1, 2]:

- value – to – value
- subsequence – to – subsequence
- subsequence pair – to – subsequence pair
- subsegment – to – subsegment
- submatrix – to – submatrix
- subset – to – subset
- subtree – to – subtree

2.2.1 "Value – to – Value" DP Problems

Sample problem

Coin-change-making problem: Given an unlimited supply of coins of given denominations, find the minimum number of coins required to get the desired change.

Example

If we have coins of $n = 3$ different denominations ($w[1 \dots 3] = \{2, 3, 5\}$) and the desired change is 11 (amount S), the minimum number of coins required is 3 ($3 + 3 + 5$).

Building the solution

Although we are interested only in the optimal solution for amount S, we solve the problem for all amounts i from 0 to S. The minimum coin number for amount i we store in the cell $c[i]$ of array $c[0 \dots S]$ (see *Figure 2.1*).

Recursive formula (see *Figure 2.2*)

$$c[i] = \begin{cases} 0 & i = 0 \\ \min\limits_{j=1\dots n} \{c[i - w[j]] + 1 \mid 0 \le i - w[j]\} & 1 \le i \le S \end{cases}$$

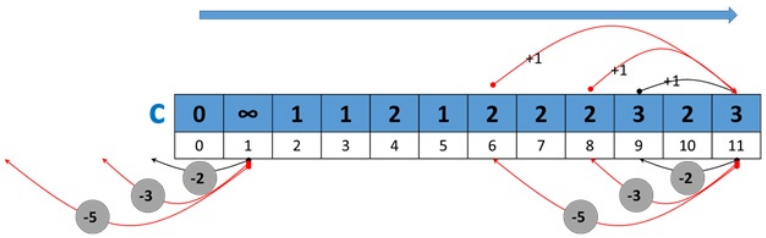


Figure 2.1. Computing values $c[i]$ for i from 0 to $S = 11$ (based on the recursive formula). Arrows mark the subproblems (represented by cells $c[i - w[j]]$, $j = 1 \ldots n$, $n = 3$) of a current problem (represented by cell $c[i]$). Value ∞ indicates that there is no solution for the corresponding subproblem.

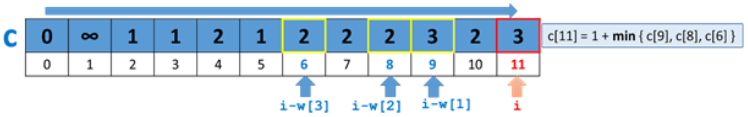


Figure 2.2. The current optimum (red cell) is computed as the minimum of the optimums of its direct subproblems (yellow cells).

2.2.2 "Subsequence – to – Subsequence" DP Problems

Sample problem

Longest increasing subsequence: Find the longest subsequence of a given sequence in which the elements are in increasing order (this subsequence is not necessarily contiguous, or unique).

Example

If the sequence is $a[1 \ldots 11] = \{3, 2, 1, 3, 5, 2, 3, 1, 7, 9, 7\}$, then the length of the longest increasing subsequence is 5 (for example, subsequence $\{2, 3, 5, 7, 9\}$).

Building the solution

For didactical reasons, we introduce $a[0] = -\infty$. Although we are interested only in the longest increasing subsequence starting with element $a[0]$, we determine the longest increasing subsequences starting with elements $a[n]$, $a[n-1]$, ..., $a[1]$, $a[0]$ respectively. We store the length of the longest increasing subsequence starting with element $a[i]$ in the cell $c[i]$ of array $c[0 \ldots n]$ (see *Figure 2.3*).

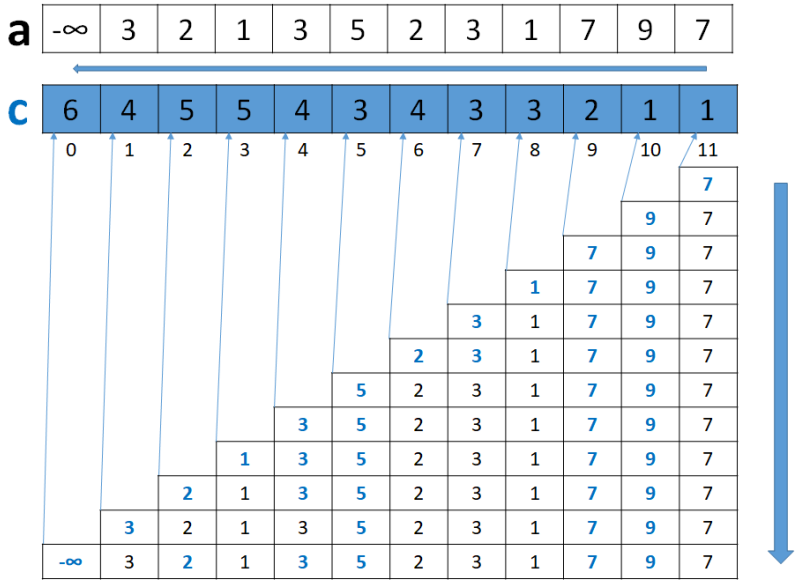


Figure 2.3. Computing values $c[i]$ for i from $n = 11$ to 0 (based on the recursive formula). For each subproblem, the corresponding longest increasing subsequence is indicated (blue values).

Recursive formula (see *Figure 2.4*)

$$c[i] = \begin{cases} 1 & i = n \\ \max\limits_{j=i+1\ldots n} \{c[j] + 1 \mid a[i] < a[j]\} & 0 \le i \le n-1 \end{cases}$$

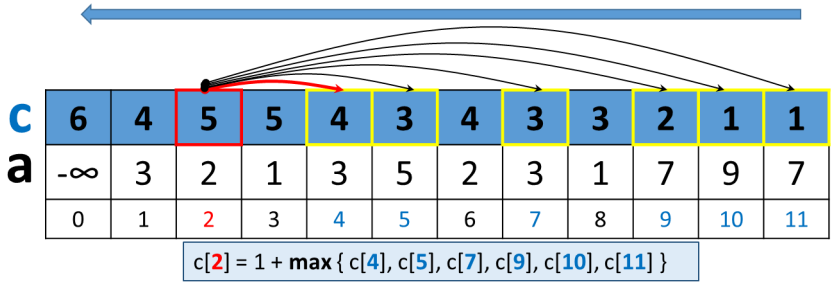


Figure 2.4. The current optimum (red cell) is computed as the minimum of the optimums of its direct subproblems (yellow cells).

2.2.3 "Subsequence Pair – to – Subsequence Pair" DP Problems

Sample problem

Longest common subsequence: Given two sequences, we are interested in the longest subsequence common to both sequences (subsequences are not required to occupy consecutive positions within the original sequences).

Example

If the sequences are $a[1 \ldots 4] = \{2, 9, 5, 7\}$ and $b[1 \ldots 3] = \{9, 3, 7\}$, then the length of the longest common subsequence is 2 and the subsequence is $\{9, 7\}$.

Building the solution

We solve the problem for all the subsequence pairs of the original sequence pair in a bottom-up manner (*Figure 2.5*).

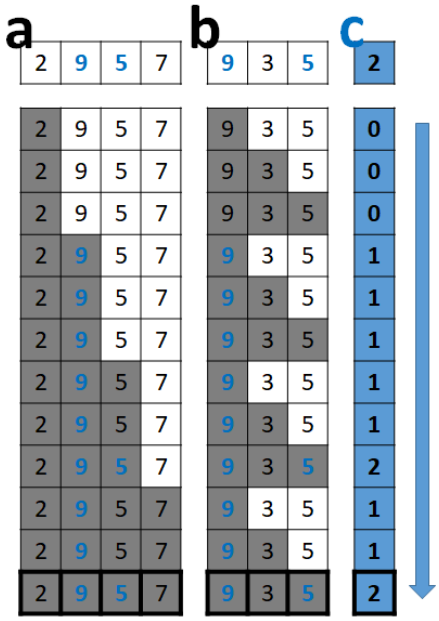


Figure 2.5. Solving the problem for all subsequence pairs. The blue column (array c represented as a one-dimensional array) indicates the length of the corresponding longest common subsequence (blue elements).

We store the length of the longest common subsequence of subsequences $a[1 \dots i]$ and $b[1 \dots j]$ in the cell $c[i][j]$ of array $c[0 \dots n][0 \dots m]$ (*Figure 2.6*).

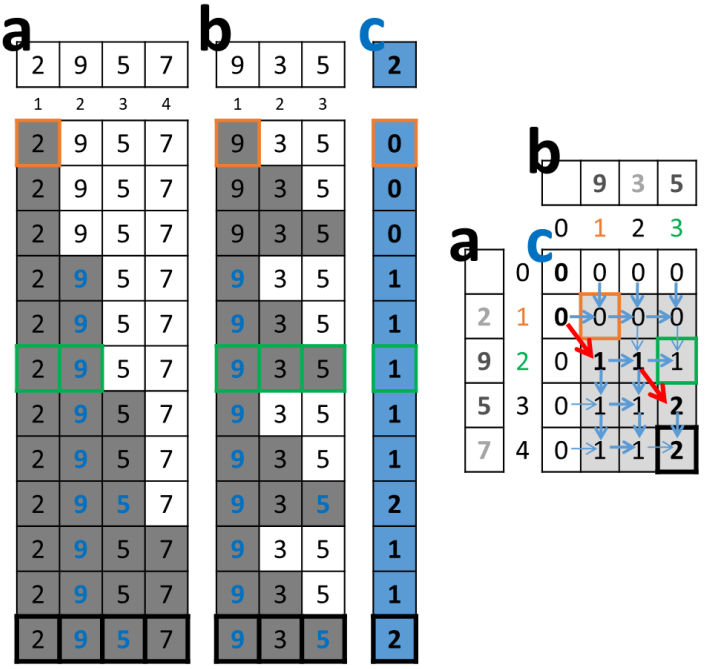


Figure 2.6. Since subproblems have two parameters, i and j, we need a bi-dimensional array for storing the corresponding optimums (the length of the longest common subsequence). Cell $c[i][j]$ stores the optimum for subsequence pair $(a[1 \dots i],b[1 \dots j])$. Row 0 and column 0 represent the trivial subproblems (one of the subsequences is empty; or both of them are empty). Cell $c[n][m]$ ($n = 4$, $m = 3$) represents the original problem. In accordance with the recursive formula, the starting points of the "in-arrows" of each cell indicate the cells that represent the corresponding direct subproblems. The bold arrows mark the optimal decisions. Bold red arrows correspond to the optimal solution (longest common subsequence).

We consider as trivial subproblems those ones where one of the subsequences is empty. These subproblems are represented by the cells from row 0 and column 0 of array c.

Recursive formula (see *Figure 2.6*)

$$c[i][j] = \begin{cases} 0 & i = 0, 0 \leq j \leq m \\ 0 & j = 0, 0 \leq i \leq n \\ 1 + c[i-1][j-1] & 1 \leq i \leq n, 1 \leq j \leq m, a[i] = b[j] \\ \max\{c[i][j-1], c[i-1][j]\} & 1 \leq i \leq n, 1 \leq j \leq m, a[i] \neq b[j] \end{cases}$$

2.2.4 "Subsegment – to – Subsegment" DP Problems

Sample problem

Longest palindrome subsequence: Determine the longest palindrome subsequence of a given sequence (this subsequence is not necessarily contiguous, or unique).

Example

If the sequence is $a[1 \ldots 5] = \{2, 1, 4, 2, 2\}$, then the length of the longest palindrome subsequence is 3, and there are five such subsequences ({2,1,2}; {2,1,2}; {2,4,2}; {2,4,2}; {2,2,2}).

Building the solution

Although we are interested only in the longest palindrome subsequence of the given sequence, we solve the problem for all the 1-length, 2-length, ... subsegments of the original sequence (see *Figure 2.7*). We store the length of the longest palindrome subsequence of segment $a[i \ldots j]$ in cell $c[i][j]$ of array $c[1 \ldots n][1 \ldots n]$ (see *Figure 2.7*).

Recursive formula (see *Figure 2.8*)

$$c[i][j] = \begin{cases} 0 & 2 \leq i \leq n, j = i - 1 \\ 1 & 1 \leq i \leq n, j = i \\ 2 + c[i+1][j-1] & 1 \leq i \leq n-1, i+1 \leq j \leq n, a[i] = a[j] \\ \max\{c[i][j-1], c[i+1][j]\} & 1 \leq i \leq n-1, i+1 \leq j \leq n, a[i] \neq a[j] \end{cases}$$

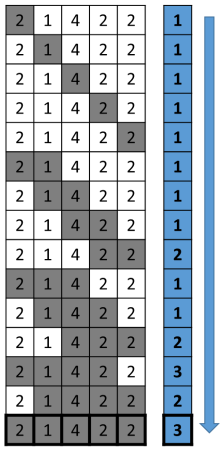


Figure 2.7. The blue cells store the length of the longest palindrome subsequence of the corresponding gray subsegments.

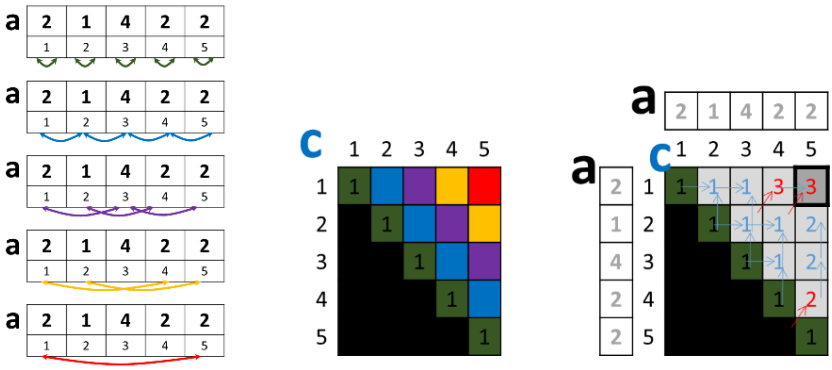


Figure 2.8. Subsegments of a given color (left figure) are represented by cells from the diagonal with the same color (middle figure). Cell $c[i][j]$ $(i \leq j)$ stores the optimum for segment $a[i \ldots j]$. The cells of the main diagonal represent the trivial subproblems $(j - i = 0)$. Cell $c[n][n]$ $(n = 5)$ represents the original problem. Cells for indices $i < j$ (right figure) can be computed, for example, diagonal by diagonal (based on the recursive formula). Arrows represent optimal choices (optimal solution can be extracted from array c alongside these arrows in a top-down way).

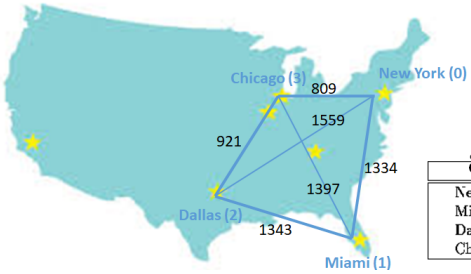
2.2.5 "Subset – to – Subset" DP Problems

Sample problem

Traveling salesman problem (TSP): Given n cities (identified by numbers 0, 1, 2, ..., n-1) and the distances between each pair of cities (stored in array $a[0 \ldots n-1][0 \ldots n-1]$), what is the shortest possible route that visits each city exactly once and returns to the origin city (optimal Hamiltonian circuit)?

Example

For the four cities from *Figure 2.9*, the optimal route is {0,1,2,3,0} (we considered city 0 as starting point).



a

	New York	Miami	Dallas	Chicago
New York	—	1334	1559	809
Miami	1334	—	1343	1397
Dallas	1559	1343	—	921
Chicago	809	1397	921	—

Figure 2.9. An example for four cities and the corresponding adjacency matrix

Building the solution

We have to determine the optimal Hamiltonian path (visit each city exactly once) from city 0 to each city i ($i \neq 0$). Let us denote the length of these optimal paths $d[i]$, where $i = 1 \ldots n-1$. The length of the optimal Hamiltonian circuit is equal to $min\{d[i] + a[i][0], i = 1 \ldots n-1\}$. Although we are interested only in the optimal Hamiltonian paths for the original set of cities ($S = \{0, 1, \ldots, n-1\}$), we solve the problem for all the subsets of S that include city 0 (valid subsets; see *Figure 2.10*). For each subset s, we compute the length of the shortest Hamiltonian path from node 0 to each element i ($i \neq 0$) of the subset. Accordingly, subproblems are characterized by pairs (s, i). Considering that there is a one-to-one correspondence between the subsets of S and the natural numbers from 0 to 2^n-1, we store the optimum values in the corresponding cells of array $c[0 \ldots 2^n-1][0 \ldots n-1]$ (see *Figure 2.11*). Cell $c[s][i]$ ($i \in s$, $i \neq 0$) stores the length of the shortest

Hamiltonian path (passes all elements of s exactly once) from city 0 to city i. (Notice that $c[S][i] = d[i]$ for all $i = 1 \dots n-1$)

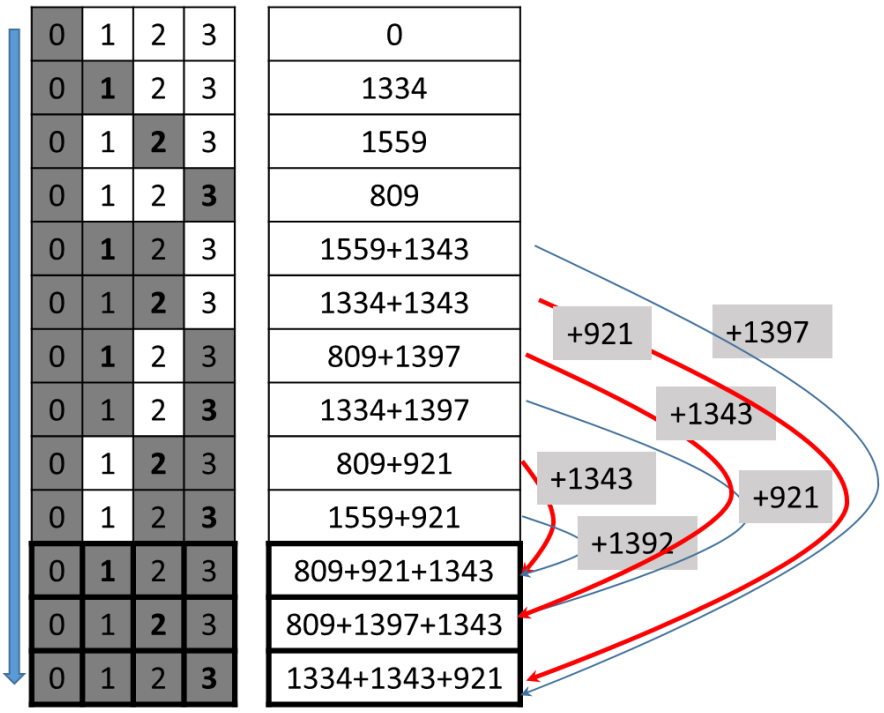


Figure 2.10. Each row corresponds to a subproblem (s, i) (subset s is indicated by gray cells; element i is bold). The right column contains the corresponding $c(s, i)$ values. The arrows indicate the possible choices for subproblems (S, i) (rows with bold cells). Red arrows mark optimal choices.

Recursive formula (s represents both a subset and the corresponding row index in array c; see *Figure 2.10*)

$$c[s][i] = \begin{cases} 0 & s = 1, i = 0 \\ a[0][i] & s = \{0, i\}, 1 \le i \le n-1 \\ \max\limits_{j \in s \setminus i, j \neq 0} \{c[s \setminus i][j] + a[j][i]\} & s \subset S, i \in s, i \neq 0 \end{cases}$$

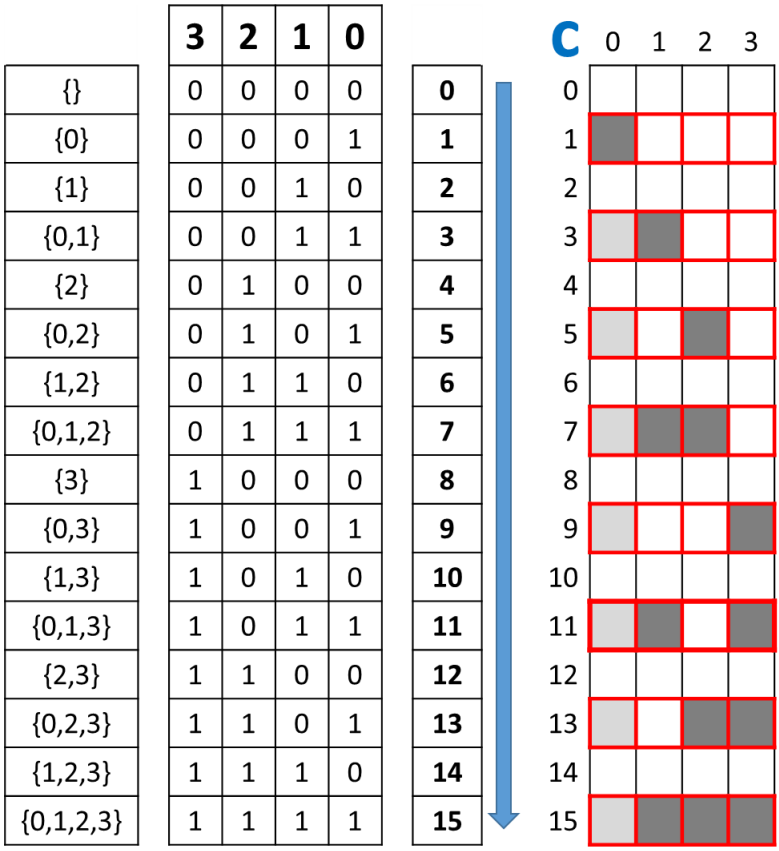


Figure 2.11. One-to-one correspondence between the subsets of set {0,1,2,3} and the natural numbers from 0 to 2^4-1 (based on the binary representation of the numbers). Subproblems to be solved are represented by the dark gray cells of array c. The light gray cells correspond to the starting city.

2.2.6 "Subtree – to – Subtree" DP Problems

Sample problem

Isolating the infected nodes: Given a bicolored (blue: not infected; red: infected) tree with n nodes (identified by numbers 0, 1, 2, ..., $n-1$) and the cost of each connection (the cost to eliminate the corresponding connection),

isolate (from each other) the infected nodes with minimum elimination cost. (Each component should contain at most one infected node.)

Example

For the tree from *Figure 2.12*, the minimum elimination cost is $4 = 1+3$.

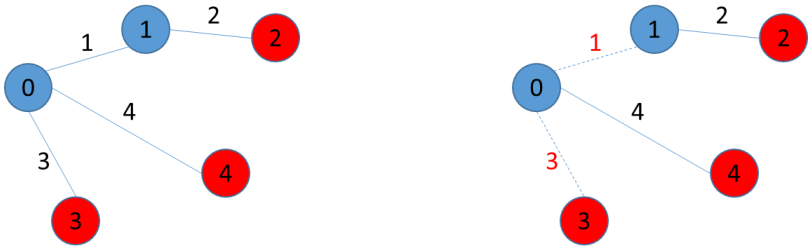


Figure 2.12. Sample tree with 5 nodes. Eliminating connections (0,1) and (0,3) the resulting subtrees contain at most one infected node, and the corresponding cost is $4 = 1 + 3$.

Building the solution

We solve the problem for all the subtrees of the original tree. The optimum for the current tree can be computed from the optima of its subtrees using a postorder DFS. For each subtree, we compute and store two optimums corresponding to the following two cases: (1) optimal partition when the root-node belongs to an infected component; (2) optimal partition when the root-node belongs to a safe component (red and blue values attached to each node in *Figure 2.13*). Value ∞ indicates that the corresponding optimum does not exist (for example, if the root is infected, then case 1 partition does not exist).

Recursive formula

Establishing the recursive formula is left to the reader.

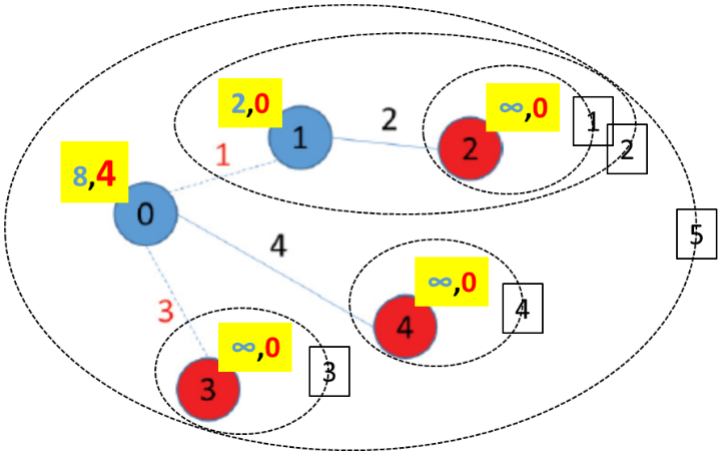


Figure 2.13. Dotted ellipses mark the subtrees. Black rectangles indicate the postorder momentums of the DFS algorithm. To the root of each subtree, two optimums (blue and red values) are associated corresponding to the two cases. The optimum we are looking for is the smallest value attached to node 0.

2.3 Recipe to Address DP Problems

In the following, we suggest a five-step recipe to address DP problems [2]:

1. We identify the general, parametric form of the subproblems resulting from the above detailed breaking-down process. For what parameter values do we get the original problem and the trivial ones?
2. We choose the array for storing the optimum values for each subproblem. The number of independent parameters are hand in hand with the dimension of the array. What cells represent the original problem and the trivial ones?
3. We establish the recursive formula that mathematically describes the bottom-up building process.
4. We implement the algorithm that fills the array for optimum values based on the recursive formula. (A bridge is built through the array from its "trivial bank" to the goal cell from the opposite bank.)
5. We extract the optimal solution (decision sequence) from the array in the opposite direction (we identify the optimal path/tree through the bridge.)

The key step of the above recipe is step 3. Another classification of DP problems (based on the nature of the recursive formula) can help us perform this step. Since subproblems are solved level by level, we distinguish between serial or non-serial formulas. In the case of serial problems, the current optimum depends only on immediate-previous-level optimums. When the optimum of the current subproblem may depend on the optimum of any previously solved subproblem, then we are facing a non-serial formula. From another perspective, the formula could be monadic or polyadic. If a single previous level optimum is built into the current optimum, then we are talking about a monadic problem, otherwise it is a polyadic one. The resulted DP problem categories are:

- serial – monadic (The triangle)
- non-serial – monadic (Longest common subsequence)
- serial – polyadic (Floyd's algorithm)
- non-serial – polyadic (Minimum number of palindromes)

In the brackets, we indicated a well-known DP problem for each category.

We illustrate the use of the above suggested recipe on two DP problems proposed at the 2011, 2016, and 2017 editions of our beloved "ECN contest."

2.4 The Pyramid (ECN-2017)

The key parameters of the "gradual square pyramid" from the figures (see *Figure 2.14*) are stair depth (x) and stair height (y).

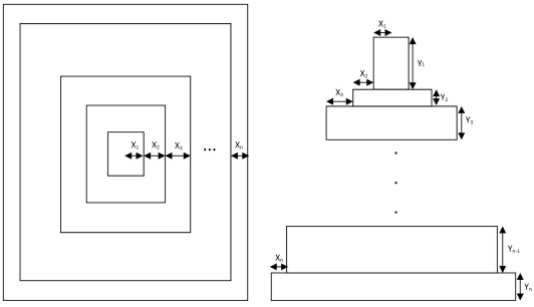


Figure 2.14. Top and side view of the pyramid

Given the $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ pair sequence ($n \leq 20, x_i < 100$, $y_i < 100, i = 1 \ldots n$), find out the optimal succession that minimizes the

volume of the pyramid. (Since different stair orders result in different pyramid volumes, we are interested in the optimal stair succession.)

Input: The first line of the input file contains the number of the given test cases: the value of n followed by pairs of integers representing the stair height and depth separated with white characters: $x_1 y_1 x_2 y_2 \ldots x_n y_n$.

Output: A line for each test case written to the standard output: the indexes of the height–depth pairs in the correct order separated with white characters.

Sample input

```
1

6 10 3 11 2 5 1 11 3 5 2 10 1
```

Sample output

```
5 1 4 3 2 6
```

2.4.1 Rationale and Solution

For a clearer explanation, we use colors to distinguish the stairs. *Figure 2.15* shows the colored pyramid for another sample input.

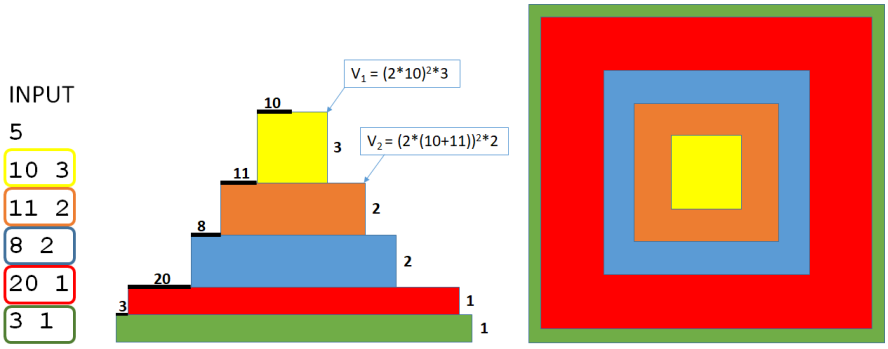


Figure 2.15. Sample input and the corresponding pyramid from two different perspectives. The formulas indicate the volumes for the first two stairs.

As it can be discerned from *Figure 2.16*, different stair orders result in different pyramid volumes.

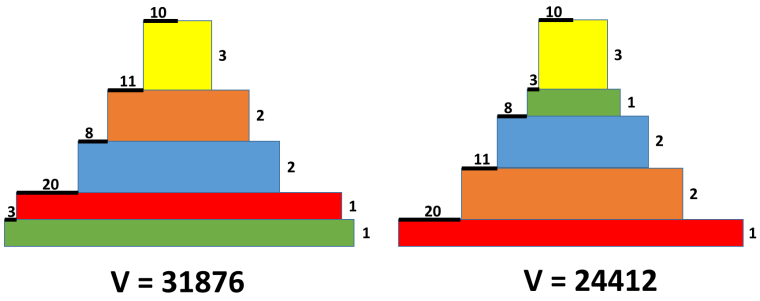


Figure 2.16. Different stair orders result in pyramids with different volumes.

This problem can be considered a "subset – to – subset DP problem." We will solve the problem for all "stair subsets" in a bottom-up way. Since the optimal solution satisfies the principle of optimality, with respect to all subsets, we are interested only in the optimal stair order. *Figure 2.17* shows the optimal pyramid with the maximum volume.

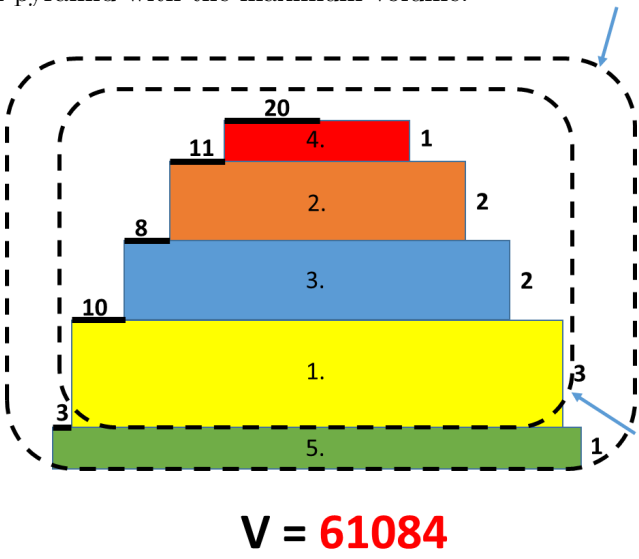


Figure 2.17. The optimal pyramid for the sample input from *Figure 2.15*

According to the principle of optimality, if for stair set $\{1, 2, 3, 4, 5\}$ the optimal order is $4-2-3-1-5$, then for subset $\{1, 2, 3, 4\}$ order $4-2-3-1$ is guaranteed as the optimal solution, and so on.

We illustrate the bottom-up solution building process on a four-level pyramid. *Figure 2.18* presents a sample input for $n = 4$ and the corresponding optimal pyramid. *Figure 2.19* shows the optimal pyramids for all subsets of set $\{1, 2, 3, 4\}$.

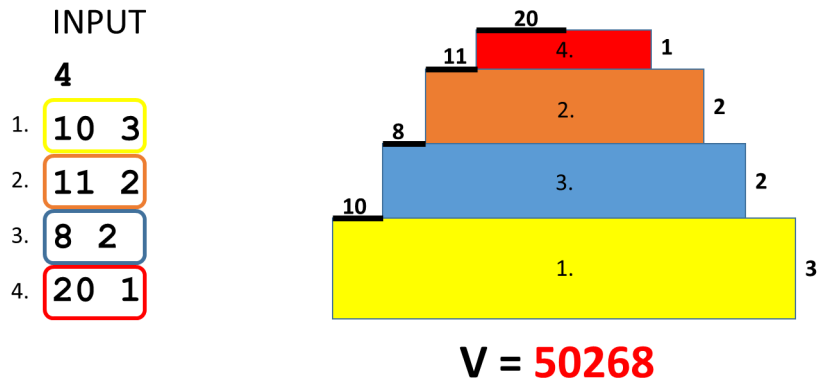


Figure 2.18. An optimal four-level pyramid and the corresponding sample input

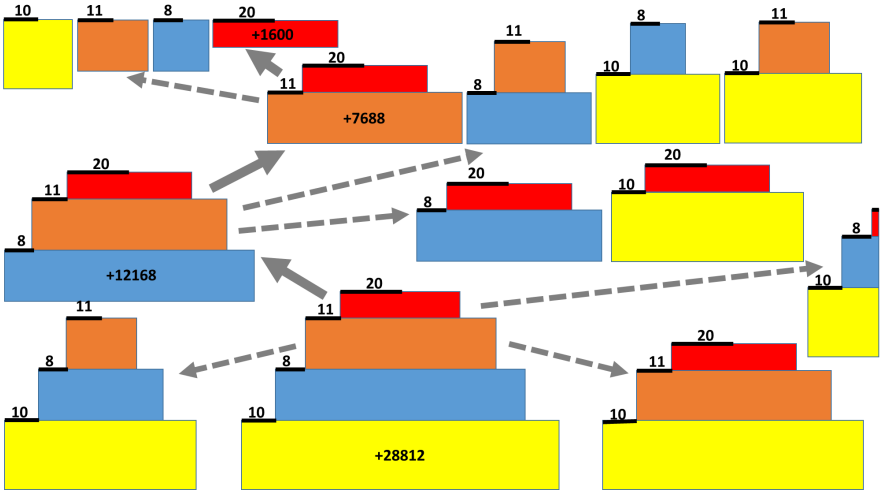


Figure 2.19. Solving the problem for all subsets in a bottom-up order. At each level, the arrows indicate the direct sub-pyramids of the optimal pyramid. The bold arrows mark the optimal sub-pyramids.

Starting with the empty set (trivial subproblem; level 0), we determine the optimal stair order (and the corresponding optimal volume) for each one-element subset (level 1), for each two-element subset (level 2), each three-element subset (level 3), and finally for the original stair set (level 4). From this point on, we continue with implementing the suggested five-step recipe.

1. Since the subsets of set $\{1, 2, \ldots, n\}$ are in one-to-one correspondence with the binary representations of numbers from 0 to $2^n - 1$ (see *Figure 2.20*), the general subproblem is to compute the optimal volume corresponding to number i ($i = 0 \ldots 2^n - 1$). The parameters of the trivial subproblem and the original one are 0 and $2^n - 1$ respectively.

2. For storing the optimal volume values, we use array $c[0 \ldots 2^n - 1]$. Cell $c[i]$ stores the optimum for subset S_i (the subset that includes those elements that correspond to digits 1 in the n-bit binary representation of i) (see *Figure 2.20*).

3. Naturally, c[i] (current parent cell) can be computed from the cells corresponding to those subsets of S_i which have $|S_i| - 1$ cardinality. The recursive formula is the following (X_i is the side length of the base of the pyramid corresponding to subset S_i):

$$c[i] = \max_{j \in S_i} \{c[S_i \setminus j] + X_i^2 \cdot y_j\}$$

$$X_i = 2 \cdot \sum_{j \in S_i} x_j$$

4. We implement the algorithm that fills array c cell by cell in a bottom-up order based on the above recursive formula (see the code below and the attached comments).

5. The parameters of the optimal pyramid (the optimal stair order) can be extracted from array c in a top-down manner (see the bold arrow sequence in *Figure 2.19*).

Note: Since the current optimum incorporates a single previous level optimum, this problem belongs to the serial monadic DP category.

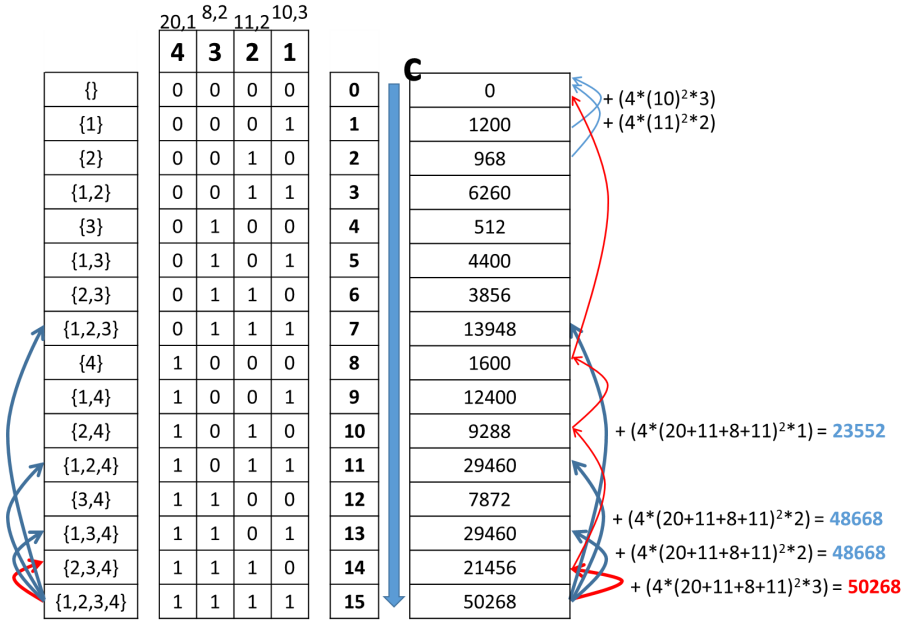


Figure 2.20. One-to–one correspondence between the subsets of set $\{1, 2, 3, 4\}$ and the natural numbers from 0 to $2^4 - 1$ (based on the binary representation of the numbers). The arrows indicate the possible choices. Red arrows mark the optimal solution.

In the following, we present the author's code. This version computes only the optimal volume. Extracting from array c the optimal stair order is left to the reader.

The C code

```
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    int x, y;
}ITEM;

int main()
{
    freopen("pyramid.txt", "r", stdin);
```

```
    int n; scanf("%i", &n);
    ITEM a[n];
    for( int i = 0 ; i < n ; ++i ){
        scanf("%i%i", &a[i].x, &a[i].y);
    }
    int N = 1 << n;
    int c[N];
    c[0] = 0|
    for( int i = 1 ; i < N ; ++i ){
//for all nonempty subsets in increasing order
       int Xi = 0;
       for( int j = 0 ; j < n ; ++j ){
          if( i & ( 1<<j ) ){
//for all elements (j) of the subset corresponding to i
               Xi += a[j].x;|
          }
       }
       Xi *= 2; //side length of the base of pyramid Si
       c[i] = 0;
       for( int j = 0 ; j < n ; ++j ){
           if( i & ( 1<<j ) ){
                int ii = i; ii &= ~( 1<<j );
                //ii represents subset Si\j
                if( c[ii] + Xi*Xi*a[j].y > c[i] ){
                    c[i] = c[ii] + Xi*Xi*a[j].y;
                }
           }
       }

    }
    printf("%i", c[N-1]); // the optimal volume
    return 0;
}
```

2.5 Fixing Bugs (ECN-2016)

Let us imagine you are the system administrator of a critical IT infrastructure composed of multiple servers running different software. Suddenly you notice that multiple components of your system are affected by different software bugs, n ($n \leq 20$) bugs in total. In this situation, the whole system becomes vulnerable to attacks, and this condition persists while you fix all the bugs.

You receive the exact instructions from the support team about the way to fix the bugs, and this way you are able to estimate the required solving time ($\sigma \leq 10^3$) separately for all the bugs. In addition, all the bugs are ranked by the security team, therefore each of them has an initial risk score assigned (r): 1 – low, 2 – medium, 3 – high. Unfortunately, time is your enemy: the risk scores are increasing direct-proportionally with the time spent as long as they are exploitable. The time-dependent risk (η) for the i^{th} bug can be calculated as follows:

$$\eta_i(t) = \begin{cases} 0, & t < 0 \quad \text{or} \quad t > \tau_i \\ r_i \cdot t, & \text{otherwise} \end{cases}$$

where τ_i represents the end time of the treating for the i^{th} bug.

Your job is to schedule the solving of the bugs in the optimal order, more precisely to find that order which minimizes the integral of the cumulative risk (q) for the whole intervention period, where

$$q(t) = \sum_{i=1}^{n} \eta_i(t)$$

$$\int_0^t q(x)\,dx = \sum_{i=1}^{n} r_i \cdot \frac{\left(\sum_{k=i}^{n} \sigma_k\right)^2}{2}$$

Input: The first line of the input file contains the number of the given test cases, after which each line contains a test case: pairs of integers representing the initial risk scores and the solving times required for the bugs $(r_1, \sigma_1, r_2, \sigma_2, \dots)$ separated with space characters.

Output: A line for each test case written to the standard output: the indexes of the risk–time pairs in the correct order separated with space characters.

Sample input

```
1
3 10 2 11 1 5 3 11 2 5 1 10
```

Sample output

```
5 1 4 3 2 6
```

Details:

$$\int_0^t q(x)\,dx = 2 \cdot \frac{5^2}{2} + 3 \cdot \frac{(5+10)^2}{2} + 3 \cdot \frac{(5+10+11)^2}{2} + 1 \cdot \frac{(5+10+11+5)^2}{2} + 2 \cdot \frac{(5+10+11+5+11)^2}{2} + 1 \cdot \frac{(5+10+11+5+11+10)^2}{2} = 4973$$

Note: If we compare the formula from this problem with the formula for the volume of the pyramid from the previous problem, we can easily realize that the two problems are essentially identical. Consequently, the solution of this problem assumes the same DP aapproach as the previous one.

2.6 The Mosaic (ECN-2011)

Let us consider a bi-dimensional array $a[1 \ldots n][1 \ldots m]$ as being a mosaic ($1 \leq n, m \leq 32$). The mosaic consists of small squares; each square is painted in a single color. Cell $a[i][j]$ is a positive integer between 0 and 255, and it stores the color of the square in the position (i, j).

Repeatedly, the mosaic is hit by small balls. Each ball hits the mosaic at exactly the corners of certain squares. As a result of the impact, the mosaic breaks into pieces along the vertical and horizontal intersection lines of the impact point (see *Figure 2.21*).

Input: The input file contains several input datasets having the following structure: the first line stores the number of datasets. This line is followed by the list of sets each corresponding to a mosaic. The first line of each dataset contains the value of n and m separated by a single space. Each of the next n lines defines a row of array a. The row's elements are separated by a single space.

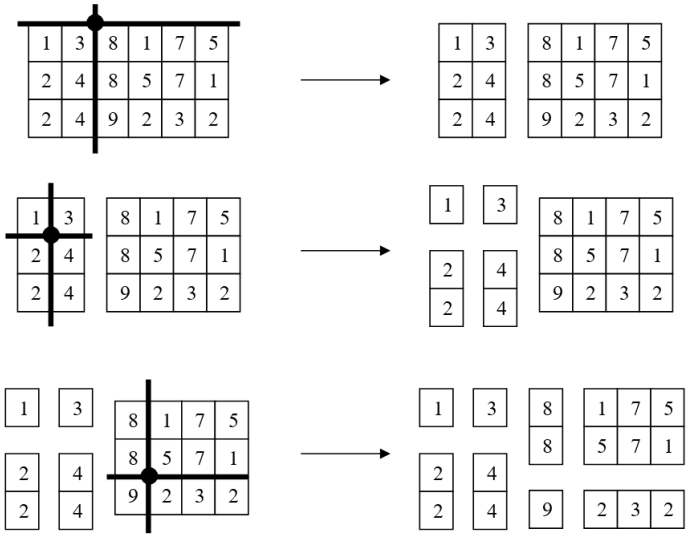


Figure 2.21. The optimal ball sequence for the first sample input and the resulting "submatrix map" with symmetric components

Output: For each input dataset, it must be printed the minimum number of balls that divides the corresponding mosaic into the minimum number of pieces such that each piece is symmetrically colored. For only the first input dataset, the details of the solution must also be presented. This means that the coordinates of the pieces (position of top left and bottom right corners) belonging to the optimal division (having the minimum number of pieces) must be printed in separate lines (if more than one optimal division has been found, then only one should be detailed).

Sample input

```
2
3 6
1 3 8 1 7 5
2 4 8 5 7 1
2 4 9 2 3 2
2 2
1 2
3 4
```

Sample output

```
3 8
1 1 1 1
1 2 1 2
2 1 3 1
2 2 3 2
1 3 2 3
3 3 3 3
1 4 2 6
3 4 3 6
1 4
```

2.6.1 Rationale and Solution

This problem belongs to the "submatrix – to – submatrix" category. We will determine the minimum ball number for all the submatrices in a bottom-up way. The suggested five-step method is followed below.

1. The general subproblem is to compute the minimum number of balls corresponding to the submatrix (i, j, li, lj), where i and j represent the indexes of the upper left corner element and li and lj are the number of rows and number of columns respectively. If we store the input matrix in array $a[0 \ldots n-1][0 \ldots m-1]$, we have the original problem for the parameters $(0, 0, n, m)$, and all symmetric submatrices can be considered trivial subproblems (1×1 submatrices are implicitly symmetric components).

2. For storing the optimal values, we need a four-dimensional array. Cell $c[i][j][li][lj]$ stores the minimum ball number for the submatrix (i, j, li, lj).

3. Since the submatrix (i, j, li, lj) can be hit by a ball in $li \cdot lj$ positions (see *Figure 2.22*) and a collision in position (pi, pj) results in four sub-subarray, the recursive formula is the following:

$$
\begin{aligned}
c[i][j][0][lj] &= 0, \quad 1 \le i \le n, 1 \le j \le m, 0 \le lj \le m - j + 1 \\
c[i][j][li][0] &= 0, \quad 1 \le i \le n, 1 \le j \le m, 0 \le li \le n - j + 1 \\
c[i][j][1][1] &= 0, \quad 1 \le i \le n, 1 \le j \le m
\end{aligned}
$$

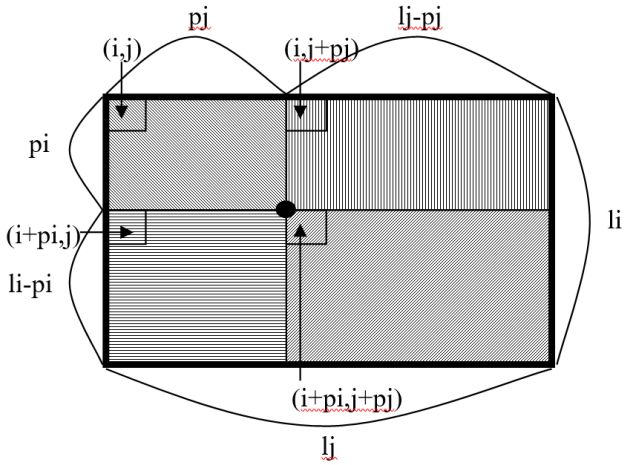


Figure 2.22. The four resulting sub-submatrices when the ball hits the current submatrix (represented by cell $c[i][j][li][lj]$) at position (pi, pj)

For all $1 \leq i \leq n, 1 \leq j \leq m, 0 \leq li \leq n - j + 1, 0 \leq lj \leq m - j + 1$.

If the corresponding matrix is symmetric $c[i][j][li][lj] = 0$.

Otherwise: for all $pi = 0 \ldots li - 1, pj = 0 \ldots lj - 1$.

$$\begin{aligned} c[i][j][li][lj] = \min\{ & c[i][j][pi][pj] \\ & + \; c[i + pi][j][li - pi][pj] \\ & + \; c[i][j + pj][pi][lj - pj] \\ & + \; c[i + pi][j + pj][li - pi][lj - pj]\} \end{aligned}$$

Since the current optimum incorporates more than one previously computed optimum from different previous levels, this problem belongs to the non-serial – polyadic category.

4. The first four nested loops implement the bottom-up solution building process (see the *C* code below). The cells representing the 1×1 trivial subproblems are filled during the first pass (for loop variable values $li = 1$ and $lj = 1$). The next two nested loops determine the optimum for the current subproblem (submatrix (i, j, li, lj)). Cell $c[i][j][li][lj]$ is computed according to the recursive formula detailed in the previous step.

5. If with respect to each subproblem we store the optimal values of the loop variables pi and pj (in another four-dimensional array: array b), then from this array the optimal solution (the optimal "submatrix map" resulted from the minimum number of collisions) can be extracted by the recursive function solution.

In the following, we present the author's code (this version does not compute the number of the submatrices resulting from the optimal decomposition of the original matrix).

The code

```
#include <stdio.h>
#include <stdlib.h>
#define N 32

typedef struct{
      int pi,pj;
}COORD;

int n,m,a[N][N],c[N][N][N][N];
COORD b[N][N][N][N];

int symmetric(int i,int j,int li,int lj);
void solution(int i,int j,int li,int lj);

int main()
{
    int t,k,i,j,li,lj,pi,pj,mpi,mpj,min,x;
    freopen("mozaic.txt","r",stdin);
    scanf("%i", &t); //number of test cases
    for( k = 0 ; k < t ; ++k ){
          scanf("%i%i", &n, &m); //reading input data
          for( i = 0 ; i < n ; ++i ){
              for( j = 0 ; j < m ; ++j ){
                  scanf("%i", &a[i][j]);
              }
          }
//considering all submatrices in bottom up way
          for(li=1; li<=n; li++){
              for(lj=1; lj<=m; lj++){
                  for(i=0; i<=n-li; i++){
                      for(j=0; j<=m-lj; j++){
                       if (symmetric(i,j,li,lj)){
//the current submatrix is symmetric
```

```
                                c[i][j][li][lj]=0;
                                b[i][j][li][lj].pi=-1;
                                b[i][j][li][lj].pj=-1;
                            }
                            else{
//the current cell is computed based on the recursive formula
                                min=n*m;
                                for(pi=0; pi<li; pi++){
                                    for(pj=0; pj<lj; pj++){
//for all possible collision points
                                     if(!(pi==0 && pj==0)){
                                          x=c[i][j][pi][pj][pj]
                                          +c[i+pi][j][li-pi]
                                          +c[i][j+pj][pi][lj-pj]
                                          +c[i+pi][j+pj][li-pi][lj-pj];
                                          if (x<min){
                                              min=x;
                                              mpi=pi;
                                              mpj=pj;
                                          }
                                       }
                                    }
                                }
                                c[i][j][li][lj]=min+1;
//storing the position of optimal collision point
                                b[i][j][li][lj].pi=mpi;
                                b[i][j][li][lj].pj=mpj;
                            }
                        }
                    }
                }
            }

            printf("%i\n", c[0][0][n][m]);
            if( k == 0 ){
//extracting the optimal "submatrix map" for the first test case
                solution(0,0,n,m);
            }
        }
        return 0;
}

int symmetric(int i,int j,int li,int lj){
    int pi,pj;
    for(pi=0; pi<li; pi++){
        for(pj=0; pj<lj; pj++){
```

```
            if( a[i+pi][j+pj] != a[i+li-1-pi][j+lj-1-pj]){
                return 0;
            }
        }
      }
      return 1;
}

void solution(int i,int j,int li,int lj){
  if( !li || !lj ){ return; }
  if( b[i][j][li][lj].pi == -1 ){
      printf("%i %i %i %i\n", i+1, j+1, i+li, j+lj);
  }
  else{
      solution(i,j,b[i][j][li][lj].pi,b[i][j][li][lj].pj);
      solution(i+b[i][j][li][lj].pi,j,li-b[i][j][li][lj].pi,
               b[i][j][li][lj].pj);
      solution(i,j+b[i][j][li][lj].pj,b[i][j][li][lj].pi,
               lj-b[i][j][li][lj].pj);
      solution(i+b[i][j][li][lj].pi,j+b[i][j][li][lj].pj,
               li-b[i][j][li][lj].pi,lj-b[i][j][li][lj].pj);
  }
}
```

A final note: If the slogan for the Greedy method is "live for today," then for dynamic programming the motto might be "look for the golden mean."

BIBLIOGRAPHY

[1] K. Kapoor: *Everything about Dynamic Programming.* Retrieved May 12, 2022 from https://codeforces.com/blog/entry/43256.

[2] Z. Kátai. 2021. *Algoritmustervezési stratégiák* [Algorithm Designing Strategies]. Scientia Publishing House, Cluj-Napoca (Romania).

CHAPTER 3

DIVIDE ET IMPERA

Author: Lehel István Kovács

Abstract. The Divide et Impera (or Divide and Conquer) algorithm strategy is a powerful problem-solving approach that involves breaking down a complex problem into smaller, more manageable subproblems, solving each subproblem independently, and then combining their solutions to solve the original problem. This technique is fundamental in computer science and is applied across various fields, from sorting algorithms to computational geometry. At its core, the Divide et Impera strategy follows three fundamental steps. First, it divides the problem into smaller, more easily solvable subproblems. Second, it conquers these subproblems by solving them recursively. Finally, it combines the solutions of the subproblems to obtain the solution to the original problem. Key examples of algorithms using this strategy include Merge Sort, Quick Sort, and Binary Search. These algorithms showcase the efficiency and effectiveness of Divide et Impera by significantly reducing the time complexity of various computational tasks. Overall, the Divide et Impera approach is a cornerstone of algorithm design, providing a structured and efficient method to tackle complex problems by breaking them down into simpler, more manageable components. Its versatility and applicability make it an invaluable tool in the realm of computer science and problem solving.

3.1 The Divide et Impera Approach

The *divide and rule* policy (Latin: *divide et impera*), or *divide and conquer*, in politics and sociology is gaining and maintaining power divisively. Historically and presently, this strategy was and is used in many different ways by empires seeking to expand their territories. The maxim *divide et impera* has been attributed to Philip II of Macedon, and it was utilized by the Roman ruler Julius Caesar. In computer science, the essence of the *Divide et Impera* method or algorithm design paradigm is that we try to divide

the problem to be solved into smaller subproblems. The subproblems are equivalent to the main problem, and their solution leads to the solution of the original problem. This is a useful strategy for solving difficult problems. It is a recursive method. A Divide et Impera algorithm recursively breaks down a problem into two or more subproblems of the same or related type until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem, as we can see it in *Figure 3.1.*

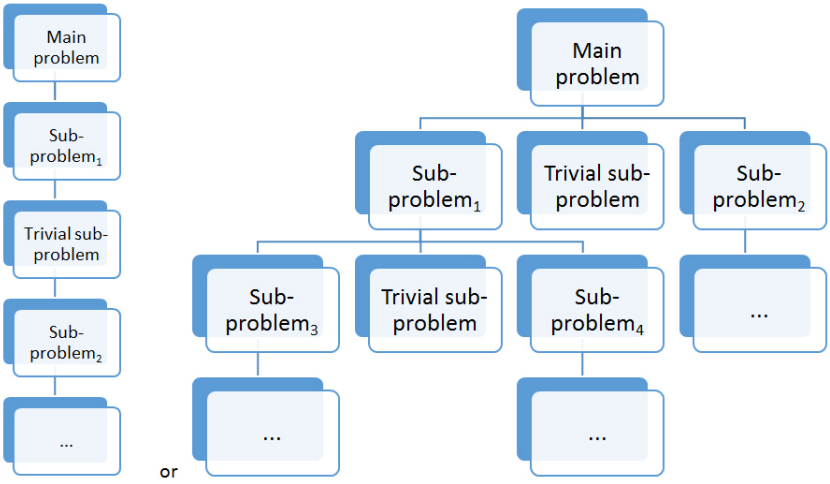


Figure 3.1. The principles of Divide et Impera algorithms

The Divide et Impera method is the basis of efficient algorithms, for example the sorting algorithms (e.g. quick sort, merge sort). Designing efficient Divide et Impera algorithms can be difficult. As in mathematical induction, it is often necessary to generalize the problem to make it amenable to a recursive solution. The correctness of a Divide et Impera algorithm is usually proved by mathematical induction (or the Master Theorem), and its computational cost is often determined by solving recurrence relations. The steps of a Divide et Impera algorithm:

1. Divide: Divide the given problem into subproblems using recursion.
2. Conquer: Solve the smaller subproblems recursively. If the subproblem is small enough, then solve it directly (Trivial subproblem).

3. Combine: Combine the solutions of the subproblems that are part of the recursive process to solve the actual problem.

Let us understand this concept with the help of an example:

A classical example of Divide et Impera algorithm is the *Towers of Hanoi Problem.*

3.2 The Towers of Hanoi Problem

3.2.1 The Legend

Once upon a time, in India, during the reign of Fo Hi, monks in the Shiva Temple in Benares (which marks the center of the world) had to move a pile of 64 sacred golden disks from one diamond peg to another. The disks were fragile; only one could be carried at a time. A disk could not be placed on top of a smaller, less valuable disk. And there was only one other diamond peg in the temple (besides the source and destination pegs) sacred enough that a pile of disks could be placed there. So, the monks started moving disks back and forth, between the original pile, the pile at the new location, and the intermediate location, always keeping the piles in order (largest on the bottom, smallest on the top) [1]. Legend has it that once the monks complete the task, the world would come to an end. So, we need to figure out how long it is going to take the monks to finish the puzzle.

How many moves will it take to transfer n disks from the source peg to the destination peg?

The abbot said that he would be happy to transfer one a single disc (the last one) if someone transferred the previous $n-1$.

The puzzle was introduced in 1883 by N. Claus (de Siam), Professor at University of Li-Sou-Stian, an anagram pseudonym for Édouard Lucas (D'Ameins), Professor at Lyceé Saint-Louis [2].

According to the literature – as we can see in the proof –, the Towers of Hanoi puzzle is a classical Divide et Impera problem. The puzzle can be solved by reducing the problem to smaller but similar subproblems; the key move is the transfer of the largest disc.

In this train of thought, we must transfer $n-1$ discs from the first peg to the third, then we can move the largest disc to the second peg, and finally we must transfer $n-1$ discs from the third peg to the second. According

to this recursive move sequence, it is easy to see that we need $2^n - 1$ steps to solve the n-discs puzzle.

So, the monks needs to make $2^{64} - 1$ (= 18 446 744 073 709 551 615) moves to solve the puzzle. Assuming that they are able to move one disc per second, the end of the world comes approximately in 590 000 000 000 years. (According to estimations, the Universe is 13,7 billion years old!) [3].

3.2.2 The Divide et Impera Solution

The following implementation is a pure Divide et Impera algorithm: the problem to be solved is divided in two/three simpler, similar subproblems.

```
1  void hanoi(int k, char s, char d, char h)
2  {
3          if(k==1) printf("%c -> %c\n", s, d);
4          else
5          {
6                  hanoi(k-1, s, h, d);
7                  printf("%c -> %c\n", s, d);
8                  hanoi(k-1, h, d, s);
9          }
10 }
```

Where:

- k is the number of discs
- s is the source peg
- d is the destination peg
- and h is the "helping peg."

By using Divide et Impera, we have solved the Towers of Hanoi problem by making recursive calls to slightly smaller Towers of Hanoi problems that, in turn, make recursive calls to yet smaller Towers of Hanoi problems. Together, the solutions form the solution to the whole problem. The concept (*Figure 3.2*): two recursive calls and moving one disk – this is the trivial subproblem (`printf("%c ->%c \n", s, d);`).

In *Figure 3.3*, we can see the solution and states of the Towers of Hanoi problem in the case of 3 discs.

In the following, we present a couple of *ECN problems* that can be solved using the Divide et Impera method.

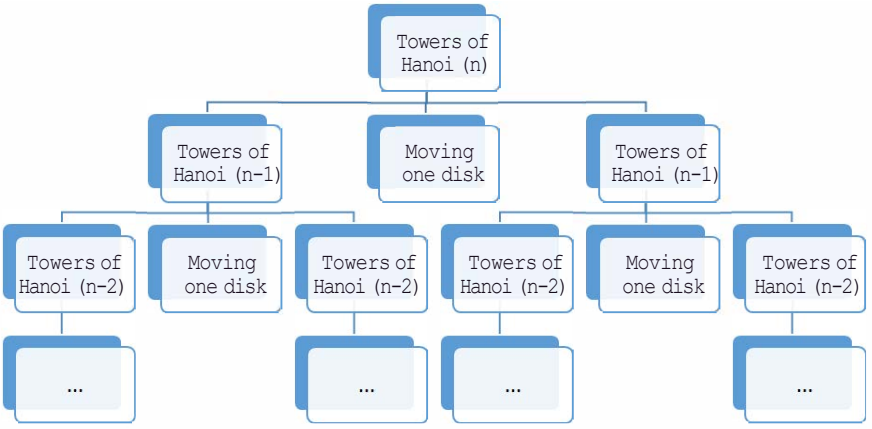


Figure 3.2. The concept of the Towers of Hanoi algorithm

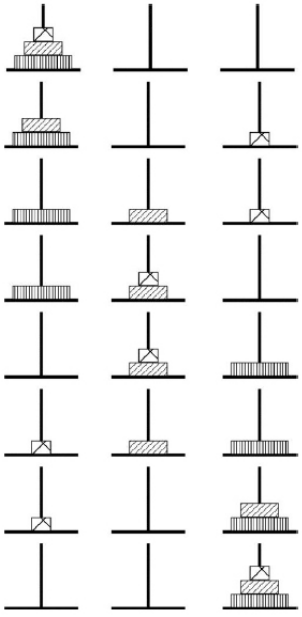


Figure 3.3. The Towers of Hanoi problem for 3 disks

3.3 ECN-2009, Problem D

3.3.1 Problem Statement

The Golden Patch

One day, while I was making experiences with *aqua regalis* – what an unfortunate accident –, I dripped it on my pretty new gold cigarette case. The drops burned out the cigarette case cover producing small, pin-sized holes on it.

I wished to remedy my mistake at once, so I went to find a golden patch to repair the cigarette case. I visited all the goldsmiths of the city, and finally I succeeded to draw the following conclusions:

- I must apply one single convex patch.
- The patch must be welded around with a thin gold filament.
- The gold is extremely expensive, so I need the smallest patch possible and the shortest filament possible.

For the given positions of holes, find out the properties of the patch and the length of the filament!

Input specification

The first line will contain the number of the test cases. Each case will contain in its first line a positive integer n, the number of points. Following this (n lines) come the integer coordinates of points: x *SPACE* y.

Output specification

For each case, the output will contain in its first line a positive integer m, the number of vertices of the convex polygon (the golden patch), then for the next m lines the integer coordinates of the vertices (x *SPACE* y).

Following these comes the length of the welder filament (real number – precision 4 decimal digits) and in the next line the area of the patch (real number – precision 4 decimal digits).

Sample input

```
1
13
-204 132
-151 147
```

```
-132 167
-81 144
-104 55
35 33
31 112
102 85
63 -61
-80 -50
67 171
49 -108
-122 -91
```

Sample output

```
7
63 -61
102 85
67 171
-132 167
-204 132
-122 -91
49 -108
981.5471
64923.5000
```

3.3.2 The Solution

The solution to the problem can be traced back to finding the convex hull.

A convex hull is the smallest convex polygon containing all the given points (*Figure 3.4*).

There are several algorithms for determining the convex hull – one approach is Divide et Impera.

The algorithm: Suppose we know the convex hull of the left half points and the right half points, then the problem now is to merge these two convex hulls and determine the convex hull for the complete set. This can be done by finding the upper and lower tangent to the right and left convex hulls.

The Divide et Impera algorithm therefore always divides the convex hull into two parts: the left and the right semi-convex hull (*Figure 3.5*). Such an algorithm is described, for example, in [4].

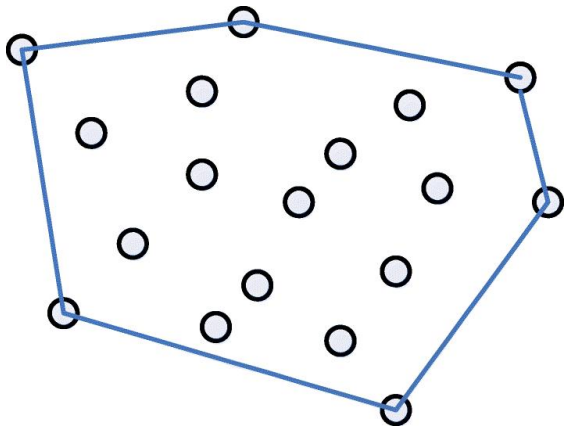


Figure 3.4. Convex hull of a set of points

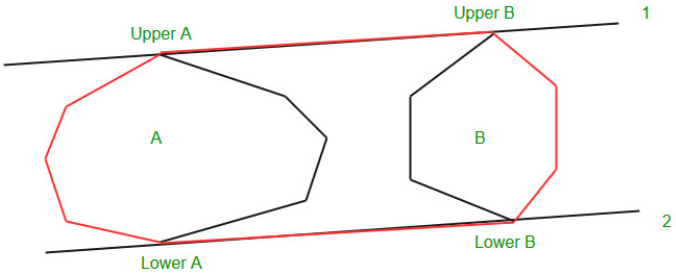


Figure 3.5. The Divide et Impera concept for finding a convex hull [4]

3.4 ECN-2010, Problem A

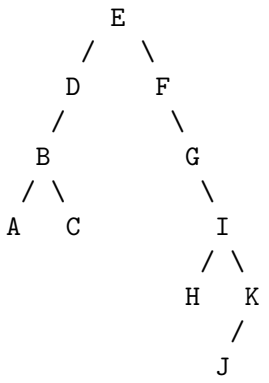
3.4.1 Problem Statement

Binary Tree

It is very easy to produce the various traversals of a binary tree; however, this problem requires producing the tree from the traversals. Specifically, given the pre-order and in-order traversals of a binary tree (not necessarily a binary search tree), reconstruct, if possible, the original tree.

(Source: NEW ZEALAND PROGRAMMING CONTEST, 2008)

For instance, the following tree will produce the traversals shown:



Pre-order: EDBACFGIHKJ
In-order: ABCDEFGHIJK
Post-order: ACBDHJKIGFE

Input specification

Input will consist of representations of several trees, each on a single line, and terminated by a #. Each line will consist of two strings of up to 26 unique uppercase letters in the form:
<pre-order> <space> <in-order>

Output specification

Output will consist of the post-order traversal of the reconstructed tree if possible, otherwise the words "invalid tree."

Sample input

```
EDBACFGIHKJ ABCDEFGHIJK
#
```

Sample input

```
ACBDHJKIGFE
```

3.4.2 The Solution

The construction of a binary tree is also a typical Divide et Impera algorithm.

Binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. So, the binary tree consists of: **LEFT**, **ROOT**, **RIGHT**.

The solution is quite simple: if the pre-order is `A B C D` and the in-order `B A D C`, then the binary tree is: `CreateTree(B)`, `Root(A)`, `CreateTree(D C)`.

The Divide et Impera approach can also be seen below (*Figure 3.6*).

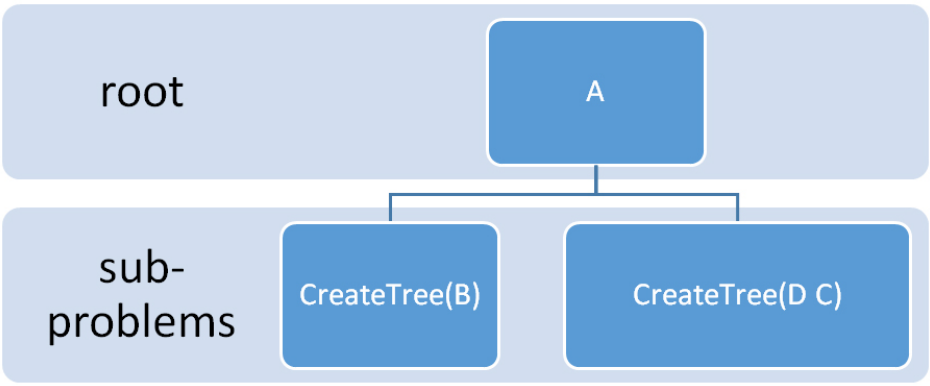


Figure 3.6. Create Tree – a Divide et Impera concept

3.4.3 The Algorithm

```
1
2  pozPreorder = 0;
3  Node * CreateTree(inorder, preorder)
4  {
5          if (length(inorder) == 0) return NULL;
6          Node * n = new Node(preorder[pozPreorder]);
7          pozInorder = find(inorder, preorder[pozPreorder]);
8          ++pozPreorder;
9          if(pozInorder == -1) "INVALID TREE"
10         left[n] = CreateTree(inorder[0..pozInorder-1], preorder);
11         right[n] = CreateTree(inorder[pozInorder+1..length-1], preorder);
12         return n;
13 }
```

3.5 ECN-2013, Problem H

3.5.1 Problem Statement

The Robot

A calculation-performing robot has been developed by engineers to be able to work very quickly with functions, and the same function can contain any number of arguments.

Unfortunately, the robot does not know anything else, so even a simple calculator can be realized by the robot as a sequence of function calls.

Write a program which converts a conventional arithmetic expression (that contains brackets "(“and”)", constant numbers, addition "+", subtraction "-", multiplication "*", division "/", exponentiation " ^ ", positive and negative signs "+" and "-") into nested function calls.

If the same arithmetic operation occurs repeatedly several times, the corresponding function will have the same number of arguments as the number of constants that carried out the operation.

Denote *add* the function for addition, *sub* the function for subtraction, *mul* – multiplication, *div* – division, *pow* – exponentiation, and *plus* and *minus* the functions for positive and negative signs.

Input specification

The first line of the input contains a positive integer T ($T \leq 100$), denoting the number of test cases. Then T lines follow, each containing a valid (syntactically correct) arithmetic expression (string – not longer than 1,000 characters).

Output specification

For each test case, write out a line (string) containing the arithmetic expression transcript to function calls.

Sample input

```
2
(3+4+5+6)/2
(1+2+3)*((7-4--2)^(3++2))
```

Sample output

```
div(add(3, 4, 5, 6), 2)
mul(add(1, 2, 3), pow(sub(7, 4, minus(2)), add(3, plus(2))))
```

3.5.2 The Solution

The solution is to build the Polish notation, where the operator comes before the operands (prefix notation). This parenthesis- and priority-free notation was invented in the 1920s by the Polish mathematician, logician, and philosopher Jan Łukasiewicz.

As presented in [5], the Polish notation can be constructed using an n-ary tree, where n is the number of operands of an operation. For the sake of simplicity, we now use binary trees when describing the algorithm, and then combine the operations in terms of operands.

From an expression, if we know the priority of the operations, we can build a binary tree by taking into account the following rules:

1. Operations are not terminal nodes.
2. Any terminal element is a variable or a constant.
3. The left and right sub-trees of any non-terminal node represent two (or one) operand(s) of the operation.
4. The root of the tree is the last operation to be performed.

First we need to establish a priority order. We use a variable priority to create a binary tree. This means that a general priority is given, and we change it to the corresponding symbols. So, we have to use two arrays; in one, we store the original expression, from which we filter out the parentheses, and in the other we store the priorities. Let the priority calculation rule be as follows: Let pr be the general priority, e the expression array, p the priority array, and s the scanned symbol.

- pr is initially 0.
- If s is a constant or a variable, then its priority is 1000, and this is added to p, and s is added to e.
- If s is an opening parenthesis "(", then we increase pr by 100.
- If s is a negative sign, we add $pr + 100$ to p, and we add some sign symbolizing s to e (e.g. "_").

- If s is a closing bracket ")", then reduce pr by 100.
- If s is "+" or "-" (binary), then $pr + 1$ goes into p and s into e.
- If s is "*" or "/", $pr + 10$ goes into p and s into e.

Thus, we have two arrays at our disposal, one containing the expression without the parentheses, the other containing the priority of the symbols in the expression. With the help of these two arrays, we recursively build a binary tree. We need the following information for the recursive algorithm:

- the two arrays;
- two values, the lower and the upper limit, which specify which part of the two arrays we are currently working with; initially, the lower one is 1 and the upper one is the length of the arrays.

The Divide et Impera algorithm:

1. In the lower upper part array, we look for the operation with the lowest priority from right or left depending on the associativity of operation – let it be the i-th.
2. Create the new node.
3. The data of the new node will be the operation received in 1.).
4. If lower = upper, the left and right part of the node will be empty.
5. Otherwise, if the operation is unary, we recursively build the right-hand sub-tree to $i + 1$ and upper limit.
6. If the operation is binary, then we build the left-hand sub-tree on lower and $i - 1$ and the right-hand sub-tree on $i + 1$ and upper limit respectively.

Example:
Let the expression be: $(a + b) * (c - d)$
The expression without parentheses and the priority array is:

- $e : a + b * c - d$
- $p : 1000\ 101\ 1000\ 100\ 1000\ 101\ 1000$

We can see the constructed binary tree in *Figure 3.7.*

For the Polish notation, we will traverse the binary tree pre-order, which is also a Divide et Impera algorithm:

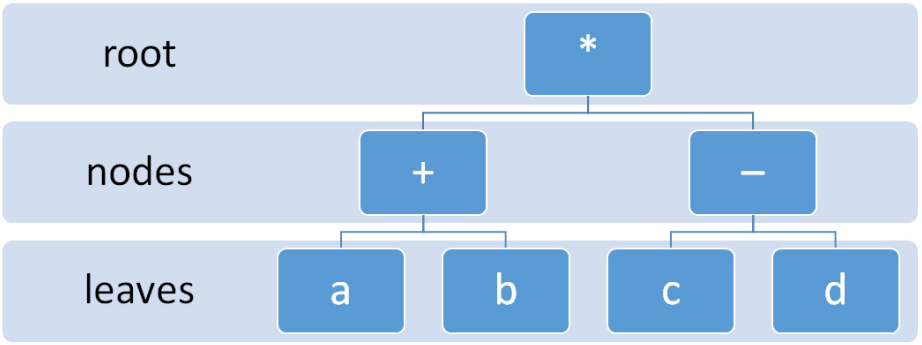


Figure 3.7. Binary tree – arithmetic operations

1. Visit the root.
2. Traverse the left sub-tree, i.e., `call Preorder(left->subtree)`.
3. Traverse the right sub-tree, i.e., `call Preorder(right->subtree)`.

The binary tree above results in the following expression in Polish notation after the pre-order traversal:

```
* + a b - c d
```

So, the robot produces the following sequence of operations:

```
mul(add(a, b), sub(c, d).
```

3.5.3 Program Parts

In the following, we provide a few more interesting details from the program in C++.

Processing the expression:

```
1  char expr[100];
2  printf("#>");
3  scanf("%s", &expr);
4  printf("The␣expression:␣%s\n", expr);
5
6  unsigned int i;
7  int E[100];
8  int P[100];
9  unsigned int index = 0;
10
11 int pr = 0;
12 int value = 0;
```

```
13
14  char constant[100] = "";
15  for(i = 0; i < strlen(expr); ++i)
16  {
17     if((strlen(constant) > 0)&&
18       ((expr[i] == '+')||
19        (expr[i] == '-')||
20        (expr[i] == '*')||
21        (expr[i] == '/')||
22           (expr[i] == '(')||
23           (expr[i] == ')')))
24     {
25           value = atoi(constant);
26           E[index] = value;
27           P[index] = const_pr;
28           ++index;
29           memset(constant, 0, sizeof(constant));
30      }
31           if(expr[i] >= '0' && expr[i] <= '9')
32           {
33                   sprintf(constant, "%s%c", constant, expr[i]);
34                   continue;
35           }
36           if(expr[i] == '+')
37           {
38               E[index] = oplus;
39             P[index] = pr + plus_pr;
40             ++index;
41                   continue;
42           };
43           if(expr[i] == '-')
44           {
45           E[index] = ominus;
46           P[index] = pr + minus_pr;
47           ++index;
48           continue;
49           };
50           if(expr[i] == '*')
51           {
52           E[index] = omul;
53           P[index] = pr + mul_pr;
54           ++index;
55           continue;
56           };
57           if(expr[i] == '/')
58           {
59           E[index] = odiv;
60           P[index] = pr + div_pr;
61                   ++index;
62                   continue;
63           };
64           if(expr[i] == '(')
65           {
66                   pr += obra_pr;
```

```
                continue;
        };
        if(expr[i] == ')')
        {
                pr += cbra_pr;
                continue;
        };
}
if(strlen(constant) > 0)
{
        value = atoi(constant);
        E[index] = value;
        P[index] = const_pr;
        ++index;
        memset(constant, 0, sizeof(constant));
}
```

Building the binary tree with the Divide et Impera algorithm:

```
void build(node *&tree, int be, int en, int* E, int* P)
{
         tree = (node*)malloc(sizeof(node));
         int i = 0, poz = en;
         int m = const_pr;
         for(i = en; i >= be; --i)
                 if(P[i]<m)
                 {
                         m = P[i];
                         poz = i;
                 }
         tree->value = E[poz];
         tree->left = NULL;
         tree->right = NULL;
         if(poz > be) build(tree->left, be, poz - 1, E, P);
    if(poz < en) build(tree->right, poz + 1, en, E, P);
}
```

Call:

```
node *tree = NULL;
build(tree, 0, index - 1, E, P);
```

Production of the Polish notation with pre-order traversal:

```
void preorder(node *tree, int *pn, unsigned int &i)
{
         pn[i] = tree->value;
```

```
4           ++i;
5           if(tree->left != NULL) preorder(tree->left, pn, i);
6       if(tree->right != NULL) preorder(tree->right, pn, i);
7   }
```

3.6 Divide et Impera – Advantages and Disadvantages

Finally, we summarize the advantages and disadvantages of Divide et Impera algorithms.

The Divide et Impera algorithmic strategy involves breaking down a problem into smaller, more manageable parts, solving each part independently and then combining the solutions to solve the original problem. This approach offers several advantages:

- *Efficiency*: Divide et Impera algorithms often have better time complexity than naive approaches to solving problems. By breaking the problem into smaller parts, the algorithm can reduce the overall computational complexity. For instance, algorithms like merge sort and quick sort use Divide et Impera to achieve efficient sorting of elements.
- *Parallelism*: Many Divide et Impera algorithms can take advantage of parallel processing or concurrent execution. Since the subproblems are often independent of each other, they can be solved simultaneously, making use of multiple processors or threads, thereby reducing the overall time taken to solve the problem.
- *Simplicity and modularity*: Breaking a complex problem into smaller, more manageable subproblems makes the overall algorithm easier to understand, implement, and maintain. It allows for a modular approach where each subproblem can be solved independently, simplifying the overall design.
- *Optimization potential*: By solving subproblems independently, there is often potential for optimization within each subproblem. This can lead to more efficient solutions, especially when optimizing within the smaller chunks of the problem.
- *Applicability*: Divide et Impera strategies are versatile and widely applicable across various domains. They are used in algorithms related to sorting, searching, optimization, and more.

- *Scalability*: Divide et Impera algorithms often exhibit good scalability. As the problem size increases, the algorithm can efficiently handle larger inputs by breaking them down into smaller parts.

However, it is important to note that Divide et Impera is not suitable for every problem, especially when the division step incurs high overhead or if combining the solutions from subproblems is complex. Additionally, there might be scenarios where the division itself introduces inefficiencies or where the subproblems are not independent, reducing the effectiveness of this strategy.

While the Divide et Impera strategy is powerful and widely used, it is not without its limitations and potential drawbacks:

- *Overhead from division*: In some cases, the process of dividing a problem into smaller subproblems incurs overhead. This division might involve additional computations, memory allocation, or recursion, which can offset the gains obtained from solving smaller subproblems independently.
- *Extra memory usage*: Some Divide et Impera algorithms might require additional memory to store intermediate results or to manage recursive function calls. This extra memory usage can be a concern, especially when dealing with large input sizes.
- *Difficulty in identifying subproblems*: Dividing a problem into appropriate subproblems is not always straightforward. Identifying the correct subproblems and determining the boundaries for division can be challenging. Inefficient division can lead to suboptimal solutions.
- *Complexity in combining solutions*: Combining the solutions from subproblems to solve the original problem might be complex in certain cases. The merging or combination step could have its own computational cost, especially if the merging process is not straightforward.
- *Not always optimal*: While Divide et Impera often improves efficiency, it is not always the most optimal strategy for every problem. There might be alternative approaches or specialized algorithms that outperform Divide et Impera for specific scenarios.
- *Recursive overhead*: Many Divide end Impera algorithms are implemented recursively, which can lead to stack overflow errors for very large input sizes due to excessive function calls. Iterative implementations might mitigate this, but they can be more complex.

– *Dependency among subproblems*: Some problems have interdependencies among subproblems, making it challenging to solve them independently. In such cases, the Divide et Impera strategy might not offer significant advantages.

Despite these disadvantages, Divide et Impera remains a powerful and widely used strategy in algorithm design. Its effectiveness depends on the nature of the problem being solved and how well the division, the solving of subproblems, and the combination of solutions are handled within the algorithm.

BIBLIOGRAPHY

[1] [no author]. 2002. Hanoi tornyai. In *Abacus Matematikai Lapok* 2001–2002.

[2] N. Claus [= Edouard Lucas]. 1883. *La Tour d'Hanoï: véritable casse-tête annamite*. Original instruction sheet printed by Paul Bousrez. Tours.

[3] Zoltán Kátai and Lehel István Kovács. 2009. Towers of Hanoi – Where programming techniques blend. In *Acta Univ. Sapientiae, Informatica* 1, 1, 89–108.

[4] *Convex Hull Using Divide and Conquer Algorithm*, Retrieved from https://www.geeksforgeeks.org/dsa/convex-hull-using-divide-and-conquer-algorithm/.

[5] D. Lehel István Kovács. 2004. *Programozási nyelvek összehasonlító elemzése – A programozási nyelvek anatómiája*. Presa Universitară Clujană, Cluj-Napoca.

CHAPTER 4

SORTING AND SEARCHING

Author: Ion Cozac

Abstract. This chapter presents some basics of sorting and searching algorithms, exemplified through several problems from the ECN programming contest. The proposed solutions are based on the C language (C99 standard).

Sorting: Introduction (see [1] for details)

To depict how to use sorting to solve problems, let us state the first problem. We read (from standard input) n integer values: $A_0, A_1, \ldots, A_{n-1}$. We must display these values in ascending order.

Table 4.1. Sample input/output

Standard input	Standard output
7 5 2 9 11 6 9 5	2 5 5 6 9 9 11

Let us sketch a very simple algorithm (Bubble sort) to solve this problem. The idea of this algorithm involves scanning the array several times and checking if two consecutive elements are properly ordered. If not, the two elements are interchanged. When the end of array is reached, the above procedure is repeated if there has been at least one swap. Every time the end of array is reached, one or more elements are properly ordered at the end of array. After the first step, at least one element (the highest value) is properly placed (the last position). After two steps, at least (the last) two elements are properly ordered.

Bubble sort algorithm

```
1 Input data: n (integer), A (array of n integer values);
2 Output data: A (array of integer values, ordered ascending);
```

```
3  Begin algorithm
4  Repeat
5      Let c := 0;
6      Let m := n - 2;
7      For i from 0 to m (step 1) do
8          If (Ai > Ai+1) then
9              Let t := Ai;
10             Let Ai := Ai+1;
11             Let Ai+1 := t;
12             Let c := 1;
13             End if;
14     End for;
15     Let m := m - 1;
16 until (c = 0);
17 End algorithm.
```

If we run the above algorithm step by step, the evolution of the array is the following:

5	2	9	11	6	9	5
2	5	9	11	6	9	5
2	5	9	6	11	9	5
2	5	9	6	9	11	5
2	5	9	6	9	5	11
-	-	-	-	-	-	-
2	5	6	9	9	5	11
2	5	6	9	5	9	11
-	-	-	-	-	-	-
2	5	6	5	9	9	11
-	-	-	-	-	-	-
2	5	5	6	9	9	11

Let us analyze this algorithm: how much time do we spend to obtain the ordered array? If the array is already sorted, we perform $n-1$ comparisons and 0 interchanges. If the array is decreasingly ordered, we perform $(n-1)+(n-2)+\ldots+1$ comparisons and interchanges. We are also interested about the average performance of this algorithm, so we attempt to make some approximations.

The minimum number of elementary operations (comparison, interchange) is $n-1$, and the maximum number is n^2. On average, the number of elementary operations is about $\frac{(n+1)(2n+1)}{6}$.

This approximation assumes that any case appears with the same probability.

We may approximate: the estimated time is proportional to n^2, so the time complexity is $\mathbf{O}(n^2)$.

There are various sorting algorithms [1], [**?**] that can be used for general cases (the structure of an element can be of any type):

- selection sort, insertion sort: time complexity is $\mathbf{O}(n^2)$;
- shell sort: time complexity is between $\mathbf{O}(n \log n)$ and $\mathbf{O}(n^2)$;
- merge sort, heap sort, quick sort: time complexity is $\mathbf{O}(n \log n)$.

In fact, the quick sort algorithm has the best performance (on average) for the majority of cases among the above algorithms. The standard library of the C language implements the `qsort` function, which can be used for any general case, so we do not have to worry about how to implement a very efficient sorting algorithm. The program below depicts the usage of `qsort` function, which is declared as follows (`qsort` manual):

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)
(const void *, const void *));
```

Source P01.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  static int A[100000], n;
5
6  static int compare(const void *A, const void *B) {
7  int *Pa = (int *)A, *Pb = (int *)B;
8  return *Pa - *Pb;
9  }
10
11 int main() {
12 int i;
13 scanf("%d",&n);
14 for (i=0; i<n; i++) scanf("%d",A+i);
15 qsort (A, n, sizeof(int), compare);
16 for (i=0; i<n; i++) printf("␣%d",A[i]);
17 printf("\n");
18 return 0;
19 }
```

The first argument of `qsort` is the array (in fact, its base) to be sorted. The second argument is the number of elements. The third argument is the size of an element of the array (every element is supposed to be of the same size). The fourth argument is the comparison function, which defines how

the array should be ordered. The programmer must define the comparison function.

To understand how this function should be defined, it is crucial to remember that `qsort` is designed for general purposes and lacks knowledge about the nature of the elements to be sorted. The comparison function takes two arguments of pointer type, each of them pointing to an element of the array. The function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the ordered array is undefined (`qsort` manual).

In this case, every element is of integer type, so we can use two pointers (`Pa` and `Pb`) to properly manage the values to be compared.

Additionally, we should also mention the issue of stability in sorting algorithms. Consider a scenario where we must sort a set of playing cards, with the sorting criterion being the rank of the cards.

Input data: 7 R, 9 B, 6 B, 7 B, 9 R

Output data (stable sort): 6 B, 7 R, 7 B, 9 B, 9 R

Output data (unstable sort): 6 B, 7 R, 7 B, 9 R, 9 B

In the second case (unstable sort), the initial order was: 9 B followed by 9 R. The sorted array fails to maintain the same order for every two elements with the same sorting key. If preserving the order of multiple elements sharing the same sorting key is essential, we must use a supplementary sort key (as demonstrated in the example below: Information about persons).

Sorting problem: National Lottery ([2], problem A)

The National Lottery records the following data:

- the identifier of the gamble (nine decimal digits);
- five distinct values between 1 and 40 (numbers chosen by the gambler).

Every identifier is greater than 0 and may appear more than once (every gambler may play one or more distinct variants), excepting the last line, where the identifier is zero. The last line is reserved for the National lottery (the extracted, winning numbers). The Lottery must identify the following winners:

- first class: the gambler guessed all five numbers (from the last line);
- second class: the gambler guessed four numbers out of five.

Every gambler must be displayed once and will be assigned to the best class he/she has achieved.

Table 4.2. Input and output for P02.c

Standard input	Standard output
100000000 5 40 3 2 1 100000000 5 40 2 3 6 200000000 40 5 6 1 2 300000000 9 8 7 1 20 200000000 40 5 6 1 3 400000000 1 2 3 4 5 0 1 2 3 40 5	First class: 100000000 Second class: 200000000 400000000

Source P02.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct {
        int key, class;
        char numbers[41];
        } T_lottery;
static T_lottery Info[350001], *Win;
static int ng;

static void readData(void) {
int i,v;
for (Win=Info; ; Win++) {
    scanf("%d",&Win->key);
    for (i=0; i<5; i++) {
        scanf("%d",&v);
        Win->numbers[v] = 1;
        }
    if (Win->key==0) break;
    Win->class = 3;
    }
ng = Win - Info;
}

static int compareA(const void *A, const void *B) {
T_lottery *Pa = (T_lottery *)A, *Pb = (T_lottery *)B;
return Pa->class - Pb->class;
}

static void identifyWinners(void) {
T_lottery *Pinf;
```

```
33 int i,s;
34 for (Pinf=Info; Pinf<Win; Pinf++) {
35     for (s=0,i=1; i<=40; i++)
36         s += Pinf->numbers[i] == Win->numbers[i];
37     if (s==40) Pinf->class = 1;
38     if (s==38) Pinf->class = 2;
39     }
40 qsort(Info,ng,sizeof(T_lottery),compareA);
41 for (Pinf=Info; Pinf->class<=2; Pinf++) ;
42 Pinf->key = 0;
43 ng = Pinf - Info;
44 }
45
46 static int compareB(const void *A, const void *B) {
47 T_lottery *Pa = (T_lottery *)A, *Pb = (T_lottery *)B;
48 int d;
49 if (d = Pa->key - Pb->key) return d;
50 return Pa->class - Pb->class;
51 }
52
53 static void purgeDuplicates(void) {
54 T_lottery *Pinf, *Paux;
55 qsort(Info,ng,sizeof(T_lottery),compareB);
56 for (Pinf=Info; Pinf->key; Pinf=Paux)
57     for (Paux=Pinf+1; Paux->key==Pinf->key; Paux++)
58         Paux->class = 3;
59 }
60
61 static void displayWinners(void) {
62 T_lottery *Pinf;
63 printf("First␣class:");
64 for (Pinf=Info; Pinf->key; Pinf++)
65     if (Pinf->class==1) printf("␣%d",Pinf->key);
66 printf("\nSecond␣class:");
67 for (Pinf=Info; Pinf->key; Pinf++)
68     if (Pinf->class==2) printf("␣%d",Pinf->key);
69 printf("\n");
70 }
71
72 int main() {
73 readData();
74 identifyWinners();
75 purgeDuplicates();
76 displayWinners();
77 return 0;
78 }
```

Data initialization

The `class` member of the structure may have three values: 3 (record is non-winning), 2 (second class), 1 (first class). Initially, every record is considered to be non-winning. After reading the input data in the `Info` array, the `Win` pointer will point to the last record, which contains the winning numbers. `ng` keeps the actual number of records, excluding the last one (which is reserved for the Lottery).

Winner identification process

How do we identify the winners? Every record uses the `members[]` array to store the chosen numbers as follows: if $v(1 \leq v \leq 40)$ is chosen, then `members[v]` is set to 1. We simply count the number of matches between the current record and the `Win` record (the last one). If the number of matches is 40, `class` is set to 1. If the number of matches is 38, `class` is set to 2.

Sorting and selection of winners

After identifying the winners, we sort the `Info` array with `class` member as key sort (`compareA` function). Next, we scan the array to identify the first non-winning record and set its `key` to zero. This position will be further used as sentinel to check if the end of records is reached. Now the actual number of records (`ng`) is set to the number of winners, and from now on we will deal only with them (the non-winning records will be ignored).

Handling duplicates in winner selection

But the list of winners may include a lot of duplicates. Some gamblers played two or more variants, maybe one of them belonging to the first class, other variants belonging to the second class. So, we must keep for every gambler the first variant, which gives him/her the best possible gain. To do this, we must resort to the `Info` array using two sort keys: the first key is the gambler identifier (`key` member), and the second key is the `class` member (using the `compareB` function).

Reassignment for Best Possible Gain

Now, for every record belonging to a specific gambler (`Pinf` pointer), we scan the next records belonging to the same gambler (`Pinf->key == Paux->key`) and mark them as non-winning (set `class` to 3).

Final winner display

Finally, we can display the winners.

Sorting problem: Information about persons ([3], problem A)

The first line of the input data contains the number of records (about persons). Every next line contains information about one person:

- personal code (nine decimal digits);
- status (two to seven non-space characters); the first character is always M or F (sex); if the second character is – (minus), the person is dead; otherwise, if one of the next characters is C, the person is computer scientist; if one character is S, the person works for the secret service;
- name of the person (8 to 35 characters, including space).

The list above may contain duplicates: whenever a person changes their status or name, a new record is simply added at the end of the list. We must display:

- how many men are alive;
- how many women are alive;
- how many computer scientists are employed;
- how many persons are working for the secret service;
- the list of computer scientists who work for the secret service (in alphabetical order).

Table 4.3. Input and output for P03.c

Standard input	Standard output
12	3 men
10 FAB Blue Mary	4 women
15 MCA Sonk Jean	4 computer scientists
10 FAS Rode Mary	5 for secret service
20 FXS Aske Vera	
20 F- Aske Vera	24 FCS Ando Maria
12 FDE Kant Mia	16 MCS Barbe Louis
12 FSE Kant Mia	21 MSC Conta Pietro
16 MUV Barbe Louis	
21 MSC Conta Pietro	
24 FCS Ando Maria	
16 MCS Barbe Louis	
25 FET Donk Laura	

Source P03.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct {
        int key, pos;
        char stat[8], name[36];
        } T_person;
static T_person *Pers;
static int npers;

static int compareKeys(const void *A, const void *B) {
int d;
T_person *Pa = (T_person *)A, *Pb = (T_person *)B;
if (d = Pa->key - Pb->key) return d;
return Pb->pos - Pa->pos;
}

static void readData(void) {
T_person *Pp;
int i;
scanf("%d",&npers);
Pers = calloc(npers+1,sizeof(T_person));
for (i=0,Pp=Pers; i<npers; i++,Pp++) {
    Pp->pos = i;
```

```
26      scanf("%d␣%s␣%[^\n]",&Pp->key,Pp->stat,Pp->name);
27      }
28  Pp->key = -1;
29  qsort(Pers,npers,sizeof(T_person),compareKeys);
30  }
31
32  static int comparePurge(const void *A, const void *B) {
33  int d, a, b;
34  T_person *Pa = (T_person *)A, *Pb = (T_person *)B;
35  a = Pa->pos >= 0;
36  b = Pb->pos >= 0;
37  if (d = b - a) return d;
38  return strcmp(Pa->name, Pb->name);
39  }
40
41  static void purgeData(void) {
42  T_person *Pa, *Pb;
43  for (Pa=Pers; Pa->key>0; Pa=Pb)
44      for (Pb=Pa+1; Pa->key==Pb->key; Pb++)
45          Pb->pos = -1;
46  qsort(Pers,npers,sizeof(T_person),comparePurge);
47  for (Pa=Pers; Pa->pos>=0; Pa++) ;
48  Pa->key = 0;
49  npers = Pa - Pers;
50  }
51
52  static void displayResults(void) {
53  T_person *Pp;
54  int nmen, nwom, ncs, nss;
55  nmen = nwom = ncs = nss = 0;
56  for (Pp=Pers; Pp->key; Pp++) {
57      if (Pp->stat[1]=='-') continue;
58      if (Pp->stat[0]=='M') nmen++;
59      if (Pp->stat[0]=='F') nwom++;
60      if (strchr(Pp->stat,'C')) ncs++;
61      if (strchr(Pp->stat,'S')) nss++;
62      }
63  printf("%d␣men\n%d␣women\n%d␣computer␣scientists\n
64  ␣␣␣␣␣␣␣␣%d␣for␣secret␣service\n\n",nmen,nwom,ncs,nss);
65  for (Pp=Pers; Pp->key; Pp++) {
66      if (Pp->stat[1]=='-') continue;
67      if (strchr(Pp->stat,'C'))
68          if (strchr(Pp->stat,'S'))
69              printf("%d␣%s␣%s\n",Pp->key,Pp->stat,Pp->name);
70      }
71  }
```

```
72
73 int main() {
74 readData();
75 purgeData();
76 displayResults();
77 return 0;
78 }
```

Before reading the input data, we allocate memory for `npers + 1` records. The last record (which does not contain any useful information) is used as sentinel (`pos` member is set to –1).

After reading the input data, we must eliminate the useless records. To do this efficiently, we must sort the array using personal code as sort key. But this is not enough because the `qsort` function does not guarantee the same order for two distinct records having the same key. So, how do we know which is the last record associated to a certain person? This is why we use a supplementary member to store the initial position of every record (`pos`). The `compareKeys` function uses two keys to sort the array: personal code (key member, ascending order) and initial position (`pos` member, descending order).

For every person in the sorted array, the first position corresponds to the last information about that person, so we can purge the data. The useless records are identified by setting the pos member to –1. We alphabetically resort the `Pers` array. The `comparePurge` function uses as first sort key the status of pos member, and name member as sort key (because we must display an alphabetically ordered list). The statistical information can be obtained no matter how the array is sorted.

We said the first key sort of `comparePurge` is the status of `pos` member (if the element is valid or not). The a variable is set to 1 if `Pa->pos` ≥ 0, and 0 otherwise; the same is valid for b variable. The `Pers` array is sorted using descending order for the status of `pos`, so the valid (last) records are placed at the beginning of the array. The second sort key is the `name` member.

First remark. When reading the input data, we use the format descriptor `\%[^\n]`, which means a set of characters until the end of line (excluding `\n`).

Second remark. The `comparePurge` function uses, as the second sort key, the name member. The comparison is made by the `strcmp` function (from standard library), which returns a negative value if the first string is less than the second one (`lexicographically`), zero if the strings match, and so on (`strcmp` manual).

Searching algorithms: Searching in a static set (see [1], [?] for details)

Let us state the following problem. We read (from standard input) n integer values: $A_0, A_1, \ldots, A_{n-1}$ (the static set). Next, we read a set of integer values (one value at a time) and display if the value is in the array or not.

We define the following searching function (linear search): it returns a positive or zero value (the position of the value) if v is in the array, or -1 otherwise.

```
1 Function position-in-array (v: integer, A: array, n: integer):
2 For i from 0 to n - 1 (step 1) do
3     If (Ai = v) then return i;
4     End for;
5 Return -1;
6 End function;
```

Let us analyze this algorithm: how much time do we spend to find value v? If $A_i = v (0 \leq i < n)$, we perform $i + 1$ comparisons to return the position in the array. If v is in the array, we perform (on average) a number of $(n + 1)/2$ comparisons to return the position. If v is not in the array, n comparisons are needed. The time complexity of this implementation is $\mathbf{O}(n)$ on average (time is linearly dependent on n).

Let us suppose n is 100.000, and we must perform 80.000 searches: 40.000 values are found in the second half of the array (position $\geq$ 50.000) and 40.000 values are not found. To process the first 40.000 values, we need at least $40.000 * 50.000$ comparisons (or more). To process the next 40.000 values, we need $40.000 * 100.000$ supplementary comparisons. We have to perform at least 6.000.000.000 comparisons, which takes a lot of time. Can we obtain better performance?

The solution is to sort the input array before reading any v value, such as the values appear in ascending order. We can do this because the input array is static (their values are not modified later). Now, given an integer value v, we can search it more efficiently using the ***binary search algorithm***.

The idea of this algorithm is the following (recursive description). First, the interval of searching is the whole array (positions from 0 to $n-1$). If the value from the middle position is equal to v, then this position is returned. Otherwise, we consider two distinct subintervals: the left half interval and the right half interval (the middle position is no longer included). If v is less than the value from the middle position, we will consider from now on

only the left half interval. If v is greater than the value from the middle position, we will consider from now on only the right half interval. If the search interval is empty, the value is not found.

Now let us analyze the complexity of the binary search algorithm. To simplify the exposition, let us suppose $n = 2k - 1$. The first step deals with $2^k - 1$ values, so we make one comparison. If the value is not found, the second step deals with $2^{k-1} - 1$ values, and we make another comparison. The worst case appears when the size of the interval is 1, leading to the final comparison. The maximum number of comparisons is k, which is (approximately) $\log n$ (regardless of the base of logarithm).

Conclusion: the time complexity of binary search algorithm is $\mathbf{O}(\log n)$. Fortunately, the standard library of the C language implements the `bsearch` function, which can be used for any general case, so we do not have to worry about how to implement a very efficient sorting algorithm.

```
void *bsearch(const void *key, const void *base, size_t nmemb,
size_t size, int (*compar)(const void *, const void *));
```

The following `program demonstrates` the usage of `qsort` and `bsearch` functions.

Do not forget: `qsort` and `bsearch` should use the same compare function.

Table 4.4. Input and output for P04.c

Standard input	Standard output
5 7 2 9 4 8	2 found
2	7 found
7	6 not found
6	10 not found
10	

Source P04.c

```
#include <stdlib.h>
#include <stdio.h>

static int A[100000], n, v, *P;

static int compare(const void *A, const void *B) {
int *Pa = (int *)A, *Pb = (int *)B;
return *Pa - *Pb;
```

```
}

int main() {
int i;
scanf("%d",&n);
for (i=0; i<n; i++) scanf("%d",A+i);
qsort (A, n, sizeof(int), compare);
while (scanf("%d",&v)>0) {
      if (P = bsearch(&v, A, n, sizeof(int), compare))
         printf("%d␣found\n",v);
      else printf("%d␣not␣found\n",v);
      }
return 0;
}
```

The `bsearch` function returns a pointer to a matching element of the array, or NULL if no match is found. If there is more than one matching element, the returned position is unspecified (`bsearch` manual).

The `bsearch` function returns a pointer of void type. The C language allows such assignments: void pointer can be assigned to a pointer of any type, and vice versa. The C++ language does not allow such assignments, so cast type is mandatory.

Searching algorithms: Searching in a dynamic set (see [4] for details)

Let us state the following problem. An initial void list of pairs (key, name) is created. We read (from standard input) one pair (positive integer value, and name: up to 35 non-space characters) at a time. Every piece of information is processed as follows:

- if the key is new, insert it in the map and display the message: *New key name*;
- if the key already exists, replace the old information with the new one and display the message: *Update code name*.

When the input data is exhausted, display the ordered content of the set.

We do not know in advance the number of pairs, and static arrays cannot be used. We must allocate memory space for every new pair.

An element of a static array may be simply accessed by A_i; its position may be easily determined. To manage a dynamic set, we must know how to scan it:

- where the first element of the set is located;

Table 4.5. Input and output for P05.c and Po7.cpp

Standard input	Standard output
7 seven	New 7 seven
5 five	New 5 five
9 nine	New 9 nine
5 cinq	Update 5 cinq
2 two	New 2 two
10 ten	New 10 ten
9 neuf	Update 9 neuf
	2 two
	5 cinq
	7 seven
	9 neuf
	10 ten

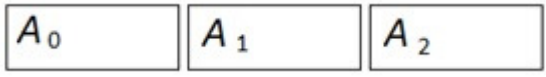


Figure 4.1. A static array uses compact memory

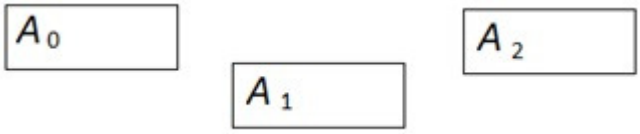
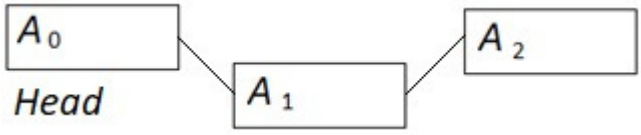


Figure 4.2. The elements of a dynamic set are scattered through memory

– if we know the position of an element, where is the next element (if it exists), or a special mark for the last element (linked list [4]).



A linked list is a set of nodes, and each node contains data members and link to the next elements (or NULL for the last element). But wait! A linked list is not efficient enough if we must perform a lot of searches, so we need a more elaborate data structure: a binary search tree.

Source P05.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

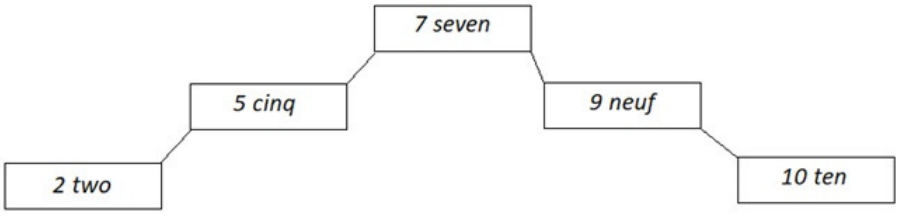
typedef struct S_bstree {
        int key;
        char name[36];
        struct S_bstree *left, *right;
        } T_bstree;
static T_bstree *Set;

static T_bstree *newElement(T_bstree *newEl,
       T_bstree *root) {
int d;
if (root==NULL) {
   root = calloc(sizeof(T_bstree),1);
   root->key = newEl->key;
   strcpy(root->name,newEl->name);
   return root;
   }
d = newEl->key - root->key;
if (d<0) root->left = newElement(newEl, root->left);
if (d>0) root->right = newElement(newEl, root->right);
return root;
}

static T_bstree *searchElement(int key, T_bstree *root) {
int d;
if (root==NULL) return NULL;
d = key - root->key;
if (d<0) return searchElement(key, root->left);
if (d>0) return searchElement(key, root->right);
return root;
}

static void printSet(T_bstree *root) {
if (root) {
   printSet(root->left);
   printf("%d␣%s\n",root->key,root->name);
```

```
40      printSet(root->right);
41      }
42  }
43
44  int main() {
45  T_bstree newEl, *tmp;
46  while (scanf("%d␣%s",&newEl.key,newEl.name)>0) {
47        tmp = searchElement(newEl.key, Set);
48        if (tmp) {
49           printf("Update");
50           strcpy(tmp->name, newEl.name);
51           }
52        else {
53           printf("New");
54           Set = newElement(&newEl, Set);
55           }
56        printf("␣%d␣%s\n",newEl.key,newEl.name);
57        }
58  printf("\n");
59  printSet(Set);
60  return 0;
61  }
```



This binary search tree is created by the above program. The root of this tree is (7, seven) which has two subtrees, one on the left and one on the right. The tree has two (2 two) and ten (10 ten) leaves: a leaf has no subtrees. The structure of an element of the tree contains the following members: key, name, link to the left subtree, link to the right subtree. If a link (pointer) is not NULL, the corresponding subtree is not empty. After reading the input data, the elements of the set are displayed using the inorder method to scan the binary tree. Recursively: first process the left subtree, then the root, and finally the right subtree. The time complexity of search and insertion operations is between $\mathbf{O}(\log n)$ and $\mathbf{O}(n)$. The worst case occurs, for example, when the input data is already ordered. This is why we should use simple binary search trees very carefully, especially if

we have no information about the nature of input data. A special case of binary tree is AVL tree (balanced binary tree [7]), which is very efficient for such problems. For implementation, we can successfully use associative containers (maps – see the last section of this chapter).

Sorting and searching problem: Timetable update ([5], problem F)

The first line of the input data contains a value ($1 \leq n \leq 150000$), the number of stations used by the previous timetable. Each of the next n lines contains information about a known station: its internal code (nine decimal digits) and its name (up to 35 characters, including spaces). A new list of station names (without internal code), used by the actual timetable, follows the first list.

Table 4.6. Input and output for P06.c

Standard input	Standard output
5	Stations not used:
1 Paris Nord	3 Berlin Westen
2 Madrid Sur	5 Wien
3 Berlin Westen	New stations:
4 Roma Est	Bucuresti Nord
5 Wien	Budapest Keleti
Paris Nord	
Roma Est	
Budapest Keleti	
Bucuresti Nord	
Paris Nord	
Bucuresti Nord	
Madrid Sur	

Some stations appear in the first list: Paris Nord, Roma Est, Madrid Sur. Some new stations do not appear in the first list: Bucuresti Nord, Budapest Keleti. Some old stations no longer appear in the new list: Berlin Westen, Wien.

We must identify which stations no longer appear in the new list, displaying the internal code and the name, ordered by the internal code. We must also identify which new stations do not appear in the first list, displaying their names, ordered lexicographically.

Source P06.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct {
        int key, used;
        char name[36];
        } T_station;
static T_station *Stns;
static int nstns;

typedef struct S_stree {
        char name[36];
        struct S_stree *left, *right;
        } T_stree;
static T_stree *Newst;

static int compareName(const void *A, const void *B) {
T_station *Pa = (T_station *)A, *Pb = (T_station *)B;
return strcmp(Pa->name,Pb->name);
}

static void readPreviousData(void) {
int i;
T_station *Pst;
scanf("%d",&nstns);
Stns = calloc(nstns,sizeof(T_station));
for (i=0,Pst=Stns; i<nstns; i++,Pst++)
    scanf("%d␣%[^\n]\n",&Pst->key,Pst->name);
qsort(Stns,nstns,sizeof(T_station),compareName);
}

static T_stree *newStation(char *Ns, T_stree *root) {
int d;
if (root==NULL) {
   root = calloc(sizeof(T_stree),1);
   strcpy(root->name,Ns);
   return root;
   }
d = strcmp(Ns,root->name);
if (d<0) root->left = newStation(Ns, root->left);
if (d>0) root->right = newStation(Ns, root->right);
return root;
}

```

```
46 static void readNewData(void) {
47 T_station *Pst, Wst;
48 while (scanf("␣%[^\n]",Wst.name)>0) {
49       Pst = bsearch(&Wst,Stns,nstns,sizeof(T_station),
50             compareName);
51       if (Pst) {
52          Pst->used = 1;
53          continue;
54          }
55       else Newst = newStation(Wst.name,Newst);
56       }
57 }
58
59 static void printStree(T_stree *root) {
60 if (root) {
61    printStree(root->left);
62    printf("␣%s\n",root->name);
63    printStree(root->right);
64    }
65 }
66
67 static int compareKey(const void *A, const void *B) {
68 T_station *Pa = (T_station *)A, *Pb = (T_station *)B;
69 return Pa->key - Pb->key;
70 }
71
72 static void displayResults(void) {
73 int i;
74 T_station *Pst;
75 qsort(Stns,nstns,sizeof(T_station),compareKey);
76 printf("Stations␣not␣used:\n");
77 for (i=0,Pst=Stns; i<nstns; i++,Pst++)
78     if (Pst->used==0) printf("␣%d␣%s\n",Pst->key,Pst->name);
79 printf("New␣stations\n");
80 printStree(Newst);
81 }
82
83 int main() {
84 readPreviousData();
85 readNewData();
86 displayResults();
87 return 0;
88 }
```

After reading the previous data, we sort the `Stns` array using the station name as sort key. This way, we can later use the binary search method. If

a name from the new list is found in the `Stns` array, the used member of the corresponding record is set to 1, otherwise we try to insert it into the binary tree (the insert operation succeeds only if the name was not previously inserted).

We use only the `newStation` function to insert a new name if it does not exist. If the name already exists, nothing is done, so there is no need for any search function in the binary tree.

To display the information about the stations which are not used, we must first resort the `Stns` array using the station identifier as sort key. Finally, we use the inorder traversal to display the list of new stations.

Introduction to associative containers (maps) (see [6] for details)

Maps are associative containers which store elements composed of key and value, every key being unique. Some basic operations supported (which are similar to those supported by a binary search tree):

- create an empty container;
- insert new element;
- remove existing element;
- find element with a given key;
- scan the container.

Associative containers are faster than simple binary search trees, and the time complexity is $\mathbf{O}(\log n)$. To depict the usage of maps, we will rewrite P05.c source, this time using C++ language.

Source P07.cpp

```
#include <iostream>
#include <iterator>
#include <map>
using namespace std;

map <int, string> List;

int main() {
int k;
string N;
map <int, string>::iterator Itr;
while (cin >> k >> N) {
      Itr = List.find(k);
      if (Itr!=List.end()) {
         cout << "Update ";
```

```
16          List.erase(k);
17          }
18       else cout << "New␣";
19       List.insert(pair<int,string>(k,N));
20       cout << k << '␣' << N << endl;
21       }
22 cout << endl;
23 for (Itr=List.begin(); Itr!=List.end(); Itr++)
24     cout << Itr->first << '␣' << Itr->second << endl;
25 return 0;
26 }}
```

The reader may solve certain problems using associative containers (or other appropriate structures) and compare the performances (running time) to those obtained by C programs (utilizing qsort and bsearch). In programming contests, we may use this technique to solve problems because it allows us to develop functional programs quickly, which are fast enough as per contest requirements. We also need to develop programs to be used by customers, which must deal with large amounts of data. In such cases, we should adopt highly efficient methods.

BIBLIOGRAPHY

[1] *Searching and Sorting*. Retrieved from
https://www.lkouniv.ac.in/site/writereaddata/siteContent
/202003251324427324himanshu_Searching_Sorting.pdf

[2] ECN contest 2012
Problems:
https://ecn.ms.sapientia.ro/problems/ECN_Informatics_2012.pdf.
Input Data and Output Results:
https://ecn.ms.sapientia.ro/problems/InOut2012.zip.

[3] ECN contest 2018
Problems:
https://ecn.ms.sapientia.ro/problems/ECN_Informatics_2018.pdf.
Input Data and Output Results:
https://ecn.ms.sapientia.ro/problems/InOut2018.zip.

[4] *Linked List Data Structure*. Retrieved from
https://www.geeksforgeeks.org/datastructures/linked-list/.

[5] ECN contest 2014
Problems:
https://ecn.ms.sapientia.ro/problems/ECN_Informatics_2014.pdf.
Input Data and Output Results:
https://ecn.ms.sapientia.ro/problems/InOut2014.zip.

[6] *Maps in C++ Standard Template Library*. Retrieved from
https://www.geeksforgeeks.org/cpp/map-associative-containers-the-c-standard-template-library-stl/.

[7] *AVL Tree*. Retrieved from
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/.

CHAPTER 5

ALGORITHMIC EXPLORATIONS IN COMBINATORIAL CHALLENGES: UNRAVELING COMPLEXITY

Authors: Zoltán Szabó, Boróka Olteán-Péter

5.1 Introduction

Several challenging ECN problems had combinatorial backgrounds. We provide a brief glimpse into this problem family through two so-called mountain problems.

Permutations, often encountered in combinatorics and algorithmic problem-solving, play a crucial role in various mathematical and computational scenarios. In this context, we explore a fascinating subset of permutations known as "mountain permutations." A permutation with N elements is considered a mountain permutation if it exhibits a distinctive peak, forming a sequence that ascends from the beginning to a certain point and then descends until the end.

This intriguing pattern is formally defined by an index element M such that $1 < M < N$, and the sequence up to M is strictly ascending, while the sequence from $M + 1$ to N is strictly decreasing. To manipulate permutations and explore their mountainous nature, two symmetric exchange operations, *swap* and *double swap*, are introduced. These operations allow for the rearrangement of elements in a permutation, offering a pathway to transform a given sequence into a mountain permutation.

The problem at hand poses a challenge: given a permutation with N elements, and the ability to apply zero or more swap and double-swap operations, can we transform it into a mountain permutation? The goal is to determine the feasibility of achieving this transformation and to explore the range of permutations attainable through these operations.

To delve deeper into the problem, we establish constraints on the size of permutations, limiting N to the range of 4 to 200,000. Additionally,

practical examples and theoretical observations are presented to illustrate the complexities and nuances of the problem. The theoretical background provides essential insights, highlighting invariants and observations crucial for understanding the impact of swap and double-swap operations on permutations.

Through these operations, we embark on a journey to unravel the possibilities of transforming permutations into captivating mountain sequences, adding a layer of elegance and complexity to the study of permutations and their structural transformations.

5.2 Mountain Permutation

Presentation of the problem

A permutation with N elements is a series $a_1, a_2, \ldots, a_N$ that contains each integer from the interval $1, 2, \ldots, N$ exactly once.

For example, a permutation with 6 elements can be represented by the series: $[3, 1, 6, 4, 5, 2]$.

A permutation is of *mountain* type if among its N elements there exists an index M such that $1 < M < N$, and the sequence formed by the first M elements is strictly increasing, while the sequence formed by the last $N \check{} M + 1$ elements is strictly decreasing. This can be expressed as:

$$a_1 < a_2 < \ldots < a_{M-1} < a_M > a_{M+1} > \ldots > a_N$$

An example of a mountain permutation is: $[1, 2, 4, 5, 6, 3]$.

In our problem, we consider a permutation with N elements indexed from 1 to N. Two types of symmetric exchange operations can be applied to the elements:

1. `swap`$(P, N - P + 1)$ – In permutation a, the elements at positions P and $N - P + 1$ are exchanged, equally spaced from the two edges of the permutation. The operation requires the conditions: $1 \leq P \leq N$ and $P \neq N - P + 1$.

2. `double_swap`$(P, Q, N-P+1, N-Q+1)$ – In permutation a, two pairs of elements are exchanged simultaneously. The elements at positions P and Q and the elements at positions $N - P + 1$ and $N - Q + 1$ will be interchanged simultaneously. The operation can be applied

if and only if all four positions are pairwise distinct. In other words, the operation requires $1 \leq P, Q \leq N$ and the set $\{P, Q, N - P + 1, N - Q + 1\}$ to satisfy the cardinality condition.

Starting from our permutation and successively applying operations of both types, where each type can be used zero or more times, a varied range of permutations can be obtained.

Question

1. Specify whether, for the given permutation, by applying any number of `swap` or `double_swap` operations, a mountain-type permutation can be obtained.

Restrictions

1. $4 \leq N \leq 200000$

Examples

Example 1. Let $N = 7$ and the series be $3, 4, 2, 7, 1, 6, 5$. The possible mountain permutations are:

1. $1, 3, 4, 7, 6, 5, 2$
2. $2, 3, 4, 7, 6, 5, 1$
3. $2, 5, 6, 7, 4, 3, 1$
4. $1, 5, 6, 7, 4, 3, 2$

Explanation for the first permutation:
- `double_swap`$(1, 5, 7, 3)$ – $[3, 4, 2, 7, 1, 6, 5] \rightarrow [1, 4, 5, 7, 3, 6, 2]$
- `double_swap`$(2, 5, 6, 3)$ – $[1, 4, 5, 7, 3, 6, 2] \rightarrow [1, 3, 6, 7, 4, 5, 2]$
- `swap`$(3, 5)$ – $[1, 3, 6, 7, 4, 5, 2] \rightarrow [1, 3, 4, 7, 6, 5, 2]$

Example 2. Let $N = 6$ and the sequence be $6, 3, 4, 5, 1, 2$. The possible mountain permutations are:

1. $1, 2, 4, 5, 6, 3$
2. $3, 6, 5, 4, 2, 1$

Example 3. Let $N = 6$ and the sequence be $1, 2, 3, 5, 4, 6$. It cannot be transformed into any mountain permutation.

Example 4. Let $N = 7$ and the sequence be $2, 3, 4, 1, 7, 6, 5$. It cannot be transformed into any mountain permutation.

Theoretical background

Observations. Firstly, to resolve the issue, it is crucial to understand the impact of `swap` and `double_swap` operations on a permutation:

- `swap` changes two elements equally spaced from the ends of the permutation.
- `double_swap` swaps two arbitrary elements and simultaneously swaps the corresponding elements equidistant from the ends of the permutation.

An observed invariant is that two elements equally spaced from the ends of the permutation, denoted as a_i and a_{n-i+1}, remain equally spaced regardless of the operations performed. These are referred to as "pair numbers."

Additionally, if the permutation has an odd length, the element $a_{\frac{n+1}{2}}$ is a fixed point and cannot change its position with either type of operation. This case is analogous to the even-length permutation, where $a_{\frac{n+1}{2}}$ is not utilized for the proof.

To visually represent even numbers, a permutation can be presented as follows:

a_1	a_2	$\dots$	a_{k-1}	a_k
a_n	a_{n-1}	$\dots$	a_{n-k+2}	a_{n-k+1}

Mountain permutation. A mountain permutation is a permutation that is strictly increasing until a certain point and then becomes strictly decreasing. We denote the position where the transition between the two monotonies occurs as *top*, where $1 < top \leq k$. If $top > k$, we can apply a `swap` transformation to all values from 1 to k. This process inverts the first line with the second.

So, without loss of generality, a mountain permutation looks like this:

a_1	$<$	a_2	$<$	$\dots$	$<$	$a_{n-top+1}$	$<$	$\dots$	$<$	a_{k-1}	$<$	a_k
												$\wedge$
a_n	$<$	a_{n-1}	$<$	$\dots$	$<$	a_{top}	$>$	$\dots$	$>$	a_{n-k+2}	$>$	a_{n-k+1}

Lexicographic minimal mountain permutation (LMMP). From all obtainable mountain permutations of a given permutation, we can obtain a unique permutation that is minimally lexicographic.

A permutation a is lexicographically smaller than another permutation b if and only if $\exists k \in \{1, 2, \dots n\}$ such that $a_i < b_i$ and $a_j = b_j,\ \forall\ i \in \{1, 2, \dots, i-1\}$.

Lemma 1. *LMMP respects the property that* $a_i < a_{n-i+1}\ \forall\ i \in \{1, 2, \dots, k\}$, *and it can be illustrated in the following way:*

a_1	$<$	a_2	$<$	$\dots$	$<$	$a_{n-top+1}$	$<$	$\dots$	$<$	a_{k-1}	$<$	a_k
$\wedge$		$\wedge$		$\wedge$		$\wedge$		$\wedge$		$\wedge$		$\wedge$
a_n	$<$	a_{n-1}	$<$	$\dots$	$<$	a_{top}	$>$	$\dots$	$>$	a_{n-k+2}	$>$	a_{n-k+1}

We want to prove that LMMP respects the property. $a_i < a_{n-i+1}\ \forall\ i \in \{1, 2, \dots, k\}$, and it looks as the above form.

Proof. We will assume that there exists an LMMP with an $i \in \{1, 2, \dots, k\}$ such that $a_i > a_{n-i+1}$ and $a_j < a_{n-j+1}\ \forall\ 1 \leq j < i$.

For $i > n - top + 1$, since the substring $a_{n-top+1}, \dots, a_{top+1}$ is sorted ascending, there cannot be a i, such that $a_i > a_{n-i+1}$ and the mountain property to be respected.

For $1 \leq i < n - top + 1$, the permutation looks like this:

a_1	$<$	a_2	$<$	$\dots$	$<$	$a_{n-top+1}$	$<$	$\dots$	$<$	a_{k-1}	$<$	a_k
$\wedge$		$\wedge$		?		?		$\wedge$		$\wedge$		$\wedge$
a_n	$<$	a_{n-1}	$<$	$\dots$	$<$	a_{top}	$>$	$\dots$	$>$	a_{n-k+2}	$>$	a_{n-k+1}

From the previous figure, we can see that $a_i > a_{n-i+1} > a_{n-i}$ and $a_{n-i+1} > a_{n-i} > a_{i-1}$, so we can reverse a_i with a_{n-i+1} using a `swap` operation. In order to preserve the mountain property and to keep the mountain property for the rest of the string as well, we will also apply `swap` on all pairs from $i + 1$ to k as follows:

a_1	$<$	a_2	$<$	$\dots$	$<$	a_{top}	$>$	$\dots$	$>$	a_{n-k+2}	$>$	a_{n-k+1}
$\wedge$		$\wedge$		?		?		$\vee$		$\vee$		$\vee$
a_n	$<$	a_{n-1}	$<$	$\dots$	$<$	$a_{n-top+1}$	$<$	$\dots$	$<$	a_{k-1}	$<$	a_k

The new permutation obtained is still a mountain permutation and is lexicographically smaller than the original permutation because $a_i > a_{n-i+1}$, so the original permutation cannot be LMMP.

Obviously, the top of the permutation can be found in the second line, forming the following shape:

a_1	$<$	a_2	$<$	$\dots$	$<$	$a_{n-top+1}$	$<$	$\dots$	$<$	a_{k-1}	$<$	a_k
												$\wedge$
a_n	$>$	a_{n-1}	$>$	$\dots$	$>$	a_{top}	$<$	$\dots$	$<$	a_{n-k+2}	$<$	a_{n-k+1}

□

Corollary 1. *Any permutation that can be reached using only* `swap` *and* `double_swap` *operations to a mountain permutation can be converted to complexity* $\mathcal{O}(n)$ *or* $\mathcal{O}(n \cdot \log_2 n)$ *in the case of LMMP.*

Using the algorithm from Lemma 1 several times, we can prove by mathematical induction that any mountain permutation can be transformed into a PMML using only `swap` operations. At each step of the algorithm, the length of the pairwise prefix $a_i > a_{n-i+1}$ increases by at least 1.

Moreover, since we know that for any possible permutation of the input data of the problem there is at least one mountain permutation that can be obtained, we can get the PMML in the following way:

- We perform `swap` operations for all pairs with $a_i > a_{n-i+1}$, where $i \in \{1, 2, \ldots, k\}$ – this guarantees us the lexicographic minimality property.
- We sort ascending using `double_swap` operations the elements from a_1 to a_k. Due to the fact that at least one mountain permutation is guaranteed to exist, the other elements will be sorted ascending from a_{n-k+1} to a_{top} and descending from a_{top} to a_n – this guarantees our mountain property.

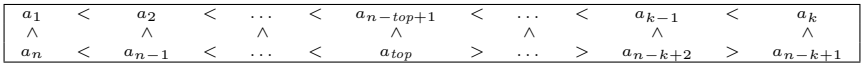
The complexity of this algorithm is given by the sorting of even numbers. Considering that the string is a permutation, the numbers are between 1 and n, and thus the complexities can be obtained:

- $\mathcal{O}(n^2)$ if we use suboptimal sorting algorithms such as Bubble Sort.
- $\mathcal{O}(n \cdot \log(n))$ if optimal comparison sorting algorithms such as Merge Sort are used.
- $\mathcal{O}(n)$ if Count Sort is used.

Calculating the number of mountain permutations. Thanks to Corollary 1, we know that we can obtain from the original PMML permutation the number of mountain permutations. At the same time, as shown in Lemma 1, we demonstrated that from any mountain permutation, an LMMP can be reached only through `swap` operations. Moreover, the `swap` operation is its own inverse: if we apply the `swap` 2 times on the same column, we return to the initial state. The converse is also true: we can get from LMMP to any obtainable mountain permutation using only `swap` operations.

Another observation is that two `swap` operations on two distinct positions $i \neq j$ are independent. Thus, the number of mountain permutations that can be obtained will be 2^C, where C represents the maximum number

of distinct `swap` operations that we can apply to LMMP such that the result remains a mountain permutation.



The previous figure shows the way to apply `\swap` on a certain position $1 \leq i \leq k$, either $i = 1$ or $a_i > a_{n-i}$.

The complexity of the algorithm is given by the complexity of bringing the initial permutation to LMMP, and it is shown in that it can be implemented in $\mathcal{O}(n)$ or $\mathcal{O}(n \cdot \log_2 n)$. This solution gets 100 points.

Solution. After understanding the theoretical background of the problem, we can study its solution.

Reading in the data. We read in the size of the array, denoted by n. We read in the first half of the array normally. In the second half of the array, at the time of reading, we arrange the pairs of values so that the smaller value is on the first line if the current element is not the middle one.

```
    for (i=1; i<=n/2; i++)
     fin>>a[i];
for (i=n/2+1; i<=n; i++)
{
     fin>>x;
     if ((n-i+1)!=i)
          if (a[n-i+1]>x)
          {
               a[i]=a[n-i+1];
               a[n-i+1]=x;
          }
          else
               a[i]=x;
     else
          a[i]=x;
}
```

Sorting the array. We sort the array using a count sort. The reason for choosing this type of sorting algorithm is that its time complexity is $O(n)$. Simultaneously, we also search for the index of the maximum element (pozmax), which will be the top of the mountain.

```
    for (i=1; i<=n/2; i++)
     b[a[i]]++;
for (i=2; i<=n; i++)
     b[i]+=b[i-1];
```

```
for (i=n/2; i>=1; i--)
{
    int poz=b[a[i]];
    c[poz]=a[i];
    c[n-poz+1]=a[n-i+1];
}
if (n%2==1)
    c[n/2+1]=a[n/2+1];
int pozmax=-1;
for (i=n/2+1; i<=n; i++)
    if (c[i]==n)
        pozmax=i;
```

Deciding whether it is a mountain permutation or not. We distinguish three cases when it is not a mountain permutation. If the array does not satisfy one of the three cases, we can confirm that it is a mountain permutation.

Cases:

1. when the top of the mountain is the last element of the array:

```
if (pozmax==n)
{
    fout<<"NO";
    return 0;
}
```

2. when to the left of the top element we can find two decreasing elements:

```
for (i=n/2+1; i<=pozmax; i++)
    if (c[i-1]>c[i])
    {
        fout<<"NO";
        return 0;
    }
```

3. when to the right of the top element I can find two growing elements:

```
for (i=pozmax+1; i<=n; i++)
    if (c[i-1]<c[i])
    {
        fout<<"NO";
        return 0;
    }
```

Counting the number of mountain permutations. If we also want to count the number of solutions, we can do it as follows:

```
 long long nr=2;
for (i=1; i<=n/2; i++)
    if (c[i]<c[i+1] and c[n-i+1]<c[i+1] and c[i]<c[n-i]
        and c[n-i+1]<c[n-i] and i+1<n-i)
        nr=(nr*2)%modulo;
```

5.3 Mountain String

Presentation of the problem

A mountain string has the property that among its n elements there is an index M inside the string such that $1 < M < N$, for which the sequence formed by the first M elements $a_1, a_2, ..., a_M$ is increasing and non-constant, and the sequence formed by the last $N - M + 1$ elements: $a_M, a_{M+1}, ..., a_N$ is decreasing and non-constant.

For example, the following string with 8 elements is a mountain string: $[1, 1, 1, 2, 2, 2, 2, 1]$.

In our problem, we will consider that we have a string with N elements indexed from 1 to N, and the elements are $a_1, a_2, ..., a_N$, which are positive integers.

On the elements of the string, we can apply two types of symmetric exchange operations:

1. `\swap` $(P, N - P + 1)$ – in string a, the elements on positions P and $N-P+1$ are equally spaced from the two edges of the string and can be exchanged in the following way: $(P \neq N - P + 1, 1 \leq P \leq N)$;

2. `\dswap` $(P, Q, N - P + 1, N - Q + 1)$ – in string a, two pairs of elements symmetrically located with respect to the two extremities of the string can be simultaneously exchanged. So, the elements on the P and Q positions, respectively the elements on the $N - P + 1$ and $N - Q + 1$ positions can be interchanged simultaneously. The operation can only be applied if all four positions are pairwise distinct $(P \neq Q, P \neq N-P+1, Q \neq N-Q+1, Q \neq N-P+1, 1 \leq P \leq N, 1 \leq Q \leq N)$.

Question: We are interested in how many distinct mountain strings can be obtained by applying any of the operations defined above.

Restrictions: n must satisfy the following inequalities: $3 \leq n \leq 2005$, and the values of each element can be found in $[0, 2 \times 10^9]$.

Examples

Example 1. For $N = 7$ and $a = \{3, 6, 2, 10, 1, 9, 8\}$, we can obtain 4 distinct mountain strings:

The first mountain string can be obtained from the sequence $\{3, 6, 2, 10, 1, 9, 8\}$ with the following operations:

1. `\dswap` $(1, 5, 7, 3)$: we swap the following elements: $a[1] = 3$ with $a[5] = 1$ and $a[7] = 8$ with $a[3] = 2$. The highlighted elements of $\{\underline{3}, 6, \underline{2}, 10, \underline{1}, 9, \underline{8}\}$ are swapped, and the mountain string of $\{1, 6, 8, 10, 3, 9, 2\}$ is obtained;
2. `\dswap` $(2, 5, 6, 3)$: the highlighted elements of $\{1, \underline{6}, \underline{8}, 10, \underline{3}, \underline{9}, 2\}$ are swapped, and the mountain string of $\{1, 3, 9, 10, 6, 8, 2\}$ is obtained;
3. `\swap` $(3, 5)$: the highlighted elements of $\{1, 3, \underline{9}, 10, \underline{6}, 8, 2\}$ are swapped, and the mountain string of $\{1, 3, 6, 10, 9, 8, 2\}$ is obtained, which is the first mountain string.

The second mountain string obtained from $\{3, 6, 2, 10, 1, 9, 8\}$ is:

1. `\dswap` $(1, 3, 7, 5)$: the string of $\{2, 6, 3, 10, 8, 9, 1\}$ is obtained;
2. `\dswap` $(2, 3, 6, 5)$: the mountain string of $\{2, 3, 6, 10, 9, 8, 1\}$ is obtained, which is the second solution.

The third mountain string obtained from $\{3, 6, 2, 10, 1, 9, 8\}$ is:

1. `\dswap` $(2, 5, 6, 3)$: the string of $\{3, 1, 9, 10, 6, 2, 8\}$ is obtained;
2. `\dswap` $(2, 1, 6, 7)$: the string of $\{1, 3, 9, 10, 6, 8, 2\}$ is obtained;
3. `\swap` $(2, 6)$ to get the mountain string of $\{1, 8, 9, 10, 6, 3, 2\}$, which is the third mountain string.

The fourth mountain string obtained from $\{3, 6, 2, 10, 1, 9, 8\}$ is:

1. `double_swap` $(7, 2, 1, 6)$: the string $\{9, 8, 2, 10, 1, 3, 6\}$ is obtained;
2. `double_swap` $(5, 7, 3, 1)$ to get the mountain string $\{2, 8, 9, 10, 6, 3, 1\}$, which is the fourth solution.

Example 2. For $N = 8$ and $a = \{2, 1, 2, 1, 1, 1, 2, 2\}$ with the help of `\swap` and `\dswap` operations, we get 2 distinct mountain strings: $\{1, 1, 1, 2, 2, 2, 2, 1\}$ and $\{1, 2, 2, 2, 2, 1, 1, 1\}$ respectively.

Theoretical background of the problem. The problem of Mountain String is a more complex version of the Mountain Permutation problem. We accept that the theoretical background of Mountain Permutation is also valid for the properties of Mountain String. In the case where the elements of the array are distinct, the Mountain String becomes a Mountain Permutation problem. For instance, $[1, 2, 4, 5, 6, 3]$ is a Mountain Permutation but is also a Mountain String. However, the example $[1, 1, 1, 2, 2, 2, 2, 1]$ is only a Mountain String, not a Mountain Permutation because the elements of the array are not distinct.

Lemma 2. *A mountain string respects the property that* $a_i \leq a_{n-i+1}$ *for all* $i \in \{1, 2, \ldots, k\}$ *and can be illustrated as follows:*

$$\begin{array}{ccccccccccccc}
a_1 & \leq & a_2 & \leq & \ldots & \leq & a_{n-top+1} & \leq & \ldots & \leq & a_{k-1} & \leq & a_k \\
\wedge| & & \wedge| & & \wedge| & & \wedge| & & \wedge| & & \wedge| & & \wedge| \\
a_n & \leq & a_{n-1} & \leq & \ldots & \leq & a_{top} & \geq & \ldots & \geq & a_{n-k+2} & \geq & a_{n-k+1}
\end{array}$$

Solution. After understanding the general version of the Mountain problem, let us see the solution to the second version of the problem.

Reading in the array. Read in the size of the array, denoted by n. The first part of the array is read in normally.

To gain the canonical form, read the second half of the array in the following way:

- Read in the following element of the array.
- If the current element is smaller than its pair ($a[n - i + 1]$), swap them.
- Otherwise, do not use the swap operation.

From an optimization point of view, also calculate the maximum value of the array simultaneously.

```
MAXIM=-1;
/// the first part of array normally
for(int i=1; i<=n/2; ++i)
{
    fin>>a[i];
    MAXIM=max(a[i],MAXIM);   /// max. number of array
}
```

```
/// we gain the canonical form
for(int i=n/2+1; i<=n; i++)
{
     fin>>x;
     MAXIM=max(x,MAXIM);
     if (x<a[n+1-i])
     {
          a[i]=a[n+1-i];
          a[n+1-i]=x;
     }
     else
          a[i]=x;
}
```

Sorting the array. Sort the elements of the array based on the first half of the array. If the elements are equal, let the second half of the array be in descending order.

- In the case of $a[i] > a[j]$ or the two elements are equal and $a[n - i + 1] > a[n - j + 1]$, where $i \in \{1, 2, ..., n/2 - 1\}$ and $j \in \{i + 1, i + 2, ..., n/2\}$.
- Use the `\swap` operation simultaneously for the pairs of elements: $a[i] \leftrightarrow a[j]$ and $a[n - i + 1] \leftrightarrow a[n - j + 1]$.

```
for (int i=1; i<n/2; i++)
     for (int j=i+1; j<=n/2; j++)
          if (a[i]>a[j] or (a[i]==a[j] and a[n-i+1]>a[n-j+1]))
          {
               swap(a[i],a[j]);
               swap(a[n-i+1],a[n-j+1]);
          }
```

Getting mountain form. To gain the mountain form of the canonical form, the elements in front of the maximum position from the second half of the array will be transformed to descending order if the elements corresponding to them have equal values in the first half of the array.

- Let $left$ be a variable and initialize it with 1.
- Navigate through the first half of the array with an i variable.
- In the case of $a[i]$ is not equal to $a[left]$:
  - Initialize a $right$ variable with $i - 1$.
  - If the values of $left$ and $right$ are not equal and we have not found the maximum value, use the `\swap` operation for all elements in the $[n - right + 1, n - left + 1]$ interval.

  - If we have found the maximum element, where the maximum element should be equal to $a[n - right + 1]$, then save the index of the maximum element.
- Update the $left$ variable with the value of i and repeat the previous steps.
- After the cycle, check if $left < n/2$ and $a[left]$ is equal to $a[n/2]$.
- In the case where we have already found the maximum value, use the `\swap` operation for all elements in the $[n-right+1, n-left+1]$ interval.
- If we have not found the maximum value, check if the middle value $(n - n/2 + 1)$ is equal to the maximum value, and then mark the position of the maximum value with the index of the middle value.

```
int left=1;
bool foundmax=false;
for (int i=1; i<=n/2; i++)
{
    if(a[i]!=a[left])
    {
        int right=i-1;
        if (left!=right)
        {
            if (foundmax)
                revers(a,n-right+1,n-left+1);
            else
            {
                if(a[n-right+1]==MAXIM)
                {
                    foundmax=true;
                    POZMAX=n-right+1;
                }
            }
        }
        left=i;
    }
}
if (left<n/2 and a[left]==a[n/2])
{
    if (foundmax)
        revers(a,n-n/2+1,n-left+1);
    else
    {
        if (a[n-n/2+1]==MAXIM)
            POZMAX=n-n/2+1;
    }
```

}

Finding the number of mountain series. Calculate the number of mountain series until the top of the mountain because the rest of the values do not influence the solution.

Duplicate the number of solutions if we reverse the "slices of the mountain." The conditions for the slice of the mountain are the following ones:

- $a[i] \geq a[n-i+2]$
- $a[i] \neq a[n-i+1]$

The above conditions are equal to the following figure:

$$\begin{array}{ccc} a_{i-1} & \leq & a_i \\ \wedge| & & \wedge| \\ a_{n-i+2} & \leq & a_{n-i+1} \end{array}$$

Distinguish different cases regarding calculating solutions:

1. If n is odd:
   - The element at $(n+1)/2$ index cannot be moved to another position, which means that we can have any solution; this element remains fixed.
2. If the top of the mountain cannot be found in the middle:
   - If the pairs' value of neighbors of the top element has an equal element, then they can be transferred to the other side of the top element, and the less than relation will be changed to the greater than relation.
   - Let p be the last index of the element from the array that is equal to the maximum value.
   - Distinguish different cases:
     a) If the first element is fixed and remains on the same side, then the number of solutions is multiplicative in each iteration:

$$\text{nr} = (\text{nr} \times (\text{counter} + 1)) \mod \text{modulo},$$

where we increase the counter until $p < n$ and

$$a[p] = a[\text{POZMAX}]$$
$$a[n-p+1] = a[n-\text{POZMAX}+1]$$

and

$$a[p] \geq a[\text{POZMAX} - 1],$$

where POZMAX is equal to the initial value of p.

b) If the first element can be transferred to the other side, then the number of solutions can be calculated as follows:
   - Calculate in each iteration the following equation:

     $$\text{solution} = (\text{solution} + \text{multiplication} \times \text{counter}) \mod \text{modulo},$$

     which follows the calculation of the multiplication variable:

     $$\text{multiplication} = (\text{multiplication} \times (\text{counter} + 1)) \mod \text{modulo}$$

   - After the cycle, calculate the number of solutions as follows:

     $$\text{nr} = (\text{nr} \times \text{solution}) \mod \text{modulo},$$

     where we increase the counter until $p \leq n$ and $a[p] = a[\text{POZMAX}]$, where POZMAX is equal to the initial value of p.
   - So, in other words, if that element can be moved to the outside of the peak, practically every element can be moved to the other side of the peak. Among these solutions, only one does not meet the condition, which is when all the elements have been moved to the outside, and the sequence forms an increasing sequence, thereby losing the property of being a peak.

```
long long nr;
if (a[1]!=a[n])
    nr=2;
else
    nr=1;

for (int i=2; i<=n/2; i++)
    if (a[i]>=a[n-i+2] and a[i]!=a[n-i+1])
        nr=(nr*2)%modulo;
if (POZMAX==0)
    fout<<nr<<endl;
else
{
```

```
int HIGH_LIMIT=a[POZMAX-1];
int p=POZMAX;
int EQUAL_VALUE=a[n-POZMAX+1];
while (a[p]==a[POZMAX])
    p++;
if (a[1]!=EQUAL_VALUE or a[n]<HIGH_LIMIT)
{
    while (p<n and a[n-p+1]==EQUAL_VALUE
    and a[p]>=HIGH_LIMIT)
    {
        int counter=0;
        int VALUE=a[p];
        while(p<n and a[p]==VALUE and a[n-p+1]==EQUAL_VALUE
                and a[p]>=HIGH_LIMIT)
        {
            counter++;
            p++;
        }
        nr =(nr*(counter+1)) % modulo;
            }
        }
        else
        {
            long long multiplication=1;
            long long solution=0;
            while (p<=n)
            {
                int counter=0;
                int VALUE=a[p];
                while(p<=n and a[p]==VALUE)
                {
                    counter++;
                    p++;
                }
                solution = (solution + multiplication*counter)
                % modulo;
                multiplication=(multiplication*(counter+1))
                % modulo;
            }
            nr=(nr*solution) % modulo;
        }
        fout<<nr<<endl;
    }
```

CHAPTER 6

NUMBER THEORY AND COMBINATORICS

Author: Gyöngyvér Márton

Number theory and combinatorics deal with the study of the properties of integers. It can be said that number theory was one of the most important fields of science at the time of the birth of mathematics, and it also has a prominent role in today's computer science. Together with combinatorics, it forms the basis of many disciplines such as cryptography, code theory, and so on. In addition to all this, both of these fields have always been enjoyable!

That is why it is important for us to include at least one problem each year in ECN competitions that requires knowledge from number theory and combinatorics. In this chapter, we have collected some problems proposed in ECN competitions whose solutions require the use of algorithms from these fields of mathematics and computer science. We would like to emphasize that having a basic understanding of number theory and combinatorics can make it much easier to solve the proposed problems. In the absence of this knowledge, they can be challenging.

Firstly, we present some basic algorithms from number theory, providing them with shorter or longer explanations.

Binary GCD

The binary GCD algorithm computes the greatest common divisor of two positive integers, a and b. Compared with the Euclidean algorithm, this variant, instead of division and multiplication, uses only incrementation, subtraction, and division and multiplication by powers of 2. The `binGCD` function calculates this value. The algorithm's running time is $O(\texttt{l\^{}2})$, where `l = max(len(a), len(b))`, [2, 11].

```
typedef long long uInteger;
#define swap(x,y) { x = x ^ y; y = x ^ y; x = x ^ y; }

uInteger binGCD(uInteger a, uInteger b) {
```

```
    uInteger k = 0;
    while (!(a & 1) && !(b & 1)) {
        a = a >> 1;
        b = b >> 1;
        k++;
    }
    while (1) {
        while (!(a & 1)) a = a >> 1;
        while (!(b & 1)) b = b >> 1;
        if (b < a) swap(a, b);
        b = b - a;
        if (b == 0) return a << k;
    }
}
```

Binary extended Euclidean algorithm

From number theory, it is well known that there exist integers x_0 and y_0 such that $a \cdot x_0 + b \cdot y_0 = d$, where a and b are integers and d is the greatest common divisor of a and b. The binary GCD algorithm presented above can be extended to compute d, x_0, and y_0 on input a, b such that $a \cdot x_0 + b \cdot y_0 = d$. The binary extended Euclidean algorithm allows us to efficiently calculate x_0 and y_0, in addition to determining d. The `binExtEuclid` function calculates these values. The algorithm running time is O(`l^2`), where `l = max(len(a), len(b))`, [10].

```
uInteger* binExtEuclid(uInteger a, uInteger b) {
    uInteger k = 0;
    while (!(a & 1) && !(b & 1)) {
        a = a >> 1;
        b = b >> 1;
        k++;
    }
    uInteger aT = a, bT = b;
    uInteger x0 = 1, x1 = 0, y0 = 0, y1 = 1;
    while (1) {
        while (!(a & 1)) {
            a = a >> 1;
            if (!(x0 & 1)  && !(y0 &1) )
```

```
                x0 = x0 >> 1, y0 = y0 >> 1;
            else
                x0 = (x0 + bT) >> 1, y0 = (y0 - aT) >> 1;
        }
        while (!(b & 1)) {
            b = b >> 1;
            if (!(x1 & 1) && !(y1 & 1))
                x1 = x1 >> 1, y1 = y1 >> 1;
            else
                x1 = (x1 + bT) >> 1, y1 = (y1 - aT) >> 1;
        }
        if (b < a) {
            swap(a, b);
            swap(x0, x1);
            swap(y0, y1);
        }
        b = b - a;
        x1 = x1 - x0, y1 = y1 - y0;
        if (b == 0) {
            uInteger* res = new uInteger[3];
            res[0] = a << k, res[1] = x0, res[2] = y0;
            return res;
        }
    }
}
```

Binary multiplicative inverse algorithm

The binary extended Euclidean algorithm can be used to calculate the multiplicative inverse of a number modulo m. The solution of the congruence $a \cdot x \equiv 1 \pmod{m}$ is called the multiplicative inverse or simply the inverse of a modulo m, where a and m are integers, and $m > 1$ and $gcd(a, m) = 1$. The multiplicative inverse exists if $gcd(a, m) = 1$. It is easy to show that the congruence $a \cdot x \equiv 1 \pmod{m}$ is equivalent with the equation $a \cdot x - m \cdot y = 1$, so if the multiplicative inverse exists, it can be determined with the extended Euclidean algorithm [1, 10]. The `modInv` function calculates this value.

```
uInteger modInv(uInteger a, uInteger b) {
    uInteger k = 0, tempB = b;
```

```
    while (!(a & 1) && !(b & 1)) {
        a = a >> 1;
        b = b >> 1;
        k++;
    }
    uInteger aT = a, bT = b;
    uInteger x0 = 1, x1 = 0;
    while (1) {
        while (!(a & 1)) {
            a = a >> 1;
            if (!(x0 & 1))
                x0 = x0 >> 1;
            else
                x0 = (x0 + bT) >> 1;
        }
        while (!(b & 1)) {
            b = b >> 1;
            if (!(x1 & 1))
                x1 = x1 >> 1;
            else
                x1 = (x1 + bT) >> 1;
        }
        if (b < a) {
            swap(a, b); swap(x0, x1);
        }
        b = b - a;
        x1 = x1 - x0;
        if (b == 0) {
            if ((a << k) != 1) return -1;
            if (x0 < 0) x0 += tempB;
            return x0;
        }
    }
}
```

Modular exponentiation

The modular exponentiation, or the repeated-squaring algorithm computes $a^x \pmod m$. During the calculations, the binary form of the exponent will

be determined, and the number of bits in the binary form of the exponent will be taken into account. The `modPow` function calculates this value. The algorithm running time is $O(\texttt{len(x)} \cdot \texttt{len(m)}^2)$ [8, 10].

```
uInteger modPow(uInteger a, uInteger x, uInteger m) {
    uInteger res = 1;
    a = a % m;
    if (a == 0) return 0;
    while (x > 0)
    {
        if (x & 1)
            res = (res * a) % m;
        a = (a * a) % m;
        x = x >> 1;
    }
    return res;
}
```

Conversion between number systems

In our everyday life, we use the decimal number system to represent various quantities, and this can be explained mostly by the fact that people have 10 fingers. But instead of ten, we can always choose another value, i.e. another number system can be adopted. There is a clear conversion process between the different number systems based on natural numbers. The function `nrToBaseB` converts a decimal number to base b, and the function `nrFromBaseB` converts a number from base b to decimal number [2, 4, 6].

```
uInteger* nrToBaseB(int &i, uInteger nr, uInteger b) {
    uInteger* Temp = new uInteger[64];
    i = 0;
    while (nr > 0) {
        Temp[i++] = nr % b;
        nr = nr / b;
    }
    Temp[i] = '\0';
    uInteger *L = new uInteger[i+1];
    for (int j = 0; j < i; j++)
        L[j] = Temp[i - j - 1];
```

```
    delete []Temp;
    return L;
}

uInteger nrFromBaseB(uInteger *L, int n, uInteger b) {
    uInteger nr = 0;
    for (int i = 0; i < n; ++i)
        nr = nr * b + L[i];
    return nr;
}
```

6.1 Farey Series – 2006, Round 1, Problem G

Farey's series of order n is defined as the set of fractions $\frac{h}{k}$ in ascending order, where h and k are integers, $0 \leq h \leq k \leq n$, and $gcd(h,k) = 1$. We write $gcd(h,k)$ for the greatest common divisor of h and k. The series contains 0 and 1 in the form of $\frac{0}{1}$ and $\frac{1}{1}$ respectively [8].

Write a program that lists the Farey series of order n.

The input contains a number n, where $n < 1000$, which indicates the order of the series to be generated.

The output must be printed to the standard output and should consist of the elements of the Farey series of order n, containing five elements in each row and a tab between each element. Put / between the numerator and denominator of an element.

Hint: We know that the first two elements of the Farey series of order n are always $\frac{0}{1}, \frac{1}{n}$. The other elements of the series are defined in such a way that a new element is always calculated using the last two elements of the series. If the last two elements of the series are $\frac{a_1}{b_1}$ and $\frac{a_2}{b_2}$, then the next $\frac{a_3}{b_3}$ can be calculated by the following formula:

$$\frac{a_3}{b_3} = \frac{a_2 \cdot k - a_1}{b_2 \cdot k - b_1}, \quad \text{where} \quad k = \left\lfloor \frac{n + b_1}{b_2} \right\rfloor$$

Table 6.1. Farey series – input/output

Input	Output				
10	0/1	1/10	1/9	1/8	1/7
	1/6	1/5	2/9	1/4	2/7
	3/10	1/3	3/8	2/5	3/7
	4/9	1/2	5/9	4/7	3/5
	5/8	2/3	7/10	5/7	3/4
	7/9	4/5	5/6	6/7	7/8
	8/9	9/10	1/1		

6.2 Arrays – 2007, Round 1, Problem G

In a two-dimensional array, each row and column contains exactly two blocked positions that are marked with 1. The other free positions are marked with 0. The array allows the following transformations: swapping an optional row with another row and swapping an optional column with another column. Starting from an initial configuration, a limited number of optional transformational rules lead to another configuration. The arrays that can be formed by these limited permitted transformational rules form a set. However, there are array configurations that cannot be transformed into each other, forming different sets [7].

Example: In *Table 6.2*, array B can transform from array A if we produce the same row and column numbering for B as in array A. They are in the same set. Array D can transform from array C if we produce the same row and column numbering for D as in array C. They are in the same set. Neither array A nor B can transform from arrays C or D, and vice versa.

Table 6.2. Arrays, examples

Array A

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	0	1	0
c	0	0	1	1	0
d	0	1	0	0	1
e	1	0	1	0	0

Array B

	a	b	e	c	d
a	0	1	1	0	0
b	1	0	0	0	1
c	0	0	0	1	1
e	1	0	0	1	0
d	0	1	1	0	0

Array C

	a	b	c	d	e
a	0	0	1	1	0
b	0	1	0	0	1
c	0	1	0	1	0
d	1	0	0	0	1
e	1	0	1	0	0

Array D

	b	e	c	a	d
a	0	0	1	0	1
e	0	0	1	1	0
b	1	1	0	0	0
d	0	1	0	1	0
c	1	0	0	0	1

Write a program that determines for an $n \times n$ array how many sets exist with such properties.

The input file contains several data. Each line stores a number that defines the array dimension.

The output must be printed to the standard output. For each input number, the number of sets with specified properties must be printed.

Table 6.3. Arrays – input/output

Input	Output
4	2
5	2
8	7
10	12

6.3 Sums of Consecutive Integers – 2007, Round 2, Problem G

Write a program that determines the number of natural numbers between 1 and m, where $m <= 10000$, which can be written as the sum of two or more consecutive positive integers [5].

The input file consists of several test cases, each on a separate line containing m.

The output must be printed to the standard output, with each found number printed separately on each row.

Table 6.4. Sums of consecutive integers – input/output

Input	Output
3	1
4	1
5	2

6.4 Decomposition of Factorial – 2007, Round 3, Problem E

Write a program that, for given pairs of natural number $2 \leq n \leq 4\ 294\ 967\ 295 \leq n$ and prime number $2 \leq p \leq n$, determines the power of p in the prime decomposition of the factorial $n! = 1 \cdot 2 \cdot 3 \ldots n$ [8, 9].

The input file consists of several test cases, each on a separate line containing two numbers. The first number is for n and the second is for p.

The output must be printed to the standard output, with the found power printed separately on each row.

Table 6.5. Sums of consecutive integers - input/output

Input		Output
10	3	4
1256	5	313
321586	2801	114
4294967295	7	715827878

6.5 Maximum Border – 2007, Round 3, Problem G

For any string x containing n letters, we use the notation $x[1 : n]$. For any integer $1 \leq j < n$, we say that $x[1 : j]$ is the prefix of x, and for any $1 < j \leq n$, we say that $x[j : n]$ is the suffix of x. A border of x is any prefix of x that equals the suffix of x [7].

Write a program that computes the length of the longest border for each prefix of string x. There is no size limit for the value of n.

The input file will contain only one string, where the characters of the string can be spread across multiple lines without any empty lines in between.

The output must be printed to the standard output, where each line will contain 40 numbers, except the last line, which can contain fewer numbers. Between the numbers, you will put one space. Each number in the output corresponds to the length of the longest border for the corresponding prefix of the string.

Table 6.6. Maximum border – input/output

Input	Output	Explanation
abaababaab aabab	0 0 1 1 2 3 2 3 4 5 6 4 5 6	0 a 0 ab 1 aba 1 abaa 2 abaab 3 abaaba 2 abaabab 3 abaababa 4 abaababaa 5 abaababaab 6 abaababaaba 4 abaababaabaa 5 abaababaabaab 6 abaababaabaaba

Hint: A good presentation can be found at the following link: *https://web.stanford.edu/class/cs97si/suffix-array.pdf*.

6.6 Regular Numbers – 2008, Problem C

We define regular numbers as the number 1 and those whose only prime factors are 2 and 3. The sequence 1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48 represents the first 15 regular numbers in ascending order [8].

Write a program to find and print the n^{th} regular number, where $1 \leq n \leq 1000$.

The input file contains several lines. The first line contains a number that indicates the number of cases, and the subsequent lines contain the indices of the regular numbers that must be calculated.

The output must be printed to the standard output. For each input index, the corresponding regular number must be printed.

Table 6.7. Regular numbers – input/output

Input	Output
4	
1	1
5	6
10	18
1000	36028797018963968

Hint: Every regular number can be written as $2^i \cdot 3^j$. Thus, you should iterate through these exactly once, but note that an ordered sequence of these numbers must be constructed.

6.7 Divisibility – 2010, Problem H

We define the base b expansion of a given number N as: $(a_0, a_1, \ldots a_k)_b$, where $N = b^0 \cdot a_0 + b^1 \cdot a_1 + \cdots + b^k \cdot a_k$. We assume that $2 < b \leq 500$, and $0 \leq k < 100$ [4, 8].

Write a program to check the divisibility of N by factors of $b - 1$.

The input file contains several test cases. Each test case consists of three lines: the first line stores a number that indicates the base b, the second line stores the length of the number's base expansion, and the third line stores the base b expansion of the given N number. After a blank line, more test

cases appear. For each test case, check the divisibility of N by factors of $b-1$.

The output must be printed to the standard output and must consist of corresponding divisors in decreasing order, with each test case's divisors printed on a separate line. If the divisibility test fails, you should print -1 on the corresponding line.

Table 6.8. Divisibility – input/output

Input	Output
256	15 5 3
15	-1
7 15 122 250 8 210 126 7 87 52 0 123 67 56	17
10	
3	
4	
2 0 1 0	
35	
14	
12 4 19 16 0 6 33 7 3 14 15 6 5 13	

Hint: We know that a given number N is divisible by a factor of $b-1$ if and only if the sum of the digits of the number's base expansion is divisible by the corresponding factor.

Example: $(3, 4, 0, 9, 2)_{10}$ is divisible by 9 and 3 if and only if 3+4+0+9+2 is divisible by 9 and 3. So the answer is $9, 3$.

6.8 Primitive Roots – 2011, Problem H

Let a and p be relatively prime positive integers, where p is a prime number. The least positive integer t such that $a^t = 1 \pmod{p}$ is called the order of a modulo p, where $x \pmod{y}$ denotes the remainder in dividing x by y.

If the order of a is $p-1$, then a is said to be primitive root modulo p. We restrict the value of p so that p must have the form $2 \cdot q + 1$, where q is also a prime number [10, 11].

Write a program that, for a given pair of numbers in the role of a and p defined above, determines whether a is a primitive root modulo p or not. Note that p is not greater than $2^{31} - 1$.

The input file contains several test cases. A test case consists of a line, which stores the numbers a and p with a space between the numbers. For each test case, check whether a is a primitive root modulo p or not. Write on the output YES if a is primitive root modulo p or NO if a is not primitive root modulo p.

The output must be printed to the standard output and must consist of the corresponding YES or NO answers in different lines for each test case.

Table 6.9. Primitive roots – input/output

Input	Output
2 11	YES
9 11	NO
20 1487	YES
34 725642747	YES
4725 375081503	NO
17928 375081503	YES

Hint: We know that $p-1 = 2 \cdot q$, and the order of a is a divisor of $p-1$, so it is sufficient to test the following:

$$a^2 = 1 \pmod{p}$$
$$a^q = 1 \pmod{p}.$$

If either of these is true, then a is not a primitive root $\pmod{p}$.

6.9 Simple Encryption – 2011, Problem I

We define the following encryption rule on strings: for each element of the string, add a number x to its ASCII code, and then determine the remainder of the obtained sum when divided by 256. The decryption rule

is the inverse of the encryption rule: use subtraction (the additive inverse) instead of addition [6].

Write a program that for the encrypted string:

```
[ixqmvjqi(MKV (:  899
```

determines the corresponding decrypted string.

There is no input file, and for each possible value of x, perform the decryption rule. Write to the output only the text that contains the "ECN" string, along with the corresponding value of x. First, print the value of x, and then put a space followed by the decrypted string.

6.10 Tenders's Invitations – 2012, Problem G

A Tender Office has been posting invitations for k years. The total value of available funds in each year was x. Your task is to determine the value of x, knowing in each year the number of successful applicants and the amount remaining after the distribution of funds. It is known that the value of x does not exceed 10^{15} [8, 9].

The input file contains several test cases. The first line indicates the number of test cases. Each test case consists of three lines, with the first line storing k, the second storing the remaining amount, and the third storing the numbers of successful applicants for each year.

Write a program that determines the value of x for each test case.

If there is no solution, print NO ANSWER. Otherwise, x is printed on the standard output. The results of each test case are written on different lines.

Hint: We can note the successful applicants with $a_1, a_2, \ldots a_k$ and the amount of remainders with $m_1, m_2, \ldots m_k$. So, the task is to resolve the following system of equations:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_r \pmod{m_r} \end{aligned}$$

The solution of the above system can be determined following the next steps:

Table 6.10. Invitations for tenders – input/output

Input	Output
3 3 3 10 0 17 16 15 4 54 50 40 12 75 61 125 73 6 17 15 3 11 3 4 19 17 5 29 25 31	3930 NO ANSWER 2256153

– First, we determine the solution of the first two equations:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \end{aligned}$$

  – we will use the extended Euclidean algorithm with input m_1, m_2 to find g, k_1, k_2 such that $g = k_1 \cdot m_1 + k_2 \cdot m_2$;
  – if $a_1 - a_2$ is not divisible by g, the system does not have any solution;
  – otherwise, the solution is: $a = ((a_1 \cdot m_2 \cdot k_2)/g + (a_2 \cdot m_1 \cdot k_1)/g) \pmod{lcm}$, where lcm stands for the least common multiple of m_1 and m_2;
– The next step is to resolve, in the same manner as above, the following two equations:

$$\begin{aligned} x &\equiv a \pmod{lcm} \\ x &\equiv a_3 \pmod{m_3}. \end{aligned}$$

– We continue to resolve in every step two equations formed by the new solution and the next equation in the initial system.
– If a solution exists, the solution of the initial system will be the last $a \pmod{lcm}$ solution.

6.11 Decryptions – 2013, Problem I

A very simple but insecure encryption method consists of multiplying each 3-byte block of a message by an invertible 3×3 key matrix and determining the remainder of division by 256 for each element. This means that you should work over $\mathbb{Z}_{256}$. Formally, the encryption of a 3-byte message is as follows:

$$Enc_K(M) : C = K \cdot M, \text{ where}$$
$$M = (m_1, m_2, m_3),\ C = (c_1, c_2, c_3), K = (k_{i,j}) \text{ is a } 3 \times 3 \text{ matrix and}$$

$$(c_1, c_2, c_3) = \begin{pmatrix} k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix} \cdot \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} \pmod{256}$$

To decrypt the ciphertext, each 3-byte block is multiplied by the inverse matrix of the key matrix. The encryption and decryption processes cannot be done if the inverse matrix does not exist [6].

Write a program that, for several encrypted texts, determines the original messages.

The input file contains several test cases. The first line shows the number of test cases followed by a blank line. There will also be a blank line between each two consecutive cases, where each case consists of two lines:

- the first line contains a 3×3 key matrix,
- the second line contains the encrypted bytes (You should notice that the number of bytes in a line is a multiple of 3.)

For each test case, try to decrypt the corresponding byte sequence. If decryption is not possible, you should print "cannot decrypt" to the standard output; otherwise, print the message obtained from decryption. The result of each decryption should be put on a separate line.

Hint: A K matrix has an inverse if and only if $gcd(detK, 256) = 1$. In that case, $K^{-1} = (detK)^{-1} \cdot (adjK)$, where K^{-1} denotes the inverse matrix, $(detK)^{-1}$ the inverse determinant, and $adjK$ the adjoint matrix.

If $K = (k_{i,j})$ is a 3×3 matrix, then

$$\begin{aligned} detK = \ & k_{1,1} \cdot k_{2,2} \cdot k_{3,3} + k_{1,2} \cdot k_{2,3} \cdot k_{3,1} + k_{1,3} \cdot k_{2,1} \cdot k_{3,2} - \\ & k_{1,1} \cdot k_{2,3} \cdot k_{3,2} - k_{1,2} \cdot k_{2,1} \cdot k_{3,3} - k_{1,3} \cdot k_{2,2} \cdot k_{3,1} \end{aligned}$$

Table 6.11. Decryptions – input

Input
3 12 33 145 111 12 33 145 111 113 220 169 104 28 191 77 171 70 104 124 142 186 103 215 10 196 97 136 152 63 229 222 59 167 13 85 253 235 253 90 70 135 208 63 113 210 204 124 161 205 129 201 29 182 94 139 230 30 119 27 132 17 192 95 236 23 137 211 31 176 189 212 31 176 189 127 40 43 61 102 123 83 3 87 100 130 122 30 42 198 240 55 143 108 165 177 16 46 191 7 117 74 143 198 230 84 246 57 25 227 159 36 47 100 107 134 51 215 74 181 149 231 53 94 124 4 64

Table 6.12. Decryptions – output

Output
Faculty of Technical and Human Sciences, Marosvasarhely cannot decrypt

$$adjK = \begin{pmatrix} +\begin{vmatrix} k_{22} & k_{23} \\ k_{32} & k_{33} \end{vmatrix} & -\begin{vmatrix} k_{12} & k_{13} \\ k_{32} & k_{33} \end{vmatrix} & +\begin{vmatrix} k_{12} & k_{13} \\ k_{22} & k_{23} \end{vmatrix} \\ -\begin{vmatrix} k_{21} & k_{23} \\ k_{31} & k_{33} \end{vmatrix} & +\begin{vmatrix} k_{11} & k_{13} \\ k_{31} & k_{33} \end{vmatrix} & -\begin{vmatrix} k_{11} & k_{13} \\ k_{21} & k_{23} \end{vmatrix} \\ +\begin{vmatrix} k_{21} & k_{22} \\ k_{31} & k_{32} \end{vmatrix} & -\begin{vmatrix} k_{11} & k_{12} \\ k_{31} & k_{32} \end{vmatrix} & +\begin{vmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{vmatrix} \end{pmatrix}$$

$(detK)^{-1}$ can be calculated using the extended Euclidean algorithm with input $detK$ and 256.

6.12 Hash Values – 2015, Problem K

Several methods are known to calculate a hash value of a given password. In this problem, you should work with a simple variant.

It is supposed that a password is formed using only the lowercase letters of the English alphabet. In addition to the password, a salt value is known, formed using only the lowercase letters of the English alphabet.

First, the normalized value of a password is calculated. Given a certain length and a salt value, repeatedly append the salt value or its truncated form after the password to obtain the given length. If the given length is shorter than the password's length, cut the password's characters to reach the given length [1, 3].

Example:

```
"passw" "salt", 10 → "passwsalts"
"passw" "salt", 3 → "pas"
```

Second, a hash number is calculated using the following formula on the $c_1c_2\ldots c_n$ normalized value:

$$\sum_{i=1}^{n} i \cdot 2^{code(c_i)}$$

where $code(c_i)$ is equal to the ASCII code of c_i minus 97.

Finally, the hash number is converted to base 26, and 97 is added to each digit. This will correspond to one of the $a, b, \ldots, z$ characters for each digit.

Write a program that examines which hash value belongs to a given password, where it is supposed that the maximum length of the normalized value is equal to 10, and the output value is an integer.

The input file contains several test cases, where between each test case there is a blank line. The first two rows of each test case contain a password and the length of the normalized value. The next rows contain several hash values followed by $ as the separation symbol, and the corresponding salt value.

The output must be printed to the standard output and must consist of the position of the valid hash value. If the password does not belong to any hash value, print "INVALID."

Table 6.13. Hash values – input/output

Input	Output
passw	3
10	INVALID
wuxigg$xyzxyz	
jptce$abcefgh	
crnmyr$salt	
sqjce$abcefgh	
passw	
7	
wuxigg$xyzxyz	
jptce$abcefgh	
crnmyr$salt	

6.13 Addition of Points – 2018, Problem G

It is given a curve E defined over a finite field $F_p = \mathbb{Z}_p^*$ and an extra point $\mathcal{O}$, where $E : y^2 = x^3 + a \cdot x$, and p is a prime number greater than 3. This means that you should work over $\mathbb{Z}_p$. The addition rule for two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on such a curve is the following:

- If $P_1 = \mathcal{O}$, then $P_1 + P_2 = P_2$.
- If $P_2 = \mathcal{O}$, then $P_1 + P_2 = P_1$.
- Otherwise:
  - If $x_1 = x_2$ and $y_1 = -y_2$, then $P_1 + P_2 = \mathcal{O}$.
  - Otherwise $P_1 + P_2 = (x_3, y_3)$, where

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda \cdot (x_1 - x_3) - y_1 \\ \lambda &= \begin{cases} (y_2 - y_1) \cdot (x_2 - x_1)^{-1}, & \text{if } P_1 \neq P_2 \\ (3 \cdot x_1^2 + a) \cdot (2 \cdot y_1)^{-1}, & \text{if } P_1 = P_2 \end{cases} \end{aligned}$$

For a given P, Q two points on E, your task is to determine the value of t, for which $P = \underbrace{P + \cdots + P}_{\text{t additions on E}} = t \cdot Q$. It is known that $t \in \{0, 1, \ldots n-1\}$, where n is the smallest integer such that $n \cdot P = \mathcal{O} \in F_p$, so n is the order of the point P [1].

Write a program that for several values of (p, a, n, P, Q) calculates the corresponding values of t.

The input file contains several test cases. Each test case consists of one line: the first three values show p, a, n in this order, followed by the value of P and Q. Both P and Q are put in brackets and separated by a comma.

For each test case, print the corresponding t value. It is true that the value of t is less than n, and the value of p consists of max 32 bit.

Hint: The task is to solve the ECC-DL problem (elliptic curve discrete logarithm problem) for small numbers, where the ECC-DL problem means the following: let P be a generator element in the curve E defined over $(\mathbb{F}_p)$ with order q so it is true that $n \cdot P = \mathcal{O}$. In that case, the DL problem means that knowing the parameters of the curve, the P, $n \cdot P$, and q, the task is to calculate the positive number n.

Table 6.14. Addition of points – input/output

Input	Output
229 1 239 (5, 116) (155, 166)	176
23 1 14 (6, 4) (13, 16)	12
743059 1 742919 (67975, 118275) (467929, 594620)	20019

It is important to note that many cryptographic protocols are based on ECC, as it is believed that if p is at least 192 bits, then there is no efficient solution to the ECC-DL problem.

To solve the problem, the following two functions are required, where `myAdd` is the addition of two points, while `myMul` works similarly to the modular exponentiation algorithm presented earlier. The `modInv` function used was also presented earlier.

Addition of two points

```
void myAdd( myTuple& R, myTuple A, myTuple B,
            int pA, uInteger p) {
    uInteger x1 = get<0>(A);
    uInteger y1 = get<1>(A);
    uInteger x2 = get<0>(B);
    uInteger y2 = get<1>(B);
    uInteger lambda, temp;
```

```
    if (x1 == p && y1 == p) {
        R = myTuple(x2, y2);
        return;
    }
    if (x2 == p && y2 == p) {
        R = myTuple(x1, y1);
        return;
    }
    if (x1 == x2 && y2 == p - y1) {
        R = myTuple(p, p);
        return;
    }
    if (x1 == x2 && y1 == y2)
        lambda = (3 * x1 * x1 + pA) * modInv((2 * y1) % p, p);
    else {
        temp = (x2 - x1) % p;
        if (temp < 0) temp += p;
        lambda = (y2 - y1) * modInv(temp, p);
    }
    lambda = lambda % p;
    uInteger x3 = (lambda * lambda - x1 - x2) % p;
    if (x3 < 0) x3 += p;
    uInteger y3 = (lambda * (x1 - x3) - y1) % p;
    if (y3 < 0) y3 += p;
    R = myTuple(x3, y3);
}
```

Multiplication of a point

```
void myMul(myTuple& A, uInteger a, myTuple g, int pA, uInteger p) {
    myTuple temp;
    while ((a & 1) != 1) {
        myAdd(g, g, g, pA, p);
        a = a >> 1;
    }
    temp = myTuple(g);
    myAdd(g, g, g, pA, p);
    a = a >> 1;
    while (a > 0) {
```

```
        if ((a & 1) == 1) {
            myAdd(temp, temp, g, pA, p);
        }
        myAdd(g, g, g, pA, p);
        a = a >> 1;
    }
    A = temp;
}
```

You can test these two functions with the following `main` function:

Testing the functions

```
#include<iostream>
#include<tuple>
using namespace std;

typedef long long uInteger;
typedef tuple <uInteger, uInteger> myTuple;

void myMul(myTuple& A, uInteger a, myTuple g, int pA, uInteger p);
void myAdd(myTuple& R, myTuple A, myTuple B, int pA, uInteger p);
uInteger modInv(uInteger a, uInteger b);

void main() {
    uInteger p = 743059, a = 1, n = 742919, t = 20019;
    myTuple P = myTuple(67975, 118275);
    myTuple Q = myTuple(467929, 594620);
    myTuple Qres;
    myMul(Qres, t, P, a, p);
    cout << "check: " << get<0>(Qres) << " " << get<1>(Qres);
}
```

BIBLIOGRAPHY

[1] D. Boneh and V. Shoup. 2023. *A Graduate Course in Applied Cryptography Version 0.6*. Retrieved from https://toc.cryptobook.us/book.pdf.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[3] Péter Diviánszky, *Haskell Exercises*. Lecture notes. Eötvös Loránd University. Available at: https://people.inf.elte.hu/divip/ (accessed February 3, 2026).

[4] Donald Knuth. 2011. *The Art of Computer Programming*. Addison-Wesley Professional.

[5] László Lovász, József Pelikán, and Katalin Vesztergombi. 2003. *Discrete Mathematics. Springer*.

[6] Gyöngyvér Márton. 2008. *Kriptográfiai alapismeretek*. Scientia Publishing House.

[7] Jaehyun Park: *Introduction to Programming Contests*. Retrieved from https://web.stanford.edu/class/cs97si/.

[8] Kenneth H. Rosen. 1999. *Handbook of Discrete and Combinatorial Mathematics*. Chapman and Hall/CRC.

[9] Steven S. Skiena and Miguel A. Revilla. 2003. *Programming Challenges: The Programming Contest Training Manual*. Springer Verlag, New York – Berlin – Heidelberg.

[10] Victor Shoup. 2009. *A Computational Introduction to Number Theory and Algebra* (2nd ed.). Cambridge University Press.

[11] I. M. Vinogradov. 2016. *Elements of Number Theory*. Dover Publications.

CHAPTER 7

ECN-2023 PROBLEMS

Editors: Zoltán Kátai, Mirtill-Boglárka Naghi, Pálma Rozália Osztián

During the Sapientia-ECN competition in 2023, 14 challenges were put forward, and teams successfully addressed a minimum of 12 of them. The victorious team managed to tackle 8 of the presented problems. This chapter will delve into the details of these 14 challenges and their corresponding solutions, drawing from the materials provided by the authors.

7.1 Problem A

7.1.1 Problem Statement

Gifts (suggested by Andrei Arhire)

Fanurie wants to visit each of his N friends and leave a gift of a certain value. He drew a map in the form of an undirected tree with N vertices and found a rule by which he would visit them. From a node X, Fanurie does the following operations:

- leave the gift ready for node X,
- go through unvisited nodes in ascending order,
- return to the node from which he arrived at X.

Fanurie decides to always start from node 1. In order not to upset his friends, he decided to compensate for the delay by increasing the value of the gift. Thus, for each $2 \leq i \leq N$, $v[i]$ is known by how much the value of the gift received by the i-th visited friend increases compared to the value of the gift received by the $i-1$-th visited friend. It is considered that the first friend visited receives the value $v[1]$. Fanurie defines the subtree induced by a list of nodes as the set of nodes in the minimal subtree that contains each node in the list. It also defines the value of a list of nodes as the sum of the values of the gifts left in the nodes. Fanurie is asking you to help him answer Q queries of the form $SKa_1a_2 \ldots a_K$ with the following meaning:

Which is the node in the subtree induced by the K elements that added to the list of nodes would make the value of the list as close as possible to S? You cannot add a node that is already present in the list. If there are several options, choose the smaller node.

Input specification

- The first line contains two integers N and Q.
- The second line has N values separated by space, elements of v.
- The following $N-1$ lines have two integers separated by space, meaning that there is an edge between them.
- The following Q lines describe the questions: $S_i K_i a_{i,1} a_{i,2} \dots a_{i,K_i}$ for $1 \leq i \leq Q$.

Output specification

The output file will contain Q integers separated by a space, the answer to each query.

Restrictions

- $1 \leq N \leq 100.000$
- $1 \leq Q \leq 50.000$
- $1 \leq v[i] \leq 1.000, \forall 1 \leq i \leq N$
- $2 \leq K \leq 3$
- $1 \leq S \leq 1.000.000.000$
- It is guaranteed that there are at least $K+1$ nodes in any induced subtree.
- Fanurie promises to give you a priceless gift if you help him.

Sample input

```
9 3
10 5 6 4 3 2 3 1 4
1 2
1 3
2 4
2 5
4 6
4 7
3 8
3 9
73 2 5 6
```

```
34 1 3
1000000000 2 7 8
```

Output for sample input

```
2 8 9
```

Explanation

In the first query, the induced subtree has the node–value pairs $(2, 15), (4, 21), (5, 30), (6, 25), (7, 28)$. Both nodes 2 and 4 lead to a list value at a distance 3 from S (which is minimum). As for the next number, 8 leads to a list value equal with S.

7.1.2 The Key Idea

The author proposed two solutions. Both algorithms are based on the Depth-First Search (DFS) traversal of the tree, establishing a fixed order for node access. In addition, the first solution explicitly constructs an Euler tour too. In response to a specific query, the lowest common ancestor, denoted as X, is determined through binary lifting. The solution to the query is then obtained through binary search within the segment delineated by the first and last occurrences of node X in the Euler tour.

7.1.3 Background

In this subsection, we introduce fundamental graph representations: the adjacency list and adjacency matrix. The current problem is addressed using the adjacency list representation. We further discuss two primary graph traversal methods: Depth-First Search (DFS) and Breadth-First Search (BFS). For the present problem, the solution utilizes DFS traversal within the context of trees. Additionally, we provide a concise overview of DFS/BFS trees and how they can be stored using Parent Pointer Arrays.

We delve into the concept of Eulerian tours within the framework of DFS trees, an integral part of the solution to the current problem. Furthermore, we touch upon essential concepts such as shortest paths in the context of BFS traversal and topological order in the context of DFS traversals.

7.1.4 Representing Graphs

In the context of graph algorithms, there are two common main memory representations: the adjacency matrix and the adjacency list. An adjacency matrix is a 2D array where each cell at (i, j) represents an edge between vertex i and vertex j. For an undirected graph, the matrix is symmetric. An adjacency list is a collection of arrays/lists/vectors to represent the edges. Each vertex has a list of its adjacent vertices.

Below is an example of C++ code that reads input from a file, including the number of nodes, the number of edges, and the edge list, and then creates both the adjacency matrix and adjacency list using a bidimensional array and a one-dimensional array of vectors respectively.

C++ code for creating adjacency matrix and adjacency list

```
const int MAX = 100; // Assuming a maximum of 100 nodes
// Function to add an edge to the adjacency matrix
void addEdgeMatrix(int adjMatrix[][100], int u, int v) {
    adjMatrix[u][v] = 1;
    adjMatrix[v][u] = 1; // Uncomment for undirected graph
}
// Function to add an edge to the adjacency list
void addEdgeList(vector<int> adjList[], int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u); // Uncomment for undirected graph
}
int main() {
    ifstream inputFile("input.txt");
    int nodes, edges;
    inputFile >> nodes >> edges;
    // Initialize adjacency matrix and list
    int adjMatrix[MAX][MAX] = {0};
    vector<int> adjList[MAX];
    int u, v;
    for (int i = 0; i < edges; ++i) {
        inputFile >> u >> v;
        addEdgeMatrix(adjMatrix, u, v);
        addEdgeList(adjList, u, v);
    }
    inputFile.close();
```

```
    return 0;
}
```

7.1.5 Traversing Graphs

Depth-First Search (DFS) and Breadth-First Search (BFS) are two fundamental graph traversal algorithms. DFS is a recursive algorithm that explores as far as possible along each branch before backtracking. BFS explores the graph level by level, visiting all neighbors of a node before moving on to the next level.

C++ code for DFS traversal

```
const int MAX = 100; // Assuming a maximum of 100 nodes
void dfsUtil(vector<int> adjList[], int vertex, bool visited[]) {
    visited[vertex] = true;
    cout << vertex << " ";
    for (const auto& neighbor : adjList[vertex]) {
        if (!visited[neighbor]) {
            dfsUtil(adjList, neighbor, visited);
        }
    }
}
void dfs(vector<int> adjList[], int nodes) {
    bool visited[MAX] = {false};
    // Start DFS from each unvisited node
    for (int i = 0; i < nodes; ++i) {
        if (!visited[i]) {
            dfsUtil(adjList, i, visited);
        }
    }
}
int main() {
    // Assume you have an adjacency list: vector<int> adjList[MAX];
    // Call DFS
    dfs(adjList, MAX);
    return 0;
}
```

C++ code for BFS traversal

```
const int MAX = 100; // Assuming a maximum of 100 nodes
void bfs(vector<int> adjList[], int startNode, bool visited[]) {
```

```
    queue<int> q;
    q.push(startNode);
    visited[startNode] = true;
    while (!q.empty()) {
        int current = q.front(); q.pop();
        cout << current << " ";
        for (const auto& neighbor : adjList[current]) {
            if (!visited[neighbor]) {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}
void bfsTraversal(vector<int> adjList[], int nodes) {
    bool visited[MAX] = {false};
    // Start BFS from each unvisited node
    for (int i = 0; i < nodes; ++i) {
        if (!visited[i]) {
            bfs(adjList, i, visited);
        }
    }
}
int main() {
    // Assume you have an adjacency list: vector<int> adjList[MAX];
    // Call BFS
    bfsTraversal(adjList, MAX);
    return 0;
}
```

A DFS/BFS tree is a tree structure that represents the order in which nodes are discovered during a DFS/BFS traversal of a graph. These trees capture the structure of these explorations. The DFS/BFS tree can be stored using Parent Pointer Array: For each node, store a pointer/reference to its parent in the DFS/BFS tree. In an unweighted graph, the BFS tree also represents the shortest paths from the root node to all other nodes.

Subtrees of the DFS tree can be identified based on the concept of Euler tour. An Euler tour, or Eulerian tour, is a traversal of the edges of a graph that visits every edge exactly once. In the context of a DFS tree, an Euler tour can be obtained by traversing each edge twice: once when descending from the parent to the child and once when backtracking from the child to the parent. The Euler tour captures the entry and exit times for each node. During the DFS traversal, record the entry time and exit time for

each node. Entry time is when the node is first visited, and exit time is when the traversal is complete for that node. The Euler tour visits nodes in the order of their entry and exit times. Subtrees in the DFS tree correspond to contiguous segments in the Euler tour. More specifically, each subtree corresponds to a segment between the entry and exit times of its root node in the Euler tour.

In this context, it is worth highlighting the intricate connections between the concepts of topological order, DFS traversal, and Eulerian tours. A topological order is essentially a linear arrangement of vertices, ensuring that for every directed edge $u \to v$, vertex u precedes v in the ordering. In the context of DFS traversal, nodes completing the traversal earlier, indicated by a later exit time, will be positioned later in the topological order. Similarly, the relationship between exit times and the Eulerian tour is noteworthy. Nodes with later exit times during DFS traversal will consequently appear later in the Euler tour, which, in turn, reflects in their positioning within the overall topological order. As a result, recording nodes based on their exit times yields a reversed topological order, highlighting the intrinsic connection between these fundamental graph concepts.

7.1.6 Lowest Common Ancestors

The lowest common ancestor (LCA) of two nodes in a tree is the lowest (deepest) node that is an ancestor of both nodes. In other words, it is the node farthest from the root that is a common ancestor of the two given nodes. For a pair of nodes u and v in a tree: (i) The node itself can be its own ancestor, so the LCA is defined even when one node is an ancestor of the other; (ii) The LCA is the deepest node that is a common ancestor of u and v. (iii) The LCA can be one of the nodes u or v if either is an ancestor of the other. The concept of the lowest common ancestor is frequently used in various tree-related algorithms and data structures. One common application is in determining the nearest common ancestor of two nodes in a tree, which can be useful in tasks such as determining the distance between two nodes or finding the path between them. In the context of the present problem, we leverage the concept of the LCA to identify the subtree induced by a given list of nodes. This subtree is defined as the minimal subtree containing each node from the provided list. The use of the lowest common ancestor facilitates the efficient determination of the common ancestor node, allowing us to isolate the smallest subtree that encompasses all the specified nodes in the list.

There are several methods for determining the LCA of two nodes in a tree. The choice of method often depends on the characteristics of the tree and the specific requirements of the problem. For example, perform a depth-first traversal from the root of the tree to compute the depth of each node and record the parent of each node. During an LCA query, compare the depths of the two nodes and move up the tree until they reach the same depth. The author of the present problem employed the Binary Lifting method, a technique that preprocesses the tree to efficiently compute LCA queries in logarithmic time. This method involves storing the 2^k-th ancestor for each node, enabling rapid traversal through the tree during LCA queries.

More precisely, the Binary Lifting method has three phases:

- Initialization: Preprocess the tree by storing information about the 2^k-th ancestor for each node. For $k = 0, 1, 2, \ldots$, the 2^k-th ancestor of a node u is denoted as u_{2^k}. Initially, set u_{2^k} to be the parent of u for $k = 0$ (the immediate parent).
- Building the ancestors: Iterate over the tree nodes and compute the ancestors using dynamic programming. For each node u, set $u_{2^k} = (u_{2^{k-1}})_{2^{k-1}}$. This process effectively jumps from the current ancestor to its 2^k-th ancestor in logarithmic time.
- Querying the ancestors: To find the ancestor $u_{2^{k-1}}$, express k in binary representation. Starting with the node itself, traverse the bits from right to left. Whenever a bit is set to 1, make the corresponding jump to the 2^k-th ancestor.

The key advantage of binary lifting is that it allows for efficient traversal to any ancestor of a node without explicitly storing all ancestors. This makes it particularly useful for solving LCA queries and related problems in tree structures.

7.1.7 Algorithmic Approach

The author's algorithmic strategy (second solution) encompasses the following key elements:

- Based on the input sequence v, the array *val* is built, where each element represents the values to be attached to the nodes according to their DFS order. The resulting sequence is ordered in increasing fashion, which is a prerequisite for the application of binary search.
- The variable *sum* initially holds the current input value S, and then the values corresponding to the nodes from the current input query are subsequently extracted from it. Consequently, our objective is

to identify the node within the induced minimal subtree that has the closest possible value to *sum*. To exclude the K nodes specified in the current query (array *currQuery*), the array *on* is employed.
- The tree is represented through an adjacency list (denoted as $adjList$).
- In the context of a DFS traversal (function dfs), pre-computations are performed for each node to determine its 2^k-th ancestors. The dynamic programming construction process relies on the assumption that, for the ancestors of the current node, their 2^k-th ancestors have already been computed. The DFS order ensures the fulfillment of this condition. A bi-dimensional array *anc* is built where $anc[p][k]$ stores the 2^k-th ancestor of node p. Cell $anc[p][0]$ stores the direct ancestor of node p. Since $2^k = 2^{k-1} + 2^{k-1}$, the 2^k-th ancestor of node p (which is going to be stored in cell $anc[p][k]$) is the 2^{k-1}-th ancestor of its 2^{k-1}-th ancestor, which is already computed and stored in cell $anc[anc[p][k-1]][k-1]$. The author set the "ceiling value" for the base-2 logarithm of the upper bound of the number of nodes in the tree to be 20.
- Within the DFS traversal, the algorithm also pre-computes and stores the following information: the DFS order of the nodes (array $dfsOrder$); the level of each node in the DFS tree (array $level$); With respect to each node, starting and ending indexes in the DFS order are stored, specifying the segment that contains the nodes of the subtree rooted at that node (arrays $left$ and $right$).
- Function *ancF* returns the d-th ancestor of node p based on previously computed bi-dimensional array *anc*. All the positions with set bits in the binary representation of d are identified, and the corresponding 2-power jumps are executed. For example, if $d = 13$ (1101), then the d-high jump is performed as a succession of 2^0, 2^2, and 2^3 jumps.
- Function *lcaF* returns the Lowest Common Ancestors of nodes x and y. Considering that node y is at a deeper level than node x, an initial jump is performed to equalize the two levels. Then, at each step, the highest possible jumps are performed, making sure not to go beyond the lowest common ancestor. After this, the lowest common ancestor is the direct ancestor of x and y. If the current query involves more than two nodes, their lowest common ancestor is determined by repeatedly calling the function *ancF*.

– Function *Solve* identifies, on the array segment corresponding to the nodes of the induced minimal subtree ($dfsOrder[left[lca]..right[lca]]$), the searched node ($bestNode$) with the closest value to $tgVal$ = sum. The binary search is implemented by calling the C++ function *lower_bound* to find the position of the first element with an attached value not less than the target value $tgVal$. Then, the vicinity of this position is searched to find the node we are looking for (having the closest associated value and the lowest ID).

Author's C++ code

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

int n, q, timer;
int level[200005], anc[200005][22];
vector<int> adjList[200005];
int left[200005], right[200005], on[200005];
int dfsOrder[200005], val[200005];

//Determining the d-level higher ancestor of node p

int ancF(int p, int d){
 for (int b = 0; b <= 20; b++)
  if ((d >> b) & 1)
   p = anc[p][b];
 return p;
}

//Determining the lowest common ancestor of nodes x and y
int lcaF(int x, int y){
 if (level[x] > level[y]) swap(x, y);
 y = ancF(y, level[y] - level[x]);
 if (x == y) return x;
```

```
 for (int k = 20; k >= 0; k--)
  if (anc[x][k] != anc[y][k]){
   x = anc[x][k]; y = anc[y][k];
  }
 return anc[x][0];
}

//DFS traversal

void dfs(int p, int par){
 level[p] = level[par] + 1;
 anc[p][0] = par;
 for (int k = 1; k <= 20; k++)
  anc[p][k] = anc[anc[p][k - 1]][k - 1];
 timer++; left[p] = timer; dfsOrder[timer] = p;
 sort(adjList[p].begin(), adjList[p].end());
 for (int it : adjList[p]){
  if (it == par) continue;
  dfs(it, p);
 }
 right[p] = timer;
}

//Looking for the best node in the induces minimal subtree
int solve(int l, int r, int tgVal){
 int p = lower_bound(val + l, val + r + 1, tgVal) - val;
 int bestDif = 1e9, bestNode = 0;
 for (int i = max(p - 10, l); i <= min(p + 10, r); i++){
  if (!on[i]) continue;
  int nDif = abs(val[i] - tgVal);
  if (nDif < bestDif){
   bestDif = nDif; bestNode = dfsOrder[i];
  }
  else if (nDif == bestDif && dfsOrder[i] < bestNode)
   bestNode = dfsOrder[i];
 }
 return bestNode;
}
```

```
int main(){
 ios_base::sync_with_stdio(false); cin.tie(0);

 //Reading input data and creating arrays val, on and adjList
 cin >> n >> q;
 for (int i = 1; i <= n; i++){
  cin >> val[i]; val[i] += val[i - 1];
  on[i] = true;
 }
 for (int i = 1; i < n; i++){
  int x, y; cin >> x >> y;
  adjList[x].push_back(y);
  adjList[y].push_back(x);
 }

 //DFS-traversal starting from node 1
 dfs(1, 0);

 //Reading the input data for each query
 while (q--){
  int sum, k; cin >> sum >> k;
  vector<int> currQuery;
  for (int i = 1; i <= k; i++){
   int x ;cin >> x;
   currQuery.push_back(x);
   sum -= val[left[x]];
  }

  //Determining the lowest common ancestor
  int lca = currQuery[0];
  for (int it : currQuery)
   lca = lcaF(lca, it);

  //Determining the searched node of the induced minimal subtree
  //The nodes from the current query are excluded
  for (int it : currQuery)
   on[left[it]] = false;
  cout << solve(left[lca], right[lca], sum) << ' ';
  for (int it : currQuery)
```

```
   on[left[it]] = true;
 }
 return 0;
}
```

7.2 Problem B

7.2.1 Problem Statement

Colors in store (based on the T-shirt buying problem)

A new shipment of n tablecloths has arrived at a store. Each tablecloth is described by three attributes: price (p_i), front color (a_i), and back color (b_i), where p_i is integer and represents the cost of the i-th tablecloth, and a_i and b_i are integers ranging from 1 to 3, indicating the colors of the front and back of the tablecloth respectively. It is guaranteed that all p_i values are unique.

The store expects m customers to visit, and each customer intends to purchase exactly one tablecloth. For each customer, we know their favorite color (c_j). A customer will buy a tablecloth if either the front or back side matches their favorite color. In such cases, the customer selects the cheapest available option among the tablecloths that match their preferred color. If there are no tablecloths with an acceptable color for the customer, they will not make a purchase. The customers arrive sequentially, and each customer is served only after the previous one has been attended to.

Your task is to calculate the price each customer will pay for the tablecloth they choose.

Input specification

- The first line contains an integer $n(1 \leq n \leq 200{,}000)$, representing the number of tablecloths.
- The following line provides a sequence of n integers: $p_1, p_2, .., p_n(1 \leq p_i \leq 1,000{,}000{,}000)$, indicating the prices of the tablecloths.
- The next line contains a sequence of n integers: $a_1, a_2, ..., a_n(1 \leq a_i \leq 3)$, representing the front colors of the tablecloths.
- The subsequent line contains a sequence of n integers: $b_1, b_2, ..., b_n$ $(1 \leq b_i \leq 3)$, indicating the back colors of the tablecloths.

- Then, there is a single integer $m(1 \leq m \leq 200{,}000)$, denoting the number of customers.
- The next line provides a sequence of m integers: $c_1, c_2, ..., c_m$ $(1 \leq c_j \leq 3)$, signifying the favorite color of each customer. The customers arrive in the order they are listed in the input, and each customer is served only after the previous one has been assisted.

Output specification

Print m integers on the first line, where the j-th integer should represent the price of the tablecloth that the j-th customer will purchase. If the j-th customer decides not to buy anything, print -1 for that customer.

Time limit: $0.5s$

Sample input

```
5
300 200 400 500 911
1 2 1 2 3
2 1 3 2 1
6
2 3 1 2 1 1
```

Sample output

```
200 400 300 500 911 -1
```

Explanation

The provided input consists of:

- The number of tablecloths: 5
- Prices of the tablecloths: 300, 200, 400, 500, 911
- Front colors of the tablecloths: 1, 2, 1, 2, 3
- Back colors of the tablecloths: 2, 1, 3, 2, 1
- The number of customers: 6
- Favorite colors of the customers: 2, 3, 1, 2, 1, 1

The input represents 5 tablecloths with respective prices and colors on their front and back sides. There are 6 customers, each specifying their favorite color.

Each value in the output represents the price that each customer will pay for the tablecloth they choose. Specifically:

- Customer 1 (favorite color 2) will pay 200
- Customer 2 (favorite color 3) will pay 400
- Customer 3 (favorite color 1) will pay 300
- Customer 4 (favorite color 2) will pay 500
- Customer 5 (favorite color 1) will pay 911
- Customer 6 (favorite color 1) will not make a purchase, as there is no tablecloth with an acceptable color.

Thus, the output displays the prices each customer will pay for the selected tablecloths or -1 if a customer does not find a suitable tablecloth matching their favorite color.

7.2.2 The Key Idea

The main objective of the problem is to determine the minimum total increase in heights of tablecloths satisfying the given customer preferences. The problem requires efficient querying and selection of tablecloths based on their colors to minimize the overall cost.

The key approach to solving this problem involves organizing tablecloth information based on colors. By associating each color with a list of tablecloths having that color on either the front or back, it becomes possible to quickly access and identify the cheapest available option for each customer's preferred color.

Utilizing a data structure like a dictionary or hash map to map colors to corresponding tablecloths optimizes lookup time. This approach aims to efficiently handle customer requests, avoiding redundant searches and ensuring swift determination of the cost associated with each customer's color preference.

The algorithm efficiently processes each customer's request by retrieving the tablecloths matching their favorite color from the organized data structure and identifying the one with the minimum price, ultimately outputting the price of the selected tablecloth for each customer or -1 if no suitable tablecloth is found.

By efficiently organizing and accessing tablecloth information based on colors, the algorithm minimizes the total cost by selecting the cheapest available options matching each customer's color preference.

7.2.3 Background

7.2.3.1 The Original Problem

The "Colors in Store" problem is a variant derived from its original form, known as the T-shirt buying problem from the Codeforces platform. In the original scenario, a shop received a new pack of t-shirts, each characterized by distinct prices and front–back color combinations. Similarly, in this variant, the problem revolves around organizing t-shirts based on buyers' color preferences. Both problems involve buyers selecting t-shirts based on their favorite colors and the price of the available options. In the "T-shirt buying" problem, buyers specifically seek a single t-shirt that matches their preferred color, while in the "Colors in Store" problem, the focus is on organizing and serving buyers efficiently based on color preferences to minimize costs. The essence of optimizing selections based on color choices remains consistent across both problem variants, emphasizing the significance of efficiently serving customers' color preferences in retail scenarios.

To effectively tackle the "Colors in Store" problem, a solid understanding of data structures, particularly sets and their application in organizing information based on colors, is essential.

Sets in data structures

Sets are a fundamental data structure that stores unique elements in a specific order. In C++, std::set provides this functionality, allowing efficient organization of data without duplicates.

Organizing data by colors

A crucial aspect of the problem involves organizing tablecloth details based on colors. Utilizing sets allows each color to be associated with a set containing prices of tablecloths with that specific color on either the front or the back.

Understanding how to insert elements into sets, retrieve information based on specific criteria (in this case, colors), and iterate through sets is critical for efficient data handling in the problem scenario.

Sample set operations

Here is an example illustrating basic set operations for organizing tablecloth prices according to colors:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> colorPrices[4]; // Assuming 3 colors (1, 2, 3)

    // Inserting tablecloth prices based on colors
    colorPrices[1].insert(200);
    colorPrices[1].insert(300);
    colorPrices[2].insert(400);
    colorPrices[3].insert(500);
    colorPrices[3].insert(600);

    // Accessing prices of tablecloths by colors
    for (int i = 1; i <= 3; ++i) {
        std::cout << "Color " << i << ": ";
        for (int price : colorPrices[i]) {
            std::cout << price << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Mastering the utilization of sets for organizing data based on colors is crucial for effectively solving the "Colors in Store" problem. It enables efficient access and selection of tablecloth prices matching customers' color preferences, ultimately minimizing costs.

Applications

The "Colors in Store" problem involves selecting optimal options based on color preferences, and its concepts find applications in various domains:

Retail and e-commerce

In the retail industry, understanding customer preferences and optimizing selections based on color choices is crucial. Similar to the problem scenario, e-commerce platforms use algorithms to recommend products based on customer preferences, enhancing user experience and increasing sales.

Inventory management

Efficiently managing inventory by categorizing items based on attributes like color helps in optimizing stock levels and meeting customer demand. Algorithms akin to the problem's approach aid in organizing and offering the right products to customers, reducing excess inventory and enhancing supply chain efficiency.

Recommendation systems

The problem's concept aligns with recommendation systems used in various platforms. Algorithms analyzing customer preferences and suggesting products based on color choices resemble recommendation engines employed in streaming services, social media, and online shopping platforms.

Fashion and design

In the fashion and design industry, understanding color trends and preferences is vital. Algorithms similar to the problem's approach assist designers in predicting trends, creating color schemes, and recommending designs based on customer preferences.

Marketing and customer engagement

Understanding the psychology of colors in marketing and using algorithms to personalize marketing strategies based on color preferences contribute to enhanced customer engagement and brand loyalty.

Data analytics and business intelligence

Analyzing customer preferences based on colors through algorithms aids in deriving insights for data-driven decision-making. Such analysis

contributes to business strategies, marketing campaigns, and product development.

These applications demonstrate the relevance and versatility of the "Colors in Store" problem in various industries and domains, emphasizing its significance in understanding customer preferences and optimizing selections based on color choices.

7.2.4 Algorithmic Approach

The problem requires determining the cost each customer will pay for a tablecloth that matches their favorite color. The following algorithm efficiently handles this task:

Data organization:

1. Read the input, including the number of tablecloths, their prices, front colors, back colors, number of customers, and the favorite color for each customer.

2. Create a data structure (e.g., a dictionary or hash map) to store tablecloth information indexed by colors.

3. Organize tablecloth details based on colors, associating each color (1, 2, 3) with a list of tablecloths having that color on either the front or the back.

Processing customer requests:

1. For each customer:

   a) Retrieve the list of tablecloths associated with the customer's favorite color from the data structure.

   b) Among the tablecloths with the preferred color, identify the one with the minimum price, which the customer will buy.

   c) Output the price of the selected tablecloth. If no tablecloth matches the customer's preferred color, output -1.

Optimization and efficiency:

- Utilize a suitable data structure (e.g., dictionary or hash map) to efficiently map colors to corresponding tablecloths, optimizing lookup time.
- Organize tablecloth information based on colors to reduce redundant searches and ensure quick access to determine the cost for each customer.

Algorithm summary:

The algorithm performs the following steps:

1. Read input data and organize tablecloth information based on colors.
2. Process each customer's request efficiently by accessing the appropriate tablecloth based on their favorite color.
3. Output the price of the selected tablecloth for each customer. If no suitable tablecloth is found, output -1.

This approach ensures efficient handling of customer requests by organizing tablecloth data based on colors and optimizing the search for the cheapest available option matching each customer's color preference.

A possible solution in C++

```
#include <iostream>
#include <fstream>
#include <set>

using namespace std;
int main() {
    long time_begin = clock();

    ifstream fin("test_5_5.in");
    ofstream fout("test_5_5.out");
    if (!fin.is_open()) {
        cout << "Unable to open file" << endl;
```

```
        exit( 1 );
    }
    int n;
    fin >> n;
    long prices[200000];
    set<long> colors[4];
    for (int i = 0; i < n; ++i) {
        fin >> prices[i];
    }
    int color;
    for (int i = 0; i < n; ++i) {
        fin >> color;
        colors[color].insert(prices[i]);
    }
    for (int i = 0; i < n; ++i) {
        fin >> color;
        colors[color].insert(prices[i]);
    }
    int m;
    fin >> m;
    cout << "Ready to process" << endl;
    for (int i = 0; i < m; ++i) {
        fin >> color;
        if (colors[color].size() == 0) {
            fout << "-1 ";
            continue;
        }
        set<long>::iterator it = colors[color].begin();
        long price = *it;
        fout << price << " ";
        for(int j=1; j<4; ++j) {
            colors[j].erase(price);
        }

    }
    long time_end = clock();
    cout << "\nSeconds: "
             << (time_end-time_begin)/(double)CLOCKS_PER_SEC
             << endl;
```

```
    return 0;
}
```

7.3 Problem C

7.3.1 Problem Statement

Golden primes (suggested by Gyöngyvér Márton)

When a prime number has the form $p = \varphi^2 - \varphi - 1$, where $\varphi = 2^n$ and n is a positive integer, it is called a "golden prime." The task is to identify them.

Develop a program that calculates all values of n where p represents a "golden prime" less than b. The input file contains a positive integer $b < 2^{1000}$. Your program should output each value found for n on separate lines.

Sample input

```
1024
```

Sample output

```
2
4
5
```

7.3.2 The Key Idea

The essence of the problem is to identify the "golden primes," which are primes with a specific form: $p = \varphi^2 - \varphi - 1$, where $\varphi = 2^n$ and n is a positive integer. These "golden primes" have serious relevance in real-world applications. For example, in the TLS 1.3 Handshake Protocol, the prime used by the ECDH (elliptic-curve Diffie–Hellman) key agreement scheme is a "golden prime" with $n = 224$, so $p = 2^{448} - 2^{224} - 1$. In literature, this prime is called a Goldilocks prime because its form defines the golden ratio.

7.3.3 Background

The golden ratio in mathematics

The value φ of the golden ratio appears in mathematics as well as in architecture, arts, nature, etc. Mathematically, we say that two quantities, let them be a and b, where $a > b > 0$, are proportional to each other according to the golden ratio if:

$$\varphi \stackrel{def}{=} \frac{a}{b} = \frac{a+b}{a}.$$

In order to determine φ, the equation can be transformed into the following form:

$$\varphi = \frac{a}{b} = \frac{a+b}{a} = 1 + \frac{1}{\frac{a}{b}} = 1 + \frac{1}{\varphi}.$$

Based on the left and right members of the equality, we get the following quadratic equation:

$$\varphi^2 = \varphi + 1.$$

Regarding the problem, the selection of the prime $p = 2^{448} - 2^{224} - 1$ is purposeful due to its unique characteristics: $224 = 32{\cdot}7 = 28{\cdot}8 = 56{\cdot}4$. This factorization holds critical significance in computational efficiency, particularly for systems employing processors in 32- or 64-bit architectures. The problem highlights the vital role of specific primes in optimizing computational performance for secure communication protocols.

Miller–Rabin primality test

In cryptography, one crucial algorithm to decide whether a large number is prime or not is the Miller–Rabin probabilistic primality test. The idea of the algorithm is based on checking whether the Euler criterion (see below) is true for t randomly generated numbers. If the criterion is fulfilled in all t cases for a number m, the output of the algorithm will be True, i.e., it accepts that m is prime. However, when it finds a case where the criterion is not met, the output of the algorithm will be False, which means that m must be composite.

7.1. Theorem (The Euler Criterion). Let $m \in \mathbb{N}$ be odd, and write $m - 1 = 2^s \cdot r$ with $s \in \mathbb{N}$ being maximal. If m is prime, then for any x coprime to m, one of the following two conditions is true:

1. $x^r \equiv 1 \pmod{m}$,
2. there exists an integer j, $0 \leq j \leq s - 1$ such that:

$$x^{2^j \cdot r} \equiv m - 1 \equiv -1 \pmod{m}.$$

It is proved that the probability that a composite number m satisfies the Euler criterion with t different x values is less than $\frac{1}{4}^t$. So, in the algorithm, it is necessary to choose a security parameter $t > 1$, which determines how many different random x to choose. In practice, it is enough to test for $t = 10$ different x values because in this case the probability of error is $\frac{1}{4}^{10} \sim 10^{-6}$, which is considered an acceptable value.

Applications

Golden primes find applications in various domains owing to their unique properties and significance in mathematical contexts. Some notable real-life applications include:

- **Cryptography:** The TLS 1.3 Handshake Protocol employs a golden prime in the ECDH key agreement scheme, where $n = 224$, utilizing $p = 2^{448} - 2^{224} - 1$ as a prime number. The choice of this specific prime is due to $n = 224$ being divisible by 32, facilitating efficient algebraic operations on 32-/64-bit processors.
- **Efficient arithmetic operations:** The nature of specific n values, such as $n = 224$, being divisible by 32 enables more efficient arithmetic operations, particularly in processor architectures where certain bitwise operations align optimally.

7.3.4 Algorithmic Approach

The `goldPrime` function is based on the implementation of the Miller–Rabin primality test. For `n = 2, 3, ...`, it is tested whether the value of `p = phi**2 - phi - 1`, where `phi = 1 « n (= 2 ** n)`, is a prime number or not. If the `millerRabinT` function outputs `True`, the number `p` is considered probably prime, and its corresponding `n` value is printed.

Author's Python code

```
from random import randint

def goldPrime(b = 1 << 1000):
    for n in range(2, ):
        phi = 1 << n
        p = phi**2 - phi - 1
        if p > b: break
        if millerRabinT(p):
            print(n)

def millerRabinT(m, t = 10):
    s, r = 0, m - 1
    while r & 1 == 0:
        s, r = s + 1, r >> 1
    for i in range(0, t):
        x = randint(2, m - 1)
        y = pow(x, r, m)
        if y == 1 or y == (m - 1): continue
        for j in range(1, s):
            y = (y * y) % m
            if y == 1: return False
            if y == m - 1: break
        if y != (m - 1): return False
    return True

goldPrime()
```

7.4 Problem D

7.4.1 Problem Statement

Egyptian Fractions (suggested by Sándor Horváth)

Mathematics is old. Very old. Imagine we are in ancient Egypt more than 3,500 years ago. An Egyptian fraction is a finite sum of distinct unit fractions like this:

$$\frac{1}{5}+\frac{1}{24}+\frac{1}{840}.$$

The sum of these fractions is 17/70. Perhaps it is not completely obvious that every fraction can be written as a sum of distinct unit fractions, so every fraction is an Egyptian fraction, too. The decomposition is not necessarily unique. For example:

$$\frac{17}{70}=\frac{1}{5}+\frac{1}{24}+\frac{1}{840}=\frac{1}{7}+\frac{1}{10}.$$

The reason the Egyptian people used these fractions was a practical one. Imagine you have to pay 17/70 of a quantity. Instead of repeating 17 times giving the fraction of 1/70, you can prepare in advance some unit fractions and just give two of them, 1/7 and 1/10; so, the usage of Egyptian fractions is far more practical. The drawback of this method is the problem of decomposition of a fraction in a sum of distinct unit fraction. Lots of related questions are still open today. For example, it is conjectured, but still not proved, that for every $n \geq 5$ the fraction $4/n$ can be decomposed in a sum of not more then 3 unit fractions (Erdős–Straus conjecture, verified up to $n < 10^{17}$).

A natural question is to ask: given a fraction, what is the minimum number of unit fractions in its Egyptian number representation? It is not known whether a polynomial time algorithm exists for this problem. Even the computational complexity of this problem is not known. We can, however, use at least a brute-force search for this computation, but you may as well come up with a more efficient algorithm. Nevertheless, right now, your task is to design an algorithm to compute the Egyptian fraction representation(s) of a given fraction with the *fewest* possible terms.

Question: Suppose you have a fraction of positive integers, with a value less than 1. Compute all of its expansions in a sum of unit fractions with the minimum number of terms.
Input specification

It contains several lines, each having a fraction written in the form *a/b*, where a and b are positive integers, $0 < a < b$.

Output specification

The output contains the same number of rows, and each row contains a number, the minimum number of unit fractions, followed by the space-delimited denominators of the minimum length expansion in parenthesis, a comma + space and other such denominators, if the expansion is not unique. In each row, use the increasing lexicographical ordering to list the denominators and the parentheses, too.

Sample input

```
33/182
223/12342
4/7
511/1404
```

Sample output

```
2 (7 26)
2 (102 121)
2 (2 14)
3 (3 36 351), (4 9 351), (4 13 27)
```

7.4.2 The Key Idea

At the first look, the problem may appear as one that involves specific mathematical background. This is not the case: the problem can be treated on a pure algorithmic base, no specific mathematical knowledge is needed to successfully solve the problem – at least for a general approach.

There is a straightforward greedy algorithm to compute a decomposition in a sum of simple fractions.

- Take the fraction $f = \frac{a}{b}, 0 < a < b$.
- If $a = 1$, you are ready.
- Otherwise, subtract from f the fraction $\frac{1}{n}$, having the smallest possible denominator, and repeat until the difference itself is of this form.

This algorithm terminates because each step the numerator strictly decreases – actually, it is a very fast algorithm. However, the decomposition obtained in this way is not the shortest one, nor it gives multiple solutions if such exist.

The key idea is to modify this algorithm in such a way to cover the analysis of all possible decompositions. The details are provided in a subsequent chapter.

7.4.3 Background

As stated before, no specific background is necessary to design a general algorithm to compute the minimal decompositions. However, the running time can vary greatly, and there could be fractions for which this approach is not efficient. Note that according to Wikipedia, "It is possible to use brute-force search algorithms to find the Egyptian fraction representation of a given number with the fewest possible terms or minimizing the largest denominator; however, such algorithms can be quite inefficient. The existence of polynomial time algorithms for these problems, or more generally the computational complexity of such problems, remains unknown."

Certainly, specific algorithms can be developed for fractions with additional properties – for example, fractions with numerator 4.

A natural question is to ask if the requirement that all denominators of the decompositions be different numbers is a restrictive one or not. The answer is simple: it is not. Indeed, a sum of two identical simple fractions can always be replaced by a sum of at most two other nonidentical simple fractions. This is easy – look at the following:

Suppose you have the sum

$$\frac{1}{n}+\frac{1}{n}.$$

Suppose n is even, $n=2k$. Then

$$\frac{1}{n}+\frac{1}{n}=\frac{2}{n}=\frac{2}{2k}=\frac{1}{k}.$$

Now suppose n is odd, $n=2k-1$. Then

$$\frac{1}{n}+\frac{1}{n}=\frac{2}{n}=\frac{2}{2k-1}=\frac{1}{k}+\frac{1}{k(2k-1)}.$$

Consequently, minimal decompositions can be thought as having all denominators different, without restricting generality.

7.4.4 Algorithmic Approach

The greedy algorithm described previously can be modified in the following way.
Start the algorithm successively searching for a decomposition of length $K = 1, 2, 3, ...$ and stop if the list of found K term decompositions is not empty – these are therefore the minimal decompositions searched for. In each step, possible chunks of decompositions are constructed, and all these are passed on to the next level. Also, each remainder fraction which has a nominator greater than 1 will cause at the next level a number of bifurcations in the range between the value of n computed as in the greedy algorithm and the greatest value of m, for which the k term sum $\frac{1}{m} + \frac{1}{m+1} + \ldots + \frac{1}{m+k-1}$ is not longer than the remainder fraction. The value of k is the number of remaining depth steps until the verification of the proposed K depth is completed.

Well, the description above has a minor "bug:" it can generate some minimal decompositions twice or several times due to the ordering variations of the simple fractions. This behavior can be eliminated by not allowing starting values of n less than the value of the previous term of the current sequence of partial decomposition.

The structure of the algorithm is not complicated. To grasp the idea, it is maybe much easier to see these steps on a concrete numeric example, which is presented below.

```
-- depth 1 -----------------------------------------------------------------
{8/11}
-- depth 2 -----------------------------------------------------------------
{{1/2,*}, {1/3,*}, {1/4,*}}
-- depth 3 -----------------------------------------------------------------
{{1/2,1/5,*}, {1/2,1/6,*}, {1/2,1/7,*},...,{1/2,1/12,*},
{1/3,1/4,*}, {1/3,1/5,*}, {1/3,1/6,*},
{1/4,1/5,*}}
-- depth 4 here begin to appear solutions for the first time ---------------
{{1/2,1/5,1/37,1/4070}, {1/2,1/5,1/38,1/1045}, {1/2,1/5,1/39,*},...,
   {1/2,1/5,1/72,*},
{1/2,1/6,1/17,*}, {1/2,1/6,1/18,*}, {1/2,1/6,1/19,*},..., {1/2,1/6,1/32,*},
{1/2,1/7,1/12,*}, {1/2,1/7,1/13,*}, {1/2,1/7,1/14,*},..., {1/2,1/7,1/23,*},
{1/2,1/8,1/10,*}, {1/2,1/8,1/11,*}, {1/2,1/8,1/12,*},..., {1/2,1/8,1/19,*},
{1/2,1/9,1/10,*}, {1/2,1/9,1/11,*}, {1/2,1/9,1/12,*},..., {1/2,1/9,1/16,*},
{1/2,1/10,1/11,*}, {1/2,1/10,1/12,*}, {1/2,1/10,1/13,*},..., {1/2,1/10,1/15,*},
{1/2,1/11,1/12,*}, {1/2,1/11,1/13,*}, {1/2,1/11,1/14,*},
{1/2,1/12,1/13,*},
{1/3,1/4,1/7,1/924}, {1/3,1/4,1/8,*}, {1/3,1/4,1/9,*},..., {1/3,1/4,1/13,*},
{1/3,1/5,1/6,*}, {1/3,1/5,1/7,*}, {1/3,1/5,1/8,*}, {1/3,1/5,1/9,*},
{1/3,1/6,1/7,*}, {1/3,1/6,1/8,*},
```

```
{1/4,1/5,1/6,*}}
-- depth 5 is not started ----------------------------------------------------
```

These are the list trees successively grown for the fraction $\frac{8}{11}$. Notice that the star "*" character stands for fractions that are not simple and that will be further expanded in case no correct decomposition is arrived at the current level.

Finally, the set of minimal solutions – collected from the lists constructed at this last level of depth – would be:

```
{1/2,1/5,1/37,1/4070}, {1/2,1/5,1/38,1/1045}, {1/2,1/5,1/40,1/440},
{1/2,1/5,1/44,1/220}, {1/2,1/5,1/45,1/198}, {1/2,1/5,1/55,1/110},
{1/2,1/5,1/70,1/77},
{1/2,1/6,1/17,1/561}, {1/2,1/6,1/18,1/198}, {1/2,1/6,1/21,1/77}, {1/2,1/6,1/22,1/66},
{1/2,1/7,1/12,1/924}, {1/2,1/7,1/14,1/77},
{1/2,1/8,1/10,1/440}, {1/2,1/8,1/11,1/88},
{1/3,1/4,1/7,1/924}
```

7.4.5 Implementation(s)

We present two implementations. The first one is the original implementation written in the Wolfram Mathematica, by the author of the problem, Sándor Horváth. The second one is the translation of this code in C++.

A rooted tree is generated level by level in a BFS manner. In each node, an interval [nMin, nMax] is determined (based on the current fraction) containing the possible next denominators. The value of nMin is determined based on the greedy choice. The value of nMax is determined based on a pre-established depth of the tree to be generated. To avoid repetition, only increasing denominator (decreasing unit fraction) sequences are generated.

The trickiest part of the algorithm is determining nMax. To illustrate, let us see how the value 4 is calculated at the first level (the corresponding nMin value is 2). We anticipate that fraction 8/11 can be decomposed in a 4-term sum. We choose the largest denominator (smallest unit fraction) for which the largest sum of a 4-length decreasing sequence of unit fractions (consecutive terms) is still bigger than 8/11 (with a choice smaller than this, even the largest possible sum would not reach the value of 8/11).

```
1/2 + 1/3 + 1/4 + 1/5 > 8/11
1/3 + 1/4 + 1/5 + 1/6 > 8/11
1/4 + 1/5 + 1/6 + 1/7 > 8/11
1/5 + 1/6 + 1/7 + 1/8 < 8/11
```

Obviously, at the next level, we consider sums of 3 terms. Let us see how the value 12 is computed as nMax for the node representing value (8/11-1/2). (The corresponding nMin value is 5.)

```
1/5 + 1/6 + 1/7 > 8/11-1/2
...
1/12 + 1/13 + 1/14 > 8/11-1/2
1/13 + 1/14 + 1/15 < 8/11-1/2
```

7.4.5.1 The Mathematica Code

We present an implementation written in the Wolfram Mathematica. This environment is very convenient for mathematical computations, is powerful, and offers even some implicit advantages: for instance, if you compute the sum of two fractions, Mathematica simplifies the fraction before performing the addition, and it also simplifies the result after the addition, without any explicit demand.

Listing 7.1. Implemantation in Mathematica

```
# The Mathematica Code
ClearAll[ShortestEgyptianFraction, SearchForShortestEgyptianFraction,
 Recurse, FindLimitsK]

FindLimitsK[f_, depth_] := Module[{nMax, nMin, n},
  nMin = Ceiling[1/f];
  For[n = nMin, f < Total[Function[x, 1/(n - 1 + x)] /@ Range[depth]],
    n++];
  nMax = n;
  Return[{nMin, nMax}]
  ]


Recurse[currentList_, depth_, terms_, bst_] :=
 Module[{best = bst, r, newListEntry, newListOfDecompositions = {},
   nMin, nMax, n, newTerms = {}, list = {}, count = 1, f, cL, f0, k},
  (*Print["XXX  ",Length[currentList],"  ",currentList];*)
  For[k = 1, k <= Length[currentList], k++,
   f = Last[currentList[[k]]];
   cL = Drop[currentList[[k]], -1];
   f0 = Last[cL, 0];
   (*Print[k,"   ",f];*)
   If[f != 0,
    {nMin, nMax} = FindLimitsK[f, depth];
    For[n = Max[nMin, If[f0 != 0, 1/f0 + 1, 0]],
     n <= nMax &&
      0 <= f - 1/n <=
       Total[Function[x, 1/(n + x)] /@ Range[depth - 1]], n++,
     r = f - 1/n;
```

```
      cL = Drop[currentList[[k]], -1];
      newListEntry = Append[Append[cL, 1/n], r];
      AppendTo[newListOfDecompositions, newListEntry];
      If[r == 0,
       newTerms = Append[terms, Drop[newListEntry, -1]];
       list = Flatten[newTerms];
       best = Append[best, list];
       (*Print["best  ",Length[best],"  ",best]*)
       ]
      ]
     ];
   If[depth >= 2, best = newTerms;
    (*Print[Length[newListOfDecompositions],"   ",
    newListOfDecompositions];*)
    best = Recurse[newListOfDecompositions, depth - 1, newTerms, best]
    ];
   Return[best]
   ]


ShortestEgyptianFraction[frac_, depth_, bst_] :=
 Module[{best = bst, listOfDecompositions = {{frac}}},
  best = Recurse[listOfDecompositions, depth, {}, best];
  Return[{Length[best], best}]
  ]


SearchForShortestEgyptianFraction[frac_, maxDepthLimit_] :=
 Module[{best = {}, size, maxDepth = 1},
  best = {};
  For[maxDepth = 1, maxDepth <= maxDepthLimit, maxDepth++,
   {size, best} = ShortestEgyptianFraction[frac, maxDepth, best];
   If[size > 0, Break[]]
   ];
  Return[{frac, size, maxDepth, best}]
  ]
```

7.4.5.2 The C++ Code

The proposed algorithm generates the tree presented above level by level in a BFS manner, storing the nodes row by row in an array of vectors (eDecomp). Element eDecomp[0][0] is the root node representing the input fraction, and row eDecomp[i+1] is generated to contain the "child nodes" of the "parent nodes" from row eDecomp[i]. For each node (struct FRACTION), the current denominator (den), the remainder fraction (a: numerator and b: denominator), and the previous row index of the "parent node" (parent) are stored. The generation process is stopped when solution leaf nodes appear at the currently generated (i+1)-th level (the resulting

remainder fraction is a unit fraction, and its denominator is the last term in the Egyptian decomposition).

```
#include <iostream>
#include <vector>
#include <math.h>
#define MAX 10

using namespace std;

typedef struct { //limits for the next denominator
    int nMin;
    int nMax;
} LIMITS;

typedef struct {
    int den; //current denominator
    int a,b; //remainder fraction
    int parent; //previous level index of the previous term
    bool last; //resulted remainder is a unit fraction
} FRACTION;

//eDecomp[0] contains only the input fraction
//eDecomp[i] contains possible i-th terms
vector <FRACTION> eDecomp[MAX];

double evalFunc(int n, int depth, double f) {
    double result = 0.0;
    for (int i = 0; i < depth; ++i) {
        result += 1.0 / (n + i);
    }
    return result;
}

LIMITS findLimitsK(int a, int b, int depth) {
    LIMITS limits;
    double f = (double)a / b;
    int nMin = (int)ceil( (double)b / a );
    int n;
    for (n = nMin; evalFunc(n, depth, f) > f; ++n);
    int nMax = n-1;
    limits.nMin = nMin;
    limits.nMax = nMax;
    return limits;
}

//prints the denominators along the root-to-leaf path leading to
```

```
//solution leaf (i, j); the j-th element on the i-th level of the tree
void printDecomp(int i, int j){
    if( i > 1 ){
        printDecomp(i-1,eDecomp[i][j].parent);
    }
    cout << eDecomp[i][j].den << " ";
}

//determining the greatest common divizor
int grComDiv( int x, int y ){
    while ( x != y ){
        if( x > y ){ x -= y; }
        else{ y -= x; }
    }
    return x;
}

void shortestEgyptianFraction(int a, int b, int maxDepth) {

    //input fraction is a unit fraction
    if( a == 1){
        printf("1 (%i)\n", b); return;
    }

    //eDecomp[0][0] represents the input fraction
    FRACTION firstF, parentF, childF;
    firstF.a = a; firstF.b = b;
    firstF.parent = -1;
    eDecomp[0].push_back(firstF);

    //the tree is generated level by level in a BFS manner
    for( int i = 0 ; ; ++i ){
        bool finish = false;

        //generating row eDecomp[i+1] from row eDecomp[i]
        for( int j = 0 ; j < eDecomp[i].size(); ++j ){
            parentF = eDecomp[i][j];
            LIMITS limits = findLimitsK(parentF.a, parentF.b, maxDepth-i);

            //generating child/next terms for eDecomp[i][j]
            for( int n = limits.nMin ; n <= limits.nMax ; ++n ){

                //we are only interested in increasing denominator
                //sequences
                if( i > 0 && n <= parentF.den ){ continue; }
```

```
            //a potentially valid child node is created
            childF.den = n;
            int aa = parentF.a*n-parentF.b;
            int bb = parentF.b*n;
            int gcd = grComDiv(aa,bb);
            childF.a = aa / gcd;
            childF.b = bb / gcd;
            childF.parent = j;
            if( childF.a == 1 ){
                //the remainder is a unit fraction
                childF.last = true;
                finish = true;
            }
            else{ childF.last = false; }

            //again: we are only interested in increasing denominator
            //sequences
            if( childF.last && childF.b <= childF.den ){ continue; }

            //valid next level node
            eDecomp[i+1].push_back(childF);
        }
    }

    //currently generated row contains last terms
    if( finish ){
        printf("%i ", i+2); //length of shortest sequences
        int firstPrint = true;

        //all shortest decompositions are printed out
        for( int j = 0 ; j < eDecomp[i+1].size(); ++j ){
            if( eDecomp[i+1][j].last ){
                if( firstPrint ){
                    printf("(");
                    firstPrint = false;
                }
                else{ printf(", ("); }

                //prints the 1..(i+1) level denominators
                printDecomp(i+1,j);

                //the (i+2)-th, term is the denominator of the remainder
                //unit fraction
                printf("%i)", eDecomp[i+1][j].b);
            }
        }
```

```
            printf("\n");
            break; //the generating process is stopped
        }
    }

}

int main() {
    int a, b, maxDepth = 5;
    //freopen("in.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    while( scanf("%i/%i\n", &a, &b) != EOF ){
        for( int i = 0 ; i < MAX ; ++i ){
            eDecomp[i].clear();
        }
        shortestEgyptianFraction(a,b,maxDepth);
    }

    return 0;
}
```

7.5 Problem E

7.5.1 Problem Statement

Lanovka – The cable car (suggested by Lehel István Kovács)

There are N peaks in the Vector Vista Mountains. We want to install cable car columns on K adjacent peaks so that if the largest height of the cable car (peak + column) is H, then on the other peaks the heights should be $H-1, H-2, ..., H-K+1$, so that the cable car continues to go up smoothly. The cable car always starts from the right and goes to the left.

We can achieve this by increasing the height of the cable car columns by 1 unit. By how many minimum units must the heights of columns be increased in total in order to create a cable car of length K?

Input specification

The first line of the input contains the number of test cases T ($1 \leq T \leq 20$). This is followed by two lines for each test case. The first line contains the number of peaks N ($1 \leq N \leq 200000$) and the length of the cable car K ($1 \leq K \leq N$). The second line contains N natural numbers H_i

($1 \leq i \leq N$), the heights of the mountain peaks ($1 \leq H_i \leq 1000000000$).

Output specification

For each test case, print a single number on a line by itself: the minimum number of units.

Sample input

```
1
5 3
8 4 5 9 6
```

Sample output

```
4
```

Explanation

The three possible cable cars are as shown in *Figure 7.1*:

- Case 1: Peaks 5, 9, 6 with heights 10, 9, 8 and differences 5, 0, 2. The sum is 7.
- Case 2: Peaks 4, 5, 9 with heights 11, 10, 9 and differences 7, 5, 0. The sum is 12.
- Case 3: Peaks 8, 4, 5 with heights 8, 7, 6 and differences 0, 3, 1. The sum is 4.

The minimum of the total differences is 4.

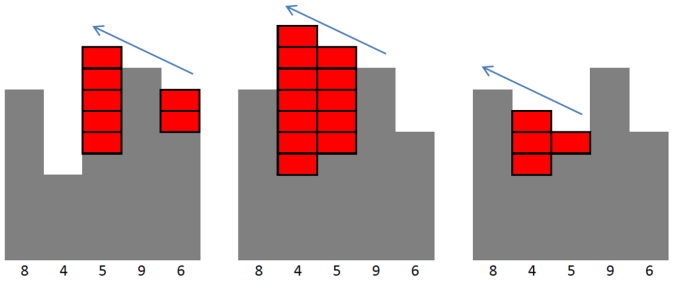


Figure 7.1. Three cases visualized

Problem proposed by Dr. Lehel István Kovács, based on a 2022–2023 problem at Nemes Tihamér International Programing Competition.

7.5.2 The Key Idea

The key part in this problem is to find the minimum total increase in heights of the cable car columns needed to satisfy the given conditions. The problem involves efficient querying and updating the maximum height and total height sum of consecutive peaks in the Vector Vista Mountains. The main idea in solving this problem is to use a range tree to efficiently perform range queries on the adjusted heights of the peaks, all the while avoiding redundant calculations.

7.5.3 Background

7.5.3.1 Range Trees

A range tree is a versatile data structure used for various range query tasks on an array or a list. It efficiently handles queries for information about subarrays such as finding the sum, minimum, maximum, or any other associative operation over a range of elements.

The main idea behind a range tree is to recursively divide the array into segments (ranges) until each segment represents a single element. Each node in the tree represents a segment of the array, and leaf nodes represent individual elements. The value stored in each node holds information about the segment it represents such as the sum, minimum, maximum, etc. *Figure 7.2* illustrates a range tree; each non-leaf node contains the maximum value within its left subtree.

Range trees are particularly useful when dealing with problems that involve frequent range queries or updates on an array. They provide a way to achieve efficient time complexity for these operations, often $\mathcal{O}(\log n)$, where n is the size of the array.

7.5.3.2 Applications

Range trees are data structures designed to efficiently answer queries involving multidimensional ranges or intervals. They have various applications in computer science and computational geometry due to their ability to handle multidimensional data. Here are some applications of range trees:

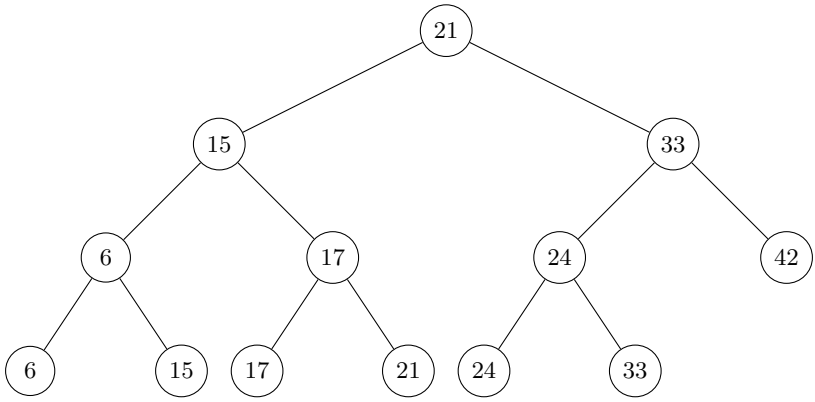


Figure 7.2. An example of a range tree. Every node other than the leaf node stores the highest value in its left subtree.

1. *Image processing:* Range trees can be used in image processing for efficient processing of multidimensional data. For instance, finding pixels within a rectangular region that satisfy certain criteria or extracting features from a region.
2. *Database query optimization:* In databases, range trees can be employed to optimize queries involving multidimensional data, especially in cases where ranges or intervals need to be considered in multiple dimensions simultaneously.
3. *Nearest neighbor search:* Range trees are useful for solving nearest neighbor search problems in multiple dimensions. They help in efficiently identifying the closest neighbors of a given point or object in a multidimensional space.
4. *Data mining and machine learning:* Range trees can be applied in data mining and machine learning tasks where multidimensional data needs to be analyzed. They facilitate efficient range-based queries and computations.
5. *Geographical information systems (GIS):* GIS applications often involve spatial data with multiple dimensions. Range trees can be used to optimize queries related to geographic information, such as finding points within a polygon or range.
6. *Spatial databases:* In spatial databases, range trees can be applied to optimize spatial queries. They help in efficiently retrieving data points that fall within specified ranges in multiple dimensions.

7. *Ray tracing in computer graphics:* In computer graphics, range trees can be used for efficient ray tracing. They help in identifying objects or points intersected by a ray in a multidimensional scene.
8. *Interval trees:* Range trees can be adapted for 1D intervals, making them useful in various applications involving one-dimensional range queries such as searching for overlapping time intervals.
9. *Statistical analysis:* Range trees can be applied in statistical analysis, especially when dealing with datasets that involve multiple dimensions. They assist in quickly obtaining statistical information within specified ranges.
10. *Geometric range queries:* Range trees are often used in computational geometry for solving geometric range queries efficiently. For example, finding all points in a 2D space within a given rectangle or identifying the nearest neighbors of a point.

In summary, range trees are versatile data structures with applications across various domains, particularly in scenarios where multidimensional data needs to be efficiently queried or analyzed.

7.5.4 Algorithmic Approach

The problem requires finding the minimum total increase in the heights of cable car columns so that the cable car can smoothly move from the rightmost peak to the leftmost peak.

A possible algorithmic approach utilizing range trees is the following: For each peak, its height must be adjusted by adding its index to it (*Figure 7.3*). This ensures that the cable car can smoothly move upward. For querying the maximum height and total height sum within a specified range of peaks, use a range tree. Iterate over all possible starting positions for the cable car (from 1 to $n - k + 1$). For each starting position, calculate the total height needed for the cable car to move smoothly over k peaks. Use the range tree to quickly query the maximum height and total height sum within the current range. Calculate the difference between the total height needed and the current total height sum (*Figure 7.4*). This represents the increase in height required for the cable car to follow the desired pattern. Update the minimum total increase needed by taking the minimum of the current minimum and the calculated increase. After iterating over all possible starting positions, output the minimum total increase needed for the cable car to traverse the mountains.

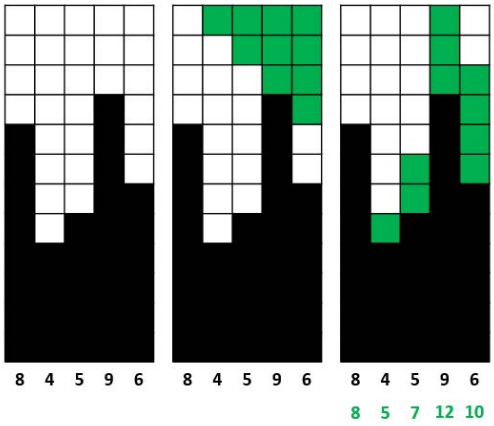


Figure 7.3. Adjusting the peak height by adding its zero-based index to it

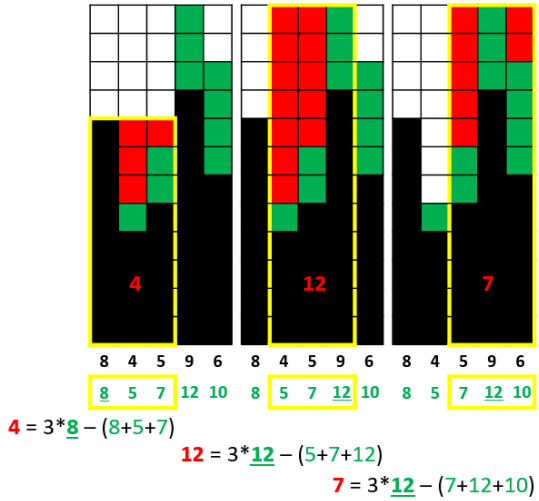


Figure 7.4. The differences between the total height sums and the current height sums

The use of the range tree is crucial for efficiently handling range queries on the adjusted heights of the peaks. This approach ensures an optimal solution to the problem by avoiding redundant calculations and efficiently querying the required information from the range tree.

C++ solution by the problem proposer

```
#include <bits/stdc++.h>
#include <iostream>
#include <fstream>

using namespace std;

// a constant representing the maximum size of the array
const long long maxn = 2e5;

// a constant representing a very large value, "infinity"
const long long inf = 1e16 + 42;

// number of peaks and length of the cable car, respectively
long long n, k;

// an array to store the adjusted heights of the peaks
long long val[maxn + 1];

// arrays representing the range tree nodes for storing
// maximum height and total height sum, respectively
long long tree_max[4 * (maxn + 1) + 1];
long long tree_sum[4 * (maxn + 1) + 1];

ifstream be("input.in");

// The build function is a recursive function to
// construct the range tree. It initializes
// leaf nodes with the adjusted heights of the
// peaks and calculates the maximum height and
// total height sum for each non-leaf node
// based on its children.
void build(long long v, long long vl, long long vr) {
    if(vl == vr) {
        tree_max[v] = tree_sum[v] = val[vl];
        return;
    }
    long long mid = (vl + vr) / 2;
```

```
    build(2 * v, vl, mid);
    build(2 * v + 1, mid + 1, vr);
    tree_max[v] = max(tree_max[2 * v], tree_max[2 * v + 1]);
    tree_sum[v] = tree_sum[2 * v] + tree_sum[2 * v + 1];
}

// The query_max function performs a query on
// the maximum height within a given range
// [ql, qr] in the range tree. It uses a
// recursive approach to traverse the tree,
// visiting only relevant nodes for
// the specified range.
long long query_max(long long v, long long vl, long long vr,
long long ql, long long qr) {
    if(ql > vr || qr < vl) {
        return 0;
    }
    if(vl == ql && vr == qr) {
        return tree_max[v];
    }
    long long mid = (vl + vr) / 2;
    return max(query_max(2 * v, vl, mid, ql, min(qr, mid)),
    query_max(2 * v + 1, mid + 1, vr, max(ql, mid + 1), qr));
}

// Similarly, the query_sum function performs
// a query on the total height sum within a
// given range [ql, qr] in the range tree.
// It also uses a recursive approach to traverse the tree.
long long query_sum(long long v, long long vl, long long vr,
long long ql, long long qr) {
    if(ql > vr || qr < vl) {
        return 0;
    }
    if(vl == ql && vr == qr) {
        return tree_sum[v];
    }
    long long mid = (vl + vr) / 2;
    return query_sum(2 * v, vl, mid, ql, min(qr, mid)) +
```

```
    query_sum(2 * v + 1, mid + 1, vr, max(ql, mid + 1), qr);
}

// The solve function is the main solving
// logic for each test case. It reads input
// values, adjusts peak heights, builds the
// range tree, and then iterates over all
// possible cable car starting positions.
void solve() {
    be >> n >> k;
    for(long long i = 1; i <= n; i++) {
        be >> val[i];
        val[i] += i;
    }
    build(1, 1, n);
    long long ans = inf;
    for(long long i = 1; i + k - 1 <= n; i++) {
        long long all = k * query_max(1, 1, n, i, i + k - 1);
        long long have = query_sum(1, 1, n, i, i + k - 1);
        long long needed = all - have;
        ans = min(ans, needed);
    }
    cout << ans << "\n";
}

// The main function reads the number of test cases,
// calls the solve function for each test case,
// and outputs the results.
int main() {
    int T;
    be >> T;
    for(int i = 0; i < T; ++i)
       solve();
    be.close();
    return 0;
}
```

7.6 Problem F

7.6.1 Problem Statement

Kings again (suggested by Csaba György Pătcaș)

Last year's problem with kings was too easy, so I made it more difficult. Determine the number of ways in which k kings can be placed on an $n \times m$ chessboard without attacking each other. Two kings attack each other if they are on adjacent cells horizontally, vertically, or diagonally.

Input specification

The first line of the input contains t, the number of test cases to follow. Each test case is described by the values of n, m, and k, in this order, separated by spaces.

Output specification

For each test case, you must output a single line, the number of ways modulo $10^9 + 7$.

Restrictions:

- $1 \leq t \leq 15000$
- $1 \leq n \leq m \leq 15$
- $1 \leq k \leq 100$

Sample input

```
3
1 4 2
3 3 4
4 4 4
```

Sample output

```
3
1
79
```

7.6.2 The Key Idea

The following concise material has been prepared by the problem's author, Csaba György Pătcaș. It outlines three potential approaches: a brute-force method and two dynamic programming algorithms. The first dynamic programming solution was introduced by the author, while the second one is derived from the source code of the only team, Unirea-1, that successfully tackled the problem during the contest.

1. Idea: $O((nm)^k)$

We could generate all the possible arrangements using backtracking. This would have worked for the problem proposed at ECN-2022, where n was up to 10, but for our problem even precalculating using this method is out of question.

2. Idea: $O(n^2m^2 2^{2m})$

We can build a directed acyclic graph having the nodes labeled by the tuple $(k, r, c, thisRow, nextRow)$, where k is the number of non-attacking kings placed on the board so far when going row by row starting from the first row and arriving at column c of row r. $thisRow$ and $nextRow$ are bitmasks representing the cells on rows r and $r+1$, respectively, that are being attacked by the kings placed so far.

After building the graph, we need to find out the number of ways to get from node $(0, 1, 1, 0, 0)$ to the nodes of the form $(k, n+1, 1, ?, ?)$, since the attacked cells are no longer relevant once k non-attacking kings have been placed. This can be calculated using standard dynamic programming after sorting the graph topologically.

Let us note that for fixed n and m we can find out the values for each $k \in [0, 100]$ by running the above algorithm once. Using this observation, we can precalculate all necessary values during the contest, which should take about $5-20$ minutes depending on the implementation and the hardware of the computers used. After placing the precalculated values inside a new source file, each query will be answered in $O(1)$ time.

3. Idea: $O(nmkC^2 + tC)$

First, let us generate all possible placements of non-attacking kings in a single row of length 15. Let us denote the number of these configurations by C ($C \simeq 1600$). Then, for each of these C configurations, we can determine the list of the configurations that can be on an adjacent row without introducing attacking kings.

Then, we can use dynamic programming to determine $dp_{c,r,x,k}$, the number of ways to place k kings up until row r on a board having c columns, the configuration of row r being x. After this step, every query can be answered in $O(C)$.

7.6.3 Code Implementation

Both program codes include comments provided by the author of the problem.

Author's C++ code

```
#include <vector>
#include <map>
#include <stack>
#include <algorithm>
#include <cstdio>
#include <iostream>

using namespace std;

#define FOR(i, a, b) for(int i = (a); i <= (b); ++i)

struct State{
 int kings, row, col, thisRow, nextRow;
 vector <int> to;
 long long cnt = 0;
};

const int MOD = 1e9 + 7;

vector <State> fifo;
map <tuple <int, int, int, int, int>, int> ind;
stack<int> ss;
vector <bool> visited;

void df(int node){
 visited[node] = true;
 for(int next : fifo[node].to)
  if (!visited[next]) df(next);
 ss.push(node);
}

void solve(int n, int m){
 int l = 0;
 fifo.clear(), ind.clear();
 fifo.push_back({0, 0, 0, 0, 0});
 ind[{0, 0, 0, 0, 0}] = 0;
 while (l < fifo.size()){
  State head = fifo[l];
  if (head.row == n) {++l; continue;}
```

```
  if ((head.thisRow & (1 << head.col)) == 0){
   //Put down a king
   int thisRow2 = head.thisRow;
   int nextRow2 = head.nextRow;
   thisRow2 |= 1 << head.col;
   nextRow2 |= 1 << head.col;
   if (head.col > 0){
\verb{    thisRow2 |= 1 << (head.col - 1);
\verb{    nextRow2 |= 1 << (head.col - 1);
   }
   if (head.col < m - 1){
\verb{    thisRow2 |= 1 << (head.col + 1);
\verb{    nextRow2 |= 1 << (head.col + 1);
   }
   int row2 = head.row, col2 = head.col + 2;
   if (col2 >= m){
    ++row2; col2 = 0;
    thisRow2 = nextRow2; nextRow2 = 0;
   }
   State next = {head.kings + 1, row2, col2, thisRow2, nextRow2};
   if (ind.count({next.kings, next.row, next.col, next.thisRow, next.nextRow}) == 0){
    fifo.push_back(next);
    fifo[l].to.push_back(fifo.size() - 1);
    ind[{next.kings, next.row, next.col, next.thisRow, next.nextRow}] = fifo.size() - 1;
   }
   else{
    fifo[l].to.push_back(ind[{next.kings, next.row, next.col, next.thisRow, next.nextRow}]);
   }
  }
  //Don't put down a king
  int thisRow2 = head.thisRow;
  int nextRow2 = head.nextRow;
  int row2 = head.row, col2 = head.col + 1;
  if (col2 >= m){
   ++row2; col2 = 0;
   thisRow2 = nextRow2; nextRow2 = 0;
  }
  State next = {head.kings, row2, col2, thisRow2, nextRow2};
  if (ind.count({next.kings, next.row, next.col, next.thisRow, next.nextRow}) == 0){
   fifo.push_back(next);
   fifo[l].to.push_back(fifo.size() - 1);
   ind[{next.kings, next.row, next.col, next.thisRow, next.nextRow}] = fifo.size() - 1;
  }
  else{
   fifo[l].to.push_back(ind[{next.kings, next.row, next.col, next.thisRow, next.nextRow}]);
  }
  ++l;
 }

 visited.clear(), visited.resize(fifo.size());
 df(0);
 fifo[0].cnt = 1;
 vector <long long> sum(101);
 while (!ss.empty()){
  int top = ss.top(); ss.pop();
  for (int next : fifo[top].to) {
   fifo[next].cnt += fifo[top].cnt;
   fifo[next].cnt %= MOD;
  }
  if (fifo[top].row == n) {
   sum[fifo[top].kings] += fifo[top].cnt;
   sum[fifo[top].kings] %= MOD;
  }
 }

 cout << "{";

 FOR(i, 0, 100) cout << sum[i] << ",";
 cout << "}," << endl;
}

int main(){
```

```
 FOR(i, 1, 15)
  FOR(j, i, 15)
  {
   solve(i, j);
  }
 return 0;
}
```

Unirea-1 team's C++ code

```
#include<iostream>
#include<vector>
using namespace std;
int dp[16][16][1600][105];
int ps[1600], st[16], x, nr[1600], l[1600];
vector<int>v[35005];
const int mod = 1e9 + 7;

//ps[x] - bitmask for configuration x
//nr[x] - number of kings for configuration x
//l[x] - column of last king in configuration x
//v[i] - list of all bitmasks representing rows
//that can come after row described by bitmask i

void Back(int k) {
 if (k == 16) {
  ++x;
  for (int i = 1; i <= 15; i++) {
   ps[x] += (1 << (i - 1)) * st[i], nr[x] += st[i];
   if (st[i])l[x] = i;
  }
  return;
 }
 if (k == 1) {
  st[k] = 0;
  Back(k + 1);
  st[k] = 1;
  Back(k + 1);
 }
 else {
  if (st[k - 1]) {
   st[k] = 0;
   Back(k + 1);
```

```
  }
  else {
   st[k] = 0;
   Back(k + 1);
   st[k] = 1;
   Back(k + 1);
  }
 }
}

void solve() {
 int n, m, k;
 cin >> n >> m >> k;
 int r(0);
 for (int i = 1; i <= x; i++)r += dp[n][m][i][k], r %= mod;
 cout << r << "\n";
}

//ok(a, b) - can rows described by
//bitmasks a and b placed next to each other?
bool ok(int a, int b) {
 bool q[18], w[18];
 for (int i = 1; i <= 15; i++) {
  q[i] = (a & 1);
  w[i] = (b & 1);
  a /= 2;
  b /= 2;
 }
 w[0] = w[16] = 0;
 for (int i = 1; i <= 15; i++) {
  if (q[i]) {
\verb{   if (w[i - 1] || w[i] || w[i + 1])return false;{
  }
 }
 return true;
}

int main() {
 freopen("a.in", "r", stdin);
```

```
freopen("a.out", "w", stdout);

Back(1);

for (int i = 1; i <= x; i++) {
 for (int j = 1; j <= x; j++) {
  if (ok(ps[i], ps[j])) {
   v[i].push_back(j);
  }
 }
}

//dp[c][r][x][k] - number of ways to place k kings up until
//row r on a board having c columns,
//the configuration of row r being x
for (int o = 1; o <= 15; o++) {
 for (int i = 1; i <= x; i++) {
  if (l[i] <= o) {
   dp[o][1][i][nr[i]] = 1;
  }
 }
}
for (int o = 1; o <= 15; o++) {
 for (int i = 2; i <= 15; i++) {
  for (int m = 1; m <= x; m++) {
   if (l[m] <= o) {
    for (int k = nr[m]; k <= 100; k++) {
     for (int j = 0; j < v[m].size(); j++) {
      if (l[v[m][j]] <= o) {
       dp[o][i][m][k] += dp[o][i - 1][v[m][j]][k - nr[m]];
       if (dp[o][i][m][k] >= mod)dp[o][i][m][k] -= mod;
      }
     }
    }
   }
  }
 }
}
int t;
```

```
 cin >> t;
 while (t--)solve();
 return 0;
}
```

7.6.4 A Simplified Version with Extra Explanations

Figure 7.5 presents the solutions for the first two test cases included in the problem statement. *Figures 7.6* and *7.7* show the valid one-row and valid consecutive-row configurations for $n = 4$.



Figure 7.5. The 3 solutions for input (1 4 2) and the 1 solution for input (3 3 4).

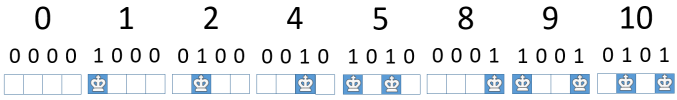


Figure 7.6. The 8 valid one-row king configurations for $n = 4$.

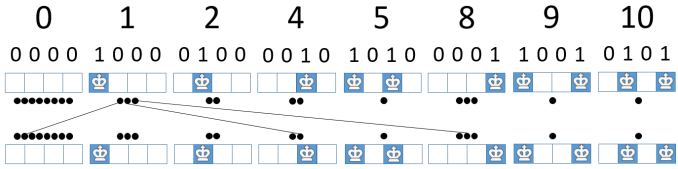


Figure 7.7. There are 21 $(8 + 3 + 2 + 2 + 1 + 3 + 1 + 1)$ valid consecutive row configurations for $n = 4$.

The third code presented below is an implementation of the following dynamic programming strategy:

– We solve the problem for the $N \times M$ maximum-size chessboard ($N = 15$, $M = 15$)

- During the solution-building process, we store the solutions for all intermediary-sized boards: 1×1, 2×1, ..., $N\times1$; 1×2, 1×2, ..., $N\times2$; ..., $1\times M$, $2\times M$,..., $N\times M$ (*Figure 7.8*).
- For each board size, we compute and store the solution for all k values ($k = 0 \ldots 100$).
  - Cell sol[row][col][k] stores the number of ways that k kings can be placed on the [0..row][0..col] board area (*Figure 7.8*).
  - For input (n m k), the output will be the value sol[n-1][m-1][k].

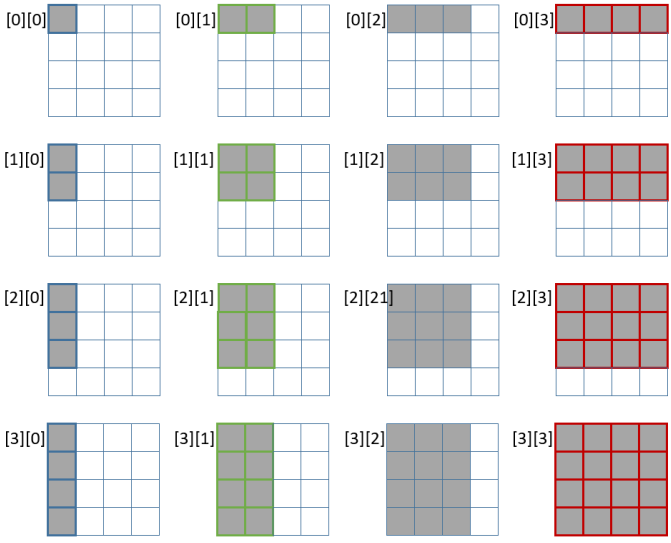


Figure 7.8. Illustrating the strategy for a 4×4 board. We solve the problem for all $1, 2 \ldots, m$ column-width chessboards.

  - We have a large number of test cases (up to 15,000).
  - Each query will be answered in O(1) time.
- Unfortunately, even a fourth dimension has to be introduced (dp[][][][]).
  - Remember: for all column widths, the solutions are built row by row.
  - For all valid last-row configurations, the solutions have to be stored separately.
    - Cell dp[row][col][k][c] stores the number of ways that k kings can be placed on the [0..row][0..col] board area when the last row configuration is c.

  - For a current last-row configuration, the solution is computed from the compatible previous-row solutions.
- Valid one-row king configurations are represented as m-length bitmasks.
  - There are 1,597 valid 15-length king configurations (precomputed value).
    - These are precomputed and stored in array $v[0 \dots 1{,}596]$.
    - Col-length configurations can be filtered by the condition: $v[j] < 2^{col}$, $j = 0 \dots 1{,}596$.
  - For all valid king configurations, vector previous $[j]$ ($j = 0 \dots 1{,}596$) stores the IDs of compatible previous row configurations.

Following this strategy results in the following code (for further details, see the comments):

```
#include <iostream>
#include <vector>

#define K 101
#define N 15
#define M 15
#define NN 1597

using namespace std;

unsigned short v[NN],nrKings[NN];
vector<int> previous[NN];
int dp[N][N][K][NN];
int sol[N][N][K];
const int MOD = 1e9 + 7;

bool validBitMask( unsigned short x ){

//x is valid if it does not contain consecutive 1s,
//representing attacking kings in the current row

    int n = 8 * sizeof(x);
    for( int i = 0 ; i < n-1 ; ++i ){
        if( ( x & (1<<i) ) && ( x & (1<<(i+1)) ) ){
            return false;
        }
    }
    return true;
}
```

```
int nrOneBits( unsigned short x ){

//It returns the number of 1s in the binary representation of x

    int n = 8 * sizeof(x), k = 0;
    for( int i = 0 ; i < n ; ++i ){
        if( ( x & (1<<i) ) ){ ++k; }
    }
    return k;
}

bool compBitMasks( unsigned short x, unsigned short y ){

//x and y are compatible if they do not have 1s
//at the same positions or neighboring positions,
//representing attacking kings in consecutive rows

    int n = 8 * sizeof(x);
    for( int i = 0 ; i < n ; ++i ){
        if( ( x & (1<<i) ) && ( y & (1<<i) ) ){
            return false;
        }
    }
    if( !validBitMask(x|y) ){
        return false;
    }
    return true;
}

int OneRowKingConfs( int n ){

//Precomputing arrays v and nrKings
//Array nrKings stores the number of kings
//in the corresponding valid configuration
//(for n=15, k will be 1597)

    int nn = 1<<n, k = 0;
    for( int i = 0 ; i < nn ; ++i ){
        if( validBitMask(i) ){
            v[k] = i;
            nrKings[k] = nrOneBits(i);
            ++k;
        }
    }
    return k;
```

```
}

void createPrevVector(){

//Precomputing vector-array previous

    for( int i = 0 ; i < NN ; ++i ){
        for( int j = 0 ; j < NN ; ++j ){
            if( compBitMasks(v[i],v[j]) ){
                previous[i].push_back(j);
            }
        }
    }
}

int main(){
    int n = N, m = M, k = K;
    int mm = OneRowKingConfs(m);
    createPrevVector();

    //building arrays dp[][][][] and sol[][][]

    //Since first rows does not have previous rows,
    //they are computed separately
    for( int col = 0 ; col < m ; ++col ){
        for( int j = 0 ; j < mm ; ++j ){
            if( v[j] >= (1<<(col+1)) ){ continue; }
            ++dp[0][col][nrKings[j]][j];
            sol[0][col][nrKings[j]] += dp[0][col][nrKings[j]][j];
        }
    }

    for( int col = 0 ; col < m ; ++col ){
    //for all column-widths
        for( int row = 1 ; row < n ; ++row ){
        //for all row-heights
            for( int i = 0 ; i < k ; ++i ){
            //for all intermediary k-values
                for( int j2 = 0 ; j2 < mm ; ++j2 ){
                //for all valid current row configurations
                    //not wider than col
                    if( v[j2] >= (1<<(col+1)) ){ continue; }
                    //not exceeding the targeted king number (i)
                    if( i - nrKings[j2] < 0 ){ continue; }
                    for( int j = 0 ; j < previous[j2].size() ; ++j ){
                    //for all valid previous row configurations
```

```
                        int j1 = previous[j2][j];
                        //not wider than col
                        if( v[j1] >= (1<<(col+1)) ){ continue; }
                        dp[row][col][i][j2] += dp[row-1][col][i-nrKings[j2]][j1];
                        dp[row][col][i][j2] %= MOD;
                    }
                    sol[row][col][i] += dp[row][col][i][j2];
                    sol[row][col][i] %= MOD;
                }
            }
        }
    }

    //freopen("kings.in","r",stdin);
    //freopen("kings.out","w",stdout);
    int n0, m0, k0, t0;
    scanf("%i", &t0);
    while( t0-- ){
        scanf("%i%i%i", &n0, &m0, &k0);
        printf("%i\n",sol[n0-1][m0-1][k0]);
    }
    return 0;
}
```

7.7 Problem G

7.7.1 Problem Statement

Breaking a quantum cryptography machine (suggested by Mirtill Boglárka Naghi)

During the Third Interstellar War, the Secret Service tasked you with cracking the enemy's quantum cryptography machine that protects their communications. The last captured cryptographic messages consist of instructions intercepted by the Secret Service and their corresponding solutions. The correctness of the messages have been verified by the Secret Service. Find out the operating principle of the quantum machine using the messages broken so far and re-implement it.

Input specification

The input contains a sequence of similar instructions to those in the captured messages, one instruction per line. The input may be considered correct in all cases. There are no conflicting instructions or such that cannot be executed at any point in time. All intermediary numbers as well as the results are integers that fit into the range of the int type.

Output specification

Print one line to the standard output, containing the solution of the sequence of instructions read from the input.

Sample input

Captured Instructions #1

```
SECRET_P 3
SECRET_P 13
SECRET_A
SECRET_P 5
SECRET_S
SECRET_E
```

Captured Instructions #2

```
SECRET_P 2
SECRET_P 8
SECRET_P 23
SECRET_S
SECRET_A
SECRET_E
```

Captured Instructions #3

```
SECRET_P 2
SECRET_P 8
SECRET_P 23
SECRET_A
SECRET_S
SECRET_E
```

Sample output

Solution #1

```
-11
```

Solution #2

```
17
```

Solution #3

```
29
```

7.7.2 The Key Idea

Each instruction in the input files can be mapped to a specific action within an imaginary stack-based virtual machine. Figuring out that a stack virtual machine must be implemented is the main challenge of this problem and is left up to the contestant.

Table 7.1. Instructions, actions, explanations

Instruction	Action	Explanation
`SECRET_P`	Push	Pushes the first argument of this instruction to the virtual stack.
`SECRET_E`	Echo	Prints the top element of the virtual stack to the output stream and removes the top element.
`SECRET_S`	Subtract	Subtracts the last element from the penultimate element, pushing the result to the top of the stack.
`SECRET_A`	Add	Adds the last element and the penultimate element together, pushing the result to the top of the stack.

Once the solution is identified, the unknown instructions must be mapped to a specific action. The input cases really help us out here – each input file can be manually executed, on paper, to find a combination of action definitions that returns the same output when actually executed. Not only that – but the last letter of each instruction also serves as a hint to the contestant. For example: `SECRET_A` refers to the `Add` action, while `SECRET_S` refers to the `Subtract` action.

By following the input cases, the subsequent actions can be untangled.

7.7.3 Background

The layout of the problem is reminiscent of how stacks are introduced in various university courses and indicates that this problem, too, must be of a similar nature. More often than not, university lectures are extremely verbose about the sort of actions that each instruction is responsible for. The challenge of this problem as such is to identify the actions taken by each instruction and to re-implement the virtual machine such as to conform to the original, input specification.

Applications

In real life, stacks are often used in actual encryption algorithms. In such algorithms, a series of transformations may be applied to a clear text using a multitude of transformation functions, finally forming a cipher text in the end. Obviously, such a simple mechanism as mentioned in this problem will never be used in actual environments. Not even aliens would exhibit such a level of incompetence. In reality, quantum communication machines would not even require encryption given that the very nature of quantum entanglement ensures that all communication between two entangled particles remains instantaneous and immune from man-in-the-middle snooping attacks.

Not only that, but stacks are also incredibly important in general-purpose computing. Most of the time, compilers of programming languages like `C` and `C++` are built upon the foundations laid down by the usage of stacks, turning human-friendly representations of computer code into architecture-specific instructions (which are, not surprisingly, also heavily built upon stack operations).

7.7.4 Algorithmic Approach

A naive approach to implement this solution is to use the `std::vector` class provided by the C++ Standard Template Library. However, this approach involves relatively convoluted index calculations, as it is not immediately clear how to access the last and penultimate values. It also requires the manual management of stack values.

An alternative solution is to use the `std::stack` class, which automatically manages these aspects for the programmer. The `std::stack` class provides an easy-to-use `pop` method, returning the last inserted

value from the stack, as well as automatically clearing said value from the stack. Armed with this knowledge and using this `std::stack` class, the resulting code is extremely clean, elegant, and, not to mention, efficient.

C++ solution by the problem proposer

```
#include <iostream>
#include <string>
#include <stack>
#include <fstream>

int main() {
    std::stack<int> memory; // The stack-based memory
    std::string instruction; // Current instruction

    std::ifstream fin("input.txt");
    if (!fin.is_open()) {
        std::cout << "Error opening input file."; return -1;
    }

    std::ofstream fout("input.out");
    if (!fout.is_open()) {
        std::cout << "Error opening output file."; return -1;
    }

    while (fin >> instruction) {
        if (instruction == "SECRET_P") {
            int x;
            fin >> x;
            memory.push(x);
        } else if (instruction == "SECRET_A") {
            int x = memory.pop();
            int y = memory.pop();
            memory.push(x + y);
        } else if (instruction == "SECRET_S") {
            int x = memory.pop();
            int y = memory.pop();
            memory.push(x - y);
        } else if (instruction == "SECRET_E") {
            int result = memory.pop();
```

```
            fout << result << std::endl;
            break;
        }
    }

    fin.close();
    fout.close();

    return 0;
}
```

7.8 Problem H

7.8.1 Problem Statement

"Optimizing" ascent: Navigating areas with bear activity in a binary matrix mountain (suggested by Zoltán Kátai)

Let us imagine a "mountain" with n levels ($i = 1, 2, \ldots, n$), where the levels (or "mountain slices") are encoded by $m \times m$ binary matrices. In each matrix, the value 1 indicates points belonging to the mountain. Both 1s and 0s form a connected region, considering two elements adjacent if they share a common side. Other two constraints are: (i) the mountain slices are "concentric" in the sense that in the positions of 1s at level i ($i = 2, \ldots, n$), there are also 1s at level $(i - 1)$; (ii) the levels have no common boundary points. In the last-level matrix, there is a single 1 representing the peak. Since all internal points at each level are considered bear-inhabited, we want to climb to the summit using the following method: (i) we can start from any boundary/edge point of level 1; (ii) from the edge of each level to the edge of the next level, we want to reach in the minimum number of steps, ultimately reaching the summit at the top level; (iii) climbing to the next level or moving along the edge of any level is considered safe (these areas are avoided by bears, as they have a fear of heights). If we follow the strategy outlined above, how many steps do we take in dangerous territory

from the base to the summit?

Input specification

The first line of the input contains t ($0 \leq t \leq 10$), the number of test cases. For each test case: (i) the first line contains n and m, the number of levels and the size of the square matrices ($1 \leq n \leq 100$, $3 \leq m \leq 100$); (ii) each of the next $n \cdot m$ lines contains m binary values (0/1), separated by one space, giving the elements of the matrices.

Output specification

For each test case, print a single line on the standard output containing the number of steps taken in dangerous territory.

Sample input

```
1
3 10
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 0 0
0 0 0 1 1 1 1 1 0 0
0 0 0 0 1 1 1 1 0 0
0 0 0 1 1 1 1 1 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Sample output

```
2
```

Explanation

The sample input consists of a single test case ($t = 1$) corresponding to a 3-level mountain coded by 10×10 matrices. Bolded 1s represent edge-points. The underlined 1s represent the accessed dangerous points. At level 1, the edge points of level 2 are accessed by 1 internal step (step (3,9)-(3,8)), and at level 2, the position of the peak point from level 3 is also accessed by 1 internal step (step (5,5)-(5,6)).

7.8.2 The Key Idea

This problem can be solved by a variant of the Lee algorithm.

7.8.3 Background

The Lee algorithm, also known as Lee's algorithm or the Wavefront algorithm, is a pathfinding algorithm used for maze solving and finding the shortest path in a grid. It is based on the principles of breadth-first search (BFS). The algorithm starts from the source point and progressively explores the neighboring points in layers (or contours) until it reaches the

destination. Here is an overview of the algorithm and some suggested applications:

Lee Algorithm

```
Initialization:
   Start with the source point.
   Assign a value (usually 0) to the source point.
   Mark all other points as unexplored.
Iteration:
   Repeat the following steps
    until the destination is reached
    or no more points can be explored.
   For each point with the current value,
    mark its unexplored neighbors with the next value.
Termination:
   If the destination is reached, the algorithm terminates.
   If no more points can be explored
    and the destination is not reached,
    then there is no path.
Path Reconstruction (Optional):
   If a path is found,
    it can be reconstructed by starting from the destination
    and moving backward through the points
    with decreasing values until the source point is reached.
```

Applications

The below applications highlight the versatility of the Lee algorithm in solving various pathfinding and optimization problems in different domains.

Maze solving:

The Lee algorithm is commonly used to find the shortest path through a maze. The maze can be represented as a grid, and the algorithm efficiently explores the maze to find the optimal path.

Robotics and autonomous vehicles:

In robotics, the Lee algorithm can be employed for path planning in environments with obstacles. Autonomous vehicles also use similar algorithms to navigate through complex road networks.

Network routing:

Lee's algorithm can be adapted for finding the shortest path in network routing problems, where nodes represent network elements, and edges represent connections or links.

Image analysis and processing:

In image processing, the Lee algorithm can be applied to find the shortest path between two points on a binary image, which is useful in various applications such as medical image analysis and computer vision.

Game development:

The Lee algorithm is useful in games where characters or agents need to navigate through a grid-based environment. It helps in determining the most efficient path for characters to reach their goals.

Resource allocation:

In scenarios where resources need to be allocated efficiently, such as in logistics or supply chain management, Lee's algorithm can be adapted to find the optimal routes for transporting goods.

Telecommunications:

Lee's algorithm can be used in telecommunications for finding the optimal paths for data transmission in a network.

Pseudocode

```
function LeeAlgorithm(maze, start, destination):
  queue = Queue()
  identifier = 0
  setIdentifier(start, identifier)
  enqueue(queue, start)
  while queue is not empty:
     current = dequeue(queue)
     identifier = getIdentifier(current) + 1
     for neighbor in getUnexploredNeighbors(current):
         setIdentifier(neighbor, identifier)
         enqueue(queue, neighbor)
         if neighbor is destination:
            return reconstructPath(start, destination)
  return "No path found"
```

7.8.4 Algorithmic Approach

It can be observed that the binary matrices representing the mountain slices can be added and stored as one matrix (a) reflecting the relief of the mountain. For the sample input, this matrix (supplemented with a zero frame) is the following:

```
0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 2 2 2 2 1 0 0
0 1 1 1 2 2 2 2 2 1 1 0
0 1 1 1 1 2 3 2 2 1 1 0
0 1 1 1 2 2 2 2 2 1 1 0
0 1 1 1 2 2 2 2 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0
```

For each level ($1 \ldots n-1$), a modified Lee algorithm can be applied to determine the minimum number of steps to reach the next level boundary from the boundary of the current level. For each level, a matrix b is initialized with (-1)s for the external points and with (-2)s for the internal points (belonging to the mountain). At each level, the "starting point" for the Lee algorithm is all boundary points, and the algorithm ends when a next level boundary point is first reached. Since the minimum-length path segments that were detected at each level can be connected by bear-safe boundary points, we have the solution for the problem by simply adding the minimum steps corresponding to levels $1 \ldots n$.

Taking into account that the size of the input is relatively small, the author did not use a queue structure for implementing the Lee algorithm. Instead, at each step, matrix b is traversed repeatedly to determine the next contour (or layer) of the current level, whose points are exactly one step further from the edge of the level than the previous contour. The matrices b below correspond to levels 1 and 2 in the case of the sample input. At each level, two contours are traced: contour 0 includes the boundary points, and contour 1 is partially traced until the next level boundary is reached.

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1  0  0  0  0  0  0  0  0  0  0 -1
-1  0  1  1  1  1  1  1  1  1  0 -1
-1  0  1 -2 -2 -2 -2 -2 -2  0 -1 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0 -2 -2 -2 -2 -2 -2 -2 -2  0 -1
-1  0  0  0  0  0  0  0  0  0  0 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1  0  0  0  0 -1 -1 -1
-1 -1 -1 -1  0  1  1  1  0 -1 -1 -1
-1 -1 -1 -1 -1  0 -2 -2  0 -1 -1 -1
-1 -1 -1 -1  0 -2 -2 -2  0 -1 -1 -1
-1 -1 -1 -1  0  0  0  0 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Since the next level boundary is reached during the tracing of contour 1 at both levels 1 and 2, the result for the sample input is $1 + 1 = 2$.

7.8.5 Code Implementation

In this subsection, we present the commented code of the author.

Author's C code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define N 102
int currentLevel( int, int, int [][N], int [][N] );
```

```
bool hasNeighborB( int, int, int, int [][N] );
void initMatrixB( int, int, int [][N], int [][N] );
int main(){
    int t,n,m,a[N][N]={0},b[N][N]={0};
    scanf("%i", &t);
    for( int c = 0 ; c < t ; ++c ){
        scanf("%i%i", &n, &m);
        for( int i = 0 ; i <= m+1 ; ++i ){
            for( int j = 0 ; j <= m+1 ; ++j ){
                a[i][j] = 0;
            }
        }
        for( int k = 0 ; k < n ; ++k ){
            for( int i = 1 ; i <= m ; ++i ){
                for( int j = 1 ; j <= m ; ++j ){
                    int x; scanf("%i", &x);
                    a[i][j] += x;
                }
            }
        }
        int unsafeSteps = 0;
        for( int k = 1 ; k < n ; ++k ){ //level by level
            initMatrixB(k,m,a,b);
            unsafeSteps += currentLevel(k,m,a,b);
            // unsafe steps at the current level
        }
        printf("%i\n", unsafeSteps);
    }
    return 0;
}
void initMatrixB( int k, int m, int a[][N], int b[][N] ){
    //(-1)/(-2) for all external/internal points
    for( int i = 0 ; i <= m+1 ; ++i ){
        for( int j = 0 ; j <= m+1 ; ++j ){
            if( a[i][j] < k ){ b[i][j] = -1; }
            else{ b[i][j] = -2; }
        }
    }
}
```

```
int currentLevel( int k, int m, int a[][N], int b[][N] ){
    int kk = 0;
    while( 1 ){
        //identifying the kk-distant contour at level k
        for( int i = 1 ; i <= m ; ++i ){
            for( int j = 1 ; j <= m ; ++j ){
                //next level boundary is reached in kk steps
                if( a[i][j] == k+1 && b[i][j] == -2
                    && hasNeighborB(i,j,kk-1,b) ){
                    return kk;
                }
                //continuing the current contour
                //with k-level internal points
                //that has neighbours on the previous contour
                if( a[i][j] == k && b[i][j] == -2
                    && hasNeighborB(i,j,kk-1,b) ){
                    b[i][j] = kk;
                }
            }
        }
        ++kk;
    }
}

bool hasNeighborB( int i, int j, int q, int b[][N] ){
    int di[] = {+1,0,-1,0};
    int dj[] = {0,+1,0,-1};
    for( int p = 0 ; p < 4 ; ++p ){
        if( b[i + di[p]][j + dj[p]] == q ){
            return true;
        }
    }
    return false;
}
```

7.9 Problem I

7.9.1 Problem Statement

Hamilton (suggested by Zoltán Szabó)

Santa Claus has decided this year to start delivering Christmas gifts on time, with the help of his elves. His favorite elf, named Hamilton, needs to traverse a 10,000-meter-long, 15-meter-wide stretch of road, where the coordinates (x, y) of each mailbox are precisely known ($0 \leq x \leq 10000, 0 \leq y \leq 15, x$ and y are natural numbers).

For any two mailboxes (x_a, y_a) and (x_b, y_b), the abscissas are different ($x_a \neq x_b$). And, of course, this condition cannot be fulfilled for the ordinates y_a and y_b. Hamilton, the elf, has to distribute approximately 4, 000 packages and would like to carry the packages to mailboxes in such a way that he travels the **shortest distance in a single round trip**, placing each package in the corresponding mailbox. In other words, the abscissas of the mailboxes continuously increase in the first segment and then continuously decrease. The length of the road should be measured from the left side, starting from the first mailbox, and it should end in the starting point.

The requirement for the task is to calculate, given the coordinates of the mailboxes, the shortest distance starting from the first mailbox.

Input specification

The first line of the input contains the number of mailboxes n. This is followed by n lines, each containing two natural numbers x and y, separated by a space, representing the coordinates of a mailbox.

Output specification

A single real number should be printed, representing the length of the shortest route, with exactly 6 decimals.

Sample input

```
5
4 5
1 1
```

```
6 5
5 3
9 1
```

Sample output

```
20.944272
```

Explanation

Hamilton, the elf, travels the following path from mailbox to mailbox (as shown in *Figure 7.9* below):

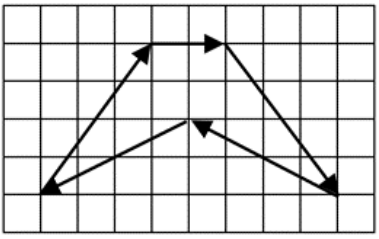


Figure 7.9. Hamilton's path

$5 + 2 + 5 + 4.47213595 + 4.47213595 = 20.944272$ (to a precision of 6 decimal places).

7.9.2 The Key Idea

The problem in its essence is finding the shortest bitonic Hamiltonian cycle in a specific graph. It can be efficiently solved by using dynamic programming to compute the shortest distances between pairs of mailboxes. The dynamic programming approach ensures that subproblems are solved only once, avoiding redundant calculations.

In the given problem, the mailboxes are sorted based on their x-coordinates, and the goal is to find the shortest bitonic Hamiltonian cycle that starts and ends at the first mailbox. The dynamic programming approach efficiently computes the distances for this specific scenario, ensuring that the traversal satisfies the conditions of increasing and then decreasing x-coordinates.

7.9.3 Background

The problem described involves finding the shortest bitonic Hamiltonian cycle. A bitonic Hamiltonian cycle is a Hamiltonian cycle in a bitonic tour. A bitonic tour is a tour (path) that first visits vertices in increasing order of a certain parameter (in this case, the x-coordinate) and then in decreasing order of that parameter.

The credit for the development of the optimal bitonic tour problem is often attributed to Jon L. Bentley. In 1990, Bentley conducted experiments comparing various heuristics for the traveling salesman problem, a classic optimization problem. However, it is worth noting that his experiments did not specifically include bitonic tours.

The first publication that formally introduces and describes the bitonic tour problem appears to be the 1990 first edition of the textbook *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, and Ron Rivest. In this publication, Bentley is acknowledged as the originator of the problem.

7.9.3.1 Applications

Bitonic Hamiltonian cycles – which involve a tour that first visits vertices in increasing order of a certain parameter and then in decreasing order of that parameter – have applications in various fields. Here are some examples:

1. *Vehicle routing:* In logistics and transportation, finding an optimal route for vehicles is crucial. Bitonic Hamiltonian cycles can model scenarios where vehicles need to visit locations in a specific order based on some parameter (e.g., distance, time) and then return to the starting point.

2. *Robotics:* In robotics, particularly in applications like path planning for autonomous robots, bitonic Hamiltonian cycles can be used to optimize the order in which a robot visits specific locations. This is essential for efficiency in tasks such as exploration or package delivery.

3. *Manufacturing and assembly lines:* In manufacturing and assembly processes, where components need to be assembled or processed in

a specific order, bitonic Hamiltonian cycles can be employed to optimize the workflow. This can lead to reduced travel distances and increased efficiency.

4. *DNA sequencing:* Bitonic Hamiltonian cycles can be applied in bioinformatics for DNA sequencing. The nodes could represent specific genes or genetic markers, and the bitonic cycle might represent an optimal order for sequencing the genes.

5. *Circuit design:* In the design of integrated circuits, bitonic Hamiltonian cycles can be used to optimize the order in which specific components are placed on the chip. This can lead to shorter interconnect distances, reducing signal delays and improving overall performance.

6. *Tour planning:* In tourism or event planning, where a sequence of attractions or events needs to be scheduled in a specific order, bitonic Hamiltonian cycles can be useful in optimizing the tour for efficiency and customer satisfaction.

7. *Data visualization:* In data visualization, particularly in the arrangement of points on a graph, bitonic Hamiltonian cycles can be applied to create visually pleasing and efficient layouts.

8. *Optical character recognition (OCR):* In OCR applications, where characters in an image need to be recognized and processed in a specific order, bitonic Hamiltonian cycles can be used to optimize the recognition process.

These are just a few examples, and the applicability of bitonic Hamiltonian cycles depends on the specific characteristics of the problem at hand. The general idea is to optimize the order in which certain elements are visited or processed based on a particular parameter.

7.9.4 Algorithmic Approach

The solution involves the commonly used top-down table filling method of dynamic programming. Table filling is particularly common in problems where optimal solutions to subproblems can be efficiently combined to solve larger problems. Each entry in the table typically represents the solution to

a subproblem, and the table is filled in a way that avoids redundant computations by storing and reusing the results of previously solved subproblems.

The provided code below is an example of the bottom-up approach with table filling. The 2D array (matrix) `d` serves as the table where solutions to subproblems are stored and reused in the iterative process. The matrix will store the distances computed between two points.

There are four cases:

1. The distance between a mailbox and itself is always zero. When visualized in a table, this represents the elements that are on the main diagonal. All elements on the main diagonal of the distance matrix will be exactly zero.

2. The distance between the j-th and the i-th mailbox is equal to the distance between the i-th and j-th mailbox. In this case, simply refer to the distance between the i-th and j-th mailbox rather than pointlessly recalculating these values. As a consequence, the table is symmetric with respect to the main diagonal – the same values can be observed in the top and the bottom portions of the table.

3. Directly above the main diagonal, there are multiple paths that can be taken between the two points. The distance is calculated by considering all possible predecessors k in the previous segment from which the minimum distance is chosen. For each predecessor k, the distance is calculated by adding the distance from k to i in the previous segment and the distance from k to the current point j. The minimum of these distances is then stored in `d[i][j]`, representing the optimal distance for the bitonic Hamiltonian cycle ending at point j. In essence, the algorithm is considering all possible ways to extend the bitonic cycle to include the current point j while minimizing the total distance traveled.

4. When $j - i > 1$, the optimal distance is the sum of the length of the optimal (already calculated) path from the i-th point to the $(j-1)$-th point and the distance between the $(j-1)$-th point and the j-th point.

Finally, to solve the actual problem, we must find the specific path that must be taken in order to create a complete bitonic Hamiltonian cycle

with the least distance travelled. In our case, there are $n-1$ final bitonic Hamiltonian cycles we can choose from. The distance travelled between a specific mailbox and the last mailbox from the table takes into account the shortest path taken between the two mailboxes and back, while also traversing through all mailboxes once (the only exception being the starting point, which is visited twice).

For example, if we have 4 mailboxes: $0, 1, 2, 3$, then we must choose the minimum distance travelled either from 0 to 3, from 1 to 3, or from 2 to 3. This minimum distance is therefore the final solution to the problem.

The complexity of this problem based on the aforementioned approach is $\mathcal{O}(n^2)$.

C++ solution by the problem proposer

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cmath>

using namespace std;

ifstream fin("input.in");
ofstream fout("output.out");
double d[3009][3009];

struct point {
    int x, y;
};

// Function to calculate Euclidean distance between two points
double dist(point u, point v) {
    double d, x1, y1, x2, y2;
    x1 = u.x;
    y1 = u.y;
    x2 = v.x;
    y2 = v.y;
    d = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
    return d;
```

```
}
// Comparator function to sort points based on x-coordinate
bool criterion(point u, point v) {
    return u.x < v.x;
}
int main() {
    int n;
    point a[3009];

    // Read the number of mailboxes and
    // their coordinates from input
    fin >> n;
    for (int i = 0; i < n; i++) {
        fin >> a[i].x >> a[i].y;
    }

    // Sort mailboxes based on x-coordinate
    sort(a, a+n, criterion);

    // Fill in the dynamic programming table
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                d[i][j] = 0;
                continue;
            }

            if (i > j) {
                d[i][j] = d[j][i];
                continue;
            }

            if (i == 0 || j - i > 1) {
                d[i][j] = d[i][j - 1] + dist(a[j - 1], a[j]);
                continue;
            }

            double mini = 2000000000;
```

```
            for (int k = 0; k < i; k++) {
                double nextv = d[i][k] + dist(a[k], a[j]);
                if (nextv < mini) {
                    mini = nextv;
                }
            }

            d[i][j] = mini;
        }
    }

    // Find the minimum distance for the
    // bitonic Hamiltonian cycle
    double mini = 2000000000;
    for (int i = 0;i < n - 1; i++) {
        double nextv = d[i][n - 1] + dist(a[i], a[n - 1]);
        if (nextv < mini) {
            mini=nextv;
        }
    }

    // Output the result
    fout << fixed << setprecision(6) << mini;
    return 0;
}
```

An alternative solution has also been proposed, utilizing a different data structure. It may be noted that only two rows are being accessed simultaneously in the original solution. As such, an alternative data structure may be to use two one-dimensional arrays instead of a fully-fledged two-dimensional matrix in order to reduce data duplication. The first array is used to compute the distances during traversal, while the second array is used to compute the minimum traversal distance, using the same concepts described previously.

Proposed alternative C++ solution

```
#include <vector>
#include <iostream>
```

```
using namespace std;
using ll = long long;
using ld = long double;

vector<pair<ld,ld>> v;
vector<ld> d;

ld dist(int i, int j) {
    return hypot(
        v[j].first - v[i].first,
        v[j].second - v[i].second
    );
}
ld path(int i, int j) {
    return d[j] - d[i];
}

int main() {
    int n;
    ld x, y;

    cin >> n;
    v = vector<pair<ld, ld>>();

    for (int i = 0; i < n; ++i) {
        cin >> x >> y;
        v.emplace_back(x, y);
    }

    sort(v.begin(), v.end());
    v.push_back(v.back());
    d = vector<ld>(n + 1,0);

    for (int i = 1; i <= n; ++i) {
        d[i] = d[i - 1] + dist(i - 1, i);
    }

    vector<ld> dp(n + 1, LDBL_MAX);
```

```
    dp[0] = dist(0, 1);

    for (int i = 1; i <= n; ++i) {
        for (int k = 0; k < i; ++k) {
            dp[i] = min(dp[i], dp[k] + dist(k, i + 1) +
            path(k + 1, i));
        }
    }

    cout << fixed << setprecision(6) << dp[n - 1] << endl;
    return 0;
}
```

7.9.5 Supplementary Explanations

Preparing the Dynamic Programming solution:

- After sorting the points based on their x-coordinates:
  - point 0 is the leftmost one;
  - point $(n-1)$ is the rightmost one;
- In the context of this problem, because of the symmetry, we can talk about left-right and right-left bitonic paths:
  - '$i \to 0 \to j'$ denotes a left-right bitonic path starting at point i and ending at point j;
  - '$i \to (n-1) \to j'$ denotes a left-right bitonic path starting at point i and ending at point j;
- If we determine for all $i = 0 \ldots (n-2)$ the shortest $i \to 0 \to (n-1)$ left-right bitonic Hamiltonian paths, then the shortest Hamiltonian cycle will result from the following formula:
  - $Min\{$`length_of_the_shortest_`$i \to 0 \to (n-1)$`_path` $+$ $dist[n-1][i]$, for all $i = 0 \ldots (n-2)\}$

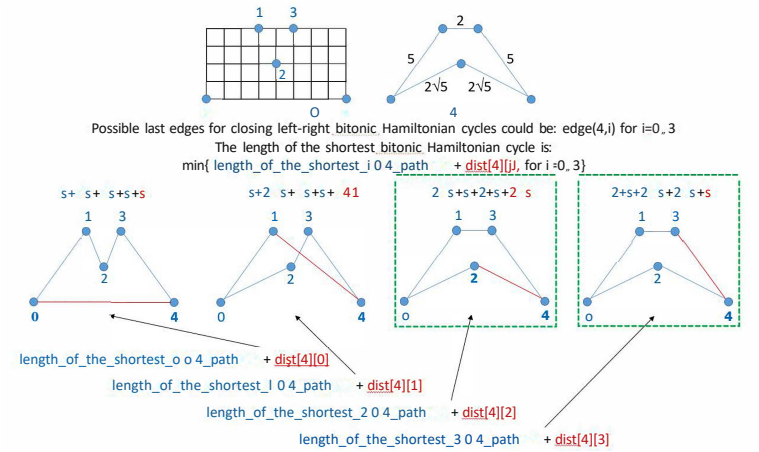


Figure 7.10. The final step of the DP solution construction process for the sample input

DP solution-building process:

- For all prefix-subsequences of the given (ordered) point sequence, we determine all the shortest left-right bitonic Hamiltonian paths that ends at the rightmost point of the subsequence:
  - $\{0,1\},\{0,1,2\},\ldots,\{0,1,\ldots,j\},\ldots,\{0,1,\ldots,(n-1)\}$;
  - Considering $\{0,1,\ldots,j\}$ as being the current prefix-subsequence, the left-right bitonic paths that end at point j could start at points $0\ldots(j-1)$;
    - Accordingly, we are interested in the shortest $i \to 0 \to j$ paths for all $i = 0\ldots(j-1)$;
    - Let cell dp[i][j] store the length of the shortest $i \to 0 \to j$ left-right bitonic path (which is Hamiltonian with respect to set $\{1,\ldots,j\}$).
- Observations:
  - During the solution-building process, we are only interested in bitonic paths (not circuits as well); hence, cells dp[i][i] can be skipped.
  - Obviously, $dp[i][j] = dp[j][i]$.
  - We focus on computing cells $dp[i][j]$ for $j > i$. (cells above the main diagonal).
    - Cells $dp[i][j]$ for $j < i$ are filled implicitly.

- Euclidean distances between all the pairs of points can be precomputed.
  - $dist[i][j] = dist[j][i] = \sqrt{((p[i] \cdot x - p[j] \cdot x)^2 + (p[i] \cdot y - p[j] \cdot y)^2)}$.
- Trivial subproblems are: $dp[0][1] = dp[1][0] = dist[0][1]$.
- It can be observed that for $j > (i+1)$, the penultimate point on an $i \to 0 \to j$ left-right bitonic path could be only point $(j-1)$.
  - Consequently: $dp[i][j] = dp[i][j-1] + dist[j-1][j]$.
- In the case of $i \to 0 \to (i+1)$ left-right bitonic paths, the penultimate points could be any point k, where $k = 0 \ldots (i-1)$ (points to the left of i).
  - Consequently: $dp[i][i+1] = min\{dp[i][k] + dist[k][i+1]$, where $k = 0 \ldots (i-1)\}$.
- The value $dp[0][1]$ results implicitly from the input data (trivial subproblem).
- Brown cells are computed based on the brown values from the prediagonal segment of their row.
  - $dp[i][i+1] = min\{dp[i][k] + dist[k][i+1]$, where $k = 0 \ldots (i-1)\}$.
  - Values on which the optimal choices rely are underlined.
- Yellow cells are computed based on their immediate left neighbor cells:
  - $dp[i][j] = dp[i][j-1] + dist[j-1][j]$, for$j > (i+1)$.

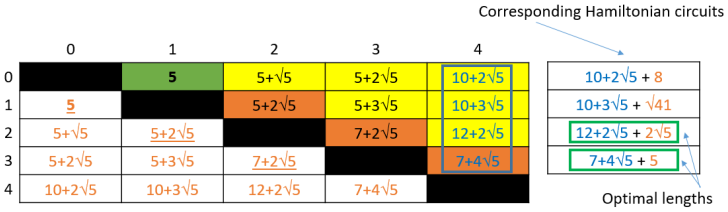


Figure 7.11. Computation of DP array cells for different subproblem types (green, brown, yellow)

Extra observations:

- It would have been enough to fill only the triangle above the main diagonal, column by column, from left to right, traversing each column from bottom to top.
  - The segment $dp[i][0 \ldots (i-1)]$ used to compute the cell $dp[i][i+1]$ is identical to the segment $dp[0 \ldots (i-1)][i]$.

- Since the current column is calculated from the one before it, storing only the current column in a one-dimensional array ($dp1[0 \dots (n-1)]$) is sufficient.
  - If the segment $dp1[0 \dots (j-1)]$ currently stores the segment $dp[0 \dots (j-1)][j]$ from column j, then the segment $dp1[0 \dots j]$ corresponding to the next column (column $j+1$) can be generated as follows:
    - Firstly: $dp1[j] = mindp1[k] + dist[k][j+1]$, where $k = 0 \dots (j-1)$.
    - Secondly: for all $i = 0 \dots (j-1) dp1[i] = dp1[i] + dist[i][j+1]$.

7.10 Problem J

7.10.1 Problem Statement

Find the Identical Twins and Triplets (suggested by Ion Cozac)

The real story (three identical strangers)

Starting with 1960, the psychiatrist Peter Neubauer conducted a controversial study about identical twins and triplets adopted apart. The purpose of this study was to determine which factor is more important in the development of a human being: genetics or environmental. Twenty years after, Robert Shafran, DavidKellman, and Eddy Galland found one another through sheer coincidence and discovered they were separated after their birth. After that, Peter Neubauer closed the study, all the collected data have been sealed at Yale University and cannot be opened until 2066.

The problem to be solved

Given a database of population, your task is to identify every group of identical twins or triplets (sharing the same DNA) and display the members of every group where at least one member is adopted.

Input specification

The database of population includes some pieces of information about each person, from which we selected only the necessary ones. The database is described as follows:

- The first line contains the number of people n ($n \leq 160000$).
- Each of the next n lines contains the following information about each person:
  - personal code (31-bit positive integer);
  - DNA signature (≤ 11 uppercase letters);
  - the character "`A`" if the person is adopted, or "`-`" (minus) otherwise;
  - name of the person (≤ 27 characters including whitespace).

After inserting a group of twins or triplets, unused positions between consecutive personal codes (considering the ascending order) will never be assigned. In other words, every group sharing the same DNA is compact, that is, if k_1 and k_2 share the same DNA, there will be no other distinct DNA between k_1 and k_2. For example, in the case of the sample input (see below), 128, 130, and 131 will never be assigned.

Output specification

After identifying all groups of identical twins or triplets, display the members (with personal code and name separated by a space) of each group where at least one member is adopted. Insert an empty line between any two consecutive groups. The list should be ordered by personal code.

Sample input

```
10
132 TCA - Lim Leo
135 GAT - Gal Vera
101 ACG A Ban Remo
105 ACT - Kan Mia
107 AGC - Mir Ando
111 AGC A Nor Ken
127 TCA - Lim Nico
103 ACT - Kor Ema
129 TCA A Pat Sam
```

```
124 TAC - Gor Alex
```

Sample output

```
107 Mir Ando
111 Nor Ken

127 Lim Nico
129 Pat Sam
132 Lim Leo
```

Explanation

The following groups are identified:
- 103, 105: no one is adopted, so we do not display this group.
- 107, 111: at least one member is adopted.
- 127, 129, 132: at least one member is adopted.

7.10.2 The Key Idea

The goal is to find groups of consecutive persons with the same genetic code in the sorted version of the input data, where at least one person in the group is adopted. The fact that such persons are found next to each other in the sorted data, consecutively, is guaranteed by the problem definition: "After inserting a group of twins or triplets, unused positions between consecutive personal codes (considering the ascending order) will never be assigned."

Given this crucial detail, that twins and triplets are always next to each other in the sorted input data, the solution of the problem is immensely simplified. A generalized solution, without noticing this small detail, would require the generation and testing of all 2- or 3-sized permutations. However, since all twin and triplet pairs are known to be found consecutively in the input data, the solution becomes clear: the problem can be solved by a simple iteration through the input data.

The key flow of the solution is the following:

1. Create a struct to hold the input data in memory.
2. Read the input data (members) into an array.

3. Sort all members by their personal code.
4. Iterate through all members and check whether the next few members share the same genetic code and whether at least one is adopted. In such a case, print all members from the group.
5. Update the next iterated member to the last member of the group found, to avoid classifying members into multiple groups.

7.10.3 Background

The problem was designed to test the contestant's ability to use structs, to sort structs, and to iterate through arrays of structs.

Applications

Learning about structs at university is crucial for several reasons. Structs provide a fundamental building block for organizing and encapsulating data that is related to a certain entity in a C++ program. They allow students to understand the concept of user-defined data types and facilitate the creation of complex data structures.

Structs are foundational to the principles of object-oriented programming (OOP) and are often a precursor to more advanced concepts like classes and objects. Understanding them lays the groundwork for grasping more advanced concepts in C++, enhancing students' ability to build efficient and well-organized software solutions throughout their academic and professional careers.

7.10.4 Algorithmic Approach

Creating a generalized solution to this problem, not taking into account the fact that all related members are found next to each other in the input data, would require the generation of all 2- and 3-sized permutations of the input data. This would result in an $O(n^3)$ solution.

However, since it is known that all pairs are found consecutively in the input data, a linear scan in $O(n)$ time is more than enough to solve this particular iteration of the problem.

First, sort all members by their personal code, which is required because the outputted members are meant to be sorted by personal code according

to the problem. You may focus on solving the actual problem after sorting the members in this fashion.

To solve the actual problem, simply iterate through all members in the input data. For each member, check the next members until you find a member with a different genetic code. You may consider all persons found after you in the list, with the same genetic code as you, part of a twin/triplet group. However, if there are no persons with the same genetic code as you right after your own entry, there is no group to consider. Before printing this group, however, you must ensure that at least one person in the group is adopted, in which case you may continue with printing all members in the twin/triplet group to the output.

C solution by the problem proposer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
  int k;
  char D[12], N[28], a;
} T_pers;

static T_pers Pers[160004];
static T_pers *Pps;

static int cmpK(const void *A, const void *B) {
  return ((T_pers *)A)->k - ((T_pers *)B)->k;
}

static T_pers *checkList(void) {
  static int prt;
  T_pers *Ppw;
  int k = 0;

  if (Pps->k == 0) {
    return NULL;
  }

  for (Ppw=Pps, k=0; strcmp(Pps->D, Ppw->D)==0; Ppw++) {
```

```
      if (Ppw->a == 'A') {
        k++;
      }
    }

    if (k > 0) {
      if (Ppw - 1 > Pps) {
        if (prt) {
          printf("\n");
        }

        for (Ppw = Pps; strcmp(Pps->D, Ppw->D) == 0; Ppw++) {
          printf("%d %s\n", Ppw->k, Ppw->N);
        }

        prt++;
      }
    }

    return Ppw;
  }

  int main() {
    int k;
    int npe;
    scanf("%d", &npe);

    for (k = 0, Pps = Pers; k < npe; k++, Pps++) {
      scanf("%d %s %c %[^\n]", &Pps->k, Pps->D, &Pps->a, Pps->N);
    }

    qsort(Pers, npe, sizeof(T_pers), cmpK);
    for (Pps = Pers; Pps = checkList(););

    return 0;
  }
```

7.11 Problem K

7.11.1 Problem Statement

Dependencies (suggested by Márk Kósa)

Amazon Web Services (AWS) CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want, and CloudFormation takes care of provisioning and configuring those resources for you. In a CloudFormation template, you specify the logical names of resources, together with their properties, including the dependent resources. The logical name (or logical ID) must be alphanumeric and unique within the template and is used to reference to the resource in other parts of the template. In the following CloudFormation template, such logical names are `MyEC2Instance` and `MyEIP`:

```
AWSTemplateFormatVersion: 2010-09-09
Description: A sample template
Resources:
    MyEC2Instance:
        Type: 'AWS::EC2::Instance'
        Properties:
            ImageId: ami-0ff8a91507f77f867
            InstanceType: t2.micro
            KeyName: testkey
            BlockDeviceMappings:
                - DeviceName: /dev/sdm
                    Ebs:
                        VolumeType: io1
                        Iops: 200
                        DeleteOnTermination: false
                        VolumeSize: 20
    MyEIP:
        Type: 'AWS::EC2::EIP'
        Properties:
            InstanceId: !Ref MyEC2Instance
```

Write a program that reads logical names of resources given in an AWS CloudFormation template from the standard input. For each logical name, you will be given all the other names of resources that are required to exist for the resource to be created. Your program has to determine whether a given sequence of resources can or cannot be created in the specified order, and if not, the program also has to give the logical names of all resources in the sequence that cannot be created.

Input specification

The first line of the input contains two integers R and S, representing the number of resources and the number of sequences respectively ($1 \leq R \leq 100, 1 \leq S \leq 100$). Each of the following R lines described are source and its dependencies (in no particular order). Each resource is represented by an identifier, which is a string of at most 20 alphanumeric characters. The first resource in each line is followed by a colon (“:”), and then optionally a space-separated list of its dependencies, also being resource identifiers. Finally, the last S lines of the input contain the sequences, in the form of space-separated resource IDs. Both the resource descriptions and the sequences will contain only valid resource IDs.

Output specification

For each sequence in the input, print one line to the output. If there are at least two resources in the sequence that cannot be created in the given order (“problematic” resources), then print “Problematic resources:”, followed by a space-separated list of the logical names of all the problematic resources in the same order as they appeared in the sequence. If there is only one problematic resource, print the message “Problematic resource:”, followed by the logical name of the problematic resource. If all the resources can be created in the order presented in the sequence, then print “No problematic resources.”

Sample input

Sample input #1

```
2 2
MyEC2Instance:
```

```
  MyEIP: MyEC2Instance
  MyEC2Instance MyEIP
  MyEIP MyEC2Instance
```

Sample input #2

```
5 2
A: B C D
B: D
C:
D:
E: A B
A B C D E
D B C A E
```

Sample output

Sample output #1

```
No problematic resources.
Problematic resource: MyEIP
```

Sample output #2

```
Problematic resources: A B E
No problematic resources.
```

Explanation

The input data consists of three sections:

1. **The input specifications:** How many resources are there (R), and how many sequences must be tried (S)?
2. **The resources:** Each of the subsequent R lines contains a single resource name, as well as their dependencies. Each dependency needed to be created previously in order to successfully create this specific resource.
3. **The sequences:** Each of the subsequent S lines contains a sequence of resources that must be created.

The first input file describes 2 resources: `MyEC2Instance` and `MyEIP`.

`MyEC2Instance` requires no dependencies – it can be created at any time. However, `MyEIP` has one dependency: `MyEC2Instance`. This means that if we try to create `MyEIP` without first creating `MyEC2Instance` previously in the same sequence, then the creation will fail. In this case, `MyEIP` would become a "problematic resource," as it cannot be created in the sequence.

The first input file also describes 2 sequences that must be created. The first sequence attempts to create `MyEC2Instance` and then `MyEIP`. `MyEC2Instance` requires no dependencies, so it can be created. `MyEIP` can also be created because it has one dependency (`MyEC2Instance`), which has already been created previously. In this case, the output is "No problematic resources." – all resources have been successfully created.

The second input file describes 5 resources: `A`, `B`, `C`, `D`, and `E`. Each resource may have dependencies on other resources. Let us analyze the dependencies:

- `A` depends on `B`, `C`, and `D`.
- `B` depends on `D`.
- `C` has no dependencies.
- `D` has no dependencies.
- `E` depends on `A`, `B`, `C`, and `D`.

Now, let us consider the sequences provided:

1. The first sequence attempts to create `A`, `B`, `C`, `D`, and `E` in order.
   - `A` cannot be created because it depends on `B`, `C`, and `D`, all of which have not been created yet.
   - `B` cannot be created either, as it depends on `D`, which is not yet part of the sequence.
   - `C` has no dependencies and can be created.
   - `D` has no dependencies and can also be created.
   - `E` depends on `A` and `B`. Since `A` and `B` have not been created yet, this resource cannot be created.
2. Therefore, the output for the first sequence is `Problematic resources: A B E` – since `A`, `B`, and `E` could not be created due to dependency issues.
3. The second sequence attempts to create `D`, `B`, `C`, `A`, and `E`, in that order.
   - `D` can be created, as it has no dependencies.

   - `B` can be created, as it depends on `D`, which has already been created.
   - `C` has no dependencies and can be created.
   - `A` can be created, as it depends on `B`, `C`, and `D`, all of which have been already created.
   - `E` depends on `A` and `B`, all of which have been created. It can also be created.

4. Therefore, the output for the second sequence is `No problematic resources`, because all resources have been successfully created.

7.11.2 The Key Idea

The key idea to solve the problem is to simulate the creation of each resource in a sequence. A resource can only be created if all of its dependencies have been previously created as well. Thus, the main idea is as follows:

1. Read the list of resource types, and their respective dependencies from the input.
2. For each line denoting a sequence, attempt to create all resources in the order they have been defined.
3. Note all problematic resources in the process: resources that could not be created because one of their dependencies have not been created previously.
4. Print the problematic resources into the output if necessary for the specific sequence.

7.11.3 Background

This problem challenges contestants to develop a program capable of efficiently determining the feasibility of creating AWS resources in a specified order based on their dependencies. The provided AWS CloudFormation template introduces participants to real-world scenarios encountered, for example, in the cloud infrastructure world. It aligns with actual challenges faced by professionals managing cloud resources, making it a relevant and valuable addition to a programming competition.

This problem can be efficiently solved using HashSet data structures. A HashSet is a data structure that implements the set abstract data type, which in itself represents a collection of distinct elements with no duplicates. The key characteristic of a HashSet is that it uses hashing to provide constant-time average complexity for basic operations such as `add`, `remove`, or `contains`.

Applications

HashSets have various applications in the programming world due to their efficient handling of unique elements and constant-time average complexity for many basic operations. Here are some common applications:

- **Removing duplicates:** A HashSet is useful for eliminating duplicate elements from a collection. By adding elements to a HashSet, duplicates are automatically filtered out.
- **Membership testing:** Checking whether a particular element is present in a collection can be done efficiently using a HashSet. This property in particular can be used to find whether a resource has been successfully created in the context of our original problem.
- **Caching:** HashSets are suitable for caching unique values, making it easy to check whether a value has been computed or retrieved before.
- **Set operations:** HashSet supports set operations such as union, intersection, and difference, which can be useful when implementing various algorithms.

7.11.4 Algorithmic Approach

The problem proposer has provided a solution to the problem implemented in the C# programming language. This solution uses HashSets to store the list of dependencies for a given resource, the list of resources that have been completed in the current sequence, as well as the list of problematic resources.

The program begins by reading the number of resources and the number of sequences from the first line read from the console input. After reading the number of resources, it then creates a Dictionary – a key-value mapping connecting resource names to their list of dependencies. The list of dependencies is stored as a separate HashSet for each resource type. The program then proceeds to read all resources into this Dictionary.

After reading the resources and their dependencies from the console input, the program is finally ready to process the sequences. For each sequence, the resources required to be created are read from the console input. The resources are tried in order. For each resource, if the list of dependencies for a given resource is a subset of the list of already created resources, the resource is created. This works because if the list of dependencies is a subset of the list of already created resources, it means that all the resources necessary for the creation of this particular resource have already been created. In other words, the dependencies for the resource are satisfied, and the creation of the resource can commence. Each resource that does not have their dependencies created is hereby placed into the `problematicResources` list.

Finally, the `problematicResources` list is populated with resources that could not be created. To fit the output scheme of the problem, a switch case can be used. There are three cases: no problematic resources, only one problematic resource, and multiple problematic resources. Each case has their own output style that has to be followed.

When all sequences are processed, the program can exit.

C# solution by the problem proposer

```
using System;
using System.Collections.Generic;
using System.Linq;

static void Main(string[] args) {
    // Read the first line of input containing
    // two integers separated by space
    int[] t = Array.ConvertAll(Console.ReadLine().Split(' '),
        item => Convert.ToInt32(item));
    int numberOfResources = t[0];
    int numberOfSequences = t[1];

    // Create a dictionary to store dependencies for each resource
    Dictionary<string, HashSet<string>> dependencies =
        new Dictionary<string, HashSet<string>>(numberOfResources);

    // Read input for each resource and its dependencies
    for (int i = 0; i < numberOfResources; i++) {
        // Read a line and split it using ':'
        // and ' ' as delimiters
        string[] line = Console.ReadLine()
            .Split(new char[] { ':', ' ' },
        StringSplitOptions.RemoveEmptyEntries);
        string resource = line[0];
```

```
        // Create a HashSet for the dependencies
        // of the current resource
        dependencies.Add(resource, new HashSet<string>(line.Skip(1)));
    }

    // Process each sequence of resources
    for (int i = 0; i < numberOfSequences; i++) {
        // Read a sequence of resources separated by space
        string[] sequence = Console.ReadLine().Split(' ');

        // Initialize sets and lists to track
        // completed and problematic resources
        HashSet<string> completedResources =
            new HashSet<string>(numberOfResources);
        List<string> problematicResources =
            new List<string>();

        // Iterate through each resource in the sequence
        for each (string resource in sequence) {
            // Get the dependencies of the current resource
            HashSet<string> currentDependencies = dependencies[resource];

            // Check if all dependencies are completed
            if (currentDependencies.IsSubsetOf(completedResources)) {
                // Mark the resource as completed
                completedResources.Add(resource);
            } else {
                // Add the resource to the problematic list
                problematicResources.Add(resource);
            }
        }

        // Determine the number of problematic resources
        // and print the result
        switch (problematicResources.Count) {
            case 0:
                Console.WriteLine("No problematic resources.");
                break;
            case 1:
                Console.WriteLine("Problematic resource: " +
                    problematicResources[0]);
                break;
            default:
                Console.WriteLine("Problematic resources: " +
                    String.Join(' ', problematicResources));
```

```
                break;
        }
    }
}
```

Racket solution by the problem proposer

```
#lang racket

(define (find-problematic-resources
         dependencies          ; hash table for dependencies
         sequence              ; sequence of resources
                               ; to be processed
         completed-resources   ; resources that can be
                               ; created
         problematic-resources ; resources that can not
                               ; be created
         )
  (cond
    [(null? sequence)
     (reverse problematic-resources)]
    [(subset?
      (list->set (hash-ref dependencies (car sequence)))
      (list->set completed-resources))
     (find-problematic-resources
      dependencies
      (cdr sequence)
      (cons (car sequence) completed-resources)
      problematic-resources)]
    [else
     (find-problematic-resources
      dependencies
      (cdr sequence)
      completed-resources
      (cons (car sequence) problematic-resources))]
    )
  )

(define (read-resources
         number-of-resources ; the number of resources
```

```
        )
  (for/hash ([lst (build-list number-of-resources values)])
    (let ([splitted-line
           (string-split
            (read-line (current-input-port) 'any))])
      (values (string-trim (car splitted-line) ":")
              (cdr splitted-line))
      )
    )
  )

(define (read-sequences
         number-of-sequences ; the number of sequences
                             ; to be processed
         dependencies        ; hash table for dependencies
         )
  (when (positive? number-of-sequences)
    (let ([problematic-resources
           (find-problematic-resources
            dependencies
            (string-split
             (read-line
              (current-input-port) 'any)) '() '())])
      (cond
        [(empty? problematic-resources)
         (displayln "No problematic resources.")]
        [(= 1 (length problematic-resources))
         (printf "Problematic resource: ~a~n"
                 (car problematic-resources))]
        [else
         (printf "Problematic resources: ~a~n"
                 (string-join problematic-resources))]
        )
      )
    (read-sequences (sub1 number-of-sequences) dependencies)
    )
  )

(let* ((splitted-first-line
```

```
        (map string->number
             (string-split
              (read-line (current-input-port) 'any))))
       (number-of-resources (car splitted-first-line))
       (number-of-sequences (cadr splitted-first-line))
       (dependencies (read-resources number-of-resources))
       )
  (read-sequences number-of-sequences dependencies)
  )
```

The below C code also implements the straightforward algorithm as suggested in the problem statement. For details, see the comments.

C code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

using namespace std;

typedef struct{
    char resource[21];
    int nrDependences;
    char** dependeces;
    int completedORproblematic; //-1:problematic, +1:completed
}ITEM;

int nrspaces( char *line ){
    int nr = 0, n = strlen(line);
    for( int i = 0 ; i < n ; ++i ){
        if( line[i] == ' ' ){ ++nr; }
    }
    return nr;
}

int cmp1( const void *p1, const void *p2 ){
    return strcmp( *(char**)p1, *(char**)p2 );
}

int cmp2( const void *p1, const void *p2 ){
    return strcmp( ((ITEM*)p1)->resource, ((ITEM*)p2)->resource );
}

int main(){
    //freopen("in.txt","r",stdin);
    int n,m;
    scanf("%i %i\n", &n, &m);
```

```
ITEM *a = (ITEM*)malloc(n*sizeof(ITEM));
char line[3000];
char *token;
for( int i = 0 ; i < n ; ++i ){
    scanf("%[^\n]\n", line);
    a[i].nrDependences = nrspaces(line);
    token = strtok(line, ": ");
    strcpy(a[i].resource,token);
    if( a[i].nrDependences == 0 ){
        continue;
    }
    a[i].dependeces = (char**)malloc(a[i].nrDependences*sizeof(char*));
    for( int j = 0 ; j < a[i].nrDependences ; ++j ){
        a[i].dependeces[j] = (char*)malloc(21*sizeof(char));
        token = strtok(NULL, " ");
        strcpy(a[i].dependeces[j],token);
    }
}
//we sort the array to allow efficient searching
//not an absolutely necessary step, given the small size of the input
qsort(a,n,sizeof(ITEM),cmp2);

//for all resource-sequences to be analyzed
for( int i = 0 ; i < m ; ++i ){
    //initially, all resources are considered as problematic ones
    for( int j = 0 ; j < n ; ++j ){
        a[j].completedORproblematic = -1;
    }
    ITEM temp1,temp2;
    int nrProblematic = 0;
    char ans[3000] = ""; //answer string containing the problematic resources
    //we assume that each sequence to be analyzed contains all resources
    for( int j = 0 ; j < n ; ++j ){
        scanf("%s",temp1.resource);
        //binary searching the current resource in the database
        ITEM *p = (ITEM *)bsearch(&temp1,a,n,sizeof(ITEM),cmp2);
        int k;
        //checking all dependences of the current resource
        for( k = 0 ; k < p->nrDependences ; ++k ){
            strcpy(temp2.resource,p->dependeces[k]);
            //binary searching the current dependence in the database
            ITEM *q = (ITEM *)bsearch(&temp2,a,n,sizeof(ITEM),cmp2);
            if( q->completedORproblematic == -1 ){
                 //an uncompleted dependence was found
                 break;
            }
        }
        if( k == p->nrDependences ) {
            //no uncompleted dependence was found
            //resources without dependences implicitly satisfy this condition
            p->completedORproblematic = 1;
        }
        else{//the current resource is problematic
            ++nrProblematic;
```

```
                //adding the problematic resource to ans
                if( nrProblematic > 1 ){
                    strcat(ans," ");
                }
                strcat(ans,p->resource);
            }
        }
        if( nrProblematic == 0 ){
            printf("No problematic resources.\n");
        }
        else if( nrProblematic == 1 ){
            printf("Problematic resource: %s\n", ans);
        }
        else{
            printf("Problematic resources: %s\n", ans);
        }
    }

    return 0;
}
```

7.12 Problem L

7.12.1 Problem Statement

Windmill lottery (suggested by Géza Vekov)

In $Windmillia$, there is a special national lottery system. Everyone can get a number and participate in countless draws. The numbers owned by the players are put in two lines ($Line1$ and $Line2$), based on the number specification, and they will belong to that line forever. The numbers and draws are generated based on the following rules:

- $LineHelper_n = (((La * LineHelper_{n-1} + Lb) * Lc) + Ld) \ mod \ (10^9 * 7)$
- $Line_n = LineHelper_n \ mod \ 3 + 1$
  The $Line_n$ number can be 1 or 2, which is used to insert a number into the specific line, or 3, which means performing a draw if it is possible.
- $Number_n = (((a * Number_{n-1} + b) * c) + d)$
  If $Line_n$ is 1 or 2, then $Number_n$ is used as a number. Else, $Number_n$ is used as $rotationCount$ for the draw.
- $Offset_i = (((La * Offset_{i-1} + Lb) * Lc) + Ld) \ mod \ 21 - 10$

Offset is computed only when $Line_n$ is 3.

A draw can take place at any time if the count of the numbers in the two lines have the same parity. A draw consists in:

- the two lines are being sorted.
- the middle points of $Line1$ and $Line2$ are placed on each other (in case of even count number, a virtual center is selected in both lines, between the two middle numbers).
- a rotation count is set ($rotationCount$), and $Line2$ is rotated around the middle point ($rotationCount$) times with 180°, while $Line1$ stays fixed.
- from now on, we consider the middle point being the middle point of the two lines if count of the elements in either is an odd number, otherwise middle point is the middle-left point if $Offset_i < 0$ and middle-right point if $Offset_i \geq 0$, in both lines.
- we take the two numbers with $Offset_i$ offset from the middle points, and calculate their sum modulo m, where m is the maximum value found already in the two lines. If a line is too short to have a corresponding position, we can take 0 for that number.
- the computed value will be the winning number of the draw.

Input

The first line contains values of a, b, c, d, and the starting value for $Number$ (five space-separated numbers). The second line contains values of La, Lb, Lc, Ld, and the starting value for $Number$ (five space-separated numbers). The third line contains the initial value of k and the value of n, the overall number of numbers generated, separated with a space.

Output

The output should contain the result of every draw, a line for every number.

Constraints

- $1 \leq lottery\ numbers \leq 10^{10}$
- $0 \leq lottery\ numbers\ count \leq 10^6$
- $-10 \leq Offset \leq 10$
- $0 \leq rotationCount \leq 10^{10}$
- $1 \leq a, b, c, d, La, Lb, Lc, Ld \leq 10^5$

Sample input

```
1 1 1 101 123
1 1 1 1 1
2 30
0
```

Sample output

```
54
```

Explanation: After generating 30 numbers, the two arrays are sorted, because a draw is generated. After sorting, they are placed in such a way that the middle points are in the same column.

*Line*1: 23 26 29 32 35 38 41 **44** 47 50 53 56 59 62 65

*Line*2: 25 28 31 34 37 40 43 **46** 49 52 55 58 61 64 67

The draw is generated with $offset$ -6 and *rotationCount* 68. Line 2 is rotated 68 times, and the -6^{th} elements relative to the middle points are added together: $(26 + 28) = 54(mod\ 67)$. The result is printed out. There was only one draw.

Maximal execution time: 2 seconds

7.12.2 The Key Idea

To successfully solve the lottery exercise, the fundamental approach involves emulating the lottery system's functioning as detailed in the problem statement. This includes adhering to the rules and guidelines presented while conducting the simulation. The process revolves around managing numbers, lines, rotations, offsets, and draw mechanisms outlined in the problem's algorithm.

One crucial step is to maintain the integrity of the two lines (Line 1 and Line 2) according to the specifications provided. This can be achieved by employing appropriate data structures such as priority queues (heaps) or arrays to organize and manage the numbers efficiently. Correctly placing numbers into the respective lines during the simulation is imperative.

Conducting draws involves implementing the rules described in the problem, which includes sorting lines, calculating rotations, determining middle points, and generating winning numbers based on offsets and rotations.

It is crucial to consider all constraints specified in the problem statement throughout the process of generating numbers, conducting draws, and managing lines.

Efficiently handling data structures and algorithmic implementations to manage the lottery simulation process is key to a successful solution. By adhering to the rules, using the appropriate data structures, and accurately following the outlined steps, one can derive the correct output based on the provided input.

7.12.3 Background

Heap data structure

Understanding heap data structures is pivotal for comprehending and successfully solving the given exercise. The exercise involves heap operations like insertion, deletion, and heapifying elements. A solid grasp of these concepts will greatly assist in implementing the required functionalities and comprehending the provided solution.

Overview

Heap data structures are fundamental in managing priority queues and efficiently implementing binary trees. There are two primary types of heaps: min heap and max heap. A min heap ensures that the parent node has a value less than or equal to its children's values, while a max heap maintains the parent node with a value greater than or equal to its children's values.

Operations

- **Heapify:** Reorganizes elements after insertions or deletions to maintain the heap property.
- **Insertion and Deletion:** Add or remove elements while preserving the heap structure and property.
- **Extract-Min or Extract-Max:** Removes the root node (minimum or maximum value) from the heap.

Applications in C++

Heap data structures find various applications in programming. Here is a C++ example using the standard library's heap functionalities:

Example in C++ using STL:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> heap {4, 1, 7, 3, 8, 5};

    // Converting vector to a min heap
    std::make_heap(heap.begin(), heap.end(), std::greater<int>());

    // Inserting an element into the heap
    heap.push_back(2);
    std::push_heap(heap.begin(), heap.end(), std::greater<int>());

    // Removing and retrieving the smallest element
    std::pop_heap(heap.begin(), heap.end(), std::greater<int>());
    int min_val = heap.back();
    heap.pop_back();

    // Displaying the resulting heap
    for (int val : heap) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Applications

Heap data structures, known for their efficient management of priority-based data, find extensive use across various domains due to their ability

to handle prioritization and resource management. Here are some real-life applications:

1. **Priority queues in operating systems:** Operating systems employ heap-based priority queues to manage processes based on their priority levels, ensuring that higher-priority processes receive resources first.
2. **Dijkstra's algorithm in pathfinding:** Dijkstra's algorithm for finding the shortest path in networks relies on min heaps to efficiently navigate graphs or networks.
3. **Job scheduling:** Various scheduling algorithms, such as Shortest Job First (SJF) or Priority Scheduling, use heap structures to prioritize and execute jobs.
4. **Network routing:** Systems managing network packet routing often use heap-based algorithms such as Weighted Fair Queuing (WFQ) to prioritize data packets for efficient transmission.
5. **Memory management:** In programming languages like C++, heap memory management utilizes structures similar to heap data structures for dynamic memory allocation and deallocation.
6. **Event-driven simulations:** Simulation systems model events using priority queues, sorting and executing events based on their timestamps or priorities.
7. **Garbage collection:** Certain garbage collection algorithms use heap-like data structures to efficiently identify and free up memory that is no longer in use.

These applications illustrate the versatility and significance of heap data structures across multiple fields, showcasing their role in optimizing processes, task prioritization, and resource allocation.

7.12.4 Algorithmic Approach

This code simulates a unique national lottery system in a fictional country called Windmillia. Let us break down the key components and operations of this lottery system:

Main Concepts

- **Lottery lines**: The lottery system involves two lines: Line 1 and Line 2. Numbers are placed into these lines based on specific rules.
- **Number generation**: Numbers are generated using a set of rules involving arithmetic operations.
- **Draw process**: A draw is conducted under certain conditions, which involves sorting the numbers, rotating Line 2 around Line 1, and computing a winning number.

Operations

- **Insertion and Sorting**: The `insert` function places a generated number into the appropriate line based on the current rules. The numbers in both lines are sorted using min-max heaps.
- **Drawing process**: The `draw` function handles the draw process. It computes the winning number by performing rotations and finding specific numbers based on the draw conditions.

Algorithm overview

1. **Number generation**: Numbers are generated using specific arithmetic operations (`pb_max`, `pb_min`, `pop_max`, `pop_min`).
2. **Insertion into lines**: Based on certain conditions, newly generated numbers are inserted into Line 1 or Line 2 and sorted accordingly.
3. **Drawing process**: When a draw is conducted, the numbers in both lines are sorted. Then, Line 2 is rotated around Line 1, and specific numbers are chosen based on the draw rules.
4. **Final result**: The winning number is computed based on the drawn numbers and the set criteria.

Understanding the code

- The `main` function initiates the lottery process, reads input values, and executes the number generation and draw operations.
- Functions like `pb_max`, `pb_min`, `pop_max`, and `pop_min` handle the min-max heap operations for sorting numbers.

– The code uses various arithmetic operations to generate, sort, and determine the winning numbers.

Input and output

– The code reads input values that define the rules, number generation criteria, and draw conditions.
– The output of the code is the winning number of each draw.

Conclusions

The below code simulates a complex lottery system with specific rules for number generation, sorting, and determining winning numbers. It showcases the use of data structures and algorithms to manage and process lottery entries. Understanding and executing the code requires a solid grasp of heap data structures, arithmetic operations, and conditional logic.

Author's C++ code

```
#include <iostream>
#include <fstream>
#include <queue>
#include <iomanip>
#include <string>
#include <ctime>
#define LL unsigned long long int

using namespace std;

LL kicsi1[500002], nagy1[500002];
LL kicsi2[500002], nagy2[500002];
int nk1,nk2, nn1, nn2;

void pb_max(LL kicsi[], int &n, LL x){
    kicsi[n++] = x;
    int i = n-1;
    while (i > 0 && kicsi[(i-1)>>1] < x){
        kicsi[i] = kicsi[(i-1)>>1];
        kicsi[(i-1)>>1] = x;
        i = (i-1)>>1;
    }
```

```
}

void hp_max(LL kicsi[], int n, int i){
    bool was_switch = true;
    while (was_switch){
        was_switch = false;
        int maxx = i;
        if ((i<<1) + 1 < n && kicsi[(i<<1) + 1] > kicsi[maxx])
                maxx = (i<<1) + 1;
        if ((i<<1) + 2 < n && kicsi[(i<<1) + 2] > kicsi[maxx])
            maxx = (i<<1) + 2;
        if (maxx != i){
            LL t = kicsi[i];
            kicsi[i] = kicsi[maxx];
            kicsi[maxx] = t;
            was_switch = true;
        }
        i = maxx;
    }
}

void pop_max(LL kicsi[], int &n){
    kicsi[0] = kicsi[--n];
    hp_max(kicsi, n, 0);
}

LL top_max(LL kicsi[], int n){
    return kicsi[0];
}

void pb_min(LL kicsi[], int &n, LL x){
    kicsi[n++] = x;
    int i = n-1;
    while (i > 0 && kicsi[(i-1)>>1] > x){
        kicsi[i] = kicsi[(i-1)>>1];
        kicsi[(i-1)>>1] = x;
        i = (i-1)>>1;
    }
}

void hp_min(LL kicsi[], int n, int i){
    bool was_switch = true;
```

```
    while (was_switch){
        was_switch = false;
        int maxx = i;
        if ((i<<1) + 1 < n && kicsi[(i<<1) + 1] < kicsi[maxx])
                maxx = (i<<1) + 1;
        if ((i<<1) + 2 < n && kicsi[(i<<1) + 2] < kicsi[maxx])
            maxx = (i<<1) + 2;
        if (maxx != i){
            LL t = kicsi[i];
            kicsi[i] = kicsi[maxx];
            kicsi[maxx] = t;
            was_switch = true;
        }
        i = maxx;
    }
}

void pop_min(LL kicsi[], int &n){
    kicsi[0] = kicsi[--n];
    hp_min(kicsi, n, 0);
}

LL top_min(LL kicsi[], int n){
    return kicsi[0];
}

LL mx = 0;

void insert(LL x, int line){
    mx = max(mx, x);
    if (line){
        if (!nk2 || x<top_max(kicsi2,nk2))
            pb_max(kicsi2,nk2,x);
        else
            pb_min(nagy2,nn2,x);

        if (nk2 > nn2 + 1){
            x=top_max(kicsi2,nk2);
            pop_max(kicsi2,nk2);
            pb_min(nagy2,nn2,x);
        } else if (nn2 > nk2 + 1){
            x=top_min(nagy2,nn2);
```

```
            pop_min(nagy2, nn2);
            pb_max(kicsi2,nk2,x);
        }
    }
    else{
        if (!nk1 || x<top_max(kicsi1,nk1))
            pb_max(kicsi1,nk1,x);
        else
            pb_min(nagy1,nn1,x);

        if (nk1 > nn1 + 1){
            x=top_max(kicsi1,nk1);
            pop_max(kicsi1,nk1);
            pb_min(nagy1,nn1,x);
        } else if (nn1 > nk1 + 1){
            x = top_min(nagy1,nn1);
            pop_min(nagy1,nn1);
            pb_max(kicsi1,nk1,x);
        }
    }

}

LL popped[10];
int poppedCount;

LL pop_from1(bool isSmall, int n){
    poppedCount = 0;
    LL ret;
    if (isSmall){
        while (poppedCount<n && nk1){
            popped[poppedCount++] = top_max(kicsi1,nk1);
            pop_max(kicsi1,nk1);
        }
        ret = (poppedCount == n && nk1) ? top_max(kicsi1,nk1) : 0;
        while (poppedCount){
            pb_max(kicsi1,nk1,popped[--poppedCount]);
        }
    }
    else {
        while (poppedCount<n && nn1){
            popped[poppedCount++] = top_min(nagy1,nn1);
```

```
            pop_min(nagy1,nn1);
        }
        ret = (poppedCount == n && nn1) ? top_min(nagy1,nn1) : 0;
        while (poppedCount){
            pb_min(nagy1,nn1,popped[--poppedCount]);
        }
    }
    return ret;
}

LL pop_from2(bool isSmall, int n){
    poppedCount = 0;
    LL ret;
    if (isSmall){
        while (poppedCount<n && nk2){
            popped[poppedCount++] = top_max(kicsi2,nk2);
            pop_max(kicsi2,nk2);
        }
        ret = (poppedCount == n && nk2) ? top_max(kicsi2,nk2) : 0;
        while (poppedCount){
            pb_max(kicsi2,nk2,popped[--poppedCount]);
        }
    }
    else {
        while (poppedCount<n && nn2){
            popped[poppedCount++] = top_min(nagy2,nn2);
            pop_min(nagy2,nn2);
        }
        ret = (poppedCount == n && nn2) ? top_min(nagy2,nn2) : 0;
        while (poppedCount){
            pb_min(nagy2,nn2,popped[--poppedCount]);
        }
    }
    return ret;
}

int draw(int offset, LL rotationCount){
    LL ret;
    bool rotated = rotationCount & 1;
    if (offset>0){
        if (nn1>=nk1) ret = pop_from1(false,offset);
        else ret = pop_from1(false,offset-1);
```

```
        if (!rotated){
            if (nn2>=nk2) ret += pop_from2(false,offset);
            else ret += pop_from2(false,offset-1);
        }
        else {
            if (nk2>=nn2) ret += pop_from2(true,offset);
            else ret += pop_from2(true,offset-1);
        }
    } else {
        if (offset == 0){
            if (nn1>=nk1) ret = pop_from1(false,0);
            else ret = pop_from1(true,0);
            if (!rotated) {
                if (nn2>=nk2) ret += pop_from2(false,0);
                else ret += pop_from2(true,0);
            } else {
                if (nk2>=nn2) ret += pop_from2(true,0);
                else ret += pop_from2(false,0);
            }
        }
        else{
            if (nk1>=nn1) ret = pop_from1(true,-offset);
                else ret = pop_from1(true,-offset-1);
                if (!rotated){
                    if (nk2>=nn2) ret += pop_from2(true,-offset);
                    else ret += pop_from2(true,-offset-1);
                } else {
                    if (nn2>=nk2) ret += pop_from2(false,-offset);
                    else ret += pop_from2(false,-offset-1);
                }
        }
    }

    return ret % mx;
}

int main(){
    LL a=2,b=3,c=4,d=1000007, value=11111025;
    int line=123, k=8;
    int la=123, lb=1123, lc=123, ld=3;
    int n=1000000;
LL op = 22;
```

```
cin >> a >> b >> c >> d >> value;
cin >> la >> lb >> lc >> ld >> op;
cin >> k >> n;

while (n)
{
op = (((la*op + lb)*lc)+ld) % 1000000007;
        line = op % 3 + 1;

value = ((a*value + b)*c % d);
if (line == 3)
{
if (!(n&1)){
k = (((k*la+lb)*lc)+ld)%21-10;
cout << draw(k,value) << endl;
}
}
else
{
insert(value, line - 1);
n--;
}
}

value = ((a*value + b)*c % d);
k = (((k*la+lb)*lc)+ld)%21-10;
cout << draw(k,value) << endl;

    return 0;
}
```

A simpler approach could involve storing the two lines as multisets and utilizing two iterators pointing to their middle elements. The implementation below is constructed based on this concept. The included comments detail further aspects of the implementation in line with the problem specification.

Implementation using multisets

```
#include <iostream>
#include <set>

#define ULL unsigned long long

using namespace std;

//lines are stored as multisets to guaranty the continuously sorted order
multiset<ULL> mset1, mset2;
//to store the cardinality of the sets
int n1 = 0, n2 = 0;
//iterators pointing to the middle elements of the lines
//in case of even length lines the implicit middle is the left-middle
multiset<ULL>::iterator middle1,middle2;

//global maximum with respect to the elements of the sets
ULL MAX = 0;

void draw(int Offset, ULL rotationCount);

int main(){
    int n, Offset;
    int a,b,c,d;
    int La,Lb,Lc,Ld;
    ULL LineHelper,Number;
    //freopen("inL.txt","r",stdin);
    //freopen("outL.txt","w",stdout);
    cin >> a >> b >> c >> d >> Number;
    cin >> La >> Lb >> Lc >> Ld >> LineHelper;
    cin >> Offset >> n;
    //middles of the initially empty sets
    middle1 = mset1.begin();
    middle2 = mset2.begin();
    while( n ){
        LineHelper = (((La * LineHelper + Lb) * Lc) + Ld) % (1000000007);
        int Line = LineHelper % 3 + 1;
        Number = (((a * Number + b) * c) % d);
        if( Line == 1 ){
            //Number is inserted in mset1 and middle1 is corrected correspondingly
            MAX = max(MAX,Number);
            mset1.insert(Number);
            //correcting the iterator pointing to the middle element
            if( n1 == 0 ){ ++middle1; } //after inserting the first element
            else{
                if( n1 % 2 && Number < *middle1 ){ --middle1; }
                if( !(n1 % 2) && Number >= *middle1 ){ ++middle1; }
            }
            --n; ++n1;
        }
        else if( Line == 2 ){
            //Number is inserted in mset2 and middle2 is corrected correspondingly
```

```
            MAX = max(MAX,Number);
            mset2.insert(Number);
            //correcting the iterator pointing to the middle element
            if( n2 == 0 ){ ++middle2; } //after inserting the first element
            else{
                if( n2 % 2 && Number < *middle2 ){ --middle2; }
                if( !(n2 % 2) && Number >= *middle2 ){ ++middle2; }
            }
            --n; ++n2;
        }
        else{
            //a draw is performed, if the parity of the lines equals
            if( n1%2 != n2%2 ){ continue; }
            Offset = (((La * Offset + Lb) * Lc) + Ld) % 21 - 10;
            //for even/odd numbers we perform 0/1 rotation
            draw(Offset,Number%2);

        }
    }
    //extra draw at the end
    Offset = (((La * Offset + Lb) * Lc) + Ld) % 21 - 10;
    Number = (((a * Number + b) * c) % d);
    draw(Offset,Number%2);

    return 0;
}

void draw(int Offset, ULL rotationCount){
    ULL nr1,nr2; //the two numbers resulting from the current draw
    multiset<ULL>::iterator i1 = middle1;
    multiset<ULL>::iterator i2 = middle2;
    //three cases when we use right-middle
    if( n1 % 2 == 0 && Offset >= 0 ){ ++i1; } //in line1
    if( n2 % 2 == 0 && Offset < 0 && rotationCount == 1 ){ ++i2; } //in line2
    if( n2 % 2 == 0 && Offset >= 0 && rotationCount == 0 ){ ++i2; } //in line2

    // in case of odd parity
    if( n1 % 2 == 1 ){
        //line1
        //"too large" offset
        if( (int)abs(Offset) > n1/2 ){ nr1 = 0; }
        //moving it1 to the target number
        else { advance(i1,Offset); nr1 = *i1; }
        //line2
        //"too large" offset
        if( (int)abs(Offset) > n2/2 ){ nr2 = 0; }
        //moving it2 to the target number
        else{
            if( rotationCount == 1 ){
                advance(i2,-Offset); nr2 = *i2;
            }
            else{
                advance(i2,Offset); nr2 = *i2;
            }
```

```
        }
    }
    // in case of even parity
    else{
        //line1
        //"too large" offset
        if( Offset < 0 && n1/2-1+Offset < 0
            || Offset >= 0 && n1/2+Offset > n1-1 ) {
                nr1 = 0; }
        //moving it1 to the target number
        else { advance(i1,Offset); nr1 = *i1; }
        //line2
        if( rotationCount == 1 ){
            //"too large" offset
            if( Offset < 0 && n2/2-Offset > n2-1
                || Offset >= 0 && n2/2-1-Offset < 0 ) {
                    nr2 = 0; }
            //moving it2 to the target number
            else { advance(i2,-Offset); nr2 = *i2; }
        }
        else{
            //"too large" offset
            if( Offset < 0 && n2/2-1+Offset < 0
                || Offset >= 0 && n2/2+Offset > n2-1 ) {
                    nr2 = 0; }
            //moving it2 to the target number
            else { advance(i2,Offset); nr2 = *i2; }
        }
    }
    cout << (nr1 + nr2) % MAX << endl;
}
```

7.13 Problem M

7.13.1 Problem Statement

Modputer (suggested by Zsolt Németh)

Modputer is a strange mathematical computing machine that works as follows. The machine has an internal state represented by an array A of N positive integers. The input of the machine is a list of M positive integers, each greater than 1. The machine reads the integers on the input one by one. For each integer D in the input, the state of the machine gets updated: every integer in the state array which is divisible by D increases by one. It is

difficult to simulate the transitions of *Modputer* using a regular computer. Your task is to count the number of times each integer in the state array got updated while processing the input of the machine and report the total number of changes. More formally, let $A[i]$ denote the value of the i-th element of the state array before reading the input, and $B[i]$ denote the value of the same element after processing the input. You must calculate the sum of $B[i] - A[i]$ over all i.

Input specification

The first line of the input contains N and M, number of elements in the state array and the length of the input list ($1 \leq N, M \leq 100000$). Each of the next N lines contains an integer $A[i]$, the initial values present in the state array ($1 \leq A[i] \leq 300000$). Each of the next M lines contains an integer $D[i]$, the input values ($2 \leq D[i] \leq 300000$).

Output specification

Print the total number of changes in the values of the state array while processing the input.

Sample input

```
3 5
10
11
12
2
11
4
13
2
```

Sample output

```
12
```

Explanation

- The initial state array is $[10, 11, 12]$.
- After reading the value $D[0] = 2$, the state changes to $[11, 11, 13]$.
- After reading the value $D[1] = 11$, the state changes to $[12, 12, 13]$.
- After reading the value $D[2] = 4$, the state changes to $[13, 13, 13]$.
- After reading the value $D[3] = 13$, the state changes to $[14, 14, 14]$.
- After reading the value $D[4] = 2$, the state changes to $[15, 15, 15]$.
- The first state value was updated 5 times, the second 4 times, and the third 3 times. The total number of changes is $5 + 4 + 3 = 12$.

7.13.2 The Key Idea

The Modputer, a mathematical computing machine, has an internal state represented by an array A of N positive integers. For each positive integer D in the input list, the machine updates its state: every integer in the state array divisible by D increases by one. The task is to count the number of times each integer in the state array is updated while processing the input and report the total number of changes, calculated as the sum of $B[i]-A[i]$ over all i, where $A[i]$ denotes the initial value of the i-th element of the state array, and $B[i]$ represents the updated value of the same element.

The key idea behind solving the Modputer problem involves simulating the machine's operations by maintaining data structures to track state changes. This simulation entails iterating through the state array elements, updating values based on the input, and employing a recursive function to compute changes while memorizing intermediate results. The solution ultimately calculates and outputs the total number of updates made to the state array during the Modputer's operation.

7.13.3 Background

Understanding certain fundamental concepts in programming is essential to tackle the Modputer problem effectively. Proficiency in concepts like Depth-First Search (DFS), recursion, and utilizing data structures such as sets, vectors, and maps in C++ is crucial for comprehending and implementing the solution efficiently. DFS aids in navigating through the state array based on input values, recursion facilitates a concise traversal mechanism, while sets, vectors, and maps from C++'s Standard Template Library

(STL) play a pivotal role in managing and manipulating data structures, integral to solving the Modputer challenge.

Depth-First Search (DFS):

DFS is an algorithm used to traverse graphs or trees deeply in a systematic manner. It starts at a root node and explores as far as possible along each branch before backtracking. DFS can be implemented using either recursion or an explicit stack. In the context of the Modputer problem, DFS can be employed to traverse and update the state array based on input values.

Recursion:

Recursion is a programming technique where a function calls itself directly or indirectly. It is commonly used in DFS to traverse nodes or elements in a depth-first manner. In DFS-based algorithms, recursion allows for a more concise and natural expression of the traversal process. However, excessive recursion can lead to stack overflow errors, so iterative alternatives might be preferred for very large datasets.

Set, Vector, Map in C++:

These are fundamental data structures in C++'s Standard Template Library (STL), providing essential functionalities for storing and managing data.

- **Set:** A set is a container that holds a collection of unique, sorted elements. It is useful for maintaining a sorted collection of values and ensures uniqueness automatically. Sets are implemented using self-balancing binary search trees like red-black trees.
- **Vector:** A vector is a dynamic array that can change in size. It provides access to elements using indexes, and its size can dynamically grow or shrink. Vectors offer constant-time random access to elements and efficient appending at the end.
- **Map:** A map is a container that stores key–value pairs. It allows fast lookup, insertion, and deletion of elements based on a key. Maps are implemented as binary search trees or hash tables, enabling efficient key-based retrieval.

Understanding these concepts is crucial for efficiently managing data, performing traversals, and structuring algorithms in C++ programs, particularly in the context of Modputer, where efficient data handling is essential for processing and updating the state array based on input values.

Applications

- **Depth-First Search (DFS):** DFS finds extensive usage in various fields such as network analysis, maze-solving algorithms, web crawling, and mapping pathways in GPS systems. In network analysis, DFS helps explore connections between nodes or vertices. It is also utilized in game development for solving puzzles or traversing game maps.
- **Recursion:** Recursion is employed in many programming applications, including traversing data structures (like trees and graphs), searching algorithms, and solving problems involving self-referential systems. It is used in algorithms like binary search, factorial calculations, and traversing directories in file systems.
- **Set, Vector, Map in C++:** These data structures from C++'s Standard Template Library (STL) are widely used in programming and real-world applications. Sets are used to maintain unique elements, vectors for dynamic arrays offering constant-time access, and maps for key–value pair associations, commonly utilized in data management, sorting, and searching applications.

7.13.4 Algorithmic Approach

The solution involves simulating the operations of the Modputer while updating and tracking changes in the state array. To achieve this, the solution maintains several data structures:

- Use a vector A to store the initial state array values.
- Store the input values in a vector D.
- Utilize a vector of maps mp to memoize the state changes at each step based on the current value and the processing stage.
- Employ a vector of sets ts to efficiently find the next indexes for each divisor present in the input.
- Create a vector of vectors $divs$ to store divisors for efficient lookup.

The steps of the solution include:

1. Read the input values and set up the necessary data structures.

2. Iterate through the state array elements and simulate the Modputer's operations.
3. Use a recursive function `dfs` to navigate through the state changes while memoizing intermediate results.
4. Compute the total number of changes in the state array by subtracting the initial values from the updated values using the `dfs` function.

Finally, output the total number of changes obtained from the simulation.

Author's C++ code

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <algorithm>

using namespace std;

int N, M;
vector<int> A, D;
vector<map<int,int>> mp;
vector<set<int>> ts;
vector<vector<int>> divs;

int get_next(int id, int val) {
    auto it = ts[val].lower_bound(id);
    if (it == ts[val].end())
        return M;
    return *it;
}
int dfs(int id, int val) {
    if (id == M) return val;
    if (mp[id].count(val))
        return mp[id][val];

    int nxtid = M;
```

```
    for (int g : divs[val])
        nxtid = min(nxtid, get_next(id, g));
    if (nxtid == M) return val;

    int res = dfs(nxtid + 1, val + 1);
    mp[id][val] = res;
    return res;
}
int main() {
    ios::sync_with_stdio(false);
    cin >> N >> M;
    A.resize(N);
    for (int i=0; i<N; ++i) cin >> A[i];
    D.resize(M);
    for (int i=0; i<M; ++i) cin >> D[i];

    mp.resize(M);
    int sz = *max_element(A.begin(), A.end()) + M + 1;
    ts.resize(max(sz, 1 + *max_element(D.begin(), D.end())));
    for (int i=0; i<M; ++i) {
        ts[D[i]].insert(i);
    }
    divs.assign(sz, {});
    for (int i=2; i<sz; ++i) {
        for (int j=i; j<sz; j+=i)
            divs[j].push_back(i);
    }

    long long ans = 0ll;
    for (int i=0; i<N; ++i) {
        ans += dfs(0, A[i]) - A[i];
    }
    cout << ans << endl;

    return 0;
}
```

7.13.5 Subsequent Lessons

It can also be instructive to review solutions that would not have been satisfactory given the size of input. The below naive approach would have been hopelessly slow:

```
long long ans = 0ll;
for (int i=0; i<N; ++i) {
    B[i] = A[i];
    for (int j=0; j<M; ++j) {
        if( B[i] % D[j] == 0 ){ ++B[i]; }
    }
    ans += B[i] - A[i];
}
cout << ans << endl;
```

The next version is improved, but not sufficiently. The example provided by the author suggests that there can be numerous overlapping subproblems. This suggests a dynamic programming approach. We define an array $C[0 \dots M][min(A) \dots max(A) + M]$, where cell $c[i][j]$ stores how much the value j in array A can be increased due to the updates dictated by the $D[i \dots M]$ array segment:

- if j is divisible by $D[i]$: $c[i][j] = 1 + c[i+1][j+1]$;
- if $D[i]$ does not divide j: $c[i][j] = 0 + c[i+1][j]$;
- obviously, $c[M][j] = 0$;
- the result will be $\sum_{n=0}^{N-1} C[0][A[i]]$;

The fact that the traces (from row 0 to row M) intersect and overlap (see *Figure 7.12*) illustrates that numerous overlapping subtasks may occur across different branches. By storing the results of previously solved tasks, we avoid repeatedly solving identical subproblems.

Remarks:

- Excellent, but we cannot allocate memory for an array with $100,000 \cdot 300,000$ elements!
- Even if we were to store only one row and build the result row by row, we would not improve the time complexity.
- We can only hope that the matrix is sparse, especially if we consider the green cells alone.

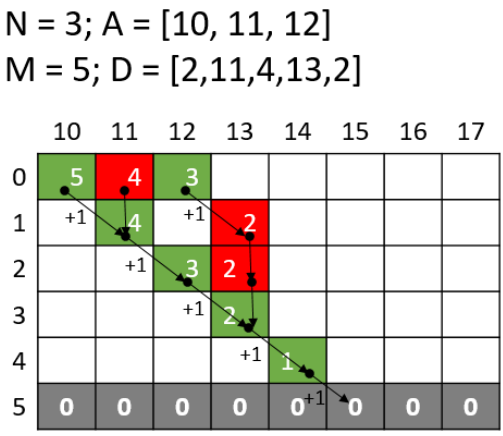


Figure 7.12. Storing the results of previously solved tasks

Consequently, we will use a dynamic data structure to store the cells of array C (the indexes and the corresponding value), a vector of maps:

```
vector<map<int,int>> C; C.resize(M);
```

The below solution for large inputs will likely result in a stack overflow because of the overly large data structures allocated on the stack.

```
int dfs0(int i, int j) {
    if (i == M) { return 0; }
    if ( C[i][j] ) { return C[i][j]; }
    if ( j % D[i] == 0 ) {
        return C[i][j] = 1 + dfs0(i + 1, j + 1);
    }
    else {
        return C[i][j] = dfs0(i + 1, j);
    }
}

...
long long ans = 0ll;
for (int i=0; i<N; ++i) {
    ans += dfs0(0, A[i]);
}
cout << ans << endl;
```

A naive method to avoid storing red cells is by simply skipping them. The below approach can yield correct results even for large inputs ($N = 100,000$, $M = 100,000$), but it is too slow!

```
int dfs1(int i, int j) {
    if (i == M) { return 0; }
    if ( C[i][j] ) { return C[i][j]; }
    while ( i < M && j % D[i] ) { ++i; } //skipping red cells
    if (i == M) { return 0; }
    if ( C[i][j] ) { return C[i][j]; }
    return C[i][j] = 1 + dfs1(i + 1, j + 1);
}
```

The key ideas used by the author are:

– avoid divisibility checking by precomputing and storing all divisors of all values from interval $[min(A), max(A) + M)$ (*divsA*: vector of vectors) and storing the positions (indexes) at which each value appears in array D (*indsD*: vector of sets) (see *Figure 7.13*);

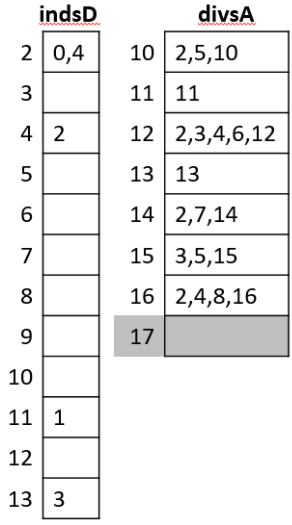


Figure 7.13. divsA: precomputing and storing all divisors. indsD: storing the indexes. For memory-saving purposes, we store the intervals offset by minD/minA (see the below code).

– avoid red cells by performing single, binary-search-accelerated jumps (see *Figure 7.14*).

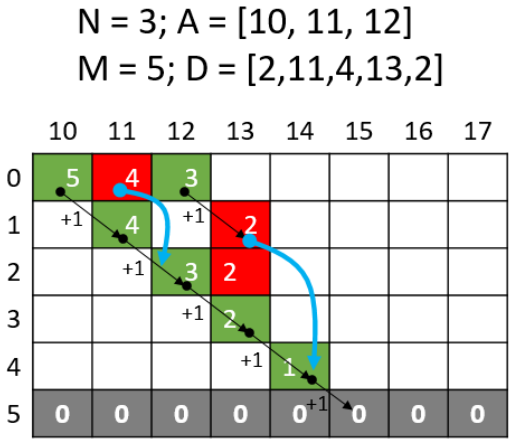


Figure 7.14. Avoiding red cells by performing single jumps.

Notice that:

– value $max(A) + M$ can only be reached in the final step as a result of the last update;
– we are interested in all divisors greater than 1 (since $D[i] > 1$);
– the set elements of vector D can be searched binary.

The code resulting from the above train of thought, which, as already mentioned, closely follows the author's approach:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <set>
#include <map>
#include <algorithm>

using namespace std;

int N, M;
vector<int> A, D;
vector<map<int,int>> C;
```

```
vector<set<int>> indsD;
vector<vector<int>> divsA;
int minA, maxA, minD, maxD;

int get_next_i(int i, int div) {
    if ( div < minD || div > maxD ) { return M; }
    auto it = indsD[div-minD].lower_bound(i);
    if (it == indsD[div-minD].end())
        return M;
    return *it;
}

int dfs2(int i, int j) {
    if (i == M) { return 0; }
    if ( C[i][j] ) { return C[i][j]; }

    //preparing a binary search accelerated jump
    int next_i = M;
    for (int div : divsA[j-minA])
        next_i = min(next_i, get_next_i(i, div));

    if ( next_i == M ) return 0;
    return C[i][j] = 1 + dfs2(next_i + 1, j + 1);
}


int main() {
    ios::sync_with_stdio(false);

    freopen("inM.txt","r",stdin);
    cin >> N >> M;
    A.resize(N);
    for (int i=0; i<N; ++i) cin >> A[i];
    D.resize(M);
    for (int i=0; i<M; ++i) cin >> D[i];

    C.resize(M);

    minA = *min_element(A.begin(), A.end());
```

```
    maxA = *max_element(A.begin(), A.end());
    maxA += M;
    minD = *min_element(D.begin(), D.end());
    maxD = *max_element(D.begin(), D.end());

    indsD.resize(maxD - minD + 1);
    for (int i=0; i<M; ++i) {
        indsD[D[i]-minD].insert(i);
    }

    divsA.assign(maxA - minA + 1,{});
    for ( int i = 2 ; i < maxA ; ++i ) {
        for ( int j = i ; j < maxA ; j += i ){
            if ( j >= minA ){
                divsA[j-minA].push_back(i);
            }
        }
    }

    long long ans = 0ll;
    for (int i=0; i<N; ++i) {
        ans += dfs2(0, A[i]);
    }
    cout << "res: " << ans << endl;

    return 0;
}
```

7.14 Problem N

7.14.1 Problem Statement

Carpathian Riders (suggested by Zsolt Németh)

Carpathian Riders is a famous motorcycle gang and competitive programming club. They decided to participate in the upcoming ECN Sapientia

programming contest, travelling there on their bikes. To get to the contest venue, they must cross the Carpathian Mountains.

The Carpathians is a grid of R rows and C columns. Each cell has an elevation value between 0 and 1,000, inclusive. Some cells are not passable on motorbikes – those cells are marked with the elevation value -1.

The gang can enter the Carpathians at any passable cell on the western edge of the grid and leave the mountains at some cell on the eastern edge. They travel from the west to the east in multiple steps. In each step, they can move to the cell either directly to the east or diagonally to the northeast or diagonally to the southeast. Each cell that they visit must be passable.

		2	5	4	3	1
3	4	1	4	1	2	1
3	4	5	5	3	4	5
2	3	2	1	2	3	2
	5	4	1	4	4	2

Figure 7.15. A sample grid representing the Carpathian Mountains

The gang does not want to be late from the contest, so they try to avoid riding up to cells with high elevation: the sum of elevations during the trip must be minimized.

Sounds easy enough? Well, there is a catch. Since this is a motorcycle gang, they love visiting *mountain passes*. A mountain pass is a grid cell that has a strictly greater elevation value than the two cells to its east and west, but it has a strictly lower elevation value than the two cells to its north and south. Cells on the edges of the grid cannot be mountain passes, similarly to cells adjacent to impassable cells. In the example above, all cells that are mountain passes are shaded in grey.

The bikers decided that they want to visit **exactly** P passes along the trip. Your task is to help them planning a trip from the west to the east which contains exactly P passes, avoids impassable cells, and the sum of elevations is minimized.

Input specification

The first line of the input contains the number of rows and columns R and $C (3 \leq R, C \leq 500)$ and the exact number of passes to visit $P (0 \leq P \leq 10)$.

Each of the next R lines contains C elevation values. Impassable locations are represented by -1, and all other elevations are between 0 and 1,000, inclusive. There is at least one cell on the western and one cell on the eastern edge of the grid that is passable.

Output specification

Print the sum of elevations along an optimal trip with exactly P passes. If no such trip exists, output the string *impossible*.

Sample input - 1

```
5 7 2
-1 -1 2 5 4 3 1
3 4 1 4 1 2 1
3 4 5 5 3 4 5
2 3 2 1 2 3 2
-1 5 4 1 4 4 2
```

Sample output - 1

```
14
```

Sample input - 2

```
4 3 1
3 4 5
2 4 2
1 5 4
1 1 1
```

Sample output - 2

```
impossible
```

Explanation

The first example corresponds to the example in the statement. An optimal trip consists of the cells with elevations $2-3-2-1-3-2-1$.

In the second example, there are no mountain passes in the grid, so it is not possible to plan a trip that contains exactly 1 pass.

7.14.2 The Key Idea

The Carpathian Riders problem presents a grid-based traversal challenge with specific constraints. The key idea revolves around employing dynamic programming (DP) in combination with backtracking techniques. It involves exploring various paths through the grid, considering elevations and passable cells while aiming to visit a predetermined number of passes. By optimizing the path's traversal and storing intermediate results using DP, the algorithm aims to minimize the sum of elevations while fulfilling the pass visitation criteria.

7.14.3 Background

DFS and backtracking

Depth-First Search (DFS) is an algorithmic approach used to explore nodes or cells deeply in a graph or grid structure. It operates by traversing as far as possible along a branch before backtracking to previous nodes to explore other potential paths. Backtracking, an essential part of DFS, involves returning to previously visited nodes to explore alternate routes when reaching dead ends. In the Carpathian Riders problem, DFS-like traversal helps to explore feasible paths while adhering to constraints.

Dynamic programming

Dynamic Programming (DP) is a problem-solving technique used to break down complex problems into smaller subproblems, solving each subproblem only once and storing its solution. It involves reusing these computed results to optimize time and space complexities, especially when subproblems overlap or recur. DP is instrumental in storing the results of subproblems related to traversing the grid, facilitating optimal path calculation.

Grid traversal

Grid traversal involves efficiently moving through a grid structure, considering cell properties such as elevation values and passability. In the context of Carpathian Riders, traversing the grid from west to east while adhering to the constraints of passable cells and elevation values is crucial. An optimal path must be determined, minimizing the total elevation while visiting the required number of mountain passes.

Data structures in C++

The provided C++ code utilizes essential data structures such as 2D vectors for grid representation, storing elevation values, and DP. It employs boolean conditions to verify cell passability and elevation values, determining the optimality of the traversed path. The chosen data structures facilitate efficient storage, access, and manipulation of grid-related information, aiding in optimal path determination.

Applications

The concepts highlighted in the Carpathian Riders problem find applications across various domains. Pathfinding algorithms, inspired by DFS and backtracking, are fundamental in route planning for navigation systems or logistics. In real-world scenarios, these algorithms optimize vehicle routes across diverse terrains, considering elevation changes and minimizing travel time. Additionally, the application of DP extends to resource optimization problems in computational domains, facilitating the efficient usage of limited resources or identifying optimal solutions in various contexts.

7.14.4 Algorithmic Approach

The provided C++ code tackles the Carpathian Riders problem using dynamic programming combined with backtracking. It initializes by reading the grid's dimensions and the number of passes to visit and stores the elevation values in the "elevations" 2D vector. The "DP" vector is established to cache previously computed results for dynamic programming.

The is_pass function determines whether a cell qualifies as a mountain pass by examining its surrounding elevation values. In the "solve" function,

dynamic programming is employed to determine the minimum sum of elevations while ensuring the specified number of passes. The function uses backtracking to traverse the grid, exploring various paths while considering pass visitation and elevation constraints.

The "main" function initializes the $best_path$ variable to an initial maximum value. It proceeds to explore different starting positions along the western edge of the grid. For each starting position, it calculates the minimum sum of elevations using the "solve" function. If a valid path adhering to the required number of passes exists, $best_path$ stores the minimum elevation sum.

Finally, the code outputs the optimal minimum elevation sum if a feasible path meeting the constraints exists; otherwise, it prints "impossible."

This elaboration explains the code's structure, functions, and purpose in solving the Carpathian Riders problem using dynamic programming and backtracking, including the comments within the code.

Author's C++ code

```
#include <iostream>
#include <vector>

using namespace std;

const int ILLEGAL_VALUE = 1e9;
int R, C, P;
vector<vector<int>> elevations;
vector<vector<vector<int>>> DP;

bool is_pass(int r, int c) {
  if (r == 0 || c == 0 || r == R-1 || c == C-1)
    return false;
  if (elevations[r][c] == -1 || elevations[r-1][c] == -1 ||
     elevations[r+1][c] == -1 || elevations[r][c-1] == -1 ||
     elevations[r][c+1] == -1)
    return false;
  if (elevations[r-1][c] > elevations[r][c] &&
     elevations[r+1][c] > elevations[r][c] &&
     elevations[r][c-1] < elevations[r][c] &&
```

```
     elevations[r][c+1] < elevations[r][c])
    return true;
  else
    return false;
}

int solve(int r, int c, int p) {
  if (p < 0)
    return ILLEGAL_VALUE;
  if (c == C-1) {
    if(p == 0)
      return 0;
    else
      return ILLEGAL_VALUE;
  }
  if(DP[r][c][p] != -2)
    return DP[r][c][p];

  // NE
  int best_score = ILLEGAL_VALUE;
  int cur;
  if (r > 0 && elevations[r-1][c+1] != -1) {
    if (is_pass(r-1, c+1))
      cur = solve(r-1, c+1, p-1) + elevations[r-1][c+1];
    else
      cur = solve(r-1, c+1, p) + elevations[r-1][c+1];
    if (cur < best_score)
      best_score = cur;
  }
  // E
  if (elevations[r][c+1] != -1) {
    if (is_pass(r, c+1))
      cur = solve(r, c+1, p-1) + elevations[r][c+1];
    else
      cur = solve(r, c+1, p) + elevations[r][c+1];
    if (cur < best_score)
      best_score = cur;
  }
  // SE
```

```
  if (r < R-1 && elevations[r+1][c+1] != -1) {
    if (is_pass(r+1, c+1))
      cur = solve(r+1, c+1, p-1) + elevations[r+1][c+1];
    else
      cur = solve(r+1, c+1, p) + elevations[r+1][c+1];
    if (cur < best_score)
      best_score = cur;
  }

  if(best_score > ILLEGAL_VALUE)
    best_score = ILLEGAL_VALUE;
  DP[r][c][p] = best_score;
  return best_score;
}

int main() {
  ios::sync_with_stdio(false);
  cin >> R >> C >> P;
  elevations.assign(R, vector<int>(C, 0));
  for (int i = 0; i < R; ++i) {
    for (int j = 0; j < C; ++j)
      cin >> elevations[i][j];
  }

  int best_path = ILLEGAL_VALUE;
  DP.assign(R, vector<vector<int>>(C, vector<int>(P+1, -2)));
  for (int start_row = 0; start_row < R; ++start_row) {
    if (elevations[start_row][0] != -1) {
      int cur = elevations[start_row][0]
                  + solve(start_row, 0, P);
      if (cur < best_path)
        best_path = cur;
    }
  }


  if (best_path < ILLEGAL_VALUE)
    cout << best_path << endl;
  else
```

```
    cout << "impossible" << endl;

  return 0;
}
```

Dynamic programming problem-solving strategies can be implemented using either memoization (top-down approach) or tabulation (bottom-up approach). In the case of the problem at hand, the author opted for the top-down approach by employing a recursive function. This approach can be viewed as a fusion of dynamic programming with a DFS-like backtracking. This amalgamation incorporates memoization, wherein solutions to subproblems are stored in a table to prevent redundant computations.

Conversely, the below code utilizes an iterative bottom-up approach, employing the tabulation techniques. This method entails solving subproblems in a predetermined order and gradually constructing the solution.

The algorithmic approach of the iterative implementation can be summarized as follows:

- Defining the *problem* and the *memoization table*:
  - Original problem: determine the lengths of the shortest valid paths to the last column of the given matrix that includes exactly P passes.
  - Generic subproblem: determine the length of the shortest valid path to each position r of column c that includes p passes.
  - Cell $dp[r][c][p]$ of array $dp[0 \dots R-1][0 \dots C-1][0 \dots P]$ stores the corresponding optimum value.
- Cells representing the *base cases* and the *original problem*:
  - Cells $dp[r][0][0]$, $r = 0, R-1$ correspond to trivial subproblems.
  - The required output is $min\{dp[r][C-1][P]$, where $r = 0, R-1\}$.
- Formulating the *recursive relation*:
  - If position [r][c] is not passable: $dp[r][c][p] = \infty$
  - If position [r][c] is not a pass: $dp[r][c][p] = e[r][c] + min\{dp[r-1][c-1][p], dp[r][c-1][p], dp[r+1][c-1][p]\}$
  - If position [r][c] is a pass: $dp[r][c][p] = e[r][c] + min\{dp[r-1][c-1][p-1], dp[r][c-1][p-1], dp[r+1][c-1][p-1]\}$
- *Base cases*:
  - If position $[r][0]$ is not passable: $dp[r][0][0] = \infty$
  - If position $[r][0]$ is passable: $dp[r][0][0] = e[r][0]$
  - For $p > 0$ also: $dp[r][0][p] = \infty$

The above approach is illustrated in *Figure 7.16*, which is based on the first sample from the problem statement. Black numbers represent the input matrix (stored in the two-dimensional array e), while red ones indicate elements of the three-dimensional array dp. We have highlighted the optimal decision sequence with arrows and bold cells.

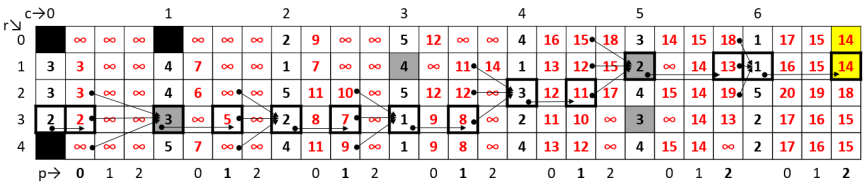


Figure 7.16. Illustrating the dynamic programming solution-building process

Iterative implementation

```
#include <iostream>
#include <fstream>
#define R 500
#define C 500
#define P 11
#define LL long long

using namespace std;

const int INF = 5e5 + 1;
int e[R][C], dp[R][C][P];
int r, c, p;

bool is_pass(int i, int j);

int main()
{
 freopen("inN.txt","r",stdin);
 cin >> r >> c >> p;
 for ( int i = 0 ; i < r ; ++i ){
  for ( int j = 0 ; j < c ; ++j ){
   cin >> e[i][j];
  }
 }
```

```
//Corresponding to the first column of the input matrix
for ( int i = 0 ; i < r ; ++i ){
 for ( int k = 0 ; k <= p ; ++k ){
  //Impassable location
  if( e[i][0] == -1 ){ dp[i][0][k] = INF; continue; }
  //In the first column there are not passes
  if( k > 0 ){ dp[i][0][k] = INF; continue; }
  dp[i][0][0] = e[i][0];
 }
}

//Corresponding to the next columns of the input matrix
int ans = INF;
for ( int j = 1 ; j < c ; ++j ){
 for ( int i = 0 ; i < r ; ++i ){
  for ( int k = 0 ; k <= p ; ++k ){
   dp[i][j][k] = INF;
   if( e[i][j] == -1 ){ continue; }
   if( is_pass(i,j) ){
    if( k == 0 ){ continue; }
    if( i > 0 ){
      dp[i][j][k] = min(dp[i][j][k], dp[i-1][j-1][k-1]+e[i][j]);
    }
    dp[i][j][k] = min(dp[i][j][k], dp[i][j-1][k-1]+e[i][j]);
    if( i < r-1 ){
     dp[i][j][k] = min(dp[i][j][k], dp[i+1][j-1][k-1]+e[i][j]);
    }
  }
  else{
   if( i > 0 ){
    dp[i][j][k] = min(dp[i][j][k], dp[i-1][j-1][k]+e[i][j]);
   }
   dp[i][j][k] = min(dp[i][j][k], dp[i][j-1][k]+e[i][j]);
   if( i < r-1 ){
    dp[i][j][k] = min(dp[i][j][k], dp[i+1][j-1][k]+e[i][j]);
   }
  }
 }
 if( j == c-1 ){ ans = min(ans,dp[i][c-1][p]); }
 }
}
```

```
 if( ans == INF ){ cout << "impossible"; }
 else { cout << ans; }

 return 0;
}

//Retrieved from author
bool is_pass(int i, int j) {
 if ( i == 0 || j == 0 || i == r-1 || j == c-1 ) { return false; }
 if ( e[i][j] == -1 ||
      e[i-1][j] == -1 ||
      e[i+1][j] == -1 ||
      e[i][j-1] == -1 ||
      e[i][j+1] == -1 ){
  return false;
 }
 if ( e[i-1][j] > e[i][j] &&
      e[i+1][j] > e[i][j] &&
      e[i][j-1] < e[i][j] &&
      e[i][j+1] < e[i][j] ){
  return true;
 }
 else { return false; }
}
```

BIBLIOGRAPHY

[1] *T-shirt Buying* [the original problem]: https://codeforces.com/problemset/problem/799/B.

[2] M. O. Rabin. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12, 1, 128–138. DOI: 10.1016/0022-314X(80)90084-0.

[3] Márton Gy. 2008. *Kriptogtáfiai alapismeretek*. Scientia Publishing House, Cluj-Napoca.

PART II
MATHEMATICS

CHAPTER 2

SELECTED PROBLEMS FROM "BEGE ANTAL" MEMORIAL MATHEMATICS CONTEST

Authors: Emil Horobeţ, Pál Kupán, Róbert Szász

2.1 Combinatorics

2019. Let $k, n \in \mathbb{N}^*$ be two natural numbers with $k \geq 2$. An object weighing n kilograms is placed in one pan of a two-armed scale. Our goal is to put weights in the pans of the scale in order to balance it. If we only have weights of $1, k, k^2,\ k^3,\ \ldots$ kilograms, then how many weights do we need at least to level the scale?

Solution
After we balance the scale, we eliminate the weights which appear in both pans of the scale. Because every natural number has a unique representation in the k-based number system, we get the following two possibilities.
Let $n = \overline{a_1 a_2 \ldots a_p}$ be the representation of n in the k-based number system, that is

$$n = \sum_{i=0}^{p-1} a_{p-i} k^i,\ 0 \leq a_{p-i} \leq k-1,\ i \in \{0, 1, , \ldots, p-1\}.$$

Taking this representation into account, it follows that $k^p - n > 0$.
Let

$$k^p - n = \overline{b_1 b_2 \ldots b_q}$$

be the representation of $p^k - n$ in the k-based number system.
Hence the desired minimal number of weights is

$$\min\{a_1 + a_2 + \ldots + a_p, 1 + b_1 + b_2 + \ldots + b_q\}.$$

2018.1. There are 432 points on a circle, dividing the circle into 432 congruent arcs. We color the points with four different colors such that

each color is used 108 times. Prove that for each color there is a triangle of these points, whose vertices are colored with that specific color, and these triangles are congruent.

Solution
Let the four colors be red, blue, green, and yellow. We rotate the 108 red points with an angle of measure $\frac{2\pi}{432}$. We repeat this rotation 431 times. Every red point will cover every blue point. This means that precisely 108×108 times does a red point cover a blue point during the 431 rotations. Since $\frac{108\times 108}{431} > 27$, it follows that there is a rotation such that after that rotation at least 28 of the red points will cover blue points.
Now we rotate these 28 red points again 431 times. Using the previous idea, the inequality $\frac{108\times 28}{431} > 7$ implies that there is a rotation such that after that specific rotation among the 28 red points there exist 8 of them which cover green points.
Now we rotate the 8 points 431 times.
The inequality $\frac{108\times 8}{431} > 2$ implies that among the 8 red points there exist 3 which cover yellow points after a suitable rotation. And we are done, because these 3 points forming a red triangle cover a blue, a yellow, and a green triangle of points as well.

2018.2.a. Can we draw 21 circles in the plane such that every circle is tangent to exactly three other circles?

Solution
The problem can be modeled with a graph. The circles are the vertices of the graph, and between two vertices, there is an edge if the circles are tangent. Our graph has 21 vertices, and the degree of every vertex is 3, but this is impossible because in a graph the sum of the degrees of all vertices is equal to twice the number of edges. Thus, the answer is negative.

2018.2.b. We have 2017 triangles and 2018 rectangles. Can we build a polyhedron with these triangles and rectangles as faces?

Solution
The answer is no, we cannot build a polyhedron with these polygons as faces.
The solution is analogous to the solution of the previous problem. If there exits a polyhedron having 2017 triangle faces and 2018 rectangle faces,

then we get a contradiction. The polyhedron can be considered again a graph. The faces are the vertices, and two vertices are connected with an edge if the faces have a common side.
The sum of the degrees of vertices is equal to $2017 \times 3 + 2018 \times 4$, which is an odd number and so cannot be equal to twice the number of edges.

2017.

1. In at most how many regions do n lines divide the plane?

2. In at most how many regions do n circles divide the plane?

Solution
1. We obtain the maximum number of regions determined by n lines if there are not three concurrent lines and no parallel lines among them.
Let a_n denote the number of regions determined by n lines.
We have $a_1 = 2$ and $a_{n+1} = a_n + n + 1$, and this leads us to

$$a_n = 1 + \frac{n(n+1)}{2}.$$

2. We have analogous conditions in the case of the circles in order to obtain the maximum number of regions.
We obtain the maximum number of regions determined by n circles if there are no three concurrent circles and every two circle has two intersection points.
Let us denote by b_n the number of regions determined by n circles.
We have $b_1 = 2$ and $b_{n+1} = b_n + 2n$, and it follows that

$$b_n = n^2 - n + 2.$$

2016. In a group of 2016 people, everyone has exactly four enemies. Prove that we can choose a subset of 403 people so that in this subset no one has enemies.

Solution
We choose a person from the 2016 people, after which we eliminate the enemies of that person from the group, and so in the group will remain 2011 people. We can repeat this operation $\left[\frac{2016}{5}\right] = 403$ times. The chosen people form a subset of 403 individuals, and no one has enemies among them.

2012.1. On a sphere of unit radius and center O, there are nine points. Prove that among these points we can find two such that the distance between them is smaller than $\sqrt{2}$.

Solution
Three concurrent planes perpendicular to each other, which intersect in O, divide the sphere into eight parts. The box principle implies that there is part of the sphere among these eight that contains two of the nine points, say points A and B. We have that the angle $\widehat{AOB}$ is less than or equal to $\frac{\pi}{2}$, and this implies $d(A,B) \leq \sqrt{2}$.

2012.2. On a sphere of unit radius and center O, there are five points. Prove that among those five points we can find two such that the distance between them is smaller than $\sqrt{2}$.

Solution
Let A, B, C, D, E be the five points on a sphere. A plane which is perpendicular to the line OA divides the sphere in two hemispheres.
The hemisphere containing A will be denoted by H_A and the other hemisphere by H_1. If $H_A \cap \{B,C,D,E\} \neq \emptyset$, and then the problem is solved. Consequently, we have to discuss the case when $\{B,C,D,E\} \subset H_1$. We determine analogously the hemispheres H_B and H_2.
If $H_B \cap \{C,D,E\} \neq \emptyset$, then the problem is solved. We have to study the case when $\{C,D,E\} \subset H_2$. Thus, we have $\{C,D,E\} \subset H_2 \cap H_1$.
The plane (A,B,O) divides $H_2 \cap H_1$ in two congruent parts. Let us denote these parts by P_1 and P_2. If $C, D \in P_1$, and then the angle $\widehat{COD}$ is less than or equal to $\frac{\pi}{2}$; hence, $d(C,D) \leq \sqrt{2}$.

2012.3. We colored the points of a plane with two colors. Does there exist a right-sided triangle whose vertices are of the same color?

Solution
We will show that the answer is yes. Suppose that the two colours are black and white. Let $A_1A_2A_3A_4A_5A_6$ be a regular hexagon with center O such that A_1 and O are white. Then A_2 and A_6 are black, otherwise the problem is solved. The triangle $A_2A_4A_6$ is equilateral. Then A_4 has to be white, otherwise the problem is solved. So, O and A_4 are white; consequently, A_5 has to be black, otherwise the problem is solved. Finally, if $P = A_1A_6 \cap A_4A_5$, then if P is white, then PA_1A_4 is an equilateral white triangle, if P is

black, then PA_6A_5 is a black equilateral triangle, and the problem is solved.

2010.1. Let n be a natural number with $n \geq 1$, and let H_n be the set of integers given by $H_n = \{-n, -n+1, -n+2, \ldots, n-1, n\}$. Determine the number of subsets $A \subset H_n$ having the property that there do not exist $a, b \in A$ such that $a + b = 0$ and the size of A is maximal.

Solution
Since $0 + 0 = 0$, it follows that $0 \notin A$, and we infer $A \subset H_n \setminus \{0\}$, for any A satisfying the said condition. We have

$$H_n \setminus \{0\} = \cup_{k=1}^{n} \{-k, k\}.$$

If $card(A) \geq n + 1$, then the box principle implies that there are $a = k \in A$ and $b = -k \in A$ such that $a + b = 0$. Thus, we have $card(A) \leq n$ for any A satisfying the said condition.
The set $\{1, 2, \ldots, n\}$ satisfies the said conditions; consequently, $card(A) = n$ for any A satisfying the said condition. There are 2^n such sets because every element of A can be chosen in two different ways, according to its sign.

2008.1.1. In how many ways can a $2 \times n$ rectangle be covered with 1×2 dominoes?

Solution
We will deduce a recurrence formula in order to determine the desired number.
Let a_n denote the number of such covers. In the first step, we can cover the first two squares by a vertical domino, and then it remains to cover a $2 \times (n-1)$ rectangle. We can do this in a_{n-1} different ways.
If we cover in the first step the first four squares, then we can do this with two horizontal dominoes, after which remains to cover a $2 \times (n-2)$ rectangle: we can do this in a_{n-2} ways.
Thus, we get the $a_n = a_{n-1} + a_{n-2}$ recurrence formula. Since $a_1 = 1$ and $a_2 = 2$, we get the formula for a_n using the characteristic equation.

$$a_n = \frac{1}{\sqrt{5}} \Big[\Big(\frac{1+\sqrt{5}}{2} \Big)^n - \Big(\frac{1-\sqrt{5}}{2} \Big)^n \Big].$$

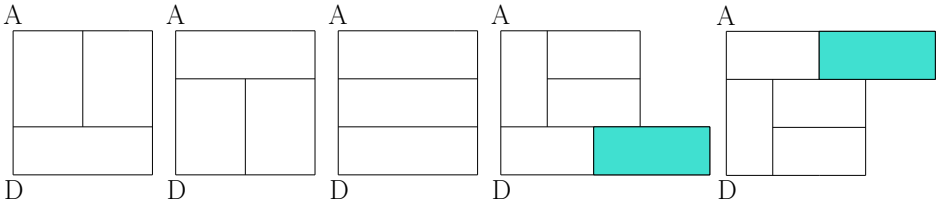
2008.1.2. Let m be an integer with $m > 1$. Prove that an $m \times 2n$ rectangle can be covered two times by 2×1 dominoes so that there is no domino in

the first layer which covers completely a domino from the second layer.

Solution
We have to discuss two cases. If m is even, then we cover the rectangle $m \times 2n$ using 2×2 squares, which are formed by two horizontal dominoes. In the second case, we cover the $m \times 2n$ rectangle by 2×2 squares, which are formed by two vertical dominoes. So, no domino from the first layer will cover any domino from the second layer.
If m is odd, then we cover the first row with n dominoes, and the rest of the rectangle will be covered by 2×2 squares formed by two vertical dominoes. In the second case, the last row is covered by n dominoes, and the rest of the rectangle will be covered by 2×2 squares formed by two vertical dominoes.
In this case then, no domino from the first layer will cover any domino from the second layer.

2008.1.3. How many ways can a $3 \times 2n$ rectangle be covered with 1×2 dominoes?

Solution
We will deduce again a recurrence formula in order to determine the desired number.
Let u_n denote the number of covers.
First, we can exactly cover the first six squares by dominoes in three different ways. Additionally, we can cover the first six squares in two more ways, by covering a total of eight squares (the first six and two more; see the picture below).

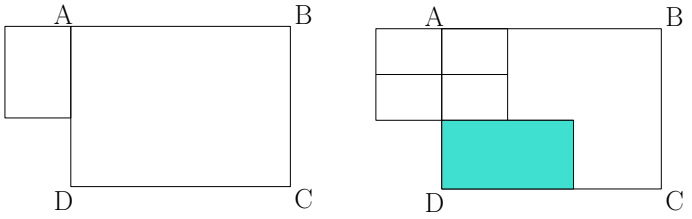


Thus, we get the equality

$$u_{n+1} = 3u_n + 2v_{n-1}, \tag{2.1}$$

where v_n denotes the number of coverings of the $3 \times 2n$ rectangle, which is supplemented by a 1×2 rectangle.

The following two diagrams suggest the following recurrence:



$$v_n = u_n + v_{n-1}.$$

These two equalities imply

$$u_{n+1} = 3u_n + 2v_{n-1} = 3u_n + 2u_{n-1} + 2v_{n-2}. \tag{2.2}$$

On the other hand, from (2.1), we infer

$$u_n = 3u_{n-1} + 2v_{n-2}. \tag{2.3}$$

Finally, from (2.2) and (2.3), we deduce

$$u_{n+1} = 3u_n + 2u_{n-1} + 2u_n - 3u_{n-1} = 4u_n - u_{n-1}.$$

Since $u_1 = 3$ and $u_2 = 11$, the method of characteristic equations implies

$$\begin{aligned} u_n &= \frac{\sqrt{3}+1}{2\sqrt{3}}\left(2+\sqrt{3}\right)^n + \frac{\sqrt{3}-1}{2\sqrt{3}}\left(2-\sqrt{3}\right)^n \\ &= \frac{2^n}{\sqrt{3}}\left[\left(\frac{1+\sqrt{3}}{2}\right)^{2n+1} - \left(\frac{1-\sqrt{3}}{2}\right)^{2n+1}\right]. \end{aligned}$$

2008.4.1. For fixed $x > 0$, $k \in \{2, 3\}$ and $n \in \mathbb{N}^*$, we denote by $A(x, k, n)$ the following assertion:
Coloring the points of a plane using n different colors, there are k points having the same color such that the distance between any two points of these k points is the same and is equal to x or to 1.
Prove the following:

(a) $A\left(\frac{\sqrt{2}}{2}, 2, 2\right)$,
(b) $A(1, 2, 3)$,
(c) $A\left(\frac{1+\sqrt{5}}{2}, 2, 4\right)$,
(d) $A(\sqrt{3}, 2, 4)$,

(e) $A\left(\frac{1+\sqrt{3}}{\sqrt{2}}, 2, 4\right)$.

Solution

(a) We have to prove that there exist two points of the same color, having distance 1 or $\frac{\sqrt{2}}{2}$. Let $ABCD$ be a square with unit sides. Let white and black be the two colors. We denote the intersection of the diagonals by O. Without losing the generality, we suppose that point A is white, and then B and D must be black. This implies that C must be white, as otherwise we would find two points of the same color and the problem is solved. Now, if O is white, then A and O have the same color and $AO = \frac{\sqrt{2}}{2}$; if O is black, then B and O have the same color and $BO = \frac{\sqrt{2}}{2}$.

(b) Now we colored the points of a plane using three colors: black, white, and yellow. Let ABC and BCD be two equilateral triangles in the colored plane. If A is yellow, then B and C must be black and white respectively and D is again yellow, otherwise the problem is solved. Let $AB_1C_1D_1$ be the image of the lozenge $ABCD$ after a rotation with center A and an angle of measure $\arccos\left(\frac{5}{6}\right)$. Then B_1 and C_1 must be black and white respectively and D_1 is yellow, otherwise the problem is solved.

Now D and D_1 are both yellow and

$$DD_1 = \sqrt{AD^2 + AD_1^2 - 2AD\dot{A}D_1 \cos\left(\arccos\left(\frac{5}{6}\right)\right)} = 1;$$

consequently, the problem is solved.

(c) We have to show that when coloring the plane using four colors, say black, white, yellow, and red, there are two points colored the same, having distance 1 or $\frac{1+\sqrt{5}}{2}$.

In order to solve the problem, we use a regular pentagon having unit sides. Every diagonal of this regular pentagon is of length $\frac{1+\sqrt{5}}{2}$. Let $A_1A_2A_3A_4A_5$ be a regular pentagon with unit sides. If A_1 is yellow, then $A_1A_2 = 1$, and, consequently, A_2 must be of other color, for example, red. Now $A_2A_3 = 1$ and $A_1A_3 = \frac{1+\sqrt{5}}{2}$, so A_3 has a different color; we suppose without losing generality that A_3 is blue. Since $A_1A_5 = 1$, $A_2A_5 = \frac{1+\sqrt{5}}{2}$ and $A_3A_5 = \frac{1+\sqrt{5}}{2}$, it follows that A_5 must be black. Now A_4 can be of any color, so it follows that the assertion $A\left(\frac{1+\sqrt{5}}{2}, 2, 4\right)$ holds.

(d) We have to show that when coloring the plane using four colors, say black, white, yellow, and red, there are two points colored the same, having distance 1 or $\sqrt{3}$.

Let $ABCDEF$ be a regular hexagon having unit sides.
Let the center O of the hexagon be red, point A yellow and B black, and then point C is necessarily white, because $CO = 1,\ CB = 1,\ CA = \sqrt{3}$.
Using an analogous argument, it follows that D is yellow, E is black, and F is white.
Thus, we obtain that if we want to color the vertices of a regular hexagon without obtaining a segment of length 1 or $\sqrt{3}$ with endpoints having the same color, then we must color the opposing vertices of the hexagon with the same color.
Let $AB_1C_1D_1E_1F_1$ be the hexagon which we get after a rotation with center A and an angle of measure $\arccos\left(\frac{7}{8}\right)$. Then $DD_1 = 1$, and the points D and D_1 have the same color.

(e) We have to show that when coloring the plane using four colors, say black, white, yellow, and red, there are two points colored the same, having distance 1 or $\frac{1+\sqrt{3}}{\sqrt{2}}$.

Let $A_1A_2\ldots A_{11}A_{12}$ be a 12-sided regular polygon with unit sides. Let r be the radius of the circumscribed circle. We have

$$1 = \sqrt{2r^2 - 2r^2\cos\frac{2\pi}{12}} \Rightarrow r = \frac{1}{2-\sqrt{3}} = \frac{1+\sqrt{3}}{\sqrt{2}}.$$

Let O be the center point of the polygon.

Let the center point be yellow, A_1 black, and A_2 white. Since $A_3A_2 = 1$ and $A_3A_1 = A_3O = \frac{1+\sqrt{3}}{\sqrt{2}}$, it follows that A_3 must be red. We color the points of the polygon step by step continuing this method, and we get that two vertices which are symmetric with respect to the middle point O have the same color (otherwise we will find two points A_i and A_j such that $A_iA_j \in \{1, \frac{1+\sqrt{3}}{\sqrt{2}}\}$). Now we apply a rotation with center A_1 and argument $\arccos\frac{7}{8}$. The image of the polygon is $A_1A_2\ldots A_{11}A_{12}$ is $A_1B_2B_3\ldots B_{11}B_{12}$. Also, in $A_1B_2B_3\ldots B_{11}B_{12}$, the vertices which are symmetric with respect to the center must be colored the same; consequently, B_7 must be black. Now $A_7B_7 = \sqrt{2^2+2^2-8\cos\left(\arccos\left(\frac{7}{8}\right)\right)} = 1$, and the proof is done.

2.2 Number Theory Problems

2010.1.

a. Prove that $2^{2^{2010}} - 1$ has 2010 different prime divisors.

Solution: First we decompose the above expression as a product by

$$2^{2^{2010}} - 1 = \left(2^{2^{2009}} + 1\right) \cdot \left(2^{2^{2009}} - 1\right),$$

then we expand the second factor in a similar way, and so on, until we get

$$2^{2^{2010}} - 1 = \left(2^{2^{2009}} + 1\right) \cdot \left(2^{2^{2008}} + 1\right) \cdot \ldots \cdot \left(2^{2^0} - 1\right).$$

Then, we observe that the above factorization has exactly 2010 different terms.
It only remains to prove that any pair of these terms is a relative prime, so each of them is divisible by different prime factors; hence, $2^{2^{2010}} - 1$ has 2010 different prime divisors.
So, for any pair $2^{2^i} + 1$, $2^{2^j} + 1$, with $0 \neq i < j$, we compute their g.c.d. Now we will apply multiple times the properties of g.c.d. that for any $a < b$ natural numbers

$$(a, b) = (a, a + b) = (a + b, b) = (a, b - a),$$

where we denote the g.c.d of a and b by (a, b).
So, we get that

$$\left(2^{2^i} + 1, 2^{2^j} + 1\right) = \left(2^{2^i} + 1, 2^{2^j} - 2^{2^i}\right) =$$

$$= \left(2^{2^i} + 1, 2^{2^i}\left(2^{2^j - 2^i} - 1\right)\right) = \left(2^{2^i} + 1, 2^{2^j - 2^i} - 1\right).$$

The latter equality holds since if a and b are odd numbers and k is any natural number; then

$$(a, 2^k b) = (a, b).$$

Now, to continue, we get

$$\left(2^{2^i} + 1, 2^{2^j - 2^i} - 1\right) = \left(2^{2^i} + 1, 2^{2^j - 2^i} + 2^{2^i}\right) =$$

$$= \left(2^{2^i} + 1, 2^{2^i}\left(2^{2^j - 2^{i+1}} + 1\right)\right) = \left(2^{2^i} + 1, 2^{2^j - 2^{i+1}} + 1\right).$$

Now if $j \neq i+1$, then we repeat this process $j-(i+1)$ times until we get that

$$\left(2^{2^i}+1, 2^{2^j-2^i}+1\right) = \left(2^{2^i}+1, 2^{2^j-2^{i+1}}+1\right) = \dots$$

$$= \left(2^{2^i}+1, 2^{2^j-2^{i+(j-i)}}+1\right) = \left(2^{2^i}+1, 2^0+1\right)$$

$$= \left(2^{2^i}+1, 2\right) = 1,$$

since $2^{2^i}+1$ is an odd number.
So, to summarize, we get that $\left(2^{2^i}+1, 2^{2^j}+1\right) = 1$ for any $0 \neq i < j$, so each factor has different prime divisors; hence, $2^{2^{2010}}-1$ has 2010 different prime divisors.

b. Prove that $6^{6^n}-1$ has at least $3n+1$ prime divisors.

Solution: We will use the following formula

$$x^6-1 = (x^3-1)(x^3+1)$$

$$= (x-1)(x^2+x+1)(x+1)(x^2-x+1).$$

So, we get that

$$6^{6^n}-1 = \left(6^{3^1\cdot 6^{n-1}}\right)^2 - 1^2 = \left(6^{3^1\cdot 6^{n-1}}+1\right)\left(6^{3^1\cdot 6^{n-1}}-1\right).$$

We continue the same expansion, and we get

$$6^{6^n}-1 = \left(6^{3^1\cdot 6^{n-1}}+1\right)\dots\left(6^{3^n\cdot 6^{n-n}}+1\right)\left(6^{3^n\cdot 6^{n-n}}-1\right).$$

Let us denote the first n factors of the product above by A, so we have that

$$6^{6^n}-1 = A\cdot\left(6^{3^n}-1\right) = A\cdot\left(\left(6^{3^{n-1}}\right)^3 - 1^3\right)$$

$$= A\cdot\left(6^{2\cdot 3^{n-1}}+6^{3^{n-1}}+1\right)\left(6^{3^{n-1}}-1\right).$$

Now we continue the same expansion to the right, and we get

$$6^{6^n}-1 = A\cdot$$

$$\left(6^{2\cdot 3^{n-1}}+6^{3^{n-1}}+1\right)\dots\left(6^{2\cdot 3^{n-n}}+6^{3^{n-n}}+1\right)\left(6^{3^{n-n}}-1\right).$$

If we denote the middle n term by B, then we get that

$$6^{6^n} - 1 = A \cdot B \cdot \left(6^{3^{n-n}} - 1\right),$$

a product of $2n + 1$ terms.
Now, finally, we observe that each of the n terms of A factors into two terms in the following way:

$$6^{3^i \cdot 6^{n-i}} + 1 = \left(6^{3^{i-1} \cdot 6^{n-i}}\right)^3 + 1^3 =$$

$$= \left(6^{3^{i-1} \cdot 6^{n-i}} + 1\right)\left(6^{2 \cdot 3^{i-1} \cdot 6^{n-i}} - 6^{3^{i-1} \cdot 6^{n-i}} + 1\right).$$

Hence, expression A does factor into $2n$ terms $A_1 \cdot A_2$.
So, finally, we get that

$$6^{6^n} - 1 = A_1 \cdot A_2 \cdot B \cdot \left(6^{3^{n-n}} - 1\right),$$

a product of $3n + 1$ terms; so, it follows that $6^{6^n} - 1$ has at least $3n + 1$ prime divisors.

Prove that if a is a natural number, such that it is relatively prime to 10, then a has a multiple, which, presented in the decimal number system, has only 9 as its digits.

Solution: We will apply Euler's theorem.

Euler's theorem
If x and n are relative prime to each other, then

$$x^{\varphi(n)} \equiv 1 (\text{mod } n).$$

Here $\varphi(n)$ is Euler's totient function, that is, the number of integers in $\{1, 2, ..., n - 1\}$ which are relatively prime to n.

Now we choose $x = 10$ and $n = a$ and apply the above theorem. Since $(a, 10) = 1$, it follows that

$$10^{\varphi(a)} \equiv 1 (\text{mod } a).$$

This means that there exists a k natural number such that

$$10^{\varphi(a)} = 1 + k \cdot a.$$

Which means that there exists a multiple k of a such that

$$k \cdot a = 10^{\varphi(a)} - 1 = 9 \cdot \frac{10^{\varphi(a)} - 1}{10 - 1} =$$

$$= 9 \cdot (10^{\varphi(a)-1} + \ldots + 10 + 1) = 9 \cdot \underbrace{\overline{11\ldots1}}_{\varphi(a)} = \underbrace{\overline{99\ldots9}}_{\varphi(a)}.$$

2015.1.

a. Let a, b be integers. Prove that if $2015|a^3+b^3$ and $2015|a^4+b^4$, then it follows that $2015|a^5+b^5$.

Solution: First observe that

$$a^5 + b^5 = (a+b)(a^4 - a^3b + a^2b^2 - ab^3 + b^4) =$$

$$= (a+b)((a^4+b^4) - ab(a^2 - ab + b^2)) =$$

$$= (a+b)(a^4+b^4) - ab[(a+b)(a^2 - ab + b^2)] =$$

$$= (a+b)(a^4+b^4) - ab(a^3+b^3).$$

So, if $2015|a^3+b^3$, then $2015|ab(a^3+b^3)$, and, also, if $2015|a^4+b^4$, then $2015|(a+b)(a^4+b^4)$; hence

$$2015 \,|(a+b)(a^4+b^4) - ab(a^3+b^3) = a^5+b^5.$$

b. Is it true that $(\underbrace{66\ldots6}_{2015})^2 + \underbrace{88\ldots8}_{2015} = \underbrace{44\ldots4}_{2\cdot2015}$?

Solution: First we have that

$$\underbrace{66\ldots6}_{2015} = 6 \cdot \underbrace{11\ldots1}_{2015} = 6 \cdot \frac{10^{2015}-1}{9},$$

and similarly

$$\underbrace{88\ldots8}_{2015} = 8 \cdot \underbrace{11\ldots1}_{2015} = 8 \cdot \frac{10^{2015}-1}{9},$$

$$\underbrace{44\ldots4}_{2\cdot2015} = 4 \cdot \underbrace{11\ldots1}_{2\cdot2015} = 4 \cdot \frac{\left(10^{2015}\right)^2-1}{9}.$$

Let us denote 10^{2015} by x, and then we have to verify that

$$\left(6 \cdot \frac{x-1}{9}\right)^2 + 8 \cdot \frac{x-1}{9} = 4 \cdot \frac{x^2-1}{9}.$$

Clearing denominators and expanding the brackets we get that the statement is true.

2015.3.

a. Prove that if a is a natural number, such that it is relative prime to 30, then a has a multiple, which, presented in the decimal number system, has only 1 as its digits.

Solution: In a similar fashion to problem **2012.4**, we will apply Euler's theorem. First observe that if $(a, 30) = 1$, then $(a, 10) = 1$, and, as a consequence, $(9a, 10) = 1$. So, we will apply Euler's theorem to $x = 10$ and $n = 9a$ and get that

$$10^{\varphi(9a)} \equiv 1 (\text{mod } 9a).$$

This means that there exists k natural number, such that

$$10^{\varphi(9a)} = 1 + k \cdot (9a).$$

So, we get that there exists a multiple k of a such that

$$k \cdot a = \frac{10^{\varphi(9a)} - 1}{9} = \frac{10^{\varphi(9a)} - 1}{10 - 1} =$$

$$= 10^{\varphi(9a)-1} + \ldots + 10 + 1 = \underbrace{11 \ldots 1}_{\varphi(9a)}$$

Observation: Since $(a, 3) = 1$, it follows that $(a, 9) = 1$, and hence, due to the property that the Euler's totient function is multiplicative on relative prime factors, we get that

$$\varphi(9a) = \varphi(9) \cdot \varphi(a) = 6 \cdot \varphi(a).$$

2016.A.1. The product of all divisors of a natural number n equals to 7^{595}. Determine the last digit of n.

Solution: Since the product of all divisors has only 7 as a prime factor, it means that $n = 7^k$, for some k natural. Then, the divisors of n are $7^0, 7^1, \ldots, 7^k$. Their product hence is

$$\prod_{i=0}^{k} 7^i = 7^{\sum_{i=0}^{k} i} = 7^{\frac{k \cdot (k+1)}{2}} = 7^{595}.$$

So, we get that $\frac{k(k+1)}{2} = 595$, and from here it follows that $k = 34$, so $n = 7^{34}$.

Now the last digit of the powers of 7, starting form the second power, have the following period: $4, 8, 6, 2,\ 4, 8, 6, 2, \ldots$ So, if the remainder of the power modulo 4 is 2, then the last digit is 4 : since 34 is 2 modulo 4, then the last digit of 7^{34} is 4.

2016.B.1. For which $p > 3$ prime and $n > 0$ integer is it true that $p^n - 4$ is a fourth power?

Solution: Suppose that $p^n - 4 = k^4$, for some k natural number. Then we get that

$$p^n = k^4 + 4 = (k^2 + 2k + 2)(k^2 - 2k + 2).$$

This equality can only hold if $k^2 + 2k + 2 = k^2 - 2k + 2$, or $k^2 + 2k + 2 = 1$, or $k^2 - 2k + 2 = 1$.

In the first case, $k = 0$, and it follows that $p^n = 4$, with p prime, so $p = 2$ and $n = 2$. But this is not the solution we are looking for because p has to be greater than 3.

In the second and third cases, we get that $k = 1$ or $k = -1$. In both cases, it follows that $p^n = 5$, with p prime. Hence, $p = 5$ and $n = 1$ is the only solution to this problem.

2019.4. Prove that for any n natural number there are always n consequent natural numbers such that none of them is a prime or a prime power.

Solution: Consider the sequence

$$(n+2)! + 2 \cdot 3, (n+2)! + 3 \cdot 4, \ldots, (n+2)! + (n+1) \cdot (n+2).$$

The i-th term is divisible by both $i + 1$ and $i + 2$, so it cannot be a prime nor a prime power.

2.3 Geometry Problems

Problem 1. (ECN-2019).

Let ABC be a triangle. Let M be the point on segment $[AB]$ such that $\frac{AM}{MB} = \frac{1}{2017}$, N be the point on segment $[BC]$ such that $\frac{BN}{NC} = \frac{1}{2018}$, and P be the point on segment $[CA]$ such that $\frac{CP}{PA} = \frac{1}{2019}$. Let Q be the intersection of AN and CM, R be the intersection of AN and BP, and S be the intersection of BP and CM. Find the ratio of the area of triangles QRS to the area of triangle ABC.

Proof. The problem is a specific application of Routh's theorem, in which the ratios are: $x = 2017, y = 2018, z = 2019$. The area of the inner triangle is

$$\mathcal{A}_{QRS} = \mathcal{A}_{ABC} - \mathcal{A}_{ABR} - \mathcal{A}_{BCS} - \mathcal{A}_{CAQ}.$$

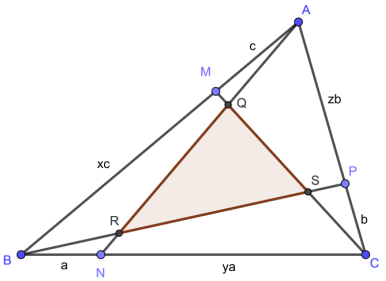


Figure 2.1. The Routh problem

From Menelaus's theorem on $\triangle ABN$ with the traversal MQC

$$\frac{MA}{MB}\cdot\frac{CB}{CN}\cdot\frac{QN}{QA}=1$$

Then $\frac{QN}{QA}=\frac{MB}{MA}\cdot\frac{CN}{CB}=\frac{x\cdot y}{y+1}$, so

$$\mathcal{A}_{AQC}=\frac{QA}{NA}\cdot\mathcal{A}_{ANC}=\frac{QA}{NA}\cdot\frac{CN}{CB}\cdot\mathcal{A}_{ABC}=\frac{y}{y+1+x\cdot y}\cdot\mathcal{A}_{ABC}$$

and

$$\frac{\mathcal{A}_{AQC}}{\mathcal{A}_{ABC}}=\frac{y}{y+1+x\cdot y}$$

Similarly,

$$\frac{\mathcal{A}_{BCS}}{\mathcal{A}_{ABC}}=\frac{x}{x+1+x\cdot z},\ \frac{\mathcal{A}_{ABR}}{\mathcal{A}_{ABC}}=\frac{z}{z+1+y\cdot z},$$

so,

$$\frac{\mathcal{A}_{QRS}}{\mathcal{A}_{ABC}}=1-\frac{x}{x+1+x\cdot z}-\frac{y}{y+1+x\cdot y}-\frac{z}{z+1+y\cdot z}=$$
$$=\frac{(xyz-1)^2}{(x+1+xz)(y+1+xy)(z+1+yz)}.$$

Note: For $x=y=z=1$, we obtain the case of the three concurrent medians, so the area of the inner triangle is zero. For $x=y=z=2$, we obtain the one-seventh-area triangle. □

Problem 2. (ECN-2018)

1. Let $[ABCD]$ be a tetrahedron such that the angles $\widehat{CAD}$ and $\widehat{CBD}$ are both obtuse. Prove that $|CD|>|AB|$.

2. Let $[ABCD]$ be a tetrahedron in which all four faces have equal area. Suppose P is an interior point of $[ABCD]$. Let d_A denote the distance from point P to the face opposite to vertex A (ie., face BCD). Define d_B, d_C, d_D analogously as the distances from P to faces ACD, ABD, and ABC respectively. Prove that

$$|PA| + |PB| + |PC| + |PD| \geq 3(d_A + d_B + d_C + d_D).$$

Proof. (1) We will keep the CAD and CBD triangle unchanged and unfold the tetrahedron along the CD edge. The maximum length of $|AB|$ is attained when the two triangles are coplanar. Even in this case, $|AB| < |CD|$ because the points A, B lie inside the circle with diameter CD (see the figure below) due to the obtuse angles $\widehat{CAD}$ and $\widehat{CBD}$.

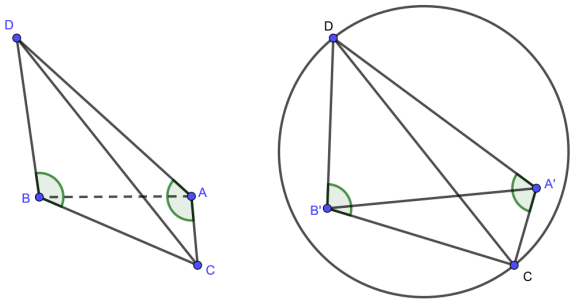


Figure 2.2. Unfolding the tetrahedron

(2) The tetrahedron is "equifacial" (all faces have equal area, $\mathcal{A}$), so the heights from each vertex have equal length: h. The volume can be expressed as

$$V = \frac{1}{3} h \cdot \mathcal{A}, \tag{2.4}$$

and also as the volume sum of the $PABC$, $PACD$, $PABD$, $PBCD$ tetrahedrons

$$V = \frac{1}{3} (d_A + d_B + d_C + d_D) \mathcal{A}.$$

From (2.4), $h \cdot \mathcal{A} = 3V$, and since $|PA| + d_A \geq h \Rightarrow (|PA| + d_A) \mathcal{A} \geq 3V = (d_A + d_B + d_C + d_D) \mathcal{A}$, so

$$|PA| \geq d_B + d_C + d_D.$$

Similarly,

$$|PB| \geq d_A + d_C + d_D, |PC| \geq d_A + d_B + d_D, |PD| \geq d_A + d_B + d_C,$$

and adding them we get the requested result. □

Problem 3. (ECN-2017)

What is the maximum number of parts into which a plane can be divided:

1. n straight lines?
2. n circles?

Proof. (1) To reach the maximum, it is necessary that the lines are not parallel and that no three lines are concurrent in one point. It is obvious that for one line the plane is divided in two parts. Suppose that k lines have already been constructed. The next ($k+1$-st) line intersects the previous lines at k points and increases the previous parts by $k+1$. Denoting by L_k the (maximum) parts obtained by k line intersection, we get:

$$L_{k+1} = L_k + (k+1)\,.$$

So, the maximal parts obtained with n lines are:

$$\begin{aligned} L_n &= L_{n-1} + ((n-1)+1) = L_{n-2} + (n-1) + n = \dots \\ &= L_1 + 2 + 3 + \dots + n = 2 + 2 + 3 + \dots + n = 1 + \frac{n(n+1)}{2}. \end{aligned}$$

(2) To reach the maximum, it is necessary that the circles intersect each other at two points and that no three circles intersect at one point. Such a situation is represented in the figure below, where the three circles with diameter lengths equal to $|AB|$ have the center on the segment AB.

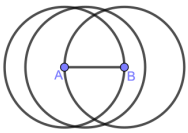


Figure 2.3. Three circles dividing the plane into a maximal number of parts

The idea is similar to the first sub-point. One circle divides the plane in two parts. The $(k+1)$st circle intersects each of the previous circles at 2 points; these $2k$ points divide the $(k+1)$st circle into $2k$ arcs. Each of these arcs divides one of the regions – formed by the first k circles – into two parts.

So, the $(k+1)$st circle increases the previous parts by $2k$: $C_{k+1} = C_k + 2k$. So, the maximal parts obtained with n circles are:

$$\begin{aligned} C_n &= C_{n-1} + 2(n-1) = C_{n-2} + 2(n-2) + 2(n-1) = \dots \\ &= C_1 + 2 + 4 + \dots + 2(n-1) = \\ &= 2 + 2 + 4 + \dots + 2(n-1) = \\ &= 2 + n^2 - n. \end{aligned}$$

□

Problem 4. (ECN-2016)

Consider a circle with diameter AB, and let C be a point on the arc $\overset{\frown}{AB}$. In triangle $\triangle ABC$, let points D and E lie on side AB such that: $AD = AC$ and $BE = BC$. Let F be the point where the circle centered at D, which is tangent to side BC, touches BC. Let G be the point where the circle centered at E, which is tangent to side AC, touches AC. Determine the bounds (minimum and maximum values) of the ratio $\frac{t}{T}$, where t is the area of $\triangle CGF$, and T is the area of $\triangle ABC$.

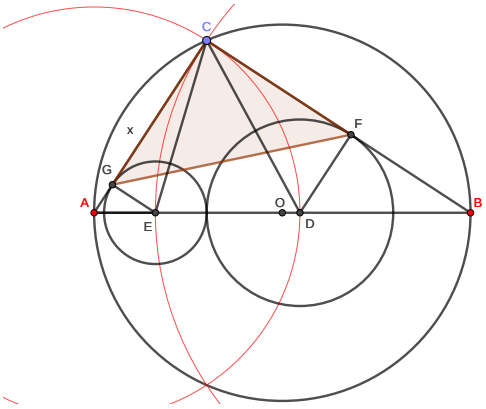


Figure 2.4. Ratio of $\triangle CGF$ and $\triangle ABC$

Proof. Suppose the diameter $AB = 2$, and let $x \in [0, 2]$ denote the length of AC. Then

$$AC = AD = x \Rightarrow BD = 2 - x.$$

From Pythagoras's theorem

$$BC = BE = \sqrt{4 - x^2} \Rightarrow AE = 2 - \sqrt{4 - x^2}.$$

The $\triangle AGE$ and $\triangle ACB$ are similar, so

$$\frac{GE}{CB} = \frac{AE}{AB} \Rightarrow GE = \frac{\sqrt{4-x^2}}{2}\left(2 - \sqrt{4-x^2}\right),$$

and from here we have

$$AG^2 = AE^2 - GE^2 = \frac{x^2}{4}\left(2 - \sqrt{4-x^2}\right),$$

and

$$CG = AC - AG = \frac{1}{2}x\sqrt{4-x^2}.$$

In the same manner,

$$CF = \frac{1}{2}x\sqrt{4-x^2},$$

so, $\triangle CGF$ is a right angle isosceles triangle, whose area is

$$\mathcal{A}_{CGF} = \frac{1}{2}CF^2 = \frac{1}{8}x^2\left(4-x^2\right),$$

and the ratio of the areas

$$\frac{t}{T} = \frac{\mathcal{A}_{CGF}}{\mathcal{A}_{ABC}} = \frac{\frac{1}{8}x^2\left(4-x^2\right)}{\frac{1}{2}x\sqrt{4-x^2}} = \frac{1}{4}x\sqrt{4-x^2}.$$

The function $x\sqrt{4-x^2} = \sqrt{4x^2-x^4}$ attains its extreme values where $4x^2 - x^4$ function does, or with $p = x^2 \in [0,4]$ we obtain the $4p - p^2$ parabola, which has minimums at $p = 0$ and $p = 4$, ($x = 0$ and $x = 2$), and maximum at $p = 2$, ($x = \sqrt{2}$). So, the ratio $\frac{t}{T}$ is between zero ($C = A$ or $C = B$) and half ($C =$ middle of $\widehat{AB}$)

$$\frac{t}{T} \in \left[0, \frac{1}{2}\right].$$

□

Problem 5. (ECN-2015)

1. Let $[ABCDA'B'C'D']$ be a unit cube, where $[ABCD]$ is the base. Point F lies on the edge $[AA']$, dividing it in a ratio of $1:2$, with F being closer to $A\prime$. Let G be the center of the top face $[A'B'C'D']$.

   a) Find the area of the polygon formed by the intersection of the plane (BFG) with the cube.

   b) Calculate the distance from point B' to the plane (BFG).

2. Let A and B be two fixed points in the plane, and let $\alpha \geq 1$ be a real number. Determine the set of points P in the plane for which the sum
$$PA^\alpha + PB^\alpha$$
is minimized.

Proof. (1)(a) We will enlarge BF until it intersects the $(A'B'C'D')$ plane in K. Connect K with G and enlarge until they intersect edge $B'C'$ in J. The requested polygon is the trapezoid $BFEJ$ $(BJ \| FE)$.

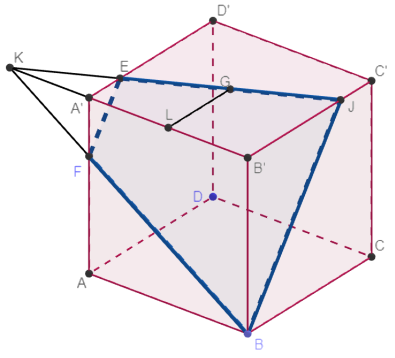


Figure 2.5. Cuts in the cube

From Pythagoras's theorem, in $\triangle BAF$, we get $BF = \frac{\sqrt{13}}{3}$. For all similar triangles, $\triangle KA'F \sim \triangle KB'B$, $\triangle KA'E \sim \triangle KB'J$, $\triangle KFE \sim \triangle KBJ$, the ratio of sides is $\frac{A'F}{B'B} = \frac{1}{3}$. It follows that

$$KA' = \frac{1}{2},\ KF = \frac{\sqrt{13}}{6}.$$

Denote L the middle of $A'B'$, then $\triangle KA'E \sim \triangle KLG$, so

$$\frac{KA'}{KL} = \frac{A'E}{LG} \Rightarrow A'E = \frac{1}{4},$$

and using $\frac{A'E}{B'J} = \frac{1}{3}$ we obtain

$$B'J = \frac{3}{4}.$$

Again, using Pythagoras's theorem, we can determine the sides

$$FE = \frac{5}{12},\ EJ = \frac{\sqrt{5}}{2},\ JB = \frac{5}{4},$$

and from $\frac{KE}{KJ} = \frac{FE}{BJ} = \frac{1}{3}$ we get

$$KE = \frac{\sqrt{5}}{4},\ FE = \frac{5}{12}.$$

The area of $BFEJ$ is obtained as the difference between the area $\triangle KBJ$ and $\triangle KFE$, whose sides are in $\frac{1}{3}$ ratio, so the area's ratio is $\frac{1}{9}$: $\mathcal{A}_{BFEJ} = \mathcal{A}_{KBJ} - \mathcal{A}_{KFE} = 8\mathcal{A}_{KFE}$. From Heron's formula, we get $\mathcal{A}_{KFE} = \frac{\sqrt{29}}{48}$, so

$$\mathcal{A}_{BFEJ} = \frac{\sqrt{29}}{6}.$$

(b) The volume of the $B'BKJ$ tetrahedron can be expressed as

$$V_{B'BKJ} = \frac{1}{6} \cdot d\left(B', (KBJ)\right) \cdot \mathcal{A}_{KBJ},\ \text{or}\ V_{B'BKJ} = \frac{1}{6} \cdot KB' \cdot \mathcal{A}_{B'BJ}.$$

From here,

$$d\left(B', (KBJ)\right) = \frac{KB' \cdot \mathcal{A}_{B'BJ}}{\mathcal{A}_{KBJ}} = \frac{\frac{3}{2} \cdot \frac{3}{8}}{\frac{3\sqrt{29}}{16}} = \frac{3}{\sqrt{29}}.$$

(2) For the particular value $\alpha = 1$, from the triangle inequality, it can be deduced that the minimal value of $PA^\alpha + PB^\alpha$ is attained for (any) P between A and B. Let us denote the affixes of A, B, P by z_A, z_B, z_P; so, we can write

$$PA = |z_A - z_P|, PB = |z_B - z_P|.$$

For $\alpha > 1$, the function $f(x) = x^\alpha,\ x > 0$ is convex, so

$$\left(\frac{x+y}{2}\right)^\alpha \leq \frac{1}{2}\left(x^\alpha + y^\alpha\right).$$

Using PA and PB for x, y, we obtain:

$$\begin{aligned}\frac{1}{2}\left(PA^{\alpha}+PB^{\alpha}\right)&=\frac{1}{2}\left(|z_A-z_P|^{\alpha}+|z_B-z_P|^{\alpha}\right)\geq\\&\geq\left(\frac{|z_A-z_P|+|z_B-z_P|}{2}\right)^{\alpha}\geq\\&\geq\left(\frac{|z_A-z_P+z_B-z_P|}{2}\right)^{\alpha}=\left(\left|\frac{z_A+z_B}{2}-z_P\right|\right)^{\alpha},\end{aligned}$$

so the minimum is attained for $z_P=\frac{z_A+z_B}{2}$, i.e., P is the middle of AB. □

Problem 6. (ECN-2012)

1. Let $ABCD$ be a rectangle. Find the position of a point P in the plane of the rectangle such that the sum $PA+PB+PC+PD$ is minimized.
2. Let $ABCD$ be a convex quadrilateral such that $\widehat{A}\geq\frac{\pi}{2}$ and $\widehat{C}\geq\frac{\pi}{2}$. Prove that $AC\leq BD$.
3. Prove that in every tetrahedron there exists at least one vertex such that the three edges adjacent to it can form a triangle.

Proof. (1) The minimum of $AP+PC$ is AC and is attained on the segment $|AC|\ni P$. The same is true for BD. So, P must be at the intersection of the two diagonals and $\min\left(PA+PB+PC+PD\right)=AC+BD$.

(2) Consider the circle with diameter BD.

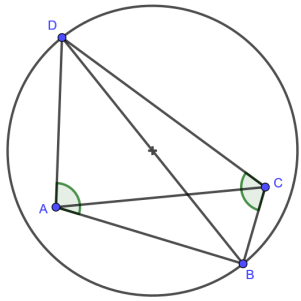


Figure 2.6. The points A, C are inside the circle

Because the angles $\widehat{A}$ and $\widehat{C}$ are obtuse, the points A and C lie in the interior of the circle, so the distance between A and C is less than the diameter BD.

(3) The conditions that $x, y, z > 0$ form the sides of a triangle are:

$$x + y > z,\ y + z > x,\ z + x > y,$$

equivalent with

$$x + y - z > 0,\ y + z - x > 0,\ z + x - y > 0,$$

or

$$(x + y - z) \cdot (y + z - x) \cdot (z + x - y) > 0.$$

For reasons of symmetry, the above product is multiplied by $(x + y + z) > 0$, so the condition is:

$$(x + y + z)\,(x + y - z)\,(y + z - x)\,(z + x - y) > 0.$$

The last product is denoted by

$$\begin{aligned}\text{heron}\,(x, y, z) &:= (x + y + z)\,(x + y - z)\,(y + z - x)\,(z + x - y) = \\ &= -x^4 - y^4 - z^4 + 2x^2y^2 + 2y^2z^2 + 2z^2x^2,\end{aligned}$$

because the Heron area formula can be expressed with this term:

$$\mathcal{A} = \frac{1}{4}\sqrt{\text{heron}(x, y, z)}.$$

Let a, b, c, d, e, f be the edges of the tetrahedron.

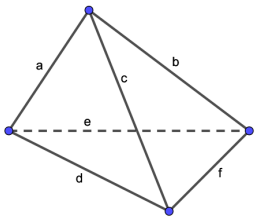


Figure 2.7. The edges of the tetrahedron

For the four faces of the tetrahedron, we can write

$$\text{heron}\,(a, b, e) > 0, \text{heron}\,(a, c, d) > 0, \text{heron}\,(b, c, f) > 0, \text{heron}\,(d, e, f) > 0,$$

and summing up and dividing by 2 gives

$$a^2b^2+b^2e^2+e^2a^2+a^2c^2+c^2d^2+d^2a^2+b^2c^2+c^2f^2+f^2b^2+ \\ +d^2e^2+e^2f^2+f^2d^2>a^4+b^4+c^4+d^4+e^4+f^4 \tag{2.5}$$

Assuming the opposite of the required statement

$$\text{heron}\,(a,b,c)\le 0, \text{heron}\,(a,e,d)\le 0, \text{heron}\,(b,e,f)\le 0, \text{heron}\,(d,c,f)\le 0,$$

and summing up and dividing by 2 gives

$$a^2b^2+b^2c^2+c^2a^2+a^2e^2+e^2d^2+d^2a^2+b^2e^2+e^2f^2+f^2b^2+ \\ +d^2c^2+c^2f^2+f^2d^2\le a^4+b^4+c^4+d^4+e^4+f^4$$

which contradicts (2.5). □

Problem 7. (ECN-2010)

Let ABC be an acute triangle with: sides a,b,c, area S, semi-perimeter p, altitudes h_a,h_b,h_c, inradius r, exradii r_a,r_b,r_c, and circumradius R. Prove that:

1.
$$p\cdot r\le h_a\cdot R$$

2.
$$\frac{1}{r_a}+\frac{1}{r_b}+\frac{1}{r_c}=\frac{1}{h_a}+\frac{1}{h_b}+\frac{1}{h_c}$$

3.
$$r_a+r_b+r_c\ge\frac{9S}{p}$$

Proof. (1)

$$p\cdot r\le h_a\cdot R\ \Leftrightarrow\ p\frac{S}{p}\le\frac{2S}{a}R\ \Leftrightarrow\ a\le 2R,$$

which is true because the diameter of the circumcircle is greater than any side of the triangle.

(2)

For the exradius r_A, we have

$$S=S_{ABJ_A}+S_{ACJ_A}-S_{BCJ_A}=\frac{c\cdot r_A}{2}+\frac{b\cdot r_A}{2}-\frac{a\cdot r_A}{2}=\frac{r_A}{2}\left(c+b-a\right);$$

consequently,

$$r_a=\frac{S}{p-a}=\frac{rp}{p-a}.$$

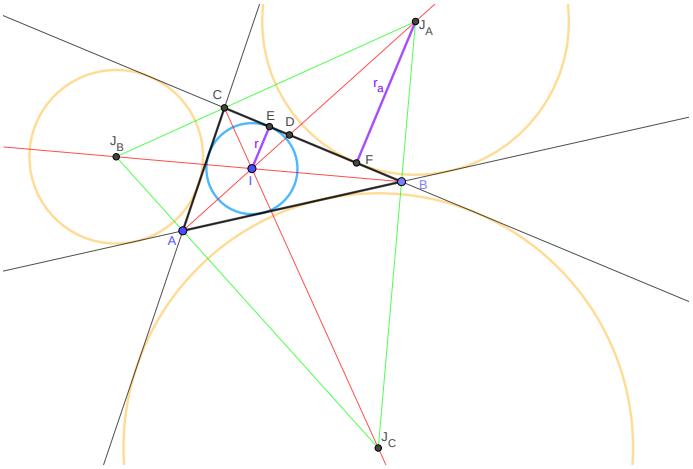


Figure 2.8. Inradius and exradii

Analogously,

$$r_b = \frac{rp}{p-b},\ r_c = \frac{rp}{p-c},$$

so

$$\frac{1}{r_a} + \frac{1}{r_b} + \frac{1}{r_c} = \frac{1}{r}. \tag{2.6}$$

On the other side:

$$S = pr \Leftrightarrow 2S = (a+b+c)\,r,$$

and also

$$2S = a \cdot h_a = b \cdot h_b = c \cdot h_c.$$

Combining this relations, we obtain

$$\frac{1}{r} = \frac{a+b+c}{2S} = \frac{1}{h_a} + \frac{1}{h_b} + \frac{1}{h_c}. \tag{2.7}$$

From (2.6) and (2.7) follows the required equality.

(3)

$$\begin{aligned} r_a + r_b + r_c \geq \frac{9S}{p} \ &\Leftrightarrow\ \frac{S}{p-a} + \frac{S}{p-b} + \frac{S}{p-c} \geq \frac{9S}{p} \\ &\Leftrightarrow p\left(\frac{1}{p-a} + \frac{1}{p-b} + \frac{1}{p-c}\right) \geq 9 \\ &\Leftrightarrow ((p-a)+(p-b)+(p-c))\left(\frac{1}{p-a} + \frac{1}{p-b} + \frac{1}{p-c}\right) \geq 9, \end{aligned}$$

which is true according to Schwarz's inequality:

$$\left(a_1^2+a_2^2+a_3^2\right)\left(b_1^2+b_2^2+b_3^2\right) \geq \left(a_1b_1+a_2b_2+a_3b_3\right)^2,$$

where

$$a_1=\sqrt{p-a}=\frac{1}{b_1}, a_2=\sqrt{p-b}=\frac{1}{b_2}, a_3=\sqrt{p-c}=\frac{1}{b_3}.$$

□

Problem 8. (ECN-2008)

Consider a square $ABCD$. Let $M \in (AB)$ and $N \in (BC)$ be fixed points, and let $P \in (CD)$ and $Q \in (AD)$ be variable points on their respective sides. Determine the positions of P and Q such that the perimeter of quadrilateral $MNPQ$ is minimized. In that case, prove that $MNPQ$ is a trapezoid.

Proof. We use two reflections of the original square – to the right and up. So, we obtain N' and M', reflections of the points N and M (through C and A): $NP = N'P$, $MQ = M'Q$.

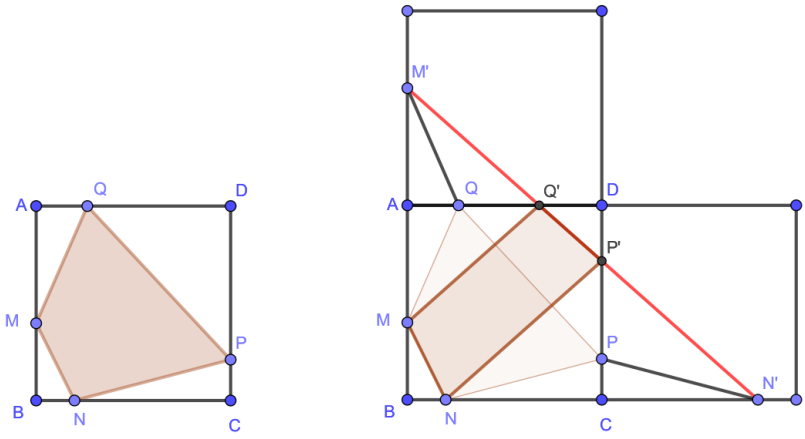


Figure 2.9. Two reflections of the original square

The perimeter of $MNPQ$ contains the fixed MN value, and for the rest of the perimeter we have:

$$NP+PQ+QM = N'P+PQ+QM' \geq N'M'.$$

This means that the sum attains its minimum if the position of P, Q is P', Q' (the intersection of $N'M'$ with the sides CD and DA respectively), so the

quadrilateral with minimum perimeter is $MNP'Q'$. We will prove that this is a trapezoid: $NP' \| MQ'$. In the isosceles triangle NPN' and MQM', we have:

$$\widehat{NP'N'} = \pi - 2 \cdot \widehat{NN'P'},\ \widehat{MQ'M'} = \pi - 2 \cdot \widehat{MM'Q'}.$$

Adding the angles, we obtain

$$\widehat{NP'N'} + \widehat{MQ'M'} = 2\pi - 2\left(\widehat{NN'P'} + \widehat{MM'Q'}\right) = 2\pi - 2\frac{\pi}{2} = \pi,$$

so $NP' \| MQ'$. □

KIVONAT

Az *ECN-feladatok. Utazás a versenyprogramozás világába* című könyv a Sapientia ECN Programozási és Matematikai Verseny válogatott feladatain keresztül vezeti be az olvasót a versenyprogramozás gondolkodásmódjába. A mesterséges intelligencia gyors fejlődése által átalakított világban a kötet az algoritmikus gondolkodás, a kreativitás, az elmélyülés és a hatékony problémamegoldás maradandó értékét hangsúlyozza. Az elméleti alapok, a részletes feladatelemzések, a versenystratégiák és a tapasztalt szerzők tanulságai gyakorlati útmutatást és motivációt nyújtanak diákoknak, programozóknak és mentoroknak egyaránt. A záró rész a Bege Antal Matematikaverseny megoldott feladatain keresztül hangsúlyozza a programozás és a matematika szoros kapcsolatát.

REZUMAT

Probleme ECN: O călătorie în programarea competitivă introduce cititorii în lumea intelectuală a programării competitive prin probleme selectate din Concursul de Programare și Matematică Sapientia ECN. Într-o epocă marcată de dezvoltarea rapidă a inteligenței artificiale, cartea subliniază valoarea durabilă a gândirii algoritmice, a creativității, a profunzimii analitice și a rezolvării eficiente a problemelor. Îmbinând fundamente teoretice cu analize detaliate ale problemelor, strategii de concurs și lecții oferite de autori cu experiență, volumul constituie atât un ghid practic, cât și o sursă de motivație pentru studenți, programatori și mentori. Secțiunea finală evidențiază legătura strânsă dintre programare și matematică prin probleme rezolvate din Concursul Memorial de Matematică „Bege Antal".

ABOUT THE AUTHORS

Ion Cozac – George Emil Palade University of Medicine, Pharmacy, Science, and Technology of Târgu Mureș. Email address: ion.cozac@umfst.ro

Dávid Iclănzan – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences. Email address: iclanzan@ms.sapientia.ro

ORCID iD: 0000-0003-2587-9106

Emil Horobeţ – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: horobetemil@ms.sapientia.ro

ORCID iD: 0009-0009-5582-9020

Zoltán Kátai – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: katai_zoltan@ms.sapientia.ro

ORCID iD: 0000-0003-2343-3629

Lehel István Kovács – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: Klehel@ms.sapientia.ro

ORCID iD: 0009-0005-7129-7678

Gyöngyvér Márton – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: mgyongyi@ms.sapientia.ro

ORCID iD: 0009-0009-6307-3729

Mirtill-Boglárka Naghi – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: naghi.mirtill@ms.sapientia.ro

ORCID iD: 0009-0005-8936-7769

Boróka Olteán-Péter – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: boroka.oltean@ms.sapientia.ro

ORCID iD: 0000-0002-4637-3780

Pálma Rozália Osztián – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences.
Email address: osztian.palma@ms.sapientia.ro

ORCID iD: 0000-0001-6917-8307

Pál Kupán – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences. Email address: kupanp@ms.sapientia.ro

ORCID iD: 0000-0002-9290-3121

Csaba György Pătcaș – Babeș–Bolyai University, Faculty of Mathematics and Computer Science. Email address: csaba.patcas@ubbcluj.ro

ORCID iD: 0009-0001-6485-8765

Zoltán Szabó – Mureș County School Inspectorate; Petru Maior College, Reghin. Email address: szabozoliposta@gmail.com

Róbert Szász – Sapientia Hungarian University of Transylvania, Faculty of Technical and Human Sciences. Email address: rszasz@ms.sapientia.ro

ORCID iD: 0000-0001-8289-6282

Scientia Publishing House
400112 Cluj-Napoca
Matei Corvin St.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Proofreading:
István Szász-Köpeczy

Technical editing:
Erzsébet Csizmadia

Typography:
Elemér Könczey

Series cover:
Tipotéka Kft.

sapientia
textbooks

ECN Problems: A Journey into Competitive Programming invites readers to explore competitive programming not merely as a contest activity but as a powerful way of developing algorithmic thinking, creativity, and systematic problem-solving.

Drawing on the tradition of the Sapientia ECN Programming and Mathematics Contest, organized since 2006, the volume combines theoretical foundations with carefully selected contest problems and detailed solution strategies. It introduces essential approaches such as dynamic programming, divide and conquer, sorting and searching, combinatorial algorithms, number theory, and graph-related techniques, while also addressing contest tactics, preparation strategies, and efficient teamwork.

Rather than presenting solutions alone, the book emphasizes the reasoning that leads to them: identifying the key idea, choosing an appropriate algorithmic approach, analyzing complexity, and turning insight into an efficient implementation. A concluding mathematics section further highlights the close connection between mathematical and algorithmic problem-solving.

The volume is intended for students, contestants, coaches, teachers, and anyone wishing to strengthen their ability to transform challenging problems into elegant and efficient solutions.

ISBN 978-606-975-120-6