Programare cu premeditare

Cătălin Frâncu

Prefață

Aici voi spune ceva introductiv.

- Problemele sînt ordonate după dificultate.
- Pentru Codeforces și Kilonova aveți nevoie de un cont pentru a vedea sursele.
- Am inclus legături către versiunile online ale surselor acolo unde evaluarea este publică. Rapoartele de evaluare arată și timpul și memoria.

Cuprins

I Structuri de date pe vectori **17**

1 Arbori de intervale **20**

1.1 Reprezentare 20

1.1.1 Memoria necesară 21

1.1.2 Reprezentări alternative 22

1.2 Operații elementare 22

1.2.1 Actualizarea punctuală 22

1.2.2 Construcția în $\mathcal{O}(n \log n)$ 23

1.2.3 Construcția în $\mathcal{O}(n)$ 23

1.2.4 Calculul sumei pe interval 23

1.2.5 Căutarea unei sume parțiale 25

1.2.6 Căutarea într-un arbore de maxime 25

1.2.7 Adaptarea la alte tipuri de operații 26

1.2.8 Implementarea recursivă 26

1.3 Probleme 26

1.3.1 Problema Xenia and Bit Operations (Codeforces) 26

1.3.2 Problema Distinct Characters Queries (Codeforces) 27

1.3.3 Problema K-query (SPOJ) 27

1.3.4 Problema Sereja and Brackets (Codeforces) 28

1.3.5 Problema Copying Data (Codeforces) 28

1.3.6 Problema PHF (FMI No Stress 2013) 29

1.3.7 Problema Points (Codeforces) 30

1.3.8 Problema Medwalk (Lot 2025) 30

2 Arbori de intervale cu propagare *lazy* **34**

2.1 Operații pe interval 34

2.2 Implementare recursivă (actualizări punctuale) 38

2.3 Implementare recursivă (actualizări pe interval) 39

2.4 Arta proiectării unui arbore de intervale 40

2.5 Probleme 41

2.5.1 Problema Polynomial Queries (CSES) 41

2.5.2 Problema Nezzar and Binary String (Codeforces) 42

2.5.3 Problema Simple (infO(1)Cup 2019) . . . 42
2.5.4 Problema Balama (Baraj ONI 2024) . . . 43

3 Arbori indexați binar 45
3.1 *Benchmarks* . . . 45
3.1.1 Varianta 1 (*point update, range query*) . . . 46
3.1.2 Varianta 2 (*range update, range query*) . . . 46
3.1.3 Concluzii . . . 46
3.2 Actualizări punctuale și interogări pe interval . . . 47
3.3 Reprezentare . . . 47
3.4 Operația de interogare (suma unui interval) . . . 47
3.5 Operația de actualizare (adăugare pe poziție) . . . 48
3.6 Construcția în $\mathcal{O}(n)$ . . . 49
3.7 Găsirea unei valori punctuale . . . 50
3.8 Căutarea binară a unei sume parțiale . . . 51
3.9 Alte operații decît adunarea . . . 52
3.10 Probleme . . . 53
3.10.1 Problema The Permutation Game Again (SPOJ) . . . 53
3.10.2 Problema Multiset (Codeforces) . . . 53
3.10.3 Problema Hanoi Factory (Codeforces) . . . 54
3.10.4 Problema Subsequences (Codeforces) . . . 54
3.10.5 Problema D-query (SPOJ) . . . 55
3.10.6 Problema Magic Board (CodeChef) . . . 56
3.10.7 Problema Ball (Codeforces) . . . 56
3.10.8 Problema Medwalk, revizitată (Lot 2025) . . . 57
3.11 Interogări punctuale și actualizări pe interval . . . 59
3.12 Interogări și actualizări pe interval . . . 59
3.13 Arbori indexați binar 2D . . . 62
3.13.1 Structura informației . . . 62
3.13.2 Calculul sumei dintr-un dreptunghi . . . 62
3.13.3 Actualizări punctuale . . . 64
3.13.4 Construcția în $\mathcal{O}(n^2)$ . . . 65

4 Descompunere în radical 66
4.1 Actualizări punctuale . . . 66
4.2 Actualizări pe interval . . . 67
4.3 Optimizări . . . 69
4.3.1 Evitați împărțirile! . . . 69
4.3.2 Alegerea mărimii blocurilor . . . 69
4.3.3 Alegerea mărimii blocurilor pentru operații inegale . . . 70
4.4 Probleme . . . 70
4.4.1 Problema Mexitate (ONI 2018 clasa a 9-a) . . . 70

4.4.2 Problema Give Away (SPOJ) . . . 71
4.4.3 Problema Holes (Codeforces) . . . 72
4.4.4 Problema Piezișă (Baraj ONI 2022) . . . 73
4.5 Descompunere după operații . . . 74
4.6 Probleme . . . 75
4.6.1 Problema Serega and Fun (Codeforces) . . . 75
4.7 Procesări diferite înainte și după $\sqrt{n}$ . . . 76
4.8 Probleme . . . 77
4.8.1 Problema Time to Raid Cowavans (Codeforces) . . . 77
4.9 Descompunere în valori distincte . . . 78
4.10 Probleme . . . 78
4.10.1 Problema Sandor (Baraj ONI 2025) . . . 78
4.10.2 Problema Puzzle-bila (Lot 2025) . . . 79

5 Algoritmul lui Mo **82**
5.1 Algoritmul lui Mo fără actualizări . . . 82
5.1.1 Analiză de complexitate . . . 83
5.1.2 Optimizare de viteză . . . 83
5.2 Algoritmul lui Mo cu actualizări . . . 84
5.2.1 Mărimea blocului, ordinea operațiilor, complexitate . . . 84
5.3 Probleme . . . 84
5.3.1 Problema Powerful Array (Codeforces) . . . 84
5.3.2 Problema Most Frequent Value (SPOJ) . . . 85
5.3.3 Problema RangeMode (Infoarena Cup 2013) . . . 85
5.3.4 Problema Machine Learning (Codeforces) . . . 86

II Măiestrie pe biți **88**

6 Operații pe biți. Compactarea variabilelor **89**
6.1 Operații elementare . . . 89
6.1.1 Noțiuni de bază . . . 89
6.1.2 Măști . . . 89
6.1.3 Operații pe măști de biți . . . 90
6.2 Numărarea biților de 1 dintr-o valoare (popcount) . . . 90
6.2.1 Metoda naivă . . . 90
6.2.2 Metoda Kernighan . . . 91
6.2.3 Funcții built-in . . . 91
6.2.4 Tabel precalculat . . . 91
6.2.5 Tabel precalculat + conversie . . . 91
6.2.6 Calcul paralel . . . 91

III Arbori **92**

7 Unelte și algoritmi esențiali **93**
7.1 Generatoare de arbori aleatorii . . . 93
7.2 Probleme . . . 94
7.2.1 Problema Subordinates (CSES) . . . 94
7.2.2 Problema Tree Matching (CSES) . . . 94
7.2.3 Problema Tree Diameter (CSES) . . . 94
7.2.4 Problema Tree Distances II (CSES) . . . 95
7.2.5 Problema White-Black Balanced Subtrees (Codeforces) . . . 95
7.2.6 Problema Blood Cousins (Codeforces) . . . 96

8 Liniarizarea arborilor **98**
8.1 Timpi de intrare și de ieșire din DFS . . . 98
8.2 Testul de strămoș . . . 99
8.3 Liniarizarea. Tipuri de liniarizare . . . 99
8.3.1 Liniarizarea DFS . . . 100
8.3.2 Liniarizarea Euler . . . 100
8.3.3 Liniarizarea Euler cu repetiție . . . 100
8.4 Probleme . . . 101
8.4.1 Problema Tree Queries (Codeforces) . . . 101
8.4.2 Problema Subtree Queries (CSES) . . . 102
8.4.3 Problema Path Queries (CSES) . . . 102
8.4.4 Problema New Year Tree (Codeforces) . . . 102
8.4.5 Problema Max Flow (USACO) . . . 103
8.4.6 Problema Distinct Colors (CSES) . . . 104
8.4.7 Problema Disconnect (Infoarena) . . . 105

9 Tehnica small-to-large **107**
9.1 Generalități . . . 107
9.2 Probleme . . . 107
9.2.1 Problema Fixed-Length Paths I (CSES) . . . 107
9.2.2 Problema Distinct Colors (CSES) (din nou) . . . 108
9.2.3 Problema Lomsat Gelral (Codeforces) . . . 109
9.2.4 Problema Tokens on a Tree (CodeChef) . . . 109
9.2.5 Problema Blood Cousins Return (Codeforces) . . . 110
9.3 DFS exclusiv . . . 111
9.4 Probleme . . . 113
9.4.1 Problema Tree and Queries (Codeforces) . . . 113

10 Cel mai apropiat strămoș comun **115**
10.1 Sumar: RMQ (*range minimum query*) . . . 115

10.1.1 RMQ cu arbore de intervale . . . 116
10.1.2 RMQ cu arbore indexat binar . . . 116
10.1.3 RMQ cu descompunere în radical . . . 116
10.1.4 RMQ cu tabelă rară . . . 116
10.1.5 RMQ cu stivă ordonată . . . 117
10.2 LCA cu liniarizare . . . 117
10.3 LCA cu descompunere în radical . . . 118
10.4 LCA cu binary lifting ($\log n$ pointeri per nod) . . . 119
10.5 LCA cu binary lifting (2 pointeri per nod) . . . 119
10.6 LCA cu algoritmul lui Tarjan (offline) . . . 119
10.7 Benchmarks . . . 120
10.8 Probleme . . . 120
10.8.1 Problema Gold Transfer (Codeforces) . . . 120
10.8.2 Problema A and B and Lecture Rooms (Codeforces) . . . 121
10.8.3 Problema Company (Codeforces) . . . 122
10.8.4 Problema Duff in the Army (Codeforces) . . . 123

11 Algoritmul lui Mo pe arbore 124
11.1 Limitele liniarizărilor . . . 124
11.2 Reducerea la algoritmul lui Mo . . . 124
11.3 Un exemplu . . . 125
11.4 Un caz particular . . . 125
11.5 Descrierea completă . . . 126
11.6 Structura de date necesară . . . 126
11.7 Probleme . . . 126
11.7.1 Problema Dating (Codeforces) . . . 126

12 Descompunere *heavy-light* 129
12.1 Limitările structurilor anterioare . . . 129
12.2 Descompunerea . . . 130
12.3 Detalii de implementare . . . 131
12.4 Probleme . . . 132
12.4.1 Problema Heavy Path Decomposition (Infoarena) . . . 132
12.4.2 Problema Disruption (USACO) . . . 132
12.4.3 Problema Rafaela (Lot 2014) . . . 134
12.4.4 Problema Doi arbori (Lot 2024) . . . 137
12.4.5 Problema Query on a Tree VI (CodeChef) . . . 140
12.4.6 Problema Adă caii (Lot 2025) . . . 141

13 Descompunere în centroizi 143
13.1 Definiție . . . 143
13.2 Proprietăți . . . 143

13.3 Găsirea unui centroid . . . 144
13.4 Descompunerea în centroizi . . . 144
13.5 Probleme . . . 146
13.5.1 Problema Finding A Centroid (CSES) . . . 146
13.5.2 Problema Mystery Tree (CodeChef) . . . 146
13.5.3 Problema Ciel the Commander (Codeforces) . . . 147
13.5.4 Problema Fixed-Length Paths I (CSES) (din nou) . . . 147
13.5.5 Problema Xenia and Tree (Codeforces) . . . 148
13.5.6 Problema Flareon (Lot 2017) . . . 149
13.5.7 Problema Digit Tree (Codeforces) . . . 151

IV Combinatorică 154

14 Elemente de bază 155
14.1 Formule pentru combinări . . . 155
14.2 Calculul combinărilor . . . 156
14.3 Permutări cu repetiție . . . 156
14.3.1 Aplicație . . . 157
14.4 Combinări cu repetiție . . . 157
14.4.1 Aplicații . . . 157
14.5 Numărarea permutărilor fără punct fix . . . 157
14.6 Numerele lui Catalan . . . 158
14.7 Lema lui Burnside . . . 159
14.8 Un puzzle . . . 159
14.9 Probleme . . . 160
14.9.1 Problema Binomial Coefficients (CSES) . . . 160
14.9.2 Problema Creating Strings II (CSES) . . . 160
14.9.3 Problema Distributing Apples (CSES) . . . 160
14.9.4 Problema Christmas Party (CSES) . . . 160
14.9.5 Problema Bracket Sequences I (CSES) . . . 161
14.9.6 Problema Bracket Sequences II (CSES) . . . 161
14.9.7 Problema Counting Necklaces (CSES) . . . 161
14.9.8 Problema Counting Grids (CSES) . . . 161
14.9.9 Problema Number of Permutations (Codeforces) . . . 161
14.9.10 Problema Counting Factorizations (Codeforces) . . . 161
14.9.11 Problema Monocarp and the Set (Codeforces) . . . 163
14.9.12 Problema Devu and Flowers (Codeforces) . . . 163
14.9.13 Problema Lengthening Sticks (Codeforces) . . . 164
14.9.14 Problema Shuffle (Codeforces) . . . 165

15 Rangul permutărilor și al combinărilor 167

15.1 Definiții . . . 167
15.2 Rangul permutărilor . . . 168
15.2.1 Următoarea permutare . . . 168
15.2.2 *Ranking* . . . 168
15.2.3 *Unranking* . . . 168
15.2.4 Implementare . . . 168
15.3 Rangul combinărilor . . . 169
15.3.1 Următoarea combinare . . . 169
15.3.2 *Ranking* lexicografic . . . 170
15.3.3 *Unranking* lexicografic . . . 170
15.3.4 *Ranking* și *unranking* colexicografic . . . 170
15.4 Probleme . . . 172
15.4.1 Problema Misha and Permutation Summation (Codeforces) . . . 172
15.4.2 Problema Inversion Sort (SPOJ) . . . 172
15.4.3 Problema Four Chips (SPOJ) . . . 173
15.4.4 Problema Long Permutation (Codeforces) . . . 173
15.4.5 Problema Arbperm2 (NerdArena) . . . 174
15.4.6 Problema Hipersimetrie (ONI 2019 clasele 11-12) . . . 175
15.4.7 Problema Trim (Baraj ONI 2023) . . . 178

V Geometrie computațională 180

16 Elemente de bază 181
16.1 Principii de implementare . . . 181
16.2 Puncte și vectori . . . 181
16.2.1 Adunare, înmulțire cu o constantă . . . 182
16.2.2 Produsul scalar (engl. *dot product*) . . . 182
16.2.3 Produsul vectorial (engl. *cross product*) . . . 182
16.2.4 Unghiuri . . . 183
16.2.5 Rotații . . . 183
16.3 Drepte și segmente . . . 183
16.3.1 Test de paralelism . . . 184
16.3.2 Test de perpendicularitate . . . 184
16.3.3 Intersecția a două drepte . . . 185
16.3.4 Separarea planului . . . 185
16.3.5 Distanța punct-dreaptă . . . 185
16.3.6 Intersecția a două segmente . . . 185
16.4 Orientare; determinanți . . . 186
16.5 Arii . . . 187
16.5.1 Aria unui poligon oarecare prin metoda trapezelor . . . 187
16.5.2 Aria unui poligon oarecare prin metoda triunghiurilor . . . 188

16.6 Probleme . . . 188
16.6.1 Problema Cuiburi (Baraj ONI 2010) . . . 188
16.6.2 Problema Ace (OJI 2017, clasa a 9-a) . . . 188
16.6.3 Problema Baba Oarba (Lot 2016) . . . 189
16.6.4 Problema Arhitect (OJI 2023, clasa a 10-a) . . . 189
16.6.5 Problema Elicoptere (OJI 2016, clasele 11-12) . . . 190
16.6.6 Problema Arpa and an Exam About Geometry (Codeforces) . . . 190
16.6.7 Problema Bear and Floodlight (Codeforces) . . . 191
16.6.8 Problema TrapEZZ (Moisil++ 2023, clasele 11-12) . . . 191
16.6.9 Problema Terenuri (Baraj ONI 2011) . . . 192
16.6.10 Problema Metin2 (Finala IIOT 2021-22) . . . 193

17 Algoritmi specifici **195**
17.1 Teorema lui Pick . . . 195
17.2 Înfășurătoarea convexă . . . 195
17.3 Algoritmi de baleiere . . . 196
17.4 Șublerul rotitor . . . 196
17.5 Probleme . . . 197
17.5.1 Problema Copaci (Infoarena) . . . 197
17.5.2 Problema Emptri (Lot 2015) . . . 197
17.5.3 Problema Înfășurătoare convexă (Infoarena) . . . 198
17.5.4 Problema Înfășurătoare convexă (NerdArena) . . . 198
17.5.5 Problema Magic (JBOI 2023) . . . 198
17.5.6 Problema Triangular Queries (CodeChef) . . . 200
17.5.7 Problema Hill Walk (USACO Gold 2013) . . . 201
17.5.8 Problema Ydist (Lot 2014) . . . 202
17.5.9 Problema Fossil in the Ice (CodeChef) . . . 205
17.5.10 Problema Rubarba (Infoarena) . . . 206

VI Grafuri **207**

18 Arbori parțiali minimi **208**
18.1 Algoritmii lui Kruskal și Prim . . . 208
18.2 Proprietăți ale arborilor parțiali minimi . . . 209
18.3 Actualizarea APM . . . 209
18.4 Probleme . . . 210
18.4.1 Problema Arbore parțial de cost minim (Infoarena) . . . 210
18.4.2 Problema Autobuze3 (AGM 2015) . . . 210
18.4.3 Problema Rusuoaica (FMI No Stress 9 Warmup) . . . 211
18.4.4 Problema MinOr Tree (Codeforces) . . . 211
18.4.5 Problema 0-1 MST (Codeforces) . . . 212

18.4.6 Problema Minimum Spanning Tree for Each Edge (Codeforces) 213
18.4.7 Problema Apm2 (Infoarena) 214
18.4.8 Problema Edges in MST (Codeforces) 215
18.4.9 Problema Envy (Codeforces) 216
18.4.10 Problema DFS Trees (Codeforces) 216

19 Conectivitate **219**
19.1 Sortarea topologică 219
19.1.1 Algoritmul lui Kahn 219
19.1.2 Algoritmul bazat pe DFS 220
19.2 Componente tare conexe 220
19.2.1 Terminologie 220
19.2.2 Algoritmul lui Kosaraju 221
19.2.3 Algoritmul lui Tarjan 222
19.3 Componente biconexe, punți, puncte de articulație 222
19.3.1 Definiții 222
19.3.2 Puzzles 223
19.3.3 Algoritmul Hopcroft-Tarjan 223
19.4 Probleme 225
19.4.1 Problema Course Schedule (CSES) 225
19.4.2 Problema Book (Codeforces) 225
19.4.3 Problema Componente tare conexe (Infoarena) 225
19.4.4 Problema Mr. Kitayuta's Technology (Codeforces) 226
19.4.5 Problema Ralph and Mushrooms (Codeforces) 226
19.4.6 Problema Bertown Roads (Codeforces) 227
19.4.7 Problema Tourist Reform (Codeforces) 228
19.4.8 Problema We Need More Bosses (Codeforces) 228
19.4.9 Problema Forbidden Cities (CSES) 229
19.4.10 Problema Case of Computer Network (Codeforces) 230
19.4.11 Problema Dog Trick Competition 2 (IIOT 2023-2024 runda 4) 231

VII Probleme diverse **233**

20 Probleme diverse **234**
20.1 Problema Liars (Baraj ONI 2025) 234
20.1.1 Generalități 234
20.1.2 Numărarea configurațiilor 234
20.1.3 Găsirea unei configurații 235

VIII Anexă: Cod-sursă **237**

A Arbori de intervale **238**
A.1 Problema Xenia and Bit Operations (Codeforces) . . . 238
A.2 Problema Distinct Characters Queries (Codeforces) . . . 239
A.3 Problema K-query (SPOJ) . . . 242
A.4 Problema Sereja and Brackets (Codeforces) . . . 246
A.5 Problema Copying Data (Codeforces) . . . 249
A.6 Problema PHF (FMI No Stress 2013) . . . 251
A.7 Problema Points (Codeforces) . . . 253
A.8 Problema Medwalk (Lot 2025) . . . 256

B Arbori de intervale cu propagare *lazy* **262**
B.1 Problema Polynomial Queries (CSES) . . . 262
B.2 Problema Nezzar and Binary String (Codeforces) . . . 268
B.3 Problema Simple (infO(1)Cup 2019) . . . 272
B.4 Problema Balama (Baraj ONI 2024) . . . 276

C Arbori indexați binar **283**
C.1 Problema The Permutation Game Again (SPOJ) . . . 283
C.2 Problema Multiset (Codeforces) . . . 284
C.3 Problema Hanoi Factory (Codeforces) . . . 286
C.4 Problema Subsequences (Codeforces) . . . 288
C.5 Problema D-query (SPOJ) . . . 290
C.6 Problema Magic Board (CodeChef) . . . 293
C.7 Problema Ball (Codeforces) . . . 296
C.8 Problema Medwalk revizitată (Lot 2025) . . . 298

D Descompunere în radical **305**
D.1 Problema Mexitate (ONI 2018 clasa a 9-a) . . . 305
D.2 Problema Give Away (SPOJ) . . . 311
D.3 Problema Holes (Codeforces) . . . 316
D.4 Problema Piezișă (Baraj ONI 2022) . . . 318
D.5 Problema Serega and Fun (Codeforces) . . . 327
D.6 Problema Time to Raid Cowavans (Codeforces) . . . 338
D.7 Problema Sandor (Baraj ONI 2025) . . . 340
D.8 Problema Puzzle-bila (Lot 2025) . . . 344

E Algoritmul lui Mo **348**
E.1 Problema Powerful Array (Codeforces) . . . 348
E.2 Problema Most Frequent Value (SPOJ) . . . 350
E.3 Problema RangeMode (Infoarena Cup 2013) . . . 352
E.4 Problema Machine Learning (Codeforces) . . . 355

F Arbori - probleme esențiale **359**
F.1 Generator simplu de arbori aleatorii . . . 359
F.2 Generator avansat de arbori aleatorii . . . 360
F.3 Problema Subordinates (CSES) . . . 363
F.4 Problema Tree Matching (CSES) . . . 365
F.5 Problema Tree Diameter (CSES) . . . 367
F.6 Problema Tree Distances II (CSES) . . . 369
F.7 Problema White-Black Balanced Subtrees (Codeforces) . . . 371
F.8 Problema Blood Cousins (Codeforces) . . . 372

G Arbori - liniarizare **376**
G.1 Problema Tree Queries (Codeforces) . . . 376
G.2 Problema Subtree Queries (CSES) . . . 378
G.3 Problema Path Queries (CSES) . . . 380
G.4 Problema New Year Tree (Codeforces) . . . 383
G.5 Problema Max Flow (USACO) . . . 387
G.6 Problema Distinct Colors (CSES) . . . 394
G.7 Problema Disconnect (Infoarena) . . . 396

H Arbori - small-to-large **401**
H.1 Problema Fixed-Length Paths I (CSES) . . . 401
H.2 Problema Distinct Colors (CSES) (din nou) . . . 402
H.3 Problema Lomsat Gelral (Codeforces) . . . 404
H.4 Problema Tokens on a Tree (CodeChef) . . . 410
H.5 Problema Blood Cousins Return (Codeforces) . . . 418
H.6 Problema Tree and Queries (Codeforces) . . . 425

I Arbori - cel mai apropiat strămoș comun **432**
I.1 LCA cu descompunere în radical . . . 432
I.2 LCA cu binary lifting ($\log n$ pointeri per nod) . . . 434
I.3 LCA cu binary lifting (2 pointeri per nod) . . . 438
I.4 LCA cu algoritmul lui Tarjan (offline) . . . 444
I.5 Problema Gold Transfer (Codeforces) . . . 446
I.6 Problema A and B and Lecture Rooms (Codeforces) . . . 448
I.7 Problema Company (Codeforces) . . . 451
I.8 Problema Duff in the Army (Codeforces) . . . 455

J Arbori - algoritmul lui Mo pe arbore **460**
J.1 Problema Dating (Codeforces) . . . 460

K Arbori - descompunere *heavy-light* **470**
K.1 Problema Heavy Path Decomposition (Infoarena) . . . 470
K.2 Problema Disruption (USACO) . . . 478

K.3 Problema Rafaela (Lot 2014) . . . 488
K.4 Problema Doi arbori (Lot 2025) . . . 502
K.5 Problema Query on a Tree VI (CodeChef) . . . 514
K.6 Problema Adă caii (Lot 2025) . . . 520

L Arbori - descompunere în centroizi 527
L.1 Problema Finding a Centroid (CSES) . . . 527
L.2 Problema Mystery Tree (CodeChef) . . . 530
L.3 Problema Ciel the Commander (Codeforces) . . . 531
L.4 Problema Fixed-Length Paths I (CSES) (din nou) . . . 535
L.5 Problema Xenia and Tree (Codeforces) . . . 538
L.6 Problema Flareon (Lot 2017) . . . 545
L.7 Problema Digit Tree (Codeforces) . . . 548

M Combinatorică - Elemente de bază 558
M.1 Problema Binomial Coefficients (CSES) . . . 558
M.2 Problema Creating Strings II (CSES) . . . 559
M.3 Problema Distributing Apples (CSES) . . . 561
M.4 Problema Christmas Party (CSES) . . . 562
M.5 Problema Bracket Sequences I (CSES) . . . 564
M.6 Problema Bracket Sequences II (CSES) . . . 565
M.7 Problema Counting Necklaces (CSES) . . . 567
M.8 Problema Counting Grids (CSES) . . . 568
M.9 Problema Number of Permutations (Codeforces) . . . 569
M.10 Problema Counting Factorizations (Codeforces) . . . 571
M.11 Problema Monocarp and the Set (Codeforces) . . . 573
M.12 Problema Devu and Flowers (Codeforces) . . . 575
M.13 Problema Lengthening Sticks (Codeforces) . . . 578
M.14 Problema Shuffle (Codeforces) . . . 579

N Combinatorică - rangul permutărilor și al combinărilor 584
N.1 Rangul permutărilor . . . 584
N.2 Rangul combinărilor . . . 591
N.3 Problema Misha and Permutation Summation (Codeforces) . . . 594
N.4 Problema Inversion Sort (SPOJ) . . . 596
N.5 Problema Four Chips (SPOJ) . . . 598
N.6 Problema Long Permutation (Codeforces) . . . 602
N.7 Problema Arbperm2 (NerdArena) . . . 604
N.8 Problema Hipersimetrie (ONI 2019 clasele 11-12) . . . 606
N.9 Problema Trim (Baraj ONI 2023) . . . 609

O Geometrie - Elemente de bază 613

O.1 Problema Cuiburi (Baraj ONI 2010) . . . 613
O.2 Problema Ace (OJI 2017, clasa a 9-a) . . . 615
O.3 Problema Baba Oarba (Lot 2016) . . . 617
O.4 Problema Arhitect (OJI 2023, clasa a 10-a) . . . 619
O.5 Problema Elicoptere (OJI 2016, clasele 11-12) . . . 622
O.6 Problema Arpa and an Exam About Geometry (Codeforces) . . . 625
O.7 Problema Bear and Floodlight (Codeforces) . . . 625
O.8 Problema TrapEZZ (Moisil++ 2023, clasele 11-12) . . . 627
O.9 Problema Terenuri (Baraj ONI 2011) . . . 631
O.10 Problema Metin2 (Finala IIOT 2021-22) . . . 633

P Geometrie - Algoritmi specifici 637
P.1 Problema Copaci (Infoarena) . . . 637
P.2 Problema Emptri (Lot 2015) . . . 638
P.3 Problema Înfășurătoare convexă (Infoarena) . . . 639
P.4 Problema Înfășurătoare convexă (NerdArena) . . . 641
P.5 Problema Magic (JBOI 2023) . . . 643
P.6 Problema Triangular Queries (CodeChef) . . . 645
P.7 Problema Hill Walk (USACO Gold 2013) . . . 648
P.8 Problema Ydist (Lot 2014) . . . 650
P.9 Problema Fossil in the Ice (CodeChef) . . . 652
P.10 Problema Rubarba (Infoarena) . . . 655

Q Grafuri - Arbori parțiali minimi 659
Q.1 Problema Arbore parțial de cost minim (Infoarena) . . . 659
Q.2 Problema Autobuze3 (AGM 2015) . . . 663
Q.3 Problema Rusuoaica (FMI No Stress 9 Warmup) . . . 665
Q.4 Problema MinOr Tree (Codeforces) . . . 667
Q.5 Problema 0-1 MST (Codeforces) . . . 670
Q.6 Problema Minimum Spanning Tree for Each Edge (Codeforces) . . . 673
Q.7 Problema Apm2 (Infoarena) . . . 676
Q.8 Problema Edges in MST (Codeforces) . . . 678
Q.9 Problema Envy (Codeforces) . . . 682
Q.10 Problema DFS Trees (Codeforces) . . . 685

R Grafuri - Conectivitate 689
R.1 Problema Course Schedule (CSES) . . . 689
R.2 Problema Book (Codeforces) . . . 692
R.3 Problema Componente tare conexe (Infoarena) . . . 697
R.4 Problema Mr. Kitayuta's Technology (Codeforces) . . . 703
R.5 Problema Ralph and Mushrooms (Codeforces) . . . 705
R.6 Problema Bertown Roads (Codeforces) . . . 708

R.7 Problema Tourist Reform (Codeforces) 710
R.8 Problema We Need More Bosses (Codeforces) 712
R.9 Problema Forbidden Cities (CSES) 714
R.10 Problema Case of Computer Network (Codeforces) 716
R.11 Problema Dog Trick Competition 2 (IIOT 2023-2024 runda 4) 719

S Probleme diverse **723**
S.1 Problema Liars (Baraj ONI 2025) 723

Partea I

Structuri de date pe vectori

Următoarele capitole tratează structuri de date care pot procesa anumite operații pe vectori în timp mai bun decît $\mathcal{O}(N)$. Ocazional aceste structuri se aplică și matricilor.

Subiectele de ONI / baraj ONI / lot din anii trecuți abundă în probleme rezolvabile cu astfel de structuri:

- 3dist (baraj ONI 2022)
- 6 de Pentagrame (lot 2024)
- Babel (baraj ONI 2025)
- Balama (baraj ONI 2024)
- Bisortare (ONI 2021)
- Circuit (lot 2025)
- Emacs (baraj ONI 2021)
- Erinaceida (lot 2022)
- Guguștiuc (baraj ONI 2022)
- Împiedicat (baraj ONI 2023)
- Lupușor (ONI 2022)
- Medwalk (lot 2025)
- Perm (baraj ONI 2024)
- Piezișă (baraj ONI 2022)
- Subiectul III (lot 2024)
- Șirbun (baraj ONI 2023)
- Trapez (lot 2025)

Pare o idee bună să le învățăm și să le stăpînim bine. 🙂 Concret, vom studia trei structuri:

1. arbori de intervale;
2. arbori indexați binar;
3. descompunere în radical.

Vom exemplifica structurile și vom face benchmarks pe două probleme didactice. Apoi vom vedea, prin probleme, cum putem extinde aceleași structuri pentru nevoi mai complicate.

Varianta 1 (actualizări punctuale, interogări pe interval): Se dă un vector de N elemente întregi și Q operații de două tipuri:

1. $\langle 1, x, val \rangle$: Adaugă val pe poziția x a vectorului.
2. $\langle 2, x, y \rangle$: Calculează suma pozițiilor de la x la y inclusiv.

Varianta 2 (actualizări pe interval, interogări pe interval): Similar, dar operația 1 este pe interval:

1. $\langle 1, x, y, val \rangle$: Adaugă val pe pozițiile de la x la y inclusiv.
2. $\langle 2, x, y \rangle$: Calculează suma pozițiilor de la x la y inclusiv.

Vom menționa ocazional și **Varianta 3 (actualizări pe interval, interogări punctuale)**:

1. $\langle 1, x, y, val \rangle$: Adaugă val pe pozițiile de la x la y inclusiv.

2. $\langle 2, x \rangle$: Returnează valoarea poziției x.

Toate implementările mele sînt disponibile pe GitHub.

Capitolul 1

Arbori de intervale

Arborii de intervale[1] (AINT) sînt o structură foarte puternică și flexibilă. Ușurința implementării depinde de natura operațiilor pe care dorim să le admitem.

1.1 Reprezentare

Ca multe alte structuri (heap-uri, AIB, păduri disjuncte), arborii de intervale se reprezintă pe un simplu vector. Ei sînt arbori doar la nivel logic, în sensul că fiecare poziție din vector are o altă poziție drept părinte.

Pentru început, să presupunem că vectorul dat are $n = 2^k$ elemente. Atunci vectorul necesar S are $2n$ elemente, în care cele n elemente date sînt stocate începînd cu poziția n. Apoi,

- Cele $n/2$ elemente anterioare stochează valori agregate (sume, minime, xor etc.) pentru perechi de valori din vectorul dat.
- Cele $n/4$ elemente anterioare stochează valori agregate pentru grupe de 4 valori din vectorul dat.
- ...
- Elementul $S[1]$ stochează valoarea agregată a întregului vector.
- Valoarea $S[0]$ rămîne nefolosită.

Iată un exemplu pentru $n = 16$. Datele de la intrare se regăsesc pe pozițiile 16-31.

[1] Există o inversiune între nomenclatura internațională și cea românească. Internațional, structura pe care o învățăm astăzi se numește segment tree, iar interval tree este o structură diferită, care stochează colecții de intervale. Cîțiva ani am înotat împotriva curentului și am fost (posibil) singurul român care se referea la această structură ca „arbori de segmente". În acest curs am adoptat și eu denumirea încetățenită.

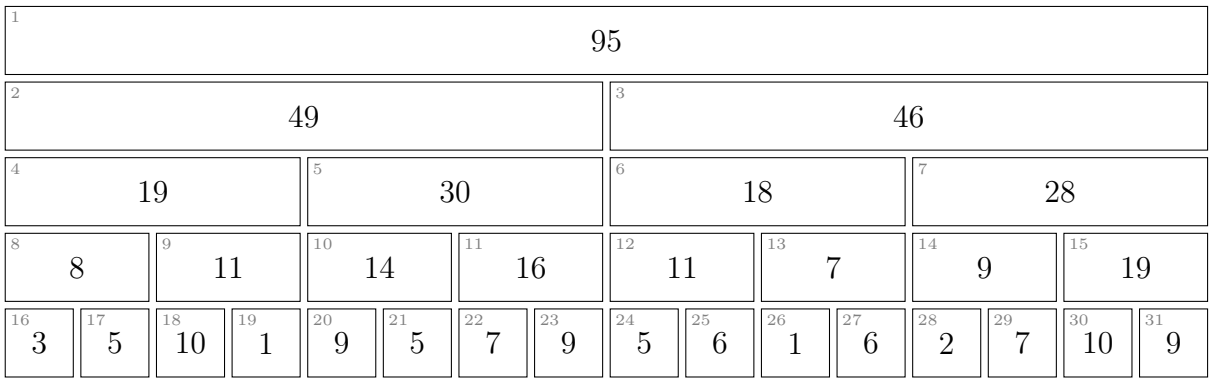


Figura 1.1: Un arbore de intervale cu 16 frunze și 15 noduri interne. Valorile din fiecare celulă reprezintă suma din frunzele subîntinse de acea celulă. Cu cifre mici este notat indicele fiecărei celule.

Facem cîteva observații preliminare:

- Fiii unui nod i sînt $2i$ și $2i + 1$.
- Părintele lui i este $\lfloor i/2 \rfloor$.
- Toți fiii stîngi au numere pare și toți fiii drepți au numere impare.

De exemplu, fiii lui 6 sînt 12 și 13, iar fiii acestora sînt respectiv 24-25 și 26-27. Aceasta corespunde cu intenția noastră ca 6 stocheze informații agregate (suma) despre nodurile 24-27.

După cum vom vedea în secțiunea următoare, arborii de intervale obțin timpi logaritmici pentru operații, deoarece numărul de niveluri este $\log n$.

1.1.1 Memoria necesară

În această formă, structura necesită $2n$ memorie pentru n elemente dacă n este putere a lui 2 sau foarte aproape. De exemplu, pentru $n = 1024$, sînt necesare 2048 de celule. Dar, dacă n depășește cu puțin o putere a lui 2, atunci el trebuie rotunjit în sus. Pentru $n = 1025$, baza arborelui necesită 2048 de celule, iar arborele în întregime necesită 4096 de celule. De aceea spunem că, în cel mai rău caz, arborele poate ajunge la $4n$ celule ocupate în cel mai rău caz.

În realitate, necesarul este doar de $3n$ cu puțină atenție la alocare. Pentru $n = 1025$, alocăm 2048 de celule pentru nivelurile superioare ale arborelui, dar putem aloca fix 1025 pentru bază (nu 2048). Totalul este circa $3n$.

Pentru a calcula următoare putere a lui 2, putem folosi bucla naivă:

```
int p = 1;
while (p < n) {
  p *= 2;
}
n = p;
```

Sau o buclă care folosește *bit hacks*:

```
while (n & (n - 1)) {
  n += n & -n;
}
```

Mai concis, putem folosi funcția `__builtin_clz(x)`, care ne spune cu cîte zerouri începe numărul `x`:

```
n = 1 << (32 - __builtin_clz(n - 1));
```

1.1.2 Reprezentări alternative

Există și reprezentări mai compacte, care ocupă exact $2n-1$ noduri, adică strictul necesar teoretic. Iată un exemplu pentru un vector cu 6 noduri.

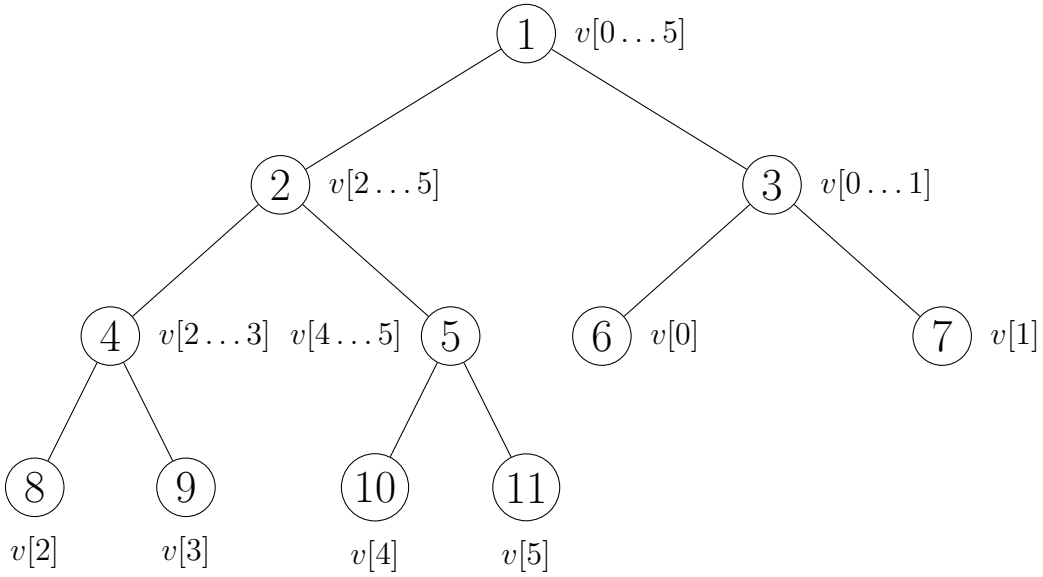


Figura 1.2: Reprezentarea arborilor de intervale cu exact $2n-1$ noduri.

Vedem că frunzele (adică vectorul dat, $v[0] \ldots v[5]$) se află pe pozițiile consecutive 6-11. În schimb, această reprezentare pare mai greu de vizualizat și încalcă o abstracție importantă: frunzele nu mai sînt la același nivel. Structura se pretează la operațiile de actualizare și interogare, dar nu sînt sigur că se pretează și la restul operațiilor pe care le discutăm în secțiunile următoare. De aceea prefer să folosesc și să predau structura rotunjită la 2^k noduri.

1.2 Operații elementare

1.2.1 Actualizarea punctuală

Nu uitați că poziția i din datele de intrare este stocată efectiv în $s[n+i]$. Apoi, cînd elementul aflat pe poziția i primește valoarea val, toate nodurile care acoperă poziția i trebuie recalculate:

```
void set(int pos, int val) {
  pos += n;
  s[pos] = val;
  for (pos /= 2; pos; pos /= 2) {
    s[pos] = s[2 * pos] + s[2 * pos + 1];
  }
}
```

Dacă nu ni se dă noua valoare absolută, ci variația δ față de valoarea anterioară, atunci codul este chiar mai simplu, căci toți strămoșii poziției se modifică tot cu δ:

```
void add(int pos, int delta) {
  for (pos += n; pos; pos /= 2) {
    s[pos] += delta;
  }
}
```

Apropo de *clean code*: Remarcați că am denumit funcțiile `set` și `add`, nu le-am denumit pe ambele `update`. Astfel am evidențiat diferența dintre ele.

1.2.2 Construcția în $\mathcal{O}(n \log n)$

O variantă de construcție este să invocăm funcția set de mai sus pentru fiecare valoare de la intrare. Complexitatea va fi $\mathcal{O}(n \log n)$.

1.2.3 Construcția în $\mathcal{O}(n)$

Putem reduce timpul de construcție dacă doar inserăm valorile frunzelor, fără a le propaga la strămoși. La final calculăm foarte simplu nodurile interne, în ordine descrescătoare.

```
void build() {
  for (int i = n - 1; i >= 1; i--) {
    s[i] = s[2 * i] + s[2 * i + 1];
  }
}
```

1.2.4 Calculul sumei pe interval

Să calculăm suma pe intervalul original [2, 12], care corespunde intervalului [18, 28] din reprezentarea internă. Ideea este să descompunem acest interval într-un număr logaritmic de segmente, mai exact [18,19], [20,23], [24,27] și [28,28]. Avantajul descompunerii este că avem deja calculate sumele acestor intervale, respectiv în nodurile 9, 5, 6 și 28.

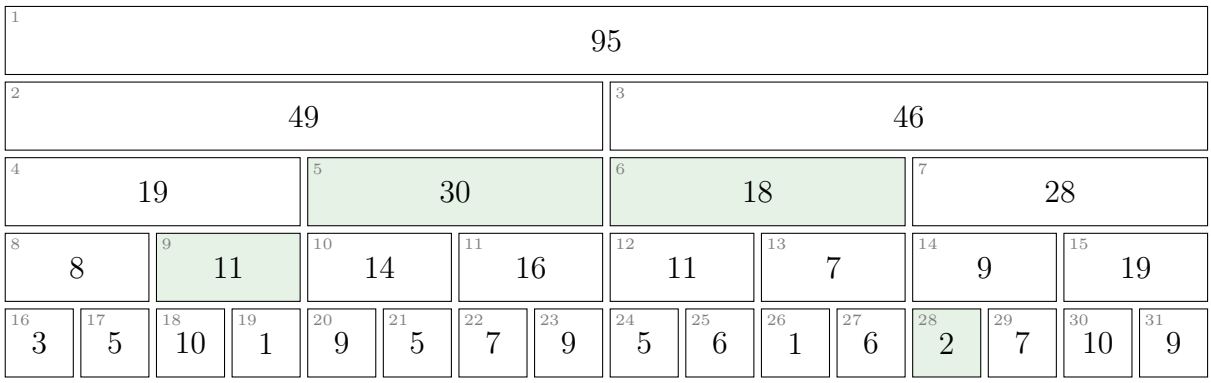


Figura 1.3: Suma intervalului [18, 28] este egală cu suma valorilor nodurilor 9, 5, 6 și 28.

Pornim cu doi pointeri l și r din capetele interogării date. Apoi procedăm astfel:

- Dacă l este fiu stîng, putem aștepta ca să includem un strămoș al său, care va include și alte poziții utile. În schimb, dacă l este fiu drept, trebuie să îl includem în sumă, căci orice strămoș al său va include și elemente inutile din stînga lui l. Apoi avansăm l spre dreapta.
- Printr-un raționament similar, dacă r este fiu stîng, includem valoarea sa în sumă și avansăm r spre stînga.
- Urcăm pe nivelul următor prin înjumătățirea lui l și r.
- Continuăm cît timp $l \leq r$.

Astfel, vom selecta cel mult două intervale de pe fiecare nivel al arborelui și vom restrînge corespunzător intervalul dat, pînă cînd îl reducem la zero. De aici rezultă complexitatea logaritmică.

```
long long query(int l, int r) { // [l, r] închis
  long long sum = 0;

  l += n;
  r += n;

  while (l <= r)  {
    if (l & 1) {
      sum += s[l++];
    }
    l >>= 1;

    if (!(r & 1)) {
      sum += s[r--];
    }
    r >>= 1;
  }

  return sum;
}
```

Clarificare: la ultimul nivel, dacă $l = r$, atunci $s[l]$ va fi selectat exact o dată, fie datorită lui l,

fie datorită lui r, după cum poziția este impară sau pară.

1.2.5 Căutarea unei sume parțiale

Ca și la AIB-uri, dacă toate valorile sînt pozitive are sens întrebarea: pe ce poziție suma parțială atinge valoarea P? Pentru simplitate, recomand să adăugați o santinelă de valoarea infinită pe poziția n. Aceasta garantează că suma parțială se atinge întotdeauna, iar dacă răspunsul este n, atunci de fapt suma parțială nu există în vectorul fără santinelă.

```
int search(int sum) {
  int pos = 1;

  while (pos < n) {
    pos *= 2;
    if (sum > s[pos]) {
      sum -= s[pos++];
    }
  }

  return pos - n;
}
```

1.2.6 Căutarea într-un arbore de maxime

Dat fiind un vector v cu n elemente, ni se cere să răspundem la interogări de tipul $\langle pos, val \rangle$ cu semnificația: găsiți cea mai mică poziție $i > pos$ pe care se află o valoare $v[i] > val$. În secțiunea următoare vom vedea problemele Points și Împiedicat care au această nevoie.

Pentru rezolvare, să construim peste acest vector un arbore de intervale de maxime. Fiecare nod stochează maximul dintre cei doi fii ai săi. Ca urmare, fiecare nod stochează maximul dintre frunzele pe care le subîntinde. Atunci soluția constă din doi pași:

- Mergi la dreapta și în sus, similar pointerului l din operația de sumă pe interval prezentată anterior. Oprește-te cînd ajungi la un nod cu o valoare $> val$. Știm că acest nod subîntinde cel puțin o frunză de valoare $> val$.
- Din acest nod, coboară în fiul care are la rîndul său o valoare $> val$. Dacă ambii fii au această proprietate, coboară în fiul stîng. Oprește-te cînd ajungi la o frunză.

Pentru a simplifica codul, putem adăuga o santinelă infinită la finalul vectorului, ca să ne asigurăm că problema are soluție.

```
int find_first_after(int pos, int val) {
  pos += n + 1;

  while (v[pos] <= val) {
    if (pos & 1) {
```

```
      pos++;
    } else {
      pos >>= 1;
    }
  }

  while (pos < n) {
    pos = (v[2 * pos] > val) ? (2 * pos) : (2 * pos + 1);
  }

  return pos - n;
}
```

Am inclus acest algoritm, deși este rar întîlnit în pratică, pentru a ilustra flexibilitatea uriașă a arborilor de intervale.

1.2.7 Adaptarea la alte tipuri de operații

Aceeași structură de date poate răspunde la multe alte feluri de actualizări și interogări. Nu detaliem aici, vom studia probleme. Ce este important este să ne dăm seama ce stocăm în fiecare nod și cum combină părintele informațiile din cei doi fii.

1.2.8 Implementarea recursivă

Există și o implementare recursivă, pe care nu o vom discuta acum (o menționez doar ca să o fac de rîs). O vom discuta mai tîrziu în acest capitol. Este păcat că mulți elevi învață și stăpînesc doar acea implementare, pe care o aplică și cînd nu este nevoie de ea, deși implementarea iterativă de mai sus este de 2-3 ori mai rapidă. Implementarea iterativă ar trebui să fie implementarea voastră de referință oricînd este suficientă.

Exemplu: din implementarea iterativă rezultă imediat că:

1. Complexitatea este $\mathcal{O}(\log n)$, întrucît l și r urcă exact un nivel la fiecare iterație.
2. De pe fiecare nivel selectăm cel mult două intervale.

Vă urez succes să demonstrați aceste lucruri în implementarea recursivă. 😈

1.3 Probleme

1.3.1 Problema Xenia and Bit Operations (Codeforces)

enunț • sursă

Problema este simplisimă. O includ doar ca exemplu de arbore care face operații diferite pe niveluri diferite.

1.3.2 Problema Distinct Characters Queries (Codeforces)

enunț • sursă

Există diverse abordări pentru această problemă. Una este să construim un AIB sau un AINT pentru fiecare caracter, cu memorie totală $\mathcal{O}(\Sigma n)$ (tradițional Σ denotă mărimea alfabetului). Fiecare structură reține pozițiile pe care apare un caracter. Modificările sînt simple: debifăm poziția în AIB-ul corespunzător vechiului caracter și o marcăm în AIB-ul noului caracter. Pentru interogări, verificăm pentru fiecare din cele 26 de caractere dacă suma pe intervalul dat este non-zero. Rezultă o complexitate de $\mathcal{O}(\Sigma q \log n)$.

Dar iată și o soluție mai elegantă, care reduce complexitatea la $\mathcal{O}(q \log n)$, folosind paralelismul nativ pe 32 de biți al procesorului. Vom folosi 26 de biți din fiecare întreg, cîte unul pentru fiecare caracter. Într-o frunză care stochează litera 'f' vom seta pe 1 doar cel de-al șaselea bit, așadar valoarea întreagă va fi `000...000100000`. Apoi, un nod intern va stoca OR-ul pe biți al frunzelor din intervalul acoperit. Acest gen de informație se numește **mască de biți** (engl. *bitmask*).

Ce semnifică acest OR pe biți? Fiecare dintre biți va fi 1 dacă și numai dacă litera corespunzătoare apare cel puțin o dată în intervalul acoperit. Să observăm că bitul 6 va fi 1 indiferent dacă intervalul conține un caracter 'f' sau multiple caractere 'f'. Rezultă că fiecare mască va avea atîția biți setați (biți 1) cîte caractere distincte există în interval.

Facem actualizări în acest arbore înlocuind masca din frunză și propagînd valoarea spre strămoși cu operația OR. Pentru a răspunde la interogări,

- colectăm cele $\mathcal{O}(\log n)$ măști care compun interogarea;
- le combinăm cu OR;
- numărăm biții din rezultat, de exemplu cu funcția `__builtin_popcount`.

1.3.3 Problema K-query (SPOJ)

enunț • surse

Problema fiind offline, este destul de natural să ordonăm interogările. Sper să vă obișnuiți și voi să luați în calcul această posibilitate.

Ordonarea după capătul stîng sau drept nu pare să ducă nicăieri. Exemplu: ordonăm interogările după capătul drept dr. Atunci, după ce adăugăm elementul $a[dr]$ la structura noastră (oricare ar fi ea), trebuie să răspundem la interogări de tipul: cîte numere $> k$ există începînd cu poziția st? Eu nu am reușit să găsesc o structură echilibrată care să răspundă la întrebări. Poate voi reușiți?

În schimb, ordonarea descrescătoare după valoare duce la o soluție relativ directă. Pentru o interogare (st, dr, k), marcăm (cu 1) într-o structură de date toate pozițiile elementelor mai mari decît k. Apoi numărăm valorile 1 din intervalul $[st, dr]$.

Pentru a găsi rapid toate elementele mai mari decît k (care nu au fost deja inserate în structură), rezultă că trebuie să sortăm și vectorul în ordine descrescătoare, reținînd și poziția originală a

fiecărei valori.

În fapt, putem implementa această soluție chiar și cu un AIB. Sursa este identică cu cea bază pe arbori de intervale cu excepția `struct`-ului. În acest caz, timpii de rulare sînt aproape egali, dar în general vă recomand să folosiți AIB unde se poate.

1.3.4 Problema Sereja and Brackets (Codeforces)

enunț • sursă

Iată și o problemă pentru a cărei rezolvare este mai puțin clar că ne ajunge un arbore de intervale. Vom construi un arbore în care nodurile stochează valori mai complexe care se combină după reguli speciale.

Să considerăm o subsecvență contiguă. Din ce constă ea? Dintr-un subșir (pe sărite) care este bine format, plus niște paranteze deschise neîmperecheate, plus niște paranteze închise neîmperecheate. De exemplu, în subșirul `))) ( () ( () ( () (` am evidențiat cu bold cele 6 caractere bine formate. Rămîn 4 paranteze deschise și 3 închise. Să notăm aceste cantități cu f (lungimea subșirului bine format), d (surplusul de paranteze deschise) și i (surplusul de paranteze închise).

Cum combinăm două subsecvențe adiacente (f_1, d_1, i_1) și (f_2, d_2, i_2)? Clar putem concatena porțiunile bine formate. Dar mai mult, putem prelua și $\min(d_1, i_2)$ perechi dintre surplusurile de paranteze deschise, respectiv închise. Șirul rezultat va fi bine format. Ne putem convinge de asta eliminînd porțiunile bine formate f_1 și f_2, ca și cînd ele nu ar exista. Dacă nu sînteți convinși, puteți apela la o definiție echivalentă pentru un șir de paranteze bine format: pentru orice prefix, diferența dintre numărul de paranteze deschise și închise este pozitivă.

Rezultă că intervalul concatenat va avea parametrii:

- $f = f_1 + f_2 + 2\min(d_1, i_2)$
- $d = d_1 + d_2 - \min(d_1, i_2)$
- $i = i_1 + i_2 - \min(d_1, i_2)$

Construcția arborelui se face ca de obicei, combinînd fiii doi cîte doi. La interogare este nevoie de puțină atenție pentru a colecta și combina intervalele în ordinea corectă (de la stînga la dreapta). Ne bazăm pe observația că operația de compunere nu este comutativă, dar este asociativă.

1.3.5 Problema Copying Data (Codeforces)

enunț • sursă

Aici întîlnim o formă complementară a arborilor de intervale: actualizări pe interval și interogări punctuale (*range update, point query*). Mecanismul necesar folosește o reprezentare puțin diferită. O problemă foarte similară este Range Update Queries (CSES).

(Cei dintre voi care stăpînesc arborii de intervale cu propagare *lazy* vor fi tentați să se repeadă la aceia: Pe fiecare nod ținem informația *lazy* că segmentul din b a fost suprascris cu un segment

din a începînd de la o poziție p (sau cu o deplasare $\pm p$, cum preferați). La actualizări, propagăm informația la fii după nevoie. La interogare, propagăm informația pînă în frunza cerută, pentru a afla de unde provine. Dar nu este nevoie de aceste complicații.)

Să pornim de la observația de bun simț: Dacă o copiere acoperă o poziție, atunci la descompunerea sa în intervale, unul dintre acele intervale va fi strămoș al poziției respective (*duh!*).

Ne vom folosi și de numerele de ordine ale interogărilor, care vor funcționa ca niște momente de timp între 1 și q. Acum, să construim un arbore de intervale care, pentru o operație de copiere (x, y, k):

- Descompune intervalul $[y, y + k - 1]$ prin metoda obișnuită.
- Notează pe fiecare interval momentul t și diferența $x - y$.

Dacă ulterior o altă copiere va acoperi unul dintre aceste intervale, vom nota acolo momentul t' și diferența $x' - y'$. Atunci ultimul moment (și, implicit, ultima proveniență) a suprascrierii unei poziții este dată de cel mai mare moment de timp **dintre toți strămoșii poziției**.

1.3.6 Problema PHF (FMI No Stress 2013)

enunț • sursă

Problema ne cere să simulăm un șir de meciuri de piatră-hîrtie-foarfecă de tip „cîștigătorul la masă" și să admitem actualizări punctuale pe acest șir. Deoarece nu ne permitem o simulare în $\mathcal{O}(n)$ pentru fiecare din cele q actualizări, vom căuta să accelerăm simularea la $\mathcal{O}(\log n)$.

Caracterul de pe fiecare poziție, să-i spunem X, este un meci între X și cîștigătorul meciului de pe poziția anterioară. Echivalent, X este o funcție definită pe mulțimea $\{P, H, F\}$ cu valori tot în $\{P, H, F\}$, unde $X(c)$ este chiar rezultatul unui meci între X și c. De exemplu, P este funcția:

$$\begin{cases} P(P) &= P \\ P(H) &= H \\ P(F) &= P \end{cases}$$

Atunci o înșiruire de caractere este o compunere de funcții. De exemplu, dintr-un șir de intrare de patru caractere, numite generic $XYZT$, îl tratăm pe X ca argument, iar rezultatul final este $T(Z(Y(X)))$ sau $(T \circ Z \circ Y)(X)$.

Orice funcție are nevoie de un argument. 🙂 De aceea, tratăm separat primul caracter, iar pe celelalte $n-1$ le punem într-o structură. (O altă abordare este să definim primul caracter ca pe o funcție care returnează acel caracter independent de intrarea fictivă). Această structură trebuie să mențină rezultatul compunerii caracterelor, cu modificări. Vom folosi un arbore de intervale unde informația dintr-un nod este funcția compusă a intervalului subîntins. Reprezentăm aceste funcții prin tabelul complet (trei valori). Tabelele frunzelor le definim manual, iar tabelul unui nod intern este compunerea tabelelor celor doi fii. Tabelul rădăcinii este ceea ce ne interesează:

compunerea pozițiilor $2 \dots n$ din șir, adică o funcție pe care o vom aplica primului caracter din șir.

Implementarea mea rotunjește numărul de noduri la o putere a lui 2. De aceea la dreapta vom avea și noduri vide, pe care le tratăm ca pe funcții identice ($X(c) = X$).

1.3.7 Problema Points (Codeforces)

enunț • sursă

Problema are rating de 2800 pentru că se compune din multe blocuri, dar niciunul nu este de speriat, căci sîntem deja versați în arbori de intervale. 😎 Aș zice că problema ar fi grea la un baraj ONI sau ușoară la lot.

Ca să putem construi un arbore de intervale, în primul rînd normalizăm coordonatele x. Păstrăm și o tabelă cu valorile originale, căci pe acelea trebuie să le afișăm.

Am putea reformula întrebarea pentru operația `find x y`: dintre toate punctele cu $x' > x$, există vreunul cu $y' > y$? Ne gîndim că am putea folosi un AINT de maxime, indexat după x, cu valori din y, cu interogarea: „Caută maximul pe intervalul $[x + 1, n)$ și spune-mi dacă este mai mare decît y”.

Dar astfel aflăm doar dacă există un punct. Ca să-l găsim, întrebarea corectă este: „dă-mi cea mai din stînga poziție după x pe care maximul depășește y”. Din fericire, putem face asta cu același AINT maxime, așa cum am explicat în secțiunea de teorie:

1. Pornind de la prima poziție validă (în cazul nostru, $x + 1$), mergem în sus și spre dreapta, spre intervale tot mai mari, pînă cînd găsim o poziție de valoare $> y$. Ca să evităm cazurile particulare, adăugăm la finalul vectorului o santinelă de valoare infinită.
2. De la această poziție, coborîm în timp ce menținem în vizor valoarea $> y$. Dacă putem coborî în orice direcție, preferăm stînga.

Astfel putem gestiona operațiile de adăugare (cînd maximul pentru un x fixat poate doar să crească). Următoarea întrebare este cum gestionăm ștergerile. Cea mai directă soluție este să menținem cîte un `set` STL pentru fiecare coordonată x. Suma mărimilor acestor seturi nu va depăși n. Cu metoda `rbegin()` putem afla noul maxim după inserări și ștergeri.

Ultima întrebare, odată ce stabilim că răspunsul pentru `find x y` este la abscisa x', este: care dintre punctele cu această abscisă este răspunsul? Folosim același set și metoda `upper_bound()` pentru a afla cel mai mic y' strict mai mare decît y.

Complexitatea soluției este $\mathcal{O}(n \log n)$, atît pentru normalizarea inițială cît și pentru procesarea operațiilor. Fiecare operație necesită o căutare în set și o căutare sau actualizare în AINT.

1.3.8 Problema Medwalk (Lot 2025)

enunț • sursă

Problema admite și o soluție diferită, mult mai rapidă, bazată pe AIB-uri 2D, dar iată o soluție care folosește doar arbori de intervale.

Din enunț putem defini forma drumului: el va merge pe linia de sus a unor coloane, apoi va folosi ambele linii de pe o coloană c pentru a coborî, apoi va merge pe linia de jos a coloanelor rămase. Acum, să presupunem că avem un oracol care, pentru orice interogare, ne spune coloana c. Atunci vom muta restul coloanelor fie în stînga, fie în dreapta lui c, pentru a folosi valoarea de sus sau de jos, oricare este mai mică.

Cu alte cuvinte, mulțimea de valori de pe drumul care minimizează medianul constă din

- minimele de pe toate coloanele;
- minimul dintre maximele de pe coloane.

Răspunsul la fiecare interogare este elementul median al acestei mulțimi. Logica pentru a afla a k-a valoare este relativ simplă și implică trei valori: al k-lea minim, al $k-1$-lea minim și minimul maximelor. De aici înainte, putem abstractiza matricea ca doi vectori, unul cu minimele perechilor și altul cu maximele. De exemplu, cînd o coloană se modifică din (3, 6) în (3, 2), atunci minimul se modifică din 3 în 2, iar maximul din 6 în 3.

De aceea, avem nevoie de două structuri independente:

- O structură pentru maxime, care să admită actualizări punctuale și interogare de minim pe interval.
- O structură pentru minime, care să admită actualizări punctuale și interogări de al k-lea element pe interval.

Pentru prima structură, ochiul nostru de-acum experimentat ne spune că putem folosi un simplu AINT. Dar pentru a doua? Am găsit pe StackOverflow o idee bine explicată, pe care o reiau.

Vom folosi un arbore de intervale **pe valori**. Așadar, nu indexăm pozițiile conform cu pozițiile din vector, ci cu valorile existente în vector. Fiecare frunză din aint, corespunzătoare unei valori v, reține o colecție ordonată (un `set`, în esență) cu pozițiile pe care apare valoarea v. Fiecare nod intern reține reuniunea colecțiilor fiilor săi. Cu alte cuvinte, dacă un nod subîntinde valorile $[l, r]$, colecția sa va enumera toate pozițiile pe care apar valori între l și r.

Remarcăm că memoria necesară este $\mathcal{O}(n \log V_{max})$, deoarece aint-ul conține V_{max} valori, deci are înălțime $\log V_{max}$, iar fiecare poziție din vectorul original va fi enumerată în $\log V_{max}$ colecții.

Pentru actualizare, trebuie să ștergem poziția modificată din lista vechii valori minime și din listele tuturor strămoșilor. Apoi inserăm poziția în listele noii valori minime. De exemplu, dacă minimul coloanei 100 se modifică din 30 în 20, atunci de la poziția 30 din aint și din toți strămoșii eliminăm elementul 100 din colecție. Apoi la poziția 20 în aint și în toți strămoșii inserăm elementul 100.

Rămîne să descriem interogările. Pentru a afla al k-lea minim dintr-un interval de coloane $[l, r]$, pornim din rădăcina arborelui de intervale (luînd așadar în calcul toate valorile de la 0 la V_{max}). Consultăm fiul stîng (valorile $1 \ldots V_{max}/2$) și ne întrebăm: cîte apariții au aceste valori pe poziții din $[l, r]$? Putem răspunde la această întrebare printr-o diferență, reducînd întrebarea la forma:

cîte apariții au aceste valori pe pozițiile $0 \dots r$? Așadar, trebuie numărate elementele mai mici sau egale cu r din setul rădăcinii. Set-ul simplu din STL nu poate gestiona această întrebare, dar putem folosi un set extins din PB/DS. Nu detaliem acum, dar ne vom reîntîlni cu acest tip de date.

Dacă numărul de valori între 0 și $V_{max}/2$ care apar pe poziții între l și r este $\geq k$, atunci acolo se va afla și al k-lea element, deci coborîm în fiul stîng. Altfel coborîm în fiul drept.

Complexitatea algoritmului este $\mathcal{O}((n+q) \log n \log V_{max})$. De exemplu, fiecare interogare coboară $\log V_{max}$ niveluri, iar la fiecare nivel face o căutare într-un set de $\mathcal{O}(n)$ elemente în timp $\mathcal{O}(\log n)$.

Bibliografie

[1] CS Academy, *Segment Trees*, URL: https://csacademy.com/lesson/segment_trees.

[2] CP Algorithms, *Segment Tree*, URL: https://cp-algorithms.com/data_structures/segment_tree.html.

Capitolul 2

Arbori de intervale cu propagare *lazy*

2.1 Operații pe interval

Să reluăm exemplul din capitolul trecut și să spunem acum că dorim să adăugăm 100 pe intervalul [2, 12], corespunzător nodurilor [12, 28] din arbore.

Ca să nu facem efort $\mathcal{O}(n)$, vom descompune intervalul ca mai înainte și vom nota informația „+100" în nodurile 9, 5, 6 și 28, cu semnificația că valoarea reală a tuturor frunzelor de sub aceste noduri a crescut cu 100.

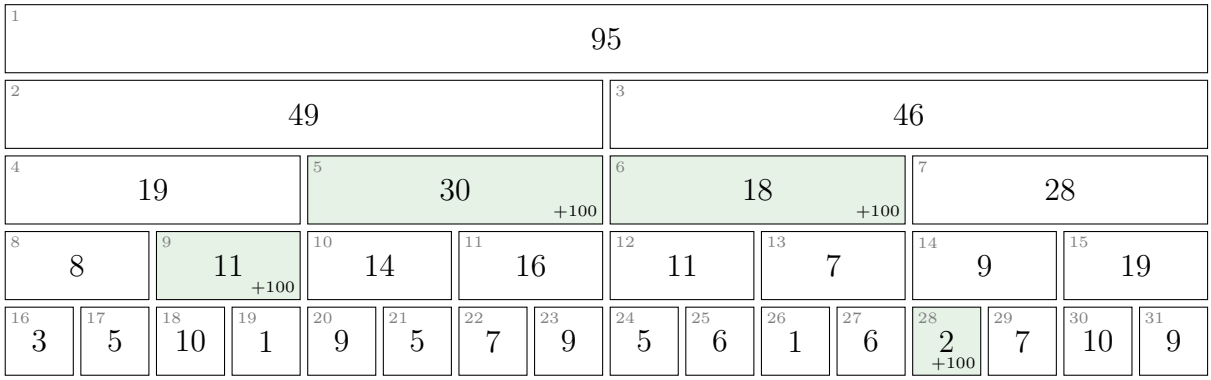


Figura 2.1: Pentru a adăuga 100 pe intervalul [18, 28], notăm valoarea *lazy* 100 pe nodurile 9, 5, 6 și 28.

Aceasta este o **informație *lazy***: o informație care stă într-un nod intern și care trebuie propagată tuturor frunzelor subîntinse de acel nod. Totuși, amînăm efortul acestei propagări pînă cînd el devine strict necesar; tocmai de aceea se numește **propagare *lazy***. (Mulți elevi denumesc întreaga structură „AINT cu *lazy*", dar asta este... lene.)

Evaluarea *lazy* este un concept des întîlnit:

- Memoizarea unor valori într-un vector / matrice, cu speranța că nu va fi nevoie să calculăm tabelul complet.
- Amînarea evaluării lui `y` în expresia booleană `x || y`, cu speranța că `x` va fi evaluat ca adevărat, iar `y` va deveni irelevant.

- Inițializarea unei componente costisitoare dintr-un program doar cînd devine necesară (o conexiune la baza de date, o zonă a hărții dintr-un joc).

Așadar, definim un al doilea vector numit `lazy` și executăm `lazy[x] += 100` pe pozițiile 9, 5, 6 și 28.

Motivul pentru care treaba se complică este următorul. Dacă acum primim o interogare de sumă pe intervalul [25,29]? Nu putem să însumăm, ca de obicei, pozițiile 25, 13 și 14, căci pierdem din vedere că unele dintre noduri au (cîte) +100. Sigur, putem lua asta în calcul, dar trebuie să clarificăm operațiile, altfel efortul poate deveni $\mathcal{O}(n)$.

În primul rînd, introducem două funcții noi (le puteți include în alte funcții, dar pentru claritate le puteți declara de sine stătătoare):

- `push()`, care propagă informația *lazy* de la un nod la fiii săi;
- `pull()`, care combină în părinte informația din cei doi fii după o actualizare.

```
void push(int node, int size) {
  s[node] += lazy[node] * size;
  lazy[2 * node] += lazy[node];
  lazy[2 * node + 1] += lazy[node];
  lazy[node] = 0;
}

void pull(int node, int size) {
  s[node] =
    s[2 * node] + lazy[2 * node] * size / 2 +
    s[2 * node + 1] + lazy[2 * node + 1] * size / 2;
}
```

Pentru problema dată (dar nu pentru toate problemele), codul are nevoie să știe numărul de frunze subîntinse (`size`).

Să presupunem acum că dorim să calculăm suma intervalului [2, 12] și că este posibil să avem niște sume *lazy* în multe alte noduri. Știm că codul descompune interogările în intervale mai scurte și nu urcă mai sus de acestea. Dacă există valori *lazy* mai sus (să zicem în rădăcină), codul nu va afla de ele. De aceea, în pregătirea interogării, trebuie să vizităm toți strămoșii intervalului și să propagăm în jos (*push*) informația *lazy*. Dar, dacă ne gîndim, lista completă a acestor strămoși constă doar din strămoșii capetelor de interval! Pentru intervalul [18,28], este nevoie să propagăm în jos informația *lazy* din strămoșii lui 18 (adică 1, 2, 4 și 9) și ai lui 28 (adică 1, 3, 7 și 14).

Dacă apelăm push din acești strămoși, de sus în jos, avem garanția că informația pe intervalele dorite este la zi. Vă rămîne vouă ca experiment de gîndire să demonstrați că, după operațiile *push*, nu va mai exista informație *lazy* în niciun strămoș al niciunei poziții din interogare.

Astfel obținem o funcție foarte similară cu cea din capitolul trecut

```
void push_path(int node, int size) {
```

```
  if (node) {
    push_path(node / 2, size * 2);
    push(node, size);
  }
}

long long query(int l, int r) {
  long long sum = 0;
  int size = 1;

  l += n;
  r += n;
  push_path(l / 2, 2); // pornim din părinte
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      sum += s[l] + lazy[l] * size;
      l++;
    }
    l >>= 1;

    if (!(r & 1)) {
      sum += s[r] + lazy[r] * size;
      r--;
    }
    r >>= 1;
    size <<= 1;
  }

  return sum;
}
```

Dacă viteza este crucială, putem scrie și o funcție `push_path` cu circa 10% mai rapidă, iterativă, folosind operații pe biți. Să considerăm nodul $22 = 10110_{(2)}$. Strămoșii lui sînt 1, 2, 5 și 11 care au respectiv reprezentările binare 1, 10, 101 și 1011, care sînt fix prefixele lui 10110! Deci îl vom deplasa pe 10110 la dreapta cu 4, 3, 2 și respectiv 1 bit pentru a-i obține strămoșii.

```
void push_path(int node) {
  int bits = 31 - __builtin_clz(n);
  for (int b = bits, size = n; b; b--, size >>= 1) {
    int x = node >> b;
    push(x, size);
  }
}

// Acum primul apel este chiar din frunză:
...
push_path(l);
```

```
push_path(r);
...
```

Actualizările sînt foarte similare. Apelăm `pull()` după terminarea actualizărilor, deoarece trebuie să lăsăm arborele într-o stare coerentă și trebuie să preluăm orice modificare de la fii.

```
void pull_path(int node) {
  for (int x = node / 2, size = 2; x; x /= 2, size <<= 1) {
    pull(x, size);
  }
}

void update(int l, int r, int delta) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      lazy[l++] += delta;
    }
    l >>= 1;

    if (!(r & 1)) {
      lazy[r--] += delta;
    }
    r >>= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}
```

Iată și o altă implementare care combină bucla `while` principală cu funcția `pull_path`.

Notă: În această implementare, valoarea *lazy* se aplică întregului subarbore, inclusiv nodului însuși. În implementarea de pe CP Algorithms, valoarea *lazy* se aplică subarborelui fără nodul însuși. Oricare dintre formulări este acceptabilă, cîtă vreme o folosiți consecvent.

Notă: În practică, cîmpurile `lazy` și `s` merită încapsulate într-un `struct`. Datorită localității acceselor la memorie, diferența de viteză este notabilă (circa 25%). Aici le-am lăsat separate pentru concizie.

2.2 Implementare recursivă (actualizări punctuale)

Lecția trecută am spus că există și o implementare recursivă. Să o examinăm acum (mulți o știți deja).

```
void update(int node, int pl, int pr, int pos, int delta) {
  if (pr - pl == 1) {
    s[node] += delta;
  } else {
    int mid = (pl + pr) >> 1;
    if (pos < mid) {
      update(2 * node, pl, mid, pos, delta);
    } else {
      update(2 * node + 1, mid, pr, pos, delta);
    }
    s[node] = s[2 * node] + s[2 * node + 1];
  }
}
```

Metoda recursivă cară după ea 5 parametri:

- `node`: nodul curent din arbore (aka poziția în vector);
- `pl`, `pr`: intervalul din vectorul inițial acoperit de `node`. Eu am optat pentru implementarea cu `pl` inclusiv și `pr` exclusiv. Dacă preferați intervale închise, este OK.
- `pos`, `delta`: poziția de modificat și valoarea de adăugat/scăzut.

Vedem că funcția coboară recursiv în fiul stîng sau fiul drept, după caz. Un exemplu de apel ar fi:

```
update(1, 0, n, some_pos, some_val);
```

Mai interesant, iată și implementarea funcției de interogare (sumă pe interval):

```
long long query(int node, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return s[node];
  } else {
    int mid = (pl + pr) >> 1;

    return
      query(node * 2, pl, mid, l, min(r, mid)) +
      query(node * 2 + 1, mid, pr, max(l, mid), r);
  }
}
```

Regăsim trei din aceiași parametri, `node`, `pl` și `pr`. În plus,

- `l`, `r`: Intervalul [închis, deschis) pe care dorim să calculăm suma.

Funcția se reapelează pe cei doi fii, restrîngînd corespunzător intervalul $[l, r)$. Iată o imagine care arată arborele de apeluri pentru calculul sumei pe intervalul [3, 10):

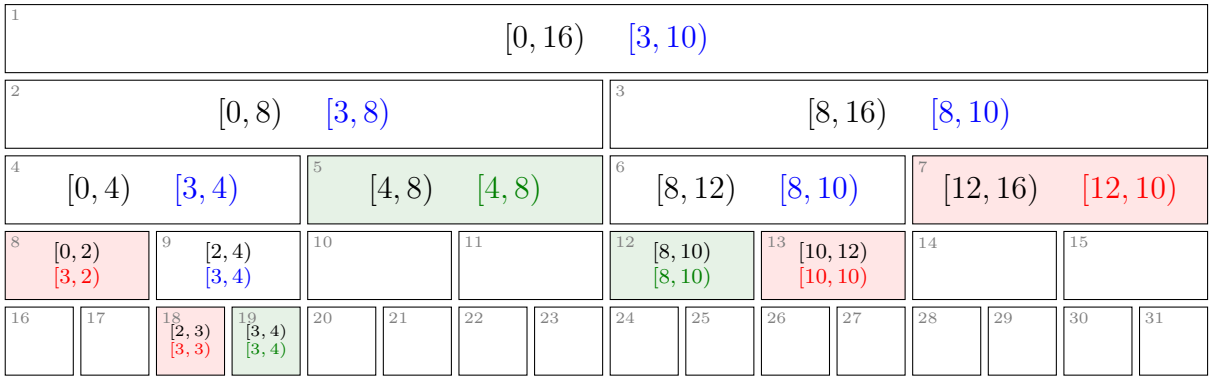


Figura 2.2: Arborele de apeluri în funcția de actualizare recursivă pe intervalul [3, 10).

Am ilustrat cu albastru nodurile care au nevoie să-și apeleze descendenții, cu verde nodurile selectate integral, iar cu roșu nodurile eliminate.

Complexitatea rămîne $\mathcal{O}(\log n)$, deși funcția se reapelează pentru ambii fii. De ce?

Discutăm implementarea recursivă pentru că ea plutește prin supa culturală și vreau să puteți citi cod scris astfel. Dar ea este un exemplu de dopaj, de implementare repetată *mot à mot* indiferent de nevoile problemei. Implementarea recursivă este de 2-3 ori mai lentă decît cea iterativă pentru actualizări punctuale. Presupun că există două motive:

Implementarea recursivă cară după ea 5-6 parametri la fiecare apel, care trebuie copiați, puși/scoși de pe stivă etc. Implementarea iterativă folosește doar 3 variabile.

Implementarea recursivă este nevoită să pornească din rădăcină, să coboare pînă la frunze, apoi să revină din recursivitate. Implementarea iterativă se oprește imediat ce termină de descompus intervalul $[l, r]$.

La varianta cu propagare *lazy* diferența de timp aproape dispare, pentru că ambele implementări trebuie să urce pînă la rădăcină.

Nu vă năpustiți la implementarea recursivă dacă nu este nevoie. Rezistați tentației de a fi leneși, de a învăța o singură structură de date, pe care să o pictați indiferent de situație! Trebuie să aspirați la mai mult de atît, dacă este să vă meritați locul în lot.

2.3 Implementare recursivă (actualizări pe interval)

În această implementare, observăm cum:

- apelăm `push` înainte de reapelarea recursivă, pentru a-i garanta fiecărui nod că deasupra sa nu mai există informații lazy;

- apelăm `pull` după revenirea din recursivitate, ca să lăsăm arborele într-o stare coerentă.

```
long long query(int node, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return s[node] + lazy[node] * (r - l);
  } else {
    push(node, pr - pl);
    int mid = (pl + pr) >> 1;
    return
      query(node * 2, pl, mid, l, min(r, mid)) +
      query(node * 2 + 1, mid, pr, max(l, mid), r);
  }
}

void update(int node, int pl, int pr, int l, int r, int delta) {
  if (l >= r) {
    return;
  } else if ((l == pl) && (r == pr)) {
    lazy[node] += delta;
  } else {

    push(node, pr - pl);
    int mid = (pl + pr) >> 1, child_size = (pr - pl) >> 1;
    update(2 * node, pl, mid, l, min(r, mid), delta);
    update(2 * node + 1, mid, pr, max(l, mid), r, delta);
    pull(node, child_size);

  }
}

void process_ops() {
  for (int i = 0; i < num_queries; i++) {
    if (q[i].t == OP_UPDATE) {
      update(1, 0, n, q[i].l - 1, q[i].r, q[i].val);
    } else {
      answer[num_answers++] = query(1, 0, n, q[i].l - 1, q[i].r);
    }
  }
}
```

2.4 Arta proiectării unui arbore de intervale

Atunci cînd primim o problemă și avem de proiectat o structură de date, cum știm dacă un arbore de intervale se potrivește scopului? În general, trebuie să urmărim trei lucruri.

În primul rînd, pentru a putea procesa rapid actualizările pe interval, dorim să facem efort $\mathcal{O}(1)$

în fiecare interval elementar din descompunere. De aceea, în general informația *lazy* stochează fix ce primim de la operațiile de *update*: o cantitate de adăugat sau de atribuit, un bit de semn, un bit care arată că intervalul curent trebuie inversat etc.

Este nevoie de atenție la compunerea actualizărilor. Dacă avem două cantități de adăugat pe același interval, cîmpul *lazy* va reține suma cantităților. Dacă avem două atribuiri succesive pe același interval, cîmpul *lazy* o va reține doar pe ultima.

În al doilea rînd, pentru a putea procesa rapid interogările pe interval, dorim să facem efort $\mathcal{O}(1)$ în fiecare interval din descompunere. De aceea, informația propriu-zisă din fiecare nod trebuie să includă valorile pe care le cer operațiile de *query*. Acestea sînt un punct de pornire, dar uneori nu sînt suficiente, ci este nevoie să menținem mai multe valori din care să le putem alege pe cele cerute.

În sfîrșit, în al treilea rînd trebuie să verificăm că avem tot ce ne trebuie pentru a compune două intervale alăturate la interogare, pentru a recalcula un părinte din cei doi fii ai săi (operația *pull*) și pentru a propaga informația *lazy* de la părinte la fii (operația *push*).

O regulă de aur este că, la începutul și la sfîrșitul fiecărei funcții, arborele trebuie să fie într-o stare **coerentă**. Aceasta înseamnă că trebuie să definim un **contract**, o promisiune despre care este structura logică a fiecărui nod. Vă recomand chiar să notați acest contract într-un comentariu de una-două fraze, la începutul codului pentru AINT. Apoi, scrieți codul astfel încît, la intrarea și la ieșirea din orice funcție, toate nodurile arborelui să respecte acel contract.

De exemplu, dacă informația *lazy* este o cantitate de adăugat pe tot subarborele, atunci operația *push* trebuie neapărat să se încheie prin a pune pe 0 valoarea *lazy* din părinte. În niciun caz nu trebuie să încheiem operația *push* lăsînd aceeași valoare *lazy* în părinte și în fii.

2.5 Probleme

2.5.1 Problema Polynomial Queries (CSES)

enunț • surse

Problema seamănă mult cu cea discutată la teorie, dar pe intervale nu mai adăugăm constante, ci progresii aritmetice. Așadar, pare natural să reținem exact această informație *lazy*: în fiecare nod reținem că în fiecare frunză acoperită de acel nod trebuie să adăugăm cîte un termen al unei progresii cu un anumit prim element și pasul (deocamdată) 1. De exemplu, dacă în figura 2.1 facem o actualizare pe intervalul [18, 28], atunci în nodul 5 notăm progresia cu primul termen 3 și pasul 1. Informația *lazy* este o pereche $\langle 3, 1 \rangle$.

Trebuie tratate atent diversele cazuri care iau naștere. Dacă două progresii acoperă același interval, vor lua naștere progresii cu pas mai mare decît 1. Să luăm un exemplu:

- Progresia cu primul termen 5 și pasul 3, așadar 5, 8, 11, 14, ...
- Progresia cu primul termen 2 și pasul 7, așadar 2, 9, 16, 23, ...

- După însumare dorim să avem termenii 7, 17, 27, 37, ...
- Rezultă că suma este și ea o progresie cu primul termen 7 și pasul 10. Cu alte cuvinte, informațiile *lazy* se pot compune ușor: $\langle 5, 3\rangle + \langle 2, 7\rangle = \langle 7, 10\rangle$.

La propagarea în jos a informației *lazy*, în cei doi fii vom adăuga progresii cu același pas. În fiul drept, primul termen trebuie calculat, dar este ușor. Dacă într-un nod care acoperă 16 elemente avem o progresie cu primul element 3 și pasul 5, atunci fiul drept va începe cu al nouălea termen al progresiei:

$$3 + 5 \cdot (16/2) = 43$$

La operațiile de adăugare, pe toate intervalele din descompunere vom aduna progresii cu pasul 1, dar primul element diferă pentru fiecare interval (la fel, nu este greu de calculat).

Contractul pe care l-am ales pentru implementarea iterativă este:

- `first` și `step` înseamnă că pe nodurile din intervalul acoperit trebuie adăugate valorile `first`, `first + step`, `first + 2 * step`, ...
- Valoarea `s` din fiecare nod **nu** include și suma progresiei dată de `<first, step>` din acel nod.

2.5.2 Problema Nezzar and Binary String (Codeforces)

enunț • sursă

Problema este relativ directă odată ce „ne prindem” că trebuie să procesăm operațiile în ordine inversă. Știm șirul final f și fie $[l, r]$ ultimul interval inspectat de Nanako. În momentul inspecției, șirul curent trebuia să fie identic cu f pe pozițiile $[1, l) \cup (r, n]$, căci pe acelea nu le putem modifica. Pe pozițiile $[l, r]$ trebuiau să fie doar biți 0 sau doar biți 1. Care dintre ele? Știm că la ultima modificare am modificat strict mai puțin de jumătate din biți. Să notăm cu z numărul de zerouri și cu u numărul de unu de pe pozițiile $[l, r]$ din f. Iau naștere trei cazuri:

1. Dacă $z > u$, înseamnă că la pasul anterior $[l, r]$ conținea doar 0.
2. Dacă $z < u$, înseamnă că la pasul anterior $[l, r]$ conținea doar 1.
3. Dacă $z = u$, problema nu are soluție, căci nu putem opera modificarea necesară.

Astfel, toate operațiile sînt forțate, mergînd înapoi în timp. Răspunsul este `YES` doar dacă putem procesa toate operațiile, iar la final ajungem la șirul s.

Rezultă că, pentru a efectua efectiv operațiile, avem nevoie de un arbore de segmente cu valori de 0 și 1 în frunze, cu funcții de sumă pe interval (pentru a stabili majoritatea) și de atribuire pe interval (pentru a face *undo* la o operație).

2.5.3 Problema Simple (infO(1)Cup 2019)

enunț • sursă

Să aplicăm regula menționată ca să proiectăm un arbore de intervale pentru această problemă.

- În cîmpul *lazy* vom stoca valorile primite la update, așadar cantitățile de adăugat pe tot subarborele.
- În cîmpurile propriu-zise vom stoca valorile necesare pentru interogări, așadar minimul par pe interval și maximul impar pe interval.
- Ce altceva ne mai trebuie ca să putem menține informația la actualizări? Să observăm că o cantitate *lazy* impară schimă paritatea valorilor pe întregul interval. Noul minim par este fostul minim impar, plus cantitatea *lazy*. De acea, vom introduce încă două cantități: maximul par și minimul impar.

Ca de obicei, este important ca la implementare să alegem dacă valoarea *lazy* este deja inclusă în nodul curent și mai trebuie aplicată doar la subarbore sau dacă ea trebuie aplicată inclusiv nodului curent. Eu am ales prima variantă. Ambele sînt bune, cîtă vreme codul respectă alegerea făcută.

Pe unele intervale nu vor exista valori pare sau impare. Dacă facem cazuri speciale pentru toate acele situații, vom avea undeva între 5 și 10 `if`-uri de presărat prin cod. O abordare mai simplă este să notăm în acele noduri $+\infty$ pentru a arăta că nu există minime și $-\infty$ pentru a arăta că nu există maxime. Apoi lăsăm aceste valori să se combine fără să mai tratăm cazuri particulare. La final, știm că orice valori definite vor fi între 1 și $2 \cdot 20^9 + 2 \cdot 20^5 \cdot 2 \cdot 20^9$, conform limitelor din enunț. Valorile din afara acestui interval le interpretăm ca fiind nedefinite.

În cod am încercat să separ funcțiile specifice nodului de funcțiile specifice arborelui.

2.5.4 Problema Balama (Baraj ONI 2024)

enunț • surse

Vă veți întîlni des cu probleme unde soluția devine simplă dacă analizăm informațiile în altă ordine. În cazul de față, în loc să luăm în calcul liniile (care sînt subsecvențe ordonate), să analizăm coloanele.

Care va fi răspunsul pe ultima coloană? Desigur, va fi maximul din vector. Mult mai interesantă este întrebarea: care va fi răspunsul pe penultima coloană? Va fi cel mai mare element care este vreodată (în cel puțin o fereastră) **al doilea maxim**.

Exemplu: fie maximul global 1.000 și fie al doilea maxim global 999. Dacă 999 stă foarte departe de 1.000, la distanță de cel puțin k, atunci 999 va fi maxim în toate ferestrele de lățime k care îl conțin. Cu alte cuvinte, 999 se va regăsi doar pe ultima coloană în matricea B (și va fi mascat de 1.000). Să spunem că următoarele valori din șir, în ordine descrescătoare, sînt 998, 997 și 996 și toate se află la distanță mare unele de altele. Niciunul dintre ele nu va apărea pe penultima coloană în B.

În schimb, să spunem că următoarea valoare ca mărime, 995, se află aproape (la distanță $< k$) de o valoare anterioară, cum ar fi 997. Atunci există o fereastră în care 995 și 997 coexistă, deci 995 va fi al doilea maxim din acea fereastră și va apărea pe penultima coloană în B. Cum alte valori mai mari nu au această proprietate, 995 este răspunsul pe penultima poziție a soluției.

(Amănunt esențial 😉: 995 nu poate fi și al treilea maxim. Dacă exista o fereastră de lățime k care îl cuprindea pe 995 și alte două valori anterioare, atunci înainte să ajungem la 995 una dintre acele două valori anterioare ar fi fost al doilea maxim).

Astfel, putem considera elemente în ordine descrescătoare și, pentru fiecare element x ne întrebăm: cîte elemente văzute anterior conține fiecare dintre ferestrele care îl conțin pe x (cel mult k la număr)? Dacă o astfel de fereastră are e elemente, și dacă a $e+1$-a valoare din soluție (numărînd de la dreapta) este încă necunoscută, atunci pune x pe poziția $e+1$ a soluției.

Exemplu: Dacă considerăm elementul 900 și constatăm că într-una din ferestrele care îl conțin pe 900 existau deja alte 5 valori, atunci în acea fereastră 900 este al 6-lea element. Dacă a 6-a poziție din soluție este încă neocupată, scriem 900 acolo, acesta fiind maximul posibil.

Implementarea sună fioros, dar nu este! În realitate avem nevoie de o structură cu două operații:

- Incrementează pozițiile de la st la dr, pentru a arăta că în ferestrele de la $[st, st+k-1]$ și pînă la $[dr, dr+k-1]$ avem cîte un element în plus.
- Află maximul de pe pozițiile de la st la dr.

Operația a doua ne este suficientă deoarece ferestrele nu vor ajunge brusc la 6 elemente. Vor apărea mai întîi ferestre cu 1, 2, 3, 4, 5 elemente. Cu alte cuvinte, soluția se completează de la dreapta spre stînga.

Vom implementa un AINT de maxime în care informația *lazy* din fiecare nod este valoarea de adăugat pe fiecare poziție din intervalul acoperit.

Capitolul 3

Arbori indexați binar

Arborii indexați binar (AIB), numiți și arbori Fenwick, iar în engleză *binary indexed trees (BIT)*, servesc ca și arborii de intervale tot la rezolvarea în $\mathcal{O}(\log n)$ a unor operații pe vectori. Ei sînt mai puțin flexibili și universali decît arborii de intervale. Nu toate problemele rezolvabile cu AINT pot fi rezolvate și cu AIB. Dar acolo unde se potrivesc, AIB-urile sînt ușor de codat și sînt de 2-3 ori mai rapide decît arborii de intervale.

3.1 *Benchmarks*

Dacă se potrivesc mai multe structuri, contează pe care o alegem? Ca să alegem în cunoștință de cauză, iată niște măsurători de viteză (*benchmarks*). Le-am făcut în 2025 pe un procesor AMD Ryzen 7 4700U, la acea vreme comparabil cu evaluatoarele de la Kilonova și Codeforces.

Am măsurat timpii de rulare pentru diverse implementări ale problemei în ambele variante (actualizări punctuale sau pe interval).

- arbori indexați binar, $\mathcal{O}(\log n)$ per operație;
- arbori de segmente iterativi, $\mathcal{O}(\log n)$ per operație;
- arbori de segmente recursivi, $\mathcal{O}(\log n)$ per operație;
- descompunere în radical, $\mathcal{O}(\sqrt{n})$ și cel mult două împărțiri per operație;
- descompunere în radical, $\mathcal{O}(\sqrt{n})$ și $\mathcal{O}(\sqrt{n})$ împărțiri per operație.

Precizez că vom discuta descompunerea în radical abia în capitolul următor, dar pare un moment bun să privim aceste *benchmarks*.

Am ales limitele $n = q = 500.000$ pentru ambele variante ale problemei. Pentru unele programe contează cîte dintre operații sînt interogări și cîte sînt actualizări. Pentru aceste situații, am măsurat doi timpi, notați astfel:

- 250u/250q: există cîte 250.000 de operații din fiecare tip;
- 100u/400q: există 100.000 de actualizări și 400.000 de interogări.

Toate testele sînt pe `long long` și 90% dintre intervale au lungime peste $n/2$. Toți timpii măsoară

strict partea de procesare (excluzînd citirea și scrierea).

3.1.1 Varianta 1 (*point update, range query*)

structură	**timp**
arbore indexat binar	23 ms
arbore de intervale iterativ (250u/250q)	46 ms
arbore de intervale iterativ (100u/400q)	55 ms
arbore de intervale recursiv (250u/250q)	116 ms
arbore de intervale recursiv (100u/400q)	130 ms
descompunere în radical (250u/250q)	113 ms
descompunere în radical (100u/400q)	177 ms
descompunere în radical cu împărțiri (250u/250q)	763 ms
descompunere în radical cu împărțiri (100u/400q)	1.219 ms

Tabela 3.1: Timpii de rulare pentru sume pe interval și actualizări punctuale.

3.1.2 Varianta 2 (*range update, range query*)

structură	**timp**
arbore indexat binar (250u/250q)	66 ms
arbore indexat binar (100u/400q)	58 ms
arbore de intervale iterativ (250u/250q)	183 ms
arbore de intervale iterativ (100u/400q)	175 ms
arbore de intervale recursiv (250u/250q)	211 ms
arbore de intervale recursiv (100u/400q)	203 ms
descompunere în radical (250u/250q)	235 ms
descompunere în radical (100u/400q)	274 ms

Tabela 3.2: Timpii de rulare pentru sume pe interval și actualizări pe interval.

3.1.3 Concluzii

Reținem că:

- Arborii indexați binar sînt de departe cei mai rapizi.
- Pentru actualizări punctuale, arborii de intervale iterativi sînt de două ori mai rapizi decît cei recursivi.
- Descompunerea în radical ține binișor pasul cu arborii de segmente. Din experiență, aceasta este o particularitate a problemei alese. Pentru alte probleme diferența poate fi mai mare.
- Împărțirile îngreunează enorm descompunerea în radical.

3.2 Actualizări punctuale și interogări pe interval

3.3 Reprezentare

AIB-ul descompune informația în felul următor: Poziția k din vector stochează suma ferestrei de p elemente care se termină la poziția k, unde p este cea mai mare putere a lui 2 care îl divide pe k. De exemplu, pentru $k = 40$, $p = 8$. Așadar, pe poziția 40 AIB-ul va stoca suma celor 8 valori de pe pozițiile $[33 \dots 40]$.

Iată un exemplu care arată sus valorile pe care dorim să le reținem, iar jos valorile concrete pe care ajunge să le stocheze vectorul. Subliniez că AIB-ul nu folosește memorie suplimentară, ci doar stochează diferit informația în același vector. Suspectez că și de aici provine eficiența lui în raport cu arborii de intervale.

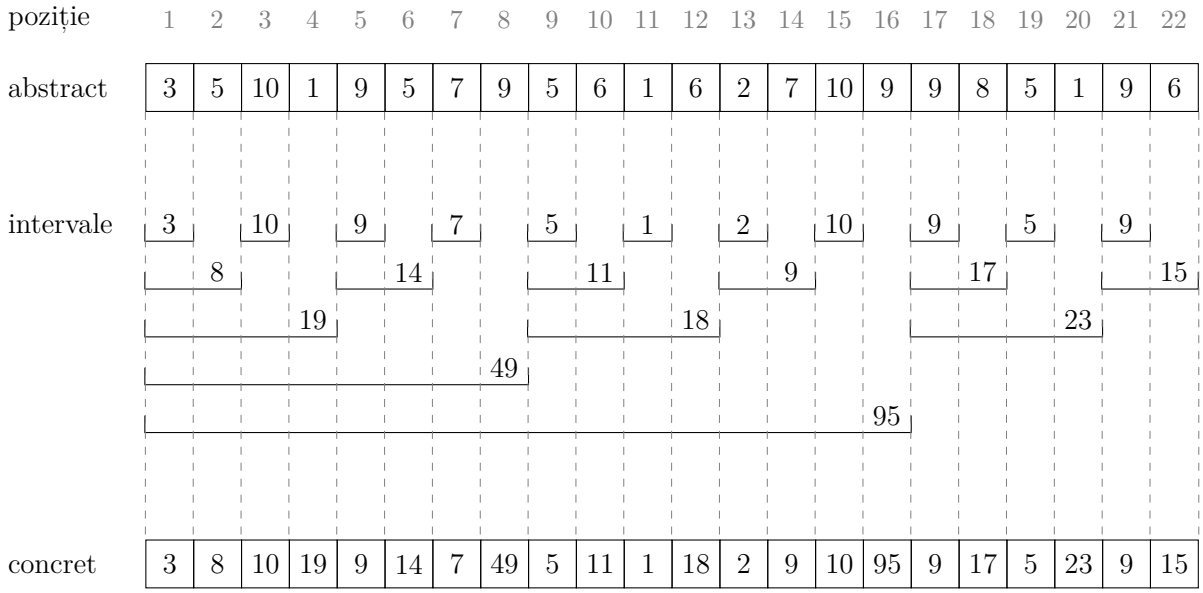


Figura 3.1: Un arbore indexat binar cu 22 de poziții. Vectorul de sus este cel abstract, iar vectorul de jos este cel stocat concret în memorie. Fiecare interval arată pozițiile a căror sumă o notăm în capătul din dreapta al intervalului.

3.4 Operația de interogare (suma unui interval)

AIB-urile tratează interogările pe un interval oarecare $[x, y]$ prin diferența a două interogări pe prefix, $[1, y]$ și $[1, x-1]$. Pentru a răspunde la o interogare pe prefix, de exemplu suma pe intervalul $[1, 21]$, descompunem acel prefix în intervale dintre cele stocate în AIB, respectiv $[1, 16]$, $[17, 20]$ și $[21, 21]$. Odată ce includem o poziție x și tot intervalul pe care îl acoperă ea, pentru a ajunge la următoarea poziție de însumat trebuie, prin definiție, să scădem cea mai mare putere a lui 2 care îl divide pe x. Rezultă codul:

```
struct fenwick_tree {
  int v[MAX_N + 1]; // indexare de la 1
```

```
  int prefix_sum(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  int range_sum(int from, int to) {
    return prefix_sum(to) - prefix_sum(from - 1);
  }
};
```

Expresia `pos &= pos - 1` elimină cel mai din dreapta bit de 1 dintr-un număr; de exemplu, din $20 = 10100_{(2)}$ ea obține $16 = 10000_{(2)}$. Pe cazul general,

```
pos             = abc...xyz1000...000
pos - 1         = abc...xyz0111...111
pos & (pos - 1) = abc...xyz0000...000
```

De aici rezultă și complexitatea $\mathcal{O}(\log n)$, căci reprezentarea oricărei poziții în baza 2 are cel mult $\log n$ biți de 1.

3.5 Operația de actualizare (adăugare pe poziție)

La actualizarea pe o poziție, trebuie actualizate toate intervalele care conțin acea poziție. De exemplu, la actualizarea poziției 11 trebuie actualizate intervalele $[11, 11]$, $[9, 12]$ și $[1, 16]$, așadar trebuie recalculate pozițiile 11, 12 și 16 din AIB. Această parte pare magică: de ce pozițiile 13, 14 și 15 nu trebuie actualizate? Dar ne putem convinge că, cît ne îndepărtăm de poziția inițială (11), ne interesează doar pozițiile responsabile de intervale suficient de mari încît să acopere poziția 11.

```
void add(int pos, int val) {
  do {
    v[pos] += val;
    pos += pos & -pos;
  } while (pos <= n);
}
```

Pentru a înțelege expresia `pos & -pos`, să facem o scurtă digresiune. Numerele cu semn sînt reprezentate în calculator în complement față de 2. Aceasta înseamnă că, pentru a reprezenta un număr negativ,

- Reprezentăm întîi numărul pozitiv (valoarea absolută).

- Îi inversăm toți biții.
- Adăugăm 1.

De exemplu, pentru a îl reprezenta pe -20 procedăm astfel:

- Îl reprezentăm pe +20: `000...00010100`.
- Îi inversăm toți biții: `111...11101011`.
- Adăugăm 1: `111...11101100`.

Această reprezentare are două avantaje:

1. Putem folosi aceleași circuite logice pentru operații pe numere cu sau fără semn.
2. Reprezentările lui $+0$ și -0 sînt identice, `000...000`. În complement față de 1 există două reprezentări, `000...000` și `111...111`, ceea ce este straniu.

Revenind, observăm acum că `20 & -20` este `000...00000100`, adică formula `pos & -pos` izolează ultimul bit, adică mărimea intervalului subîntins de `pos`. Prin adăugarea acestei cantități la `pos`, obținem intervalul imediat următor care include poziția `pos`.

Dacă din orice motiv această expresie vă scapă din memorie, puteți inventa pe loc formule echivalente, de exemplu:

```
pos = (pos | (pos - 1)) + 1;
```

Complexitatea este tot $\mathcal{O}(\log n)$ deoarece la fiecare pas eliminăm cel puțin un bit 1 din reprezentarea binară a lui pos.

3.6 Construcția în $\mathcal{O}(n)$

Dat fiind un vector-sursă `src`, este tentant să construim arborele într-un al doilea vector apelînd de n ori rutina de adăugare:

```
void build(int* src) {
  for (int i = 1; i <= n; i++) {
    add(i, src[i]);
  }
}
```

Această metodă cere timp $\mathcal{O}(n \log n)$. Există însă o metodă care refolosește vectorul și rulează în $\mathcal{O}(n)$:

```
void build() {
  for (int i = 1; i <= n; i++) {
    int j = i + (i & -i);
    if (j <= n) {
      v[j] += v[i];
    }
```

```
  }
}
```

Explicație: fiecare element `v[i]` este propagat doar la următorul element `j` care include poziția `i`. Este treaba acelui segment să propage adaosul și mai departe. Pentru scenariul relativ comun în care construim AIB-ul, apoi facem n interogări și n actualizări, construcția în $\mathcal{O}(n)$ reduce costul de rulare cu circa 20%.

Paradoxal, tocmai fiindcă este atît de rapid, costul de funcționare al unui AIB este adesea înecat de costul altor operații (în special intrarea/ieșirea). De aceea optimizările sînt greu de observat. Dar *the hacker spirit* ne obligă să folosim oricum soluția inteligentă.

3.7 Găsirea unei valori punctuale

Pentru a găsi valoarea pe o singură poziție `k`, o putem calcula în $\mathcal{O}(\log n)$ ca pe `sum(k) - sum(k - 1)`. Sau, desigur, putem păstra o copie a vectorului real. Dar există și o implementare în $\mathcal{O}(1)$ amortizat.

Să considerăm poziția $k = 60$. Dacă o calculăm prin diferența sumelor parțiale, obținem

```
val(60) = sum(60) - sum(59) = (v[60]          + v[56] + v[48] + v[32]) -
                              (v[59] + v[58] + v[56] + v[48] + v[32])
                            = v[60] - (v[59] + v[58])
```

Se vede că, de la poziția 56 încolo, sumele de intervale se anulează în cele două paranteze. Nu este o coincidență. Fie:

```
k     = bbb...bbb10000
k - 1 = bbb...bbb01111
```

Unde `b` sînt niște biți oarecare, iar poziția k se termină într-un bit 1 urmat de cîțiva (posibil 0) biți de 0. Atunci, în calculul sumelor parțiale, pozițiile k și $k-1$ vor elimina biți de la coadă pînă cînd vor ajunge la strămoșul comun, care este

```
str   = bbb...bbb00000
```

De aceea, codul este:

```
int get_value_at(int pos) {
  int result = v[pos];
  int ancestor = pos & (pos - 1);
  pos--;
  while (pos != ancestor) {
    result -= v[pos];
    pos &= pos - 1;
  }
  return result;
```

```
}
```

Acest cod pare tot logaritmic. În realitate, jumătate din valorile din AIB (cele de pe poziții impare) stochează chiar valoarea în acel punct, deci bucla din `get_value_at()` va face 0 iterații. Un sfert din valorile din AIB vor face o iterație, o optime dintre ele vor face două iterații. În general, pentru o poziție k, funcția `get_value_at()` va face atîtea iterații cîte zerouri are la coadă reprezentarea binară a lui k. Media acestei valori este 1 (așadar constantă) dacă distribuția lui k este uniformă și aleatorie.

3.8 Căutarea binară a unei sume parțiale

Dacă vectorul (abstract) are doar valori non-negative, atunci sumele parțiale sînt nedescrescătoare și are sens întrebarea: Care este prima poziție pe care se atinge suma parțială S?

Putem face o căutare binară naivă: examinăm suma parțială la poziția $n/2$, apoi la una dintre pozițiile $n/4$ sau $3n/4$ după caz, etc. Dar fiecare dintre aceste interogări durează $\mathcal{O}(\log n)$, deci complexitatea totală a algoritmului va fi $\mathcal{O}(\log^2 n)$. Dar iată și o metodă în $\mathcal{O}(\log n)$, similară cu căutarea binară prin „metoda Mihai Pătrașcu” (ca să adoptăm nomenclatura din supa culturală olimpică).

Fie p cea mai mare putere a lui 2 cel mult egală cu n. Pentru exemplul inițial, $n = 22$, deci $p = 16$. Observația-cheie este că putem afla suma parțială pe pozițiile $[1 \ldots p]$ printr-o singură operație: ea este exact `v[p]`! După cum `v[p] < S` sau `v[p] > S`, ne îndreptăm atenția către pozițiile din stînga sau din dreapta lui p. Următoarea interogare o vom face la `v[p / 2]` sau la `v[3 * p / 2]`.

În practică, invariantul este: `pos` reprezintă cea mai mare poziție cunoscută pe care suma parțială **nu** atinge valoarea S. La final, funcția returnează `pos + 1`. De asemenea, ținem cont că nu întotdeauna putem avansa la dreapta (nu putem depăși valoarea n).

```
int bin_search(int sum) {
  int pos = 0;

  for (int interval = max_p2; interval; interval >>= 1) {
    if ((pos + interval <= n) && (v[pos + interval] < sum)) {
      sum -= v[pos + interval];
      pos += interval;
    }
  }

  return pos + 1;
}
```

Exemplu: pentru AIB-ul din figura 3.1 și suma parțială 72, algoritmul funcționează astfel:

- Verifică poziția 16. Suma este 95, prea mare.

- Verifică poziția 8. Suma este 49. Așadar, căutăm suma parțială 72 - 49 = 23 începînd dincolo de poziția 8.
- Verifică poziția 12. Suma este 18. Așadar, căutăm suma parțială 23 - 18 = 5 începînd dincolo de poziția 12.
- Verifică poziția 14. Suma este 9, prea mare.
- Verifică poziția 13. Suma este 2. Așadar, căutăm suma parțială 5 - 2 = 3 începînd dincolo de poziția 13.
- Răspunsul este poziția 14.

Aici, `max_p2` este cea mai mare putere a lui 2 care nu depășește n. O putem afla naiv, de exemplu astfel:

```
max_p2 = n;
while (max_p2 & (max_p2 - 1)) {
  max_p2 &= max_p2 - 1;
}
```

, sau într-o singură linie cu funcția `__builtin_clz(n)`, care returnează numărul de zerouri la stînga lui n:

```
max_p2 = 1 << (31 - __builtin_clz(n));
```

Corolar: putem folosi căutarea binară pentru a afla prima poziție cu o valoare nenulă într-un AIB. Aceasta este poziția pe care suma parțială atinge valoarea 1. Similar putem afla ultima poziție cu o valoare nenulă. Mai trebuie doar să menținem și suma elementelor din AIB, ceea ce cere două linii de cod.

Corolar: dacă AIB-ul ține valori de 0 și 1, putem folosi căutarea binară pentru a afla poziția celui de-al k-lea bit 1 (în engleză această valoare se numește *k-th order statistic*). Ea este fix poziția pe care suma parțială atinge valoarea k. Multe probleme de permutări, care necesită evidența elementelor văzute / nevăzute, se încadrează aici (exemplu: codificarea / decodificarea permutărilor).

3.9 Alte operații decît adunarea

Arborii Fenwick pot gestiona și alte operații, cîtă vreme ele sînt **inversabile**. Reamintesc că **suma** pe intervalul $[l \ldots r]$ se calculează ca **diferența** sumelor pe intervalele $[1 \ldots r]$ și $[1 \ldots l-1]$. Deci operația inversă (scăderea) trebuie să fie definită. Exemplu: operația xor, operația de înmulțire modulo un număr prim etc.

Deoarece operația *max* nu este inversabilă, AIB-urile nu suportă, pe cazul general, operațiile:

1. actualizare punctuală;
2. maxim pe interval.

Totuși, putem folosi AIB-uri pentru interogări de maxim dacă următoarele condiții sînt adevărate:

1. Toate interogările sînt pe prefix (capătul stînga este întotdeauna 1).
   - Sau toate interogările sînt pe sufix, caz în care putem reflecta toți indicii față de n.
2. Prin actualizare, valorile pot doar să crească.

Temă de gîndire: De ce este necesar ca toate valorile să crească? Ce poate să meargă prost dacă într-un AIB de maxime valorile pot să și scadă?

Raționamente similare putem aplica pentru operațiile *min*, *and* și *or*. Cum reformulăm condiția 2 pentru aceste operații?

Un exemplu mai extravagant este operația *or* pe `bitset`-uri, vezi problema Erinaceida.

3.10 Probleme

3.10.1 Problema The Permutation Game Again (SPOJ)

enunț • sursă

Problema ne cere să aflăm **rangul** unei permutări (engl. *rank*). Acesta este numărul de ordine al permutării în lista ordonată lexicografic a tuturor permutărilor mulțimii $\{1, 2, \ldots, n\}$.

Echivalent, trebuie să răspundem eficient la întrebarea: cîte permutări vin înaintea celei date în lista permutărilor?

Să considerăm un exemplu. Dacă primul element al permutării este 9, atunci toate permutările care încep cu $1, 2, \ldots, 8$ o vor preceda în listă. Există $8(n-1)!$ astfel de permutări.

Similar, dacă al doilea element este 3, atunci toate permutările care încep cu 91 sau 92 o vor preceda în listă. Există $2(n-2)!$ astfel de permutări.

Dar dacă al treilea element este 7? Acum trebuie să ținem cont de faptul că pe 3 l-am văzut deja. Trebuie să socotim permutările care încep cu $931, 932, 934, 935, 936$. Există $5(n-3)!$ astfel de permutări.

Cu alte cuvinte, trebuie să răspundem eficient la întrebarea: cîte elemente mai mici decît cel curent am văzut în prefixul dinaintea elementului curent? Putem gestiona această informație cu un AIB de 0 și 1. Cînd procesăm un element de valoare x, adunăm 1 pe poziția x în AIB. Astfel, suma parțială din AIB pe o poziție y ne va arăta cîte elemente mai mici decît y am procesat pînă în prezent.

Pentru un plus de eficiență, sursa nu reține întreaga permutare, ci doar citește cîte un element, îl ia în calcul la rang, îl bifează în AIB, apoi îl aruncă.

3.10.2 Problema Multiset (Codeforces)

enunț • sursă

„Aproape” putem rezolva problema cu un singur vector de frecvențe. Dar avem nevoie să găsim eficient al k-lea element ca să-l putem șterge. Un vector de frecvențe ne dă inserări în $\mathcal{O}(1)$, dar ștergeri în $\mathcal{O}(n)$.

De aceea, înlocuim vectorul cu un AIB în care pe poziția x notăm frecvența lui x în multiset. Astfel putem căuta al k-lea element reformulînd definiția: al k-lea element este poziția p pe care suma parțială atinge sau depășește valoarea k.

3.10.3 Problema Hanoi Factory (Codeforces)

enunț • sursă

Pare natural să sortăm inelele descrescător după diametrul exterior. Ce facem la egalitate? Toate inelele de același diametru exterior pot fi stivuite, caz în care îl vom prefera deasupra pe cel cu diametrul interior minim, ca să ne maximizăm șansele de a putea pune alt inel deasupra lui. Așadar, ca departajare, sortăm inelele descrescător după diametrul interior.

Acum orice turn valid va fi un subșir din șirul sortat, pe sărite, dar fără reordonare. Și atunci putem defini relativ ușor o recurență calculabilă în $\mathcal{O}(n^2)$. Fie H_i înălțimea maximă a unui turn care are în vîrf inelul i. Atunci inelul aflat imediat sub i, fie el j, respectă condițiile $j < i$ și $in_j < out_i$. Așadar,

$$H_i = h_i + \max_{j<i,\ in_j<out_i} H_j$$

Pentru a reduce complexitatea la $\mathcal{O}(n \log n)$, procesăm inelele de la stînga la dreapta. Atunci $j < i$ este întotdeauna respectată și trebuie doar să răspundem la întrebarea: dintre toate inelele cu $in_j <$ un x dat (unde $x = out_i$), care este valoarea maximă pentru H_j? Putem răspunde la întrebare cu un AIB de maxime, indexat după diametrele interioare, pe care îl interogăm despre maximul pe pozițiile $[1 \dots out_i - 1]$. După calcularea lui H_i, optimizăm maximul din AIB pe poziția in_i cu valoarea H_i.

Diametrele pot fi mari, dar le putem normaliza în intervalul $[1 \dots 2n]$.

Există și o soluție mai ingenioasă, cu o stivă ordonată, care nu face obiectul acestui capitol.

3.10.4 Problema Subsequences (Codeforces)

enunț • sursă

Problema pare abordabilă cu programare dinamică. Să căutăm întîi formula de recurență, care nu este dificilă. Fie $C_{l,i}$ numărul de subsecvențe crescătoare de lungime l terminate pe poziția i. Atunci:

$$C_{l,i} = \sum_{j<i,\ a_j<a_i} C_{l-1,j}$$

Implementarea în $\mathcal{O}(kn^2)$ este așadar directă. Cum procedăm să reducem calculul sumei de la $\mathcal{O}(n)$ la $\mathcal{O}(\log n)$? Observăm aici un mecanism pe care îl vom regăsi și la alte probleme. Pare că dorim o interogare bidimensională (suma valorilor $C_{l-1,j}$ pe poziții unde $j < i$ și simultan $a_j < a_i$). În realitate, însă, putem reduce interogarea la una unidimensională.

Să inserăm într-un AIB valorile $C_{l-1,1} \dots C_{l-1,i-1}$. Atunci condiția $j < i$ este automat satisfăcută și dorim suma valorilor pe pozițiile unde $a_j < a_i$. De aceea, vom indexa AIB-ul nu după j, ci după a_j. Cu alte cuvinte, vom scrie $C_{l-1,j}$ nu la poziția j, ci la poziția a_j. Desigur, după ce calculăm $C_{l,i}$ adăugăm și $C_{l-1,i}$ la AIB.

Ca fapt divers, puteam face reducerea la interogări unidimensionale pe dos: iterăm prin elemente în ordinea crescătoare a valorilor, astfel încît condiția $a_j < a_i$ să fie automat satisfăcută. Atunci AIB-ul ar fi fost indexat după pozițiile propriu-zise, deci $C_{l-1,j}$ ar fi stat chiar la poziția a_j.

Noua complexitate este $\mathcal{O}(kn \log n)$: pentru fiecare dintre cele $k \times n$ valori ale lui C facem o interogare și o actualizare în AIB. Ca optimizare de memorie, ne este suficientă o singură linie din matricea C.

3.10.5 Problema D-query (SPOJ)

enunț • sursă

La prima vedere AIB-urile ne sînt inutile aici, pentru că funcția „numărul de elemente distincte" nu poate fi calculată prin diferențe de intervale: dacă în intervalul $[1, 10]$ avem 5 elemente distincte, iar în $[1, 20]$ avem 8 elemente distincte, nu știm destule despre numărul de elemente distincte din $[11, 20]$.

Dar, ca la multe alte probleme, varianta offline (în care primim de la început toate interogările) este considerabil mai simplă decît varianta online (în care trebuie să răspundem la o interogare înainte de a o primi pe următoarea).

Pare natural să sortăm interogările și să le scanăm cumva, dar cum? Să fixăm o poziție r și să considerăm toate intervalele care se termină la r. Dacă un interval $[l, r]$ conține o valoare x, atunci o poate conține o dată sau de multiple ori, dar în mod sigur va include **cea mai din dreapta** apariție a lui x înainte de r. Și atunci, pentru o poziție fixată r, dorim să bifăm toate pozițiile $i \in [1, r]$ pentru care nu există o altă poziție $j \in [i + 1, r]$ cu $v[i] = v[j]$.

De exemplu, pentru $r = 8$ și prefixul (1 1 7 6 1 2 6 2), dorim să stocăm bifele (0 0 1 0 1 0 1 1), pentru a indica cele mai din dreapta apariții ale lui 1, 2, 6 și 7. Atunci răspunsul la interogarea $[5, 8]$ va fi tocmai numărul de bife (adică suma) din intervalul $[5, 8]$, pentru că astfel ne asigurăm că numărăm exact o apariție, ultima, a fiecărei valori din interval.

AIB-ul de bife este ușor de actualizat. Dacă, de exemplu, următorul element din vector este 7, trebuie să ștergem bifa de la ultima apariție a lui 7 (poziția 3) și să o aplicăm pe poziția 9. Avem nevoie de un vector cu poziția ultimei apariții a fiecărei valori (dacă există), ceea ce este fezabil deoarece valorile nu depășesc 1.000.000.

La final, reordonăm interogările conform ordinii inițiale și afișăm răspunsurile.

3.10.6 Problema Magic Board (CodeChef)

enunț • sursă

Pare că avem de-a face cu o matrice binară uriașă, dar secretul este să reținem separat informații despre linii și despre coloane. Observația-cheie este că operațiile `Set` modifică doar linii și coloane întregi.

Putem reformula o interogare de tipul `RowQuery i` astfel. Dacă ultima resetare a liniei i a fost la momentul de timp t (operația cu numărul t) și la valoarea 0, atunci cîte coloane au fost modificate din 0 în 1 după momentul t? Dacă au fost k coloane modificate, răspunsul la interogare este $n-k$.

Similar, dacă ultima resetare a liniei i a fost la momentul de timp t (operația cu numărul t) și la valoarea 1, și dacă ulterior k coloane au fost modificate din 1 în 0, atunci răspunsul la interogare este chiar k.

Astfel, trebuie să răspundem la întrebări de tipul: cîte modificări există la valoarea v la timpi $> t$? De aceea, vom ține două AIB-uri pe coloane, indexate după timp (adică după numărul operației), în care stocăm timpul ultimei resetări a fiecărei coloane în 0 și respectiv în 1.

De asemenea, cînd primim o interogare, trebuie să știm timpul ultimei modificări a acelei linii sau coloane, ceea ce putem stoca naiv: un vector de perechi (timp, valoare).

Informațiile pe linii și pe coloane sînt perfect simetrice. Vom studia codul ca să vedem cum putem elimina duplicarea codului. Asta doar dacă evitarea duplicării codului este importantă pentru noi. 😉

3.10.7 Problema Ball (Codeforces)

enunț • sursă

Să abstractizăm problema: date fiind n puncte în spațiu, cîte dintre ele sînt dominate de un alt punct? Spunem că un punct (x, y, z) domină un punct (x', y', z') dacă $x > x'$, $y > y'$ și $z > z'$.

Ca și la problema Subsequences, un prim pas este să reducem interogările tridimensionale la interogări bidimensionale. Să sortăm punctele descrescător după z. Dacă toate z-urile ar fi diferite, atunci am ști că toate punctele procesate anterior au z-ul mai mare decît toate cele viitoare. Dat fiind că z-urile pot fi egale, trebuie să ne adaptăm. Este suficient să procesăm punctele în grupuri cu același z. Pentru toate punctele dintr-un grup calculăm întîi răspunsul și abia apoi le adăugăm la structura de date (oricare ar fi ea).

Astfel, problema pentru punctul $p(p_x, p_y, p_z)$ devine: există vreun punct văzut anterior care să aibă și x-ul și y-ul mai mare? Interogarea pare bidimensională: trebuie să aflăm dacă există vreun punct în cadranul de la (p_x, p_y) la (∞, ∞). Iar coordonatele sînt prea mari pentru un AIB 2D.

Aici intervine ultimul artificiu. Dorim să reducem problema la o interogare unidimensională pe sufix: dintre toate punctele văzute anterior, considerîndu-le doar pe cele cu $x > p_x$, există vreunul cu $y > p_y$? Putem reformula această întrebare ca pe una de maxim: Dă-mi maximul lui y pe domeniul $[p_x, \infty)$. Dacă acest maxim este $> p_y$, atunci există un punct care îl domină pe p, altfel nu.

Putem răspunde la aceste întrebări cu un AIB de maxime, cu două observații:

Trebuie să normalizăm coordonatele x la intervalul $[1, n]$. Nu ne interesează valorile exacte, ci doar relațiile între ele.

AIB-ul de maxime știe să calculeze doar maxime pe prefix, nu pe sufix. Dar putem să reflectăm toate valorile x normalizate față de n pentru a transforma interogările pe sufix în interogări pe prefix.

Observație tangențială: Întotdeauna estimați mărimea fișierului de intrare! În acest caz, avem 1,5 milioane de numere pe 9 cifre plus spații, deci circa 15 MB. Timpul de rulare este efectiv dominat de citire. Sursa inclusă a rulat în 1.100 ms. O a doua sursă, cu citire rapidă, a rulat în 170 ms.

3.10.8 Problema Medwalk, revizitată (Lot 2025)

enunț • sursă

Am discutat această problemă și în capitolul de arbori de intervale. Am găsit o soluție bazată pe un AINT de seturi, construit peste valorile din matrice. Să vedem acum una de 5 ori mai rapidă, probabil cea pe care a dorit-o comisia.

Primele observații se mențin. Decuplăm (conceptual) matricea în doi vectori, unul cu minimele și unul cu maximele de pe fiecare coloană. Pentru a minimiza medianul (scorul unui interval $[l, r]$), căutăm un drum minim lexicografic. Acesta va consta din minimele de pe intervalul $[l, r]$ și din minimul maximelor. Medianul acestei mulțimi va fi una din trei valori posibile (logica deciziei este simplă):

- fie minimul maximelor;
- fie medianul minimelor;
- fie elementul anterior medianului minimelor.

Pentru cazul (1), minimul maximelor îl menținem într-un aint simplu, cu actualizări punctuale și cu interogări de minim pe interval. Pentru cazurile (2) și (3) trebuie să răspundem la interogări de al k-lea element pe interval. Aici introducem următoarea soluție, constînd dintr-un AIB offline 2D construit peste minimele din matrice.

Dorim să căutăm binar al k-lea element. Bunăoară, prima dată ne întrebăm: este el mai mare decît $V_{max}/2$? Dacă da, îl căutăm între $V_{max}/2$ și V_{max}. Dacă nu, îl căutăm între 1 și $V_{max}/2$. În general, pentru plaja de valori curentă, $[x, y]$, vom calcula mijlocul $m = (x+y)/2$ și îi vom adresa structurii de date întrebarea: cîte valori între x și m apar la intrare pe poziții între l și r? Dacă

răspunsul este $\geq k$, atunci al k-lea element are valoarea între x și m. Altfel, el are valoarea între m și y.

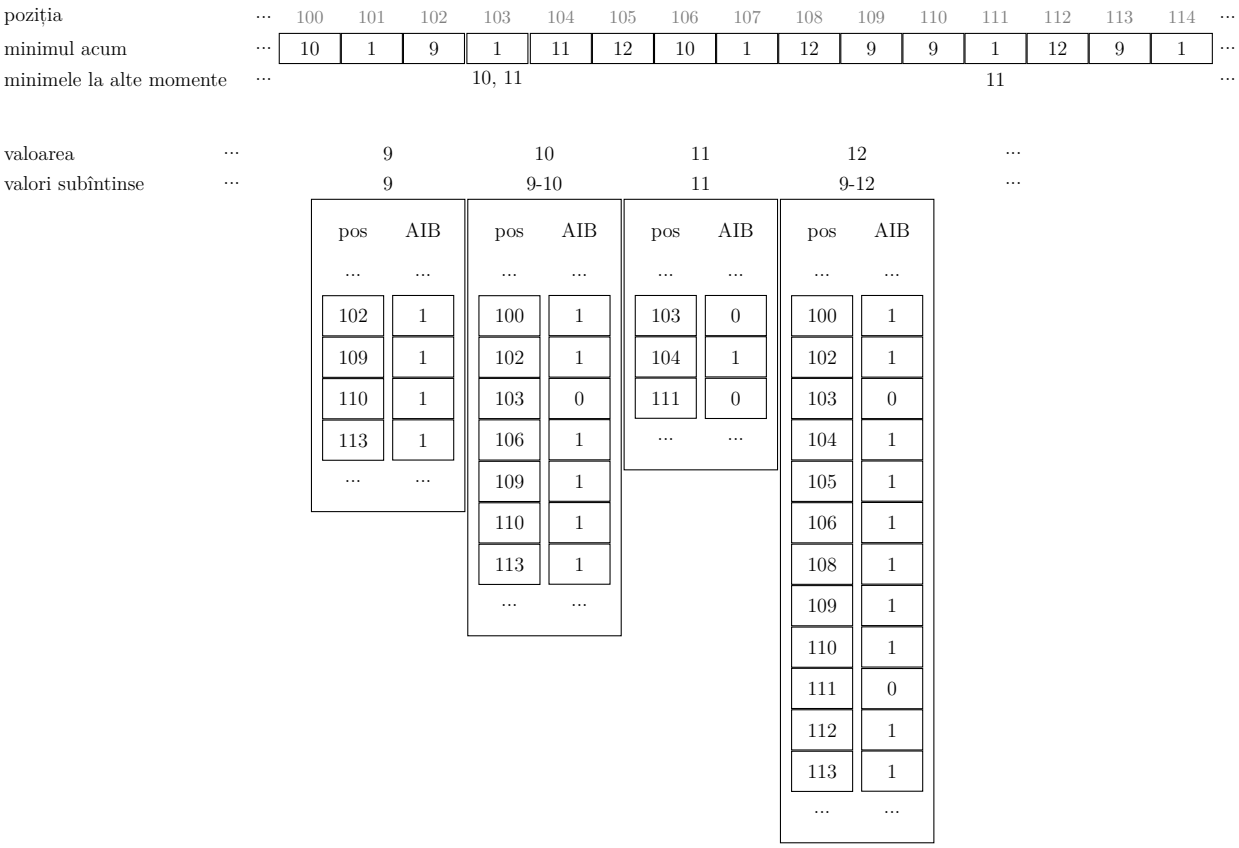
Să încercăm o structură naivă: o matrice binară uriașă A de dimensiuni $n \times V_{max}$, unde $A_{p,v}$ reține 1 dacă valoarea v apare în matrice la poziția p, 0 altfel. Atunci răspunsul la întrebarea „cîte valori între v_1 și v_2 apar pe poziții între l și r?" este suma dreptunghiului $[l, r] \times [v_1, v_2]$. Sună bine, dar memoria nu ne permite această matrice.

De aceea, vom comprima fiecare coloană într-un AIB pornind de la următoarea observație. Coloana v va reține 1, oricînd pe durata întregului program, doar pe liniile p cu proprietatea că poziția p în vectorul de minime va avea, cel puțin după o operație, valoarea v. De aceea, în primă fază simulăm operațiile și colectăm toate minimele posibile pe toate pozițiile, care vor fi cel mult $n + q$. (De aici provine partea de *offline* din nume.) Pe fiecare coloană v stocăm:

- O listă a pozițiilor posibile pentru valori 1, ordonate crescător.
- Un AIB de 0/1 construit peste această listă.

De exemplu, dacă valoarea 9 este măcar o dată minimă pe pozițiile 102, 109, 110, 113, atunci pe coloana 9 stocăm vectorul $[102, 109, 110, 113]$ și un AIB de 0/1 cu 4 poziții disponibile.

Dar facem mai mult decît atît, căci v-am promis un AIB 2D. 🙂 Vectorul de coloane este el însuși un AIB. Coloana v nu reține doar pozițiile aparițiilor lui v, ci ale oricărei valori din $(v - p, v]$, unde p este cea mai mare putere a lui 2 mai mică sau egală cu v. În particular, cele 4 valori 102, 109, 110, 113 exemplificate mai sus nu apar doar în lista coloanei 9, ci și în listele coloanelor $10, 12, 16, 32, 64, \ldots$.



Să observăm că fiecare poziție apare în lista unui număr logaritmic de coloane. De aceea, suma lungimilor tuturor listelor (și a tuturor AIB-urilor) este $\mathcal{O}(n \log V_{max})$.

Pe această structură putem răspunde în timp $\mathcal{O}(n \log n \log V_{max})$ la interogări de al k-lea element.

3.11 Interogări punctuale și actualizări pe interval

Putem privi actualizările pe interval ca pe un fel de „Șmen al lui Mars online". În șmenul lui Mars prelucrăm operațiile „adaugă x pe pozițiile $[l \dots r]$" adăugînd x pe poziția l și $-x$ pe poziția $r+1$. Deosebirea este că în Șmenul lui Mars interogările vin doar la sfîrșit, după toate actualizările, pe cînd în varianta online interogările pot veni și pe parcurs.

AIB-ul poate fi adaptat foarte ușor la această nevoie:

1. Actualizare: adaugă x pe poziția l și $-x$ pe poziția $r+1$.
2. Interogare pe poziția k: calculează suma prefixului $[1 \dots k]$.

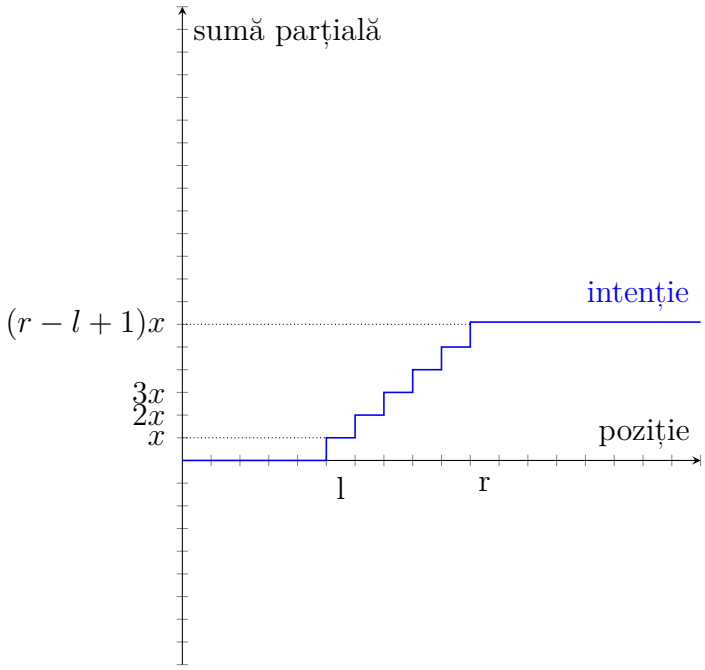
Aceasta funcționează deoarece, la interogarea pe poziția k,

1. Dacă $k < l$, atunci suma prefixului $[1 \dots k]$ nu va include pozițiile l și $r+1$.
2. Dacă $k > r$, atunci suma prefixului $[1 \dots k]$ va include pozițiile l și $r+1$, care se vor anula reciproc. Dorim aceasta, deoarece $k \notin [l, r]$, deci variația intervalului $[l, r]$ nu trebuie să afecteze poziția k.
3. Dacă $l \leq k \leq r$, atunci suma prefixului $[1 \dots k]$ va include doar poziția l, nu și poziția $r+1$, deci poziția k va fi afectată de variația pe intervalul $[l, r]$.

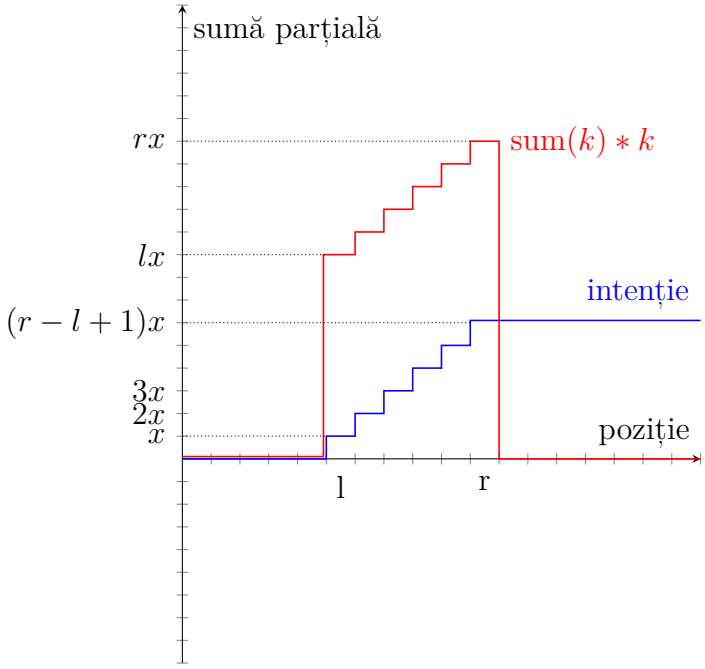
3.12 Interogări și actualizări pe interval

Să rezolvăm și această versiune, doar de amorul artei. Nu cred că discuția de mai jos se aplică la altceva decît la sume (la xor-uri, de pildă).

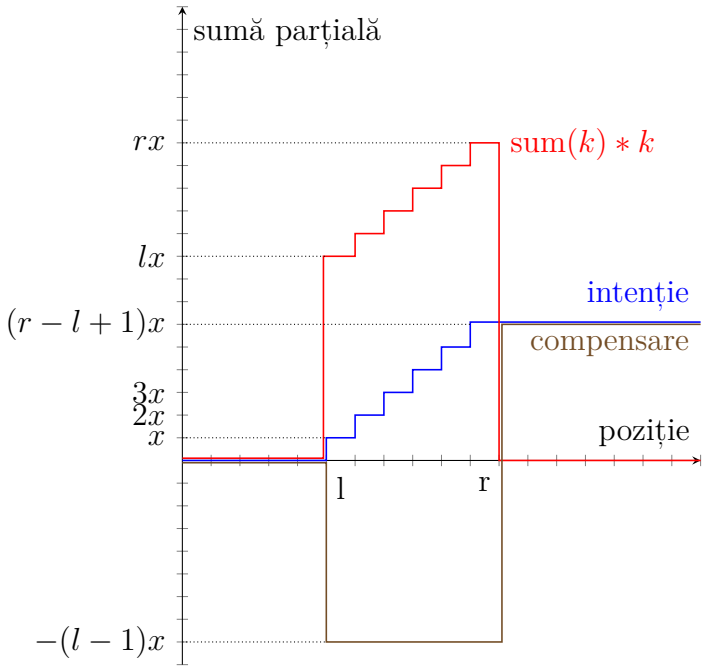
Avînd în vedere că AIB-urile se preocupă în special de sume parțiale, să examinăm grafic sumele parțiale după operația de adăugare a lui x pe intervalul $[l, r]$. Pe pozițiile $[l \dots r]$ ia naștere o funcție scară: dorim ca sumele parțiale să crească cu x, $2x$, ..., $(r-l+1)x$.



Intuitiv, deoarece suma parțială crește cu poziția, sîntem tentați să returnăm, la poziția k, valoarea $\text{sum}(k) \times k$. Pentru ca după r suma parțială să se oprească din crescut, adăugăm x la poziția l și scădem x la poziția $r+1$, similar cu șmenul lui Mars. Doar că atunci funcția $\text{sum}(k) \times k$ are graficul:



Într-adevăr, observăm că suma parțială la stînga lui l este 0, iar la dreapta lui r este $x - x = 0$. Totuși, pare că cele două grafice nu au nicio legătură! Dar nu este chiar așa. Observăm că diferența între cele două funcții arată relativ simplu:



Pentru a corecta al doilea grafic ca să arate ca primul, trebuie să menținem un al doilea AIB în care:

- să scădem pe pozițiile $l, \ldots, r$ valoarea $(l-1)x$;
- să adăugăm pe pozițiile $r+1, \ldots, n$ valoarea $(r-l+1)x$.

În termeni de sume parțiale, trebuie:

- să scădem pe poziția l valoarea $(l-1)x$;
- să adăugăm pe poziția $r+1$ valoarea rx.

Astfel ia naștere codul care, dacă nu-l înțelegem, poate părea foarte ezoteric:

```
struct fenwick_tree_2 {
  fenwick_tree v, w;

  void from_array(long long* src, int n) {
    v.n = n;
    w.from_array(src, n);
  }

  void range_add(int l, int r, long long val) {
    v.add(l, val);
    v.add(r + 1, -val);
    w.add(l, -val * (l - 1));
    w.add(r + 1, val * r);
  }

  long long prefix_sum(int pos) {
    return v.prefix_sum(pos) * pos + w.prefix_sum(pos);
  }
```

```
  long long range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};
```

3.13 Arbori indexați binar 2D

Putem extinde arborii indexați binar la matrice, cu costuri $\mathcal{O}(\log^2 n)$ pentru operații. Să considerăm următoarele operații (*point update, rectangle query*):

- `1 row col val`: Adaugă val la coordonatele (row, col)
- `2 row1 col1 row2 col2`: Calculează suma dreptunghiului (row_1, col_1) - (row_2, col_2) inclusiv.

3.13.1 Structura informației

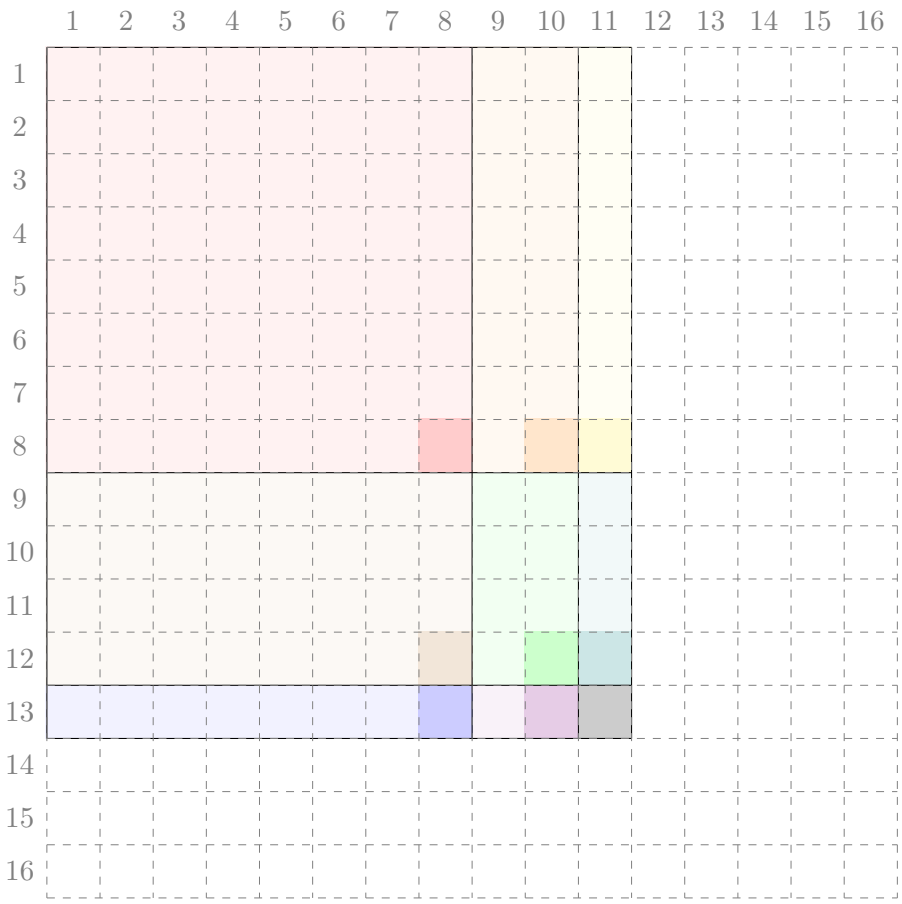
Așa cum arborele 1D modifică structura informației din vector, arborele 2D modifică structura informației din matrice. El stochează anumite sume parțiale. Mai exact, arborele stochează la poziția (r, c) suma elementelor din submatricea (originală) de dimensiuni $p \times q$ cu colțul dreapta-jos la coordonatele (r, c), unde

- p este cea mai mare putere a lui 2 care îl divide pe r
- q este cea mai mare putere a lui 2 care îl divide pe c.

De exemplu, la poziția 40, 60 arborele stochează suma elementelor din matricea de dimensiuni 8×4 cuprinsă între liniile 33 și 40 și coloanele 57 și 60.

3.13.2 Calculul sumei dintr-un dreptunghi

Putem folosi această structură pentru a calcula suma dreptunghiurilor de forma $(1, 1) - (r, c)$. Iată o figură pentru $r = 13$ și $c = 11$.



Dreptunghiul de dimensiune 13×11 se compune din 3×3 dreptunghiuri cu laturile puteri ale lui 2. Trebuie să însumăm valorile din colțurile jos-dreapta ale acestor dreptunghiuri (desenate mai întunecat). Codul este:

```
int prefix_sum(int row, int col) {
  int s = 0;

  for (int r = row; r; r &= r - 1) {
    for (int c = col; c; c &= c - 1) {
      s += mat[r][c];
    }
  }

  return s;
}
```

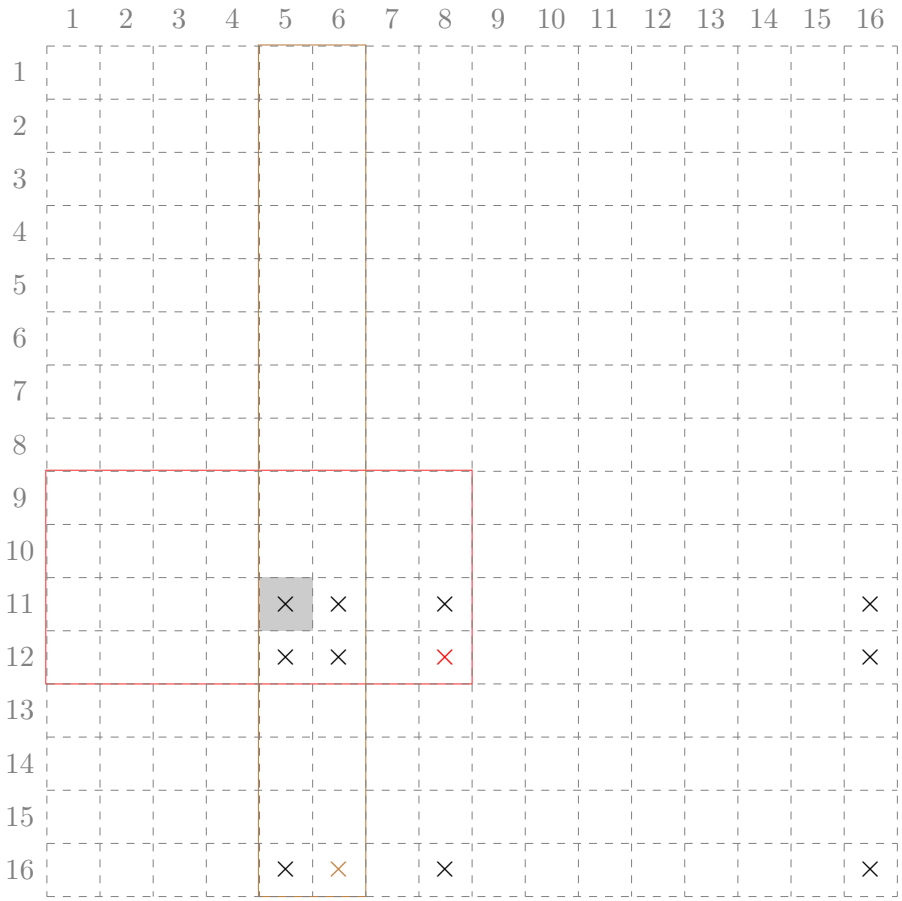
Desigur, putem calcula prin includeri și excluderi suma unui dreptunghi arbitrar:

```
int rectangle_sum(int row1, int col1, int row2, int col2) {
  return
    + prefix_sum(row2, col2)
    - prefix_sum(row2, col1 - 1)
    - prefix_sum(row1 - 1, col2)
    + prefix_sum(row1 - 1, col1 - 1);
```

```
}
```

3.13.3 Actualizări punctuale

Ca și la varianta 1D, cînd modificăm valoarea de la coordonatele (r, c) trebuie să modificăm corespunzător toate dreptunghiurile care includ acele coordonate. Iată un exemplu pentru linia 11, coloana 5:



Se observă că trebuie actualizate liniile 11, 12 și 16, adică exact cele care ar include poziția 11 într-un AIB unidimensional. Similar, trebuie actualizate coloanele 5, 6, 8 și 16, adică exact cele care ar include poziția 5. Am evidențiat cu roșu și cu auriu două dintre aceste coordonate, $(12, 8)$ și $(16, 6)$, și dreptunghiurile aferente lor, pentru a evidenția că ele includ celula (11, 5).

Codul pentru actualizare este:

```
void add(int row, int col, int val) {
  for (int r = row; r <= n; r += r & -r) {
    for (int c = col; c <= n; c += c & -c) {
      mat[r][c] += val;
    }
  }
}
```

3.13.4 Construcția în $\mathcal{O}(n^2)$

Și acest arbore poate fi construit in-place, refolosind matricea dată la intrare și în timp $\mathcal{O}(1)$ per element.

În varianta unidimensională, trebuia să propagăm valoarea de la poziția x la poziția $x + (x \,\&\, -x)$. În varianta bidimensională am vrea să procedăm astfel:

- Fie $r' = r + (r \,\&\, -r)$ și $c' = c + (c \,\&\, -c)$.
- Propagăm valoarea de la poziția (r, c) la poziția (r, c'). De acolo, ea se va propaga automat la coloanele următoare.
- Propagăm valoarea de la poziția (r, c) la poziția (r', c). De acolo, ea se va propaga automat la liniile următoare, iar pe fiecare linie pe coloanele următoare.

Dar apare o problemă: valoarea de la (r, c) se va propaga la (r', c') de două ori, pe căi diferite! Din fericire, este suficient să o scădem o dată. Iată codul:

```
void build() {
  for (int r = 1; r <= n; r++) {
    for (int c = 1; c <= n; c++) {
      int next_r = r + (r & -r);
      int next_c = c + (c & -c);

      if (next_r <= n) {
        mat[next_r][c] += mat[r][c];
      }

      if (next_c <= n) {
        mat[r][next_c] += mat[r][c];
      }

      if ((next_r <= n) && (next_c <= n)) {
        mat[next_r][next_c] -= mat[r][c];
      }
    }
  }
}
```

Capitolul 4

Descompunere în radical

Pe lîngă faptul că oferă complexitate optimă pentru unele probleme, metoda oferă și un substitut bun pentru algoritmi în $\mathcal{O}(n \log n)$ și în special $\mathcal{O}(n \log^2 n)$, în cazul în care nu găsim ideea. Descompunerea în radical este, în experiența mea, mult mai ușor de implementat.

4.1 Actualizări punctuale

Revenim la problema inițială: actualizări punctuale, sume pe interval. Împărțim vectorul în blocuri de lungime $k = \sqrt{n}$. Așadar, vor exista $\lceil n/k \rceil$ blocuri (engl. *blocks* sau *buckets*). Pentru fiecare bloc, menținem o informație suplimentară: suma elementelor din acel bloc.

Memorie suplimentară: $\mathcal{O}(\sqrt{n})$.

```
int v[MAX_N];
int b[MAX_BUCKETS];
int bs, nb;

// Se poate implementa și cu împărțiri pentru concizie (sînt doar n).
void init_buckets() {
  nb = sqrt(n + 1);
  bs = n / nb + 1;

  for (int i = 0; i < nb; i++) {
    int bucket_start = i * bs;
    for (int j = 0; j < bs; j++) {
      b[i] += v[j + bucket_start];
    }
  }
}

int array_sum(int* v, int l, int r) {
  int sum = 0;
  while (l < r) {
    sum += v[l++];
```

```
  }
  return sum;
}

int fragment_sum(int l, int r) {
  return array_sum(v, l, r);
}

int bucket_sum(int l, int r) {
  return array_sum(b, l, r);
}

int range_sum(int l, int r) { // [l, r)
  int bl = l / bs, br = r / bs;
  if (bl == br) {
    return fragment_sum(l, r);
  } else {
    return
      // capete
      fragment_sum(l, (bl + 1) * bs) +
      fragment_sum(br * bs, r) +
      // blocuri complete
      bucket_sum(bl + 1, br);
  }
}

void point_add(int pos, int val) {
  v[pos] += val;
  b[pos / bs] += val;
}
```

Vă recomand să lucrați pe intervale închise la stînga, deschise la dreapta, ca să simplificați aritmetica.

Încheiem secțiunea cu observația că funcția `range_sum` are complexitatea $\mathcal{O}(k + n/k)$: Ea iterează naiv prin blocurile acoperite parțial, așadar $\mathcal{O}(k)$, și iterează rapid prin blocurile acoperite complet, așadar $\mathcal{O}(n/k)$. Alegem $k = \sqrt{n}$ ca să minimizăm acea sumă, dar vom vedea în unele exemple că pot lua naștere și alte sume care duc la valori diferite pentru k.

4.2 Actualizări pe interval

Folosim o informație suplimentară care amintește de informația propagată *lazy* din arborii de segmente. Pentru problema dată (sume pe interval), am denumit această informație `bdelta`. Ea are semnificația: `bdelta[j]` este o valoare care trebuie adăugată la fiecare element din blocul `j`. Așadar, valoarea reală a unui element `i` din blocul `j` este `v[i] + bdelta[j]`.

```
int fragment_sum(int l, int r, int bucket) {
```

```
  return
    (r - l) * bdelta[bucket] +
    array_sum(v, l, r);
}

int bucket_sum(int l, int r) {
  return
    array_sum(bsum, l, r) +
    bs * array_sum(bdelta, l, r);
}

int range_sum(int l, int r) {
  int bl = l / bs, br = r / bs;
  if (bl == br) {
    return fragment_sum(l, r, bl);
  } else {
    return
      // capete
      fragment_sum(l, (bl + 1) * bs, bl) +
      fragment_sum(br * bs, r, br) +
      // blocuri complete
      bucket_sum(bl + 1, br);
  }
}

void array_add(long long* v, int l, int r, int val) {
  while (l < r) {
    v[l++] += val;
  }
}

void fragment_add(int l, int r, int bucket, int val) {
  bsum[bucket] += (r - l) * val;
  array_add(v, l, r, val);
}

void range_add(int l, int r, int val) {
  int bl = l / bs, br = r / bs;
  if (bl == br) {
    fragment_add(l, r, bl, val);
  } else {
    // capete
    fragment_add(l, (bl + 1) * bs, bl, val);
    fragment_add(br * bs, r, br, val);

    // blocuri complete
    array_add(bdelta, bl + 1, br, val);
  }
}
```

Paranteză: conceptual, aint-urile fac același lucru cu descompunerea în radical, deci multe concepte se translatează între cele două, cum ar fi propagarea *lazy*. Marea inovație a arborilor de intervale este că garantează descompunerea în $\mathcal{O}(\log n)$ blocuri.

4.3 Optimizări

4.3.1 Evitați împărțirile!

Codul anterior face doar două împărțiri per operație. În general, aveți nevoie strict de o împărțire pentru fiecare indice primit ca parametru. Evitați stilul de mai jos, care face $\mathcal{O}(\sqrt{n})$ împărțiri (sau operații modulo) per operație. El poate fi **de cîteva ori mai lent** (vedeți *benchmark*-urile).

```
int bl = l / bs, br = r / bs;
int sum = 0;

// stînga
do {
  sum += v[l++];
} while (l % bs);

// dreapta
while (r % bs) {
  sum += v[--r];
}

// blocuri complete
for (int i = bl + 1; i < br; i++) {
  sum += b[i];
}

return sum;
```

4.3.2 Alegerea mărimii blocurilor

Am auzit că mărimea blocului merită declarată constantă, deoarece compilatorul va optimiza împărțirile. Am experimentat, dar diferențele nu sînt semnificative. Am încercat chiar să declar mărimea o putere a lui 2, pentru ca împărțirile să fie ieftine. Codul rezultat a fost mai lent! Cred că motivul este că se dezechilibrează acea funcție $k + n/k$ pe care descompunerea în radical o optimizează.

Diferența de viteză este vizibilă cînd codul face multe împărțiri. Cînd codul face $\mathcal{O}(1)$ împărțiri per operație, nu mai contează.

Are sens să declarați mărimea constantă dacă vi se pare codul mai simplu (evitați cîteva calcule la inițializare). În acest caz, vă recomand să puneți constanta mică (3-4) cînd lucrați local și să-i

dați valoarea reală cînd trimiteți sursa. Altfel, dacă testați local pe teste mici și mărimea blocului 300, nu veți testa decît codul cu interogări și actualizări într-un singur bloc.

4.3.3 Alegerea mărimii blocurilor pentru operații inegale

Dacă programul vostru depășește timpul, merită să variați mărimea blocurilor în jurul lui $\sqrt{n}$ ca să vedeți dacă se schimbă ceva. Fie b numărul de blocuri și fie l lungimea unui bloc. Operațiile pot depinde doar de una dintre aceste variabile sau pot depinde de ambele, dar în mod inegal. Exemplu: dacă constanta pentru cele două capete de segment (de lungime medie $l/2$) este mult mai mare decît constanta pentru blocurile întregi, merită redus l-ul puțin.

Iată un exemplu mai interesant. Să spunem că nu știm să rezolvăm corect una dintre operații și găsim doar o implementare în $\mathcal{O}(b+l^2)$. Pare catastrofic: dacă alegem $b = l = \sqrt{n}$, avem de fapt o soluție în $\mathcal{O}(n)$. 🤔

Dar se poate asimptotic mai bine. Să alegem $b = n^{2/3}$, caz în care $l = n/b = n^{1/3}$. Dacă $n = 100.000$, atunci $b = 2.174$ și $l = 46$. O soluție în $\mathcal{O}(\sqrt{n})$ per operație ar face circa 600 de operații, pe cînd a noastră va face circa 4.300 de operații. Desigur, diferența este notabilă, dar, cu o implementare eficientă, avem șanse să „ținem aproape".

Pauză de matematică: mai exact, noi vrem să găsim minimul funcției

$$f(x) = x^2 + \frac{n}{x}$$

Funcția își atinge minimul cînd derivata este zero, iar derivata este

$$f'(x) = 2x - \frac{n}{x^2} = \frac{2x^3 - n}{x^2}$$

Rezultă că l minim este $\sqrt[3]{n/2} \approx 37$, iar funcția va face sub 4.100 de operații.

4.4 Probleme

4.4.1 Problema Mexitate (ONI 2018 clasa a 9-a)

enunț • surse

Această problemă nu mi se pare nici în ruptul capului de clasa a 9-a. Posibil de baraj juniori.

Am notat cu m numărul de linii deoarece m vine înaintea lui n în alfabet, după cum și k vine înaintea lui l în alfabet.

Să luăm în calcul soluția naivă: calculăm mex-ul matricei din colțul stînga-sus, apoi translatăm în diverse feluri matricea pentru a recalcula incremental mex-urile celorlalte ferestre. De exemplu, putem urma un traseu șerpuit: translatăm matricea la dreapta pînă la capăt, apoi o dată în jos, apoi în stînga pînă la capăt etc.

La fiecare translație, menținem într-o structură S informații despre frecvența elementelor din matrice. Ștergem din structură elementele care ies din fereastră și le inserăm pe cele care intră în fereastră. Atunci complexitatea translatărilor va fi $\mathcal{O}(mnk + ml)$. Ca să minimizăm efortul, rotim sau transpunem matricea cînd $k > l$. Astfel, $k \leq l$ și complexitatea translatărilor va fi $\mathcal{O}(mn\sqrt{mn})$.

Odată ce am adus raționamentul pînă aici, eu am și trimis o sursă, cu structura S implementată naiv ca vector de frecvențe. Astfel m-am asigurat că restul programului funcționează, căci el are suficient de multă logică (matrice de dimensiuni arbitrare, transpoziții, șerpuire...). Deoarece am gîndit totul modular, am programat structura `frequency_tracker` să expună funcțiile `add`, `remove` și `mex`. Pentru versiunea optimizată, doar am rescris acele funcții.

Pentru punctaj maxim, ce ne dorim de la structura S? Dorim să facem $\mathcal{O}(mn\sqrt{mn})$ inserări și ștergeri și $\mathcal{O}(mn)$ calcule de mex. Rezultă că ne permitem $\mathcal{O}(\sqrt{mn})$ per apel de mex, dar avem nevoie de $\mathcal{O}(1)$ pe inserare și ștergere.

Atunci putem folosi descompunerea în radical. Stocăm vectorul naiv de frecvențe. În plus, în fiecare bloc stocăm numărul de valori nenule. Inițial această valoare este 0. Ea crește cu 1 ori de cîte ori frecvența unui element din bloc crește de la 0 la 1 și, invers, scade cu 1 ori de cîte ori frecvența unui element din bloc scade de la 1 la 0. Funcția mex caută din bloc în bloc pînă cînd găsește un bloc cu cel puțin o valoare nulă, apoi caută naiv prin acel bloc.

4.4.2 Problema Give Away (SPOJ)

enunț • surse

Problema are limită de timp mare (1-2 secunde, iar conform paginii *status*, chiar peste 6 secunde). Acesta este un indiciu bun că o soluție în $O((n+q)\sqrt{n})$ este arhisuficientă. Doar că pare mai rău de atît! Să presupunem că ținem pe fiecare bloc o structură S (nu știm încă ce). Atunci:

- Sigur putem procesa actualizările în $O(\sqrt{n})$, chiar și naiv la o adică.
- La căutare, pare simplu să procesăm naiv blocurile acoperite parțial, în $O(\sqrt{n})$.
- Dar ce facem cu blocurile acoperite complet? Dacă folosim orice structură echilibrată, introducem un factor logaritmic în plus pentru căutarea lui c.

Rezultă o complexitate de $\mathcal{O}(q\sqrt{n}\log n)$. Dar în 6 secunde, este acceptabil.

Mărimea blocurilor

Dacă am vrea să optimizăm suma $k + n/k$, am alege $k = \sqrt{n} \approx 707$. Dar noi dorim să optimizăm suma $k + n/k \log k$. Ochiometric log-ul va fi undeva între 7 și 12, deci este important să-l mărim pe k. Atunci $\log k$ va crește lent, dar n/k va scădea rapid. Din cîteva încercări pe hîrtie aflăm că valoarea optimă pentru k este 2.000 sau 3.000. Într-adevăr, timpii de execuție pe acolo ating optimul.

Detalii de implementare

Așadar, dorim o structură S care să admită inserări, ștergeri, și numărarea elementelor mai mari sau egale cu o valoare dată. Eu am izolat această structură într-un `struct` și am scris o primă implementare naivă (S este doar un vector). Cu doar 5 linii de cod în plus, sursa naivă mă ajută să verific corectitudinea restului programului.

A doua încercare a fost cu structuri de date din STL, care trece în 3 secunde. Dar este nevoie să memorați papagalicește două noțiuni (le vom relua în capitolele viitoare):

1. `set`-ul simplu nu este suficient, căci el poate să caute valoarea c, dar nu și să numere elementele mai mari sau egale cu c. Este nevoie de structuri cu statistici de ordine (PBDS).
2. Blocurile pot conține valori egale, deci ne trebuie un multiset. Multisetul PBDS este o încropeală, iar ștergerea trebuie rescrisă.

Dar a treia încercare este elementară și trece în 1.3 secunde: pe fiecare bloc ținem vectorul original (ca să știm ce element înlocuim) și o copie sortată. Putem căuta binar în acea copie, iar la inserare/ștergere o actualizăm prin deplasări naive.

4.4.3 Problema Holes (Codeforces)

enunț • sursă

Îmi place această problemă pentru că este nestandard. Nu facem efectiv împărțirea în blocuri și nu stocăm informații agregate pe fiecare bloc. În loc de aceasta, fiecare poziție *pos* reține informații ca să își accelereze trecerea prin blocul său:

- care este ultima destinație din același bloc pe care o vizitează o bilă pornind de la *pos*;
- cîte salturi face bila pînă la acea destinație.

Atunci operațiile sînt:

- La interogare, urmărim traseul bilei din bloc în bloc, în timp $\mathcal{O}(\sqrt{n})$.
- La actualizare, trebuie să recalculăm poziția modificată și toate pozițiile din stînga ei, din același bloc, tot în $\mathcal{O}(\sqrt{n})$.

Sursa mea se apropie de limita de timp (700 ms din 1000 ms). Am încercat următoarea optimizare care cred că ajută. Am ales o mărime mai mare pentru blocuri, 1.000 în loc de cea teoretică ($\sqrt{100.000} \approx 316$). Astfel accelerăm interogările, care traversează mai puține blocuri, în defavoarea actualizărilor, care au de recalculat mai multe poziții. Dar actualizările operează foarte local, pe 1.000 de poziții vecine și vor beneficia de cache. În schimb, interogările sar de colo-colo prin vector.

Remarc și că cele mai rapide soluții ajung la 122 ms. Ele folosesc *link-cut trees* pentru a obține $\mathcal{O}(n \log n)$. Putem percepe structura ca fiind arborescentă: părintele unei poziții este poziția unde sare bila. Atunci actualizările schimbă structura arborelui, ancorînd poziția modificată de un alt părinte. De aici (cred) decurge soluția cu *link-cut trees*.

4.4.4 Problema Piezișă (Baraj ONI 2022)

enunț • surse

Un element comun tuturor soluțiilor este: renumerotăm vectorul de la 1. Acum, dacă xorul pe $[l, r]$ este 0, atunci xorurile pe $[0, l-1]$ și $[0, r]$ sînt egale. Așadar, calculăm xorurile parțiale și le normalizăm, ca să fie indexabile. Acum răspunsul la orice interogare $[l, r]$ este o pereche (x, y) cu $0 \leq x < l$, $r \leq y \leq n$ și $v[x] = v[y]$.

Brute force de 100p

Menționez pentru completitudine că următorul *brute force* optimizat ia 100p: Răspundem la interogări online, imediat ce le primim. Fie o interogare $[l, r]$. Căutăm binar ultima apariție a lui $v[r]$ pe o poziție anterioară lui l (pentru aceasta, colectăm în prealabil listele de poziții pentru fiecare valoare distinctă din v). Fie această poziție q. Atunci avem o soluție de mărime r–q, posibil ∞ dacă elementul $v[r]$ nu apare înainte de poziția l. Repetăm aceeași întrebare la pozițiile $r+1, r+2, \ldots$. Menținem minimul răspunsurilor la aceste căutări, fie el m. Ne oprim la poziția $l+m$ (sau, desigur, la n), deoarece dincolo de ea am putea primi doar răspunsuri mai mari decît optimul curent. Afișăm răspunsul m.

Metoda 1

Fără sortarea interogărilor nu am găsit nicio soluție. Așadar, să sortăm interogările după capătul stîng și să răspundem la ele baleind l de la 1 la n. Acum, pentru o interogare $[l, r]$, dorim ca pentru fiecare poziție $r' \geq r$ să aflăm dacă valoarea $v[r']$ există pe vreo poziție $l' \leq l$. Aceasta este mult prea scump, deci dorim cumva să răspundem la întrebări pentru mai multe poziții simultan.

Să descompunem vectorul în bucăți de mărime $\sqrt{n}$. Pentru o interogare $[l, r]$, fie blocurile celor două capete bl și br. Atunci răspunsul poate avea capătul stîng fie în blocul bl (în stînga lui l), fie într-un bloc anterior. Similar pentru capătul drept. Cazul care ne încurcă este cel în care ambele capete sînt în blocurile bl, respectiv br. Vrem să evităm să comparăm naiv aceste $\mathcal{O}(\sqrt{n} \times \sqrt{n}) = \mathcal{O}(n)$ posibilități.

Astfel ne vine ideea să calculăm o singură informație pentru toate pozițiile din stînga lui l. Fie $left[x]$ ultima poziție (înaintea lui l) pe care apare valoarea x. Evident, putem menține vectorul $left$ în $\mathcal{O}(1)$ pe măsură ce l avansează.

Capătul drept al interogării poate varia oricum, și aici intervine descompunerea în radical. Fie $best[b]$ cea mai mică soluție cunoscută pînă în prezent între orice valoare din blocul b și orice valoare din stînga lui l. Putem menține vectorul $best$ în $\mathcal{O}(1)$ per bloc pe măsură ce l avansează (așadar efort $\mathcal{O}(n\sqrt{n})$ efort global). Pentru aceasta, trebuie să cunoaștem, pentru elementul în curs de procesare $v[l]$, cea mai din stînga apariție a sa în fiecare bloc următor. Putem precalcula această informație la început, într-o manieră similară cu colectarea listelor de apariții a fiecărei valori. Vezi vectorul `ptr` și funcția `preprocess_values`.

Cu aceste informații, răspunsul pentru interogarea curentă $[l, r]$ este minimul dintre:

- Valorile *best* ale blocurilor mai mari decît br.
- Diferențele $r' - left[v[r']]$ pentru elementele din blocul br începînd cu poziția r.

Codul este lung, cam urît și cam lent, dar ia 100p.

Metoda 2

Citind surse mai rapide decît a mea, am găsit-o pe aceasta, care folosește o metodă mai simplă. Principala diferență este că sortează interogările diferit: crescător după blocul lui l, iar la egalitate descrescător după r. Această sortare amintește de algoritmul lui Mo, pe care îl vom discuta în curînd.

Vectorul $left$ are aceeași definiție ca mai înainte, dar include doar elementele dinaintea blocului bl.

La procesarea unui bloc, avem avantajul că toate r-urile scad. De aceea, și la dreapta putem ține un vector $right$ similar cu $left$: $right[x]$ indică prima apariție a lui x pe o poziție mai mare decît r-ul curent. Atenție, vectorul $right$ trebuie golit și recalculat pentru fiecare bloc bl. Acest efort este $\mathcal{O}(n\sqrt{n})$, deci acceptabil.

Pe măsură ce r scade, menținem și valoarea minimă m a oricărei perechi $right[x] - left[x]$. Deoarece r doar scade, intervalele doar scad, deci m poate doar să scadă.

Acum, pentru $[l, r]$, răspunsul poate fi:

- chiar m, dacă soluția are capătul stîng înaintea blocului bl;
- altfel, minimul dintre $right[x] - x$ pentru toate valorile x din blocul bl, din stînga lui l.

Implementarea este conceptual mai simplă și de peste două ori mai rapidă!

4.5 Descompunere după operații

Iată acum o tehnică înrudită. Proiectăm o structură relativ naivă pentru procesarea operațiilor. Prin „naiv” înțelegem, de exemplu, inserarea într-un vector prin deplasarea elementelor, fără a folosi structuri de date echilibrate.

Facem aceste operații naive cîtă vreme ele se încadrează în $\mathcal{O}(\sqrt{n})$ sau $\mathcal{O}(\sqrt{q})$ per operație.

Periodic, trebuie să intervenim pentru ca structura naivă să nu degenereze în timp și să nu ajungă la $\mathcal{O}(n)$ sau $\mathcal{O}(q)$. De aceea, aproximativ o dată la $\sqrt{q}$ operații iterăm prin structură și o „consolidăm” (ce înseamnă asta depinde de la problemă la problemă). Această consolidare poate dura $\mathcal{O}(n)$, pentru ca efortul total al consolidărilor să fie $\mathcal{O}(n\sqrt{q})$. Această limită de timp face consolidările să fie facile, în general.

Să studiem o problemă concretă.

4.6 Probleme

4.6.1 Problema Serega and Fun (Codeforces)

enunț • surse

Trebuie să procesăm eficient operațiile:

1. Rotește circular la dreapta, cu o poziție, un interval $[l, r]$.
2. Raportează frecvența unei valori k într-un interval $[l, r]$.

Iată întîi soluția „clasică", cu descompunere în blocuri de mărime egală. Soluția relativ directă. În fiecare bloc putem menține:

1. O listă a elementelor.
2. Informații despre frecvență (`map` sau vector simplu).

Atunci, la rotire, trebuie să:

1. Rotim naiv blocurile acoperite parțial.
2. Rotim eficient blocurile acoperite complet. La rotire, adăugăm la începutul listei elementul care ne parvine de la blocul anterior și trimitem ultimul element din listă în blocul următor.

La interogare, trebuie să:

1. Consultăm element cu element blocurile acoperite parțial.
2. Consultăm vectorii de frecvență ai blocurilor acoperite complet.

Discuție despre implementarea cu structuri STL

Putem rezolva problema și cu structuri ca `deque`, `map` sau `unordered_map`, dar este nevoie de o implementare atentă ca să nu depășim timpul și memoria.

În particular, mare atenție la operatorul `[]`! Este tentant să îl folosim ca să aflăm frecvența unui element. Dacă elementul nu există în `map`, operatorul va returna 0, ceea ce este corect. Dar **operatorul inserează elementul** dacă nu exista deja.

Ce înseamnă asta? În mod normal, suma mărimilor tabelelor hash din toate blocurile va fi n. Dar, dacă pentru fiecare din cele q interogări noi căutăm o valoare inexistentă în fiecare dintre cele $\sqrt{n}$ blocuri, și dacă toate acele valori sînt inserate, suma mărimilor tabelelor va ajunge la $q\sqrt{n}$. Necesarul de memorie va crește enorm. Este obligatoriu să folosim iteratori pentru căutarea sau decrementarea frecvențelor, pentru ca mărimea tabelelor să rămînă $\mathcal{O}(\sqrt{n})$.

Detalii de implementare

Prime sursă stochează lista de elemente din fiecare bloc într-un `deque` din STL. A doua sursă folosește un simplu buffer circular: un vector de mărime `BUCKET_SIZE` împreună cu un pointer la primul element din listă. Atunci putem face rotirea completă în $\mathcal{O}(1)$ mutînd pointerul de start

cu un element spre stînga. Pe poziția noului element de start scriem valoarea provenită din blocul anterior.

O altă diferență este că prima sursă stochează frecvențele din fiecare bloc într-o tabelă hash, pe cînd a doua folosește un vector de frecvențe, căci elementele au valori de cel mult n. Necesarul de memorie crește la $\mathcal{O}(n \times numBuckets)$. Din acest motiv, merită să experimentăm cu mai puține blocuri de dimensiune mai mare. Într-adevăr, pentru blocuri de mărime $2\sqrt{n} \approx 640$, timpul scade la jumătate față de $\sqrt{n}$.

Mai remarcăm că frecvența într-un singur bloc încape pe tipul `short`, nu este nevoie de `int`. Doar cu această modificare am redus timpul de rulare cu 20%. Am experimentat și cu `unsigned char`, dar atunci frecvența maximă stocabilă ar fi 256, deci mărimea maximă a unui bloc ar trebui să fie 256, iar timpul se înrăutățește.

Soluție cu descompunere după operații

Iată acum și rezolvarea în care lăsăm blocurile să fluctueze ca lungime. La rotire, extragem efectiv elementul de pe poziția r din blocul său și îl inserăm la indicele corect în blocul poziției l. Blocurile dintre l și r rămîn nemodificate. Ca fapt divers, complexitatea rotirii depinde doar de mărimea blocului, nu și de numărul de blocuri.

Cu timpul, lungimile blocurilor pot degenera, ceea ce poate încetini operațiile: dacă un bloc devine foarte mare, atunci inserarea, ștergerea și numărarea de elemente din acel bloc vor fi lente. De aceea, periodic refacem structura blocurilor:

1. Colectăm toate blocurile într-un vector. Acesta este chiar conținutul real al vectorului.
2. Redistribuim elementele în blocuri de mărime egală.

Putem face redistribuirea fie periodic (la fiecare $\mathcal{O}(\sqrt{q})$ operații), fie la nevoie, cînd în momentul inserării detectăm că unul dintre blocuri a atins un anumit prag, să zicem dublu față de lungimea inițială. În ambele cazuri, complexitatea rămîne $\mathcal{O}(q\sqrt{n})$.

Această implementare este relativ elementară și se mișcă de circa două ori mai repede decît precedenta!

4.7 Procesări diferite înainte și după $\sqrt{n}$

Următoarele două secțiuni teoretice prezintă două tehnici care ating timp $\mathcal{O}(n\sqrt{n})$ din motive matematice. Tehnicile nu duc la descompunere în radical „convențională”, cu blocuri, dar nu știu unde altundeva să le încadrez.

Prima tehnică pornește de la observațiile:

1. Între 1 și $\sqrt{n}$ există $\sqrt{n}$ valori[1].
2. Valorile de forma $\lfloor n/k \rfloor$, cu $k > \sqrt{n}$, iau doar $\mathcal{O}(\sqrt{n})$ valori distincte.

[1] From the Department of Redundancy Department.

4.8 Probleme

4.8.1 Problema Time to Raid Cowavans (Codeforces)

enunț • sursă

În foarte multe cuvinte, problema ne dă un vector cu n elemente și q interogări $\langle prim, pas \rangle$. Răspunsul fiecărei interogări este suma valorilor de pe pozițiile care formează progresia cu primul termen $prim$ și pasul pas.

Desigur, codul naiv care însumează progresiile este lent:

```
long long naive_prog_sum(int first, int step) {
  long long sum = 0;
  for (int i = first; i <= n; i += step) {
    sum += w[i];
  }
  return sum;
}
```

Dar... nu chiar atît de lent! Dacă pasul este cel puțin $\sqrt{n}$, progresia va avea cel mult $\sqrt{n}$ termeni. Rămîne să tratăm progresiile cu pasul mic (așadar $1, 2, \ldots, \sqrt{n}$). Ne permitem să facem o preprocesare în $\mathcal{O}(n)$ pentru fiecare din acești pași. Pentru un pas pas, preprocesăm efectiv $prep[i]$ = răspunsul la progresia cu primul termen i și pasul pas. Apoi putem răspunde la interogările pentru acel pas în $\mathcal{O}(1)$.

```
void preprocess(int step) {
  for (int i = n; (i > n - step) && (i >= 1); i--) {
    prep[i] = w[i];
  }
  for (int i = n - step; i >= 1; i--) {
    prep[i] = w[i] + prep[i + step];
  }
}
```

Complexitatea totală este:

- $\mathcal{O}(q\sqrt{n})$ pentru progresiile cu pas mare (calculate naiv);
- $\mathcal{O}(n\sqrt{n} + q)$ pentru preprocesarea tuturor răspunsurilor pentru pași mici.

Alte probleme similare:

- Train Maintenance (Codeforces);
- Căsuța (Lot juniori 2025) – atenție, la momentul scrierii acestei secțiuni problema nu are checker.

4.9 Descompunere în valori distincte

Această tehnică pornește de la observația: dacă suma unor numere naturale este n, atunci numerele au $\mathcal{O}(\sqrt{n})$ valori distincte. Demonstrația decurge din inversa sumei Gauss.

4.10 Probleme

4.10.1 Problema Sandor (Baraj ONI 2025)

enunț • sursă

Observăm că algoritmul lui Sandor este un algoritm *greedy* pentru problema rucsacului. Să facem trei experimente de gîndire despre natura acestui algoritm.

În primul rînd, dacă eliminăm un obiect pe care algoritmul oricum nu l-ar selecta (pentru că nu încape), atunci vom obține aceeași sumă ca și cînd nu am elimina nimic. Practic, obiectul nu există pentru algoritm.

Mai interesant, putem generaliza prima observație. Dacă există k obiecte de aceeași greutate și, la acel pas în algoritm, în rucsac încap mai puțin de k din aceste obiecte, atunci pe oricare dintre ele încercăm să-l eliminăm vom obține aceeași greutate ca și cînd nu am elimina nimic. De ce? Dacă, de exemplu, există 5 obiecte identice și doar 3 încap în rucsac, atunci dacă îl eliminăm pe primul algoritmul îl va adăuga pe al 4-lea.

În sfîrșit, deoarece în rucsac punem obiecte cu greutatea totală cel mult g, rezultă că vom pune $\mathcal{O}(\sqrt{g})$ greutăți distincte.

De aceea, merită să stocăm mai degrabă un vector de serii (în sursă le-am numit *runs*) de elemente egale, $\langle val, cnt \rangle$, decît valorile originale. Peste acest vector precalculăm niște *jump pointers*, adică răspunsurile la întrebări de tipul „Cu ce serie să continui algoritmul dacă rucsacul mai are capacitate rămasă c?” Cu această reprezentare, evaluarea algoritmului lui Sandor este elementară și necesită timp $\mathcal{O}(\sqrt{g})$. Mai mult, putem parametriza algoritmul ca să-l putem rula începînd cu oricare dintre serii, nu neapărat cu prima.

Cerința 1

De aici rezultă un algoritm relativ naiv pentru cerința 1. El necesită $\mathcal{O}(\sqrt{g}^2)$, adică $\mathcal{O}(g)$.

Mergînd de la stînga la dreapta prin vector, considerăm fiecare serie $\langle val, cnt \rangle$. Dacă în prezent în rucsac încap mai puțin de cnt obiecte, atunci conform observației din preambul oricum am elimina un element din serie obținem tot soluția inițială (avem cnt astfel de moduri).

Dacă încap toate obiectele, atunci are sens să ne punem întrebarea „dar dacă eliminăm unul, ce obținem?”. Deci punem în rucsac doar $cnt - 1$ obiecte și simulăm algoritmul lui Sandor (naiv) pentru restul vectorului. Apoi revenim la problema originală, punem în rucsac toate cele cnt obiecte și continuăm.

Așadar, facem $\mathcal{O}(\sqrt{g})$ simulări ale algoritmului, pentru o complexitate totală de $\mathcal{O}(g)$. Este important să nu luăm în calcul valorile care nu încap în rucsac, deoarece complexitatea ar crește la $\mathcal{O}(n\sqrt{g})$. Acele valori le sărim folosind *jump pointers*. Doar le contorizăm, prin diferența între n și numărul de valori pe care le-am luat în calcul. Fiecare valoare sărită ne dă un mod de a obține greutatea algoritmului lui Sandor fără eliminări.

Vom parametriza și cerința 1 pentru a o putea rula începînd cu orice serie și cu orice capacitate reziduală a rucsacului, nu neapărat cu prima serie și cu capacitatea g.

Cerința 2

Dacă reușim să facem același gen de trecere prin vector și pentru cerința 2, și să delegăm subprobleme la algoritmul naiv (fără eliminări) și la cerința 1 (o eliminare), atunci vom obține o complexitate de $\mathcal{O}(g\sqrt{g})$. Pentru aceasta, trebuie să analizăm cazurile posibile.

În primul rînd, cîtă vreme din seria curentă nici măcar un obiect nu încape un rucsac, trecem la seria următoare și contorizăm numărul de obiecte ignorate. Fie acest contor *mult*. Să spunem că avem capacitatea $c = 100$ și am ignorat serii de greutate 200, 150 și 130, totalizînd $mult = 10$ obiecte. Atunci avem $C^2_{mult} = 45$ de moduri de a elimina două din aceste obiecte. Rulăm algoritmul lui Sandor naiv și vedem ce sumă obținem. Adăugăm 45 la răspunsul pentru acea sumă.

Dacă măcar un exemplar încape în rucsac, atunci putem încerca să eliminăm un obiect din seriile anterioare și unul din seria curentă. Deci apelăm cerința 1 începînd cu poziția curentă, și îi transmitem că fiecare soluție găsită trebuie socotită de *mult* ori, deoarece există *mult* moduri de a elimina un obiect dinaintea seriei curente.

Dacă seria curentă include cel puțin două obiecte, putem încerca să eliminăm două obiecte din seria curentă, punînd $cnt-2$ în rucsac. Desigur, există C^2_{cnt} moduri de a face asta, iar pentru restul vectorului rulăm Sandor varianta de bază. Îi trimitem algoritmului parametrizat multiplicatorul C^2_{cnt} pentru soluția pe care o va găsi.

Similar, putem elimina doar un obiect din grupa curentă, punînd $cnt-1$ în rucsac, ceea ce putem face în *cnt* moduri distincte. Iar pentru seriile următoare apelăm cerința 1, spunîndu-i că soluțiile găsite trebuie numărate de cîte *cnt* ori.

În sfîrșit, putem să punem toate obiectele în rucsac și să reconsiderăm cerința 2 de la seria următoare.

4.10.2 Problema Puzzle-bila (Lot 2025)

enunț • sursă

Formularea programării dinamice

Vom denumi **fereastră** o serie de celule libere consecutive.

Pare firesc să ne dorim să calculăm valori de următorul tip. Fie $C_{r,c}$ costul pentru a aduce bila pe rîndul r, coloana c. E destul de clar că $C_{r,c}$ va depinde doar de valori din C de pe rîndul $r-1$. Dacă $C_{r,c}$ va depinde de $C_{r-1,c'}$, atunci va trebui să calculăm și costul de a aduce pe rîndul r o fereastră pe intervalul $[c', c]$.

Așadar, să definim și $D_{r,c,l}$ ca fiind costul de a aduce pe rîndul r o fereastră de lățime (cel puțin) l terminată la coloana c. Acum putem defini recurența pentru C:

$$C_{r,c} = \min_{c'=1}^{c}(C_{r-1,c'} + D_{r,c,c-c'+1})$$

Recurența modelează ideea că mergem pe rîndul $r-1$ pînă la coloana c', apoi coborîm pe rîndul r unde am pregătit o fereastră care ne duce pînă la coloana c.

Reducerea complexității

O soluție în $\mathcal{O}(nm^2)$ procedează astfel: pentru fiecare celulă (r, c) considerăm fiecare fereastră de pe rîndul r. Dacă fereastra are lățime l și trebuie deplasată cu p celule pentru a o alinia la dreapta cu coloana c, atunci putem optimiza $C_{r,c}$ cu cantitatea:

$$p + \text{rmq}(C_{r-1,c-l+1}, \ldots, C_{r-1,c})$$

, unde desigur rmq este funcția de minim pe interval. Doar că există $\mathcal{O}(m)$ ferestre pe fiecare rînd, deci complexitatea este prea mare. Aici intervine descompunerea în valori distincte! Există doar $\mathcal{O}(\sqrt{m})$ lățimi distincte de ferestre pe fiecare rînd. De aceea putem itera doar prin aceste lungimi. Pentru o lungime fixată l, nu are sens să testăm decît cele mai apropiate ferestre din stînga, respectiv din dreapta coloanei curente c. De aici rezultă complexitatea totală $\mathcal{O}(nm\sqrt{m})$.

Implementarea nu este deloc simplă datorită următoarei situații pe care o enunțăm, dar nu o detaliem (ea este descrisă cu exemple în cea de-a doua sursă). Cînd aducem o fereastră din stînga coloanei c, decizia este simplă: trebuie să o deplasăm pînă cînd capătul ei drept atinge coloana c. Dar, cînd aducem o fereastră din dreapta coloanei c, avem libertatea să o deplasăm fie pînă cînd capătul ei stîng atinge coloana c, fie mai mult de atît. Depinde ce coloană $c' < c$ ne interesează să „prindem" în recurență. Poate există o valoare c' destul de mică (costul deplasării pînă acolo este mare), dar unde $C_{r-1,c'}$ este foarte mic și justifică costul deplasării. Ia naștere un al doilea tabel RMQ construit nu peste valorile $C_{r-1,c}$, ci peste $C_{r-1,c} - c$.

Cîteva cuvinte despre Arpa's trick

Eu sînt mereu în căutare de noi structuri pentru RMQ. 😂 Îmi displace tabela rară deoarece folosește $\mathcal{O}(n \log n)$ memorie și evit să mă reped la ea. Iată o structură interesantă, numită Arpa's trick. Aparent, și alte nații au șmenurile lor... CP Algorithms are o lecție foarte bună.

Algoritmul răspunde la interogări în ordine crescătoare după capătul drept. Deci putem întîi să sortăm interogările sau să le distribuim în liste după capătul drept. Apoi folosim o pădure de

mulțimi disjuncte, pe care o instanțiem pe măsură ce baleiem poziția capătului drept.

Cînd ajungem la o poziție nouă p unde se află valoarea x, în primul rînd instanțiem o nouă rădăcină în DSF: părintele lui p este p. În plus, p devine părinte și pentru toate rădăcinile anterioare p' care aveau valori $x' > x$. Procedăm astfel deoarece, pentru orice interogări viitoare, valoarea x' nu va fi niciodată RMQ, deoarece x va face și ea parte din orice interogare care îl conține pe x'. Pentru implementarea acestei părți folosim o stivă ordonată.

La interogarea $\text{rmq}[p', p]$, unde p este poziția curentă, răspunsul este rădăcina arborelui lui p'. Într-adevăr, dorim cea mai mică valoare cu care a fost p' unit la orice moment.

Am rezolvat problema Puzzle-bila folosind *Arpa's trick*. Implementarea este dificilă și ineficientă, deoarece colectarea în avans și ordonarea interogărilor de care vom avea nevoie îngreunează codul (ca să folosesc un eufemism). Dar codul pentru *Arpa's trick* în sine este scurt și clar. El funcționează în $\mathcal{O}(\log^* n)$ amortizat cu memorie $\mathcal{O}(n)$.

```
struct arpa_s_trick {
  int val[MAX_N];
  int parent[MAX_N];
  int n;

  // O stivă cu pozițiile care încă nu au un element mai mic la dreapta.
  int st[MAX_N], ss;

  void reset() {
    ss = 0;
    n = 0;
  }

  void append(int x) {
    while (ss && (val[st[ss - 1]] >= x)) {
      parent[st[--ss]] = n;
    }
    st[ss++] = n;
    parent[n] = n;
    val[n++] = x;
  }

  int find(int p) {
    return (parent[p] == p)
      ? p
      : (parent[p] = find(parent[p]));
  }

  int rmq(int p) {
    return val[find(p)];
  }
};
```

Capitolul 5

Algoritmul lui Mo

Algoritmul lui Mo atinge tot complexitatea de $\mathcal{O}((q+n)\sqrt{n})$, ca și metoda descompunerii în radical, dar printr-o metodă destul de diferită. El duce adesea la cod mai simplu, dar necesită ca interogările să fie date în avans (*offline*).

5.1 Algoritmul lui Mo fără actualizări

Algoritmul lui Mo ordonează interogările și le procesează în această ordine:

1. După blocul capătului stîng.
2. La egalitate, după capătul drept.

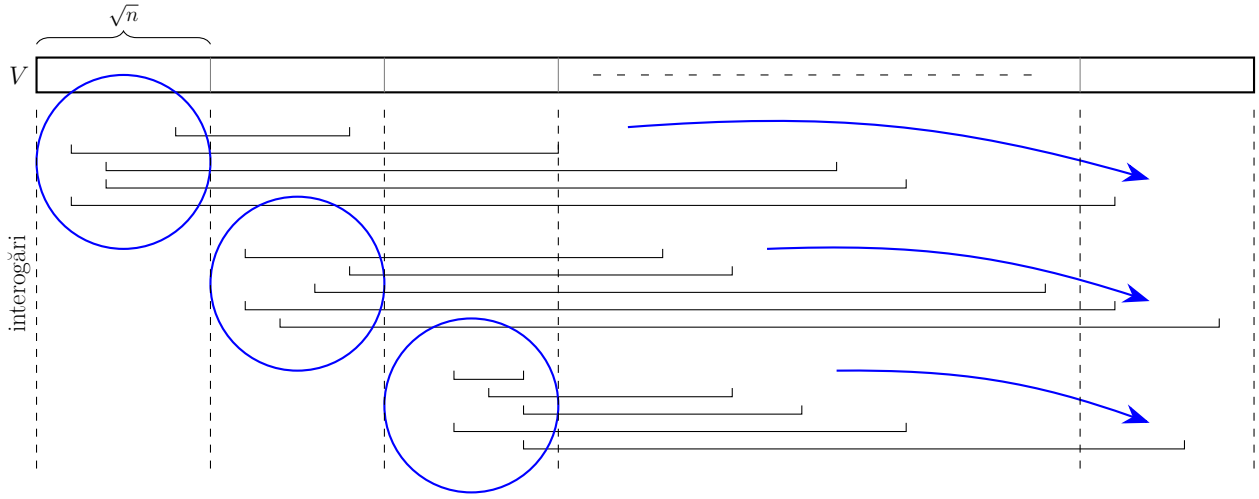


Figura 5.1: Ordonarea interogărilor în algoritmul lui Mo.

Algoritmul lui Mo ține minte informații despre interogarea curentă (răspunsul, dar posibil și alte informații). Pentru a trece la următoarea interogare, algoritmul:

1. Extinde intervalul curent cu cîte un pas, pînă cînd ajunge să includă și următoarea interogare.

2. Restrînge intervalul curent cu cîte un pas, pînă cînd ajunge să coincidă cu următoarea interogare.

Atenție! Pașii trebuie executați în această ordine. Altfel puteți ajunge la intervale negative, din care încercați să eliminați elemente care nu au fost adăugate, cu consecințe neprevăzute.

Vom relua problema D-query (SPOJ) și vom examina implementarea algoritmului lui Mo. Logica este considerabil mai simplă decît la implementarea cu AIB, în $\mathcal{O}(n \log n)$. Surprinzător, timpul de rulare este identic!

5.1.1 Analiză de complexitate

La fiecare interogare, trebuie să deplasăm capetele intervalului curent ca să se suprapună cu interogarea. Care este efortul total?

- Capătul stîng se poate deplasa cu cel mult $\sqrt{n}$ la fiecare interogare, plus n pași pentru toate trecerile între blocuri. Efortul total este de cel mult $q\sqrt{n} + n$ pași.
- Capătul drept se deplasează cu cel mult n pași la dreapta pentru toate interogările dintr-un bloc, apoi cu cel mult n pași înapoi la stînga la trecerea între blocuri. Există $\sqrt{n}$ blocuri, deci efortul total este de cel mult $2n\sqrt{n}$ pași.

Să spunem că includerea sau excluderea unei poziții durează $\mathcal{O}(f(n))$. Pentru problema Dquery efortul este $\mathcal{O}(1)$, dar pentru alte probleme costul poate diferi. Atunci complexitatea totală a algoritmului lui Mo este:

$$\mathcal{O}((q+n)\sqrt{n}f(n))$$

5.1.2 Optimizare de viteză

Nu ne costă aproape nimic (doar un `if` în plus la sortare) ca, în cadrul unui bloc, să ordonăm interogările astfel:

- crescător după capătul drept în blocurile impare;
- descrescător după capătul drept în blocurile impare.

Atunci nu mai plătim costul deplasării spre stînga „în gol” a capătului drept al intervalului curent. Iată o sursă cu această optimizare la problema Dquery. Timpii de rulare nu diferă, dar în general optimizarea poate ajuta.

De amorul artei, strict pentru problema Dquery, am normalizat valorile din vectorul dat. Pot fi cel mult 30.000 de valori distincte, ceea ce înseamnă că vectorul de frecvențe (pe `short`) poate ocupa doar 60 kB, în loc de 2 MB. Din nou, timpii nu diferă.

Aș preciza, tot pentru Dquery, că soluțiile țin foarte bine pasul cu soluția cu AIB (70-75 ms în loc de 60-65).

5.2 Algoritmul lui Mo cu actualizări

Algoritmul lui Mo se pretează și la probleme care au actualizări, cu o complexitate de $\mathcal{O}((q+n)n^{2/3})$. El este în continuare offline, avînd nevoie să citească și să sorteze interogările (și, posibil, să normalizeze valorile).

Esența este să introducem o nouă dimensiune pentru operații, în afară de capetele de interval l și r: timpul t, adică indicele operației (între 1 și q). Structura de date curentă stochează informații despre un interval $[L, R]$ la un moment de timp T, adică despre vectorul cu toate actualizările cu indici mai mici sau egali cu T. Pentru a răspunde la interogarea $[l, r]$ la momentul t, trebuie să:

1. Extindem intervalul curent pînă îl include pe $[l, r]$ (ca și pînă acum).
2. Restrîngem intervalul curent pînă devine egal cu $[l, r]$ (ca și pînă acum).
3. „Avansăm" timpul dacă $T < t$, procesînd actualizările din intervalul $(T, t]$.
4. „Dăm înapoi" timpul dacă $t < T$, inversînd actualizările din intervalul $[t, T)$.

Dacă actualizările efectuate sau inversate se întîmplă în intervalul curent $[l, r]$, actualizăm și structura de date, altfel nu.

5.2.1 Mărimea blocului, ordinea operațiilor, complexitate

Să alegem B blocuri de mărime S. Acum, să sortăm operațiile după blocul lui l, apoi după blocul lui r, apoi după t. Atunci:

Poziția lui l se schimbă cu $\mathcal{O}(S)$ la fiecare operație, plus $\mathcal{O}(n)$ la toate trecerile între blocuri, așadar cu $O(n + qS)$ în total.

Poziția lui r se schimbă cu $\mathcal{O}(S)$ la fiecare operație. În plus, pentru fiecare bloc al lui l (deci de B ori), r va face o trecere prin cele n poziții cu un efort total de $\mathcal{O}(qS + nB)$.

Pentru fiecare pereche de blocuri, t poate trece prin toate cele q operații, cu un efort total de $\mathcal{O}(qB^2)$.

Așadar, complexitatea algoritmului este $\mathcal{O}(qS + nB + qB^2)$. De aceea dorim ca $B^2 \approx S$ și alegem $B \approx n^{1/3}$ și $S \approx n^{2/3}$, pentru o complexitate totală de $\mathcal{O}((q+n)n^{2/3})$.

5.3 Probleme

5.3.1 Problema Powerful Array (Codeforces)

enunț • sursă

Problema este o aplicație directă a algoritmului lui Mo. Am copiat sursa de la D-query și am adaptat-o. Atenție doar la lucrul pe 64 de biți.

5.3.2 Problema Most Frequent Value (SPOJ)

enunț • sursă

Problema este doar puțin mai complicată decît precedentele. Odată ce ne vine ideea că am putea-o rezolva cu algoritmul lui Mo, întrebarea este: ce ne trebuie ca să menținem cea mai mare frecvență?

Răspunsul, desigur, este: menținem toate frecvențele. Dar nu este suficient. Cînd frecvența lui x crește, să spunem de la 7 la 8, atunci frecvența maximă:

- se păstrează dacă era deja 8 sau mai mare;
- crește la 8 dacă era 7 (notă: nu putea fi mai mică de 7, căci x avea frecvența 7).

Pînă aici, toate bune. Dar cînd frecvența lui x scade, să spunem de la 8 la 7? Atunci frecvența maximă:

- se păstrează dacă era mai mare decît 8 (un alt element y avea frecvență maximă);
- se păstrează dacă era 8 și mai există un alt element y cu frecvență 8;
- altfel scade la 7.

Cazul al doilea este critic. Cum aflăm dacă alt element are frecvența egală cu frecvența elementului care iese din interval? Răspunsul este: menținem numărul de elemente care au fiecare frecvență. Un fel de frecvență a frecvențelor, dacă vreți. 🙂 Cînd frecvența lui x scade de la 8 la 7, decrementăm numărul de elemente cu frecvență 8 și îl incrementăm pe cel cu frecvență 7. Dacă frecvența maximă era 8, dar acum nu mai există elemente cu frecvență 8, atunci ne aflăm în cazul al treilea.

5.3.3 Problema RangeMode (Infoarena Cup 2013)

enunț • sursă

Problema seamănă cu precedenta, dar nu cere frecvența maximă, ci elementul minim care atinge această valoare. Nu mai putem folosi fix aceeași abordare din cauza ștergerilor. Dacă există mai multe elemente cu frecvența maximă, și dacă cel mai mic dintre ele dispare, care este următorul minim?

Soluția este simplă: nu facem ștergeri! 😅 Glumesc, iar soluția nu este deloc evidentă. Ea împrumută sortarea interogărilor din algoritmul lui Mo. Atunci o interogare $[l, r]$ constă din:

1. „Partea stîngă”: porțiunea de la l pînă la sfîrșitul blocului curent, care poate varia cu $\mathcal{O}(\sqrt{n})$ elemente între interogări.
2. „Partea dreaptă”: porțiunea de la începutul blocului următor pînă la r, care poate doar să crească între interogări.

Desigur, mai există și cazul particular în care l și r se află în același bloc.

Încercăm să aflăm răspunsul la fiecare interogare aditiv, compunînd partea dreaptă cu partea stîngă. Structura de date pe care o proiectăm, S, menține pentru partea dreaptă un vector de

frecvențe simplu, în $\mathcal{O}(n)$ pentru toate interogările din bloc. Cînd trecem la blocul următor de interogări, golim acest vector. Așadar, efortul total pentru partea dreaptă este $\mathcal{O}(n\sqrt{n})$.

Ce facem cu partea stîngă? Dacă adăugăm elemente în S, va trebui să le ștergem la final. Am vrea să clonăm S pentru fiecare interogare, dar acea operație este scumpă. Iată o soluție struțo-cămilă, dar care funcționează:

1. După ce terminăm de adăugat elementele din partea dreaptă, salvăm din structura S doar răspunsul (elementul cu frecvența maximă).
2. Continuăm să adăugăm în S elementele din partea stîngă și să recalculăm răspunsul.
3. La final, notăm răspunsul. Acesta este răspunsul la interogare.
4. Eliminăm din S elementele din partea stîngă, fără să mai recalculăm răspunsul.
5. Restaurăm răspunsul salvat la pasul (1).

Soluția merge pentru că face ștergeri, dar evită componenta pe care nu știm s-o rezolvăm, a recalculării răspunsului la ștergere.

5.3.4 Problema Machine Learning (Codeforces)

enunț • sursă

Problema cere să admitem 100.000 operații de două tipuri pe un vector cu 100.000:

1. Pentru un interval $[l, r]$ tipărește mex-ul frecvențelor acelui interval. Așadar, tipărește valoarea minimă $f > 0$ pentru care nu există un element de frecvență f în intervalul $[l, r]$.
2. Modifică punctual vectorul, $a[i] = x$.

Implementarea mea este școlărească și medie ca eficiență. Iată ce am învățat din ea.

Pare mai natural să stocăm separat actualizările și interogările.

Ca să pot face *undo* la actualizări, cel mai simplu mi s-a părut să stochez și vechea valoare în fiecare operație de actualizare. Putem obține vechile valori în mod elementar, scriind actualizările într-un vector pe măsură ce le facem (avem nevoie de o copie a vectorului original, pe care o distrugem).

Ca să descriu momentul curent, am ales să mențin un indice în vectorul de actualizări, care pointează la prima operație încă neaplicată. Ca să „derulez" sistemul la momentul de timp t, avansez spre dreapta în vectorul de actualizări dacă timpul operației neaplicate este mai mic decît t, respectiv avansez spre stînga dacă timpul ultimei operații aplicate este mai mare decît t. Ca fapt divers, timpii nu pot fi egali, căci t este timpul unei interogări, care nu va apărea în vectorul de actualizări.

Mi-a fost mai simplu să adaug santinele la timpii 0 și $q + 1$, ca să nu verific condiții suplimentare de ieșire din vector.

Strict referitor la problema Machine Learning: funcția mex pare greu de menținut incremental. Dacă o frecvență f încetează să mai aibă elemente, iar f este mai mic decît mex-ul curent, atunci

f devine noul mex. Aceasta este partea simplă. Dar dacă apare un nou element de frecvența mex-ului, cum recalculăm noul mex? Riscăm să introducem un factor suplimentar la complexitate.

Soluția este de fapt brutală. Pot exista cel mult $\sqrt{n}$ frecvențe diferite (deoarece suma frecvențelor este n, așa cum am arătat în capitolul de descompunere în radical). De aceea, putem calcula funcția mex naiv pentru fiecare interogare cu o complexitate totală de $\mathcal{O}(q\sqrt{n})$, care nu este dominantă.

O ultimă observație specifică problemei: ne interesează doar frecvențele valorilor, nu ordinea lor relativă. De aceea, am ales o variantă mai simplă pentru codul de normalizare.

Partea II

Măiestrie pe biți

Tehnici pentru calcule compacte și eficiente

Capitolul 6

Operații pe biți. Compactarea variabilelor

6.1 Operații elementare

Multe dintre structurile de date pe care le folosim au legătură cu puterile lui 2: arborii indexați binar, arborii de segmente, heapurile, *bitsets*... De aceea, este important să fiți familiari cu o listă de operații utile.

Această listă este inspirată din articolele Bit Twiddling Hacks și Bit manipulation.

6.1.1 Noțiuni de bază

- C are suport pentru constante hexazecimale (`0xdeadbeef`) și binare (`0b1100101101`).
- De asemenea, puteți tipări valori în bazele 8 și 16 cu printf sau folosind STL.
- O cifră hexa are 4 biți în baza 2. Deci `0xce = 0b1100'1110`.
- Uneori valorile hexa ajută la claritate. `0xff'ffff` arată că avem 24 de biți 1, poate mai clar decît `((1 << 24) - 1)`. Dar depinde de gusturi.

6.1.2 Măști

Conceptual, o mască pe n biți este un număr binar care descrie o submulțime a unei mulțimi cu n elemente, indexate de la 0 la $n-1$. Bitul k are valoarea 1 dacă și numai dacă submulțimea descrisă include elementul k. Măștile se pretează la submulțimi mici (32 sau 64 de biți). Dincolo de aceasta avem nevoie de vectori de măști (cum ar fi `bitset` din STL).

Exemple din lumea reală:

- Multe constante predefinite în limbajele de programare ocupă un singur bit (sînt puteri ale lui 2), iar programatorul le poate combina cu OR pe biți. Vezi codurile de eroare din PHP. Programatorul poate decide să afișeze sau să ascundă anumite tipuri de eroare. De exemplu, apelînd funcția `error_reporting(E_ALL & ~E_NOTICE & ~E_DEPRECATED)`, programatorul

arată că vrea să vadă toate erorile în afară de cele minore (`E_NOTICE`) și cele despre perimare (`E_DEPRECATED`).

- În multe programe, utilizatorii au permisiuni, care sînt definite ca puteri ale lui 2, iar permisiunile unui utilizator sînt o mască (o submulțime din mulțimea totală).
- Programele de șah moderne descriu tabla ca pe o colecție de măști pe 64 de biți: una pentru pozițiile pionilor albi, una pentru pozițiile pionilor negri etc. Este foarte convenabil că tabla de șah are fix 64 de pătrate.

Măștile au anumite avantaje, cum ar fi:

1. Este foarte ușor să facem operații de intersecție, reuniune, diferență folosind operatorii pe biți `&`, `|`, `^` și `~`.
2. Valoarea numerică a măștii creează o bijecție naturală între mulțimea $\{0, 1, \ldots, 2^n - 1\}$ și mulțimea submulțimilor unei mulțimi. Deci putem stoca eficient informații despre toate submulțimile într-un vector de 2^n elemente indexate după ordinea lexicografică a submulțimii.

6.1.3 Operații pe măști de biți

operația	**efectul**
`x & 31` sau `x & 0x1f`	Restul împărțirii lui x la 32.
`x & 1`	Test de (im)paritate.
`x & (1 << k)`	Test dacă al k-lea bit este 1. Atenție, nu returnează 0/1, ci $0/2^k$.
`(x >> k) & 1`	Test dacă al k-lea bit este 1. Returnează 0/1.
`x |= (1 << k)`	Setează bitul k pe 1.
`x &= ~(1 << k)`	Setează bitul k pe 0.
`x ^= (1 << k)`	Neagă bitul k.
`x & (x-1)`	Elimină ultimul bit 1 din n. Util și ca test dacă n este putere a lui 2.

Tabela 6.1: Operații pe măști de biți.

6.2 Numărarea biților de 1 dintr-o valoare (popcount)

De exemplu, pentru 187, care în binar este 10111011, dorim răspunsul 6. Vom discuta metodele de mai jos și vom studia codul. Puteți citi și programul complet care măsoară timpii de rulare.

6.2.1 Metoda naivă

Shiftăm în mod repetat numărul la dreapta și numărăm biții de 1.

```
pop = 0;
while (x) {
  pop += x & 1;
  x >>= 1;
```

```
}
```

6.2.2 Metoda Kernighan

Eliminăm și contorizăm cîte un bit de 1.

6.2.3 Funcții built-in

Folosim funcțiile `__builtin_popcount` (pentru `int`) și `__builtin_popcountll` (pentru `long long`) sau `std::popcount` din STL. Atenție, aceasta din urmă apare doar în standardul C++20.

6.2.4 Tabel precalculat

Construim un tabel de 256 de valori care precalculează rezultatul pentru un octet. Extragem octeții numărului prin shiftare.

6.2.5 Tabel precalculat + conversie

Construim același tabel de 256 de valori. Extragem octeții numărului prin conversia la `char*`.

6.2.6 Calcul paralel

Iată o metodă bazată pe calcul paralel, care face `log(sizeof(n))` operații.

Partea III

Arbori

Capitolul 7

Unelte și algoritmi esențiali

Acest curs presupune deja cunoscute reprezentarea arborilor și parcurgerile DFS și BFS. De aceea, în acest capitol vom prezenta doar o unealtă pentru depanare (generatoarele de arbori) și vom rezolva cîteva probleme de bază.

7.1 Generatoare de arbori aleatorii

Generarea unor arbori aleatorii este și interesantă din punct de vedere teoretic, dar vă poate ajuta și la depanarea problemelor pe arbori.

Dacă nu ne interesează forma arborelui, generatoarele de arbori sînt la fel de ușor de scris ca generatoarele de vectori + interogări. Iată un generator de bază care generează un arbore cu n noduri și k interogări de tip pereche de noduri. L-am folosit la problema Max Flow (USACO).

În schimb, dacă dorim să testăm viteza unei soluții, sau dacă compunem o problemă și dorim date de test greu de fentat, avem nevoie ca:

- Arborele să aibă un lanț foarte lung.
- Arborele să aibă un nod cu foarte mulți vecini.
- Aceste structuri să nu poată fi decelate din datele de intrare (de exemplu, lanțul să nu conțină fix nodurile $1, 2, \ldots, n/2$.
- Generarea să dureze $\mathcal{O}(n)$.

Iată un generator avansat care răspunde acestor nevoie și poate genera, la cerere, și interogări sau actualizări. El poate fi apelat în trei moduri și poate fi ușor adaptat la altele:

1. Doar structura arborelui, fără valori în noduri, fără operații.
2. Valori inițiale în fiecare nod, actualizări în noduri, interogări în noduri.
3. Fără valori în noduri, interogări pe căi (perechi de noduri).

El acceptă trei parametri pentru definirea structurii:

1. n: numărul de noduri;
2. c: lungimea minimă garantată a cel puțin unui lanț;

3. d: gradul minim garantat al cel puțin unui nod.

Generatorul funcționează astfel:

- Generează o permutare aleatorie a nodurilor.
- Unește primele c noduri în lanț.
- Unește următoarele $n - d - c$ noduri de noduri dinaintea lor, alese aleatoriu.
- Unește ultimele d noduri de un nod dintre primele $n - d$, ales aleatoriu.
- După ce a generat cele $n - 1$ muchii, le amestecă, astfel încît primele $c - 1$ muchii tipărite să nu formeze un lanț etc.

7.2 Probleme

7.2.1 Problema Subordinates (CSES)

enunț • surse

Cerința este clasică: aflarea mărimii subarborelui fiecărui nod. Am inclus două implementări: cu vectori STL și cu liste înlănțuite scrise de la zero. Implementarea cu liste este de două ori mai rapidă.

7.2.2 Problema Tree Matching (CSES)

enunț • sursă

Problema se rezolvă cu un singur DFS. Ea este un bun exemplu de raționament recursiv pe arbore. Ca în multe alte situații, putem trata nodul curent în relație cu fiul său (dacă nodul și fiul sînt necuplați, cuplează-i) sau în raport cu părintele (dacă nodul și părintele sînt necuplați, cuplează-i).

7.2.3 Problema Tree Diameter (CSES)

enunț • surse

Există două soluții relativ diferite. Una face două parcurgeri DFS (puteți citi aici o demonstrație de corectitudine):

- Facem un DFS pornind din orice nod x. Fie a nodul cel mai depărtat de x.
- Facem un al doilea DFS pornind din a. Fie b nodul cel mai depărtat de a.
- $a \rightsquigarrow b$ este unul dintre diametrele arborelui.

Soluția cu o singură parcurgere este cu 20% mai rapidă (ambele neoptimizate). Implementăm recursiv observația că diametrul constă din două lanțuri descendente care pornesc dintr-un strămoș comun u. (Este ușor de tratat și cazul cînd diametrul este doar un lanț pornind din rădăcină, considerînd atunci al doilea lanț ca fiind rădăcina însăși, o cale de lungime zero.) Atunci fiecare nod are două sarcini:

- Să actualizeze un maxim global cu suma maximă a căilor raportate de oricare doi fii distincți. La fiecare cale, u adaugă 1 (muchia care pleacă din u însuși).
- Să raporteze la părinte distanța maximă de la u pînă la orice frunză din subarbore.

Ambele soluții trebuie să fie scurte (5-6 linii în plus față de șablonul de declarații, citire, DFS).

Discuție secundară: pentru claritate, `diam` nu trebuia să fie variabilă globală, ci DFS-ul trebuia să-l returneze, ca maxim dintre valorile raportate de fii. Pentru viteză, însă, și pentru economisirea memoriei pe stivă, cred că putem face concesia să-l declarăm pe `diam` global. Cititorii mai pedanți decît mine 😅 pot încapsula `diam` și parcurgerea DFS într-o clasă.

7.2.4 Problema Tree Distances II (CSES)

enunț • sursă

Pentru un nod fixat (să zicem 1), putem calcula răspunsul cu un singur DFS. Apoi, vom introduce un concept întîlnit ocazional: recalcularea unei valori la schimbarea rădăcinii. Să spunem că suma cerută pentru nodul u este S_u și dorim să o calculăm pe S_v pentru nodul v, vecin cu u. Să spunem că, raportat la muchia $u-v$, de partea lui u avem n_u noduri, iar de partea lui v avem $n_v = n - n_u$ noduri. Atunci, ca să trecem din S_u în S_v, observăm că:

- pentru n_v noduri distanțele scad cu 1;
- pentru n_u noduri distanțele cresc cu 1;

Rezultă tranziția simplă $S_v = S_u - n_v + n_u = S_u + n - 2n_v$.

Dacă facem toate tranzițiile dinspre rădăcina inițială (1) spre fii, atunci n_v va fi întotdeauna mărimea subarborelui lui v.

7.2.5 Problema White-Black Balanced Subtrees (Codeforces)

enunț • sursă

Includ această problemă pentru a discuta o altă parcurgere decît DFS, în cazul în care arborele este dat printr-un vector de părinți. Am ales o problemă simplă. Pentru una cu mai multă substanță, vedeți Arbsumpow (baraj ONI 2021).

Avantajul acestei reprezentări este că simplitatea și memoria redusă: $n-1$ întregi în loc de $2(n-1)$.

Desigur, puteți converti formatul dat la cel obișnuit (liste de vecini). Dar uneori vectorul de părinți este suficient. În multe probleme trebuie să calculăm pentru fiecare nod o valoare care depinde, recurent, de valorile fiilor. Atunci putem folosi codul:

```
for (int u = 1; u <= n; u++) {
  scanf("%d", &p[u]);
  num_children[p[u]]++;
}
```

```
for (int u = 1; u <= n; u++) {
  int s = u;
  while (s && !num_children[s]) {
    process(s);  // raportează la părinte ce trebuie raportat
    s = p[s];
    num_children[s]--;
  }
}
```

Rolul funcției `process` este să raporteze la părinte informațiile dintr-un nod. Pe parcurs, valoarea `num_children[u]` arată cîți fii ai lui `u` încă nu și-au raportat informațiile. Astfel, cînd îi va veni rîndul părintelui să fie procesat, el va fi acumulat informațiile de la toți fiii.

Ordinea nodurilor este *bottom-up*. Programul încearcă vizitarea fiecărui nod de cel mult două ori: fie cînd îi vine rîndul în ordine numerică, fie în momentul în care ultimul său fiu este vizitat.

7.2.6 Problema Blood Cousins (Codeforces)

enunț • sursă

Soluția propusă în editorial, în $\mathcal{O}(q \log n)$, procedează astfel.

- Măsoară timpii DFS și adîncimea fiecărui nod.
- Pentru a afla eficient al p-lea strămoș, construiește în fiecare nod lista strămoșilor de gradele $1, 2, \ldots, 2^k$.
- Colectează nodurile arborelui, în ordinea DFS, în liste separate pentru fiecare adîncime. Așadar $L[0]$ va conține doar rădăcina, $L[1]$ va conține fiii rădăcinii etc.
- Pentru a afla numărul de descendenți la distanță p ai unui nod u care are timpii DFS $t_i[u]$ și $t_o[u]$, căutăm binar în lista $L[d[u] + p]$ primul și ultimul nod cu timpi DFS între $t_i[u]$ și $t_o[u]$.

Iată și o soluție în timp liniar. Mi se pare mai simplă, căci folosește doar liste și două parcurgeri DFS.

- Distribuim interogările (v, p) în liste după nodul v.
- Rulăm un DFS în care menținem o stivă explicită de noduri în curs de vizitare. Concret, la intrarea în nodul u la adîncime $d[u]$ notăm $st[d[u]] = u$.
- La intrarea în nodul v, procesăm toate interogările (v, p). Al p-lea strămoș al lui v este $st[d[v] - p]$ (dacă există). Înlocuim fiecare p cu acest strămoș.

Astfel, după un prim DFS, toate interogările au luat forma: cîți descendenți are nodul v la adîncime p? Vom scădea 1 pentru a răspunde la cerința problemei. Putem răspunde la aceste întrebări cu un al doilea DFS.

- Redistribuim interogările (v, p) în liste după noul nod v.

- Rulăm un DFS în care menținem numărul de noduri vizitate pe fiecare nivel. Concret, la intrarea în nodul u la adîncime $d[u]$ îl incrementăm pe $st[d[u]]$.
- Acum răspunsul pentru o interogare (v, p) este diferența între $st[d[v] + p]$ la intrarea și la ieșirea din nodul v. Iterăm prin toate interogările referitoare la nodul v la intrarea și la ieșirea din recursivitate.

Capitolul 8

Liniarizarea arborilor

Să considerăm un arbore cu n noduri și cu valori în noduri. Liniarizarea acestui arbore este o parcurgere DFS care calculează informații suplimentare. Putem folosi aceste informații ca să gestionăm operații pe arbore, cum ar fi:

- Modificarea valorii unui nod sau a unei muchii.
- Modificarea valorilor pe calea de la un nod la rădăcină.
- Modificarea valorilor în tot subarborele unui nod.
- Interogări despre valoarea unui nod.
- Interogări despre valorile pe calea de la un nod spre rădăcină.
- Interogări despre valorile din subarborele unui nod.

8.1 Timpi de intrare și de ieșire din DFS

Să considerăm următoarea parcurgere DFS. Ea este elementară, cu o singură noutate: menține un contor global pe care îl incrementează la intrarea și la ieșirea dintr-un nod. (Din motive inginerești, ca să evidențiez că variabila `time` este folosită doar în funcția `dfs`, nu am declarat-o globală, ci statică în funcție. Efectul este același.)

```
void dfs(int u, int parent) {
  static int time = 0;

  time_in[u] = ++time;
  for (int v: adj[u]) {
    if (v != parent) {
      dfs(v, u);
    }
  }
  time_out[u] = ++time;
}
```

Dacă vreți, `time` este un metronom care bate la fiecare parcurgere a unei muchii, fie în jos (la

intrarea într-un nod), fie în sus (la ieșirea dintr-un nod). Toate valorile din vectorii `time_in` și `time_out` vor fi distincte și cuprinse între 1 și $2n$. Iată un exemplu.

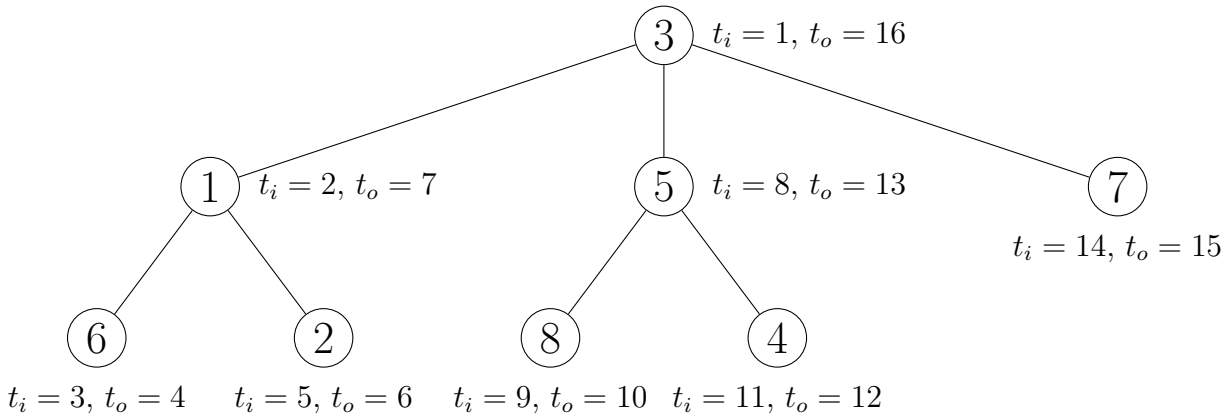


Figura 8.1: Timpii de intrare și de ieșire din noduri, varianta 1.

Într-o altă variantă, incrementăm timpul numai la intrarea în nod, nu și la ieșire. Atunci în vectorii t_i și t_o vom avea valori între 1 și n. Iată aceste valori pentru același arbore:

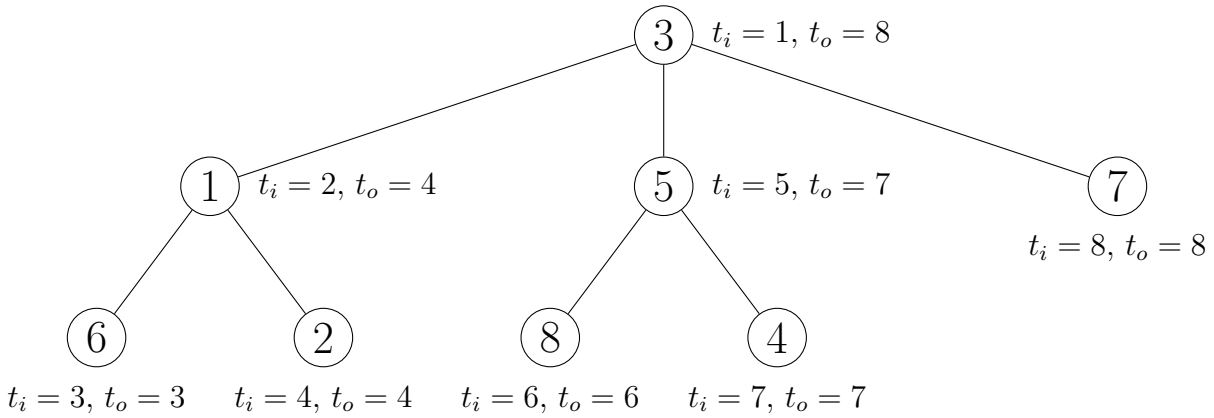


Figura 8.2: Timpii de intrare și de ieșire din noduri, varianta 2.

8.2 Testul de strămoș

O aplicație directă a timpilor de intrare și de ieșire este că putem decide în $\mathcal{O}(1)$ dacă un nod u este strămoș al altui nod v. Este necesar ca $t_i[u] < t_i[v] < t_o[v] < t_o[u]$. Inegalitatea poate fi strictă ($<$) sau permisivă ($\leq$) în funcție de varianta aleasă mai sus și dacă dorim ca u să fie considerat propriul său strămoș sau nu.

8.3 Liniarizarea. Tipuri de liniarizare

Liniarizarea unui arbore este un vector. Obținem acest vector printr-o parcurgere DFS în care emitem, la anumite momente, numărul nodului curent. Acest instrument puternic ne permite

să rezolvăm probleme pe arbori folosind structuri familiare pe vectori: AIB, arbori de segmente, căutări binare, algoritmul lui Mo...

Există trei tipuri de liniarizări, foarte asemănătoare și la fel de ușor de obținut. Fiecare este utilă în alte situații.

8.3.1 Liniarizarea DFS

În această liniarizare, emitem nodul curent (adică îl adăugăm la vectorul rezultat) cînd intrăm în nod. Pentru arborele din Figura 8.2, vectorul este:

poziție	1	2	3	4	5	6	7	8
nod	3	1	6	2	5	8	4	7

Tabela 8.1: Liniarizarea de tip DFS.

Cu alte cuvinte, această liniarizare constă din nodurile arborelui în ordinea în care le descoperă DFS-ul. Tocmai de aceea, fiecare nod u apare în vector la poziția $t_i[u]$ (în varianta 2, din figura 8.2). De altfel, pentru acest tip de liniarizare în practică nu vom construi efectiv vectorul, ci doar vectorii t_i și t_o.

O observație importantă pentru toate liniarizările este că subarborele oricărui nod corespunde unui interval contiguu din vector. De exemplu, nodurile 5, 8 și 4 apar pe poziții consecutive. În liniarizarea DFS, subarborele oricărui nod u acoperă pozițiile dintre $t_i[u]$ și $t_o[u]$ inclusiv. De exemplu, subarborele nodului 5 ocupă pozițiile de la 5 la 7. Pentru a exploata această structură, în probleme de arbori cu valori în noduri vom stoca într-un vector valoarea fiecărui nod u pe poziția $t_i[u]$. Atunci, folosind structurile cunoscute pe vector, putem face actualizări și interogări pe subarbori întregi.

8.3.2 Liniarizarea Euler

În această liniarizare, emitem nodul curent de două ori: la intrare și la ieșire. Pentru arborele din Figura 8.1, vectorul este:

poziție	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
nod	3	1	6	6	2	2	1	5	8	8	4	4	5	7	7	3

Tabela 8.2: Liniarizarea de tip Euler.

Observăm că pozițiile nodului u în vector sînt $t_i[u]$ și $t_o[u]$. Vom studia probleme care arată cum folosim această liniarizare pentru a admite actualizări și interogări pe calea de la rădăcină la un nod oarecare u.

8.3.3 Liniarizarea Euler cu repetiție

Pe Internet nu am găsit o distincție clară între liniarizarea anterioară și aceasta, așa că am inventat un nume, sper că acceptabil. În această liniarizare, emitem nodul curent la intrare și la revenirea

din fiecare fiu. În particular, emitem frunzele o singură dată. Pentru arborele din Figura 8.1, vectorul este:

poziție	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
nod	3	1	6	1	2	1	3	5	8	5	4	5	3	7	3

Tabela 8.3: Liniarizarea de tip Euler cu repetiție.

Lungimea acestei liniarizări este întotdeauna $2n-1$. Emitem fiecare nod la intrare, ceea ce ocupă n poziții. În plus, emitem fiecare nod de atîtea ori cîți fii are. Dar suma numărului de fii pentru toate nodurile este $n-1$, deoarece fiecare nod este fiul cuiva, cu excepția rădăcinii.

Folosim această liniarizare cu preponderență pentru interogări de LCA (engl. *lowest common ancestor*), pe care le vom studia în capitolul 10.

8.4 Probleme

8.4.1 Problema Tree Queries (Codeforces)

enunț • sursă

Probabil am putea implementa următoarea soluție. Pentru fiecare interogare, pornim din cel mai de jos dintre noduri, mergînd pînă la rădăcină, și vedem dacă întîlnim toate celelalte noduri sau părinții acestora. Dar această soluție necesită $\mathcal{O}(n)$ per interogare pentru un arbore degenerat. Probabil ea poate fi adusă la $\mathcal{O}(\sqrt{n})$ sau $\mathcal{O}(\log n)$ prin tehnici mai complicate.

În loc de acesta, să procedăm astfel. Fie u cel mai de jos nod dintr-o interogare. Atunci toate celelalte noduri **sau** părinții lor trebuie să se afle pe calea de la u la rădăcină. Cu alte cuvinte, să fie strămoși ai lui u. Mai mult, decizia se simplifică astfel: dacă un nod v este pe calea de la u la rădăcină, atunci și părintele lui v se va afla pe calea de la u la rădăcină. Deci este suficient să testăm doar părinții și să răspundem la întrebarea: Sînt părinții tuturor nodurilor dintr-o interogare strămoși ai celui mai de jos nod? (Complicația cu noduri și părinți este mai mult un *red herring*, o diversiune care poate păcăli pe cineva.)

Vom folosi testul de strămoș studiat anterior: v este strămoș al lui u dacă și numai dacă $t_i[v] < t_i[u] < t_o[u] < t_o[v]$.

Detalii de implementare

Am preferat să declar variabila `time` statică, în interiorul funcției `euler_tour`, ca să evidențiez că nu este folosită altundeva.

Nodurile dintr-o interogare nu trebuie stocate, sortate etc. Nu avem nevoie de adîncimea nodurilor. Pur și simplu îl reținem pe cel mai de jos găsit pînă atunci, fie el l. Cînd citim un nod nou, u, îi aflăm părintele p. Pot lua naștere trei cazuri:

- Dacă p este strămoș al lui l, atunci l nu se modifică, iar calea care conține toate nodurile de pînă acum rămîne calea de la l la rădăcină.
- Dacă l este strămoș al lui p, atunci p îl înlocuiește pe l, iar calea care conține toate nodurile de pînă acum este calea de la p la rădăcină.
- Altfel nu există nicio cale care să le conțină pe l și pe p, deci răspunsul la interogarea curentă este `NO`.

8.4.2 Problema Subtree Queries (CSES)

enunț • sursă

Această problemă este o aplicație directă a liniarizării. Facem o liniarizare de tip DFS pentru a calcula doar vectorul t_i (timpul de intrare în fiecare nod). Apoi construim un vector indexat după timp, în care notăm valoarea fiecărui nod u la poziția $t_i[u]$. Peste acest vector construim un simplu arbore Fenwick sau orice structură preferați care să ofere actualizare punctuală + sumă pe interval.

O altă variantă clasică a acestei probleme ar fi ca operațiile de actualizare să fie pe subarbore: atribuie tuturor nodurilor din subarborele lui u o valoare x, incrementează toate nodurile cu o cantitate x etc. Desigur, aceste operații s-ar traduce în operații de actualizare pe interval, pentru care putem folosi un arbore de intervale cu propagare *lazy*.

8.4.3 Problema Path Queries (CSES)

enunț • sursă

Această problemă ne arată utilitatea liniarizării Euler. Să considerăm momentul în care DFS-ul intră în nodul u, adică $t_i[u]$. El corespunde prefixului din liniarizare de la poziția 1 pînă la poziția $t_i[u]$. Observăm că:

1. Nodurile complet vizitate înainte de intrarea în u (numite și *noduri negre*) apar de cîte două ori.
2. Nodurile în curs de vizitare (calea de la u la rădăcină, numite și *noduri gri*) apar cîte o dată.
3. Nodurile încă nevizitate (numite și *noduri albe*) nu apar niciodată.

Putem exprima punctul (2) și în termeni de strămoși, discutați anterior: strămoșii lui u sînt singurele noduri ale căror intervale cuprind intervalul lui u.

De aici rezultă cum putem folosi paritatea aparițiilor ca să notăm informații de pe calea spre rădăcină. Notăm valoarea fiecărui nod v cu semnul + la poziția $t_i[v]$ și cu semnul - la poziția $t_o[v]$. Astfel, suma prefixului $[1, t_i[u]]$ va fi exact suma valorilor pe calea de la u la rădăcină.

8.4.4 Problema New Year Tree (Codeforces)

enunț • sursă

Problema este relativ directă, doar cu puțin mai complicată decît precedentele. După liniarizare, avem nevoie de o structură de date care să admită operațiile:

1. `range_set(l, r, val)`: scrie valoarea val pe tot intervalul $[l, r]$
2. `range_count_distinct(l, r)`: returnează numărul de valori distincte din intervalul $[l, r]$.

Problema numărării de elemente distincte cu actualizări nu este trivială, dar varianta de față are un avantaj: valorile sînt între 1 și 60, deci putem folosi măști de biți. Rezultă că avem nevoie de un arbore de intervale cu propagare *lazy*, cu operațiile:

1. atribuiri pe interval;
2. OR pe biți pe interval.

8.4.5 Problema Max Flow (USACO)

enunț • surse

Metoda 1: Vectori de diferențe pe arbore

O primă soluție aduce cu vectori de diferențe („șmenul lui Mars"), dar în varianta arborescentă. Pentru fiecare nod dorim să calculăm numărul de căi care trec prin acel nod. Atunci, pentru fiecare cale (u, v), dorim să incrementăm valorile tuturor nodurilor de pe cale. Fie w cel mai de jos strămoș comun (LCA) al perechii (u, v). Atunci dorim, echivalent,

- Să incrementăm toate valorile de la u la rădăcină.
- Să incrementăm toate valorile de la v la rădăcină.
- Să scădem cu 1 valoarea lui w (care a fost incrementat de două ori).
- Să scădem cu 2 valorile de la părintele lui w la rădăcină.

Putem face asta adunînd ± 1 în nodurile respective, apoi propagînd în sus acele valori. Implementarea este lungă întrucît trebuie să calculeze LCA și am ales să fac asta cu algoritmul lui Tarjan, cel mai eficient în situația offline (vom detalia în viitorul apropiat).

Metoda 2: AIB peste liniarizare

La momentul vizitării nodului u, putem împărți arborele în 3 zone disjuncte:

1. Zona 1 este porțiunea vizitată înainte de u.
2. Zona 2 este subarborele lui u.
3. Zona 3 este porțiunea care va fi vizitată după revenirea din u.

Atunci, căile care trec printr-un nod u sînt de trei feluri:

1. Cele care încep în zona 1 și se termină în zona 2.
2. Cele care încep în u, indiferent dacă se termină în zona 2 sau în zona 3.
3. Cele care pornesc dintr-un descendent al lui u și trec prin u, fie că se termină în alt descendent al lui u, fie în zona 3.

Pentru a afla aceste informații pentru fiecare nod, facem întîi o liniarizare de tip DFS imediat ce citim arborele. Reamintim că subarborele fiecărui nod ocupă un interval contiguu în această liniarizare.

Abia acum citim restul datelor (căile). Pentru fiecare nod u colectăm lista de căi care încep din u și lista de căi care se termină în u. Mai exact, pentru fiecare cale (u, v), cunoscînd liniarizarea, ne asigurăm că $t_i[u] < t_i[v]$ (le interschimbăm la nevoie). Apoi adăugăm v la lista de căi care pornesc din u și u la lista de căi care se termină în v.

Acum putem rula o a doua parcurgere DFS în care ținem evidența căilor active. La intrarea într-un nod marcăm ca active căile care pornesc din u. La ieșire, marcăm ca inactive căile care se termină în u (adică le ștergem din evidență).

Care este structura necesară pentru această evidență? De exemplu, pentru întrebarea (1) de mai sus avem nevoie să știm ce căi active se termină în zona 2. Putem folosi un arbore Fenwick peste liniarizare în care notăm $+1$ la momentul nodului de sfîrșit al căilor active. Atunci răspunsul la (1) este suma pe $[t_i[u], t_o[u]]$.

Pentru (2) răspunsul este pur și simplu lungimea listei de căi care încep în u.

Pentru (3), la revenirea dintr-un fiu v avem nevoie să știm ce căi active au început în subarborele lui v. De aceea, vom folosi un al doilea arbore Fenwick în care notăm $+1$ la momentul nodului de început al căilor active.

Această metodă nu mai necesită calculul LCA.

Metoda 3 (cred): *Small to large*

Cred că fiecare nod își poate menține colecția de căi care trec prin el. Un părinte va unifica aceste colecții, eliminîndu-le pe cele care apar de două ori (pentru care părintele este LCA). Dacă folosim cea mai mare colecție dintre cele ale fiilor, timpul rezultat va fi $\mathcal{O}(n + q \log q)$.

8.4.6 Problema Distinct Colors (CSES)

enunț • sursă

După liniarizare și după normalizarea culorilor, problema se reduce destul de direct la numărarea culorilor distincte dintr-un interval. O variantă este cu algoritmul lui Mo, în $\mathcal{O}(n\sqrt{n})$. O altă variantă posibilă este cu tehnica *small to large* (vom reveni cînd studiem tehnica).

Soluția mai eficientă, în $\mathcal{O}(n \log n)$, este cea studiată în capitolele anterioare (problema D-query). Procesăm interogările în ordinea crescătoare a capătului drept. Pe parcursul procesării, într-un AIB ținem valori de 1 pentru poziția celei mai din dreapta apariții a fiecărei culori.

În practică implementarea se simplifică mult, căci interogările pe liniarizare au o structură foarte specială. Dacă răspundem la interogări la revenirea din nod, atunci în mod evident ele vor fi ordonate după capătul drept, deoarece capătul drept este dat de timpul curent din DFS, care poate doar să crească! De aceea,

1. Nu mai este nevoie să colectăm și să sortăm interogările. Le putem procesa chiar în DFS.
2. Putem renunța complet la stocarea timpilor de intrare în nod.
3. Nu (prea) mai are sens să normalizăm culorile în prealabil. Putem extinde AIB-ul să se ocupe de asta.

De aceea vă încurajez mereu să adaptați cunoștințele voastre la particularitățile problemei. Per ansamblu, veți cîștiga timp și eficiență.

8.4.7 Problema Disconnect (Infoarena)

enunț • sursă

Precizez că testele par incorect alese. Soluțiile cele mai rapide traversează naiv arborele. Ne facem că n-am observat. 😀

O rezolvare directă este similară cu problema Max Flow. Marcăm muchiile șterse cu 1. Apoi, două noduri sînt conectate dacă suma pe calea dintre ele este 0. Pentru a calcula suma pe o cale, calculăm sumele de la fiecare nod pînă la rădăcină, minus dublul sumei de la LCA la rădăcină.

Iată și o altă rezolvare, care evită nevoia de a calcula LCA. Alegem o rădăcină oarecare (1). Cînd ștergem o muchie, marcăm nodul de adîncime mai mare ca șters. Acum definim un concept clasic: pentru orice nod u, fie $LMA(u)$ (engl. *lowest marked ancestor*) cel mai de jos nod marcat pe calea de la u la rădăcină, inclusiv.

Pentru a răspunde la o interogare de conectivitate (u, v), considerăm 4 cazuri:

1. $LMA(u)$ și $LMA(v)$ sînt nedefinite. Atunci, desigur, calea este liberă și u și v sînt conectate.
2. Exact unul dintre $LMA(u)$ și $LMA(v)$ este definit. Atunci nodul marcat există undeva mai jos de $LCA(u, v)$, așadar el deconectează u de v.
3. $LMA(u)$ și $LMA(v)$ există și sînt egale. Atunci nodul marcat există undeva mai sus de $LCA(u, v)$, deci nu există alte obstacole. u și v sînt conectate.
4. $LMA(u)$ și $LMA(v)$ există și sînt diferite. Atunci fiecare nod are propriul său blocaj mai jos de $LCA(u, v)$, deci nodurile sînt deconectate.

Condiția este mai ușor de testat decît pare. Dacă lăsăm $LMA(u) = 0$ pentru valori nedefinite, atunci trebuie doar să testăm dacă $LMA(u) = LMA(v)$.

Cum menținem informația de LMA? Este suficient un singur arbore de segmente, fără propagare *lazy*, construit peste liniarizare. La marcarea unui nod u ca șters, marcăm tot intervalul subîntins de u chiar cu valoarea u. La interogarea $LMA(v)$, pornim din frunza din AINT corespunzătoare nodului v și raportăm prima valoare întîlnită (sau 0 dacă nu întîlnim valori scrise).

Mai există un singur aspect important. Un nod din AINT poate fi acoperit de mai multe noduri din arbore. Dacă mai multe actualizări (ștergeri) acoperă același nod, cînd suprascriem valori și cînd nu? Este important că nodurile respective din arbore sînt toate în relația strămoș-descendent. De aceea, cele mai de jos au prioritate. Echivalent, cele care subîntind un interval mai mic în

AINT au prioritate. AINT-ul procedează exact astfel: acordă valorilor scrise priorități egale cu lungimea intervalului scris.

Capitolul 9

Tehnica small-to-large

9.1 Generalități

Există o clasă de probleme pe arbori care, implementate naiv, au complexitate $\mathcal{O}(n^2)$. În aceste probleme, informația pentru un nod are mărime $\mathcal{O}(n)$ și trebuie compusă din informațiile fiilor. Exemplu: fiecare nod u dorește să-și cunoască frecvențele adîncimilor nodurilor din subarbore, raportate la u. Cu alte cuvinte, u dorește să știe cîți fii, nepoți, strănepoți etc. are. Pentru a calcula acest vector, u trebuie:

1. să însumeze vectorii primiți de la fii;
2. să translateze cu 1 toate valorile (fiul fiului lui u este nepotul lui u);
3. să se adauge pe sine însuși la distanță 0.

Pentru un arbore degenerat (un lanț de lungime n), o implementare naivă va copia și modifica succesiv vectori de lungime $1, 2, 3, \ldots, n$, așadar $\mathcal{O}(n^2)$ în total.

Tehnica *small-to-large* spune: este OK să transferăm O(mărimea fiului) informații de la fiecare fiu la părinte, **cîtă vreme ne asigurăm că transferăm din structura mai mică în cea mai mare**. Astfel, fiecare informație va fi transferată într-o structură de (cel puțin) două ori mai mare, deci numărul de transferuri este limitat la $\log n$. Complexitatea totală este $\mathcal{O}(n \log n)$ sau, în unele cazuri, chiar $\mathcal{O}(n)$.

Pentru exemplul de mai sus, am putea stoca vectorii de frecvență ca liste, astfel încît părintele să se poată adăuga la începutul listei. Aceasta rezolvă cerințele (2) și (3) în $\mathcal{O}(1)$. Pentru cerința (1), este suficient ca părintele să compare acumulatorul (lista cu frecvențele fiilor văzuți pînă acum) cu lista primită de fiul curent și să o adune pe cea mai scurtă la cea mai lungă.

9.2 Probleme

9.2.1 Problema Fixed-Length Paths I (CSES)

enunț • sursă

Putem rezolva problema exact cu ideile expuse teoretic. Fiecare nod raportează la părintele său o listă cu numărul de noduri aflate la distanțe 0 (el însuși), 1 (fiii), 2 (nepoții) etc. Pentru a-și calcula lista, fiecare nod:

1. Însumează listele primite de la toți fiii.
2. Deplasează lista rezultată cu o poziție (fiul fiului meu este nepotul meu).
3. Se adaugă pe sine însuși la distanță 0.

Pentru implementare, putem folosi un deque, care oferă inserarea la început în $\mathcal{O}(1)$. Pentru însumarea listelor, adăugăm mereu lista mai scurtă la cea mai lungă. În plus, fiecare nod u răspunde la întrebarea: pentru cîte căi de lungime k sînt eu LCA (nodul cel mai de sus de pe cale)? Apoi adaugă această valoare la un contor global. Pentru a calcula răspunsul, nodul u confruntă lista primită de la fiecare fiu v cu listele obținute și însumate anterior: dacă fiul v raportează a noduri la o distanță l și fiii anteriori raportaseră b noduri la o distanță $k-2-l$, adăugăm $a \cdot b$ la contorul global.

Care este complexitatea? Pare că ar fi $\mathcal{O}(n \log n)$, după cum spuneam: cînd copiem un nod din lista L_1 în lista L_2, la final L_2 va stoca informații (frecvențe) despre cel puțin dublul numărului de noduri din L_1. Deci fiecare nod poate fi copiat de cel mult $\log n$ ori.

Dar putem da o limită și mai strînsă. Pentru această problemă soluția este, de fapt, $\mathcal{O}(n)$! Programul nu operează cu noduri, ci doar cu distanțe. Cînd frecvența unei distanțe d în L_1 este 2 sau mai mare, nu mai pierdem timp individual ca să copiem acele noduri din L_1 în L_2, ci le „copiem" pe toate simultan, însumînd niște frecvențe. Practic, fiecare nod este copiat o singură dată. Vă puteți convinge de asta adăugînd contoare globale în interiorul buclelor critice și verificînd că ele nu depășesc valoarea totală n.

Notă: pentru a interschimba două liste, folosiți întotdeauna metoda `swap()`, care doar schimbă pointeri între ei. Niciodată nu folosiți operatorul `=`, care face copieri de elemente și are complexitate $\mathcal{O}$(lungime)!

9.2.2 Problema Distinct Colors (CSES) (din nou)

enunț • sursă

Să reluăm această problemă și să o rezolvăm cu tehnica *small-to-large*. O variantă ar fi ca fiecare nod să țină o listă sortată de culori. Atunci părintele trebuie să interclaseze listele fiilor. Dar nu putem face interclasarea în O(lista mai scurtă), ci doar în O(suma lungimilor). Aceasta duce la o soluție în $\mathcal{O}(n^2)$. Exemplu: $n = 1.000.000$, iar rădăcina are 1.000 de fii, fiecare cu cîte 1.000 de noduri în subarbore. Toate culorile sînt distincte. Atunci costul total al interclasărilor în rădăcină ar fi:

$$2.000 + 3.000 + \cdots + 1.000.000$$

Ca să combinăm doi fii în O(fiul mai mic), putem menține culorile într-o tabelă hash (`unordered_set`).

Atunci este ușor să „vărsăm” tabela mai mică în cea mai mare.

Două observații de implementare:

1. Această variantă consumă mai mult spațiu pe stivă, căci are variabile locale mai mari. Pe calculatorul meu local, testele mari chiar umplu stiva!
2. Această variantă este mai lentă decît cea cu AIB (nicio surpriză).

9.2.3 Problema Lomsat Gelral (Codeforces)

enunț • surse

Folosim tehnica *small-to-large*, așa cum se poate deduce și din numele problemei 🦊. Este important ca fiecare fiu să stocheze $\mathcal{O}$(subarbore) informații, nu $\mathcal{O}(n)$. Deci vom folosi o tabelă hash de culori cu frecvențele lor.

Ca observație interesantă, este mult mai eficient să calculați și să returnați din DFS tabela hash și celelalte informații, decît să le stocați în fiecare nod. Iată și o astfel de implementare, considerabil mai lentă.

Există și o soluție cu algoritmul lui Mo, puțin mai lentă, dar care consumă mai puțină memorie. Este necesară atenție la detalii. Ce informații stochează intervalul curent? Vă recomand să izolați acele structuri de date într-un `struct` sau o clasă separată.

Problemă similară: Christmas Balls (IIOT 2021/22 runda 2).

9.2.4 Problema Tokens on a Tree (CodeChef)

enunț • surse

O problemă echivalentă este Short Code (inspirată din viața reală, avînd în vedere cum își denumesc unii elevi variabilele).

Să demonstrăm teoretic ce avem de făcut. Iată diverse observații.

Observația 1. O monedă A poate „sări” aparent peste altă monedă B: practic, monedele fiind identice, o urcăm pe B pînă la destinația dorită, apoi pe A în locul lui B.

Observația 2. În orice soluție, monedele se vor afla la vîrful arborelui. Niciodată nu vom avea o monedă sub un nod gol, căci am putea face o mutare în plus.

Observația 3. Dacă rădăcina conține inițial o monedă, atunci acea monedă nu pleacă nicăieri și nicio altă monedă nu îi va lua locul. Deci putem rezolva problema independent pentru subarbori.

Observația 4. Dacă rădăcina nu conține inițial o monedă, atunci dintr-unul dintre subarborii fiilor o monedă va urca în rădăcină. Restul fiilor vor fi rezolvați independent, conform Observației 3. Moneda care va urca în rădăcină este cea de adîncime maximă după ce rezolvăm fiii, ca să facem cît mai mulți pași în plus. Urcarea este aparentă, conform Observației 1.

Așadar, implementarea trebuie să combine în mod eficient fiii unui nod, apoi să găsească cea mai de jos monedă și să o urce în părinte. Am găsit două variante de implementare.

Implementare cu *small-to-large*

Fiecare nod u calculează un vector/listă f unde $f[i]$ pentru $i \geq 0$ este numărul de monede la distanță i de u. Fiecare părinte însumează listele fiilor așa cum știm, iar algoritmul este $\mathcal{O}(n)$, căci nodurile odată comasate își pierd identitatea. În plus, nodul u se adaugă pe sine însuși la începutul listei și, dacă este loc, simulează urcarea unei monede transferînd o unitate de pe ultima poziție din f pe prima.

Detalii de implementare. Implementarea cu `std::list` este cu vreo 25 de linii mai scurtă decît cea cu liste proprii, dar este de două ori mai lentă și consumă dublul memoriei. Implementarea cu `std::deque` este nefolosibilă ca timp și ca memorie (spre 1 GB). Probabil, clasa `deque` are costuri fixe, iar multe deque-uri mici sînt foarte scumpe.

Implementare cu liniarizare + RMQ

Construim o liniarizare DFS și notăm, în nodurile care conțin monede, adîncimea acelor noduri. În nodurile care nu conțin monede nu notăm nimic. Pentru a găsi cea mai de jos monedă din subarborele unui nod u, găsim maximul pe intervalul subîntins de u. Pentru a muta moneda, scriem 0 în locul acelui maxim (moneda dispare) și scriem adîncimea lui u la poziția nodului u (moneda apare acolo). Apoi creștem numărul total de mutări cu diferența dintre cele două adîncimi. Avem nevoie de RMQ cu actualizare, deci putem folosi varianta pe arbori de segmente. Rezultă o complexitate de $\mathcal{O}(n \log n)$. Codul este de 2-3 ori mai lent și consumă mai multă memorie decît implementarea cu *small-to-large*.

9.2.5 Problema Blood Cousins Return (Codeforces)

enunț • surse

Toate soluțiile încep prin a transforma numele în numere. Cea mai la îndemînă implementare folosește un `unordered_map`.

Implementarea cu *small-to-large*

O soluție destul de brutală cu *small-to-large* este: fiecare nod u ține un vector de set-uri, unde setul de pe poziția i reține numerele distincte regăsite la adîncime i (raportat la adîncimea lui u) în subarborele lui u. Cînd combinăm doi vectori, combinăm două cîte două seturile reprezentînd aceeași adîncime. Trebuie să avem grijă să folosim *small-to-large* în două locuri:

1. Copiem vectorul mai mic în cel mai mare.
2. Pentru fiecare pereche de seturi, îl copiem pe cel mai mic în cel mai mare.

Note de implementare:

1. `unsorted_set` este mai lent decît `set`, deși algoritmic vorbind ne-ar trebui prima, care oferă timp constant. Dar tabelele hash (adică `unsorted_set`) au o mărime minimă dată de vectorul pe care se bazează.
2. Reprezentarea listelor de seturi cu `deque` este de două ori mai lentă decît cu `vector`.

Implementarea este rezonabil de scurtă, dar nu prea eficientă.

Implementarea cu tehnici de bază

Iată și o abordare care necesită doar parcurgeri, liniarizare și căutare binară. Soluția este greu de codat, dar este mai rapidă.

1. Calculăm ordinea BFS. Atunci răspunsul la orice interogare se va traduce în numărarea elementelor distincte de pe un interval contiguu din BFS. Știm să facem asta folosind un simplu AIB și ordonînd interogările după capătul drept (vezi problema D-query).
2. Pentru fiecare interogare, rămîne să aflăm primul și ultimul nod de la o anumită adîncime din subarborele lui u, ca să știm pe ce interval din BFS facem numărarea. Putem face asta cu două parcurgeri DFS și o stivă. La intrarea în fiecare nod, notăm valoarea sa pe stivă în dreptul adîncimii sale. La revenirea din nod, lăsăm stiva intactă. Atunci, la revenirea într-un nod u aflat la adîncimea d, putem ști care este **ultimul** său descendent de la adîncimea d': fie nu există, fie este nodul de la poziția d' de pe stivă. Pentru a afla **primul** descendent de la adîncimea dorită, facem același DFS, dar iterînd prin fiii nodurilor în ordine inversă.

Sună rezonabil din vorbe, dar prima implementare a fost migăloasă.

O soluție considerabil mai scurtă (cea inclusă în anexă) procedează astfel:

1. Calculăm aceeași ordine BFS.
2. Înlocuim fiecare interogare $\langle u, d\rangle$ cu interogarea $\langle depth[u]+d, t_{in}[u], t_{out}[u]\rangle$. Cu alte cuvinte, fiecare interogare interoghează un interval aflat la o anumită adîncime și cu noduri între timpii dați. Sortăm și procesăm interogările cu un AIB ca de obicei.

O a treia implementare este: colectăm nodurile separat pe fiecare nivel, în ordinea timpilor de vizitare (care este chiar ordinea DFS). Acum intervalul de interes pentru o interogare referitoare la nodul u este pe un nivel cunoscut și este delimitat de timpii de intrare și de ieșire din u.

Este greu să construim cîte un AIB pe fiecare nivel, dar putem folosi un `set` cu aceeași informație: pozițiile ultimelor apariții ale fiecărui element distinct. Iată o implementare curată a fostului olimpic Alex Nuță.

9.3 DFS exclusiv

Am învățat această tehnică din sursa unui legendary grandmaster. 🤩 Ulterior am aflat că ea se mai numește și Sack sau DSU pe arbori. Pot fi de acord cu prima denumire, dar nu și cu a doua, în primul rînd pentru că DSU este un nume greșit pentru DSF (engl. *disjoint set forest*) și în al doilea rînd pentru că structura face adăugări și ștergeri care n-au nicio legătură cu implementarea

canonică de mulțimi disjuncte. Prefer să îi spun **DFS exclusiv**, care descrie mult mai bine cum funcționează tehnica.

Implementarea din acel tutorial mi se pare criptică. Sper că o pot clarifica.

Să ne gîndim la clasa de probleme care fac interogări pe subarbore: frecvențe, sume, numere distincte etc. De exemplu, problema următoare, Tree and Queries, face interogări pe un arbore cu valori în fiecare nod. Interogările au forma (v, k) = numărul de valori distincte care apar de cel puțin k ori în subarborele lui v. Multe din aceste probleme se pot rezolva cu tehnica *small-to-large* deja studiată.

Iată și o rezolvare cu un DFS „pe steroizi". În loc să calculăm cîte o structură în fiecare nod, vom menține o singură structură globală, S. DFS-ul va adăuga și va șterge din S informații despre noduri la diverse momente. El garantează că, pentru orice nod u, va exista un moment în DFS cînd S va conține informații exact despre subarborele lui u. La acel moment vom putea răspunde la interogările despre u.

Desigur, nu putem face un DFS simplu, căci atunci la intrarea într-un nod S va fi deja populată cu date din afara subarborelui său, ceea ce va da răspunsuri incorecte la orice interogări despre u. În loc de aceasta, începem prin a calcula pentru fiecare nod u care este fiul său cu subarborele cel mai mare. Notăm această informație cu $h[u]$ (de la *heavy*). Acum, $DFS(u)$ procedează astfel:

1. Apelează recursiv toți fiii cu excepția lui $h[u]$.
2. La revenirea din fiecare fiu v, elimină recursiv din S toate nodurile din subarborele lui v.
3. Apelează recursiv $DFS(h[u])$. De data aceasta, lasă informațiile în S.
4. Adaugă la loc subarborii eliminați la pasul (2).
5. Adaugă în S nodul u însuși.
6. Răspunde la interogări despre subarborele lui u.

Se nasc două întrebări. Prima: de ce funcționează corect algoritmul? Mai exact, de ce la pasul (6) vectorii conțin doar informații despre subarborele lui u? Să considerăm un alt nod w, din afara subarborelui lui u. Există trei posibilități.

1. w este strămoș al lui u („nod gri"). Atunci, în mod garantat, w nu apare în structura de date la momentul procesării lui u, deoarece w este adăugat în structură doar la finalul recursivității.
2. w este un „nod negru" într-un subarbore deja vizitat. Fie a strămoșul comun al lui u și v și fie a_w și a_u fiii lui a care pornesc către w, respectiv către u. Deoarece sîntem în procesul de vizitare a lui a_u, înseamnă că a_w și tot subarborele său (inclusiv w) au fost temporar șterși din structură.
3. w este un „nod alb", undeva într-un subarbore încă nevizitat. Atunci w încă nu a fost descoperit de DFS.

A doua întrebare: care este complexitatea algoritmului? Dacă nu am trata special nodul $h[u]$, atunci complexitatea ar fi pătratică. Fiecare nod este eliminat la pasul (2) și readăugat la pasul (5) pentru fiecare strămoș al său.

Dacă tratăm special fiii *heavy*, să analizăm numărul de eliminări și readăugări. Ori de cîte ori un strămoș w cauzează eliminarea unui descendent u, înseamnă că w **nu** este fiu *heavy* al părintelui său. Echivalent, părintele lui w este cel puțin de două ori mai mare decît w. Așadar, eliminarea nodului u poate fi cauzată de cel mult $\log n$ dintre strămoșii săi. Complexitatea parcurgerii este $\mathcal{O}(n \log n)$.

Cum fiecare eliminare și adăugare necesită o operație în AIB, rezultă că complexitatea întregului algoritm este $\mathcal{O}(n \log^2 n)$.

Vom citi codul care rezolvă problema următoare.

9.4 Probleme

9.4.1 Problema Tree and Queries (Codeforces)

enunț • surse

Implementare brută

Să vedem mai întîi abordarea cu *small-to-large* clasic. Proiectăm o structură de date care:

1. Să mențină frecvențele culorilor din subarborele curent.
2. Să raporteze numărul de frecvențe cel puțin egale cu o valoare dată.

Pentru (2) am dori un vector de frecvențe ale frecvențelor: $g[x]$ stochează numărul de culori care au frecvența x. Pe acest vector, răspunsul la o interogare (u, x) este suma pe sufixul $g[x \dots n]$. Dar implementarea este alunecoasă, căci dorim ca structura să ocupe spațiu $\mathcal{O}$(subarbore), nu $\mathcal{O}(n)$. Deci orice AIB sau arbore de intervale construit peste g trebuie extins dinamic pe măsură ce urcăm în arbore.

În schimb, putem folosi un multiset: un set ordonat al tuturor frecvențelor (nenule). Ca să putem răspunde la întrebarea „cîte frecvențe mai mari sau egale cu x există?”, avem nevoie de PBDS (*policy-based data structure*), o colecție extinsă de structuri de date din STL. Ne-am mai întîlnit cu seturi PBDS la problema Give Away, în capitolul de descompunere în radical.

Soluția este relativ directă, dar ca să o puteți scrie în timp de concurs trebuie să memorați două lucruri:

1. Incantația magică necesară pentru a declara o structură de date PBDS.
2. Codul necesar pentru ștergerea dintr-un multiset. Cînd o frecvență se modifică, noi dorim să ștergem din multiset o singură apariție a vechii frecvențe, dar un simplu apel la `erase` le-ar șterge pe toate.

Implementare cu DFS exclusiv

Vom menține aceleași informații, dar global:

- un vector de frecvențe;
- un AIB peste vectorul de frecvențe ale frecvențelor.

Notă de implementare: putem stoca listele de adiacență și listele de interogări ca `vector` sau ca `list`, căci nu avem nevoie de acces aleatoriu. Implementarea cu `list` este considerabil mai lentă și consumă mai multă memorie.

Capitolul 10

Cel mai apropiat strămoș comun

Dat fiind un arbore cu rădăcină, pentru orice două noduri u și v definim **cel mai apropiat strămoș comun** ca fiind nodul de adîncime maximă (sau nodul „cel mai jos") care îi are ca descendenți atît pe u, cît și pe v. Considerăm că un nod este propriul său descendent, ceea ce înseamnă că cel mai apropiat strămoș comun al lui u și v poate fi chiar unul dintre aceste noduri, dacă este strămoș al celuilalt.

În literatura română am întîlnit și denumirea „cel mai mic strămoș comun", doar că această denumire mi se pare improprie. Nu comparăm nodurile ca mărime, ci ca adîncime. Pentru abreviere, o vom folosi pe cea din limba engleză (LCA - *lowest common ancestor*).

Formal, dat fiind un arbore cu n noduri, dorim să răspundem la q întrebări de forma:

- $LCA(u, v)$: găsește nodul de adîncime maximă care este strămoș atît al lui u, cît și al lui v.

CP Algorithms inventariază multe dintre metodele disponibile.

O aplicație directă a LCA-ului este aflarea distanțelor între noduri. Notăm cu $d[u]$ adîncimea unui nod u. Atunci

$$dist(u, v) = (d[u] - d[LCA]) + (d[v] - d[LCA]) = d[u] + d[v] - 2 \cdot d[LCA]$$

10.1 Sumar: RMQ (*range minimum query*)

RMQ este o problemă clasică pe vectori. Dat fiind un vector V cu n elemente, trebuie să răspundem la q întrebări de forma $\langle l, r \rangle$ cu semnificația: să se găsească minimul valorilor $V[l \dots r]$.

Problema are diverse variațiuni. Ni se poate cere *valoarea* minimă sau *poziția* valorii minime. Vectorul poate fi static sau poate avea actualizări.

Acest curs nu are un capitol dedicat pentru RMQ, dar o vom trata superficial aici întrucît ea se leagă de algoritmii pentru LCA. Iată un sumar al metodelor folosite în programarea competitivă.

Precizări:

metodă	**preprocesare**	**interogări**	**memorie extra**
Arbore de intervale	$\mathcal{O}(n)$	$\mathcal{O}(q \log n)$	$\mathcal{O}(n)$
Arbore Fenwick (AIB)	$\mathcal{O}(n)$	$\mathcal{O}(q \log n)$	0
Descompunere în radical	$\mathcal{O}(n)$	$\mathcal{O}(q\sqrt{n})$	$\mathcal{O}(\sqrt{n})$
Tabelă rară (*sparse table*)	$\mathcal{O}(n \log n)$	$\mathcal{O}(q)$	$\mathcal{O}(n \log n)$
Stivă ordonată	$\mathcal{O}(q \log q)$	$\mathcal{O}(n + q \log n)$	$\mathcal{O}(n)$

Tabela 10.1: Metode pentru problema RMQ.

- Tabela rară și stiva ordonată nu permit actualizări.
- Arborele Fenwick permite doar interogări pe prefix și doar actualizări descrescătoare.

Să trecem în revistă, doar pentru completitudine, structurile de date.

10.1.1 RMQ cu arbore de intervale

Construim un arbore de intervale în care nodul părinte reține minimul fiilor săi. Admite actualizări punctuale sau, în varianta cu propagare *lazy*, și actualizări pe interval. Am tot studiat aceste structuri de date, nu mai detaliem aici.

10.1.2 RMQ cu arbore indexat binar

Construim un AIB peste vector. Admite doar interogări pe prefix și doar actualizări descrescătoare.

10.1.3 RMQ cu descompunere în radical

Pentru fiecare bloc reținem minimul. Asimptotic operațiile sînt mai lente, dar aș zice că actualizarea pe interval este mai simplu de codat decît la arborii de intervale.

10.1.4 RMQ cu tabelă rară

Elevii se dau în vînt după această metodă. Codul este aproximativ:

```
void compute_rmq(int* v, int n) {
  // r[p][i] = minimul pe intervalul v[i]...v[i + 2^p - 1]
  for (int i = 0; i < n; i++) {
    r[0][i] = v[i];
  }
  for (int p = 1; (1 << p) <= n; p++) {
    for (int i = 0; i + (1 << p) <= n; i++) {
      r[p][i] = min(r[p - 1][i], r[p - 1][i + (1 << (p - 1))]);
    }
  }
}
```

```
int rmq(int left, int right) { // inclusiv
  int p = 31 - __builtin_clz(right - left + 1); // log_2 din lungime
  return min(r[p][left], r[p][right - (1 << p) + 1];
}
```

Totuși, consumul de memorie al metodei *sparse table* este enorm. În afară de cazul în care avem multe interogări, timpul de $\mathcal{O}(1)$ per interogare nu justifică risipa de memorie. Exemplu: Cu *sparse table*, pentru $n = 200\,000$ vom folosi $200\,000 \times 18$ întregi, adică 14,4 MB. Pentru arborele de intervale, presupunînd că îl completăm pînă la 256K elemente, vom folosi 512K întregi, adică 2 MB. Nu am rulat benchmark-uri, dar mă aștept să existe o diferență de viteză din cauza cache-ului.

10.1.5 RMQ cu stivă ordonată

Includ și această metodă ad-hoc. Ordonăm interogările după capătul drept. Parcurgem vectorul de la stînga la dreapta și menținem o stivă crescătoare de minime parțiale. La poziția r putem răspunde la toate interogările $[l, r]$. Răspunsul este valoarea minimă din stivă aflată pe o poziție mai mare sau egală cu l. Așadar este suficientă o căutare binară.

Exemplu: fie vectorul $V = (6, 3, \mathbf{2}, 10, 8, \mathbf{6}, 9, 15, \mathbf{9}, \mathbf{20}, \dots)$. Cînd ajungem la elementul 20, stiva constă din valorile trecute cu aldin. Cînd răspundem la interogările cu capătul r pe elementul 20, răspunsul va fi una dintre valorile din stivă, în funcție de poziția capătului l.

10.2 LCA cu liniarizare

Facem o liniarizare Euler cu repetiție. În vector notăm nodurile, dar ne interesează de fapt adîncimea acelor noduri. Astfel, putem reformula interogările LCA ca „găsește nodul de adîncime minimă dintre ultima apariție a lui u și prima apariție a lui v”. Așadar, toate metodele de RMQ pe vector se aplică și aici.

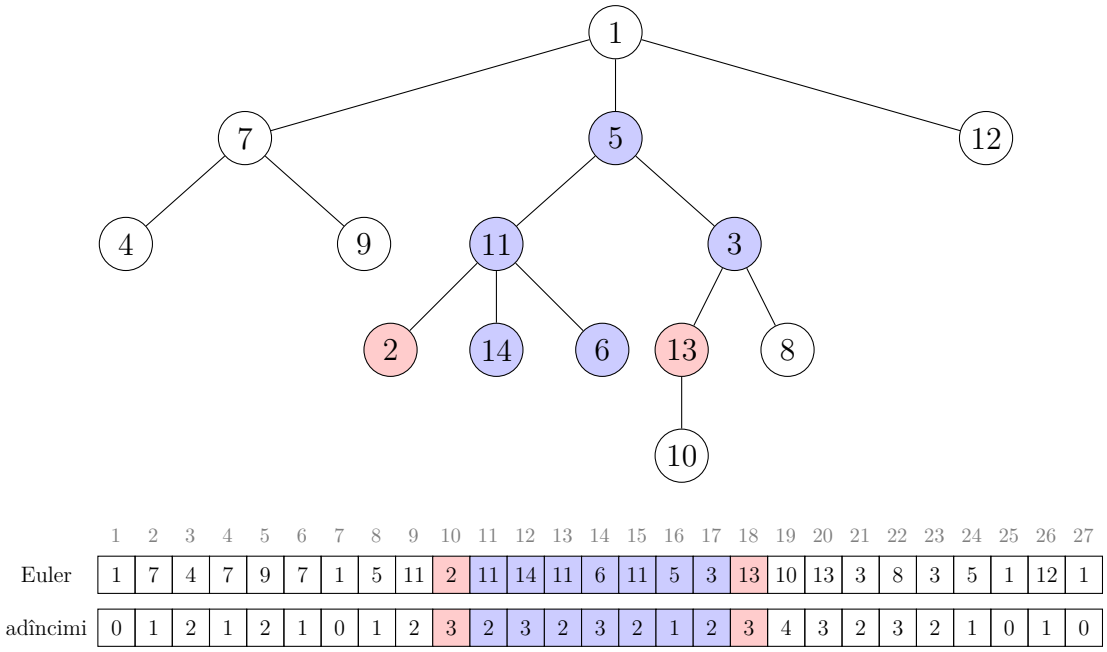


Figura 10.1: Aflarea LCA printr-o liniarizare Euler cu repetiție. LCA(2,13) se reduce la o interogare de minim pe intervalul [10,18]. Adîncimea minimă este 1, corespunzătoare nodului 5.

În continuare, vom mai discuta patru metode specifice arborilor, respectiv:

metodă	**preprocesare**	**interogări**	**memorie extra**
Descompunere în radical	$\mathcal{O}(n)$	$\mathcal{O}(q\sqrt{n})$	$\mathcal{O}(n)$
Binary lifting ($\log n$ pointeri)	$\mathcal{O}(n \log n)$	$\mathcal{O}(q \log n)$	$\mathcal{O}(n \log n)$
Binary lifting (2 pointeri)	$\mathcal{O}(n)$	$\mathcal{O}(q \log n)$	$\mathcal{O}(n)$
Tarjan offline	$\mathcal{O}(q + n \log^* n)$	—	$\mathcal{O}(n)$

Tabela 10.2: Metode pentru aflarea LCA.

10.3 LCA cu descompunere în radical

Următoarele trei metode folosesc noțiunea de *jump pointers*: pointeri la strămoși care ne ajută să accelerăm urcarea din u și din v în căutarea LCA-ului. Pentru descompunerea în radical,

- Fiecare nod ține un pointer la părinte și unul la strămoșul aflat cu $\sqrt{n}$ noduri mai sus.
- Construcția evidentă durează $\mathcal{O}(n\sqrt{n})$: din fiecare nod, urcăm $\sqrt{n}$ niveluri. Ea poate fi redusă la $\mathcal{O}(n)$ dacă menținem stiva DFS pe durata parcurgerii.
- Fiecare interogare durează $\mathcal{O}(\sqrt{n})$.
- Variantă: fiecare nod stochează un pointer la cel mai de jos strămoș de adîncime multiplu de $\sqrt{n}$.

implementare

10.4 LCA cu binary lifting ($\log n$ pointeri per nod)

- Fiecare nod ține pointeri la strămoșii aflați mai sus cu 1, 2, 4, 8... niveluri.
- Construcția durează $\mathcal{O}(n \log n)$, similară algoritmului de RMQ.
- Necesită $\mathcal{O}(n \log n)$ memorie.
- Fiecare interogare durează $\mathcal{O}(\log n)$.
  - Deci nu prea mai merită. Pe vector plăteam memoria $\mathcal{O}(n \log n)$ pentru că interogările durau $\mathcal{O}(1)$.

Detaliu de implementare

Metodele cu *jump pointers* pot fi implementate în două feluri:

1. Aducem nodurile la aceeași adîncime. Apoi urcăm în paralel cu ambele pînă la strămoșul comun.
2. Îl urcăm pe u cît timp nu devine strămoș al lui v. La final, u este LCA-ul.

A doua implementare (cu test de strămoș) este mai scurtă și puțin mai rapidă.

implementări

10.5 LCA cu binary lifting (2 pointeri per nod)

- Un articol bun pe Codeforces (imaginile sînt foarte utile).
- Fiecare nod x ține un pointer la părinte și un pointer numit jump, construit după regula:
- Fie y părintele lui x, fie $z = jump[y]$ și fie $t = jump[z]$.
- Dacă distanțele între y-z și z-t sînt egale, atunci $jump[x] = t$.
- Altfel $jump[x] = y$.
- Iau naștere niște pointeri cu o structură curioasă. Cei pasionați pot citi despre secvență pe OEIS, secvențele A082850 și A182105.
- Folosim această informație pentru a găsi LCA în $\mathcal{O}(\log n)$.
- Orice structură de pointeri „aproximativ" logaritmică funcționează aici. Am făcut și un experiment cu formula LSB (exemplu: un nod la adîncime $20 = 10100_2$ va pointa patru nivele mai sus).

implementări

10.6 LCA cu algoritmul lui Tarjan (offline)

- Funcționează cînd interogările sînt date în avans.
- Distribuie interogările după noduri. Interogarea (u, v) este distribuită în listele lui u și v.
- Răspunde la interogări într-un singur DFS (!).
- Principiu de bază: grupăm nodurile deja vizitate („negre") și pe cele în curs de vizitare („gri") în mulțimi disjuncte (cu *union-find*).

- Fiecare mulțime va conține exact un nod gri, plus toți descendenții săi, cu excepția nodului gri în care se află acum DFS-ul. La terminarea unui nod gri, el este unit cu părintele său.
- Putem răspunde la o interogare (u, v) în momentul în care v este negru, iar pe u tocmai îl vizităm. Răspunsul (LCA) este nodul gri din mulțimea lui v.

implementare

10.7 Benchmarks

Am făcut aceste teste pe un arbore cu 200.000 de noduri, cu un lanț de cel puțin 150.000 de noduri. Fișierele de intrare au 5,2 MB.

- Tarjan: 160 ms
- *binary lifting*, doi pointeri (cu test de strămoș sau nu): 190 ms
- *binary lifting*, doi pointeri (metoda LSB): 250 ms
- *binary lifting*, $\log n$ pointeri, cu test de strămoș: 290 ms
- *binary lifting*, $\log n$ pointeri: 320 ms
- descompunere în radical: 900 ms

Concluzii: cînd interogările sînt date în avans, Tarjan cîștigă. În rest, metodele cu *binary lifting* cu doar doi pointeri per nod sînt mai rapide și consumă mai puțină memorie decît metoda cu $\log n$ pointeri per nod.

10.8 Probleme

10.8.1 Problema Gold Transfer (Codeforces)

enunț • sursă

Atenție mare la garanția din enunț: $c_i > c_{p_i}$. Cu alte cuvinte, pentru orice operație de cumpărare trebuie să pornim din rădăcină spre nodul u și să cumpărăm orice cantități disponibile, pînă satisfacem cererea.

Cu timpul, nodurile de la rădăcină se vor goli, deci pare o idee bună să găsim rapid cel mai de jos strămoș care încă are aur disponibil. Dacă reușim să-l găsim în $\mathcal{O}(f(n))$ (intenția problemei fiind $f(n) = \log n$), atunci complexitatea globală va fi $\mathcal{O}(q \cdot f(n) + n)$. De ce? Din acel strămoș putem parcurge lanțul în jos pas cu pas, cumpărînd tot aurul disponibil, pînă satisfacem cererea. Fiecare nod poate fi golit cel mult o dată, iar ultimul nod din fiecare interogare poate păstra niște aur. De aceea, efortul total pentru parcurgerea lanțurilor este $\mathcal{O}(n + q)$.

Detalii de implementare

Arborele este dinamic, deci nu putem construi o liniarizare.

Am ales să accelerez urcarea în arbore cu *binary lifting* cu doi pointeri, dar orice altă metodă este acceptabilă.

Odată ce golim un nod, avem nevoie să coborîm în fiul său care duce spre nodul original. Am ales să fac acest lucru cu o stivă, dar soluția este cam lentă (2x față de altele).

Întrucît nu avem de ce să parcurgem toți fiii unui nod, nu avem nevoie de liste de adiacență, ci doar de pointeri la părinte.

10.8.2 Problema A and B and Lecture Rooms (Codeforces)

enunț • sursă

Problema cere să răspundem la m întrebări de forma: Date fiind nodurile u și v (posibil egale), cîte noduri din arbore se află la distanță egală de u și de v?

Pentru soluția teoretică, următoarea vizualizare este utilă. Să „atîrnăm" arborele de nodurile u și v, ca pe o ghirlandă bine întinsă. Atunci calea cea mai scurtă $u - v$ va fi orizontală, iar de lanțurile de pe cale vor atîrna restul subarborilor.

Dacă distanța $u - v$ este impară, atunci răspunsul este 0. Dacă distanța este pară, atunci fie w nodul de la jumătatea distanței. Nodurile aflate la distanță egală de u și de v vor fi w și orice nod din subarborii care atîrnă din w.

În practică vom alege o rădăcină și vom precalcula informații pentru LCA. Apoi, eu am redus problema la următoarele cazuri (poate se poate și mai simplu):

1. Dacă distanța $u - v$ este impară, răspunsul este 0.
2. Dacă $u = v$, răspunsul este n.
3. Dacă u și v au adîncimi egale, atunci răspunsul este: mărimea subarborelui lui $LCA(u, v)$ minus mărimile subarborilor fiilor lui $LCA(u, v)$ care pornesc către u, respectiv către v.
4. Altfel, să presupunem că u are adîncime mai mare decît v. Atunci nodul w este undeva mai jos de LCA, mergînd către u. Răspunsul este: mărimea subarborelui lui w minus mărimea subarborelui fiului lui w care pornește către u.

Detalii de implementare

Punctul (4) presupune să calculăm al k-lea strămoș. De exemplu, dacă adîncimile sînt $d[u] = 50$, $d[v] = 20$ și $d[LCA] = 10$, atunci distanța $u - v$ este $40 + 10 = 50$, jumătatea distanței este 25, iar nodul w este al 25-lea strămoș al lui u.

De aceea, metoda lui Tarjan nu ne prea ajută, ci avem nevoie de o metodă cu *jump pointers*.

La punctele (3) și (4), pentru a afla „fiul lui w care pornește către u", putem refolosi informațiile pentru al k-lea strămoș. Dacă notăm cu k diferența pe înălțime între w și u, atunci răspunsul este al $k - 1$-lea strămoș al lui u.

La arbori, multe informații sînt redundante și putem stoca doar o submulțime. Exemple:

- Mărimea subarborelui lui u nu trebuie stocată implicit dacă cunoaștem timpii DFS, ci poate fi calculată ca $t_o[u] - t_i[u] + 1$.
- Paritatea distanței poate fi calculată fără a calcula distanța efectivă. Calculăm doar suma adîncimilor.
- Al k-lea strămoș al unui nod poate fi găsit ca strămoșul de la adîncimea $d[u] - k$ al unui nod, dacă cunoaștem adîncimile nodurilor.

10.8.3 Problema Company (Codeforces)

enunț • sursă

Problema poate fi rezumată astfel: dintre toate nodurile cu numere de la l la r, cum putem elimina un nod astfel încît LCA-ul celor rămase să aibă o adîncime cît mai mare, raportată la rădăcină?

Întrebarea teoretică la care trebuie să răspundem este: care este LCA-ul unei submulțimi de noduri? Observăm că nu este nevoie să iterăm prin toate, ci este suficient să calculăm LCA-ul între primul și ultimul nod în ordinea descoperirii lor în DFS. De aceea, ca să încercăm să coborîm LCA-ul, trebuie să eliminăm unul dintre aceste noduri. Altfel LCA-ul submulțimii după eliminarea unui nod va rămîne nemodificat față de LCA-ul original.

Să presupunem că găsim aceste noduri, fie ele x și y. Încercăm să îl eliminăm pe x și să calculăm LCA-ul submulțimii $[l, x-1] \cup [x+1, r]$. Apoi îl eliminăm pe y și calculăm LCA-ul submulțimii $[l, y-1] \cup [y+1, r]$. Afișăm varianta care duce la un LCA de adîncime maximă.

Detalii de implementare

Ca să găsim timpii DFS minim/maxim ai nodurilor din intervalul $[l, r]$, putem construi un vector $t[]$ unde $t[u]$ este timpul vizitării nodului u. Astfel reducem subproblema la una de RMQ. Pentru a deduce, din acești timpi, nodul efectiv (primul nod vizitat dintre toate din $[l, r]$) este suficient un vector $inv[]$ unde $inv[i]$ este nodul vizitat la momentul i. Practic t și inv sînt permutări inverse.

Pentru intervalele la care ajungem după eliminarea lui x și y (respectiv $[l, x-1]$, $[x+1, r]$ și celelalte) avem nevoie să aflăm LCA-ul, deci avem nevoie tot de RMQ ca să aflăm primul și ultimul nod în ordinea DFS. Dar am făcut următoarea observație care reduce mult numărul de interogări.

Fie a, b, c, d primul, al doilea, penultimul și respectiv ultimul nod din intervalul $[l, r]$, în ordinea DFS. Atunci iau naștere două cazuri:

- Dacă îl eliminăm pe a, LCA-ul nodurilor rămase este $LCA(b, d)$.
- Dacă îl eliminăm pe d, LCA-ul nodurilor rămase este $LCA(a, c)$.

De aceea, am proiectat o structură de date care să returneze, pentru o interogare $[l, r]$, primele două minime și primele două maxime din intervalul $[l, r]$. Practic structura returnează exact cele patru valori a, b, c, d.

Putem folosi RMQ cu sparse table, cu atenție la detalii la calculul celui de-al doilea maxim/minim (intervalele de lungime putere a lui 2 pot fi suprapuse). Dar nu-mi place ideea de a consuma memorie $\mathcal{O}(n \log n)$ fără rost. 😐 De aceea, am implementat structura ca pe un arbore de intervale. Operația critică este combinarea a două tupluri (a', b', c', d') și (a'', b'', c'', d''). Implementarea este relativ ușoară:

- Dacă $a' < a''$ atunci primul minim este a', iar al doilea minim este $\min(b', a'')$.
- Dacă $a' \geq a''$ atunci primul minim este a'', iar al doilea minim este $\min(b'', a')$.
- Similar pentru maxime.

Sursa mea este printre cele mai rapide, cu excepția celor care parsează intrarea. Suspectez că ajută mult (1) economia de memorie și (2) calculul simultan al minimelor și al maximelor, în aceeași structură de date.

10.8.4 Problema Duff in the Army (Codeforces)

enunț • sursă • coloană sonoră 🙂

Rezumat: Se dă un arbore cu n noduri și m locuitori. Locuitorul i locuiește în orașul c_i. Răspundeți la q interogări de forma (u, v, a) cu semnificația: tipăriți primii a locuitori, ordonați după ID, care locuiesc pe calea (u, v). Notă: $a \leq 10$.

Problema se duce tot în direcția aflării LCA. Dacă $LCA(u, v) = w$, și dacă aflăm populațiile pe căile (u, w) și (v, w), putem să le interclasăm în $\mathcal{O}(a)$ și să păstrăm cele mai mici a valori.

Cum aflăm populația pe o cale? Nu văd o soluție în $\mathcal{O}(a + \log n)$, dar $\mathcal{O}(a \log n)$ este facilă. Orice algoritm de LCA ne oferă această complexitate dacă precalculăm, împreună cu pointerii în sus, și populația acoperită de acei pointeri. De exemplu, cu metoda *binary lifting*, fiecare din pointerii peste 1, 2, 4, 8, ... niveluri reține și primii a membri ai populației de pe acea cale. Populația pe calea de lungime 16 se obține interclasînd cele două populații pe căile de lungime 8 și păstrînd cel mult 10 minime.

Atenție la metoda folosită! *Binary lifting* cu memorie $\mathcal{O}(n \log n)$ va necesita $100\,000 \times 18 \times 10$ întregi, adică 72 MB în plus. Este fezabil, dar este de evitat. Implementarea mea este printre cele mai rapide și nu face nimic deosebit, doar folosește *binary lifting* cu doi pointeri. Economia de memorie înseamnă economie de timp.

Ne întîlnim, din nou, cu o situație care cere să particularizați structurile de date, nu doar să le pictați. Un alt exemplu de astfel de problemă este: să se preproceseze un arbore astfel încît să putem răspunde eficient la interogări de forma $(u, v) =$ maximul pe calea $u - v$. Și aici putem face fiecare *jump pointer* să rețină maximul pe calea subîntinsă. Problema are și soluții mai eficiente, dar aceasta este relativ elementară.

Capitolul 11

Algoritmul lui Mo pe arbore

11.1 Limitele liniarizărilor

Ca o scurtă recapitulare, liniarizările ne ajută:

- pentru interogări pe subarbore: liniarizarea DFS atribuie fiecărui subarbore un interval compact din vector;
- pentru interogări pe calea de la un nod la rădăcină: liniarizarea Euler + vectorii de diferențe atribuie fiecărei căi un prefix din vector.

Uneori știm să procesăm și interogări pe căi arbitrare, vezi problema Max Flow. Dar algoritmii funcționează exprimînd calea (u, v) ca pe o combinație de căi (u, r), (v, r) și (l, r), unde r este rădăcina arborelui, iar $l = LCA(u, v)$. Așadar, avem nevoie ca funcțiile pe fiecare cale să fie **inversabile** (sume, xor-uri). În probleme de minim/maxim, valori distincte etc., ce știm pînă acum este insuficient.

11.2 Reducerea la algoritmul lui Mo

Putem folosi algoritmul lui Mo pentru a răspunde la interogări pe căi în unele cazuri neinversabile. Găsiți un tutorial destul de bun pe Codeforces, pe care îl vom relua aici. În esență,

1. Construim o liniarizare de tip Euler (două apariții pentru fiecare nod).
2. Transformăm interogările pe căi (u, v) în interogări pe intervale în liniarizare.
3. Sortăm interogările și le aflăm răspunsurile cu algoritmul lui Mo.

Desigur, misterul este la pasul 2. De aceea, să considerăm...

11.3 Un exemplu

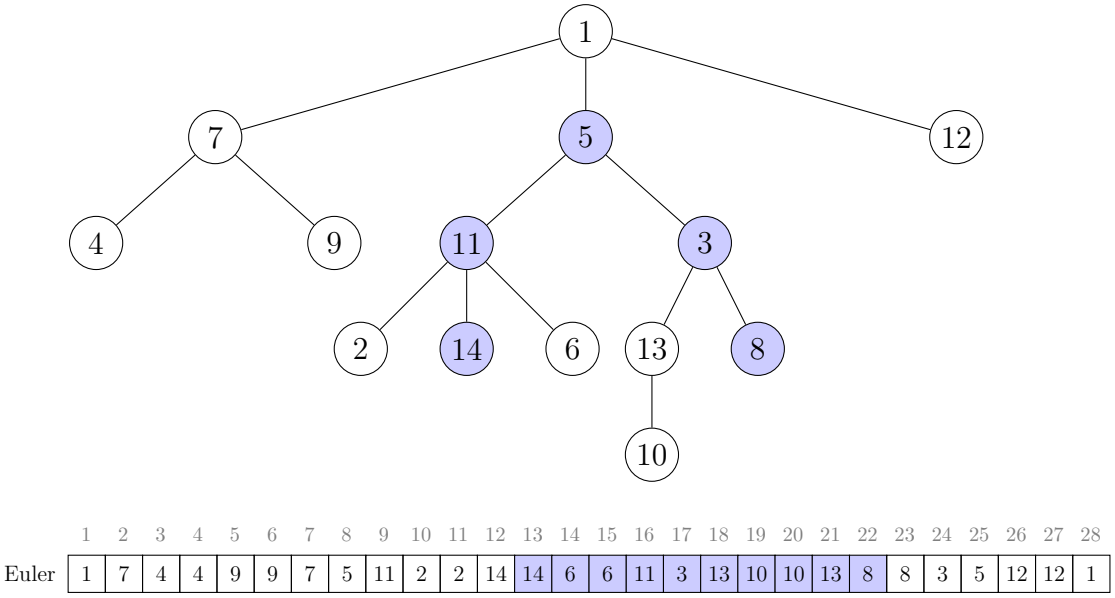


Figura 11.1: Un arbore cu liniarizarea Euler și intervalul corespunzător interogării pe calea $(14, 8)$.

Să considerăm interogarea $(14, 8)$, privitoare la nodurile 14, 11, 5, 3, 8. În liniarizarea Euler, ne interesează intervalul dintre timpii 13 și 22. Am ales acest interval deoarece el se întinde de la **ultima** apariție a lui 14 pînă la **prima** apariție a lui 8. Să facem niște observații privitoare la acest interval.

1. Nodurile 14, 11, 3 și 8 apar exact o dată.
2. Nodul $5 = LCA(14, 8)$ nu apare.
3. Nodurile 6, 13 și 10 apar de două ori.
4. Celelalte noduri (1, 2 etc.) nu apar.

Ne putem convinge ușor că aceste observații sînt generale. Trebuie doar să urmărim evoluția DFS-ului. De exemplu, observația (1): pornim de la momentul cînd DFS-ul părăsește nodul 14, deci va părăsi în curînd și nodul 11. Apoi explorează nodurile 3 și 8, dar nu apucă să le părăsească în intervalul ales. De aceea, toate aceste noduri apar exact o dată.

11.4 Un caz particular

Mai trebuie să tratăm și cazul cînd, pentru o interogare (u, v), avem $LCA(u, v) = u$, cu alte cuvinte u este strămoș al lui v.

Clarificare: mereu vom ordona perechea (u, v) în ordinea descoperirii în DFS. De aceea LCA-ul poate fi doar u, niciodată v. Aceasta deoarece un nod este descoperit înaintea tuturor descendenților săi.

Pentru interogarea (5,14), privitoare la nodurile 5, 11, și 14, vom considera intervalul de timp [8,

12], cuprins între primele apariții ale lui 5 și 14. Se schimbă doar observația (2): LCA-ul apare și el exact o dată.

11.5 Descrierea completă

Pentru o interogare (u, v) cu $t_i[u] < t_i[v]$:

1. Dacă u este strămoș al lui v, atunci considerăm intervalul din liniarizare $[t_i[u], t_i[v]]$. Nodurile de pe cale sînt cele care apar exact o dată în acest interval.
2. Dacă u **nu** este strămoș al lui v, atunci considerăm intervalul $[t_o[u], t_i[v]]$. Nodurile de pe cale sînt cele care apar exact o dată, plus nodul $LCA(u, v)$.

11.6 Structura de date necesară

Acum putem specifica ultimul amănunt: ce anume stochează structura de date pentru intervalul curent? Desigur, depinde de problemă, dar un mecanism este general.

Structura trebuie să țină minte ce noduri apar în ea exact o dată. De exemplu, pentru subsecvența (5 11 2 2 14), structura trebuie să știe că nodurile 5, 11 și 14 apar exact o dată. Dacă extindem spre dreapta secvența și încorporăm încă un 14, informația despre nodul 14 trebuie **ștearsă** din structură, căci 14 nu mai este pe cale.

Pe măsură ce extindem și contractăm intervalul cu algoritmul lui Mo, la anumite momente structura va reține date pentru intervale care nu corespund unor căi. Dar asta este OK cîtă vreme, în momentul unei interogări, structura este coerentă.

11.7 Probleme

11.7.1 Problema Dating (Codeforces)

enunț • surse

Dacă doriți probleme suplimentare, puteți încerca:

- ușoară: Count on a Tree II (SPOJ)
- grea: So Close Yet So Far (CodeChef)

Problema se încadrează destul de clar în situația sus-menționată. Interogările sînt pe cale și nu sînt ușor de combinat din bucăți. Dacă notăm cu $l = LCA(u, v)$, nu este evident cum am putea calcula, apoi însuma, valorile pe căile (u, l) și (l, v).

În schimb, dacă ne vine ideea să încercăm algoritmul lui Mo pe arbore, soluția este facilă. Structura curentă menține

- un boolean pentru fiecare nod, ca să știm ce noduri se află în structură;
- frecvența numerelor favorite, separat pentru fete și pentru băieți.

Includerea / excluderea unui nod este ușoară. Pe parcurs, menținem în permanență răspunsul (numărul de perechi pe care le putem forma).

Anexa include tot codul, dar esența este:

```
struct mo_tracker {
  bool on[MAX_NODES + 1];  // nodurile care apar exact o dată în interval
  int f[MAX_NODES + 1][2]; // frecvența numerelor favorite pentru fiecare gen
  int l, r;
  unsigned num_pairs;

  void init() {
    l = 1;
    r = 0;
  }

  void toggle(int pos) {
    int u = euler[pos];
    on[u] = !on[u];
    int sign = on[u] ? +1 : -1;
    f[nd[u].fav][nd[u].gender] += sign;
    num_pairs += sign * f[nd[u].fav][!nd[u].gender];
  }

  unsigned query(int target_l, int target_r, int extra_fav, bool extra_gender) {
    while (l > target_l) {
      toggle(--l);
    }
    while (r < target_r) {
      toggle(++r);
    }
    while (l < target_l) {
      toggle(l++);
    }
    while (r > target_r) {
      toggle(r--);
    }

    if (extra_fav) {
      return num_pairs + f[extra_fav][!extra_gender];
    } else {
      return num_pairs;
    }
  }
};
```

Comparația implementărilor

Implementarea pe care o consideram mai eficientă este cea cu Tarjan pentru LCA și cu liste proprii. Dar ea este doar cu 10% mai rapidă decît implementarea mai scurtă, cu metoda celor doi pointeri pentru LCA și cu vectori STL. Fie Codeforces este foarte neconsecvent, fie eu nu mai înțeleg lumea. 🥲

Capitolul 12

Descompunere *heavy-light*

Descompunerea *heavy-light* (engl. *heavy-light decomposition* sau HLD, numită și *heavy path decomposition*) este încă o tehnică de liniarizare a arborilor, utilă pentru interogări pe căi.

HLD costă un factor suplimentar de $\mathcal{O}(\log n)$, așadar genul de întrebări la care răspundem în $\mathcal{O}(\log n)$ pe vector ne vor costa $\mathcal{O}(\log^2 n)$ pe arbore.

Înainte de a studia HLD, subliniez că este o unealtă puternică, dar greu de codat și lentă. Înainte de a vă repezi la ea, întrebați-vă dacă nu există o soluție mai simplă, adaptată nevoilor problemei. HLD intră doar în materia de lot (nu de baraj).

12.1 Limitările structurilor anterioare

Să pornim de la următoarea cerință. Se dă un arbore cu n noduri. Fiecare nod are o valoare. Trebuie să procesăm q operații de două tipuri:

1. `update(v, x)`: Valoarea nodului v devine x.
2. `max(u, v)`: Găsește valoarea maximă de pe lanțul $u - v$.

Să trecem în revistă uneltele pe care le-am studiat pînă acum.

- Un simplu DFS nu pare că poate propaga suficiente informații.
- Liniarizarea DFS se pretează la interogări pe subarbore, nu pe cale.
- Liniarizarea Euler se pretează la interogări pe calea de la un nod u la rădăcină. Pentru interogări de maxime, nu putem descompune căi arbitrare în diferențe de căi pînă la rădăcină.
- Tehnica *small-to large* nu ne ajută aici.
- LCA + *binary lifting* rezolvă doar problema fără actualizări. Fiecare pointer notează și maximul nodurilor peste care trece. Dar la actualizare, dacă avem un nod cu $\mathcal{O}(n)$ fii, vom fi nevoiți să actualizăm informația din $\mathcal{O}(n)$ pointeri.
- Similar și pentru orice tentativă de descompunere în radical.
- Algoritmul lui Mo pe arbore funcționează, dar lent. Structura pentru intervalul curent

trebuie să admită inserarea și ștergerea de valori și interogarea de maxim. Complexitatea va fi $\mathcal{O}(q\sqrt{n}\log n)$.

12.2 Descompunerea

Știm deja că liniarizarea DFS garantează că orice subarbore corespunde unui interval compact. Dar dorim mai mult de atît. Să examinăm Figura 12.1

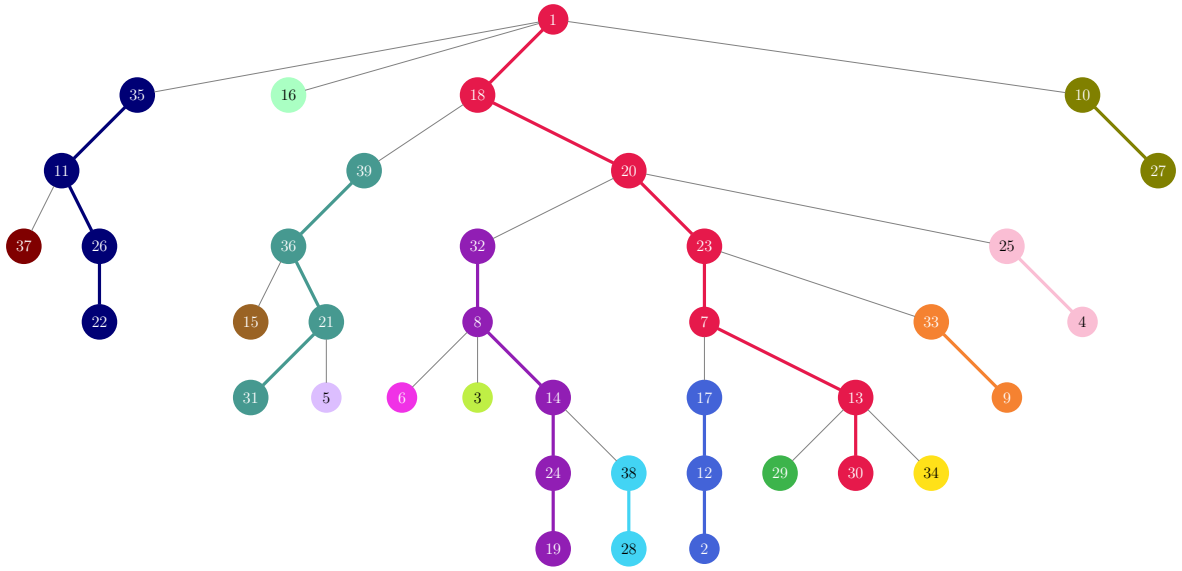


Figura 12.1: Un arbore în care muchia grea a fiecărui nod este îngroșată și colorată. Fiecare culoare indică un lanț de muchii grele consecutive.

Pentru fiecare nod intern u din arbore, identificăm fiul v cu subarborele maxim (cu cele mai multe noduri). Numim nodul v **fiu greu** (engl. *heavy*) al lui u, iar muchia (u, v) **muchie grea**. Ceilalți fii ai lui u și celelalte muchii care coboară din u se numesc **fii ușori** (engl. *light*), respectiv **muchii ușoare**. De exemplu, fiul greu al rădăcinii 1 este 18. Figura 1 indică muchiile ușoare cu linii subțiri, iar muchiile grele cu linii groase și colorate. Dacă un nod are mai mulți fii cu același număr maxim de noduri în subarbore, îl putem alege pe oricare ca greu.

Observăm că, dacă pornim din orice nod, putem urma muchia grea pînă la o frunză. Astfel iau naștere **lanțuri grele**. Am colorat muchiile grele din același lanț folosind aceeași culoare.

Mai facem un pas esențial: reorganizăm lista de adiacență a fiecărui nod ca să mutăm fiul greu primul (vom vedea cum implementăm asta în practică). Este important că această modificare nu schimbă răspunsurile la problemele tipice de arbori. Ordinea fiilor nu contează. Pentru arborele din Figura 12.1, versiunea reorganizată apare în Figura 12.2.

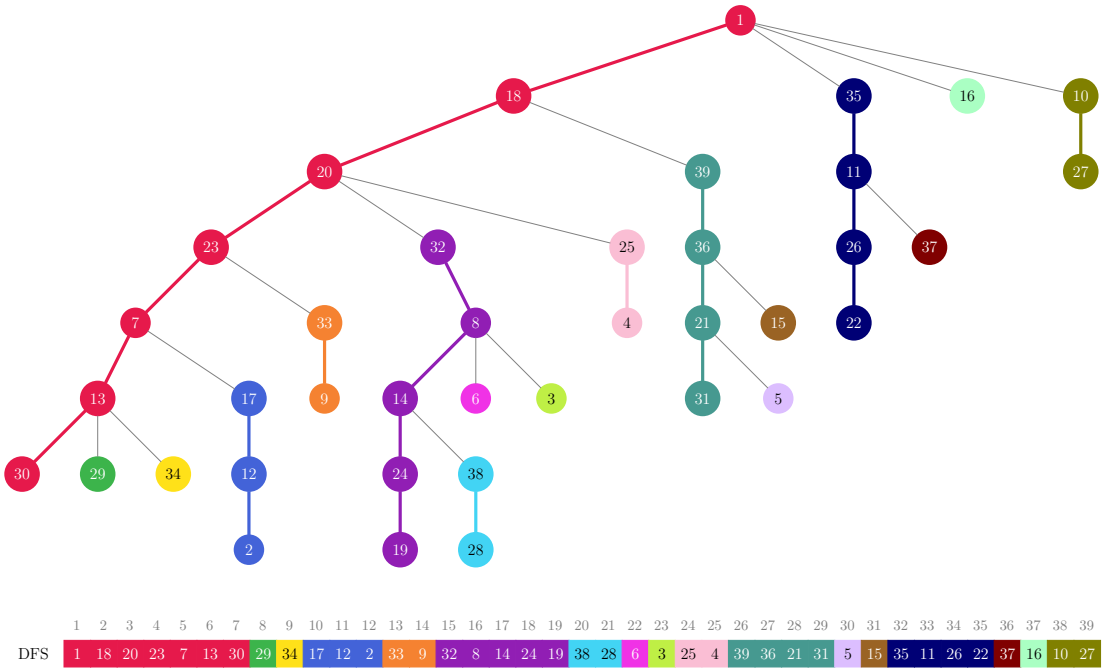


Figura 12.2: Același arbore în care, în fiecare listă de adiacență, am mutat fiul greu primul. Lanțurile corespund la intervale contigue în liniarizarea DFS.

Despre acest arbore putem da două garanții foarte puternice.

1. Lanțurile formate din muchii grele corespund la intervale contigue în liniarizare. Într-adevăr, fiecare fiu greu este primul nod pe care îl vizitează părintele său. De exemplu, la intrarea în nodul 32, următoarele noduri vizitate vor fi 8, 14, 24 și 19.
2. Calea de la orice nod la rădăcină vizitează, total sau parțial, cel mult $\log n$ lanțuri. Echivalent, calea conține cel mult $\log n$ muchii ușoare. Demonstrația este identică cu cea de la metoda small to large: când urcăm dintr-un fiu v în părintele u aflat pe alt lanț, prin definiție traversăm o muchie ușoară. Deci v este fiu ușor al lui u, ceea ce înseamnă că v are un frate greu h. Subarborele lui h este cel puțin la fel de mare ca al lui v, deci subarborele lui u este cel puțin dublu față de subarborele lui v.

Proprietatea (1) ne spune că putem trata lanțurile ca pe niște vectori. În particular, putem construi peste ele structuri de date ca arbori Fenwick sau arbori de intervale, care ne dau informații despre porțiuni de lanțuri în $\mathcal{O}(\log n)$. Proprietatea (2) ne spune că orice cale $u - v$ vizitează $\mathcal{O}(\log n)$ lanțuri. Avem, așadar, o metodă generală de a răspunde la interogări pe căi în $\mathcal{O}(\log^2 n)$.

12.3 Detalii de implementare

Facem întîi un DFS pentru calcularea mărimii subarborilor și a fiilor grei. Nu modificăm listele de adiacență, ci doar calculăm un cîmp `heavy` pe fiecare nod. Apoi facem un al doilea DFS, care liniarizează arborele, apelînd întîi fiul greu, apoi pe ceilalți.

Nu construim cîte o structură (AIB, AINT etc.) pe fiecare lanț! Ca la orice liniarizare, construim

o singură structură peste toate cele n noduri. Este datoria programului să nu acceseze intervale care „încalecă” mai multe lanțuri.

Ce structură folosim depinde de natura problemei. Dacă problema cere informații pe căi de la noduri la rădăcină, atunci căile vizitează doar prefixe (capetele de sus) ale lanțurilor. Deci s-ar putea ca un AIB să fie suficient. Dacă problema cere informații între orice două noduri, atunci căile pot vizita și porțiuni din mijlocul lanțurilor. Deci probabil avem nevoie de arbori de segmente.

Nu este necesar să implementăm LCA. În loc de asta, fiecare nod menține un pointer la capătul superior al lanțului. Astfel urcăm „naiv” din lanț în lanț, căci știm deja că numărul de lanțuri este logaritmic.

12.4 Probleme

12.4.1 Problema Heavy Path Decomposition (Infoarena)

enunț • surse

Aceasta este exact problema studiată la teorie. Vom citi codul.

Prima versiune este cea „canonică”. A doua versiune face o optimizare minoră care elimină nevoia calculării înălțimii nodurilor. Ea pornește de la următoarea observație. Dorim ca, odată ce u ajunge pe lanțul comun, să se oprească din urcat și să-l aștepte și pe v (pe acel lanț comun vom face ultima interogare). Dar, dacă u a ajuns pe lanțul comun, iar v încă nu, atunci în mod necesar $t_{in}[u] < t_{in}[v]$, căci toate nodurile de pe acel lanț sînt vizitate primele în DFS, fiind noduri heavy. Așadar, este suficient să comparăm timpii DFS ai lui u și v.

Subliniem că în funcția `query` parcurgem mereu lanțul al cărui capăt de sus are adîncime mai mare. De ce nu comparăm pur și simplu înălțimile lui u și v? Dați un contraexemplu!

O problemă echivalentă este Path Queries II (CSES).

12.4.2 Problema Disruption (USACO)

enunț • surse

Vom discuta trei soluții, de amorul artei.

Soluția cu HLD

Să considerăm o cale de înlocuire (u, v). Căror muchii le poate ea suplini dispariția? Doar celor de pe calea $u - v$ din arbore. Așadar, o soluție teoretică este: pentru fiecare cale de înlocuire (u, v) de cost c, notifică toate muchiile de pe calea $u - v$ că au la dispoziție o înlocuire de cost c. După procesarea tuturor căilor, pentru fiecare muchie tipărește costul minim despre care a fost notificată, sau -1 dacă muchia nu a primit notificări.

Concret, ce înseamnă aceste notificări? Fie $l = LCA(u, v)$. Atunci vrem să marcăm costul c pe căile $u - l$ și $v - l$. De aici ne poate veni ideea descompunerii *heavy-light*, care garantează că o cale de înlocuire va genera cel mult $\mathcal{O}(\log n)$ intervale de actualizat.

Ce structură folosim? Pe fiecare lanț avem nevoie de operațiile:

1. `range_set(l, r, c)`: pe fiecare poziție $l \leq i \leq r$, atribuie $v[i] = \min(v[i], c)$.
2. `query(i)`: returnează $v[i]$.

Desigur, `range_set` va descompune $[l, r]$ în niște intervale și va scrie valoarea c pe acele intervale. Cum procedăm după aceasta? Nu vă repeziți la implementarea cu propagare *lazy*! Interogările sînt punctuale, nu pe interval, deci este suficient să vizităm toți strămoșii unui nod (din aint) și să returnăm minimul valorilor găsite. Mai mult, toate interogările vor veni după toate actualizările. Deci putem face o singură propagare top-down la sfîrșitul actualizărilor. Atunci în fiecare frunză din aint vom avea exact răspunsul dorit.

Mai sînt două mici goluri de umplut:

1. Arborele de intervale trebuie inițializat cu ∞.
2. Conceptual, problema cere informații despre muchii. Dar, datorită DFS-ului, fiecare muchie unește un fiu de părintele său. Vom stoca informațiile în fiu.

Soluția cu *small-to-large*

Soluția oficială pornește de la altă observație teoretică. Orice muchie (u, v) poate fi înlocuită de o cale de înlocuire (x, y) cu proprietatea că x și y se află de părți diferite ale muchiei (u, v). Formulată în termeni de DFS, condiția este: muchia de la orice nod u la părintele său poate fi înlocuită de orice cale de înlocuire (x, y) care are **exact** un capăt în subarborele lui u. Să denumim o astfel de cale de înlocuire **viabilă pentru** u.

Așadar, dorim ca fiecare nod u să-și calculeze mulțimea de căi viabile și să o raporteze pe cea de cost minim. Atunci mulțimea unui nod u provine din:

1. reuniunea mulțimilor fiilor,
2. plus căile care au un capăt chiar în u,
3. minus căile care apar de două ori în (1) și (2), deoarece o cale care începe și se termină în subarborele lui u nu este viabilă pentru u.

Putem folosi tehnica *small-to-large* pentru a garanta că fiecare element este transferat între mulțimi de cel mult $\log m$ ori. Mulțimile însăși trebuie să poată raporta valoarea minimă, deci vor avea un cost logaritmic (de exemplu, cu `std::set`). Complexitatea totală este $\mathcal{O}(n \log m + m \log^2 m)$.

Puteți găsi o implementare foarte concisă (dar cam lentă) aici. Ea stochează în seturi perechi $(cost, id)$: costul unei căi servește la găsirea costului minim, iar ID-ul servește la găsirea duplicatelor, pentru eliminarea lor. Eu am ales o modalitate diferită: am sortat căile de înlocuire după cost, astfel încît ID-urile mai mici să corespundă la costuri mai mici.

Iată încă un detaliu de folosire a `set`-urilor. O implementare naivă va insera sau șterge căi cu următorul cod:

```
if (s.count(id)) {
  s.erase(id);
} else {
  s.insert(id);
}
```

Totuși, acest cod face două căutări per operație: una pentru găsirea elementului și a doua pentru inserare sau ștergere, după caz. Dar se poate și cu o singură căutare! Funcția insert, dacă eșuează, returnează un iterator la elementul care a cauzat eșecul.

```
std::pair<set::iterator, bool> p = s.insert(id);
if (!p.second) {
  // Inserarea a eșuat, iar p.first pointează chiar la elementul-duplicat.
  s.erase(p.first);
}
```

Soluția cu DFS exclusiv

Dacă vă amintiți, la problema Tree and Queries am discutat un DFS special care folosește o singură structură de date globală. DFS-ul garantează că fiecare nod u va avea acces, la un moment dat în timp, la structura de date care conține informații doar despre subarborele lui u.

Putem folosi acest algoritm și aici. Avem nevoie să includem/excludem noduri din structură, ceea ce presupune să includem/excludem căile care au un capăt în acele noduri. Așadar, avem nevoie de o structură de date cu operațiile:

1. Activează/dezactivează o poziție.
2. Returnează poziția minimă activă (sau o valoare specială dacă nu există poziții active).

Un AIB este suficient, cu căutare binară pentru (2). Ce complexitate are această soluție?

Benchmarks

Testele sînt mici, deci nu prea concludente, dar iată rezultatele:

- Cu descompunere *heavy-light*: 30 ms, 7 MB.
- Cu *small-to-large*: 74 ms, 18 MB.
- Cu *small-to-large* exclusiv: 55 ms, 12 MB.

12.4.3 Problema Rafaela (Lot 2014)

enunț • surse

Ca observație preliminară, interogările pot fi reformulate astfel: dacă aș înrădăcina arborele în nodul u, care este populația maximă a unui subarbore al rădăcinii?

În practică vom fixa rădăcina în nodul 1. Fie $S(u)$ populația subarborelui nodului u. Iau naștere două cazuri:

1. Răspunsul la interogare este $S(1) - S(u)$ dacă populația maximă sosește pe muchia de la u la părinte. Desigur, $S(1)$ este populația întregului arbore, care este trivial de întreținut.
2. Răspunsul la interogare este $\max S(v)$, unde v este un fiu al lui u.

Soluție doar cu arbore de intervale

Pare natural să încercăm să menținem $S(u)$ pentru toate nodurile. Cînd populația lui u se modifică cu Δ, toți strămoșii lui u cîștigă Δ populație, deci parcurgem toate lanțurile de la u la rădăcină și adăugăm Δ pe prefixele corespunzătoare ale acestor lanțuri.

Cum stabilim maximul pentru cazul (2) de mai sus? Este nevoie de puțin spirit de observație. Dacă insistăm să interogăm doar fiii lui u, aceștia sînt dispersați prin liniarizare. Dar este suficient să interogăm **tot subarborele** lui u, fără u însuși. Nu avem cum să greșim, căci, dacă maximul s-ar afla într-un nepot sau strănepot al lui u, cu atît mai mult fiul din care provine acel nepot sau strănepot ar avea populație și mai mare. Așadar, aflăm subarborele maxim cu o interogare pe intervalul $[t_{in}[u] + 1, t_{out}[u]]$.

Implementarea necesită doar un arbore de intervale cu adăugare pe interval și maxim pe interval.

Soluție cu arbore de intervale + caz separat pentru fiul greu

Nu-mi place să dau doar soluții de idee. Pentru aceea există matematica. 😁 Iată și o soluție mai muncitorească.

Din nou, la actualizarea unui nod facem operații de adăugare pe interval pe toate lanțurile. Acum diferențiem cazul (2) în două subcazuri: Subarborele cu populație maximă îl are fiul greu sau unul dintre fiii ușori. Ca să evităm confuzia, reamintesc că fiul greu este ales după numărul de noduri, care nu are neapărat și populația maximă.

Fiul greu îl putem interoga în $\mathcal{O}(\log n)$. Putem chiar să folosim doar un arbore Fenwick (AIB) cu actualizare pe interval și interogare punctuală.

Nu ne permitem să interogăm toți fiii ușori, dar este suficient să-l aflăm eficient pe cel mai populat. Cel mai „ciobănește”, fiecare nod poate menține un `std::multiset` cu mărimile subarborilor ușori (așadar nu și mărimea subarborelui greu). Atunci pentru a răspunde la interogări comparăm trei valori:

1. Părintele (populația totală minus populația nodului însuși).
2. Fiul greu.
3. Fiul ușor (maximul din `multiset`).

Ce implică actualizările în această structură? Partea frumoasă este că **nu** trebuie să actualizăm seturile nodurilor de pe fiecare lanț vizitat, căci acele lanțuri constau, prin definiție, din muchii grele. Trebuie actualizate doar nodurile-părinte ale fiecărui lanț, căci doar acele muchii sînt ușoare.

De amorul artei, putem folosi și o structură mai elegantă decît seturile STL. O parcurgere BFS este suficientă, căci în acea parcurgere fiii fiecărui nod devin vecini. Așadar, ne este suficient un arbore de intervale peste parcurgerea BFS, cu actualizare punctuală și interogare de maxim pe interval. Iată o implementare.

Am observat că testele nu pedepsesc iterarea naivă prin fiii ușori. Am trimis și o ultimă sursă care ia 100p astfel.

Soluție cu descompunere în radical

Includ și această soluție pentru familiarizarea cu astfel de alternative la HLD. Ea obține 70 de puncte.

Procesăm operațiile în blocuri de mărime circa $\sqrt{q}$. La începutul fiecărui bloc facem un DFS ca să calculăm $S(u)$ pentru toate nodurile. Acum, pentru o interogare în nodul u, am dori să luăm maximul dintre valorile pentru fiii lui u ai căror arbori nu au suferit modificări (valori pe care le știm deja) și valorile fiilor modificați, aduse la zi. Similar, exteriorul subarborelui lui u poate să fi suferit modificări.

Dacă avem 10 interogări într-un singur nod u, pe permitem să consultăm toți fiii lui u **o singură dată**. Asta face, de exemplu, DFS-ul, al cărui efort total este $\mathcal{O}(n)$. Nu ne permitem să iterăm prin toți fiii lui u pentru fiecare interogare, căci ajungem la $\mathcal{O}(q \cdot n)$, de exemplu pentru un arbore-stea cu toate interogările în centrul arborelui.

Observațiile-cheie sînt că există interogări despre cel mult $\sqrt{q}$ noduri, iar pentru fiecare nod u cel mult $\sqrt{q}$ dintre subarborii lui u au modificări.

Restul nu este trivial, dar sînt doar detalii de implementare care urmăresc să obțină:

1. Efort $\mathcal{O}(n)$ la începutul blocului.
2. Vizitarea fiilor nemodificați ai lui u o singură dată per bloc $\rightarrow$ efort total $\mathcal{O}(n)$ per bloc.
3. Vizitarea fiilor modificați o dată per interogare $\rightarrow$ efort $\mathcal{O}(\sqrt{q})$ per interogare.

Dacă facem asta, obținem complexitatea totală $\mathcal{O}((n+q)\sqrt{q})$.

Distribuim actualizările din bloc în noduri. Colectăm actualizările din bloc și le sortăm după timpul DFS. Este important să avem în vedere că nu toate actualizările se aplică tuturor interogărilor. Depinde de ordinea lor cronologică dinaintea sortării. Din punct de vedere al unui nod u, actualizările vor fi:

- actualizări în afara subarborelui lui u;
- actualizări în u însuși;
- actualizări în primul fiu al lui u;

- ...
- actualizări în ultimul fiu al lui u;
- actualizări în afara subarborelui lui u.

Astfel putem afla trei cantități pentru fiecare interogare

1. modificările din afara subarborelui (spre părinte);
2. modificările pe cel mai populat fiu nemodificat;
3. modificările pe fiecare fiu modificat.

Calculăm aceste informații într-o singură trecere prin fiii lui u. Categoriile (1) și (2) cer efort $\mathcal{O}(1)$, iar categoria (3) cere efort $\mathcal{O}(1)$ per interogare, căci trebuie să decidem dacă modificarea se aplică fiecărei interogări sau nu.

12.4.4 Problema Doi arbori (Lot 2024)

enunț • surse

Facem întîi o observație teoretică. Drumul cel mai scurt de la G_u la H_v este calea $u - v$ plus un ocol dus-întors de la unul dintre nodurile de pe calea $u - v$ la o frunză activă. Așadar problema se reduce la două subprobleme:

1. (simplă) Aflarea distanțelor între noduri.
2. (grea) Aflarea distanței minime de la orice drum de pe calea $u - v$ la o frunză activă.

Facem și o altă observație: nodul 1 poate fi și el frunză (în sensul că poate avea grad 1). Dacă înrădăcinați arborele în nodul 1 ca toată lumea, aceasta poate fi o sursă de buguri. Pare totuși că testele nu acoperă acest caz.

Soluția cu HLD

Să înrădăcinăm arborele în 1 și să construim HLD. Acum fie o interogare (u, v) și fie $w = LCA(u, v)$. Prin definiție, orice cale în arbore întîi urcă, apoi coboară (și urcușul și coborîșul pot avea lungime 0). De aceea, cînd căutăm calea spre cea mai apropiată frunză activă de orice nod de pe calea $u - v$ este suficient să tratăm cazurile:

1. Pornim dintr-un nod de pe cale și coborîm pînă la frunză.
2. Pornim din w, urcăm și apoi coborîm.

Vom stoca pe lanțuri informații care să ne permită să tratăm aceste două cazuri.

Cazul I: Drumul spre frunză coboară

Să considerăm unul dintre lanțurile vizitate pe calea $(u - w)$. Din acest lanț vom interoga o porțiune $[x, y]$. x este fie capătul de sus al lanțului, fie w. y este fie u, fie nodul în care se ancorează lanțul vizitat anterior.

Dacă drumul spre o frunză pornește chiar din y, atunci ne-ar ajuta să aflăm rapid adîncimea minimă a unei frunze active din subarborele lui y. Știm să facem asta cu un arbore de intervale S_1

cu actualizări punctuale și interogări de minim. La activarea unei frunze, notăm chiar adîncimea frunzei pe poziția corespunzătoare în liniarizare. La dezactivarea frunzei, notăm ∞. Putem afla adîncimea minimă a unei frunze din subarborele lui y cu o interogare în S_1 pe intervalul subîntins de y. Distanța pînă la frunza f este diferența de adîncimi dintre f și y.

Dacă drumul spre frunză pornește de undeva dintr-un nod-ancoră $a \in [x, y)$, atunci distanța pînă la frunză este diferența de adîncimi dintre frunză și a. Desigur, nu vrem să interogăm fiecare nod din $[x, y)$, dar facem următoarea observație. O frunză de adîncime 30 ancorată într-un nod de adîncime 20 are aceeași distanță (10) ca și o frunză de adîncime 31 ancorată într-un nod de adîncime 21. De aceea, menținem un al doilea arbore de intervale S_2 care stochează pentru un nod u **diferența** dintre adîncimea oricărei frunze din subarborele lui u și adîncimea lui u.

Cazul II: Drumul spre frunză urcă din w

Vom urca pe lanțuri pînă la rădăcină. Din nou, fie $[x, y]$ porțiunea din lanțul curent pe care o interogăm.

Dacă drumul spre frunză pornește chiar din y, atunci ne interesează chiar adîncimea frunzei, iar din lanțul $w - y -$ frunză aflăm imediat distanța de la calea $u - v$ la frunză. Arborele de intervale S_1 este util și aici.

Dacă drumul spre frunza f pornește dintr-un nod-ancoră $a \in [x, y)$, atunci distanța totală de la calea $u - v$ la f este (notînd cu $d(u)$ adîncimea nodului u):

$$[d(w) - d(a)] + [d(f) - d(a)] = d(w) + [d(f) - 2 \cdot d(a)]$$

De aceea, avem nevoie de un al treilea arbore de intervale, S_3, care stochează pentru un nod u diferența dintre adîncimea oricărei frunze din subarborele lui u și **dublul** adîncimii lui u.

Detaliu (esențial)

La schimbarea stării unei frunze f vom recalcula în arborii de intervale valorile următoarelor noduri:

- frunza f;
- nodul în care se ancorează lanțul frunzei;
- nodul în care se ancorează lanțul anterior;
- etc.

Remarcăm, în particular, că frunza f nu notifică alți strămoși de pe propriul ei lanț. Fie x un astfel de strămoș. Se poate întîmpla următorul scenariu:

- Frunza f este activată.
- O altă frunză f' din subarborele lui x este activată.
- f' declanșează actualizarea lui x, care va include și informații despre f.
- Frunza f este deactivată.
- x nu mai află niciodată despre dezactivarea lui f.

Soluția este ca, la propagarea în sus a unei modificări, să interogăm în S_1 **doar subarborele light al unui nod**. Ce modificare este necesară ca să putem identifica intervalul corespunzător din liniarizare?

Soluția cu descompunere în radical

Menționez foarte pe scurt și o soluție în $\mathcal{O}((n+q)\log q)$. Ea nu trece testele mari. Dar, în general, soluțiile cu descompunere în radical sînt o alternativă viabilă și posibil mai ușor de înțeles. Iar matematic $\sqrt{n}$ nu este departe de $\log^2 n$ pe plaja de valori pentru n din problemele obișnuite.

Precalculăm liniarizarea Euler cu repetiție a arborelui, peste care construim tabela rară de RMQ. Astfel putem răspunde la interogări de LCA în $\mathcal{O}(1)$ (avem nevoie de $\mathcal{O}(1)$ deoarece vom face $\mathcal{O}(q\sqrt{q})$ astfel de interogări).

Precalculăm *binary lifting*, preferabil varianta cu doar doi pointeri. Astfel vom putea calcula în $\mathcal{O}(\log n)$ o anumită informație pe căi.

Procesăm interogările în blocuri de mărime $\sqrt{q}$. Clasificăm frunzele active în două categorii:

1. Frunze safe: frunze active pe durata întregului bloc.
2. Frunze unsafe: frunze active pe porțiuni din bloc.

La începutul fiecărui bloc, facem următoarea preprocesare în $\mathcal{O}(n)$:

1. Clasifică frunzele active în *safe* și *unsafe*.
2. Calculează distanța minimă de la fiecare nod la o frunză *safe*. Este suficient un BFS multi-sursă.
3. Pentru fiecare pointer de salt, calculează distanța minimă de la orice nod acoperit de salt la o frunză *safe*. Este suficient un DFS.

Efortul total pentru preprocesări este $\mathcal{O}(n\sqrt{q})$. Procesăm operațiile din bloc și ținem evidența mulțimii de frunze *unsafe*. Pentru o interogare (u, v), răspunsul va fi minimul dintre

1. Distanța minimă de la cale la o frunză *safe*, calculată în $\mathcal{O}(\log n)$ cu *jump pointers*.
2. Pentru fiecare frunză *unsafe* f, distanța $d(u, f) + d(f, v)$.

Deoarece fiecare dintre cele q interogări poate consulta $\mathcal{O}(\sqrt{q})$ frunze *unsafe*, efortul total este $\mathcal{O}(q \log q)$ pentru interogări.

Momentul de inginerie: măsurarea timpului

Pentru soluția cu descompunere în radical, nu am reușit să reduc timpul de rulare sub 4-5 secunde pe testele mari. Nevenindu-mi să cred că poate fi nevoie de atît de mult timp, am măsurat numărul de apeluri la diverse funcții și timpul petrecut în ele. Las aici codul pe care l-am folosit, în caz că îl ajută pe inginerul din voi.

```
#include <sys/time.h>
```

```
long long t0, t_total, cnt;

void start_clock() {
  timeval tv;
  gettimeofday(&tv, NULL);
  t0 = 1'000'000LL * tv.tv_sec + tv.tv_usec;
}

void stop_clock() {
  timeval tv;

  gettimeofday(&tv, NULL);
  long long t = 1'000'000LL * tv.tv_sec + tv.tv_usec;
  t_total += t - t0;
}

int preprocess_block(int start) {
  cnt++;
  start_clock();   // <---------------
  int end = classify_leaves(start);
  bfs_from_safe_leaves();
  jdist_dfs(1);
  stop_clock();    // <---------------

  return end;
}

int main() {
  ...
  fprintf(stderr, "Time: %0.6lf cnt: %lld\n", t_total * 0.000001, cnt);
}
```

12.4.5 Problema Query on a Tree VI (CodeChef)

enunț • sursă

Problema are un enunț simplu și elegant, dar este foarte laborioasă.

Am „trișat” un pic la rezolvare, căci știam că este problemă de HLD și mi-am pus întrebarea: HLD ne ajută să facem operații pe căi, deci cum aș folosi-o la o problemă despre subarbori? Așa am ajuns la o soluție parțială.

Definim **domeniul unui nod** u ca fiind mulțimea de noduri din subarborele lui u, de aceeași culoare cu u, și legate de u prin noduri de aceeași culoare. Fie $S(u)$ = mărimea domeniului lui u. Atunci, ca să răspundem la o interogare despre u, urcăm din u cît timp se poate, mergînd pe noduri de aceeași culoare. Fie w ultimul nod atins. Răspunsul este $S(w)$.

Dacă folosim HLD, îl putem găsi pe w dacă menținem pe fiecare lanț un AIB de culori (0/1) cu suport pentru căutare binară.

Treburile se complică la actualizare. Cînd un nod u își schimbă culoarea, schimbările necesare sînt:

1. Toți strămoșii consecutivi care au vechea culoare a lui u pierd $S(u)$ din domeniu.
2. Recalculăm $S(u)$.
3. Toți strămoșii consecutivi care au noua culoare a lui u cîștigă $S(u)$ la domeniu.

Operațiile (1) și (3) sînt operații de adăugare / scădere pe interval. Dar nu am reușit să implementez eficient operația (2). Astfel ajungem la soluția oficială. Menținem pentru fiecare nod **două valori**:

1. $S_B(u)$ = mărimea domeniului lui u dacă u ar fi negru.
2. $S_W(u)$ = mărimea domeniului lui u dacă u ar fi alb.

Actualizarea acestor valori presupune niște cazuri particulare. De exemplu, dacă un nod u devine alb, atunci toți strămoșii săi negri, **dar și primul strămoș alb**, pierd $S_B(u)$ din $S_B(w)$.

Temă de gîndire: „Aint pe timp"

Am implementat și o altă soluție bazată pe metoda TODO:referință-internă ștergerii dintr-o structură care nu admite (ușor) ștergeri, cu un arbore de intervale indexat după timp. Problema fiind una de conectivitate online, ideea mi s-a părut bună. Dar am făcut o greșeală copilărească: aici nu activăm și dezactivăm muchii, ci noduri. Deci o operație poate cauza $\mathcal{O}(n)$ operații în pădurea de mulțimi disjuncte. Sursa mea ia TLE pe teste adversariale.

Nu știu dacă putem reduce complexitatea. Voi ce credeți?

12.4.6 Problema Adă caii (Lot 2025)

enunț • sursă

Să pornim de la o descompunere *heavy-light* tradițională. Nu ne batem prea mult capul cu operația de interogare. Aceea este punctuală, deci probabil că orice structură vom construi peste lanțuri vom putea să aflăm o valoare punctuală. În schimb, să analizăm migrația spre un nod u. Observăm că:

1. Pe un lanț L aflat între u și rădăcină, caii migrează către nodul în care calea către u se desprinde din L.
2. Pe alte lanțuri, caii migrează în sus.
3. Caii pot sări de pe un lanț pe altul.

De exemplu, în figura 12.2, migrația către nodul 28 (albastru-deschis) înseamnă că:

1. Pe lanțul roșu 1-30, caii migrează către nodul 20.
2. Caii din nodul 20 coboară pe lanțul violet în nodul 32.
3. Pe lanțul violet 32-19 caii migrează către nodul 14.
4. Caii din nodul 14 coboară pe lanțul albastru deschis în nodul 38.
5. Pe alte lanțuri, caii migrează în sus.

6. Caii aflați în nodurile din vîrful altor lanțuri migrează pe lanțul-părinte.

Așadar, pe fiecare lanț dorim să stocăm o structură de date care implementează rezonabil de eficient operațiile:

1. Migrează toate valorile spre stînga (spre rădăcină).
2. Migrează toate valorile spre o poziție din structură.
3. Citește / modifică o poziție.

Operațiile (1) și (3) sînt ușor de implementat în $\mathcal{O}(1)$ cu un buffer circular pe fiecare lanț. Am ales să stochez aceste buffere într-un singur vector global, separat de nodurile arborelui.

În schimb, operația (2) pare a fi $\mathcal{O}(\text{lungime\_lanț})$. De aceea, vom pune o limită de $\mathcal{O}(\sqrt{n})$ pe lungimea oricărui lanț. Dacă un lanț ar fi, în mod natural, mai lung de atît, după primele $\mathcal{O}(\sqrt{n})$ noduri vom forța începerea unui lanț nou. Aceasta va cauza apariția mai multor lanțuri, dar nu multe.

Cu această modificare, și pentru o optimizare posibil valoroasă, vom implementa (2) copiind naiv jumătatea mai scurtă a vectorului și shiftînd-o pe cea mai lungă conform bufferului circular.

În plus, ținem evidența lanțurilor nevide, deoarece doar pe acelea avem de implementat operația (1). Pot exista $\mathcal{O}(n)$ lanțuri, de exemplu într-un arbore stea, dar pare imposibil să menținem populații de cai pe $\mathcal{O}(n)$ lanțuri. Cu fiecare operație de migrare, unele populații de cai se vor ciocni și se vor însuma.

Sînt 99% sigur că acest algoritm este eficient, dar demonstrația este alunecoasă. Intuiția mea îmi spune astfel. Temerea noastră este că poate exista o migrație (sau mai multe) care să necesite efort $\mathcal{O}(n)$. Aceasta s-ar putea întîmpla dacă există $\mathcal{O}(\sqrt{n})$ lanțuri pe calea de la u la rădăcină (datorită limitei de lungime), iar pe fiecare lanț este nevoie să facem operația (2) și să copiem naiv $\mathcal{O}(\sqrt{n})$ elemente.

Dar, dacă drumul de la u la rădăcină include vreun lanț complet, pe acela nu vom face efort $\mathcal{O}(\text{lungime})$, ci doar $\mathcal{O}(1)$, deoarece migrația în acel lanț va fi doar în jos, spre u! De exemplu, în figura 12.2, pentru $u = 30$ elementele de pe lanțul roșu migrează doar în jos.

De aceea, un caz adversarial ar fi o cale de la u la rădăcină care, ori de cîte ori schimbă lanțul, se „înțeapă" undeva la jumătatea noului lanț. Or acest lucru este imposibil datorită modului în care funcționează decompunerea *heavy-light*. Din punct de vedere al nodului de înțepare, lanțul ar prefera să continue pe calea spre u, dacă aceasta este destul de lungă. Așadar, dacă am avea $\mathcal{O}(\sqrt{n})$ lanțuri, toate de lungime $\mathcal{O}(\sqrt{n})$, numărul total de noduri de pe acea cale ar deveni $\mathcal{O}(n)$ și majoritatea lanțurilor s-ar începa în capătul de jos al lanțului anterior.

Capitolul 13

Descompunere în centroizi

Încheiem studiul arborilor cu tehnica descompunerii în centroizi. Ea ne permite să rezolvăm probleme rezistente la alte abordări.

Ca și descompunerea *heavy-light*, descompunerea în centroizi este o unealtă avansată. Este bine să o știți, căci uneori nu găsim o soluție mai elementară. Dar problemele de ONI și Baraj ONI se pot rezolva și cu tehnici mai simple.

13.1 Definiție

Fiind dat un arbore cu n noduri, un **centroid** este un nod al arborelui cu proprietatea că, dacă îl eliminăm, atunci toți arborii rezultați au cel mult $\lfloor n/2 \rfloor$ noduri.

Centroidul este un centru de masă, definit în funcție de greutatea subarborilor (ca număr de noduri). A nu se confunda cu **centrul**, care este definit în funcție de distanțe (el minimizează distanța maximă pînă la alt nod).

13.2 Proprietăți

Orice arbore are un singur centroid sau doi centroizi adiacenți. Pentru demonstrație, să considerăm figura următoare.

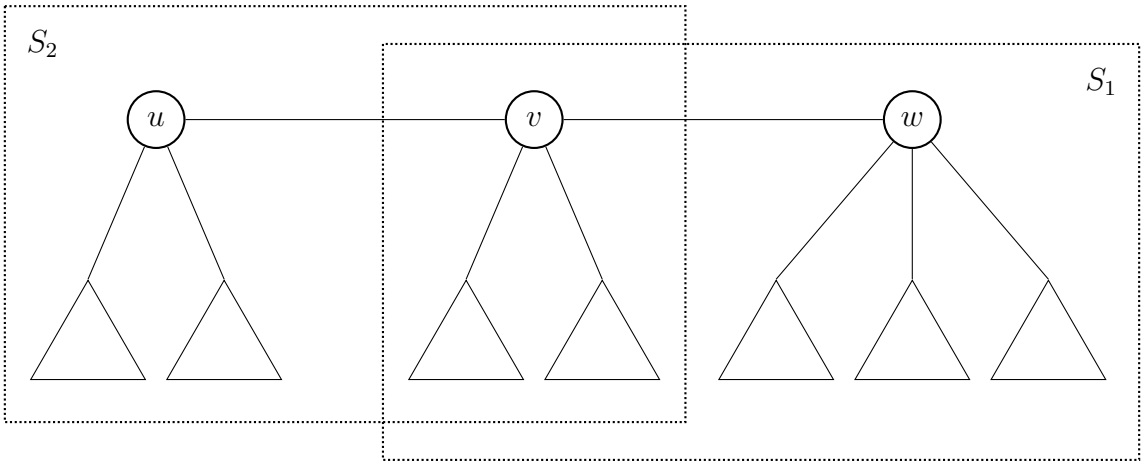


Figura 13.1: Un arbore ipotetic cu doi centroizi aflați la distanță 2.

Să presupunem că u și w ar fi centroizi aflați la distanță 2. Din punctul de vedere al lui u, subarborele S_1 are cel mult $\lfloor n/2 \rfloor$ noduri. Din punctul de vedere al lui w, subarborele S_2 are, de asemenea, cel mult $\lfloor n/2 \rfloor$ noduri. Rezultă că

$$|S_1| + |S_2| \leq n$$

Dar acest lucru este imposibil, căci în mod evident S_1 și S_2 acoperă întreg arborele, iar nodul v (și orice eventuali subarbori ai săi) sînt acoperiți de două ori. Deci $|S_1| + |S_2| \geq n + 1$.

Așadar, dacă există mai mulți centroizi, atunci ei sînt adiacenți. Dar într-un arbore nu putem amplasa mai mult două noduri astfel încît oricare două să fie adiacente. Deci există cel mult doi centroizi.

13.3 Găsirea unui centroid

După cum vom vedea în problema Finding a Centroid, algoritmul de găsire a unui centroid este simplu. Facem un DFS pentru calculul mărimilor subarborilor. Apoi, pornind din rădăcină, căutăm succesiv fii cu mai mult de $n/2$ noduri și coborîm în ei pînă cînd nu mai găsim niciunul. Complexitatea este $\mathcal{O}(n)$.

13.4 Descompunerea în centroizi

Descompunerea în centroizi repetă de mai multe ori următoarea procedură:

1. Găsește un centroid pentru subarborele curent.
2. Colectează informații din subarborele curent care ajută la rezolvarea problemei. Probabil folosește unul sau mai multe DFS-uri pornind din centroid.
3. Elimină centroidul.
4. Reapelează recursiv algoritmul pentru subarborii disjuncți care iau naștere.

5. Cazul de bază este un subarbore cu un singur nod. Nodul este centroid.

Iată un exemplu în care descompunerea în centroizi necesită cinci niveluri de adîncime.

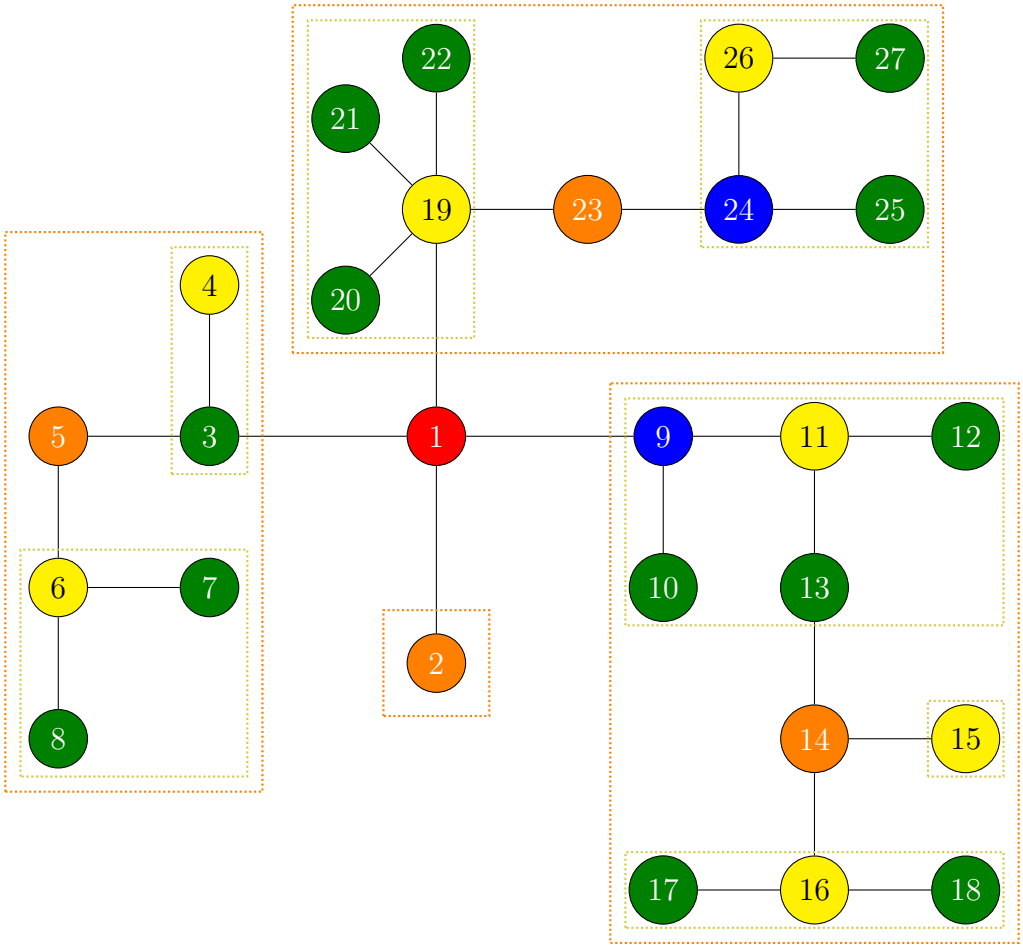


Figura 13.2: Un arbore descompus în centroizi. Nodul 1 (roșu) este centroid de nivelul 1. Pentru subarborii rezultați prin eliminarea lui 1, nodurile portocalii sînt centroizi de nivelul 2. Nodurile galbene, verzi și albastre sînt centroizi de nivelurile 3, 4 și respectiv 5.

Această abordare are avantajul că fiecare nod face parte din cel mult $\lceil \log n \rceil$ descompuneri. De exemplu, nodul 8 face parte din:

- Arborele inițial (cu centroidul 1).
- Arborele cu centroidul 5 (chenar portocaliu).
- Arborele cu centroidul 6 (chenar galben).
- Arborele constînd doar din nodul 8, cu centroidul 8.

De aceea, o implementare care face efort $\mathcal{O}(\text{mărime\_subarbore})$ în fiecare subarbore, cum ar fi de exemplu un DFS, va ajunge la o complexitate totală de $\mathcal{O}(n \log n)$.

Atenție! Efortul trebuie să depindă de mărimea subarborelui, **nu** de n. Fiecare nod va deveni centroid la un nivel mai mare sau mai mic de fărîmițare. Așadar, dacă orice bucată din cod face efort $\mathcal{O}(n)$ per nod, atunci complexitatea totală va deveni $\mathcal{O}(n^2)$. Uneori această condiție

necesită un efort conștient, de exemplu la dimensionarea, inițializarea și ștergerea structurilor de date pentru subarborele curent.

Dacă efortul într-un subarbore este mai mare decît liniar, atunci descompunerea în centroizi adaugă un factor logaritmic. De exemplu, dacă efortul într-un subarbore de mărime k este $\mathcal{O}(k \log k)$, atunci complexitatea totală devine $\mathcal{O}(n \log^2 n)$. Pentru detalii, vedeți Master Theorem, cazul 2.

Descompunerea în centroizi ne spune „este OK să apelezi un DFS din fiecare nod". Desigur, ce face acel DFS depinde de la problemă la problemă. Dar ocazional putem scrie algoritmi relativ naivi (plus șablonul de cod pentru descompunere) și totuși să obținem complexitate $\mathcal{O}(n \log n)$.

13.5 Probleme

13.5.1 Problema Finding A Centroid (CSES)

enunț • surse

Nu este foarte mult de spus, problema este directă. Reținem șablonul util pentru toate problemele de descompunere în centroizi:

1. DFS-ul inițial dintr-un nod oarecare, cu aflarea mărimii subarborilor.
2. Caută un fiu greu al nodului curent.
3. Dacă există un fiu greu, coboară în el și mergi la pasul anterior.
4. Ultimul nod găsit este centroidul.

Există două implementări. Cea recursivă (la coadă) combină pașii (2) și (3) într-o singură funcție. Odată identificat fiul greu, reapelăm căutarea centroidului din acel fiu. Implementarea iterativă separă pașii (2) și (3) și rezolvă pasul (3) cu un `while`.

13.5.2 Problema Mystery Tree (CodeChef)

enunț • sursă

Iată o problemă simpatică, interactivă. Se dă un arbore cu $n \leq 200\,000$ de noduri, cu valori în noduri. Structura arborelui este cunoscută, dar valorile nu. Trebuie să găsim un maxim local: un nod cu valoarea mai mare sau egală cu valorile tuturor vecinilor săi. Putem pune cel mult 20 de întrebări de forma „Este u un maxim local?". Interactorul ne răspunde cu -1 dacă am găsit un maxim local sau, în caz contrar, cu un nod v cu valoare strict mai mică decît a lui u.

Cum am rezolva problema pe un vector?

Pentru generalizarea pe arbore, trebuie să punem fiecare întrebare despre un nod care reduce problema la jumătate sau mai puțin. Acesta este exact centroidul subproblemei curente.

Ca implementare, aici putem vedea cum ștergem un nod. Nu modificăm listele de adiacență, ci doar îl marcăm ca fiind șters.

Din păcate, la această problemă nu putem trimite surse. Dar am încredere în corectitudinea soluției mele!

13.5.3 Problema Ciel the Commander (Codeforces)

enunț • surse

Soluția cu descompunere în centroizi

Problema se reduce absolut elementar la decompunerea în centroizi. În centroidul întregului arbore scriem A, în centroizii de nivel 2 scriem B etc. Alfabetul este suficient deoarece $\lceil \log_2 100\,000 \rceil = 17$.

Soluție cu un singur DFS și măști de biți

Putem folosi și următoarea abordare *greedy*. În frunze scriem Z, căci nu avem niciun motiv să nu facem asta. În părinții frunzelor putem scrie Y. Complicațiile apar cînd un nod are un fiu Y și un fiu Z. Ce facem în cazul general?

Să introducem noțiunea de **vizibilitate**. Spunem că o literă X este **vizibilă** dintr-un nod u dacă există o apariție a lui X undeva în subarborele lui u care să nu fie **mascată** de o literă mai mică decît X pe calea pînă la u.

Ideea centrală este că o literă mai mică maschează toate literele mai mari din subarbore, care nu mai necesită alte acțiuni.

Atunci am putea pune în u cea mai mare literă care satisface două condiții:

1. Trebuie să fie mai mică decît orice literă care este vizibilă din doi fii diferiți ai lui u, deoarece trebuie să separăm cele două apariții.
2. Trebuie să fie diferită de orice literă vizibilă din u.

Odată ce ia această decizie, u returnează la părinte mulțimea de litere vizibile. Putem folosi măști de biți și operații de aflare a LSB pentru a obține o soluție în $\mathcal{O}(n)$.

13.5.4 Problema Fixed-Length Paths I (CSES) (din nou)

enunț • sursă

Ne reîntîlnim cu această problemă, pe care am rezolvat-o cu tehnica *small-to-large*. În lipsa acelei idei, descompunerea în centroizi ne poate da o idee mai directă.

O cale de lungime k (și orice cale în general) fie trece prin centroidul arborelui, fie este conținută complet într-unul dintre subarbori. Deci putem însuma răspunsurile întrebărilor, pentru fiecare centroid, „Cîte căi de lungime k trec prin tine?”.

Iar răspunsul la această întrebare este comparativ mai simplu decît tehnica *small-to-large*. Facem un DFS din centroid. Fiecare fiu al centroidului raportează distribuția pe adîncimi a nodurilor

din subarborele său. Părintele (centroidul) combină aceste informații. Putem face asta în diverse feluri, dar esența este: un nod aflat la adîncime d formează căi de lungime k cu toate nodurile aflate la adîncime $k - d$ în fiii centroidului vizitați anterior.

Această implementare este de peste 2 ori mai lentă decît implementarea cu *small-to-large*.

Întrebare despre cod: care este rostul variabilei `max_depth`?

13.5.5 Problema Xenia and Tree (Codeforces)

enunț • surse

Soluția cu descompunere în radical

Menționăm sumar și această soluție, ca să fiți mereu alerți la posibilitățile de rezolvare. Este o soluție lunguță, dar nu grea. Soluția seamănă mult cu cea de la Doi arbori și este mai simplă decît aceea.

Procesăm operațiile în blocuri de circa $\sqrt{q}$. La începutul unui bloc facem o parcurgere BFS din toate nodurile roșii. Acesta ne va oferi pentru fiecare nod u o cantitate $d[u]$ care reprezintă distanța minimă de la u pînă la orice nod roșu.

Acum, răspunsul la o interogare din acel bloc va fi minimul dintre $d[u]$ și distanța de la u pînă la orice nod colorat în roșu cîndva în blocul curent. Dar în blocul curent colorăm în roșu cel mult $\sqrt{q}$ noduri, deci putem răspunde la interogări calculînd cel mult $\sqrt{q}$ distanțe.

Pentru a obține timp $\mathcal{O}(\sqrt{q})$ per interogare, este necesar să putem calcula distanțe în $\mathcal{O}(1)$, ceea ce se reduce la a calcula LCA în $\mathcal{O}(1)$. Vom folosi structura cu care ne-am mai întîlnit: o parcurgere Euler peste care construim tabela rară de RMQ.

Soluția se încadrează lejer în timp (sub 1s pe limită de timp de 5s).

Soluția cu descompunere în centroizi

Și acum, la subiectul zilei! Calea de la un nod u la orice nod roșu (sau, în general, orice alt nod) va fi cuprinsă complet într-un subarbore centroid: în cel mai rău caz subarborele centroid al nodului 1 (care este întregul arbore), dar posibil într-un subarbore mai mic.

Prin natura ei, descompunerea în centroizi garantează că orice nod u face parte din cel mult $\log n$ subarbori centroizi sau, echivalent, are cel mult $\log n$ strămoși centroizi. Și atunci, la o interogare, am putea să-i verificăm pe toți acești strămoși. Așadar, pentru interogarea u și pentru fiecare v strămoș centroid al lui v, ne întrebăm:

1. Care este distanța de la u la v?
2. Care este distanța de la v pînă la orice nod roșu din subarborele centroid al lui v?

Partea frumoasă este că putem accesa aceste distanțe în $\mathcal{O}(1)$ fără *binary lifting*, LCA, RMQ sau alte structuri suplimentare. Reamintiți-vă că **ne permitem cîte un DFS din fiecare**

centroid. Într-un astfel de DFS dintr-un centroid de nivel k vom calcula distanțele de la fiecare nod din subarborele centroidului pînă la centroid. Facem această preprocesare înainte de a procesa interogările.

Memoria necesară pentru aceste informații este $\mathcal{O}(n \log n)$.

Ce implică actualizările? La colorarea unui nod u, toți strămoșii centroizi ai lui u trebuie notificați că în subarborele lor a apărut un nod nod roșu. Fiecare strămoș v își reține distanța pînă la cel mai apropiat nod roșu din subarborele său și și-o minimizează cu distanța $u - v$ (pe care u o cunoaște din DFS-urile de preprocesare).

Complexitatea provine din:

1. Descompunerea în centroizi, care știm că este $\mathcal{O}(n \log n)$.
2. Procesarea celor q operații. Colorările și interogările presupun modificarea sau consultarea unui vector de $\log n$ elemente.

Rezultă o complexitate totală de $\mathcal{O}((n + q) \log n)$. Cum este și de așteptat, sursa este de peste două ori mai rapidă decît cea bazată pe descompunere în radical.

13.5.6 Problema Flareon (Lot 2017)

enunț • sursă

Să presupunem că ne-a venit deja ideea să folosim descompunerea în centroizi. 😈 Să considerăm o flacără care pornește dintr-un nod u. Fie $c_1, c_2, \ldots, c_k$ strămoșii centroizi ai nodului u, unde c_1 este rădăcina arborelui (nodul 1 în majoritatea implementărilor), iar c_k este chiar nodul u. Vom contabiliza separat contribuția flăcării pe căi care trec prin c_1, pe căi care trec prin c_2 fără să ajungă la c_1 etc.

Deci vom încerca o soluție în doi pași. Fie r rădăcina (centroidul) subarborelui curent.

1. Colectăm în r lista de flăcări care provin din subarborele lui r. Pentru fiecare flacără ne interesează puterea cu care ajunge în rădăcină.
2. Propagăm această listă de flăcări în tot subarborele. Nu avem voie să propagăm o flacără în subarborele din care provine.

Notînd cu s mărimea subarborelui curent, este important ca ambii pași să funcționeze în $\mathcal{O}(s)$. În particular, nu putem colecta flăcările într-o listă, ci trebuie să le compactăm cumva într-o structură de mărime cel mult s.

Reprezentarea flăcărilor

Ideea compactării este următoarea. Dacă dintr-un nod pornesc 3 flăcări de mărime 2, 5 și 10, le putem unifica într-o singură flacără de mărime 17, a cărei putere scade cu 3 la fiecare muchie parcursă. Așadar, reținem doar suma puterilor și numărul de flăcări. La fiecare deplasare, suma puterilor scade cu numărul de flăcări.

Aceasta funcționează... o vreme. În primul nod avem putere 17, în următorul 14 (= 1 + 4 + 9), iar în următorul 11 (= 0 + 3 + 8). Dar aici flacăra mică se stinge, iar puterea trebuie să continue să scadă doar cu 2. Următoarea putere este 9 (= 2 + 7).

Totuși, sîntem pe calea bună. Putem grupa flăcările după putere. Să spunem că în rădăcina r am acumulat $f[p]$ flăcări de putere p, unde $p \geq 1$. Atunci cînd propagăm aceste flăcări în subarbore, la un nod de la adîncimea d trebuie însumate doar flăcările cu $p \geq d$. Cînd reapelăm DFS-ul pentru un fiu, ne aflăm la adîncime $d+1$, deci le scădem din numărul total de flăcări pe cele $f[d]$ care s-au stins.

Remarcăm că puterile pot fi mari (problema nu specifică, dar ele se apropie de 10^9). Nu putem stoca un vector atît de mare, dar nici nu este nevoie. Față de abordarea inițială (doar cu numărul de flăcări și numărul puterilor), ne interesează și frecvențele **doar pentru flăcările care se vor stinge cîndva**. Flăcările care au o putere mai mare sau egală cu s nu se vor stinge în acest subarbore.

Recapitulînd, informațiile necesare în rădăcină sînt (1) numărul de flăcări, (2) suma puterilor acestora și (3) distribuția pe frecvențe a flăcărilor de putere mai mică decît s.

Evitarea propagării în același fiu

Cum spuneam, o flacără nu trebuie propagată din r înspre același fiu din care flacăra a ajuns în r. Văd două abordări aici.

Una este să stocăm informații despre fiecare fiu în parte: ce flăcări ajung acolo și cu ce puteri rămase? Cînd propagăm flăcările printr-un fiu u, calculăm în fiecare nod diferența dintre informațiile din r și u.

Dintr-o sursă de pe Kilonova am învățat și o altă abordare elegantă.

1. Inițial contabilizăm toate flăcările cu semnul „+”.
2. Înainte de propagarea flăcărilor într-un fiu, facem un DFS ca sa contabilizăm flăcările din acel fiu cu semnul „-”.
3. După terminarea propagării, apelăm același DFS ca să contabilizăm flăcările din acel fiu, de data aceasta cu semnul „+”.

Din păcate, incluzînd și DFS-ul pentru găsirea centroidului, ajungem la 5 DFS-uri din fiecare centroid. Este o constantă măricică.

Idee de implementare: ștergerea nodurilor

Tocmai fiindcă codul include multiple DFS-uri, mi-am dat seama că „tîrăsc” după mine peste tot condiția `!nd[u].dead` prin tot codul. Astfel mi-a venit ideea: nu este mai scurt/rapid/clar să ștergem efectiv nodul?

Mai scurt nu este, căci la ștergerea lui u trebuie consultate toate listele de adiacență ale vecinilor lui u și șterse aparițiile lui u din acele liste. Deci codul se lungește cu 10-15 linii.

Mai rapid... discutabil. A doua sursă a mea este mai rapidă cu 14%, dar poate fi și un dram de noroc acolo. Eliminarea unui nod face totuși un efort proporțional cu suma gradelor vecinilor. Paradoxal, am optimizat codul ca să fac eliminarea în $\mathcal{O}(1)$ per muchie ștearsă și a devenit mai lent.

Dar aș argumenta că codul devine mai clar. Efectiv, arborele devine o pădure prin fărîmițare, iar subarborii nu au cunoștință unul de celălalt. Listele de adiacență nu mai sînt poluate de gunoaie.

Rămîne la alegerea voastră ce variantă folosiți.

13.5.7 Problema Digit Tree (Codeforces)

enunț • surse

Problema amintește puțin de Fixed Length Paths I, doar că nu trebuie să numărăm căile de o anumită lungime, ci pe cele care, citite ca număr zecimal, sînt divizibile cu M.

Din nou, vom arăta cum să rezolvăm problema pentru o rădăcină fixată, iar descompunerea în centroizi va face restul pentru noi.

Căi în sus și în jos

Cum procesăm o cale care trece prin rădăcină? Pare natural să o spargem în:

- calea care urcă pînă la rădăcină, care dă restul r_1 modulo M;
- calea care coboară din rădăcină, de lungime d muchii, care dă restul r_2 modulo M.

Întreaga cale va avea atunci restul $r_1 \cdot 10^d + r_2 \pmod M$. Dacă dorim ca acest rest să fie 0, atunci putem scrie

$$r_1 = (-r_2) \cdot 10^{-d}$$

Deci, pentru fiecare nod u, calculăm restul r_2 al căii de la rădăcină la u, apoi căutăm numărul de căi de la alte noduri v la rădăcină care dau restul r_1 dorit. Atenție, u și v trebuie să se găsească **în fii diferiți ai rădăcinii**. De asemenea, perechile (u, v) sînt ordonate. Cel mai corect pare să facem două DFS-uri separate, întîi pentru căile care urcă, apoi pentru cele care coboară.

Recapitulînd, pentru fiecare centroid:

1. Facem o parcurgere DFS. Calculăm resturile pe care le putem obține pe căi care urcă pînă în rădăcină și frecvențele acestor resturi.
2. Facem o a doua parcurgere DFS. Menținem lungimea căii curente și restul modulo M. În fiecare nod, calculăm restul-pereche necesar pentru a obține restul total 0. Aflăm frecvența acelui rest-pereche, ignorînd frecvența din fiul rădăcinii în care se află DFS-ul curent.

Mai trebuie tratate și două cazuri particulare, relativ simple:

1. Căi care doar urcă. După primul DFS, creștem răspunsul cu frecvența restului 0, indiferent din ce fiu.
2. Căi care doar coboară. În al doilea DFS, ori de cîte ori restul căii curente este 0, incrementăm răspunsul.

`map` și `unordered_map`

Concret, ce structură de date stocăm? Prima mea încercare a fost cu doar două tabele. Dat fiind un rest r pe o cale care urcă din u în rădăcină, într-un fiu v al rădăcinii, am incrementat două valori:

1. $t[r]$, unde r este un `map` de la întreg (restul) la întreg (frecvența restului). Așadar, t menține frecvențele resturilor indiferent din ce fiu provin.
2. $c[\langle r, v\rangle]$, unde c este un map de la (întreg,întreg) (restul și fiul rădăcinii pe care am pornit) la întreg (frecvența).

Atunci, în al doilea DFS, pornind pe un fiu al rădăcinii v și aflîndu-ne în nodul curent u, calculăm restul căii care coboară, r_2. Apoi calculăm restul corespunzător necesar r_1. În sfîrșit, aflăm numărul de apariții utile ale lui r_1 cu expresia

$$t[r_1] - c[\langle r_1, v\rangle]$$

Procedăm astfel deoarece calea nu poate să urce și să coboare tot prin v.

Am avut surpriza (deși nu mai este chiar surpriză) că `unordered_map` este mult mai lent decît `map`. O explicație poate fi că pentru fiecare centroid (așadar, de n ori) instanțiem două `unordered_maps`, ceea ce este lent. Cu `map` am obținut un timp mediu pe CF (1500-1600 ms). Complexitatea teoretică a crescut la $\mathcal{O}(n \log^2 n)$.

`map` cu eliminare și reinserare

Pentru un cod mai lent, dar considerabil mai scurt, folosim același artificiu ca la problema Flareon:

1. Menținem un singur map de frecvențe, t.
2. În acesta stocăm, ca mai sus, frecvențele resturilor pe căi care urcă.
3. La al doilea DFS, pe căile care coboară, iterăm prin fiii rădăcinii.
4. Înainte de a lansa DFS-ul dintr-un fiu, eliminăm căile care urcă prin acel fiu.
5. După revenirea din fiu, adăugăm la loc căile care urcă prin acel fiu.

Astfel evităm să combinăm căile care urcă și coboară prin acel fiu. Esența codului este:

```
void count_pairs_through(int u) {
  // ...

  for (edge e: nd[u].adj) {
    if (!nd[e.v].dead) {
```

```
      head_dfs(e.v, u, 1, e.digit, +1);
    }
  }

  for (edge e: nd[u].adj) {
    if (!nd[e.v].dead) {
      head_dfs(e.v, u, 1, e.digit, -1);
      tail_dfs(e.v, u, 1, e.digit);
      head_dfs(e.v, u, 1, e.digit, +1);
    }
  }

  // ...
}
```

Vectori + sortare + căutare binară

De amorul artei, am încercat și următoarea abordare. Să observăm că întîi facem toate inserările în structura noastră de date, apoi toate interogările de frecvență. De aceea, am înlocuit `map`-urile cu doi vectori:

1. În locul lui t, un vector de perechi <rest, frecvență>.
2. În locul lui c, un vector de tripleți <rest, fiu, frecvență>.

După primul DFS, am sortat tripleții și am comasat duplicatele (însumînd frecvențele). Pentru a căuta informații, am folosit căutarea binară.

Timpul de rulare s-a înjumătățit. De reținut!

Partea IV

Combinatorică

Capitolul 14

Elemente de bază

14.1 Formule pentru combinări

Încercați să memorați formule de combinatorică. Oricare dintre ele vă poate servi la un moment dat.

Coeficientul binomial:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{14.1}$$

O formulă echivalentă:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \cdots \cdot (n-k+1)}{k!} \tag{14.2}$$

O exprimare recurentă, care duce la triunghiul lui Pascal:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{14.3}$$

O listă de proprietăți, preluate din articolul foarte bun de pe CP Algorithms:

$$\binom{n}{k} = \binom{n}{n-k} \tag{14.4}$$

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{14.5}$$

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \tag{14.6}$$

$$\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1} \tag{14.7}$$

$$\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m} \tag{14.8}$$

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n} \tag{14.9}$$

$$\sum_{k=0}^{n} k\binom{n}{k} = n \cdot 2^{n-1} \tag{14.10}$$

14.2 Calculul combinărilor

Dacă putem stoca $\mathcal{O}(n^2)$ numere în memorie, putem calcula tabelul de combinări cu triunghiul lui Pascal.

Dacă avem de calculat doar o combinare ocazională și dacă rezultatul încape pe `long long`, putem folosi formula 14.2. Nici măcar nu este nevoie să calculăm inverse, căci nu facem împărțiri! Calculăm rezultatul de la dreapta spre stînga:

$$(n-k+1)/1 \cdot (n-k+2)/2 \cdot (n-k+3)/3...$$

Ne bazăm pe observația că, atunci cînd vine vremea să împărțim prin x, vom fi înmulțit deja x termeni consecutivi, dintre care unul se divide cu x.

Pentru valori mari, de regulă problema ne va cere să operăm modulo un număr prim. În această situație, avem nevoie să calculăm factorialele pînă la $n!$ și inversele lor modulare. Apoi, putem calcula în $\mathcal{O}(1)$ orice C_n^k.

Ca optimizare, putem calcula toate inversele factorialelor în $\mathcal{O}(n)$, apelînd o singură dată funcția de calcul al inversei. Secretul este să pornim de la n spre 0:

```
inv_fact[MAX_N] = inverse(fact[MAX_N]);
for (int i = MAX_N - 1; i >= 0; i--) {
  inv_fact[i] = (long long)inv_fact[i + 1] * (i + 1) % MOD;
}
```

14.3 Permutări cu repetiție

O mulțime de n elemente distincte are, desigur, $n!$ permutări. Cînd elementele nu sînt distincte (mulțimea este un multiset), atunci numărul de permutări este

$$\frac{n!}{n_1! \cdot n_2! \cdot \dots \cdot n_k!}$$

Unde $n_1, n_2, \dots, n_k$ sînt multiplicitățile elementelor din set și $n_1+n_2+\dots+n_k = n$. Vezi Wikipedia pentru detalii.

14.3.1 Aplicație

Dacă desfacem parantezele polinomului $(a + b + c + d)^{10}$ și simplificăm, care este coeficientul monomului $a^2bc^4d^3$?

14.4 Combinări cu repetiție

Problema clasică de combinări cu repetiție este: în cîte moduri putem așeza n obiecte (identice) în k cutii?

Metoda Stars and bars reduce problema la una de combinări obișnuite, iar soluția este C_{n+k-1}^{n}. Ea procedează astfel:

- Reprezintă fiecare obiect printr-o stea. Vor exista n stele.
- Separă fiecare două cutii printr-o bară. Vor exista $k - 1$ bare.
- Orice mod de a amesteca stelele și barele corespunde în mod unic unei soluții. De exemplu, reprezentarea `**|***||*` înseamnă că sînt 4 cutii, care conțin respectiv 2, 3, 0, 1 obiecte.

O altă interpretare este: încărcăm un robot cu cele n obiecte și îl lăsăm să viziteze cele k cutii. Stelele și barele reprezintă instrucțiuni în programul robotului. O stea înseamnă „lasă un obiect în cutia curentă”, iar o bară înseamnă „avansează la cutia următoare”.

14.4.1 Aplicații

- (Variantă) În cîte moduri putem așeza n obiecte în k cutii astfel încît fiecare cutie să conțină **cel puțin un obiect**?
- În cîte moduri îl putem scrie pe n ca sumă de k termeni numere naturale? Ordinea termenilor contează.
- (Variantă) În cîte moduri îl putem scrie pe n ca sumă de k termeni numere naturale **nenule**?
- Cîte soluții în numere naturale are ecuația $a + b + c + d = 10$?
- Cîte soluții în numere naturale are inecuația $a + b + c + d \leq 10$?
- Cîți termeni are polinomul $(a + b + c + d)^{10}$ după ce desfacem parantezele și simplificăm?

14.5 Numărarea permutărilor fără punct fix

Am întîlnit inițial problema ca puzzle de logică: n persoane vin la o petrecere și își pun pălăriile în cuier. La plecare, ei se întreabă: cîte moduri există de a redistribui pălăriile, cîte una fiecărei

persoane, astfel încît nimeni să nu plece acasă cu pălăria proprie? Vezi pagina Wikipedia pentru (muuuulte) informații.

Valoarea teoretică este numărul întreg cel mai apropiat de $\frac{n!}{e}$.

Formula de recurență este $D_n = (n-1) \cdot (D_{n-1} + D_{n-2})$. Vom urmări demonstrația.

Există și o formulă de calcul cu principiul includerii și excluderii. Vom urmări demonstrația. Implementarea clasică, cu inverse factoriale, merge, dar există și una mai simplă.

14.6 Numerele lui Catalan

Al n-lea număr al lui Catalan, C_n, indică numărul de șiruri de n paranteze deschise și n paranteze închise care sînt parantezate corect. Pagina Wikipedia listează numeroase probleme echivalente. Reținem în special problema: cîte căi există de la (0,0) la (n, n), mergînd doar la dreapta și în sus, care nu trec deasupra diagonalei?

Pe baza acestei formulări vom urmări demonstrația a 2-a (una dintre multele). Rezultă formula

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Demonstrația ne ajută să rezolvăm problema generalizată: cîte parantezări există care să înceapă obligatoriu cu $k \leq n$ paranteze deschise? Demonstrația este identică cu cea pentru problema originală:

- Pornind de la coordonatele $(k, 0)$ mai avem de făcut $n-k$ pași la dreapta și n în sus.
- Există o bijecție între căile rele și căile răsturnate.
- Căile răsturnate ajung la $(n-1, n+1)$, deci orice cale răsturnată mai are de făcut $n-k-1$ pași la dreapta și $n+1$ în sus.

De aceea, numărul total de căi pornind de la $(k, 0)$ este C^n_{2n-k}, iar numărul de căi rele este C^{n+1}_{2n-k}. Atunci numărul de căi bune este

$$\begin{aligned}
&\binom{2n-k}{n} - \binom{2n-k}{n+1} \\
&= \frac{(2n-k)!}{n!(n-k)!} - \frac{(2n-k)!}{(n+1)!(n-k-1)!} \\
&= \left[\frac{1}{n-k} - \frac{1}{n+1}\right] \cdot \frac{(2n-k)!}{n!(n-k-1)!} \\
&= \frac{k+1}{(n-k)(n+1)} \cdot \frac{(2n-k)!}{n!(n-k-1)!} \\
&= \frac{k+1}{n+1} \cdot \frac{1}{n-k} \cdot \frac{(2n-k)!}{n!(n-k-1)!} \\
&= \frac{k+1}{n+1} \cdot \binom{2n-k}{n}
\end{aligned}$$

Așadar,

$$C_n^{(k)} = \frac{k+1}{n+1}\binom{2n-k}{n}$$

.

14.7 Lema lui Burnside

Din secțiunea următoare, ultimele două probleme de pe CSES sînt rezolvabile cu această lemă. Din păcate, demonstrația este formulată doar în termeni de grupuri și nu am investit timpul necesar ca să o înțeleg. Dar principiul este următorul. Lema ne ajută să numărăm obiecte-unicat dintr-o colecție, în condițiile în care unele transformări (rotații, simetrii etc.) transformă un obiect într-altul. Procedăm astfel:

- Fie t numărul de transformări posibile.
- Fie C_r numărul de obiecte care rămîn nemodificate în urma unei transformări de tipul r.
- Atunci numărul de obiecte-unicat din colecție este $\frac{1}{t}\sum_{r=0}^{t-1} C_r$.

Pentru problema Counting Necklaces, definim transformarea r drept rotirea colierului cu r poziții. Atunci

- $C_0 = m^n$, deoarece putem asambla m^n coliere, iar dacă nu modificăm nimic... toate colierele rămîn nemodificate. 🤓
- $C_1 = m$, deoarece dacă un colier nu se modifică prin rotirea cu o poziție, atunci mărgeaua 1 are aceeași culoare cu mărgeaua 2, mărgeaua 2 are aceeași culoare cu mărgeaua 3 etc. Deci toate mărgelele au aceeași culoare.
- $C_2 = m$ dacă n este impar. De exemplu, pentru $n = 7$, mărgeaua 1 va urma pozițiile 3, 5, 7, 2, 4, 6, 1. Din nou, este necesar ca toate mărgelele să aibă aceeași culoare.
- $C_2 = m^2$ dacă n este par. Putem alege culorile primelor două mărgele, iar ele vor dicta culorile restului de mărgele.
- În general, $C_r = m^{\gcd(r,n)}$.

Pentru problema Counting Grids, raționamentul este foarte similar, dar numărul de rotații este fixat (4), iar numărul de libertăți pentru o transformare aleasă trebuie calculat. Pentru $r = 0$, numărul de posibilități este 2^{n^2} etc.

Suspectez că unele dintre probleme pot fi rezolvate și cu principiul includerii și excluderii.

14.8 Un puzzle

Iată și un puzzle care se rezolvă cu una dintre tehnicile de mai sus. Nu spunem care. 🙂

Soluția pe care o știu eu este inductivă. Fie Ana și Zaharia primul și respectiv ultimul pasager. Pentru $n = 2$, desigur probabilitatea ca Zaharia să-și găsească locul liber este $1/2$. Acum să presupunem că am rezolvat problema pentru $1, 2, \ldots, n$ și să o rezolvăm pentru $n + 1$. Ana poate alege 3 tipuri de loc: (a) propriul său loc, caz în care Zaharia are 100% șanse să-și ocupe locul; (b) locul lui Zaharia, caz în care Zaharia are 0 șanse să-și ocupe locul; (c) locul unui alt pasager, Xavier, caz în care putem reduce problema la cel mult n pasageri. Vor urca (posibil) un număr de pasageri diferiți de Xavier, care își găsesc locurile libere. Apoi va urca și Xavier.

În acest moment regăsim exact problema originală! Avem $m < n$ locuri libere, dintre care unul este al Anei, și locul lui Xavier care este ocupat. Efectiv, Xavier joacă în acest moment rolul Anei. Dacă Xavier ocupă locul Anei, ciclul se închide, iar Zaharia își va ocupa locul, similar cum în problema inițială, dacă Ana își ocupă propriul loc, ciclul se închide.

Media ponderată a celor trei probabilități este $1/2$.

14.9 Probleme

Primele 8 probleme sînt educaționale, clasice.

14.9.1 Problema Binomial Coefficients (CSES)

enunț • sursă

Este o problemă clasică de calcul al combinărilor. Pentru a calcula $C_n^k = \frac{n!}{k!(n-k)!}$ avem nevoie de inversele factorialelor. Repet că le putem calcula pe toate n în $\mathcal{O}(n)$.

14.9.2 Problema Creating Strings II (CSES)

enunț • sursă

Este o problemă clasică de permutări cu repetiție. Ca optimizare, cum rezolvăm problema fără memorie $\mathcal{O}(n)$, ci doar $\mathcal{O}(\Sigma)$?

14.9.3 Problema Distributing Apples (CSES)

enunț • sursă

Este o problemă clasică de combinări cu repetiție (*Stars and bars*). Atenție la implicația tacită: copiii sînt distincți (contează dacă Gigel primește un măr și Ionel două sau invers), dar merele sînt identice (nu contează care sînt cele două mere primite de Ionel).

14.9.4 Problema Christmas Party (CSES)

enunț • surse

Este o problemă clasică de numărare a permutărilor fără puncte fixe. Includ trei surse, două cu principiul includerii și excluderii și una cu formula de recurență.

14.9.5 Problema Bracket Sequences I (CSES)

enunț • sursă

Este o problemă elementară de calcul al numărului lui Catalan.

14.9.6 Problema Bracket Sequences II (CSES)

enunț • sursă

Probleme cere să calculăm numărul de parantezări cu prefix impus.

14.9.7 Problema Counting Necklaces (CSES)

enunț • sursă

Este o aplicație directă a Lemei lui Burnside.

14.9.8 Problema Counting Grids (CSES)

enunț • sursă

Și acestă problemă este o aplicație directă a Lemei lui Burnside.

14.9.9 Problema Number of Permutations (Codeforces)

enunț • sursă

Problema este directă, rezolvabilă cu principiul includerii și excluderii. Din toate cele $n!$ permutări,

- le scădem pe cele sortate după a;
- le scădem pe cele sortate după b;
- le adunăm pe cele sortate simultan după a și după b.

Sursa include și un exemplu de inginerie care vă poate ajuta. Ca să evităm duplicarea (sau triplicarea) codului, putem trimite funcții ca parametru.

14.9.10 Problema Counting Factorizations (Codeforces)

enunț • sursă

Soluția urmează editorialul, care este bine scris.

Să remarcăm că, dacă $f(m) = \{a_1, a_2, \ldots, a_{2n}\}$, atunci n dintre cele $2n$ numere citite la intrare trebuie să fie bazele, numere prime și distincte. Altfel problema nu are soluție. Sursa nu tratează special acest caz, dar el merita remarcat. Deoarece n dintre numere trebuie să fie prime, să citim cele n numere și să le clasificăm în prime și compuse (pe 1 îl tratăm ca număr compus). Putem folosi testul naiv de primalitate, fiind vorba despre doar 2 022 de numere mici.

Mai departe, să remarcăm că exponenții nu trebuie neapărat să fie distincți. De aceea, cele n numere rămase după alegerea bazelor pot fi permutate, dar cu eliminarea repetițiilor. De aceea, nu ne interesează numerele însele, ci frecvențele acestora. Să spunem că avem p numere prime distincte cu frecvențele $g_1, g_2, \ldots, g_p$ și c numere compuse distincte cu frecvențele $h_1, h_2, \ldots, h_c$.

Orice mod de a construi un m înseamnă că n dintre frecvențele numerelor prime vor scădea cu 1 (acelea sînt bazele alese). Să notăm frecvențele rămase cu $g'_1, g'_2, \ldots, g'_p$, unde $g'_i = g_i$ sau $g'_i = g_i - 1$. Celelalte numere rămase reprezintă cei n exponenți, iar frecvențele lor sînt $g'_1, g'_2, \ldots, g'_p, h_1, h_2, \ldots h_c$ (unele dintre aceste valori pot fi 0). Numărul de moduri de a permuta acești exponenți pentru a obține m-uri diferite este:

$$\frac{n!}{g'_1!g'_2!\ldots g'_p!h_1!h_2!\ldots h_c!}$$

Acum trebuie să însumăm această cantitate peste toate modalitățile de atribuire a valorilor g'_i. Scoatem factor comun expresia

$$T = \frac{n!}{h_1!h_2!\ldots h_c!}$$

și rămîne să însumăm rezultatele posibile pentru

$$\frac{1}{g'_1!g'_2!\ldots g'_p!}$$

Aici ne permitem să aplicăm o programare dinamică de complexitate $\mathcal{O}(n^2)$. Fie $d(i, j)$ suma termenilor de forma

$$\frac{1}{g'_1!g'_2!\ldots g'_i!}$$

unde exact j dintre factorii g' sînt **diferiți** de factorul g corespunzător (adică am ales j numere prime dintre primele i frecvențe). Atunci pe poziția i putem să alegem valoarea $g'_i = g_i$ sau valoarea $g'_i = g_i - 1$. Rezultă recurența:

$$d(i, j) = \frac{1}{g_i!} \cdot d(i-1, j) + \frac{1}{(g_i - 1)!} \cdot d(i-1, j-1)$$

Aici, primul termen semnifică că **nu** am folosit un număr prim din frecvența g_i, deci dintre primele $i - 1$ numere prime încă avem de ales j, iar al doilea termen semnifică că am folosit un număr prim din frecvența g_i, deci pînă la $i - 1$ mai avem de ales $j - 1$ numere prime. Răspunsul la problemă va fi $T \cdot d(p, n)$.

Putem implementa programarea dinamică folosind un singur vector.

14.9.11 Problema Monocarp and the Set (Codeforces)

enunț • sursă

Problema necesită o mică observație, dar sper să aveți deja acest reflex format. Este mult mai simplu să considerăm operațiile de la sfîrșit spre început. Iau naștere trei cazuri:

- Dacă ultimul caracter este `>`, atunci ultimul număr inserat este n. Reducem problema la una identică de mărime $n-1$.
- Similar, dacă ultimul caracter este `<`, atunci ultimul număr inserat este 1.
- Dacă ultimul caracter este `?`, atunci avem $n-2$ variante pentru ultimul număr inserat. Problema se reduce tot la una identică de mărime $n-1$, dar nu direct, ci printr-o renumerotare. De exemplu, dacă am alege să punem `3` pe ultima poziție, atunci mulțimea numerelor anterioare ar fi $\{1,2,4,5,6\}$, pe care însă o putem trata ca și pe $\{1,2,3,4,5\}$ fără a schimba natura problemei.

De aceea, este suficient să menținem produsul valorilor p pentru pozițiile p care conțin caracterul `?`. Aritmetica ține cu noi dacă indexăm șirul natural, de la 0. Cu alte cuvinte, poziția 0 de la intrare corespunde celui de-al doilea element din permutare. Apoi, ori de cîte ori poziția *pos* capătă valoarea `?`, înmulțim acest produs cu *pos*. Ori de cîte ori poziția *pos* capătă valoarea `<` sau `>` înmulțim produsul cu inversul lui *pos*.

Este nevoie de un caz special cînd `s[1] = '?'`, deoarece 0 nu are invers. În acel caz, răspunsul este oricum 0, deoarece după primul număr inserat, al doilea va fi obligatoriu fie minim (`<`), fie maxim (`>`).

14.9.12 Problema Devu and Flowers (Codeforces)

enunț • surse

Problema este o formă de *stars and bars*. În loc să punem s obiecte în n cutii, trebuie să le luăm, dar matematica este aceeași. În plus, cutiile au capacități. Spunem că o distribuție *saturează* o cutie de capacitate f_i dacă ia strict mai mult de f_i obiecte din cutie.

Limita $n \leq 20$ este un indiciu puternic că soluția va fi exponențială în numărul de cutii. De aici ajungem la următoarea soluție cu principiul includerii și excluderii:

- Numărăm toate cele C_{s+n-1}^{n-1} distribuții, care saturează 0 sau mai multe cutii.
- Scădem toate distribuțiile care saturează cel puțin o cutie.
- Adăugăm toate distribuțiile care saturează cel puțin două cutii.
- etc.

Odată ce fixăm mulțimea de cutii de saturat, cum anume le saturăm? Răspunsul este: plasăm $f_i + 1$ obiecte în fiecare cutie saturată i. Apoi distribuim restul de obiecte cu metoda *stars and bars* simplă, ceea ce, eventual, va adăuga și alte obiecte în cutiile saturate. Astfel ne asigurăm că generăm toate distribuțiile cu care am putea satura respectivele cutii.

Detalii de implementare

Toate combinările de calculat for avea forma C_x^{n-1}, unde n este constant, iar x este de ordinul a 10^{14}. De aceea, nu putem folosi formula naturală

$$\frac{x!}{(x-n+1)! \cdot (n-1)!}$$

întrucît nu putem calcula factoriale atît de mari. În schimb, putem folosi formula

$$\frac{x \cdot (x-1) \cdot \cdots \cdot (x-n+2)}{(n-1)!}$$

Ca optimizare, putem precalcula numitorul la începutul problemei. Astfel ia naștere prima sursă.

Putem chiar să definim constante pentru numitorul $1/(n-1)!$ pentru cele 20 de valori posibile pentru n. Sursa se scurtează, căci putem renunța la codul pentru calcularea inverselor. Totuși, sursa pierde din claritate.

Pentru elevii care se dau în vînt după programe scurte, indiferent de claritate, am scris și o ultimă sursă aliniată la standardele de lizibilitate ale lui Codeforces!

14.9.13 Problema Lengthening Sticks (Codeforces)

enunț • sursă

Soluție directă

Există o soluție bazată pe numărare directă, dar are multe cazuri particulare. Fie x, y și z cantitățile adăugate la a, b și respectiv c. Iterăm prin toate valorile lui z de la 0 la l și scriem inegalitățile în x și y care rezultă din inegalitatea triunghiului. De exemplu:

$$(a+x) < (b+y) + (c+z) \implies x - y < b + c - a + z$$

Astfel deducem limite superioare și inferioare pentru $x+y$ și $x-y$. Acestea delimitează un dreptunghi în planul xOy rotit cu 45°. Aria acestui dreptunghi este ușor de aflat. Din păcate, dreptunghiul mai trebuie decupat cu condițiile suplimentare $x \geq 0$ și $y \geq 0$, ceea ce duce la figuri complicate.

Soluție prin excludere

O altă abordare este să numărăm toate modurile de a distribui cel mult l unități, apoi să excludem tripletele ilegale.

Știm că putem distribui **exact** l unități în C_{l+2}^{l} moduri (este o problemă clasică de *stars and bars*). Rezultă că numărul de moduri de a distribui cel mult l unități este

$$C_{l+2}^{l} + C_{(l-1)+2}^{l-1} + \cdots + C_{2}^{0}$$

Această sumă este egală cu C_{l+3}^{3} (vezi ecuația 14.8). Dacă preferați, putem introduce un al patrulea băț fictiv care să preia surplusul din l nefolosit cu a, b și c. Ajungem la aceeași formulă. Ea poate fi calculată și cu un `for` în $\mathcal{O}(l)$.

Acum să trecem la excludere. Din nou, fie x, y și z cantitățile adăugate la a, b și respectiv c. Presupunem că dorim ca $a+x \geq b+y+c+z$ (vom proceda similar pentru $b+y$ și pentru $c+z$). Obținem inegalitățile:

$$\begin{cases} y+z & \leq l-x \\ y+z & \leq a+x-b-c \end{cases}$$

Fie $h = \min(l-x, a+x-b-c)$ limita superioară pentru $y+z$. Dacă h este negativ, nu există combinații valide pentru x și y. Dacă $h \geq 0$, atunci numărul de combinații valide este

$$0+1+\cdots+(h+1) = \frac{(h+1)(h+2)}{2}$$

14.9.14 Problema Shuffle (Codeforces)

enunț • surse

Ca la orice problemă de numărare, trebuie să ne asigurăm de două lucruri:

1. Că numărăm toate posibilitățile.
2. Că nu numărăm nicio posibilitate de mai multe ori.

Facem și observația universală că șirul dat trebuie să aibă cel puțin k caractere `1`.

Soluția în $\mathcal{O}(n^2)$

Este tentant să încercăm următoarea abordare: pentru fiecare subsecvență $W = [l, r]$ adăugăm la răspuns cantitatea C_{r-l+1}^{x}, unde x este numărul de caractere `1` din W. Dar este incorect, căci vom număra și unele permutări care lasă nemodificate porțiuni din W. Din această cauză, două subsecvențe parțial suprapuse, W și W', vor număra de două ori permutările care modifică doar zona comună $W \cap W'$.

Să contabilizăm subsecvențele într-un mod care elimină duplicatele. Fie $[l, r]$ o subsecvență amestecată astfel încît l și r sînt prima, respectiv ultima poziție modificată. Secvența merită luată în calcul dacă întrunește trei condiții:

1. Are lungime cel puțin 2. O secvență de lungime 1 nu poate fi amestecată. 🤔
2. Conține cel mult k cifre 1. Mai multe nu avem voie să selectăm, dar putem selecta mai puține (ne prefacem că amestecăm o secvență care include $[l, r]$, dar lasă nemodificate prefixul și

sufixul).

3. Secvența conține cifrele necesare pentru a nega pozițiile l și r. Dacă $s[l] = s[r] = 1$, secvența trebuie să conțină și două zerouri. Similar dacă $s[l] = s[r] = 0$. Iar dacă $s[l] \neq s[r]$, atunci nu mai punem alte condiții, căci întotdeauna putem interschimba $s[l]$ cu $s[r]$.

Odată fixate capetele, adăugăm la răspuns o cantitate de forma C_{x+y}^{x}.

Soluția în $\mathcal{O}(n)$

Iată un alt mod de a privi condiția (2). Cînd alegem o subsecvență care conține **exact** k de 1 și o permutăm, unele dintre permutări vor lăsa nemodificate un prefix și un sufix al subsecvenței. În acele situații este ca și cînd am permuta o subsecvență cu **cel mult** k de 1.

Pornind de la această observație, putem optimiza soluția de mai sus astfel. Să fixăm doar capătul stîng, l. Atunci capătul drept r poate fi oriunde începînd de la $l + 1$ pînă la cea de-a k-a apariție a unui 1, plus toate zerourile următoare. Apoi, impunem o singură condiție:

1. Undeva în $[l + 1, r]$ există un caracter diferit de $s[l]$. Altfel nu ne putem asigura că prima modificare este pe poziția l.

Odată fixat capătul stîng, orice amestecare distinctă a caracterelor din $[l, r]$, cîtă vreme garantează că îl modifică pe $s[l]$, este o soluție distinctă și numărată o singură dată. Dacă am grupa acum aceste amestecări după poziția ultimului caracter modificat, r, am regăsi subsecvențele $[l, r]$ din soluția anterioară.

Capitolul 15

Rangul permutărilor și al combinărilor

Programa de clasa a 10-a conține și *Determinarea numărului de ordine pentru elementele combinatoriale*. Ceea ce poate însemna orice din vreo 10 subiecte diferite. 🙂

Vom discuta despre permutări și combinări. Aceleași principii se aplică și la alte elemente combinatoriale (aranjamente, submulțimi etc.).

15.1 Definiții

Rangul unui obiect este numărul lui de ordine în lista ordonată a tuturor obiectelor de acel tip. Exemple:

- Rangul permutării (3 1 2) este 4 (pornind de la 0). Cele 4 permutări anterioare, în ordine, sînt (1 2 3), (1 3 2), (2 1 3) și (2 3 1).
- Rangul combinării de 5 luate cîte 3 (1 4 5) este 5. Combinările anterioare sînt (1 2 3), (1 2 4), (1 2 5), (1 3 4), (1 3 5).

Vom folosi termenii englezești *ranking* pentru aflarea rangului unui obiect dat și *unranking* pentru aflarea obiectului dîndu-i-se rangul. Ordinea lexicografică nu este unica posibilă. De exemplu, problema Arbperm2 specifică altă ordine, iar în problemele Inversion Sort și Four Chips sîntem liberi să ne alegem orice ordine, singurul scop fiind să mapăm permutări/combinări la întregi ca să ținem evidența obiectelor deja întîlnite.

Următorul obiect al unui obiect dat este obiectul de rang cu 1 mai mare. Exemple:

- Permutarea care urmează după (3 1 2) este (3 2 1).
- Combinarea de 5 luate cîte 3 care urmează după (1 4 5) este (2 3 4).

Aveți deja toate uneltele ca să calculați ad-hoc toate aceste informații, dar este bine să ne familiarizăm cu unele subtilități.

15.2 Rangul permutărilor

15.2.1 Următoarea permutare

Fie permutările

$$P = (2\ 5\ 4\ 8\ 7\ 6\ 3\ 1\ 0)$$
$$next(P) = (2\ 5\ 6\ 0\ 1\ 3\ 4\ 7\ 8)$$

Procedura pentru a trece de la P la $next(P)$ este simplă:

1. Găsește cel mai lung sufix descrescător, (8 7 6 3 1 0).
2. Schimbă elementul anterior sufixului, 4, cu cel mai mic element din sufix mai mare decît acesta (6).
3. Răstoarnă sufixul.

Complexitatea amortizată pentru mai multe apeluri este $\mathcal{O}(1)$ (de ce?). Cred că funcția din STL `std::next_permutation()` aplică fix acest algoritm. Ea garantează că face cel mult $n/2$ interschimbări (de ce?).

15.2.2 *Ranking*

Deoarece $21! > 2^{64}$, calculul complet este de interes doar pentru permutări de cel mult 20 de elemente (problema Inversion Sort). În rest, trebuie să operați cu ranguri fără a le calcula efectiv.

Calculați manual rangul pe exemplul de mai sus (103679)! Vom deduce că întrebarea esențială este: pentru o poziție dată k, cîte valori mai mici decît $P(k)$ se află pe pozițiile anterioare?

15.2.3 *Unranking*

Întrebarea de bază la *unranking* este cea inversă: care este valoarea potrivită de așezat la poziția curentă? Ea va fi a k-a valoare cu proprietatea că numărul de permutări care încep cu primele $k-1$ valori însumate nu depășește rangul dorit, dar dacă includem și a k-a valoare depășim rangul.

Pentru ordinea naturală (lexicografică), algoritmul se reduce la o împărțire la $(n-k)!$ și la întrebarea: care este a k-a cea mai mică valoare încă nescrisă la stînga poziției curente?

Calculați permutarea din rangul de mai sus!

15.2.4 Implementare

Putem răspunde la aceste întrebări în timp total $\mathcal{O}(n \log n)$ folosind un AIB sau alte structuri cu interogare în timp logaritmic. Atenție însă, pentru n mic implementarea naivă (pătratică) va fi

mai rapidă. În schimb, putem cîștiga timp folosind măști de biți.

Pentru *unranking*, trebuie să răspundem rapid la întrebarea: care este al k-lea bit zero dintr-o mască de biți? O variantă este să precalculăm acest răspuns pentru toate măștile. Dar pentru $n = 20$, aceasta va necesita memorie $2^n \cdot n = 20$ MB, ceea ce nu este ideal. Există și o operație primitivă, PDEP, dar ea nu este implementată pe toate arhitecturile și poate funcționa chiar mai lent decît căutarea naivă.

Vom citi implementările acestor algoritmi. Iată și un *benchmark* pentru $n = 20$:

```
[cata@neil combinatorics]$ ./permutation-rank
inițializare: 942 ms pentru 10000000 operații (10 Mops/sec)
--
rank naiv: 890 ms pentru 10000000 operații (11 Mops/sec)
rank cu AIB: 1984 ms pentru 10000000 operații (5 Mops/sec)
rank cu popcount: 180 ms pentru 10000000 operații (55 Mops/sec)
--
unrank naiv: 3424 ms pentru 10000000 operații (2 Mops/sec)
unrank cu AIB: 4016 ms pentru 10000000 operații (2 Mops/sec)
unrank cu popcount și tabel: 1195 ms pentru 10000000 operații (8 Mops/sec)
unrank cu popcount și pdep: 5739 ms pentru 10000000 operații (1 Mops/sec)
```

15.3 Rangul combinărilor

În cazul combinărilor, discuțiile sînt mai nuanțate. Problemele de rang au valori mici pentru k, dar nu neapărat și pentru n. Bunăoară, pentru $k = 3$, valoarea maximă pentru n astfel încît $C_n^k < 2^{64}$ (adică rangul să încapă pe `unsigned long long`) este circa 2.000.000, pe cînd pentru $k = 20$, valoarea maximă pentru n este 86.

15.3.1 Următoarea combinare

Să analizăm un exemplu. Fie combinarea de 20 luate cîte 6 (numerotate de la 0):

$$C = (4\ 7\ 12\ 17\ 18\ 19)$$
$$next(C) = (4\ 7\ 13\ 14\ 15\ 16)$$

Rezultă că algoritmul este:

1. Găsește ultima poziție i pentru care $C[i] - i < n - k$.
2. Dacă acea poziție nu există, atunci C este ultima lexicografic.
3. Altfel, incrementează $C[i]$ și pe toate pozițiile $j > i$ atribuie $C[j] = C[j-1] + 1$.

Și acest algoritm are complexitate amortizată $\mathcal{O}(1)$ (de ce?).

15.3.2 *Ranking* lexicografic

Să considerăm combinarea de 4 elemente dintr-un set de 20 (numerotate începînd de la 0):

$$C = (4\ 7\ 13\ 16)$$

Dacă dorim să îi calculăm rangul lexicografic, atunci facem următoarele observații:

- Există $C_{19}^3 = 969$ combinări care încep cu 0, deoarece dacă îl alegem pe 0 mai rămîn 19 alte elemente din care trebuie să alegem 3.
- Există $C_{18}^3 = 816$ combinări care încep cu 1.
- Există $C_{17}^3 = 680$ combinări care încep cu 2.
- Există $C_{16}^3 = 560$ combinări care încep cu 3.
- Toate combinările de mai sus sînt mai mici lexicografic decît P.
- Există $C_{14}^2 = 91$ combinări care încep cu 4 5, deoarece mai rămîn valorile 6-19 din care trebuie să alegem două.
- Există $C_{13}^2 = 78$ combinări care încep cu 4 6.
- Există $C_{11}^1 = 11$ combinări care încep cu 4 7 8.
- ...
- Există $C_7^1 = 7$ combinări care încep cu 4 7 12.
- Există $C_5^0 = 1$ combinări care încep cu 4 7 13 14.
- Există $C_4^0 = 1$ combinări care încep cu 4 7 13 15.

Rangul lui (4 7 13 16) este suma combinărilor de mai sus, adică 3241.

Algoritmul funcționează în $\mathcal{O}(n)$. El poate fi îmbunătățit pînă la $\mathcal{O}(k)$ cu formula

$$\sum_{m=0}^{n} C_m^k = C_{n+1}^{k+1}$$

15.3.3 *Unranking* lexicografic

Calculați combinarea din rangul de mai sus! Algoritmul naiv funcționează în $\mathcal{O}(n+k)$ urmînd exact pașii de mai sus. De exemplu, va scădea succesiv valorile 919, 816, 680 și 560 din rangul dat și va constata că nu mai poate scădea încă $C_{15}^3 = 455$, deci primul element trebuie să fie 4.

Putem reduce complexitatea la $\mathcal{O}(k+\log n)$ folosind sume parțiale și căutări binare. Nu sînt sigur dacă există vreun algoritm în $\mathcal{O}(k)$.

15.3.4 *Ranking* și *unranking* colexicografic

O altă ordine notabilă este ordinea colexicografică. În acestă ordine, combinările sînt ordonate de la dreapta către stînga. Ne ajută mai mult să considerăm că elementele au valori între 0 și $n-1$, iar pozițiile între 0 și $k-1$, ca în tabelul următor:

rang	combinare	proveniență
0	(0 1 2)	$C_0^1 + C_1^2 + C_2^3$
1	(0 1 3)	$C_0^1 + C_1^2 + C_3^3$
2	(0 2 3)	$C_0^1 + C_2^2 + C_3^3$
3	(1 2 3)	$C_1^1 + C_2^2 + C_3^3$
4	(0 1 4)	$C_0^1 + C_1^2 + C_4^3$
5	(0 2 4)	$C_0^1 + C_2^2 + C_4^3$
6	(1 2 4)	$C_1^1 + C_2^2 + C_4^3$
7	(0 3 4)	$C_0^1 + C_3^2 + C_4^3$
8	(1 3 4)	$C_1^1 + C_3^2 + C_4^3$
9	(2 3 4)	$C_2^1 + C_3^2 + C_4^3$
10	(0 1 5)	$C_0^1 + C_1^2 + C_5^3$
11	(0 2 5)	$C_0^1 + C_2^2 + C_5^3$
12	(1 2 5)	$C_1^1 + C_2^2 + C_5^3$
13	(0 3 5)	$C_0^1 + C_3^2 + C_5^3$
14	(1 3 5)	$C_1^1 + C_3^2 + C_5^3$
15	(2 3 5)	$C_2^1 + C_3^2 + C_5^3$
16	(0 4 5)	$C_0^1 + C_4^2 + C_5^3$
17	(1 4 5)	$C_1^1 + C_4^2 + C_5^3$
18	(2 4 5)	$C_2^1 + C_4^2 + C_5^3$
19	(3 4 5)	$C_3^1 + C_4^2 + C_5^3$

Tabela 15.1: Combinări de $n = 6$ luate cîte $k = 3$, ordonate colexicografic

Calculul rangului unei combinări este facil în $\mathcal{O}(k)$. Să luăm exemplul combinării $X = (2\ 3\ 5)$ și să procedăm de la dreapta la stînga.

- Toate combinările terminate cu 2, 3 sau 4 vin înaintea lui X. Numărul lor este exact numărul de combinări care folosesc doar valorile 0, 1, 2, 3 sau 4, deci C_5^3.
- Similar, combinările terminate cu $(a\ 5)$, unde $a < 3$, vin înaintea lui X. Numărul lor este C_3^2.
- În sfîrșit, combinările terminate cu $(b\ 3\ 5)$, unde $b < 2$, vin înaintea lui X. Numărul lor este C_2^1. Astfel obținem rangul lui X, $10 + 3 + 2 = 15$.

```
u64 rank_colex(int* c) {
  u64 result = 0;
  for (int i = 0; i < K; i++) {
    result += choose[c[i]][i + 1];
  }
  return result;
}
```

Și partea de *unranking* este simplă în $\mathcal{O}(n)$. Să pornim cu rangul 5. Vom completa combinarea X de la dreapta la stînga.

- Ar putea ultima valoare să fie 5? Știm că există $C_5^3 = 10$ combinări care folosesc doar valorile 0-4 și toate ar trebui să vină înaintea lui X. Dar $10 > 5$, deci ultima valoare **nu**

poate fi 5.

- Ar putea ultima valoare să fie 4? Știm că există $C_4^3 = 4$ combinări care folosesc doar valorile 0-3 și toate ar trebui să vină înaintea lui X. Deoarece $4 \leq 5$, ultima valoare este 4. Scădem 4 din rang și dorim să aflăm permutarea de rang 1 dintre cele terminate cu 4.
- Ar putea penultima valoare să fie 3? Știm că există $C_3^2 = 3$ moduri de a lua două din valorile 0-2 și toate ar trebui să vină înaintea lui X. Deoarece $3 > 1$, penultima valoare **nu** poate fi 3.
- Ar putea penultima valoare să fie 2? Știm că există $C_2^2 = 1$ mod de a lua două din valorile 0-1 și el ar trebui să vină înaintea lui X. Deoarece $1 \leq 1$, penultima valoare este 2. Scădem 1 din rang și dorim să aflăm permutarea de rang 0 dintre cele terminate cu 2 4.
- Similar raționăm că prima valoare nu poate fi 1, dar va fi 0. Aflăm că $X = (0\ 2\ 4)$.

```
void unrank_colex(u64 rank, int* dest) {
  int n = N - 1;
  for (int k = K; k; k--) {
    while (choose[n][k] > rank) {
      n--;
    }
    rank -= choose[n][k];
    dest[k - 1] = n;
  }
}
```

Implementarea completă este anexată.

15.4 Probleme

15.4.1 Problema Misha and Permutation Summation (Codeforces)

enunț • sursă

Este o problemă directă de calcul al rangului unei permutări (*ranking*) și de aflare a permutării cu un rang dat (*unranking*). Singura neplăcere este că nu putem stoca rangul pe un tip de date scalar. Îl vom stoca pe un vector în baza factorial (adică poziția i poate lua valori între 0 și i).

15.4.2 Problema Inversion Sort (SPOJ)

enunț • sursă

Problema este una de BFS. Trebuie să aflăm distanțele (în număr de operații) de la *abcdefghij* la toate celelalte permutări ale șirului.

Este nevoie de o observație ca să rezolvăm cazul cînd permutarea de pornire este diferită de *abcdefghij*. Putem rescrie permutarea de pornire, redenumind literele ca să ajungem la *abcdefghij*, și aplicăm aceeași redenumire și permutării-destinație. Atunci soluția trebuie să fie identică, doar

cu alte nume pentru litere.

Cum tratăm BFS-ul? O variantă este să stocăm un `map` de la șiruri la distanțe. Dar cred că această metodă va fi lentă. Este mai eficient să mapăm fiecare permutare la rangul său. Astfel ne ajunge un vector de $10! \approx 3\,600\,000$ octeți pentru distanțe.

Noi nu știm să facem mutări pe ranguri, ci doar pe permutări. De aceea, avem două variante:

1. Ținem coada BFS doar de ranguri. La scoaterea din coadă, apelăm *unrank* ca să recuperăm permutarea, pe care facem mutări.
2. Ținem coada BFS de perechi (permutare, rang). Va fi probabil mai rapid și nu mai este nevoie să scriem funcția *unrank*.

Dat fiind că n este fixat și mic, putem calcula rangul în $\mathcal{O}(1)$ per poziție cu măști de biți.

Am încercat și 3 optimizări:

1. v2 calculează mai eficient mutările, făcînd o singură transpoziție per mutare. Efectul este mic, semn că apelurile la *rank* sînt predominante.
2. v3 calculează *popcount* cu tabel predefinit. Timpul de rulare scade sub jumătate. Aceasta este varianta pe care am inclus-o în anexă.
3. v4 operează pe cifre în loc de litere. Timpul de rulare scade cam cu 5%.

15.4.3 Problema Four Chips (SPOJ)

enunț • sursă

Problema este similară cu Invesort, în sensul că necesită un BFS de la starea inițială. Ca și acolo, putem evita partea de *unranking*.

Este nevoie de ceva mai multă logistică, în special la generarea mutărilor. Există cel mult 20 de mutări posibile din fiecare poziție: 4 piese × (2 deplasări + 3 salturi). Ca să nu includ acel cod în bucla principală a BFS-ului, am decis să parametrizez generatorul de mutări, ca să accepte un parametru între 0 și 19.

Putem discuta despre diverse idei de optimizare a generatorului de mutări. De exemplu, putem simplifica testul dacă o celulă este goală (funcția `empty()`) dacă stocăm și un bitset de 70 de biți pentru ocuparea tablei.

15.4.4 Problema Long Permutation (Codeforces)

enunț • sursă

Problema este relativ clasică de *ranking*. Este nevoie doar de următoarea observație teoretică. Pot exista cel mult $2 \cdot 10^5$ operații de avansare a rangului cu cel mult 10^5 fiecare, deci rangul maxim la care putem ajunge este $2 \cdot 10^{10}$.

De cîte elemente este nevoie ca să atingem acest rang? Există $14! \approx 8,7 \cdot 10^{10}$ moduri de a permuta ultimele 14 elemente, deci primele $n - 14$ elemente nu se modifică niciodată. De aceea,

spargem permutarea în:

1. Primele $n-14$ elemente (nu mai puțin de 0), care vor avea întotdeauna valorile $1, 2, \ldots, n-14$.
2. Restul, pe care chiar le vom permuta. Pentru simplitate, le putem renumerota de la 0, iar la operațiile de sumă parțială ne vom aminti să adunăm la fiecare cîte $n-14$.

Cele două operații se reduc la:

1. Menține rangul curent al permutării (suma x-urilor de pînă acum). La fiecare nou x, generăm permutarea de 14 elemente.
2. Descompune intervalul dat într-unul inclus în prefixul imutabil și unul inclus în permutarea generată.

Pentru claritate, sursa încapsulează separat permutarea „clasică” și permutarea „din bucăți”.

15.4.5 Problema Arbperm2 (NerdArena)

enunț • sursă

Remarcăm că definiția rangului este semnificativ diferită, dar rangul este încă ușor de calculat. Teoretic problema constă tot din *ranking* – adaugă k – *unranking*. Dar rangul depășește rapid `long long`.

Rezolvarea greșită

Să socotim rangurile de la dreapta spre stînga. Atunci vrem să calculăm permutarea cu rang cu k mai mic.

Începem să scoatem elemente din permutare și să calculăm rangul. Elementul n va contribui cu $0 \le x < n$, în funcție de poziția sa. Elementul $n-1$ va contribui cu $y \cdot n$, unde $0 \le y < n-1$. De exemplu, elementul 3 poate contribui cu 0 (pentru cea mai din dreapta grupă de 4 permutări) sau cu 4 pentru a doua cea mai din dreapta grupă de 4. Elementul $n-2$ va contribui cu $z \cdot n(n-1)$, unde $0 \le z < n-2$. De exemplu, elementul 2 va contribui cu 0 pentru cele 12 permutări din dreapta și cu 12 pentru cele 12 permutări din stînga.

Cînd ajungem la un rang cel puțin egal cu k, ne oprim din eliminat. Scădem k din rang și reinserăm elementele în ordinea impusă de rangul rămas. Practic, urcăm în arbore pînă cînd nodul curent subîntinde cel puțin k frunze la dreapta.

Nici măcar nu este nevoie de structuri de date pe vectori. Putem face operațiile de eliminare/inserare în vector în $\mathcal{O}(n)$, căci sînt puține. Această sursă ia 100p... dar de fapt greșește. Doar că testele din problema originală (Arbperm) nu expun cazul special care trebuie tratat. De aceea am creat problema Arbperm2, cu ultimul test modificat.

Unde este greșeala?

Greșeala apare pe testul:

```
100000 1
2 1 3 4 5 6 7 8 ... 99999 100000
```

Răspunsul este:

```
100000 99999 ... 8 7 6 5 4 3 1 2
```

Exemplul arată că pot exista $\mathcal{O}(n)$ elemente care migrează și pe care soluția de mai sus trebuie să le elimine și să le reinsereze. Mai rău, trebuie să urcăm în arbore pînă cînd subîntindem 100!/2 noduri, ceea ce depășește reprezentarea.

Rezolvarea corectă

Facem următoarea observație. Valoarea n migrează circular prin pozițiile $0, 1, \ldots, n-1$, începînd de la poziția pe care o are la intrare. Deci îi putem afla poziția finală. Pentru exemplul 2 din enunț, valoarea 4 începe de pe poziția 2 și ciclează prin pozițiile 3, 0, 1, 2, 3, 0, 1.

Mult mai interesantă este valoarea $n-1$. Acesta migrează cu o poziție spre dreapta ori de cîte ori valoarea n ajunge pe prima poziție. De asemenea, pozițiile acestei valori trebuie calculate între 0 și $n-2$, ignorînd aparițiile lui n.

Pe cazul general al unei valori v, aflată pe poziția inițială $a[v]$:

- Poziția finală $b[v]$ este dată de $a[v]$ și numărul de avansuri modulo v.
- $a[v]$ și $b[v]$ țin cont doar de elementele mai mici decît v.
- Numărul de avansuri al valorii $v-1$ este dat de numărul de treceri peste 0 ale lui v.

Putem găsi pozițiile inițiale $a[v]$ ale fiecărei valori cu un AIB. Apoi calculăm în mod banal pozițiile finale $b[v]$. Din acestea putem recupera permutarea finală. Procesăm valorile v de la mare la mic. Valoarea n stă fix pe poziția $b[n]$. Apoi, fiecare valoare x stă pe a $b[x]$-a poziție încă liberă din permutarea finală.

15.4.6 Problema Hipersimetrie (ONI 2019 clasele 11-12)

enunț • sursă

În acest capitol ne-am concentrat pe rangul permutărilor și al combinărilor, dar iată și o problemă de *unranking* pe matrice / pe șiruri binare. Este genul meu de problemă: 😁 un experiment de gîndire interesant care duce la un program clar și relativ scurt.

O imagine

Aceasta este figura pe care am folosit-o la rezolvarea problemei, redesenată pentru claritate, dar altfel nemodificată. Mi se pare că pe baza acestei figuri, codul se scrie singur (Hans Niemann dă din cap aprobator). Vă recomand și vouă ca, dacă ideea problemei nu este clară, să alegeți exemple mai mari decît $n = 3$ sau $n = 4$. Investiți timp într-un exemplu mare, căci el va simplifica înțelegerea algoritmului.

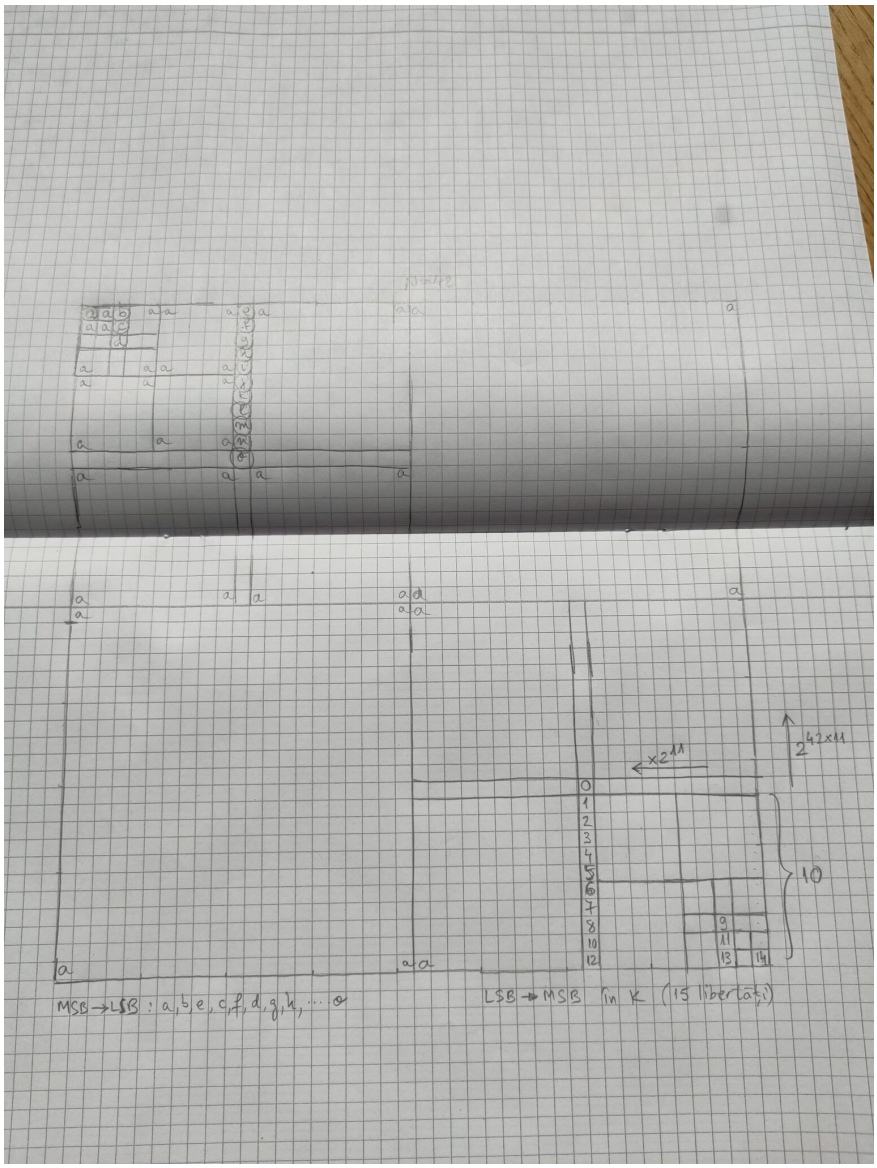


Figura 15.1: O imagine pentru $n = 42$.

Apropo de acest efort, menționez un citat favorit al meu:

> People said I did the impossible, but that's wrong: I merely did something so boring that nobody else had been willing to do it. – *Jacob Kaplan-Moss, creatorul Django*

Este trist că o olimpiadă se poate numi „de informatică" în timp ce concurenților nu le este permis accesul în sală cu foi de matematică la discreție. Mai degrabă aș concura fără o mînă decît fără hîrtie de matematică.

Și acum, să vedem ce învățăm din figură.

Algoritmul teoretic

Dacă desenăm cîteva simboluri și urmărim unde se propagă ele prin hipersimetrie, observăm că:

- O matrice de dimensiune pară este formată din patru sferturi identice.
- O matrice impară este formată din patru sferturi identice și o cruce cu patru brațe identice.

Începem să ne dăm seama că sîntem liberi să alegem niște biți, iar restul vor fi impuși. De aici rezultă că numărul de matrice hipersimetrice va fi o putere a lui 2, acesta fiind și motivul pentru care îl primim pe k în baza 2. Fie $L(n)$ numărul de libertăți al unei matrici, adică numărul de biți independenți pe care îi putem alege. Atunci:

- $L(2k) = L(k)$ deoarece, după ce completăm un sfert, celelalte trei sînt impuse.
- $L(2k+1) = L(k) + k + 1$ deoarece, după ce completăm un sfert, mai putem da valori independente unia dintre brațele crucii (inclusiv centrul).

Astfel am ales pentru figura de mai sus $n = 42$, care ilustrează ambele cazuri. Există 15 libertăți, pe care le-am denumit alfabetic, și deci există 2^{15} matrice hipersimetrice binare. Șirul de la intrare va avea cel mult 15 biți. A k-a matrice (*0-based*) este cea în care scriem biții lui k în locul literelor, apoi îi propagăm prin hipersimetrie. Deoarece în enunț k pornește de la 1, trebuie să îl decrementăm înainte de a începe.

Problema ne cere să tratăm matricea rezultată ca pe un număr cu n^2 biți, citit de la stînga la dreapta și de sus în jos.

Detalii de implementare

Am preferat să operez în colțul din dreapta-jos al matricei (acolo unde sînt biții mai puțin semnificativi). Astfel restul matricei poate fi completat prin înmulțiri, nu prin împărțiri. Am reprezentat și acolo cele 15 libertăți, de data aceasta cu valori de la 0 la 14 corespunzătoare biților lui k (unde 0 = cel mai puțin semnificativ bit).

Se vede că biții sînt distribuiți pe $\mathcal{O}(\log n)$ coloane, unde fiecare coloană este cel mult jumătate din precedenta. Mai vedem că vom avea nevoie de coloane de biți, dar prioritatea lor crește pe linie, deci coloanele **nu** constau din subsecvențe contigue din k de la intrare, ci pe sărite.

Nu în ultimul rînd, suma înălțimilor coloanelor poate fi de sute de milioane. Cum $k \leq 1\,000\,000$, rezultă că o mare parte dintre biți vor fi implicit 0. Este important ca programul să nu-i proceseze decît pe cei de la intrare. Putem face asta dacă îl parcurgem pe k de la LSB spre MSB, iar libertățile de la centru spre periferie. Ne oprim cînd epuizăm biții lui k, deoarece restul ar fi zerouri, care nu afectează valoarea matricei.

Așadar, am izolat într-o clasă numită `bit_stream` codul care

- primește dimensiunea n;
- iterează prin biții lui k și îi distribuie pe coloanele potrivite;
- la cerere, returnează una cîte una coloanele în ordinea descrescătoare a înălțimii (astfel le va cere codul principal).

Recurența

Ne dorim ca funcția recursivă `compute_matrix(int s)` să genereze matricea de $s \times s$ din colțul dreapta-jos al matricei totale. Funcția returnează valoarea matricei generate. Atenție: matricea nu va ocupa $s \times s$ biți compacți, ci fiecare s biți vor ocupa porțiunea cel mai puțin semnificativă

dintr-un grup de n biți. De exemplu, o matrice de 3×3 va ocupa 9 biți dintre primii $3n$, și anume biții $0, 1, 2, n, n+1, n+2, 2n, 2n+1, 2n+2$.

Procedăm astfel deoarece ne va fi mult mai ușor să translatăm un colț în celelalte trei colțuri. O matrice pară de mărime s trebuie copiată astfel:

- La stînga cu s coloane, prin înmulțirea valorii sale cu 2^s.
- În sus cu s linii, prin înmulțirea valorii sale cu 2^{sn}.
- În sus-stînga cu s linii și s coloane, prin înmulțirea cu 2^{sn+s}.

Așadar, dacă notăm cu V_s valoarea matricei de latură s, atunci

$$V_{2s} = V_s \cdot (1 + 2^s) \cdot (1 + 2^{sn})$$

Același raționament se aplică și pentru matricele impare, dar pentru acestea trebuie să inserăm și crucea centrală. Este acceptabil să facem efort $\mathcal{O}(1)$ per element, căci efortul total nu va depăși $\mathcal{O}(k)$ (din nou, dacă avem grijă să nu cerem decît primii k biți). Iar cu efort $\mathcal{O}(1)$ per element, completarea crucii este ușoară:

- Centrul trebuie amplasat într-un singur loc.
- Celelalte elemente trebuie amplasate de cîte patru ori.

Amplasarea unui bit la linia i și coloana j (numărînd din colțul de jos-dreapta) înseamnă pur și simplu creșterea valorii matricei cu 2^{ni+j}.

Puterile lui 2

O ultimă componentă necesară este posibilitatea de a calcula puteri mari ale lui 2. Din secțiunea de mai sus vedem că exponenții pot ajunge la n^2, adică 10^{18}. Știm să facem asta cu exponențiere binară, dar se poate și în $\mathcal{O}(1)$ (vezi capitolul TODO). Reiau aici ideea pe scurt.

În primul rînd, deoarece rezultatul este modulo $M = 1\,000\,000\,007$, putem aplica mica teoremă a lui Fermat, conform căreia $2^e \equiv 2^{e \bmod (M-1)} \pmod M$. Deci am redus exponenții la valori mai abordabile.

Apoi aplicăm descompunerea în radical. Calculăm doi vectori:

- $2^0, 2^1, 2^2, \ldots, 2^{32767}$ și
- $2^0, 2^{1 \cdot 32768}, 2^{2 \cdot 32768}, \ldots, 2^{32767 \cdot 32768}$

Astfel putem scrie $2^e = 2^{(e/32768) \cdot 32768} \cdot 2^{e \bmod 32768}$, pe care îl putem calcula în $\mathcal{O}(1)$ operații.

15.4.7 Problema Trim (Baraj ONI 2023)

enunț • sursă

Nu am citit editorialul, dar este posibil ca la această problemă Comisia să fi scăpat din vedere o soluție. Cel puțin pe Kilonova, la o limită de 0,9 secunde, există multe soluții în sub 0,1 secunde.

Pare natural să sortăm interogările, apoi să începem să emitem șirul de biți. Doar că nu îl vom emite unul cîte unul, ci în tranșe mai mari. La fiecare tranșă emisă, calculăm răspunsurile pentru interogările aflate în acea tranșă.

Fie $n = 13$. Atunci vom emite, în această ordine, numerele cu 1, 2, 3, ..., 13 biți de 1. Să considerăm numerele cu 7 biți de 1. Cel de-al doilea criteriu este valoarea numărului. La rîndul ei, valoarea numărului este dată în primul rînd de lungimea numărului: un număr cu 7 biți de 1, de lungime totală 10, va fi cuprins între 513 și 1023 și va fi întotdeauna mai mic decît un număr cu 7 biți de 1, de lungime totală 11, care va fi cuprins între 1025 și 2047.

Abordarea mea denumește **tranșă** o astfel de combinație de (număr de biți 1) $\times$ (lățimea numerelor). Algoritmul emite o tranșă de lățime w biți în $\mathcal{O}(1 + w \cdot \text{nr\_interogări})$. Deoarece numărul de tranșe este $\mathcal{O}(n^2)$, va rezulta o complexitate totală de $\mathcal{O}(n^2 + qn)$.

Să considerăm tranșa de numere cu 7 biți 1 de lățime 10. Cum arată această tranșă?

1. Toate numerele au forma 1*bbbbbbbb*1, unde exact 5 dintre cei 8 biți centrali au valoarea 1.
2. Fiecare număr are o multiplicitate: el apare de 4 ori consecutiv, deoarece operăm pe numere care aveau inițial 13 biți. Numărul de 10 biți poate fi completat cu 3 zerouri la stînga sau la dreapta în 4 moduri distincte.
3. Există C_8^5 grupe de cîte 4 numere, ordonate conform ordinii lexicografice a combinărilor.
4. Lungimea totală a tranșei este $C_8^5 \cdot 4 \cdot 10$ biți.

Piesa finală din puzzle este să tipărim al p-lea bit din tranșă. Calculînd cîtul împărțirii lui p la 40 știm în zona cărei combinări se află el, iar calculînd restul împărțirii lui p la 10 știm al cîtelea bit din combinare îl dorim. Aceasta este o operație clasică de *combination unranking*, implementată în funcția `find_bit()`.

Partea V

Geometrie computațională

Capitolul 16

Elemente de bază

Acest capitol se axează pe formule pentru

- puncte;
- drepte;
- teste de orientare;
- arii;
- intersecții.

Formulele de trigonometrie de la orele de matematică vă vor fi de ajutor!

16.1 Principii de implementare

Evitați împărțirile! Aceasta vă scapă de diverse erori de precizie și cazuri-limită (împărțiri prin zero). Ca bonus, codul devine mai rapid.

Luați în calcul cazurile particulare: unghiuri de 90°, drepte paralele, puncte coliniare etc. De multe ori, programul le tratează fără cod suplimentar.

Preferați comparațiile în locul valorilor exacte. Un exemplu frecvent: ne pasă mai mult dacă C se află pe dreapta AB, la stînga sau la dreapta ei. Ne pasă mai puțin care este valoarea exactă a unghiului ABC.

Puteți găsi multe tutoriale teoretice utile, de exemplu pe CP Algorithms sau pe UCF.

16.2 Puncte și vectori

Vom folosi interschimbabil următoarele trei noțiuni:

- Punctul $P = (x_P, y_P)$.
- Vectorul $\vec{p}$ de la origine la P.
- Matricea de 2×1 $\begin{pmatrix} x_P \\ y_P \end{pmatrix}$.

Notăm cu $|\vec{p}|$ sau pur și simplu cu p lungimea vectorului.

16.2.1 Adunare, înmulțire cu o constantă

Vectorii pot fi adunați între ei și scalați cu o constantă, ceea ce se traduce prin adunarea și înmulțirea cu o constantă a matricelor.

16.2.2 Produsul scalar (engl. *dot product*)

Se numește așa pentru că rezultatul este o valoare scalară. Are două definiții echivalente (demonstrați!):

- $\vec{p} \cdot \vec{q} = x_p x_q + y_p y_q$
- $\vec{p} \cdot \vec{q} = p \cdot q \cdot \cos\theta$, unde θ este unghiul dintre vectori.

Motivație (de exemplu): deplasarea unei greutăți pe orizontală cu o forță oblică.

Produsul scalar este comutativ și distributiv.

Aplicație: **Proiecții**. Lungimea proiecției vectorului p pe direcția vectorului q este

$$\text{proj}_{\vec{q}}\, \vec{p} = \frac{\vec{p} \cdot \vec{q}}{|\vec{q}|}$$

Aplicație: **Test de perpendicularitate**. Doi vectori sînt perpendiculari dacă produsul lor scalar este 0. Exemplu: pentru punctele $P = (5, 2)$ și $Q = (-4, 10)$, unghiul POQ este drept, iar $-4 \cdot 5 + 2 \cdot 10 = 0$. Mai mult, produsul este pozitiv / negativ după cum θ este mai mic sau mai mare de 90°.

Aplicație: **Comparații de unghiuri**. Vom avea ocazional nevoie să sortăm puncte radial, după unghiul pe care îl fac față de un reper comun. Nu ne interesează unghiurile în sine, ci să le comparăm între ele. Se aplică pe orice domeniu unde funcția cosinus este monotonă (de exemplu 0-180°).

16.2.3 Produsul vectorial (engl. *cross product*)

Se numește așa pentru că rezultatul este o valoare vectorială. Vectorul este perpendicular pe planul determinat de p și q, cu sensul dat de regula șurubului. Mărimea vectorului are două definiții echivalente (din nou, demonstrați!):

- $|\vec{p} \times \vec{q}| = x_p y_q - y_p x_q$
- $|\vec{p} \times \vec{q}| = p \cdot q \cdot \sin\theta$.

Motivație: mărimea produsului vectorial este aria paralelogramului descris de $\vec{p}$ și $\vec{q}$. Alternativ, este dublul ariei triunghiului. Faptul că produsul este orientat înseamnă că aria poate fi pozitivă sau negativă: produsul vectorial „iese din" sau „intră în" planul tablei după cum unghiul θ este pozitiv (p se rotește peste q în sensul trigonometric) sau negativ.

Aplicație: **Coliniaritate**. Dacă trei puncte A, B, C sînt coliniare atunci produsul vectorial $\vec{AB} \times \vec{AC} = 0$.

Aplicație: **Aria unui poligon** prin două metode: (1) ca sumă de trapeze și (2) ca sumă de triunghiuri cu vîrful în origine (vezi secțiunea Arii pentru detalii).

16.2.4 Unghiuri

Putem afla concret unghiul dintre doi vectori cu funcțiile `arcsin`, `arccos`, `atan` sau `atan2`. Atenție la cazurile particulare și erorile de precizie! Funcția `atan2` are meritul că primește la intrare valorile pe componente, `y` și `x`.

Funcțiile trigonometrice inverse sînt lente! Dacă problema permite, înlocuiți-le cu comparații de unghiuri, unde produsele de mai sus sînt suficiente.

16.2.5 Rotații

Pentru a roti punctul (x, y) în jurul originii cu θ radiani, putem aplica o matrice de rotație:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{pmatrix}$$

Putem calcula manual un exemplu, să zicem rotația punctului $(\frac{\sqrt{3}}{2}, \frac{1}{2})$ cu 30°.

Pentru a face rotația relativ la un alt punct, (x_0, y_0), putem face întîi o translație cu $(-x_0, -y_0)$, apoi rotația, apoi translația la loc cu $(+x_0, +y_0)$.

16.3 Drepte și segmente

Există multe forme echivalente pentru ecuația unei drepte în plan. Majoritatea rezultă imediat dintr-un desen (cu asemănare de triunghiuri).

1. $ax + by + c = 0$ – forma generală. Nu are cazuri particulare. Poate descrie cu ușurință drepte verticale ($a = 1$ și $b = 0$) sau orizontale ($a = 0$ și $b = 1$). Doar că nu întotdeauna decurge ușor din datele de la intrare.
2. $\frac{x-x_1}{x_2-x_1} = \frac{y-y_1}{y_2-y_1}$ – dreapta care trece prin punctele (x_1, y_1) și (x_2, y_2). De regulă aceasta decurge din datele de intrare (puncte). Este nedefinită pentru drepte verticale și orizontale (căci $x_1 = x_2$ sau $y_1 = y_2$), dar putem ușor preveni asta înmulțind mezii și extremii.
3. $y - y_1 = m(x - x_1)$ — dreapta care trece prin punctul (x_1, y_1) și are pantă m. Panta este tangenta unghiului format de dreaptă cu axa Ox. Este nedefinită pentru drepte verticale, căci unghiul este de 90° și panta este infinită.
4. $y = mx + y_0$ – dreapta care trece prin punctul $(0, y_0)$și are pantă m (engl. slope + y-intercept).

5. $y_0x + x_0y - x_0y_0 = 0$ – dreapta care taie axele Ox și Oy în punctele $(x_0, 0)$ și $(0, y_0)$ (engl. x-intercept + y-intercept).

16.3.1 Test de paralelism

Trecînd de la forma (1) la forma (3) aflăm că, în forma generală, panta este $-\frac{a}{b}$. Două drepte $ax+by+c=0$ și $dx+ey+f=0$ sînt paralele dacă și numai dacă au aceeași pantă, deci $-\frac{a}{b} = -\frac{d}{e}$ sau, echivalent, $ae = bd$.

16.3.2 Test de perpendicularitate

Două drepte sînt perpendiculare dacă și numai dacă produsul pantelor este egal cu -1:

$$-\frac{a}{b} \cdot -\frac{d}{e} = -1 \iff ad + be = 0$$

Demonstrație: în figura de mai jos, date fiind pantele m_1 și m_2 (unde, atenție, m_2 este negativă), construim punctele Q și R pe cele două drepte și scriem teorema lui Pitagora în triunghiul dreptunghic PQR:

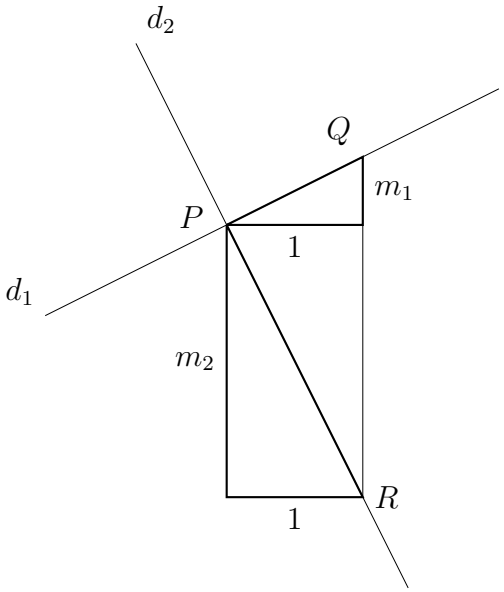


Figura 16.1: Dreptele d_1 și d_2 au respectiv pantele m_1 și m_2.

$$PQ^2 + PR^2 = QR^2$$
$$1^2 + m_1^2 + 1^2 + m_2^2 = (m_1 - m_2)^2$$
$$m_1 m_2 = -1$$

16.3.3 Intersecția a două drepte

Se rezolvă prin rezolvarea sistemului de două ecuații cu două necunoscute:

$$\begin{cases} ax + by + c &= 0 \\ dx + ey + f &= 0 \end{cases}$$

Cazuri particulare: dacă $a/d = b/e$ (sau, echivalent, $ae = bd$), atunci dreptele sînt paralele și nu există soluții. Dacă raportul este egal și cu c/f, atunci dreptele sînt confundate și există o infinitate de soluții.

16.3.4 Separarea planului

Valoarea $ax + by + c$ va fi 0 pentru toate punctele de pe dreaptă și doar pentru acelea. Dar, mai mult decît atît, valoarea va fi în mod consecvent pozitivă sau negativă pentru punctele de o parte și de cealaltă a dreptei. Așadar, două puncte (x_1, y_1) și (x_2, y_2) se află de aceeași parte a dreptei dacă $ax_1 + by_1 + c$ și $ax_2 + by_2 + c$ au același semn.

16.3.5 Distanța punct-dreaptă

Dacă, în plus, ecuația dreptei are **forma canonică**, adică $a^2 + b^2 = 1$, atunci înlocuind un punct în ecuația dreptei nu obținem doar un număr pozitiv/negativ oarecare, ci distanța de la punct la dreaptă.

Putem aduce ecuația dreptei la forma canonică împărțind a, b și c prin $\sqrt{a^2 + b^2}$.

16.3.6 Intersecția a două segmente

Date fiind segmentele AB și CD, dacă avem nevoie de intersecția lor, putem să calculăm intersecția dreptelor și să vedem dacă ea se află în interiorul ambelor segmente. Dacă, în schimb, vrem doar să știm dacă segmentele se intersectează, putem folosi separarea planului, care nu necesită împărțiri:

- Se află A și B de părți diferite ale dreptei CD?
- Se află C și D de părți diferite ale dreptei AB?

Discuție: ce lipsește din implementarea de mai jos? Are erori de precizie? Erori de împărțire prin zero? Cazuri particulare degenerate?

```
typedef struct {
  int x, y;
} point;

int sgn(long long x) {
  return (x > 0) - (x < 0);
}
```

```
int det(point a, point b, point c) {
  return sgn((long long)(b.x - a.x) * (c.y - a.y) -
             (long long)(c.x - a.x) * (b.y - a.y));
}

bool segments_intersect(point a, point b, point c, point d) {
  return
    (det(a, b, c) != det(a, b, d)) &&
    (det(c, d, a) != det(c, d, b));
}
```

16.4 Orientare; determinanți

Am ajuns deja la observația că dreapta separă planul în valori pozitive, negative și nule. Atunci cînd ecuația dreptei are o orientare, de exemplu cînd este specificată prin două puncte P și Q care îi dau un sens, putem căpăta informații ca stînga / dreapta și sens trigonometric / sens orar.

Concret, dîndu-se trei puncte A, B, și C, sunt ele (a) date în ordine trigonometrică sau (b) date în ordine orară sau (c) coliniare? Echivalent, vectorul $\vec{BC}$ cotește, față de vectorul $\vec{AB}$, (a) la stînga sau (b) la dreapta sau (c) nu cotește?

Abordarea directă este să calculăm ecuația dreptei AB sub forma $ax + by + c = 0$ și să înlocuim coordonatele punctului C în această ecuație. Mai simplu, lăsăm ecuația dreptei AB în forma determinată de două puncte. Înlocuind punctul generic (x, y) cu (x_C, y_C) obținem:

$$\frac{x_C - x_A}{x_B - x_A} = \frac{y_C - y_A}{y_B - y_A}$$
$$(x_C - x_A)(y_B - y_A) - (y_C - y_A)(x_B - x_A) = 0$$

Putem scrie partea stîngă ca pe un determinat:

$$\begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} \gtrless 0$$

Valoarea determinantului este:

- pozitivă dacă A, B și C sînt în sens trigonometric;
- negativă dacă A, B și C sînt în sens antitrigonometric;
- 0 dacă A, B și C sînt coliniare.

Aplicație: **test de incluziune a punctului în triunghi** sau în orice poligon convex. Testăm determinanții triunghiurilor formate de punct cu fiecare latură a poligonului, P_iP_{i+1}. Dacă punctul

este strict în poligon, vom obține peste tot același semn. Dacă punctul este pe o latură sau într-un vîrf, vom obține și una sau două valori 0. Dacă punctul este exterior poligonului, vom obține atît valori pozitive, cît și negative. Dacă, în plus, știm ordinea în care sînt specificate vîrfurile poligonului, atunci știm exact ce semn așteptăm, ceea ce scurtează puțin implementarea.

16.5 Arii

De fapt, determinantul de mai sus ne spune mult mai mult decît un simplu semn. Valoarea lui, luată în modul, este dublul ariei triunghiului ABC:

$$\mathcal{A}_{ABC} = \frac{1}{2}\begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix}$$

16.5.1 Aria unui poligon oarecare prin metoda trapezelor

Pentru fiecare latură a poligonului, $P_{i-1}P_i$, să considerăm trapezul dreptunghic determinat de P_{i-1}, P_i și proiecțiile lor pe Ox.

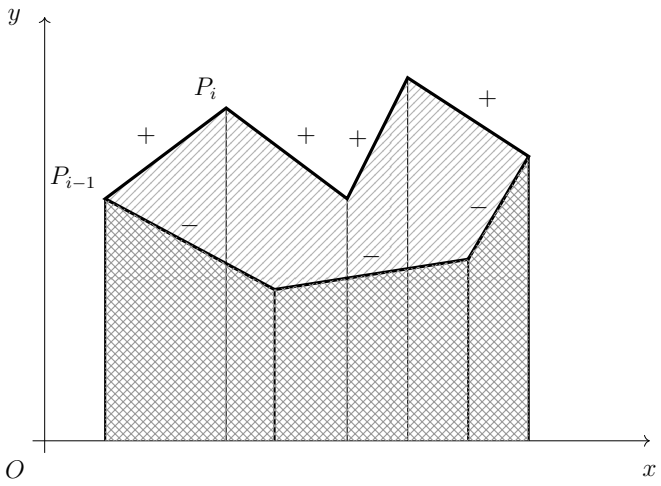


Figura 16.2: Aria heptagonului cu contur gros se obține prin însumarea ariilor cu semn ale celor 7 trapeze.

Acest trapez are aria:

$$\mathcal{A} = (x_i - x_{i-1}) \cdot \frac{y_i + y_{i-1}}{2}$$

Însumăm toate aceste arii și luăm rezultatul în modul. Frumusețea constă în faptul că trapezele pentru care $x_i > x_i - 1$ vor contribui cu arii pozitive, iar celelalte cu arii „negative”. Diferența este exact aria poligonului.

16.5.2 Aria unui poligon oarecare prin metoda triunghiurilor

O metodă similară, dar cu o formulă puțin mai simplă, consideră pentru latura $P_{i-1}P_i$ triunghiul $OP_{i-1}P_i$. Prin fix același raționament deducem că modulul sumei ariilor triunghiurilor este aria poligonului. Iar, deoarece unul dintre puncte este $(0, 0)$, aria se calculează foarte simplu:

$$\mathcal{A}_{OP_{i-1}P_i} = \frac{1}{2}(x_i y_{i-1} - x_{i-1} y_i)$$

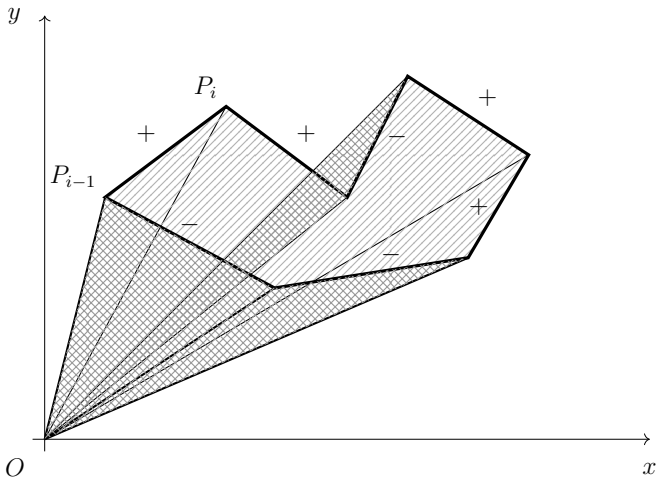


Figura 16.3: Aria heptagonului cu contur gros se obține prin însumarea ariilor cu semn ale celor 7 triunghiuri.

16.6 Probleme

16.6.1 Problema Cuiburi (Baraj ONI 2010)

enunț • sursă

2010 erau vremuri mai simple. 😅 Cărămida de bază a problemei este să testăm incluziunea între două figuri geometrice, care pot fi dreptunghiuri sau cercuri. Deci există un `if` cu patru cazuri, pe care le vom discuta.

Problema cere un fel de drum de lungime maximă într-un graf în care figurile sînt noduri, iar muchiile indică incluziunea. Dar putem observa că graful este aproape aciclic. Poate exista un ciclu doar între figuri identice. Rezultă că putem căuta o sortare topologică astfel încît toate muchiile să meargă doar spre dreapta. Putem sorta după arie sau după perimetru (eu am ales aria).

După sortare, definim $l[i]$ = lungimea maximă a unei cuibăreli terminate cu cuibul i și calculăm vectorul l în complexitate totală $\mathcal{O}(n^2)$.

16.6.2 Problema Ace (OJI 2017, clasa a 9-a)

enunț • sursă

Nu este foarte mult de povestit. Examinăm fiecare rază pornind de la (n, m). Pentru o rază cu progresia (dr, dc), iterăm prin toate punctele și menținem maximul. Prin „maxim” înțelegem acul care ne obligă să „ridicăm” privirea cel mai mult de la orizontală, adică acul al cărui vîrf formează un unghi maxim cu orizontala și cu punctul (n, m).

Am ales să citesc matricea invers ca să pot porni razele de la $(0, 0)$.

16.6.3 Problema Baba Oarba (Lot 2016)

enunț • sursă

Va fi nevoie să facem D pași pentru a ajunge la Zeratul, deci problema ne permite 50 de apeluri ca să ne așezăm pe direcția potrivită. Este oare suficient?

O idee destul de directă este să folosim căutarea binară pentru a înjumătăți sectorul de cerc în care se poate afla Zeratul relativ la Tassadar. Să detaliem!

La o distanță de cel mult 100.000 de metri ne este permisă o abatere de cel mult 1 metru. Ce înseamnă asta ca unghiuri? Cît de precisă trebuie să fie direcția aleasă? Pentru unghiuri foarte mici, știm că $\sin \alpha \approx \alpha$. Așadar, trebuie să calculăm direcția cu o precizie de $1/100.000$. Inițial, direcția poate fi oriunde între 0 și 2π, deci trebuie să reducem intervalul de circa 6,3 milioane de ori. Rezultă că ne trebuie 23 de iterații și avem dreptul să facem cam două interogări per iterație.

Astfel, am redus problema la întrebarea: știind sectorul curent în care se află Zeratul, cum îl înjumătățim punînd doar două întrebări? (Cum înjumătățim sectorul, nu pe Zeratul.)

O soluție imperfectă, dar suficient de bună, calculează bisectoarea sectorului curent și face un pas la stînga, **perpendicular pe bisectoare**. Zeratul este în stînga sau în dreapta bisectoarei după cum Tassadar s-a apropiat sau s-a depărtat de el. Apoi Tassadar face pasul înapoi.

Aș mai preciza și că putem testa problema offline, ca să nu consumăm *submissions* aiurea. Este nevoie să ne scriem propriul babaoarba.h, dar nu este o treabă prea grea.

16.6.4 Problema Arhitect (OJI 2023, clasa a 10-a)

enunț • surse

Problema este elementară. Trebuie să grupăm niște segmente după pante, avînd grijă să numărăm pantele m împreună cu pantele $-1/m$ pentru orice m. Ca să nu operez cu pante infinite, am ales să rotesc toate segmentele pentru a le aduce în cadranul I, mai exact, în intervalul [0°, 90°). Putem face asta cu cîteva `if`-uri, dar mi s-a părut mai simplu să fac efectiv cel mult trei rotații:

```
while ((dx <= 0) || (dy < 0)) {
  int tmp = dy;
  dy = -dx;
  dx = tmp;
}
```

De aici, implementarea se ramifică:

1. Putem opera chiar pe perechi de numere. Pentru a testa egalitatea a două pante `a` și `b`, testăm egalitatea produselor `a.dx * b.dy` și `a.dy * b.dx`.
2. Putem opera pe `double`, căci `dx` este nenul și putem face împărțirea. Nu am întîmpinat erori de precizie, deși în teorie ele sînt posibile.
3. Pentru numărarea duplicatelor, putem să colectăm pantele și să le sortăm sau să le stocăm într-un map.

16.6.5 Problema Elicoptere (OJI 2016, clasele 11-12)

enunț • sursă

Problema cere:

1. Să stabilim ce perechi de triunghiuri pot fi unite prin segmente orizontale sau verticale de lungime cel mult k.
2. Să construim graful indus în care triunghiurile sînt noduri, iar segmentele sînt muchii.
3. Să calculăm arborele parțial minim al a cestui graf și să raportăm diverse informații.

Mai departe, trebuie doar să evităm complicațiile la punctul (1). Cu $n \leq 100$ eficiența este complet irelevantă. Mai important este să programăm îngrijit.

Sper să fie evident că distanța minimă se obține (și) printr-un segment care conține un vîrf al unui triunghi. O demonstrație poate fi: să considerăm că distanța minimă ia naștere undeva între două laturi AB și respectiv CD, dar fără a conține niciun vîrf. Atunci:

- fie $AB \nparallel CD$, caz în care putem translata segmentul de lungime „minimă” pentru a îl scurta,
- fie $AB \parallel CD$, caz în care putem translata segmentul pînă atinge un vîrf, păstrîndu-i lungimea.

Codul meu evidențiază subprobleme succesiv mai mici. Cărămida de bază este aflarea distanței orizontale de la un punct la un segment. Nu am reușit să evit cazul particular în care segmentul este și el orizontal. Pentru tratarea cazului vertical, am preferat să reflect punctele față de diagonala principală ((x, y) devine (y, x)), apoi să rulez tot cazul cu segment orizontal.

16.6.6 Problema Arpa and an Exam About Geometry (Codeforces)

enunț • sursă

Și această problemă este relativ elementară. Pentru a putea roti A peste B, trebuie să alegem un centru P astfel încît $PA = PB$. Așadar, P trebuie să fie pe mediatoarea segmentului AB. Similar, P trebuie să fie pe mediatoarea segmentului BC. În concluzie, P este centrul cercului circumscris triunghiului ΔABC. Din fericire, nu trebuie să aflăm acest centru, ci doar să știm dacă el există. Este suficient să verificăm că A, B și C nu sînt coliniare.

Pentru a putea roti **simultan** A peste B și B peste C, să ne închipuim că rotim cercul centrat în P. Este nevoie ca $\angle APB = \angle BPC$, ceea ce se poate exprima mai simplu ca $AB = BC$.

16.6.7 Problema Bear and Floodlight (Codeforces)

enunț • sursă

Problema are două componente, ambele ușoare (la valoarea noastră...).

Componenta de geometrie

Cărămida de bază este: presupunînd că am reușit, folosind o submulțime de reflectoare, să ducem ursul în punctul $(x, 0)$, pînă unde îl putem duce folosind un nou reflector aflat la x_c, y_c și de unghi a?

Metoda 1: Aflăm, de exemplu cu funcția `atan2`, unghiul format de vectorul $(x_c, y_c) - (x, 0)$ cu orizontala. Adăugăm a la acest unghi. Obținem ecuația unei drepte care trece prin (x_c, y_c) și are un unghi determinat. În ecuația acestei drepte înlocuim y cu 0 și aflăm noul x.

Metoda 2: Aplicăm o matrice de rotație punctului $(x, 0)$ raportat la (x_c, y_c). Astfel obținem un punct (x', y'). Din punctele coliniare (x_c, y_c),(x', y') și $(x_{nou}, 0)$ aflăm prin asemănare x_{nou}. Mi se pare o metodă mai simplă și este și mai rapidă, întrucît putem precalcula sinusul și cosinusul unghiului de rotație a. În rest, facem doar înmulțiri. În metoda 1 nu avem această posibilitate, căci mereu aplicăm arctangenta unui segment diferit.

Este nevoie să tratăm cazul cînd reflectorul ajunge să bată orizontal sau chiar în sus, caz în care noul x poate fi nedefinit sau poate „fugi" înapoi în stînga. Dar este suficient doar un `if`: dacă $y' \geq y_c$, înseamnă că am rotit reflectorul într-o poziție care acoperă semidreapta orizontală de la (x_c, y_c) spre dreapta.

Componenta de programare dinamică

Dat fiind că n este mic, ne permitem să scriem o programare dinamică pe măști de biți. Pentru o mască m reținem x-ul maxim la care putem ajunge folosind doar reflectoarele cu bitul setat în mască, în orice ordine.

Cum calculăm acest maxim? Dintre reflectoarele din m, unul trebuie să fie ultimul. Deci, rînd pe rînd, eliminăm cîte un bit, luăm x-ul maxim oferit de masca rămasă și îl extindem cu reflectorul al cărui bit l-am eliminat. Astfel obținem complexitatea totală $\mathcal{O}(n \cdot 2^n)$.

16.6.8 Problema TrapEZZ (Moisil++ 2023, clasele 11-12)

enunț • sursă

Soluțiile relativ evidente sînt $\mathcal{O}(n^4)$, care iterează prin toate mulțimile de 4 puncte, și $\mathcal{O}(n^3 \log n)$ sau chiar $\mathcal{O}(n^3)$, care fixează 3 puncte și îl caută pe al 4-lea.

Pentru o soluție în $\mathcal{O}(n^2 \log n)$, să vedem ce putem face pentru perechi de puncte. Dacă fixăm o bază și ne întrebăm „unde ar putea fi cealaltă bază?”, nu (cred că) ajungem nicăieri, pentru că bazele trebuie să fie bine aliniate pentru ca trapezul să fie isoscel. Însă așa ne poate veni ideea să folosim mediatoarele pe baze: două segmente reprezintă bazele unui trapez isoscel dacă au aceeași mediatoare.

Restul sînt detalii de implementare. Decît să grupez segmentele într-un `map` după ecuația mediatoarei, am preferat să le sortez după mediatoare, apoi să le procesez în blocuri cu aceeași mediatoare, ceea ce s-a dovedit a fi mai rapid.

Este nevoie de atenție și la condițiile (3) și (4): trebuie să eliminăm trapezele degenerate și dreptunghiurile.

16.6.9 Problema Terenuri (Baraj ONI 2011)

enunț • sursă

Problema definește exact diagramele Voronoi, dar rezolvarea nu se bazează pe ele (din fericire). Vom studia natura diagramelor Voronoi și vom trage concluzia că un țăran este liber dacă și numai dacă el se află pe înconjurătoarea convexă a setului curent de puncte (inclusiv pe o latură). Deocamdată nu avem nevoie să cunoaștem algoritmi pentru înfășurătoarea convexă. Îi vom studia în lecția viitoare.

Așadar, problema se reduce la menține incremental înfășurătoarea convexă, pe măsură ce setul de puncte crește. Înfășurătoarea poate doar să crească, deci țăranii care sînt liberi pot deveni dominați, niciodată invers. Întrebarea-cheie este: cum procesăm un nou punct? Dacă el se află în interiorul înfășurătoarei, atunci țăranul aferent este dominat de la bun început și îl putem ignora. Dacă punctul se află pe înfășurătoare sau în afara ei, trebuie să îl adăugăm la înfășurătoare și eventual să scoatem punctele care formează unghiuri concave.

Acest lucru poate fi codat scurt și elegant, cu un singur `map<double, point>`. Alegem un centru arbitrar, de exemplu medianul primelor două puncte de la intrare. Raportat la centru, menținem punctele de pe înfășurătoare sortate polar. Am folosit funcția `atan2` pentru calculul unghiului, căci datele sînt deja numere reale. Restul implementării cere doar o navigare îngrijită printre vecinii, în `map`, ai punctului nou inserat.

Merită să avem o discuție aici despre evitarea redundanței în `std::map`. Dorim să inserăm punctul și să obținem un iterator la el, pentru a elimina unghiurile reflexe. Cel mai scurt cod ar arăta așa:

```
auto it = hull.find(angle);

if (it == hull.end()) {
hull[angle] = p;
it = hull.find(angle);
} else {
```

```
  it->second = p;
}
```

Doar că acest cod face căutarea de trei ori! De două ori cu funcția `find()` și a treia oară cu operatorul `[]`. Putem reduce aceasta la doar două căutări, căci funcția `map::insert` returnează un iterator la elementul inserat, exact ce ne trebuie:

```
iter it = h.find(angle);

if (it == h.end()) {
  it = h.insert({angle, p}).first;
} else {
  it->second = p;
}
```

Sursa completă anexată arată cum putem face o singură căutare, ceea ce mai cîștigă 10% din timp. Secretul este o combinație de două lucruri.

- Funcția `lower_bound()` returnează un iterator la primul element mai mare sau egal cu cheia căutată, adică exact la poziția inserării.
- Funcția `insert()` acceptă iteratorul de mai sus ca parametru și face inserarea în $\mathcal{O}(1)$.

16.6.10 Problema Metin2 (Finala IIOT 2021-22)

enunț • sursă

Ideea de rezolvare nu este excesiv de grea. Totul este să nu cădem în capcana de a menține intersecția poligonului, care pare o abordare nerezonabil de dificilă.

Mai rezonabil este să colectăm toate cele $4n$ laturi ca pe niște drepte **orientate**. Apoi le sortăm după pantă. Acum știm că poligonul-intersecție trebuie să se afle **în stînga tuturor dreptelor**. O primă observație este că, dacă există mai multe drepte paralele (cu aceeași pantă), este suficient să o păstrăm pe cea mai din stînga. Dacă intersecția o va respecta pe aceasta, le va respecta automat și pe restul.

Acum să considerăm că am iterat printr-o parte din drepte și am construit o parte din poligon, $ABC \ldots OPQ$. Să procesăm următoarea dreaptă în ordinea pantei, d. Dacă ea continuă „natural” poligonul, în sens trigonometric, atunci Q se va afla la stînga lui d, iar poligonul se continuă cu o porțiune din d. Dacă, în schimb, Q se află la dreapta lui d, atunci prin definiție Q nu are cum să facă parte din intersecție.

(TODO: figură)

Rezultă că putem ține o stivă de drepte presupuse a forma laturile intersecției. Pentru ușurință, ținem și o stivă de intersecții între perechi de drepte consecutive. Acestea formează vîrfurile poligonului. Fiecare dreaptă procesată elimină din stivă vîrfurile aflate în dreapta ei, precum și

dreptele aferente acelor vîrfuri. Apoi se inserează pe ea însăși în stivă, împreună cu punctul de intersecție între ea și ultima dreaptă din stivă.

La final, rezultă o linie frîntă în formă de „6", care se poate autointersecta și poate conține multe segmente redundante dincolo de poligonul propriu-zis. Acestea trebuie eliminate, de la ambele capete, folosind același test de orientare (stînga = OK, dreapta = elimină).

Detaliu de implementare)

Cînd sortăm dreptele după pantă, la egalitate (drepte paralele) dorim să le sortăm de la stînga la dreapta. Cum definim „stînga" și „dreapta"? Dreapta d_1 este în stînga dreptei d_2 dacă, înlocuind un punct (orice punct) de pe dreapta d_1 în ecuația dreptei d_2 obținem o valoare pozitivă.

Cel mai simplu mi-a fost să rețin, pentru fiecare dreaptă, un punct cunoscut a fi pe dreaptă (unul dintre cele două puncte date la intrare). Există și o metodă de a afla un punct de pe dreapta d_1 în momentul testului, ceea ce reduce memoria, dar codul devine mai lung și mai greoi.

Capitolul 17

Algoritmi specifici

17.1 Teorema lui Pick

Teorema lui Pick (Wikipedia, CP Algorithms) definește o relație între aria unui poligon cu vîrfuri de coordonate întregi (numit și poligon laticeal) și numărul de puncte laticeale din interiorul său și de pe conturul său. Dacă poligonul are I puncte laticeale interioare și B puncte pe contur (engl. *boundary*), atunci aria sa A este:

$$A = I + \frac{B}{2} - 1$$

Demonstrația nu este grea, dar este laborioasă. Ea demonstrează corectitudinea formulei de la figuri simple spre complexe:

1. Dreptunghiuri cu laturile paralele cu axele.
2. Triunghiuri dreptunghice cu laturile paralele cu axele.
3. Triunghiuri oarecare, prin completarea lor, în exterior, cu triunghiuri dreptunghice pentru a forma un dreptunghi.
4. Poligoane oarecare, prin triangulare.

17.2 Înfășurătoarea convexă

Definiție: **Înfășurătoarea convexă** a unei mulțimi de n puncte este cel mai mic poligon convex care include mulțimea, pe contur sau în interior.

Vom descrie trei algoritmi clasici:

1. Graham scan, de complexitate $\mathcal{O}(n \log n)$.
2. Lanțuri monotone, tot de complexitate $\mathcal{O}(n \log n)$.
3. Împachetarea cadoului (numit și „marșul lui Jarvis"), de complexitate $\mathcal{O}(n \cdot h)$.

Complexitatea $\mathcal{O}(n \log n)$ este optimă. Dacă ar exista un algoritm asimptotic mai bun, l-am

putea folosi ca să sortăm numere mai rapid decît $\mathcal{O}(n \log n)$. Fie $v[1...n]$ vectorul de sortat. Pentru fiecare element $v[i]$ creăm un punct în plan $(v[i], v[i]^2)$. Calculăm înfășurătoarea convexă a mulțimii de puncte. Cum punctele sînt pe o parabolă, toate vor fi pe înfășurătoarea convexă, care va enumera aceste puncte în ordine... adică va sorta vectorul v.

Ca la multe alte implementări de geometrie computațională, dificultatea constă în evitarea cazurilor particulare.

17.3 Algoritmi de baleiere

Un **algoritm de baleiere** (engl. *sweep line algorithm*) este o tehnică prin care iterăm printr-o colecție de obiecte (puncte, segmente, dreptunghiuri etc.) prin translatarea în plan a unei drepte imaginare, de exemplu prin translatarea orizontală a unei drepte verticale. Menținem o structură de date relevantă pentru poziția curentă a dreptei. Examinăm momentele în care dreapta atinge o poziție notabilă: coordonata unui punct, extremitatea stîngă sau dreaptă a unui dreptunghi etc. Aceste momente se numesc **evenimente**. Pentru fiecare eveniment actualizăm structura de date, ceea ce de regulă ne permite să actualizăm și răspunsul la problemă.

Folosim această tehnică pentru a obține timpi de rulare asimptotic mai buni decît cu o soluție naivă. Wikipedia oferă puțin context istoric și un exemplu vizual pentru calculul diagramelor Voronoi.

Unele dintre problemele de baleiere necesită efectiv geometrie computațională, folosind formule pentru intersecții, orientări etc. Altele sînt rezolvabile și fără, doar cu menținerea unor structuri de date peste numere întregi.

Vom discuta teoretic două probleme clasice:

1. Dîndu-se o colecție de n segmente, găsiți două care se intersectează sau raportați că nu există nicio intersecție. Puteți găsi explicații în detaliu pe CP Algorithms și o prezentare cu mai multă grafică aici. Pentru o problemă înrudită, vedeți Lights (CodeChef).
2. Calculați aria / perimetrul reuniunii a n dreptunghiuri cu laturile paralele cu axele. Pentru o formulare apropiată, vedeți Vika and Segments (Codeforces).

17.4 Șublerul rotitor

Această metodă (engl. *rotating calipers*) amintește un pic de tehnica celor doi pointeri. Ea reduce la $\mathcal{O}(n)$ timpul de rezolvare pentru probleme pe care le-am putea rezolva în $\mathcal{O}(n \log n)$ prin n căutări binare. Ea rezolvă multe probleme clasice ca:

- Diametrul unui poligon (distanța maximă între două puncte din poligon).
- Lățimea unui poligon (distanța minimă între două drepte paralele care cuprind poligonul).
- Distanța minimă/maximă dintre două poligoane.
- Dreptunghiul de arie/perimetru minim care cuprinde un poligon.

- ... și multe altele, enumerate *pe Wikipedia*.

Există cîteva noțiuni comune acestor probleme:

1. O **dreaptă-suport** este o dreaptă care trece printr-un vîrf al poligonului fără a trece prin interiorul acestuia.
2. O pereche de **puncte antipodale** constă din puncte care admit drepte-suport paralele.
3. Metoda șublerului rotitor se bazează pe enumerarea tuturor perechilor de puncte antipodale.
4. Trecerea de la o pereche la următoarea se poate face în $\mathcal{O}(1)$.

17.5 Probleme

17.5.1 Problema Copaci (Infoarena)

enunț • sursă

Soluția aplică direct Teorema lui Pick.

17.5.2 Problema Emptri (Lot 2015)

enunț • sursă

Teorema lui Pick ne pune pe direcția corectă. Dorim ca $I = 0$ și $B = 3$, de unde rezultă $A = \frac{1}{2}$. Restul este contabilitate. Fie cele două puncte (a, b) și (c, d) și fie $c \leq a$. Aria triunghiului va fi $\frac{1}{2}|ad - bc|$, calculabilă prin orice metodă preferați. Așadar, dorim ca $ad - bc = \pm 1$. Mai mult, trebuie ca $b \leq a$ și $\gcd(a, b) = 1$, căci altfel segmentul $(0, 0) - (a, b)$ va conține puncte în plus.

Încercînd să găsesc o soluție în $\mathcal{O}(n^2)$, m-am împiedicat de cea în $\mathcal{O}(n \log \log n)$. Să fixăm valorile a și b. Cîte variante există pentru c și d? Observăm că formula $ad - bc = 1$, tocmai pentru că a și b sînt coprime, „seamănă” cu Identitatea lui Bézout și/sau algoritmul lui Euclid extins, din care știm că soluția este unică modulo a. Similar, va exista o soluție unică și pentru $ac - bd = -1$.

De exemplu, pentru punctul $(a, b) = (5, 2)$ obținem ecuațiile $5d - 2c = \pm 1$, care ne dă punctele $(c, d) = (2, 1)$ și respectiv $(c, d) = (3, 1)$.

Dacă pentru fiecare pereche de numere coprime (a, b) există fix două soluții, rezultă că trebuie doar să numărăm aceste perechi. Pentru aceasta ne folosim de funcția Euler, căci prin definiție $\varphi(a)$ este exact numărul de valori b coprime cu a. Răspunsul final este suma valorilor $\varphi(a)$, dublată conform raționamentului de mai sus, și $+1$ pentru triunghiul special $(0, 0) - (1, 0) - (1, 1)$:

$$R = 1 + 2 \sum_{a=2}^{n} \varphi(a)$$

Soluția oficială (descărcabilă de pe Infoarena) pomenește despre secvențe Farey. Nu știu ce sînt acelea, misiunea mea se încheie aici. 🙂

17.5.3 Problema Înfășurătoare convexă (Infoarena)

enunț • sursă

Aceasta este o problemă educațională. Enunțul este permisiv: el promite că nu vor exista puncte coliniare pe înfășurătoare. Funcția `graham_scan()` implementează strict algoritmul de stivă ordonată, în 10 linii de cod. Orice detalii dorim, le putem obține din sortarea atentă a punctelor. Apoi, algoritmul

- nu are cazuri particulare;
- nu procedează diferit pe jumătatea de sus și pe cea de jos;
- nu tratează special niciun punct în afară de primul.

17.5.4 Problema Înfășurătoare convexă (NerdArena)

enunț • sursă

Problema este aproape identică cu precedenta. Enunțul este mai strict și ne cere să nu includem puncte pe laturile înfășurătorii, ci doar în colțuri. Este nevoie de un `if` în plus la sortare (la unghi polar egal, luăm în calcul și distanța față de `p[0]`). De asemenea, trebuie să folosim operatorul corect la scoaterea din stivă, ca să excludem punctele de pe laturile înfășurătorii.

17.5.5 Problema Magic (JBOI 2023)

enunț • sursă

Problema este încadrată la o tehnică numită *Convex Hull Trick*. Puteți citi mai multe despre CHT pe Codeforces (de exemplu). Mărturisesc că nu înțeleg diferența față de algoritmul normal de înfășurătoare convexă. Este fix aceeași abordare cu o stivă ordonată. În plus, mă nemulțumesc termenii ca „trick” și „șmen”, pentru că ei induc noțiunea falsă că algoritmica este o colecție de trucuri și șmenuri.

Reducerea la geometrie computațională este destul de clară. Vrăjitorului i îi corespunde punctul în plan (x_i, e_i). Pentru fiecare vrăjitor i, trebuie să aflăm un alt vrăjitor j pentru care panta $(x_j, e_j) - (x_i, 0)$ este maximă. Vom rezolva întîi problema la stînga ($j < i$). Pentru a rezolva problema la dreapta, vom oglindi toate coordonatele x și vom reapela aceeași rutină, ca să nu duplicăm cod. La final, vom oglindi la loc coordonatele x ca să tipărim răspunsurile în ordinea corectă.

Să ne imaginăm că vrăjitorul i locuiește în vîrful unui stîlp de înălțime e_i la coordonata x_i. Stîlpii maschează parțial sau complet alți stîlpi aflați dincolo de ei. Pentru a alege un mentor la stînga, vrăjitorul i trebuie să coboare la baza stîlpului și să „privească în sus” căutînd stîlpul al cărui vîrf face unghi maxim cu orizontala.

Fie P punctul vrăjitorului curent. Cînd comparăm doi mentori posibili, A și B, unde $x_A < x_B < x_P$, P îl va prefera pe A dacă B se află sub segmentul AP; altfel P îl va prefera pe B. Echivalent, vrăjitorul A va fi preferabil tuturor vrăjitorilor de sub linia AP.

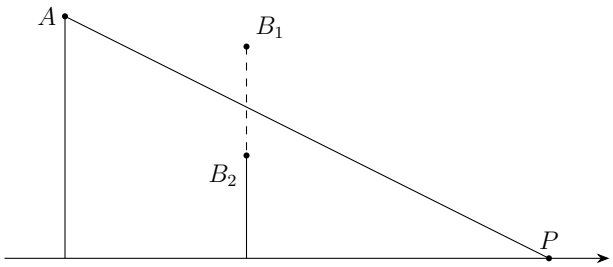


Figura 17.1: Privind din punctul P, stîlpul B_1 maschează stîlpul A complet, dar stîlpul B_2 nu. De aceea, vrăjitorul P l-ar prefera pe B_1 față de A ca mentor, dar l-ar prefera pe A față de B_2.

Să considerăm acum un exemplu mai complex. În figura 17.2, fiecare vrăjitor de la A la F este mentor pentru vecinul său din dreapta. Remarcăm că arcul $ABCDEF$ este convex. Pentru claritate, am împărțit zona de deasupra arcului în triunghiuri.

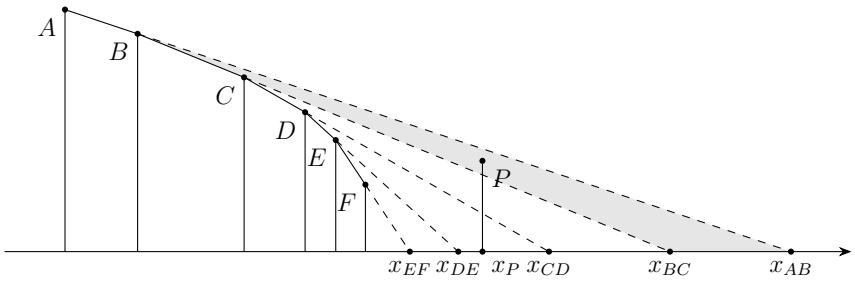


Figura 17.2: Mentorul vrăjitorului P este D deoarece x_P se află în triunghiul lui D. Mai mult, punctul P se află în triunghiul lui B, deci vrăjitorii $C - F$ nu vor mai fi mentori niciodată.

Fiecare vrăjitor va fi mentor pentru punctele din triunghiul corespunzător. De exemplu, triunghiul lui B este colorat cu gri. Ce avem de făcut cînd procesăm următorul vrăjitor, P? Mai întîi, stînd la $(x_P, 0)$, „privim în sus" către cel mai de sus vrăjitor vizibil. În figură, acela este vrăjitorul D, deci îl alegem pe D ca mentor la stînga pentru P. Dar putem face mai mult decît atît. Deoarece $x_P > x_{DE}$, vrăjitorul P este în afara triunghiurilor lui E și F. **Indiferent cît de înalt este stîlpul lui** P, vrăjitorii E și F nu vor mai fi niciodată mentori pentru coordonate $x > x_P$.

Mai mult, punctul $P(x_P, e_P)$ se află în triunghiul vrăjitorului B. Aceasta ne spune că nici vrăjitorii C și D nu vor mai fi vreodată mentori pentru valori mai mari ale lui x, deoarece stîlpul lui P separă punctul C de porțiunea din triungi de coordonate $x > x_P$ (și similar pentru D). Pentru acele coordonate, doar A, B și P mai pot fi mentori.

În practică, vom menține o stivă de vrăjitori care mai au vreo șansă să fie mentori pe viitor. Cînd procesăm vrăjitorul curent P, ștergem din vîrful stivei toți vrăjitorii care devin invizibili. Există un mod mai simplu de a decide dacă un vrăjitor trebuie șters, unul care folosește doar testul de orientare stînga-dreapta. Examinăm ultimii doi vrăjitori de pe stivă, X și Y. Dacă $\angle XYP$ este

reflex (mai mare decît 180°, atunci îl ștergem pe Y. Acest proces este echivalent cu raționamentul bazat pe triunghiuri din figura 17.1.

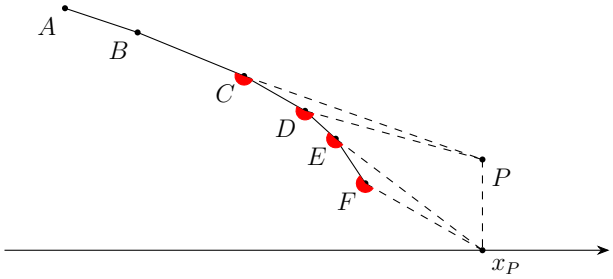


Figura 17.3: Punctul $(x_P, 0)$ șterge vrăjitorii F și E din stivă. Apoi, punctul P șterge vrăjitorii D și C din stivă. Unghiurile marcate cu roșu sînt reflexe.

Deoarece fiecare vrăjitor poate fi șters din stivă o singură dată, complexitatea amortizată este $\mathcal{O}(n)$.

17.5.6 Problema Triangular Queries (CodeChef)

enunț • sursă

Problema nu este de geometrie, dar ilustrează bine conceptul de „eveniment”.

Ideea de baleiere ia naștere din observația că ne este mai simplu să ținem evidența punctelor dacă avem garanția că toate sînt de aceeași parte a liniei de baleiere. Putem alege multe direcții de baleiere; eu am ales direcția NE $\rightarrow$ SV. Așadar, considerăm o dreaptă orientată de la NV la SE, care va avea ecuația $x + y = a$, și o translatăm de la $a = +\infty$ la $a = -\infty$.

Să considerăm o interogare, reprezentată în figura de mai jos prin triunghiul E. Iau naștere șase zone marcate cu literele $A \ldots F$:

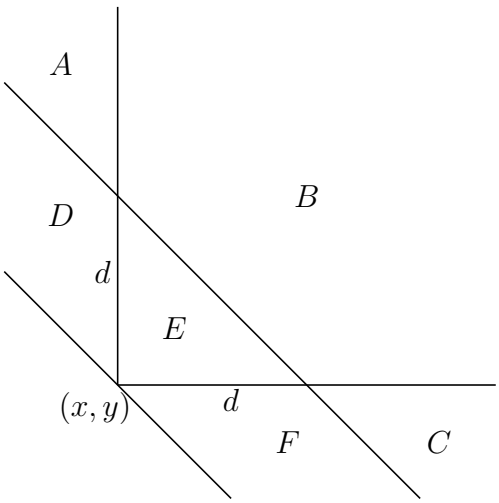


Figura 17.4: Poziția liniei de baleiere la două evenimente cauzate de interogarea triunghiulară E: baza și vîrful triunghiului.

Calculăm numărul de puncte din E prin diferență:

$$\begin{aligned} E &= (B + E) - B \\ &= [(A + B + C + D + E + F) - (A + D) - (C + F)] - [(A + B + C) - (A) - (C)] \end{aligned}$$

Toate cele 6 cantități din paranteze sînt ușor disponibile, dar la două momente diferite în timp. Cînd procesăm diagonala de sumă $x + y + d$ (cea de sus), atunci:

- $A + B + C$ este pur și simplu numărul de puncte văzute pînă atunci.
- A este numărul de puncte de coordonată x mai mică decît x-ul interogării. Putem menține ușor această informație într-un AIB.
- C este numărul de puncte de coordonată y mai mică decît y-ul interogării. De asemenea, putem menține această informație într-un AIB.

Similar putem afla celelalte cantități cînd procesăm diagonala de sumă $x+y$ (cea de jos). Rezultă că iau naștere trei tipuri de evenimente pe măsură ce linia de baleiere avansează:

1. Întîlnim un punct nou.
2. Întîlnim începutul unei interogări (ipotenuza triunghiului).
3. Întîlnim sfîrșitul unei interogări (vîrful dreptunghic al triunghiului).

O necesitate tipică pentru multe probleme este să clarificăm în ce ordine procesăm mai multe evenimente care survin la aceeași coordonată. În cazul nostru,

- Un punct nou trebuie procesat **după** ce procesăm orice bază care îl conține. Altfel îl vom scădea la contorizarea punctelor din triunghi, ceea ce este incorect.
- Un punct nou trebuie procesat **înainte** să procesăm vîrful dreptunghic al oricărui triunghi care se suprapune peste punct. De data aceasta vrem să contorizăm punctul.
- Între baza unui triunghi și vîrful altuia care coexistă pe aceeași linie de baleiere nu există niciun conflict; le putem procesa în orice ordine.

Pentru claritate, vă îndemn să definiți și să colectați explicit evenimentele. La procesarea unui eveniment, în funcție de tipul său, apelați o funcție bine denumită. Atunci scrierea programului devine ușoară.

17.5.7 Problema Hill Walk (USACO Gold 2013)

enunț • sursă

Vom face o baleiere de la stînga la dreapta, în care evenimentele sînt cele $2n$ capete de interval, ordonate după x. La poziția curentă a liniei de baleiere dorim să avem evidența intervalelor active, ordonate de jos în sus. Atunci ne trebuie o structură de date care să admită operațiile:

1. Adaugă un interval (cînd întîlnim un eveniment de tip capăt stîng).
2. Șterge un interval (cînd întîlnim un eveniment de tip capăt drept).

3. Dă-mi intervalul anterior celui tocmai șters (pentru a actualiza poziția lui Bessie cînd intervalul ei curent se termină).

Implementarea este destul de alunecoasă. Intuitiv, am vrea să sortăm intervalele de la „jos" la „sus". Deci operatorul $a < b$ ar trebui să returneze `true` dacă și numai dacă a este poziționat „dedesubtul lui" b. Dacă două intervale acoperă un x comun (cu alte cuvinte, dacă proiecțiile lor pe axa Ox se suprapun), atunci noțiunea de „dedesubt" este bine definită și poate fi determinată folosind testul standard de orientare trigonometrică.

Dacă două intervale sînt disjuncte pe x, atunci noțiunea de „dedesubt" nu este bine definită. În special, nu putem încerca o sortare a tuturor celor n intervale simultan, căci relația de ordine este parțială, nu totală. Este important să comparăm doar intervale care coexistă la o coordonată x.

Putem folosi un `set` și operatorul `<` redefinit ca test de orientare.

17.5.8 Problema Ydist (Lot 2014)

enunț • sursă

Problema ne dă n puncte în plan și q raze (semidrepte) pornind din origine, toate aflate în cadranul I. Pentru fiecare rază, trebuie să determinăm distanța minimă pe verticală pînă la un punct aflat deasupra razei.

Problema este una de baleiere; ne gîndim la asta pentru că este mult mai simplu să răspundem la o întrebare (constînd dintr-o rază) cînd pînă acum am procesat doar puncte aflate deasupra razei. Din același motiv, vom face baleierea în ordine invers polară (de la 90° spre 0°).

Conceptul de baleiere

Așadar, avem de procesat două tipuri de evenimente: (1) apare un punct nou și (2) trebuie să răspundem la o interogare. Să considerăm un moment al baleierii, ilustrat în figura 17.5. Fie R o interogare și fie P răspunsul la interogare. Ce putem spune despre restul planului? Mai exact, unde se pot afla puncte, unde nu se pot afla, și pe care le putem șterge?

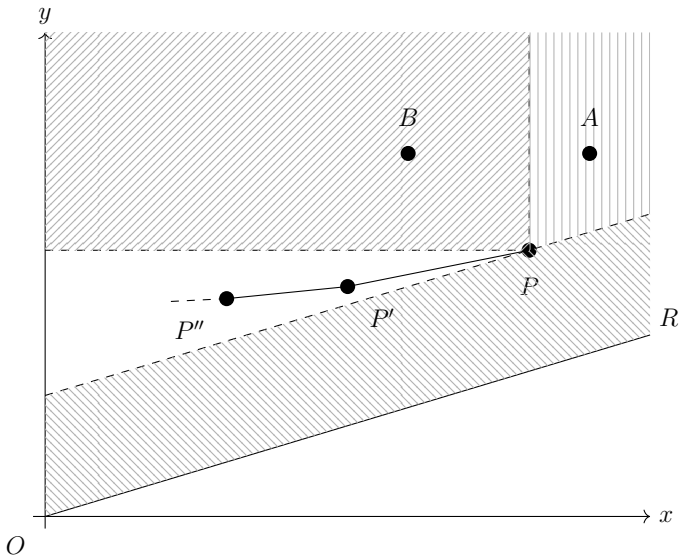


Figura 17.5

1. Nu se pot afla puncte în zona hașurată de sub P, situată între R și paralela la R prin P. Dacă ar exista vreun punct în acea zonă, el ar fi mai aproape de R decît P, deci el ar fi răspunsul.
2. Dacă există puncte în zona hașurată la nord-est de P, ele pot fi șterse, așa cum este cazul lui A. Punctul A este mai departe de R decît P. Mai mult, pe măsură ce raza de baleiere coboară, ea se va îndepărta de A mai repede decît de P. Cu alte cuvinte, niciodată în viitor A nu va mai putea fi răspunsul la vreo interogare.
3. Dacă există puncte în zona hașurată la nord-vest de P, ele pot fi șterse, așa cum este cazul lui B. Punctul B este, în prezent, mai departe de R decît P. Pe măsură ce raza de baleiere coboară, B recuperează din diferență. Dar la ce moment va egala el situația? La ce moment se vor afla B și P la distanțe egale de raza de baleiere? Doar atunci cînd raza va fi paralelă cu BP... adică niciodată. Așadar, nici B nu va mai fi vreodată răspunsul la o interogare.

Ordonarea după x și y

Astfel ne vine ideea să menținem un set de *puncte-candidat*: puncte care ar mai putea fi vreodată răspunsul la o interogare viitoare. Aceste puncte candidat se pot afla doar în triunghiul de la vest de P. Să privim din nou Figura 1. Fie P' cel mai din dreapta punct-candidat, cu excepția lui P. Punctului P' i se aplică aceeași regulă ca și lui P: punctele la nord-vest de el pot fi șterse. Așadar, următorul punct-candidat, P'', se va afla la sud-vest de P'.

Am demonstrat astfel că punctele-candidat sînt ordonate după x și după y.

Dacă ați avut în acest moment intuiția că putem menține punctele-candidat folosind o simplă stivă ordonată, sînteți pe drumul cel bun. Dar mai avem de demonstrat două lucruri.

Concavitatea

Prima întrebare este: cînd primim o interogare, care dintre punctele-candidat este răspunsul? Trebuie să le evaluăm pe toate? Poate exista o situație ca cea din figura 17.6, în care punctele

ba se apropie, ba se depărtează de raza de baleiere?

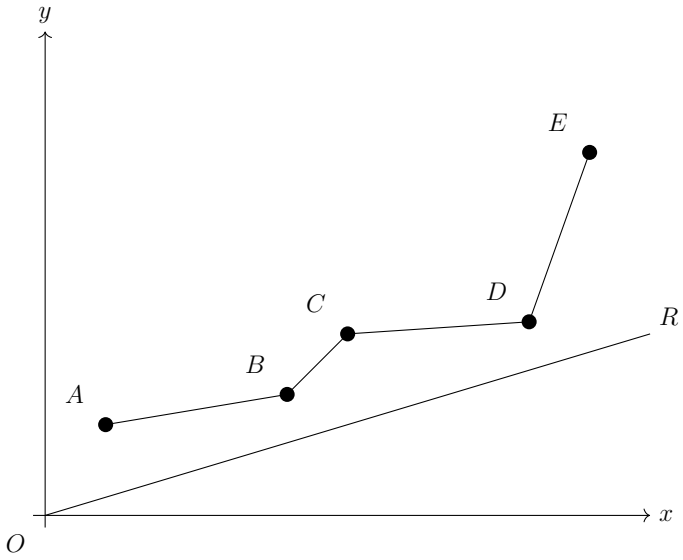


Figura 17.6

Răspunsul este „nu”. Curba punctelor-candidat este concavă (are forma literei U). (Fapt divers: Aici am pierdut circa 30 de minute încercînd să demonstrez că curba ar fi convexă...) Să urmărim figura 17.7, în care vedem 3 puncte-candidat A, B și C într-o formă convexă. Să încercăm să obținem o contradicție.

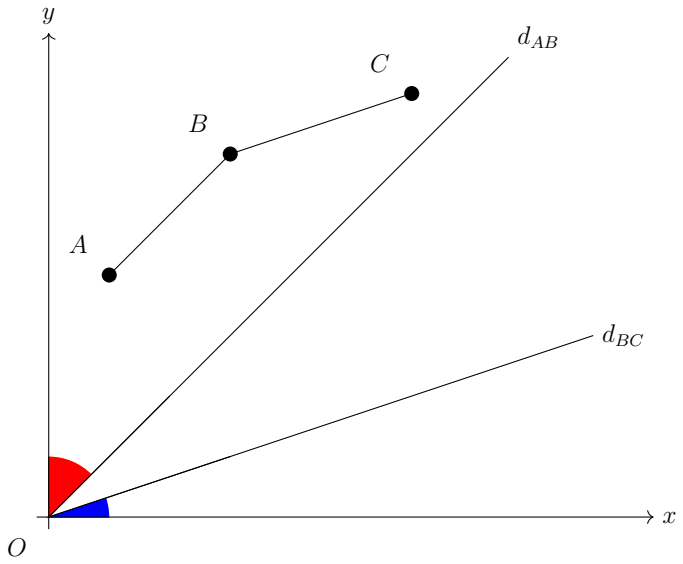


Figura 17.7

Există două raze de interes, $d_{AB} \parallel AB$ și $d_{BC} \parallel BC$. Să observăm că B este preferabil lui A (adică mai aproape de rază) doar pentru razele de deasupra lui d_{AB} (unghiul roșu). Similar, B este preferabil lui C doar pentru razele de sub d_{BC} (unghiul albastru). Intersecția celor două unghiuri este vidă, așa că nu există niciun moment cînd B să le fie preferabil și lui A, și lui C. El poate fi șters. Astfel am demonstrat că lista punctelor-candidat este convexă.

De aici deducem că distanța la o rază, cînd iterăm prin toate punctele-candidat, este bitonă: ea scade pornind de la extreme spre centru. De aceea, putem aplica algoritmul clasic de eliminare din stiva ordonată: cînd primim o interogare, eliminăm ultimul punct din vîrful stivei cît timp penultimul punct are o distanță mai mică pînă la interogare.

Ordonarea polară

Ultima observație necesară privește inserarea punctelor. Cînd linia de baleiere întîlnește un nou punct P, unde este locul lui în lista de candidați? Poate fi undeva la mijloc, cauzînd o inserare în $\mathcal{O}(n)$?

Răspunsul este „nu”. Punctul P va avea cel mai mic unghi polar de pînă acum (căci baleierea este polară), iar lista punctelor-candidat este ordonată polar. Ca să ne convingem, să urmărim figura 17.8.

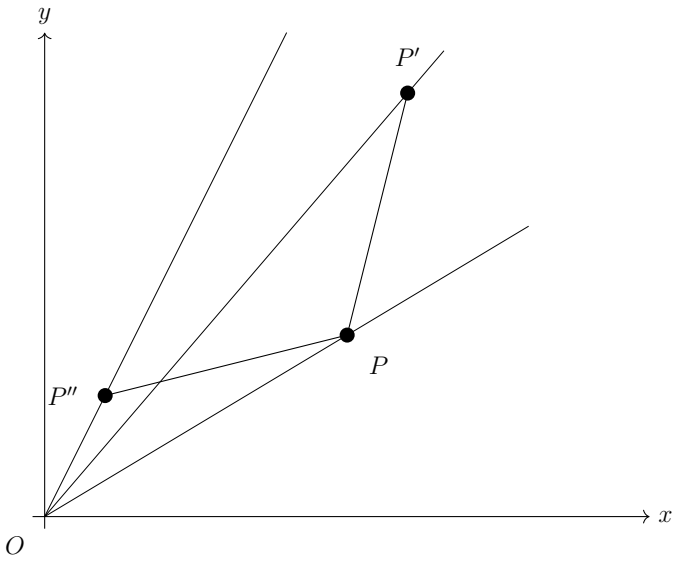


Figura 17.8

Punctul P este cel mai recent descoperit de raza de baleiere. Să presupunem că locul său, conform sortării după x sau y, ar fi între punctele anterioare P' și P''. Dar acum să observăm că, de fapt, puncul P' se află exact în situația punctului A din Figura 1. El nu va mai fi niciodată răspunsul la o interogare și îl putem șterge.

Aceasta este o veste bună. Punctul nou venit P se va așeza întotdeauna în vîrful unui stive. În prealabil, el elimină din stivă punctele care încalcă ordonarea polară / după x / după y.

17.5.9 Problema Fossil in the Ice (CodeChef)

enunț • sursă

Problema ne cere să găsim diametrul unui poligon.

Să fixăm un vîrf i al poligonului. Cum am putea găsi vîrful j cel mai depărtat de i? Remarcăm

că, dacă pornim din i și parcurgem complet poligonul, distanța de i este bitonă: întîi crește, atinge maximul în vîrful j dorit, apoi scade. Deci am putea face o căutare binară.

Așadar, o primă soluție ar fi să facem n căutări binare, cîte una pornind din fiecare vîrf i. Dar, dacă avem reflexul de la two pointers bine format, știm că j avansează pe măsură ce i avansează. Mai exact, odată ce găsim punctul cel mai depărtat de $P[0]$ în $\mathcal{O}(n)$, putem găsi restul punctelor cele mai depărtate de $P[1], P[2], \ldots$ în $\mathcal{O}(1)$ amortizat.

17.5.10 Problema Rubarba (Infoarena)

enunț • sursă

Aceasta este exact problema dreptunghiului de arie minimă care cuprinde o mulțime de puncte.

Începem prin a calcula înfășurătoarea convexă. Acum, facem două observații:

1. Fiecare latură a dreptunghiului va conține cel puțin un vîrf. Demonstrația este banală: dacă o latură nu ar conține niciun vîrf, am putea restrînge dreptunghiul din acea direcție.
2. Există o latură a dreptunghiului care conține o latură a poligonului. Această demonstrație este netrivială, dar articolul original demonstrează că, în caz contrar, putem roti dreptunghiul pînă cînd se sprijină pe o latură a poligonului, iar aria dreptunghiului va scădea.

De aici, raționamentul seamănă cu cel anterior. Considerăm pe rînd fiecare latură-suport, între două vîrfuri consecutive i și $i+1$ (iar la final, circular, între $n-1$ și 0). Raportat la direcția acelei laturi, aflăm punctul cel mai depărtat. Am putea afla acel punct printr-o căutare binară, căci distanța de la vîrfuri la latura-suport este bitonă cînd parcurgem poligonul. Distanța maximă găsită ne dă înălțimea dreptunghiului. Prin procedee diferite, dar asemănătoare, putem căuta binar punctul cel mai din stînga și cel mai din dreapta relativ la latura-suport.

De aici, ne dăm seama că putem aplica metoda șublerului rotitor ca să trecem de la o latură-suport la următoarea în $\mathcal{O}(1)$ amortizat. Implementarea mea răspunde elegant (zisei eu cu modestie) la întrebările aparent complexe „Care este punctul cel mai din stînga/dreapta relativ la latura-suport?” și „Care este aria unui dreptunghi rotit?”.

Codul meu are un bug de eficiență, chiar în funcția `rotating_calipers()`. Îl vedeți?

Partea VI

Grafuri

Capitolul 18

Arbori parțiali minimi

Definiție: Dat fiind un graf neorientat conex $G = (V, E)$ cu n noduri și m muchii, unde fiecare muchie are asociat un cost, un **arbore parțial** (engl. *spanning tree*) este o colecție de $n - 1$ muchii care conectează toate cele n noduri. Un **arbore parțial minim** (APM, engl. *minimum spanning tree, MST*) este un arbore parțial care minimizează suma costurilor celor $n - 1$ muchii.

18.1 Algoritmii lui Kruskal și Prim

Nu vom relua aici algoritmii, pe care sper că îi cunoașteți deja de la clasă. Ambii algoritmi sînt relativ intuitivi. Totuși, nu strică să ne familiarizăm și cu demonstrațiile. Ele sînt experimente de gîndire (ușoare) care ne ajută în probleme mai puțin directe.

Iată două demonstrații foarte elegante. Ele sînt inductive: arătăm că, la orice moment, colecția de muchii T pe care o mențin algoritmii este inclusă într-un APM. APM-ul ne este necunoscut, dar el există, iar algoritmii arată cum să extindem T cu o muchie astfel încît noua colecție T' să fie și ea inclusă într-un APM.

Vom introduce noțiunea de **tăietură** (engl. *cut*): O partiție a mulțimii de noduri V în două submulțimi X și Y. Spunem că o muchie **traversează tăietura** dacă are un capăt în X și unul în Y. Spunem că o tăietură **respectă** o mulțime de muchii A dacă nicio muchie din A nu traversează tăietura.

Cu această terminologie, vom arăta principiul comun al celor doi algoritmi. Nu pare, dar există unul! Ambii algoritmi construiesc un set de muchii A. La fiecare pas, ei definesc o tăietură care respectă mulțimea A, apoi caută și adaugă muchia minimă care traversează tăietura. Concret:

1. Algoritmul lui Prim definește tăietura ca fiind X = mulțimea de noduri conectate, Y = restul nodurilor. De aceea menține, pentru fiecare nod din Y, distanța minimă pînă la un nod din X.
2. Algoritmul lui Kruskal nu definește explicit tăietura. Orice tăietură este satisfăcătoare, atîta timp cît ea nu taie componentele conexe existente (pentru orice componentă conexă existentă, toate nodurile sînt de aceeași parte a tăieturii). De aceea algoritmul interzice

muchii între două noduri din aceeași componentă: acele muchii **nu** traversează tăietura.

Cormen (Teorema 21.1) demonstrează următoarea teoremă: Dacă mulțimea curentă A este inclusă într-un APM, și dacă alegem orice tăietură care respectă A, atunci muchia minimă e care traversează acea tăietură poate fi adăugată la A. Muchia e nu închide cicluri în A, iar $A \cup \{e\}$ continuă să fie inclusă într-un APM.

18.2 Proprietăți ale arborilor parțiali minimi

Pagina Wikipedia listează două proprietăți interesante și relativ evidente: proprietatea ciclului și proprietatea tăieturii.

Proprietatea ciclului spune că este necesar ca orice muchie din afara APM să fie cea mai scumpă de pe ciclul pe care îl închide în APM. Editorialul pentru problema Minimum Spanning Tree for Each Edge împinge aceasta cu un pas mai departe, spunînd că condiția este și suficientă: dacă alegem un arbore și dacă toate muchiile din afara acestui arbore au proprietatea ciclului, atunci arborele ales este APM. Editorialul numește asta „criteriul lui Tarjan". Demonstrația este constructivă: putem elimina din graf toate muchiile care încalcă proprietatea ciclului. Ceea ce rămîne este, în mod necesar, un APM.

Toți arborii parțiali minimi ai unui graf au aceeași distribuție de costuri ale muchiilor. Intuitiv, acest lucru se întîmplă deoarece algoritmul lui Kruskal va grupa, la sortare, muchiile de costuri egale.

18.3 Actualizarea APM

Această secțiune este interesantă teoretic, dar nu am întîlnit-o niciodată în probleme de olimpiadă.

Cum actualizăm un APM, T, cînd costul unei muchii se modifică?

- Dacă muchia este în T, iar costul ei scade, T rămîne APM.
- Dacă muchia nu este în T, iar costul ei crește, T rămîne APM.
- Dacă muchia nu este în T, iar costul ei scade, adăugăm muchia la T, ceea ce cauzează închiderea exact a unui ciclu, apoi eliminăm muchia maximă de pe acel ciclu.
  - Demonstrație: pornind de la algoritmul de bază (cu tăieturi). Fie $e = (u, v)$ muchia al cărei cost scade. Fie C ciclul închis de e în T. Fie $f = (x, y)$ muchia de cost maxim din C. Dacă $e = f$, am terminat. Altfel, $c(e) < c(f)$. Să ștergem muchia f din T. Rezultă o partiționare a grafului în două. Muchiile e și f (și posibil altele) sînt muchii care traversează tăietura. Muchia e este muchia minimă care traversează tăietura, căci, dacă ar exista o muchie mai mică g cu $c(g) < c(e) < c(f)$, atunci g ar fi făcut parte din T în locul lui f. Conform algoritmului, rezultă că $T - f + e$ este noul APM.
  - Același lucru îl facem și la adăugarea unei muchii: adăugarea este echivalentă cu scăderea costului de la ∞ la ceva finit.

- Dacă muchia este în T, iar costul ei crește, eliminăm muchia din T și o înlocuim cu cea mai ieftină muchie care reconectează T.
  - Demonstrația este identică: ia naștere o tăietură și selectăm cea mai ieftină muchie care traversează tăietura. Nu știu cît de eficient putem face acest lucru.

Cum actualizăm un APM, T, la adăugarea unui nod z? Nu putem pur și simplu să privim $(V, \{z\})$ ca pe o tăietură, deoarece z poate aduce cu el muchii foarte ușoare, ceea ce face ca T **să nu mai fie APM** pentru mulțimea sa de noduri. Putem folosi o parcurgere DFS specifică problemei, în $\mathcal{O}(n)$, sau putem rula algoritmii lui Kruskal și Prim pe mulțimea formată din T și muchiile nou adăugate de z. De ce?

18.4 Probleme

18.4.1 Problema Arbore parțial de cost minim (Infoarena)

enunț • surse

Problema este educațională. Vom citi sursele și vom explica algoritmii.

Algoritmul lui Kruskal rulează în $\mathcal{O}(m \log m)$ datorită sortării. Puteam face sortarea în $\mathcal{O}(m + c)$ dacă distribuiam muchiile în liste după cost, dar am decis să nu încarc programul.

Algoritmul lui Prim rulează în $O(m \log n)$. Într-adevăr, fiecare muchie evaluată poate duce la o îmbunătățire a distanței fiului, ceea ce necesită o inserare în coada de priorități. Remarcăm că, în această implementare, un nod u poate avea multiple copii în coadă, dacă distanța lui este optimizată de mai multe ori înainte ca u să fie selectat ca minim și adăugat la arbore.

Putem insista ca coada să conțină noduri distincte, dar nu este banal.

- CP Algorithms folosește un set. Cînd constatăm că distanța unui nod scade, îi cunoaștem vechea distanță d, deci putem căuta și șterge vechea pereche (u, d) din set.
- Putem folosi și un heap, dar trebuie să ni-l implementăm singuri. Fiecare nod menține și un pointer (un indice întreg) la poziția sa în heap. După modificarea distanței, acel nod din heap poate să urce sau să coboare, după caz.

18.4.2 Problema Autobuze3 (AGM 2015)

enunț • sursă

Și aceasta este o problemă relativ directă. Costul minim este dat de arborele parțial minim. Pentru trasarea drumurilor este necesar să parcurgem arborele din frunze spre rădăcină. Pentru a limita la 25 numărul de transferuri ale șoferilor, trebuie ca, atunci cînd un autobuz sosește din nodul fiu în nodul părinte, să-l comparăm cu autobuzul existent în nodul părinte și să transferăm autobuzul mai gol în cel mai plin (*small to large*).

Detalii de implementare

Am stat puțin pe gînduri dacă să folosesc algoritmul lui Kruskal sau pe al lui Prim. Voi ce părere aveți? Realitatea este că Kruskal pare mai greu de greșit și este mai scurt cu cîteva linii. În cazul de față, **poate** merita folosit Prim, care oferă pentru fiecare nod u pointerul la părinte (adică la nodul care l-a adăugat pe u la arbore). Sînt destul de sigur că acel cod ar cîștiga la memorie, căci nu mai este necesar DFS-ul. Dar în rest... pare mai greu de scris, iar Prim este ceva mai lent.

Pentru a reduce numărul de transferuri, meritau făcute două iterații în DFS: una pentru identificarea fiului heavy și una pentru transferul celorlalți fii, o singură dată. Dar, așa cum am învățat în capitolul 9, este mai ușor de codat, și suficient de eficient, să facem unificarea la revenirea din fiecare fiu. Încă rămîne valabil că fiecare șofer va fi transferat de cel mult $\log n$ ori.

Pentru reutilizarea `vector`-ilor între teste, îi ștergem cu `clear()`. Putem economisi niște memorie (de la 36 MB la 30 MB) dacă apelăm și `shrink_to_fit()` după `clear()`. Altfel, `clear()` nu garantează recuperarea memoriei.

18.4.3 Problema Rusuoaica (FMI No Stress 9 Warmup)

enunț • sursă

(Sper că știți cum se scrie corect *rusoaică*. Probabil scrierea greșită este doar o glumă între colegi.)

Problema este un experiment de gîndire pornind de la Kruskal. Putem opri algoritmul la costul A, cu semnificația că vindem toate muchiile mai scumpe decît A. Apoi, dacă rămînem cu k componente conexe, le putem interconecta cu $k-1$ muchii de cost A. Pentru simplitate, încă de la citire renunțăm la muchiile de cost mai mare decît A.

Ca detaliu de implementare, am ales să las structura de mulțimi disjuncte să țină și evidența numărului de arbori din pădure.

18.4.4 Problema MinOr Tree (Codeforces)

enunț • sursă

Problema este mai mult de idee; implementarea este elementară.

Putem încerca să sortăm crescător muchiile, în ideea să lăsăm algoritmul lui Kruskal să conecteze graful cu muchii ale căror biți sînt cît mai nesemnificativi. Dar garanția de optimalitate nu se mai aplică la OR la fel ca la sume. Odată ce Kruskal decide că este obligatoriu să folosească muchii cu bitul cel mai semnificativ (să spunem) 6, s-ar putea să fie optim să ștergem unele dintre muchiile anterioare, ca să evităm să folosim bitul 5.

În loc de asta, vom construi soluția bit cu bit. Să spunem că masca OR a tuturor costurilor ocupă 20 de biți.

- Încercăm să vedem dacă putem să conectăm graful doar cu muchii care **nu** folosesc bitul 19. Dacă da, ștergem acele muchii.

- Încercăm să vedem dacă putem să conectăm graful doar cu muchii care folosesc sau nu bitul 19, conform răspunsului anterior, dar **nu** folosesc bitul 18. Dacă da, ștergem acele muchii.
- Etc.

Ca optimizare, dacă un bit nu apare deloc în niciuna dintre muchii, putem omite complet acea iterație.

Ideea de APM ne trimite, de fapt, pe o pistă falsă. Problema este una de conexitate. De aceea nici nu trebuie să sortăm muchiile.

18.4.5 Problema 0-1 MST (Codeforces)

enunț • surse

Această problemă simpatică și un pic anapoda ne cere să găsim APM-ul într-un graf complet cu n noduri, în care toate muchiile au cost 0 în afară de m dintre ele, care au cost 1. Provocarea, desigur, este să facem asta în $\mathcal{O}(m+n)$ sau pe-acolo, fără a depinde de numărul real de muchii, care este $\mathcal{O}(n^2)$.

Problema este ușoară și are multiple soluții. Iată două dintre ele.

Soluția cu Kruskal

Putem adapta algoritmul lui Kruskal. Procesăm doar muchiile de 0 (cele care **nu apar** la intrare). Doar că trebuie să facem asta în $\mathcal{O}(m)$ (numărul de muchii care **apar** la intrare). Dacă rămînem cu k componente, atunci costul arborelui este $k-1$.

Pornim de la observația cu care ne-am mai întîlnit: nu contează ce muchii selectăm, ci cîte putem selecta fără a închide cicluri. Procesăm nodurile de la 1 la n și construim structura de mulțimi disjuncte ca de obicei. Ne permitem să petrecem în fiecare nod u timp $\mathcal{O}(\deg_1(u))$, unde $\deg_1(u)$ este numărul de muchii de cost 1 care pornesc din u (cele date la intrare).

Menținem pădurea de mulțimi disjuncte ca în algoritmul lui Kruskal, dar în plus reținem și mărimea fiecărei mulțimi. Pentru nodul curent u contorizăm cîte muchii de cost 1 se duc către fiecare mulțime disjunctă. Numim o mulțime **saturată** dacă în ea intră un număr de muchii de cost 1 egal cu mărimea ei. Dacă o mulțime nu este saturată, atunci există și muchii de cost 0 către ea (nu uitați că graful este complet). Deci, conform algoritmului lui Kruskal, unim mulțimea lui u cu toate mulțimile nesaturate.

Surprinzător, ne permitem să testăm fiecare mulțime din structură. Efortul pare să fie $\mathcal{O}(n)$ per nod, $\mathcal{O}(n^2)$ în total, dar realitatea este mai bună. Pentru un nod u, există cel mult $\deg_1(u)$ mulțimi saturate, deci efortul de a considera toate mulțimile saturate este $\mathcal{O}(m)$. Cît despre mulțimile nesaturate, acelea sînt imediat unificate și dispar din lista de componente, deci efortul global necesar pentru a le procesa este $\mathcal{O}(n \log^* n)$.

Soluția cu BFS

Există și o soluție frumoasă care folosește doar parcurgeri și conexitate, ca la numărarea componentelor conexe: din fiecare nod nevizitat pornim o parcurgere folosind doar muchiile de cost 0. Trebuie să avem grijă ca vizitarea fiecărui nod să coste $\mathcal{O}(\deg_1(u))$. Să denumim lista de muchii de cost 1 care pleacă din noul u **lista de nonadiacență** a lui u. Implementarea mea stochează listele de nonadiacență în cîte un `unordered_set` pentru fiecare nod u.

Apoi, menținem o listă L de noduri nevizitate. Cînd scoatem din coada BFS un nod u consultăm lista L și inserăm în coada BFS toate nodurile care nu apar în lista de nonadiacență a lui u (așadar, cele aflate la distanță 0 de u).

Poate părea că efortul este $\mathcal{O}(n^2)$, căci pentru fiecare nod vom consulta lista L care are $\mathcal{O}(n)$ noduri. Dar, în realitate, există două scenarii pentru fiecare element v din L:

1. v este în lista de adiacență a lui u, și deci rămîne în lista L. Dar v nu se va afla în această situație decît de $\deg(v)$ ori, deci efortul global este $\mathcal{O}(m)$.
2. v nu este în lista de adiacență a lui u. Atunci îl scoatem pe v din L și îl inserăm în coada BFS. Fiecare element poate fi eliminat doar o dată din L, deci efortul global este $\mathcal{O}(n)$.

De aceea, această soluție obține complexitatea $\mathcal{O}(m+n)$.

De amorul artei, am scris și o sursă optimizată care (1) folosește o coadă proprie și (2) folosește un singur vector în loc de `unordered_set` pentru a testa existența muchiilor.

18.4.6 Problema Minimum Spanning Tree for Each Edge (Codeforces)

enunț • sursă

Ideea teoretică

Iată o soluție care dă răspunsul corect, dar este prea lentă. Pentru a forța APM-ul să includă o muchie (u, v), setăm costul acelei muchii la 0. Alternativ, preinițializăm structura de mulțimi disjuncte (în algoritmul lui Kruskal) respectiv mulțimea de noduri incluse (în algoritmul lui Prim) ca să includem muchia (u, v). Dar nu ne permitem să calculăm un APM pentru fiecare muchie.

Soluția eficientă procedează astfel. Calculăm APM-ul o singură dată. Apoi, pentru fiecare muchie (u, v), aflăm un APM care include muchia:

1. Adăugăm muchia (u, v) la APM. Aceasta creează exact un ciclu.
2. Eliminăm ciclul ștergînd cea mai grea muchie de pe calea (u, v) din APM.

Să demonstrăm că această abordare este corectă. Fie $\mathcal{A}$ APM-ul nostru de cost C, fie c costul muchiei (u, v) și fie c_1 costul celei mai scumpe muchii de pe ciclul închis de (u, v). Atunci vom da răspunsul $C + c - c_1$.

Cînd ar putea fi acest răspuns greșit? Să presupunem că există un al doilea APM $\mathcal{A}'$, cu același cost C dar cu o structură foarte diferită, în care cea mai scumpă muchie de pe ciclul închis de

(u, v) are un cost $c_2 > c_1$. Atunci am avea că $C + c - c_2 < C + c - c_1$. Să vedem unde este contradicția.

Observăm că muchia c_2 nu poate face parte din $\mathcal{A}$. Într-adevăr, dacă ar face parte, ea s-ar afla pe ciclul închis de (u, v) și am fi selectat-o pe ea în locul lui c_1. Dar atunci muchia c_2 închide un ciclu în $\mathcal{A}$. De aici apare contradicția. Din $\mathcal{A}$ știm că c_2 este cea mai scumpă muchie de pe acel ciclu (altfel am putea optimiza APM-ul $\mathcal{A}$). Dar, din proprietatea ciclului, știm că muchia c_2 nu poate face parte din niciun APM, deci nici din $\mathcal{A}'$.

Implementare cu LCA și *binary lifting*

Așadar, trebuie să răspundem la m interogări de maxime pe căi. Știm deja abordarea clasică, în $\mathcal{O}(\log n)$ per interogare: alegem o rădăcină pentru arbore, să zicem nodul 1, și precalculăm informații de LCA cu *binary lifting*. Fiecare pointer reține și valoarea maximă a unei muchii pe care o traversează.

Implementare cu mulțimi disjuncte modificate

Soluția mea împrumută o altă idee, pe care o vom detalia. Ea folosește o structură de date S care este „un fel de” *disjoint set forest* de complexitate $\mathcal{O}(\log n)$ per operație.

Concret, procesăm muchiile în ordine crescătoare a costului. Începem să unificăm componentele conexe conform algoritmului lui Kruskal. Dar nu folosim compresia căii, căci ne interesează ca S să păstreze o informație esențială, care s-ar pierde prin reorganizare. Pentru orice pereche de noduri deja conectate, u și v, dorim ca S să faciliteze găsirea muchiei celei mai scumpe de pe calea $u - v$.

Cînd procesăm muchia (u, v) de cost c, dacă u și v provin din componente diferite, găsim rădăcinile u' și v' ca de obicei. Apoi legăm u' de v' și notăm costul c. Aceasta garantează că, pentru a ajunge (în S) de la orice nod din componenta lui u' la orice nod din componenta lui v', trebuie să trecem prin muchia (u', v').

Dacă componenta lui u' este mai mare, legăm v' la u', altfel invers. Aceasta garantează că numărul de muchii de pe orice cale este logaritmic.

Cum folosim, concret, structura S ca să aflăm răspunsul pentru fiecare muchie? Dacă u și v provin din componente diferite, știm din algoritmul lui Kruskal că (u, v) face parte din APM, deci răspunsul pentru ea va fi costul APM-ului, odată ce îl determinăm.

Dacă, în schimb, procesăm o muchie (u, v) și determinăm că nodurile sînt deja conectate, atunci urcăm cu u și v prin S pînă cînd se întîlnesc, apoi returnăm muchia maximă întîlnită pe drum.

18.4.7 Problema Apm2 (Infoarena)

enunț • sursă

Problema este foarte similară cu cea anterioară (Minimum Spanning Tree for Each Edge). Dar merită reluată observația teoretică.

Intuiția ne spune să calculăm un APM. Apoi, pentru o interogare (u, v), răspunsul este $c - 1$, unde c este costul maxim de pe calea (u, v) din APM. Justificarea este că dorim să punem un cost cît mai mare pe (u, v), dar suficient de mic încît, la refacerea APM-ului, muchia (u, v) să rămînă în APM.

Cum demonstrăm asta? Nu putem demonstra doar pe APM-ul particular pe care l-a construit programul nostru, ci pe orice APM. Să studiem mai atent algoritmul lui Kruskal. Motivul pentru care pe o cale (u, v) există o muchie de cost maxim c este că, după procesarea muchiilor de cost $\leq c - 1$, componentele lui u și v nu erau încă conectate.

De aceea, dacă pentru interogarea (u, v) alegem costul $c - 1$, atunci algoritmul lui Kruskal ar alege garantat muchia. Detaliu la care vom reveni: algoritmul ar alege muchia indiferent unde ar fi ea sortată între alte muchii de cost $c - 1$.

Dacă punem costul muchiei $(u, v) = c$, atunci în mod evident algoritmul poate omite muchia (u, v), căci poate uni componentele folosind cealaltă muchie de cost c.

Implementarea mea folosește același pseudo-*disjoint set forest* cu timp logaritmic. Dar, ca și mai înainte, soluția cu LCA funcționează și ea perfect.

18.4.8 Problema Edges in MST (Codeforces)

enunț • sursă

Problema duce un pas mai departe experimentele de gîndire despre APM. Să considerăm că algoritmul lui Kruskal a procesat toate muchiile de cost mai mic decît c și a obținut niște componente conexe. Subliniem că partiționarea în componente nu depinde de sortarea pe care algoritmul lui Kruskal s-a întîmplat să o aleagă pentru muchii de costuri egale. Exemplu: dintr-un triunghi de muchii de cost 4, Kruskal va alege două. Cele trei noduri vor fi conectate indiferent de muchiile alese.

Atunci să considerăm graful componentelor, în care fiecare nod corespunde unei componente conexe, și să considerăm „pachetul" de muchii de cost c.

- O muchie va apărea în **toate APM-urile** dacă este punte.
- Altfel, o muchie va apărea în **unele APM-uri** dacă unește două componente distincte.
- Altfel, o muchie unește două noduri din aceeași componentă și nu va apărea în **niciun APM.**

Vom defini noțiunea de **punte**, iar deocamdată vom lua fără demonstrație codul de găsire a punților. Vom reveni la el în capitolul 19.

Implementarea constă din algoritmul lui Kruskal + un DFS pentru fiecare bloc de muchii egale. Atenție la două detalii de implementare:

- Fiecare DFS pe un pachet de k muchii trebuie să fie $\mathcal{O}(k)$. Orice `for` neatent, în $\mathcal{O}(n)$, poate duce complexitatea totală la $\mathcal{O}(n^2)$ dacă toate muchiile au costuri distincte.
- Deși graful inițial este simplu, în graful componentelor pot exista muchii paralele (între perechi de noduri din aceleași componente). Este nevoie de puțină atenție în algoritmul lui Tarjan de găsire a punților. Este acceptabil (și necesar) să luăm în calcul toate muchiile care duc înapoi la părinte, doar nu fix pe cea pe care am venit. Sursa mea folosește indicele original al muchiei. CP Algorithms oferă o soluție mai generală.

18.4.9 Problema Envy (Codeforces)

enunț • sursă

Să pornim (din nou...) de la algoritmul lui Kruskal. Ne reamintim că, după procesarea muchiilor de cost $\leq c$, structura componentelor conexe va fi identică. Pot varia doar conexiunile interne din fiecare componentă. Așadar, putem procesa muchiile din fiecare interogare grupate după greutăți.

Cînd va avea o interogare răspunsul „nu”? Cînd unul dintre aceste grupuri include muchii care închid cicluri. De exemplu, dacă o interogare include (printre altele) trei muchii de greutate 10, iar la acel moment în algoritmul lui Kruskal muchiile unesc componentele $C_1 - C_2$, $C_2 - C_3$ și $C_3 - C_1$, interogarea are răspunsul „nu”.

Cum testăm existența acestor cicluri? Editorialul propune, pentru fiecare interogare și pentru fiecare grup de muchii de același cost c, cîte un DFS doar pe acele muchii **la momentul potrivit** în algoritmul lui Kruskal, adică înainte de a procesa muchiile de cost c.

Soluția mea folosește un *disjoint set forest* cu suport pentru *undo*: procesăm muchiile de cost c din interogare, verificăm dacă vreuna dintre ele închide un ciclu, apoi restaurăm pădurea la starea anterioară. Abia la finalul interogărilor pentru toate greutățile c le încorporăm (permanent) în pădure.

Aveam așteptări mari de la implementare, dar este cam lentă. 🥲

18.4.10 Problema DFS Trees (Codeforces)

enunț • sursă

Aceasta este mai puțin o problemă de APM și mai mult un experiment de gîndire despre DFS.

Teoria

Să definim doi termeni specifici DFS-ului în grafuri neorientate. Cînd ajungem într-un nod u,

- **Muchiile de arbore** sînt acele muchii (u, v) pe care DFS-ul se reapelează din v.
- **Muchiile înapoi** sînt acele muchii (u, v) pe care DFS-ul nu se reapelează, căci v a fost vizitat anterior.

Acum, să observăm că APM-ul este unic, costurile fiind distincte. Îl putem calcula cu algoritmul lui Kruskal (este foarte convenabil, căci primim deja muchiile în ordinea crescătoare a costului). Apoi, întrebarea pentru fiecare nod de pornire $1 \leq x \leq n$ este dacă DFS-ul va folosi ca muchii de arbore exact cele $n-1$ muchii din APM.

Să considerăm întrebarea echivalentă: cum știm dacă DFS-ul din x va evita cele $m-n+1$ muchii care nu fac parte din APM? Să considerăm o astfel de muchie, $e = (u, v)$. Dacă e nu face parte din APM, înseamnă că închide un ciclu (u, v) în APM. Acum (pentru a suta oară în acest curs), să schimbăm modul de agregare a datelor. În loc să fixăm x și să luăm în calcul toate muchiile, să fixăm muchia e și să luăm în calcul toate nodurile de pornire x. Marcăm ca „rău" orice nod de pornire care vizitează muchia e ca muchie de arbore. După ce luăm în calcul toate muchiile, răspunsul va fi `0` pentru nodurile care au fost marcate ca „rele" cel puțin o dată, `1` pentru restul.

Am ajuns astfel la întrebarea: pentru o muchie $e = (u, v) \notin$ APM, ce noduri de pornire x vizitează e ca muchie de arbore? Fie C ciclul format de e cu lanțul $u - v$ din APM. Fie y primul nod descoperit de DFS dintre toate cele din C. Dacă $y = u$, atunci DFS-ul nu va pleca pe muchia e, care este scumpă, ci va pleca (la un moment dat) spre v. Tot lanțul $u - v$, inclusiv v, va deveni subarbore al lui u în DFS.

Aceasta este observația crucială despre DFS: odată ce marchează u ca vizitat și pornește pe lanțul care duce spre v, DFS-ul nu va mai reveni în u pînă nu termină de explorat toate nodurile noi, care includ tot lanțul $u - v$. Cîndva în acest proces DFS-ul va descoperi nodul v și muchia e și va constata că este muchie înapoi. Nu este garantat că DFS-ul va explora lanțul în ordinea din APM. Pot exista alte muchii și lanțuri care să încîlcească traseul. Dar este garantat că va explora tot lanțul și va ajunge la v.

Același argument se aplică și pentru $y = v$. Dacă în schimb y este alt nod de pe lanț, atunci se aplică argumentul invers. Fie f și g muchiile către cele două noduri vecine cu y în C. DFS-ul va porni pe una dintre ele, să zicem pe f. Tot ciclul va deveni subarbore al lui y prin f, iar muchia g va fi muchie înapoi. Aceasta este o greșeală, căci g este muchie din APM și DFS-ul o omite.

Implementarea

Așadar, pentru fiecare muchie $e = (u, v)$ din afara APM-ului, nodurile care o vor traversa în DFS, și care trebuie marcate ca „rele", sînt cele din care, pornind, descoperim ciclul C printr-un nod diferit de u și de v. Sună monstruos de implementat! 😨

În realitate, este suficient să considerăm doar APM-ul. Dacă „atîrnăm" acest APM ca pe o frînghie de nodurile u și v, atunci trebuie să marcăm ca „rele" toate nodurile dintre u și v și toți subarborii acelor noduri. În practică arborele este înrădăcinat, deci iau naștere două cazuri:

1. Dacă u este strămoș al lui v, atunci marcăm ca rău tot subarborele fiului w al lui u care pornește către v, cu excepția subarborelui lui v. Similar dacă v este strămoș al lui u.
2. Dacă u și v nu sînt în relația strămoș-descendent, atunci marcăm ca rău tot arborele cu excepția subarborilor lui u și v.

Putem face aceste operații cu vectori de diferențe pe arbore:

1. Pentru cazul (1), notăm +1 în w și -1 în v.
2. Pentru cazul (2), notăm +1 în rădăcină, -1 în u și -1 în v.
3. Apoi propagăm valorile în jos (fiecare nod calculează suma valorilor de la el la rădăcină).

La final, nodurile „rele" sînt cele care au valori pozitive, nodurile bune sînt cele care au valori 0.

Capitolul 19

Conectivitate

19.1 Sortarea topologică

Sortarea topologică a unui graf orientat înseamnă să-i enumerăm cele n noduri într-o ordine astfel încît toate muchiile să meargă doar de la stînga spre dreapta. Un exemplu intuitiv este graful de dependințe între noțiunile învățate la școală: geometria 2D este necesară pentru geometria 3D etc. Wikipedia enumeră multe alte exemple.

Un graf orientat admite o sortare topologică dacă și numai dacă este aciclic. Un astfel de graf se numește dag (engl. ***d**irected **a**cyclic **g**raph*). Condiția este necesară deoarece, dacă graful conține un ciclu, atunci orice ordine în care enumerăm nodurile ciclului în sortare va face ca muchia dintre ultimul nod și primul să meargă spre stînga. Condiția este și suficientă, iar demonstrația este constructivă.

Problema educațională Course Schedule ne cere exact să sortăm topologic un graf orientat sau să raportăm dacă el conține un ciclu.

Există doi algoritmi clasici, ambii în $\mathcal{O}(m + n)$. Ambii necesită cod similar de lung, cu viteze și consumuri de memorie similare. Este bine să fiți familiari cu ambii, pentru că unele probleme se mulează mai bine peste unul dintre algoritmi.

19.1.1 Algoritmul lui Kahn

Acest algoritm este practic un BFS. Inițializăm o coadă cu nodurile cu grad la intrare 0. Acestea pot fi enumerate primele în sortarea topologică, în orice ordine. Explorăm graful, tipărind nodurile pe măsură ce le scoatem din coadă. De cîte ori un nod a fost explorat de un număr de ori egal cu gradul său la intrare, îl introducem și pe el în coadă, cu semnificația că și el poate fi tipărit.

Graful conține un ciclu dacă coada rămîne goală înainte să explorăm toate nodurile.

19.1.2 Algoritmul bazat pe DFS

Pornim un DFS dintr-un nod oarecare. La fiecare nivel recursiv al DFS-ului, la revenirea din descendenți adăugăm nodul curent la ordinea topologică. Dacă după încheierea DFS-ului mai există noduri nedescoperite, alegem la întîmplare unul dintre acestea și pornim un nou DFS. De remarcat că algoritmul produce o sortare topologică **inversă**.

Graful conține un ciclu dacă DFS-ul încearcă, la orice moment, să viziteze un nod gri (un nod din stiva DFS).

19.2 Componente tare conexe

Într-un graf orientat $G = (V, E)$, o **componentă tare conexă** (abr. *CTC*, engl. *strongly connected component*, abr. *SCC*) este un subgraf maximal în care pentru orice pereche de noduri u și v, v este accesibil din u și invers. Notăm acest lucru cu $u \rightsquigarrow v$ și $v \rightsquigarrow u$.

Vom folosi această figură din *Introduction to Algorithms*.

Problema educațională Ctc ne cere exact să tipărim componentele tare conexe ale unui graf orientat.

19.2.1 Terminologie

Graful transpus se notează cu $G^T = (V, E^T)$ și se obține inversînd sensul tuturor muchiilor din E. Remarcăm că G^T are aceleași componente tare conexe ca și G: dacă $u \rightsquigarrow v$ și $v \rightsquigarrow u$ în G, parcurgînd căile în sens invers obținem că $v \rightsquigarrow u$ și $u \rightsquigarrow v$ în G^T.

Graful componentelor se notează cu G^{SCC} și este definit astfel:

- Contractăm fiecare componentă tare conexă, cu nodurile și cu muchiile ei, într-un singur nod.
- Unificăm toate muchiile dintre două componente într-o singură muchie.

Graful componentelor este un dag. Demonstrația este prin reducere la absurd: dacă în G^{SCC} ar exista un ciclu prin două noduri u și v, atunci în graful original ar exista căi între orice două noduri din componentele tare conexe corespunzătoare lui u și v, deci cele două componente ar fi una și aceeași.

Generalizăm **timpii de descoperire și de părăsire** a unui nod ($t_{in}[u]$ și $t_{out}[u]$) la submulțimi de noduri. Pentru $U \subseteq V$, definim:

- $t_{in}[U] = \min_{u \in U} t_{in}[u]$
- $t_{out}[U] = \max_{u \in U} t_{out}[u]$

Vom studia doi algoritmi, comparabili ca viteză. Ca și în cazul sortării topologice, vă încurajez să fiți familiari cu ambii.

19.2.2 Algoritmul lui Kosaraju

Acest algoritm face două parcurgeri DFS pentru a găsi componentele tare conexe:

1. Face o parcurgere DFS în G și reține $t_{out}[u]$ pentru fiecare nod u.
2. Calculează G^T.
3. Face o parcurgere DFS în G^T, considerînd nodurile în ordinea descrescătoare a lui t_{out}.
4. Fiecare submulțime găsită de DFS-ul de la pasul 3 este o componentă tare conexă.

De ce funcționează acest algoritm? Să studiem relația între valorile lui t_{out} (în graful original) și graful componentelor. Fie două componente C și C' între care există o muchie (u, v) cu $u \in C$ și $v \in C'$. Atunci putem demonstra că $t_{in}[C] > t_{in}[C']$. Într-adevăr, există două cazuri:

1. Dacă C este descoperită prima, fie $x \in C$ primul nod vizitat. Toate nodurile din C și din C' devin descendenți ai lui x în arborele DFS, deci nodul x va fi ultimul părăsit. Rezultă că $t_{out}[C] = t_{out}[x] > t_{out}[C']$.
2. Dacă C' este descoperită prima, fie $y \in C'$ primul nod vizitat. Toate nodurile din C' devin descendenți ai lui y în arborele DFS, dar nu și cele din C. Componenta C' va fi vizitată complet, apoi DFS-ul va reveni la bucla principală și la un moment viitor va vizita și componenta C. Rezultă că $t_{out}[C] > t_{out}[C']$.

În graful transpus G^T avem relația inversă: pentru orice muchie $(u, v) \in E^T$ cu $u \in C$ și $v \in C'$ vom avea $t_{out}[C] < t_{out}[C']$. **Atenție**: valorile t_{out} sînt cele calculate în graful original.

De aici rezultă corectitudinea algoritmului. Pasul 3 începe cu nodul u care maximizează $t_{out}[u]$. Acel DFS va explora o componentă C (în G^T) din care nu vor exista muchii spre alte componente. Cu alte cuvinte, DFS-ul nu se va revărsa în nicio altă componentă, ci va tipări strict nodurile din C. Apoi va alege alt nod de pornire dintr-o altă componentă C' care nu avea muchii spre alte componente (decît cel mult spre C, care a fost deja tipărită). Prin inducție, la orice moment cînd bucla principală va apela algoritmul DFS recursiv, algoritmul va tipări exact o componentă tare conexă.

Nu întîmplător, acest algoritm seamănă (un pic) cu o sortare topologică. Într-adevăr, ordinea în care componentele sînt tipărite este o sortare topologică a grafului G^{SCC} sau, alternativ, o sortare invers topologică a grafului G^{SCC^T}.

- Adevărat sau fals? Putem ca, la pasul 3, să parcurgem tot graful original, dar în ordinea crescătoare a valorilor t_{out}? Dați o demonstrație sau un contraexemplu. Indiciu: probabil nu folosim de 50 de ani un algoritm mai complicat decît trebuie. 🙂

Iată și un detaliu de implementare. Multe programe de pe Internet (CP Algorithms, Topcoder, GeeksforGeeks) țin în memorie două grafuri cu cod de genul:

```
while (m--) {
  cin >> a >> b;
  g[a].adj.push_back(b);
  g[b].rev_adj.push_back(a);
```

```
}
```

Această abordare folosește $\mathcal{O}(M + N)$ memorie suplimentară (N vectori cu un total de M elemente). Putem inversa listele de adiacență în $\mathcal{O}(M)$, atunci cînd devine necesar. Folosim doar $\mathcal{O}(N)$ memorie suplimentară pentru capetele de listă:

```
for (int u = 1; u <= n; u++) {
  for (int v: g[u].adj) {
    g[v].rev_adj.push_back(u);
  }
  g[u].adj.resize(0);
}
```

19.2.3 Algoritmul lui Tarjan

Acest algoritm face o singură parcurgere DFS. Pentru fiecare nod u stocăm două valori:

- $d[u]$ este adîncimea nodului
- $l[u]$ (*lowpoint*) este adîncimea minimă a oricărui vecin al oricărui nod din subarborele lui u.

Observația crucială este că, dacă niciun nod din subarborele lui u nu poate urca mai sus decît u, adică dacă $l[u] = d[u]$, atunci

- u este punctul de intrare într-o CTC;
- muchia de la părintele din DFS al lui u la u este o muchie între două CTC din G^{SCC}.

Pentru enumerarea efectivă a nodurilor din fiecare componentă, punem nodurile într-o stivă pe măsură ce le descoperim, dar **nu le scoatem din stivă** la revenirea din DFS. În schimb, cînd determinăm că un nod u este punctul de intrare într-o CTC, atunci scoatem toate nodurile din stivă, pînă la u inclusiv. Ele formează o CTC și le putem procesa după cum cere problema (de exemplu, le putem eticheta cu aceeași etichetă numerică).

Acest algoritm generează componentele într-o ordine invers topologică a grafului G.

19.3 Componente biconexe, punți, puncte de articulație

Ne îndreptăm acum atenția către probleme similare de conexitate în grafuri neorientate. Vom urmări exemplul propus în Cormen, fig. 20.10.

19.3.1 Definiții

- Un **punct de articulație** este un nod prin a cărui eliminare numărul de componente conexe crește.
- O **punte** este o muchie prin a cărei eliminare numărul de componente conexe crește.
- Un graf este **biconex** dacă, prin eliminarea oricărui nod, graful rămîne conex. Echivalent, un graf biconex este un graf fără puncte de articulație.

- O **componentă biconexă** este un subgraf biconex maximal.

Ultima definiție de mai sus este de pe Wikipedia. Cormen propune o definiție alternativă: o componentă biconexă este un set maximal de muchii astfel încît pentru orice pereche de muchii există un ciclu simplu care trece prin ambele. Diferența apare în cazul punților: o punte și nodurile de la capetele ei formează o componentă biconexă conform definiției de mai sus, dar nu și conform definiției din Cormen.

Conform Wikipedia, un punct de articulație poate face parte din mai multe componente biconexe. De aceea, Wikipedia definește componentele biconexe referitor la muchii, nu la vîrfuri. Mai exact, componentele biconexe sînt o partiție a mulțimii de muchii E.

Nu trebuie să vă împiedicați de aceste diferențe. Cîtă vreme știți ce doriți să codați, vă va fi ușor.

În problemele de astăzi vom întîlni și conceptul de **graf k-conex pe muchii** ca fiind un graf neorientat care rămîne conex dacă eliminăm mai puțin de k muchii. De exemplu, un graf 1-conex este pur și simplu un graf conex, iar un graf 2-conex este un graf fără punți.

19.3.2 Puzzles

- Adevărat sau fals? O muchie (u, v) aflată între două puncte de articulație este punte.
- Adevărat sau fals? Capetele oricărei punți sînt puncte de articulație.
- Adevărat sau fals? Un punct de articulație are cel puțin o muchie adiacentă care este punte.

19.3.3 Algoritmul Hopcroft-Tarjan

Este remarcabil că putem calcula toate aceste informații printr-o singură parcurgere DFS. Ca și la aflarea componentelor tare conexe, pentru fiecare nod u stocăm valorile $d[u]$ și $l[u]$. Atunci:

- Rădăcina parcurgerii este punct de articulație dacă și numai dacă are doi sau mai mulți fii.
- Un nod u diferit de rădăcină este punct de articulație dacă și numai dacă are cel puțin un fiu v pentru care $l[v] \geq d[u]$. Cu alte cuvinte, un fiu care nu poate accede la restul grafului dincolo de u.
- O muchie (u, v) vizitată din u spre v este punte dacă $l[v] > d[u]$ sau, echivalent, dacă $l[v] = d[v]$. Cu alte cuvinte, v nu poate ajunge la u decît prin muchia (u, v).

Ca și la aflarea CTC, pentru partiționarea muchiilor în componente biconexe, le vom pune într-o stivă cînd le descoperim. La revenirea din recursivitate, dacă nodul curent u este punct de articulație, el îi datorează asta unui fiu v. Scoatem din stivă toate muchiile pînă la (u, v) inclusiv; ele formează o componentă biconexă.

Iată mai jos codul. El calculează toate proprietățile. Dacă problema nu le cere, desigur, codul se scurtează. Bunăoară, stiva nu este necesară decît pentru componentele biconexe.

```
// Stiva de la (u, v) pînă la vîrf formează o CBC.
void print_bcc(int u, int v) {
  printf("CBC:");
```

```
  do {
    --ss;
    printf(" %d-%d", st[ss].u, st[ss].v);
  } while (st[ss].u != u || st[ss].v != v);
  printf("\n");
}

void dfs(int u, int parent) {
  bool is_ap = false;
  int num_children = 0;
  nd[u].d = nd[u].low = ++time;

  for (int v: nd[u].adj) {
    if (!nd[v].d) {

      // Vizitează fiul și vezi cît poate urca.
      num_children++;
      st[ss++] = { u, v };
      dfs(v, u);
      nd[u].low = min(nd[u].low, nd[v].low);

      // Dacă v nu poate ajunge la u (sau mai sus), atunci (u,v) este punte.
      // În funcție de definiție, punțile pot sau nu să fie CBC de sine stătătoare.
      if (nd[v].low > nd[u].d) {
        printf("punte: %d-%d\n", u, v);
      }

      // Dacă v nu poate ajunge mai sus de u, atunci (1) o CBC începe la v și
      // (2) fie u este rădăcina, fie este punct de articulație.
      if (nd[v].low >= nd[u].d) {
        is_ap = true;
        print_bcc(u, v);
      }

    } else if (v != parent) {

      // Muchie înapoi -- asigură-te că nu o traversăm mergînd înainte.
      if (nd[v].d < nd[u].d) {
        nd[u].low = min(nd[u].low, nd[v].d);
        st[ss++] = { u, v };
      }

    }
  }

  // Rădăcina este punct de articulație dacă are copii multipli.
  // Alte noduri sînt puncte de articulație dacă au fost marcați de vreunul dintre copii.
  if ((!parent && (num_children > 1)) || (parent && is_ap)) {
    printf("PA: %d\n", u);
  }
```

```
}
```

Întrebări despre cod:

- De ce nu este nevoie să mai apelăm `print_bcc()` o dată la final?
- De ce tipărim componenta cînd aflăm că u este punct de articulație și nu cînd aflăm că (u, v) este punte?
- De ce optimizăm $l[u]$ o dată cu $l[v]$ și o dată cu $d[v]$?
- De ce ne asigurăm că $d[v] < d[u]$ pentru muchiile „înapoi”?

19.4 Probleme

19.4.1 Problema Course Schedule (CSES)

enunț • sursă

Problema este educațională de sortare topologică. Includ surse cu cei doi algoritmi studiați.

19.4.2 Problema Book (Codeforces)

enunț • surse

Problema se duce destul de direct spre sortarea topologică. Observăm că, dacă vrem să învățăm un capitol v după un capitol u, atunci le putem învăța în aceeași trecere dacă $u < v$, altfel este nevoie de o trecere în plus. Problema se poate formaliza ca un graf orientat unde costul unei muchii (u, v) este 0 dacă $u < v$ sau 1 dacă $u > v$. Astfel, trebuie:

- Să tipărim -1 dacă graful nu poate fi sortat topologic (adică dacă conține vreun ciclu).
- Altfel, să tipărim costul maxim al unei căi.

Problema se rezolvă cu sortare topologică și o valoare pentru fiecare nod u, care arată numărul de treceri necesar pentru a învăța capitolul u. Valoarea lui u se calculează în funcție de valorile succesorilor.

Pentru completitudine, am implementat sortarea topologică și cu DFS, și cu BFS.

19.4.3 Problema Componente tare conexe (Infoarena)

enunț • surse

Această problemă este educațională, de componente tare conexe.

Pentru algoritmul lui Kosaraju, subliniez că nu este nevoie de calculul explicit al timpilor t_{out}. Ce ne interesează este ca în a doua parcurgere să procesăm nodurile descrescător după t_{out}. Așadar, este suficient ca, în prima parcurgere, să colectăm fiecare nod într-un vector chiar înainte de a părăsi DFS-ul (adică în locul unde am incrementa acel contor global `time`).

De amorul artei, includ și o sursă pentru algoritmul lui Kosaraju cu structuri de date proprii. Ea transpune graful fără memorie suplimentară, este de peste două ori mai rapidă și consumă de 4 ori mai puțină memorie decît implementarea cu STL.

19.4.4 Problema Mr. Kitayuta's Technology (Codeforces)

enunț • sursă

Problema este mai mult de idee, fiind greu de încadrat la „ceva": este o problemă *core* de grafuri. O putem rezuma astfel: dîndu-se n noduri izolate și m perechi ordonate de noduri, să se afle numărul minim de arce care trebuie trasate astfel încît pentru fiecare pereche (u, v) să existe o cale de la u la v.

Prin analogie cu componentele tare conexe, o **componentă slab conexă** (CSC) este un subgraf maximal astfel încît orice nod v să fie accesibil din orice nod u dacă ignorăm sensurile arcelor.

Fie G graful cerințelor date la intrare. Observăm că putem rezolva problema individual pe fiecare CSC din G, căci nu avem de ce să tragem vreo muchie între două CSC diferite. Acum, să considerăm o componentă cu k noduri. Iau naștere două cazuri.

1. Dacă componenta **nu conține** cicluri orientate, atunci ea este un dag. Fiind un dag, ea admite o sortare topologică S. Putem satisface toate cerințele componentei dacă unim cele k noduri cu $k-1$ muchii, două cîte două în ordinea din S. Putem demonstra că acest rezultat ($k-1$ muchii) este și optim. Să presupunem că ar exista o soluție care satisface toate cerințele componentei cu mai puțin de $k-1$ muchii. Aceste muchii nu ar putea conecta (slab) toate cele k noduri, deci există o tăietură a nodurilor componentei $A \cup B$ astfel încît nicio muchie a soluției să nu traverseze tăietura. Pe de altă parte, există în mod sigur o cerință care traversează tăietura, întrucît componenta este slab conexă. Acea cerință ar rămîne nesatisfăcută.
2. Dacă componenta **conține** un ciclu orientat, C, atunci putem oferi o soluție cu k muchii: conectăm toate nodurile într-un ciclu, formînd o componentă tare conexă. Din nou, putem demonstra că această soluție este optimă. Să presupunem că ar exista o soluție care satisface toate cerințele componentei cu doar $k-1$ muchii. Prin același argument de mai sus, pentru a păstra conexitatea, cele $k-1$ muchii trebuie să formeze un arbore (ignorînd orientarea). Dar atunci ele nu vor putea satisface toate cerințele de pe ciclul C.

19.4.5 Problema Ralph and Mushrooms (Codeforces)

enunț • sursă

Observăm că, în cadrul unei CTC, putem parcurge toate muchiile de o infinitate de ori, deci vom aduna profitul maxim de pe fiecare muchie. O subproblemă interesantă este: ce profit putem obține pe o muchie cu x ciuperci? Această funcție este OEIS A060432. Am încercat să găsesc o formulă în $\mathcal{O}(1)$ fără radicali, dar nu am reușit.

Odată ce părăsim o CTC, putem folosi muchia de plecare o singură dată, căci prin definiție nu mai putem reveni pe ea. Apoi putem consuma integral noua componentă etc. Rezultă că putem construi G^{SCC}, care este un dag, în care valoarea fiecărui nod este profitul maxim al muchiilor din toate nodurile din componenta asociată, iar valoarea fiecărei muchii este numărul de ciuperci din graful inițial. În acest nod trebuie să aflăm profitul maxim.

Aceasta este o problemă clasică de sortare topologică + programare dinamică. Vezi de exemplu:

- CSES 1680 Longest Flight Route (cea mai lungă cale într-un dag)
- CSES 1681 Game Routes (numărul de căi într-un dag)
- CSES 1686 Coin Collector (foarte asemănătoare cu problema curentă).

Soluția teoretică este să parcurgem componentele în ordine invers topologică, folosind orice algoritm de CTC. Fie $w[C]$ numărul total de ciuperci pe care le putem culege fără a părăsi componenta C. Atunci profitul maxim plecînd dintr-o componentă C este

$$profit[C] = w[C] + \max_{(u,v)}\{mushrooms[(u,v)] + profit[D]\}$$

unde D este o altă componentă tare conexă, iar (u,v) este o muchie cu $u \in C$ și $v \in D$.

Putem implementa explicit acest algoritm, construind G^{SCC}. De amorul artei, putem însă rezolva problema cu o singură parcurgere DFS! Sursa anexată oferă o astfel de implementare.

În majoritatea problemelor întîlnite, nu este strict necesar să construiți graful condensat, deși sînt de acord că implementarea cu o singură parcurgere poate fi alunecoasă.

19.4.6 Problema Bertown Roads (Codeforces)

enunț • sursă

Problema cere să orientăm muchiile unui graf neorientat conex astfel încît să obținem un graf tare conex. O problemă identică este CSES 2177. O observație relativ simplă este că, dacă graful neorientat are punți, atunci problema nu are soluție. Dacă orientăm muchia (u,v) ca $u \to v$, în graful rezultat nu va exista nicio cale de la v la u.

Dacă graful neorientat nu are punți, problema are întotdeauna soluție. Interesant este că implementarea este incredibil de simplă. Facem o parcurgere DFS și orientăm muchiile în sensul în care le vizitează parcurgerea. Dintr-un nod u, pentru un vecin v, orientăm muchia ca $u \to v$ indiferent dacă este muchie de arbore sau muchie înapoi.

Trebuie doar să avem grijă să nu parcurgem aceeași muchie de două ori. Mai exact, muchia u, v va apărea în reprezentarea uzuală de două ori:

- nodul v în lista de adiacență a lui u;
- nodul u în lista de adiacență a lui v.

Dacă stăm în nodul u, iar nodul v este deja vizitat, atunci:

- Dacă $d[v] < d[u]$, atunci v este strămoș al lui u, muchia (u, v) este muchie înapoi și o vom orienta în sensul $u \to v$.
- Dacă $d[v] > d[u]$, atunci v este descendent al lui u, muchia (u, v) a fost deja vizitată și trebuie ca de această dată să o ignorăm.

19.4.7 Problema Tourist Reform (Codeforces)

enunț • sursă

Problema este din aceeași sferă cu precedenta. Trebuie să orientăm un graf neorientat pentru a maximiza accesibilitatea minimă, adică să maximizăm numărul minim de noduri accesibile din orice punct de pornire.

Știm de la problemele anterioare că, pentru o CTC, ne putem plimba la infinit printre nodurile ei. Interesant este cum să orientăm punțile din graful original. Fie X componenta 2-conexă (așadar, componenta mărginită de punți) din graful original cu număr maxim de noduri. Presupunem că X este unică, dar raționamentul este similar și dacă nu este. X va deveni o CTC în graful orientat. Rezultă că răspunsul este cel puțin $|X|$: putem orienta toate punțile înspre X, astfel că orice nod din X va putea vizita exact $|X|$ noduri, iar orice nod din afara lui X va putea vizita mai mult de $|X|$ noduri.

Acest răspuns este și optim. Să presupunem că orientăm orice punte astfel încît să se îndepărteze de X. Atunci ea delimitează din graful orientat o zonă în care toate CTC au mai puțin de $|X|$ noduri. Întrucît graful G^{SCC} este un dag, va exista în el o componentă Y din care nu iese nicio muchie. Pornind din acea componentă vom putea ajunge doar la $|Y| < |X|$ noduri.

Și această implementare necesită o singură parcurgere Tarjan. Mai mult, nu avem nevoie de stivă. Ne pasă doar **cîte** noduri are o CTC în clipa în care o identificăm, iar pentru aceasta este suficient să ținem un contor, nu toată stiva.

19.4.8 Problema We Need More Bosses (Codeforces)

enunț • sursă

Problema ne cere să calculăm diametrul maxim (ca număr de muchii) în arborele componentelor 2-conexe asociat unui arbore neorientat.

Așa cum am mai văzut, putem proceda incremental:

1. Calculăm punțile.
2. Condensăm fiecare componentă 2-conexă într-un nod și obținem arborele.
3. Calculăm diametrul arborelui.

Dar putem rezolva și această problemă cu un singur DFS. Aici este util să cunoașteți algoritmul de aflare a diametrului unui arbore cu un singur DFS, nu cu două. Pentru un nod u, diametrul **subarborelui** lui u poate fi de două feluri:

1. Fie trece prin u, caz în care u are nevoie să cunoască suma celor mai lungi două drumuri de la sine la frunze, care pornesc prin doi fii diferiți.
2. Fie nu trece prin u, caz în care u are nevoie să cunoască maximul diametrelor din subarborii fiilor.

Peste această recurență rămîne doar să adăugăm algoritmul lui Tarjan pentru găsirea punților. Dacă fiecare nod u își calculează distanța maximă, în număr de punți, pînă la orice frunză, atunci valoarea pentru u va proveni din maximul valorilor fiilor lui u. Nodul u adaugă 1 la valoarea returnată de un fiu v dacă muchia (u, v) este punte, altfel o preia ca atare.

19.4.9 Problema Forbidden Cities (CSES)

enunț • sursă

Întrebarea teoretică este, pentru fiecare triplet $\langle a, b, c\rangle$: Dacă elimin nodul c, mai rămîn a și b conectate? Intuiția ne împinge spre puncte de articulație. Răspunsul este: nu, dacă c este punct de articulație și dacă a și b cad în componente biconexe diferite raportat la c.

O variantă ar fi, așadar:

- Să construim graful condensat al componentelor biconexe, care este un arbore.
- Să construim LCA pe acest arbore.
- Să verificăm, pentru fiecare triplet $\langle a, b, c\rangle$ în care c este punct de articulație, dacă c se află pe calea a, b.

Mie îmi sună monstruos. 😨 Am preferat să investesc puțin timp într-o soluție... cu o singură parcurgere! Ideea de bază este: dacă un nod c observă că are un fiu d care nu poate urca mai sus decît c, atunci nodul c este punct de articulație, iar subarborele lui d nu poate comunica cu restul grafului decît prin c. Dacă există vreun triplet $\langle a, b, c\rangle$ astfel încît exact unul dintre a și b se află în subarborele lui d, atunci răspunsul pentru $\langle a, b, c\rangle$ este „nu”.

Pentru implementare, am extins algoritmul lui Tarjan pentru găsirea punctelor de articulație cu următorul sistem de notificări. Observați că ocazional schimb numele nodului. Fac aceasta ca să clarific perspectiva din care se întîmplă lucrurile, dar rutina DFS este una singură, relativ scurtă.

- La intrarea în nodul c, activăm o bifă prin care nodul spune „vreau să fiu notificat ori de cîte ori este vizitat vreun nod a sau b din tripletele pentru care eu sînt centrul”.
- La ieșirea din nodul c, dezactivăm această bifă. În particular, dacă a și b se regăsesc în afara subarborelui lui c, atunci c nu blochează calea $a \rightsquigarrow b$.
- Tot la ieșirea dintr-un nod, să-i zicem de această dată a, trecem prin toate tripletele în care a este extremitate și notificăm c-urile acestor triplete, dacă ele sînt în prezent abonate la notificări.
- Revenind în nodul c, după fiecare fiu d, și dacă c constată că este punct critic pentru d, atunci c trece prin lista de triplete care l-au notificat. Dacă pentru orice triplet $\langle a, b, c\rangle$ exact unul dintre a și b este descendent al lui d, atunci marcăm tripletul cu răspunsul „nu”.

- Indiferent dacă c este sau nu punct critic pentru d, c își golește lista de notificări după fiecare fiu. Este important ca, dacă a și b sînt accesibile prin doi fii diferiți, să le tratăm separat.

19.4.10 Problema Case of Computer Network (Codeforces)

enunț • sursă

Dacă am vedea această problemă „de la zero", probabil nu am avea șanse să o rezolvăm. Dar, după cele anterioare, ea devine abordabilă. Din nou, trebuie să orientăm muchiile unui graf, inițial neorientat, asigurîndu-ne că în graful orientat există q căi impuse, de forma (u, v). Problema este rezonabilă în sensul că ne cere doar răspunsul da/nu, nu și orientarea efectivă.

Ca și mai înainte, putem transforma o componentă 2-conexă într-o componentă tare conexă, în care putem călători între orice două noduri. Mai rămîne să orientăm punțile. Să observăm că, dacă există o punte și două mesaje care au nevoie să traverseze acea punte în sensuri diferite, atunci problema nu are soluție.

Un mod de a implementa această observație este să construim arborele componentelor 2-conexe, în care muchiile sînt punți din graful original. Înrădăcinăm acest arbore în nodul 1. Raportat la un nod u și la muchia sa de intrare e, iau naștere patru tipuri de mesaje:

1. Mesaje care încep și se termină în subarborele lui u inclusiv.
2. Mesaje care încep și se termină în afara subarborelui lui u.
3. Mesaje care încep în subarborele lui u și se termină în afara lui.
4. Mesaje care încep în afara subarborelui lui u și se termină în el.

Mesajele de tipurile 1 sau 2 nu limitează în niciun fel orientarea lui e. Dacă avem doar mesaje de tipul 3, orientăm e dinspre u înspre rădăcină. Dacă avem doar mesaje de tipul 4, orientăm e dinspre rădăcină înspre u. Iar dacă avem mesaje de ambele tipuri (3 și 4), atunci problema nu are soluție.

Rezultă că putem ține evidența mesajelor de tip 3 și 4 în postordine. În plus, avem nevoie de informații LCA ca să putem șterge mesajele din evidență cînd ajungem la strămoșul lor comun. Din nou, putem implementa asta cu cărămizile cunoscute: graful componentelor, LCA cu $\log n$ pointeri per nod 😉 etc.

Dar și aici ne putem descurca cu o singură parcurgere. Dacă tot facem un DFS, ar fi păcat să nu îl folosim și pentru a afla LCA-urile mesajelor. Deci combinăm algoritmul lui Tarjan pentru aflarea punților cu algoritmul lui Tarjan pentru LCA. 🤣

Un aspect care poate părea straniu este că noi operăm pe un graf, pe cînd algoritmul de LCA funcționează pe arbori. Dar ne putem adapta, cu condiția să lăsăm toate deciziile în seama nodurilor-șef ale componentelor 2-conexe (cele aflate imediat sub o punte). Celelalte noduri dintr-o componentă nu fac decît să propage contoarele către șef. Tot fiindcă operăm pe un graf, este posibil să ajungem într-un descendent de mai multe ori. Este important ca, la propagarea

contoarelor spre părinte, să le ștergem din fiu, ca să nu avem surpriza că le-am propagat de mai multe ori.

19.4.11 Problema Dog Trick Competition 2 (IIOT 2023-2024 runda 4)

enunț • sursă

În esență, problema ne dă un graf orientat și un șir T de noduri și ne cere să vizităm, în ordine, un prefix cît mai lung al lui T, în ordinea $T_1, T_2, \ldots$, eventual folosind și alte noduri intermediare.

Faptul că sîntem la lecția de tare conexitate ar trebui să fie un indiciu. 🙂 Să observăm că putem rămîne oricît este nevoie în CTC-ul nodului T_1, vizitînd cît de mult se poate din T. Fie T_a primul nod nevizitat astfel. Dacă există un drum de la componenta lui T_1 la componenta lui T_a, putem continua traseul. Vizităm în continuare nodurile din T aflate în componenta lui T_a. Fie T_b primul nod nevizitat. Dacă există un drum de la componenta lui T_a la T_b, putem continua traseul etc. Remarcăm că, odată ce părăsim o CTC, nu mai putem reveni în ea. Dacă am putea, am închide un ciclu, ceea ce încalcă definiția CTC.

Implementare cu construcția grafului condensat

De aici, implementarea se ramifică. Am citit sursele din statistici și am observat, de regulă, această abordare:

- Algoritmul lui Kosaraju pentru determinarea CTC.
- Construcția explicită a grafului condensat.
- Pentru fiecare pereche (T_i, T_{i+1}) din componente diferite, BFS din T_i pe graful condensat pentru a determina dacă există o cale către T_{i+1}.
- Ordonarea crescătoare a vecinilor fiecărui nod, ceea ce permite răspunsuri cu căutare binară la întrebări de tipul „există muchia (T_i, T_{i+1})”?

Soluția este $\mathcal{O}(m + n)$ în ciuda BFS-urilor repetate, căci mergem mereu înainte prin dag-ul componentelor tare conexe. De aceea, BFS-urile au mereu loc pe porțiuni din dag încă neexplorate în BFS-urile anterioare.

Ca fapt divers, această sursă sortează topologic componentele, apoi decide dacă există un drum între două componente comparînd pur și simplu poziția componentelor în sortare. Această abordare este insuficientă. Problema răspunde 3, în loc de 2, pe testul

```
2 3
2 1
2
1 3
2 3
```

Implementare cu un singur DFS

Implementarea mea extinde algoritmul lui Tarjan pentru aflarea CTC. Știm că acesta emite componentele în ordinea invers topologică. Noi am prefera exact opusul, ca să pornim din componenta lui T_1 și să mergem înainte prin componente. Dar ne putem descurca și cu ordinea inversă.

Să definim **intervalul unei componente** C ca fiind intervalul de indici din T pe care îl vom vizita pe durata șederii în componenta C (dacă vom ajunge în ea, ceea ce deocamdată nu știm). Acesta este cel mai din stînga interval contiguu format exclusiv din noduri din C. Pot exista mai multe astfel de intervale în T, dar îl alegem pe cel mai din stînga deoarece, odată ce părăsim componenta C, nu mai putem reveni la ea pentru a vizita și alte intervale.

Pentru a determina intervalul componentei, încă de la citirea datelor calculăm, pentru fiecare nod u, indicele primei sale apariții în T (sau $+\infty$ dacă u nu apare deloc în T). Odată ce explorăm componenta C și îi cunoaștem nodurile, începutul intervalului lui C este minimul primelor apariții în T ale oricărui nod din C. Aflăm sfîrșitul intervalului extinzîndu-l naiv spre dreapta, pas cu pas, cît timp întîlnim noduri din C. Efortul total pentru aceste extinderi este $\mathcal{O}(n)$.

Acum, algoritmul lui Tarjan, în momentul în care termină de explorat componenta C, va fi întîlnit și muchii către alte componente (succesoare în dag-ul componentelor). Deci putem pune întrebarea: dacă ajungem în această componentă, cu ce altă componentă continuăm? *Obligatoriu* cu una care are acces la componenta care extinde spre dreapta intervalul lui C. De aceea, dintre toate componentele succesoare, păstrăm intervalul minim. Concret, fie componenta curentă C cu intervalul I_C și fie I_D intervalul minim al unei componente succesoare D. Iau naștere trei cazuri:

- Dacă I_D îl precede pe I_C, reținem doar I_D. Realitatea este că vom sări complet componenta C și doar o vom traversa ca să ajungem la D, unde putem vizita noduri din T.
- Dacă I_D urmează imediat după I_C, vom reține $I_C \cup I_D$. Vom vizita nodurile din C, apoi călătorim în D și vizităm și acele noduri.
- Dacă I_D urmează după I_C, dar la o distanță nenulă, reținem doar I_C. Realitatea este că ne vom opri după I_C, căci nu putem ajunge la următorul nod din T.

Cu această recurență, este suficient să facem un singur DFS din T_1. La final, ultima componentă explorată îl va conține pe T_1, iar intervalul său va avea forma $[1, X]$, semnificînd că putem vizita primele X noduri din T.

Pentru a răspunde la întrebări de tipul „există muchia (u, v)?”, eu am folosit pur și simplu un `unordered_set` de `long long` și am mapat fiecare pereche (u, v) la valoarea $u \cdot n + v$.

Partea VII

Probleme diverse

Capitolul 20

Probleme diverse

Acest capitol include probleme care fie necesită doar cunoștințe de bază (engl. *core*, numite în supa culturală și probleme ad-hoc), fie necesită cunoștințe specifice, dar pentru care cursul încă nu are un capitol dedicat.

20.1 Problema Liars (Baraj ONI 2025)

enunț • sursă

20.1.1 Generalități

Problema este una tipică de recurență pe arbore[1], dar acest curs nu are un capitol dedicat subiectului. Poate într-o zi cu soare. Problema necesită multă atenție la implementare.

Să începem cu numărarea configurațiilor posibile. În problemele de recurență pe arbore, de obicei judecăm recursiv. Informația pe care o dorim să o calculăm într-un nod u decurge din combinarea informațiilor similare calculate în fiecare dintre fiii lui u. În plus, **încercăm să decuplăm subarborele lui u de restul arborelui**. Pentru aceasta, luăm în calcul toate posibilitățile pentru u sau (în alte probleme) pentru părintele lui u. Nu știm care dintre posibilități ne va trebui în final, dar astfel ne asigurăm că problema nu „se revarsă" în restul arborelui.

20.1.2 Numărarea configurațiilor

În cazul de față, vom considera pe rînd că u este sincer, apoi mincinos. Să considerăm cazul în care u este sincer și fie r numărul de vecini mincinoși ai lui u (citit de la intrare). **În principiu** dorim să calculăm recursiv, pentru fiecare fiu v al lui u, numărul de configurații valide pentru

[1]Simt nevoia unei digresiuni despre termenul *programare dinamică*. Olimpicii tind să denumească orice recurență programare dinamică (și, pentru maximum de efect, să-și denumească variabilele aferente `dp`). Dar această denumire este incorectă în cazul arborilor. Pentru a putea încadra o soluție ca programare dinamică, ea are nevoie de două trăsături: substructura optimă și suprapunerea subproblemelor. În cazul recurențelor pe arbore, prin definiție **nu** există subprobleme suprapuse, deoarece în acest caz subproblemele unui nod sînt fiii nodului, deci prin definiție sînt disjuncte.

v și subarborele său, apoi să contorizăm numărul de moduri în care putem alege r fii mincinoși pentru u. Doar că ne mai lipsește o informație: valoarea de adevăr a părintelui lui u, fie el p. Dacă p este mincinos, atunci lui u îi mai trebuie doar $r-1$ fii mincinoși.

Astfel ajungem la mulțimea finală de informații necesare în fiecare nod u, respectiv numărul de configurații valide pentru subarborele lui u în patru cazuri:

1. u este sincer, iar p este sincer;
2. u este sincer, iar p este mincinos;
3. u este mincinos, iar p este sincer;
4. u este mincinos, iar p este mincinos.

Să considerăm acum că avem aceste 4 cantități calculate în toți fiii lui u. Dacă u este sincer, atunci, după cum am spus, ne interesează în cîte moduri putem asigna r sau $r-1$ fii mincinoși (în funcție de valoarea lui p. Dacă u este mincinos, atunci dimpotrivă ne interesează să asignăm orice număr de fii mincinoși în afară de r, respectiv $r-1$. Această din urmă cantitate este mai simplu de calculat dacă aflăm numărul total de configurații în subarborele lui u (indiferent de numărul de fii mincinoși) din care scădem cantitatea care l-ar face pe u să fie sincer.

Cum combinăm informațiile din fii? Limitele problemei ne permit complexitatea $\mathcal{O}(d_u^2)$ în fiecare nod, unde cu d_u am notat numărul de fii ai lui u. Aceasta ne permite o logică rezonabil de simplă (la acest nivel). Pentru nodul curent u, și pentru o valoare fixată pentru u, să calculăm numărul de configurații în care printre fiii lui u există $0, 1, 2, d_u$ mincinoși.

Aceasta este o subproblemă de programare dinamică (reală, în acest caz). Fie $C(i, j)$ numărul de moduri de a obține j mincinoși folosind primii i fii. Inițial, $C(0, 0) = 1$, căci cu primii 0 fii putem obține (doar) 0 mincinoși într-un singur mod. Apoi, $C(i, j)$ provine

- din $C(i-1, j)$ dacă asignăm al i-lea fiu ca fiind sincer;
- din $C(i-1, j-1)$ dacă asignăm al j-lea fiu ca fiind mincinos.

20.1.3 Găsirea unei configurații

Odată numărate configurațiile în această manieră *bottom up*, putem găsi una dintre ele într-o manieră *top down*. Ce valoare să punem în rădăcină (nodul 1)? Prin convenție, să asignăm rădăcina ca sinceră dacă ea raportează cel puțin o configurație validă în care ea este sinceră, altfel o asignăm ca mincinoasă. Din nou, efectul acestei asignări este că decuplăm subarborii rădăcinii.

O dată fixată valoarea unui nod u, îi putem asigna și fiii. Acest proces este elementar, dar necesită un pic de cod, în două faze.

În primă fază, asignăm toți fiii lui u care sînt impuși. Dacă un fiu v nu are nicio asignare în care el să fie sincer, iar părintele său u să aibă valoarea fixată, atunci obligatoriu v trebuie să fie mincinos. Similar decidem dacă v trebuie să fie sincer. Dintre restul fiilor, care pot lua orice valoare, știm cîți trebuie să fie sinceri și cîți mincinoși, în funcție de numărul de vecini mincinoși declarați de u și de valoarea acestuia de adevăr. Asignăm și restul de fii ai lui u respectînd aceste

cantități. Procedăm astfel pînă în frunze.

Partea VIII

Anexă: Cod-sursă

Anexa A

Arbori de intervale

A.1 Problema Xenia and Bit Operations (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 1 << 17;

struct segment_tree {
  int v[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = n;
  }

  void set(int pos, int val) {
    v[pos + n] = val;
  }

  void build() {
    bool is_or = true;
    for (int i = n - 1; i; i--) {
      int l = 2 * i, r = 2 * i + 1;
      v[i] = is_or ? (v[l] | v[r]) : (v[l] ^ v[r]);
      if ((i & (i - 1)) == 0) {
        is_or = !is_or;
      }
    }
  }

  int update(int pos, int val) {
    pos += n;
    v[pos] = val;
```

```
    bool is_or = true;
    for (pos /= 2; pos; pos /= 2) {
      int l = 2 * pos, r = 2 * pos + 1;
      v[pos] = is_or ? (v[l] | v[r]) : (v[l] ^ v[r]);
      is_or = !is_or;
    }
    return v[1];
  }
};

segment_tree st;
int num_queries;

void read_array() {
  int n;
  scanf("%d %d", &n, &num_queries);
  n = 1 << n;
  st.init(n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    st.set(i, x);
  }

  st.build();
}

void process_queries() {
  while (num_queries--) {
    int pos, val;
    scanf("%d %d", &pos, &val);
    int root = st.update(pos - 1, val);
    printf("%d\n", root);
  }
}

int main() {
  read_array();
  process_queries();

  return 0;
}
```

A.2 Problema Distinct Characters Queries (Codeforces)

‹ înapoi • versiune online

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

const int MAX_N = 1 << 17;
const int T_UPDATE = 1;

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

struct segment_tree {
  int v[2 * MAX_N];
  int n;

  void init(char* s) {
    int len = strlen(s);
    this->n = next_power_of_2(len);

    for (int i = 0; i < len; i++) {
      v[i + this->n] = 1 << (s[i] - 'a');
    }

    build();
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i] | v[2 * i + 1];
    }
  }

  void update(int pos, char val) {
    pos += n;
    v[pos] = 1 << (val - 'a');

    for (pos /= 2; pos; pos /= 2) {
      v[pos] = v[2 * pos] | v[2 * pos + 1];
    }
  }

  int popcount_query(int l, int r) {
    int mask = 0;

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        mask |= v[l++];
```

```
      }
      l >>= 1;

      if (!(r & 1)) {
        mask |= v[r--];
      }
      r >>= 1;
    }

    return __builtin_popcount(mask);
  }
};

segment_tree st;

void read_string() {
  char s[MAX_N + 1];
  scanf("%s", s);
  st.init(s);
}

void process_ops() {
  int num_ops, type;
  scanf("%d", &num_ops);

  while (num_ops--) {
    scanf("%d", &type);

    if (type == T_UPDATE) {
      int pos;
      char val;
      scanf("%d %c", &pos, &val);
      st.update(pos - 1, val);
    } else {
      int l, r;
      scanf("%d %d", &l, &r);
      int num_distinct = st.popcount_query(l - 1, r - 1);
      printf("%d\n", num_distinct);
    }
  }
}

int main() {
  read_string();
  process_ops();

  return 0;
}
```

A.3 Problema K-query (SPOJ)

înapoi

Sursă cu arbori de intervale.

```
// Complexitate: O(n log n)
#include <algorithm>
#include <stdio.h>

const int MAX_N = 32'768;
const int MAX_Q = 200'000;

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
}

struct segment_tree {
  short v[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos) {
    pos += n;
    while (pos) {
      v[pos]++;
      pos /= 2;
    }
  }

  short range_count(int l, int r) {
    int sum = 0;

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        sum += v[l++];
      }
      l >>= 1;

      if (!(r & 1)) {
```

```
        sum += v[r--];
      }
      r >>= 1;
    }

    return sum;
  }
};

struct elem {
  int val;
  short pos;
};

struct query {
  short left, right;
  int val;
  int orig_pos;
};

elem v[MAX_N];
query q[MAX_Q];
// Mai curat ar fi să punem răspunsurile în struct. Dar atunci ar trebui să
// sortăm interogările încă o dată la final.
short answer[MAX_Q];
segment_tree st;
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &v[i].val);
    v[i].pos = i + 1;
  }

  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%hd %hd %d", &q[i].left, &q[i].right, &q[i].val);
    q[i].orig_pos = i;
  }
}

void sort_array_by_val() {
  std::sort(v, v + n, [](elem a, elem b) {
    return a.val > b.val;
  });
}

void sort_queries_by_val() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
```

```
    return a.val > b.val;
  });
}

void process_queries() {
  st.init(n);

  int j = 0;
  for (int i = 0; i < num_queries; i++) {
    while ((j < n) && (v[j].val > q[i].val)) {
      st.set(v[j++].pos);
    }
    answer[q[i].orig_pos] = st.range_count(q[i].left, q[i].right);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_data();
  sort_array_by_val();
  sort_queries_by_val();
  process_queries();
  write_answers();

  return 0;
}
```

Sursă cu arbori indexați binar.

```
// Complexitate: O(n log n)
#include <algorithm>
#include <stdio.h>

const int MAX_N = 30'000;
const int MAX_Q = 200'000;

struct fenwick_tree {
  short v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
  }
```

```
  void set(int pos) {
    do {
      v[pos]++;
      pos += pos & -pos;
    } while (pos <= n);
  }

  short prefix_count(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }

  short range_count(int l, int r) {
    return prefix_count(r) - prefix_count(l - 1);
  }
};

struct elem {
  int val;
  short pos;
};

struct query {
  short left, right;
  int val;
  int orig_pos;
};

elem v[MAX_N];
query q[MAX_Q];
// Mai curat ar fi să punem răspunsurile în struct. Dar atunci ar trebui să
// sortăm interogările încă o dată la final.
short answer[MAX_Q];
fenwick_tree fen;
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &v[i].val);
    v[i].pos = i + 1;
  }

  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%hd %hd %d", &q[i].left, &q[i].right, &q[i].val);
```

```
    q[i].orig_pos = i;
  }
}

void sort_array_by_val() {
  std::sort(v, v + n, [](elem a, elem b) {
    return a.val > b.val;
  });
}

void sort_queries_by_val() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.val > b.val;
  });
}

void process_queries() {
  fen.init(n);

  int j = 0;
  for (int i = 0; i < num_queries; i++) {
    while ((j < n) && (v[j].val > q[i].val)) {
      fen.set(v[j++].pos);
    }
    answer[q[i].orig_pos] = fen.range_count(q[i].left, q[i].right);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_data();
  sort_array_by_val();
  sort_queries_by_val();
  process_queries();
  write_answers();

  return 0;
}
```

A.4 Problema Sereja and Brackets (Codeforces)

‹ înapoi • versiune online

```cpp
// Complexitate: O(n + q log n).
#include <stdio.h>
#include <string.h>

const int MAX_N = 1 << 20;

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

struct node {
  int matched, open, closed;

  node combine(node& other) {
    int new_matches = min(open, other.closed);
    return {
      .matched = matched + other.matched + 2 * new_matches,
      .open = open + other.open - new_matches,
      .closed = closed + other.closed - new_matches,
    };
  }
};

struct segment_tree {
  node v[2 * MAX_N];
  int n;

  void init(char* s) {
    int len = strlen(s);
    this->n = next_power_of_2(len);

    for (int i = 0; i < len; i++) {
      v[i + this->n] = {
        .matched = 0,
        .open = (s[i] == '('),
        .closed = (s[i] == ')'),
      };
    }

    build();
  }
```

```
  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i].combine(v[2 * i + 1]);
    }
  }

  int query(int l, int r) {
    node left = { 0, 0, 0 };
    node right = { 0, 0, 0 };

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        left = left.combine(v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        right = v[r--].combine(right);
      }
      r >>= 1;
    }

    node answer = left.combine(right);
    return answer.matched;
  }
};

segment_tree st;

void read_string() {
  char s[MAX_N + 1];
  scanf("%s", s);
  st.init(s);
}

void process_ops() {
  int num_ops;
  scanf("%d", &num_ops);

  while (num_ops--) {
    int l, r;
    scanf("%d %d", &l, &r);
    printf("%d\n", st.query(l - 1, r - 1));
  }
}
```

```
int main() {
  read_string();
  process_ops();

  return 0;
}
```

A.5 Problema Copying Data (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(q log n)
#include <stdio.h>

const int MAX_N = 1 << 17;
const int T_COPY = 1;

struct change {
  int time;
  int shift;
};

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

struct segment_tree {
  change v[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  change query(int pos) {
    change latest = { 0, 0 };
    for (pos += n; pos; pos >>= 1) {
      if (v[pos].time > latest.time) {
        latest = v[pos];
      }
    }
    return latest;
  }

  void update(int l, int r, change c) {
    l += n;
    r += n;
```

```
    while (l <= r)  {
      if (l & 1) {
        v[l++] = c;
      }
      l >>= 1;

      if (!(r & 1)) {
        v[r--] = c;
      }
      r >>= 1;
    }
  }
};

segment_tree st;
int a[MAX_N], b[MAX_N];
int n, num_ops;

void read_arrays() {
  scanf("%d %d", &n, &num_ops);
  for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
  for (int i = 0; i < n; i++) {
    scanf("%d", &b[i]);
  }
}

void process_ops() {
  st.init(n);
  int type, x, y, k;
  for (int op = 1; op <= num_ops; op++) {
    scanf("%d", &type);
    if (type == T_COPY) {
      scanf("%d %d %d", &x, &y, &k);
      x--;
      y--;
      change c = { .time = op, .shift = x - y };
      st.update(y, y + k - 1, c);
    } else {
      scanf("%d", &x);
      x--;
      change latest = st.query(x);
      int val = latest.time ? a[x + latest.shift] : b[x];
      printf("%d\n", val);
    }
  }
}

int main() {
```

```
  read_arrays();
  process_ops();

  return 0;
}
```

A.6 Problema PHF (FMI No Stress 2013)

‹ înapoi • versiune online

```
#include <stdio.h>

typedef unsigned char byte;

const int MAX_N = 1'000'000;
const int MAX_SEGTREE_NODES = 1 << 21;

const int IDENTITY = 3;
const byte CROSS_TABLE[4][3] = {
  { 0, 1, 0 }, // P
  { 1, 1, 2 }, // H
  { 0, 2, 2 }, // F
  { 0, 1, 2 }, // identitate
};

char FROM_CHAR[27] = ".....2.1.......0..........";
char TO_CHAR[4] = "PHF";

byte from_char(char c) {
  return FROM_CHAR[c - 'A'] - '0';
}

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

struct segment_tree_node {
  byte f[3];

  void make_leaf(byte c) {
    for (int i = 0; i < 3; i++) {
      f[i] = CROSS_TABLE[c][i];
    }
  }

  segment_tree_node compose(segment_tree_node other) {
    return {
      other.f[f[0]],
```

```
      other.f[f[1]],
      other.f[f[2]],
    };
  }
};

struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n;

  void init(char* s, int _n) {
    n = next_power_of_2(_n);
    for (int i = 0; i < _n; i++) {
      v[n + i].make_leaf(s[i]);
    }

    for (int i = _n; i < n; i++) {
      v[n + i].make_leaf(IDENTITY);
    }

    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i].compose(v[2 * i + 1]);
    }
  }

  void update(int pos, byte b) {
    pos += n;
    v[pos].make_leaf(b);
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = v[2 * pos].compose(v[2 * pos + 1]);
    }
  }

  char get_value(byte input) {
    return TO_CHAR[v[1].f[input]];
  }
};

char s[MAX_N + 1];
segment_tree st;
int n, num_queries;

void read_string() {
  scanf("%d %d ", &n, &num_queries);
  for (int i = 0; i < n; i++) {
    s[i] = from_char(getchar());
  }
}

void process_updates() {
```

```
  int pos;

  while (num_queries--) {
    scanf("%d ", &pos);
    pos--;
    s[pos] = from_char(getchar());
    if (pos) {
      st.update(pos - 1, s[pos]);
    }
    putchar(st.get_value(s[0]));
  }
}

int main() {
  read_string();
  st.init(s + 1, n - 1);
  putchar(st.get_value(s[0]));
  process_updates();
  putchar('\n');

  return 0;
}
```

A.7 Problema Points (Codeforces)

înapoi • versiune online

```
#include <algorithm>
#include <set>
#include <stdio.h>

const int MAX_N = 1 << 18;
const int INFINITY = 2'000'000'000;
const int T_ADD = 1;
const int T_REMOVE = 2;
const int T_FIND = 3;

struct query {
  char type;
  int x, y;
};

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
```

```
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct max_segment_tree {
  int v[2 * MAX_N];
  int n;

  void init(int n) {
    n++; // santinelă infinită la final
    n = next_power_of_2(n);
    this->n = n;

    for (int i = 1; i < 2 * n; i++) {
      v[i] = -1; // fără puncte
    }

    set(n - 1, INFINITY);
  }

  void set(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
    }
  }

  // Returnează prima poziție după pos unde valoarea depășește val.
  int find_first_after(int pos, int val) {
    pos += n;
    pos++;

    // Urcă pînă cînd găsim un maxim mai mare decît val.
    while (v[pos] <= val) {
      if (pos & 1) {
        pos++;
      } else {
        pos >>= 1;
      }
    }

    // Coboară spre valoarea dorită, mergînd la stînga oricînd putem.
    while (pos < n) {
      pos = (v[2 * pos] > val) ? (2 * pos) : (2 * pos + 1);
    }

    return pos - n;
```

```
  }

};

query q[MAX_N];
int pos[MAX_N]; // pentru normalizare
int orig_x[MAX_N];
max_segment_tree segtree;
std::set<int> whys[MAX_N];
int n;

void read_queries() {
  char s[10];
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%s %d %d", s, &q[i].x, &q[i].y);
    switch (s[0]) {
      case 'a': q[i].type = T_ADD; break;
      case 'r': q[i].type = T_REMOVE; break;
      default: q[i].type = T_FIND;
    }
  }
}

void normalize_x() {
  for (int i = 0; i < n; i++) {
    pos[i] = i;
  }

  std::sort(pos, pos + n, [](int a, int b) {
    return q[a].x < q[b].x;
  });

  int ptr = 0;
  orig_x[ptr] = q[pos[0]].x;
  for (int i = 0; i < n; i++) {
    if (q[pos[i]].x != orig_x[ptr]) {
      orig_x[++ptr] = q[pos[i]].x;
    }
    q[pos[i]].x = ptr;
  }
}

void add_query(int x, int y) {
  whys[x].insert(y);
  segtree.set(x, *whys[x].rbegin());
}

void remove_query(int x, int y) {
  whys[x].erase(y);
```

```
  if (whys[x].empty()) {
    segtree.set(x, -1);
  } else {
    segtree.set(x, *whys[x].rbegin());
  }
}

void find_query(int x, int y) {
  int first = segtree.find_first_after(x, y);
  if (first < n) {
    int first_above = *whys[first].upper_bound(y);
    printf("%d %d\n", orig_x[first], first_above);
  } else {
    printf("-1\n");
  }
}

void process_queries() {
  segtree.init(n);
  for (int i = 0; i < n; i++) {
    switch (q[i].type) {
      case T_ADD: add_query(q[i].x, q[i].y); break;
      case T_REMOVE: remove_query(q[i].x, q[i].y); break;
      default: find_query(q[i].x, q[i].y);
    }
  }
}

int main() {
  read_queries();
  normalize_x();
  process_queries();

  return 0;
}
```

A.8 Problema Medwalk (Lot 2025)

‹ înapoi • versiune online

```
// Complexitate: O((N + Q) log N log MAX_VAL).
//
// v2: Nu actualiza aint-ul de minime dacă valoarea nu s-a schimbat.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <stdio.h>

const int MAX_N = 100'000;
```

```
const int MAX_VALUE = 300'000;
const int MAX_POS_SEGTREE_NODES = 1 << 18;
const int MAX_VAL_SEGTREE_NODES = 1 << 20;
const int INF = 1'000'000;
const int OP_UPDATE = 1;

typedef __gnu_pbds::tree<
  int,
  __gnu_pbds::null_type,
  std::less<int>,
  __gnu_pbds::rb_tree_tag,
  __gnu_pbds::tree_order_statistics_node_update
  > set;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct pair {
  int v[2];

  void set(int pos, int val) {
    v[pos] = val;
  }

  int get_min() {
    return min(v[0], v[1]);
  }

  int get_max() {
    return max(v[0], v[1]);
  }
};

pair mat[MAX_N];
int n, num_queries;

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

// Admite actualizări punctuale și interogări de minim pe interval.
struct min_segment_tree {
  int v[MAX_POS_SEGTREE_NODES];
  int n;
```

```
  void init(int size) {
    n = next_power_of_2(size);
  }

  void update(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = min(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int range_min(int l, int r) {
    l += n;
    r += n;
    int result = INF;

    while (l <= r)  {
      if (l & 1) {
        result = min(result, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = min(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

// Kudos https://stackoverflow.com/a/22075025/6022817
//
// Arbore de intervale pe valori. Fiecare nod care subîntinde valorile [x, y)
// reține un set cu pozițiile pe care apar acele valori.
struct val_segment_tree {
  set s[MAX_VAL_SEGTREE_NODES];
  int n;

  void init(int size) {
    n = next_power_of_2(size);
  }

  void insert(int pos, int val) {
    for (val += n; val; val /= 2) {
      s[val].insert(pos);
    }
  }
```

```
  void erase(int pos, int val) {
    for (val += n; val; val /= 2) {
      s[val].erase(pos);
    }
  }

  // Cîte poziții din [l, r] sînt ocupate de valori subîntinse de acest nod?
  int count_positions_between(int node, int l, int r) {
    return s[node].order_of_key(r + 1) - s[node].order_of_key(l);
  }

  // Contract: k este 0-based, iar [l, r] este interval închis.
  int kth_element(int k, int l, int r) {
    int node = 1;

    while (node < n) {
      int on_left = count_positions_between(2 * node, l, r);
      if (on_left > k) {
        node = 2 * node;
      } else {
        k -= on_left;
        node = 2 * node + 1;
      }
    }

    return node - n;
  }
};

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int r = 0; r < 2; r++) {
    for (int i = 0; i < n; i++) {
      int x;
      scanf("%d", &x);
      mat[i].set(r, x);
    }
  }
}

min_segment_tree maxima;
val_segment_tree occur;

void build_segment_trees() {
  maxima.init(n);
  for (int i = 0; i < n; i++) {
    maxima.update(i, mat[i].get_max());
  }
```

```
  occur.init(MAX_VALUE + 1);
  for (int i = 0; i < n; i++) {
    occur.insert(i, mat[i].get_min());
  }
}

void update(int row, int col, int val) {
  int old_min = mat[col].get_min();
  mat[col].set(row, val);
  int new_min = mat[col].get_min();

  maxima.update(col, mat[col].get_max());

  if (new_min != old_min) {
    occur.erase(col, old_min);
    occur.insert(col, new_min);
  }
}

int query(int left, int right) {
  int len = right - left + 2;
  int median_pos = (len - 1) / 2;
  int median = occur.kth_element(median_pos, left, right);
  int best_max = maxima.range_min(left, right);

  if (best_max >= median) {
    return median;
  } else {
    // best_max trebuie inserat înainte de median_pos. Răspunsul poate fi la
    // poziția median_pos - 1 sau poate fi chiar best_max.
    int prev = occur.kth_element(median_pos - 1, left, right);
    return max(prev, best_max);
  }
}

void process_queries() {
  while (num_queries--) {
    int type;
    scanf("%d", &type);
    if (type == OP_UPDATE) {
      int r, c, x;
      scanf("%d %d %d", &r, &c, &x);
      r--;
      c--;
      update(r, c, x);
    } else {
      int left, right;
      scanf("%d %d", &left, &right);
      left--;
      right--;
```

```
      printf("%d\n", query(left, right));
    }
  }
}

int main() {
  read_data();
  build_segment_trees();
  process_queries();

  return 0;
}
```

Anexa B

Arbori de intervale cu propagare *lazy*

B.1 Problema Polynomial Queries (CSES)

înapoi

Sursă cu AINT iterativ.

```
// Complexitate: O(q log n)
//
// Metodă: menține un arbore de intervale care admite sume pe interval și
// adăugarea unei progresii pe interval.
#include <stdio.h>

const int MAX_SEGTREE_NODES = 1 << 19;
const int T_UPDATE = 1;

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

long long progression_sum(long long first, long long step, int len) {
  return first * len + step * (len - 1) * len / 2;
}

// Invarianți:
//
// 1. Valorile reale ale nodurilor subîntines sînt valorile lor v respective
//    plus o progresie aritmetică definită prin first și step.
// 2. Valoarea v a unui nod nu include progresia nodului.
struct segment_tree_node {
  long long s;
  long long first, step;

  long long get_value(int size) {
    return s + progression_sum(first, step, size);
```

```
  }
};


struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos, int val) {
    v[pos + n].s = val;
  }


  void build() {
    for (int i = n - 1; i; i--) {
      v[i].s = v[2 * i].s + v[2 * i + 1].s;
    }
  }

  void push(int t, int size) {
    int l = 2 * t, r = 2 * t + 1;

    v[l].first += v[t].first;
    v[l].step += v[t].step;

    long long first_right = v[t].first + v[t].step * size / 2;
    v[r].first += first_right;
    v[r].step += v[t].step;

    v[t].s += progression_sum(v[t].first, v[t].step, size);
    v[t].first = 0;
    v[t].step = 0;
  }

  void push_path(int node, int size) {
    if (node) {
      push_path(node / 2, size * 2);
      push(node, size);
    }
  }

  void pull(int t, int size) {
    int l = 2 * t, r = 2 * t + 1;
    v[t].s = v[l].get_value(size / 2) + v[r].get_value(size / 2);
  }
```

```
void pull_path(int node) {
  for (int x = node / 2, size = 2; x; x /= 2, size <<= 1) {
    pull(x, size);
  }
}

void add_progression(int l, int r) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r, size = 1;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    // Prima frunză subîntinsă de nodul x este x * size.
    if (l & 1) {
      v[l].first += l * size - orig_l + 1;
      v[l].step++;
      l++;
    }
    l >>= 1;

    if (!(r & 1)) {
      v[r].first += r * size - orig_l + 1;
      v[r].step++;
      r--;
    }
    r >>= 1;
    size <<= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}

long long range_sum(int l, int r) {
  long long sum = 0;
  int size = 1;

  l += n;
  r += n;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      sum += v[l++].get_value(size);
    }
    l >>= 1;
```

```
      if (!(r & 1)) {
        sum += v[r--].get_value(size);
      }
      r >>= 1;
      size <<= 1;
    }

    return sum;
  }

};

segment_tree st;
int n, num_ops;

void read_array() {
  scanf("%d %d", &n, &num_ops);
  st.init(n);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    st.set(i, x);
  }
  st.build();
}

void process_ops() {
  while (num_ops--) {
    int type, l, r;
    scanf("%d %d %d", &type, &l, &r);
    l--; r--;
    if (type == T_UPDATE) {
      st.add_progression(l, r);
    } else {
      printf("%lld\n", st.range_sum(l, r));
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

Sursă cu AINT recursiv.

```
// Complexitate: O(q log n)
```

```
//
// Metodă: menține un arbore de intervale care admite sume pe interval și
// adăugarea unei progresii pe interval.
#include <stdio.h>

const int MAX_N = 1 << 18;
const int T_UPDATE = 1;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

long long progression_sum(long long first, long long step, int len) {
  return first * len + step * (len - 1) * len / 2;
}

// Invarianți:
//
// 1. Valorile reale ale nodurilor subîntines sînt valorile lor v respective
//    plus o progresie aritmetică definită prin first și step.
// 2. v[k] include first[k] și step[k], dar nu și valorile de la strămoși.
// 3. Toate intervalele sînt [închis, deschis).
struct segment_tree {
  long long v[2 * MAX_N];
  long long first[2 * MAX_N];
  long long step[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos, int val) {
    v[pos + n] = val;
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i] + v[2 * i + 1];
    }
  }
```

```
void push(int t, int len) {
  first[2 * t] += first[t];
  step[2 * t] += step[t];
  v[2 * t] += progression_sum(first[t], step[t], len / 2);

  int right_first = first[t] + step[t] * len / 2;
  first[2 * t + 1] += right_first;
  step[2 * t + 1] += step[t];
  v[2 * t + 1] += progression_sum(right_first, step[t], len / 2);

  first[t] = 0;
  step[t] = 0;
}

void pull(int t) {
  v[t] = v[2 * t] + v[2 * t + 1];
}

// pos1 = poziția unde scriem 1
void add_progression_helper(int t, int pl, int pr, int l, int r, int pos1) {
  if (l >= r) {
    return;
  } else if ((l == pl) && (r == pr)) {
    int value_at_l = l - pos1 + 1;
    first[t] += value_at_l;
    step[t]++;
    v[t] += progression_sum(value_at_l, 1, r - l);
  } else {
    push(t, pr - pl);
    int mid = (pl + pr) >> 1;
    add_progression_helper(2 * t, pl, mid, l, min(r, mid), pos1);
    add_progression_helper(2 * t + 1, mid, pr, ::max(l, mid), r, pos1);
    pull(t);
  }
}

void add_progression(int left, int right) {
  add_progression_helper(1, 0, n, left, right, left);
}

long long range_sum_helper(int t, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return v[t];
  } else {
    push(t, pr - pl);
    int mid = (pl + pr) >> 1;
    return range_sum_helper(2 * t, pl, mid, l, min(r, mid)) +
      range_sum_helper(2 * t + 1, mid, pr, ::max(l, mid), r);
```

```
    }
  }

  long long range_sum(int left, int right) {
    return range_sum_helper(1, 0, n, left, right);
  }
};

segment_tree s;
int n, num_ops;

void read_array() {
  scanf("%d %d", &n, &num_ops);
  s.init(n);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    s.set(i, x);
  }
  s.build();
}

void process_ops() {
  while (num_ops--) {
    int type, l, r;
    scanf("%d %d %d", &type, &l, &r);
    if (type == T_UPDATE) {
      s.add_progression(l - 1, r);
    } else {
      printf("%lld\n", s.range_sum(l - 1, r));
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

B.2 Problema Nezzar and Binary String (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(q log n).
#include <stdio.h>
```

```
const int MAX_N = 256 * 1024;
const int MAX_OPS = 200'000;
const int ST_CLEAN = 2;

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

// Arbore de intervale cu valori 0/1, atribuire pe interval și sumă pe
// interval.
//
// Contract: state[x] poate fi:
// * 0/1 pentru a indica o valoare de atribuit pe toți descendenții lui x, sau
// * ST_CLEAN pentru a arăta că am apelat deja push.
//
// sum[node] ține cont și de state[node].
struct segment_tree {
  int sum[2 * MAX_N];
  int state[2 * MAX_N];
  int n;

  void init(char* s, int len) {
    n = next_power_of_2(len);
    for (int i = 0; i < len; i++) {
      sum[n + i] = s[i] - '0';
    }
    for (int i = len; i < n; i++) {
      sum[n + i] = 0;
    }
    for (int i = 1; i < 2 * n; i++) {
      state[i] = ST_CLEAN;
    }

    build();
  }

  void build () {
    for (int i = n - 1; i; i--) {
      sum[i] = sum[2 * i] + sum[2 * i + 1];
    }
  }

  void push(int x) {
    if (state[x] != ST_CLEAN) {
      state[2 * x] = state[2 * x + 1] = state[x];
      sum[2 * x] = sum[2 * x + 1] = sum[x] / 2;
      state[x] = ST_CLEAN;
    }
  }
```

```
void push_all() {
  for (int i = 1; i < n; i++) {
    push(i);
  }
}

void push_path(int x) {
  int bits = __builtin_popcount(n - 1);
  for (int b = bits; b; b--) {
    push(x >> b);
  }
}

void pull_path(int x) {
  for (x /= 2; x; x /= 2) {
    if (state[x] == ST_CLEAN) {
      sum[x] = sum[2 * x] + sum[2 * x + 1];
    }
  }
}

void range_set(int l, int r, int val) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;
  int size = 1;

  push_path(l);
  push_path(r);

  while (l <= r)  {
    if (l & 1) {
      sum[l] = size * val;
      state[l++] = val;
    }
    l >>= 1;

    if (!(r & 1)) {
      sum[r] = size * val;
      state[r--] = val;
    }
    r >>= 1;
    size <<= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}

int range_sum(int l, int r) {
```

```
    int result = 0;

    l += n;
    r += n;
    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        result += sum[l++];
      }
      l >>= 1;

      if (!(r & 1)) {
        result += sum[r--];
      }
      r >>= 1;
    }

    return result;
  }

  bool equals(char* s, int len) {
    push_all();

    int i = 0;
    while ((i < len) && (s[i] - '0' == sum[n + i])) {
      i++;
    }

    return (i == len);
  }
};

struct operation {
  int l, r;
};

segment_tree st;
char start[MAX_N + 1], finish[MAX_N + 1];
operation op[MAX_OPS];
int len, num_ops;

void read_data() {
  scanf("%d %d %s %s", &len, &num_ops, start, finish);
  for (int i = 0; i < num_ops; i++) {
    scanf("%d %d\n", &op[i].l, &op[i].r);
    op[i].l--;
    op[i].r--;
  }
```

```
}

bool process_op(operation op) {
  int num_ones = st.range_sum(op.l, op.r);
  int num_zeroes = (op.r - op.l + 1) - num_ones;
  if (num_ones == num_zeroes) {
    return false;
  } else {
    int majority = (num_ones > num_zeroes) ? 1 : 0;
    st.range_set(op.l, op.r, majority);
    return true;
  }
}

void process_test() {
  read_data();
  st.init(finish, len);
  int i = num_ops - 1;
  while ((i >= 0) && process_op(op[i])) {
    i--;
  }

  bool success = (i < 0) && st.equals(start, len);
  printf("%s\n", success ? "YES" : "NO");
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    process_test();
  }

  return 0;
}
```

B.3 Problema Simple (infO(1)Cup 2019)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 200'000;
const int MAX_SEGTREE_NODES = 1 << 19;
const long long INF = 1LL << 60;
const int OP_ADD = 0;

long long min(long long x, long long y) {
```

```
  return (x < y) ? x :  y;
}

long long max(long long x, long long y) {
  return (x > y) ? x :  y;
}

// Un arbore de intervale cu propagare lazy. Fiecare nod reține
// * minimul par,  maximul par, minimul impar și maximul impar;
// * o cantitate delta de adăugat pe tot subarborele.
//
// Contract: Nodul curent și-a aplicat deja delta sie însuși.
struct node {
  long long e, E, o, O;
  long long delta;

  void empty() {
    // Astfel operațiile de minim/maxim funcționează fără cazuri particulare.
    // 2x pentru că ne lăsăm loc să creștem/scădem.
    e = o = 2 * INF;
    E = O = -2 * INF;
    delta = 0;
  }

  void set(int val) {
    empty();
    if (val % 2) {
      o = O = val;
    } else {
      e = E = val;
    }
  }

  void swap() {
    long long tmp = e; e = o; o = tmp;
    tmp = E; E = O; O = tmp;
  }

  void push(node& a, node& b) {
    a.add(delta);
    b.add(delta);
    delta = 0;
  }

  void pull(node a, node b) {
    // Dacă nodul este *dirty*, atunci el știe mai bine situația curentă. Fiii
    // săi au informație perimată.
    if (!delta) {
      e = min(a.e, b.e);
      E = max(a.E, b.E);
```

```
      o = min(a.o, b.o);
      O = max(a.O, b.O);
    }
  }

  void add(long long val) {
    delta += val;
    if (val % 2) {
      // Exemplu: [9,15] și [6,30] +5 ⇒ [11,35] și [14,20].
      swap();
    }
    e += val;
    E += val;
    o += val;
    O += val;
  }
};

struct segment_tree {
  node v[MAX_SEGTREE_NODES];
  int n, bits;

  void init(int _n) {
    bits = 32 - __builtin_clz(_n - 1);
    n = 1 << bits;

    for (int i = n + _n; i < 2 * n; i++) {
      // Necesar deoarece altfel nodurile de la _n la n rămîn pe zero.
      v[i].empty();
    }
  }

  void raw_set(int pos, int val) {
    v[pos + n].set(val);
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i].pull(v[2 * i], v[2 * i + 1]);
    }
  }

  void push_path(int pos) {
    for (int b = bits - 1; b; b--) {
      int t = pos >> b;
      v[t].push(v[2 * t], v[2 * t + 1]);
    }
  }

  void pull_path(int pos) {
```

```
  for (pos /= 2; pos; pos /= 2) {
    v[pos].pull(v[2 * pos], v[2 * pos + 1]);
  }
}

void range_add(int l, int r, int val) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;
  push_path(l);
  push_path(r);

  while (l <= r)  {
    if (l & 1) {
      v[l++].add(val);
    }
    l >>= 1;

    if (!(r & 1)) {
      v[r--].add(val);
    }
    r >>= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}

node range_query(int l, int r) {
  node accumulator;
  accumulator.empty();

  l += n;
  r += n;
  push_path(l);
  push_path(r);

  while (l <= r)  {
    if (l & 1) {
      accumulator.pull(accumulator, v[l++]);
    }
    l >>= 1;

    if (!(r & 1)) {
      accumulator.pull(accumulator, v[r--]);
    }
    r >>= 1;
  }

  return accumulator;
```

```
  }
};

segment_tree st;
int n;

void read_array_into_segtree() {
  scanf("%d", &n);
  st.init(n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    st.raw_set(i, x);
  }

  st.build();
}

void process_ops() {
  int num_ops, type, l, r, val;

  scanf("%d", &num_ops);
  while (num_ops--) {
    scanf("%d %d %d", &type, &l, &r);
    l--;
    r--;
    if (type == OP_ADD) {
      scanf("%d", &val);
      st.range_add(l, r, val);
    } else {
      node nd = st.range_query(l, r);
      long long e = (nd.e > INF) ? -1 : nd.e;
      long long O = (nd.O < -INF) ? -1 : nd.O;
      printf("%lld %lld\n", e, O);
    }
  }
}

int main() {
  read_array_into_segtree();
  process_ops();
  return 0;
}
```

B.4 Problema Balama (Baraj ONI 2024)

înapoi

Sursă cu AINT iterativ (versiune online).

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

const int MAX_N = 200'000;
const int MAX_SEGTREE_NODES = 1 << 19;

struct hinge {
  int r, pos;
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

struct segment_tree_node {
  int m, delta;
};

struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n, bits;

  void init(int _n) {
    n = next_power_of_2(_n);
    bits = 31 - __builtin_clz(n);
  }

  void push_path(int node) {
    for (int b = bits; b; b--) {
      int t = node >> b;
      v[2 * t].delta += v[t].delta;
      v[2 * t].m += v[t].delta;
      v[2 * t + 1].delta += v[t].delta;
      v[2 * t + 1].m += v[t].delta;
      v[t].delta = 0;
    }
  }

  void pull_path(int node) {
```

```
    for (int t = node / 2; t; t /= 2) {
      v[t].m = v[t].delta + max(v[2 * t].m, v[2 * t + 1].m);
    }
  }

  int range_max_and_inc(int l, int r) {
    l += n;
    r += n;
    push_path(l);
    push_path(r);

    int result = 0;
    int orig_l = l, orig_r = r;

    while (l <= r)  {
      if (l & 1) {
        result = max(result, v[l].m);
        v[l].m++;
        v[l].delta++;
        l++;
      }
      l >>= 1;

      if (!(r & 1)) {
        result = max(result, v[r].m);
        v[r].m++;
        v[r].delta++;
        r--;
      }
      r >>= 1;
    }

    pull_path(orig_l);
    pull_path(orig_r);

    return result;
  }
};

hinge h[MAX_N];
int sol[MAX_N];
segment_tree st;
int n, k;

void read_data() {
  FILE* f = fopen("balama.in", "r");
  fscanf(f, "%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &h[i].r);
    h[i].pos = i;
```

```
  }
  fclose(f);
}

void sort_by_r() {
  std::sort(h, h + n, [](hinge a, hinge b) {
    return a.r > b.r;
  });
}

void scan_hinges() {
  int next_to_fill = 0;
  int i = 0;

  st.init(n);

  while (next_to_fill < k) {
    int left = max(h[i].pos - k + 1, 0);
    int right = min(h[i].pos, n - k);
    int m = st.range_max_and_inc(left, right);
    if (m == next_to_fill) {
      sol[next_to_fill++] = h[i].r;
    }
    i++;
  }
}

void write_answers() {
  FILE* f = fopen("balama.out", "w");
  for (int i = k - 1; i >= 0; i--) {
    fprintf(f, "%d ", sol[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  sort_by_r();
  scan_hinges();
  write_answers();

  return 0;
}
```

Sursă cu AINT recursiv (versiune online).

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

```
const int MAX_N = 1 << 18;

struct hinge {
  int r, pos;
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
}

// Arbore de segmente cu operațiile:
//
// 1. update: inc(left, right) -- incrementează pozițiile [left, right]
// 2. query: max(left, right) -- returnează maximul din [left, right]
//
// Invariant:
//
// * lazy_sum este valoarea de adăugat pe fiecare nod din subarbore
// * m este maximul real din subarbore, inclusiv lazy_sum
struct segment_tree {
  int m[2 * MAX_N];
  int lazy_sum[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void push(int t) {
    lazy_sum[2 * t] += lazy_sum[t];
    m[2 * t] += lazy_sum[t];
    lazy_sum[2 * t + 1] += lazy_sum[t];
    m[2 * t + 1] += lazy_sum[t];
    lazy_sum[t] = 0;
  }

  void pull(int t) {
```

```
    m[t] = ::max(m[2 * t], m[2 * t + 1]);
  }

  void inc_helper(int t, int pl, int pr, int l, int r) {
    if (l >= r) {
      return;
    } else if ((l == pl) && (r == pr)) {
      lazy_sum[t]++;
      m[t]++;
    } else {
      push(t);
      int mid = (pl + pr) >> 1;
      inc_helper(2 * t, pl, mid, l, min(r, mid));
      inc_helper(2 * t + 1, mid, pr, ::max(l, mid), r);
      pull(t);
    }
  }

  void inc(int left, int right) {
    inc_helper(1, 0, n, left, right + 1);
  }

  int max_helper(int t, int pl, int pr, int l, int r) {
    if (l >= r) {
      return 0;
    } else if ((l == pl) && (r == pr)) {
      return m[t];
    } else {
      push(t);
      int mid = (pl + pr) >> 1;
      return ::max(max_helper(2 * t, pl, mid, l, min(r, mid)),
                   max_helper(2 * t + 1, mid, pr, ::max(l, mid), r));
    }
  }

  int max(int left, int right) {
    return max_helper(1, 0, n, left, right + 1);
  }
};

hinge h[MAX_N];
int sol[MAX_N];
segment_tree st;
int n, k;

void read_data() {
  FILE* f = fopen("balama.in", "r");
  fscanf(f, "%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &h[i].r);
```

```
    h[i].pos = i;
  }
  fclose(f);
}

void sort_by_r() {
  std::sort(h, h + n, [](hinge a, hinge b) {
    return a.r > b.r;
  });
}

void scan_hinges() {
  int next_to_fill = 0;
  int i = 0;

  st.init(n);

  while (next_to_fill < k) {
    int left = max(h[i].pos - k + 1, 0);
    int right = min(h[i].pos, n - k);
    int m = st.max(left, right);
    st.inc(left, right);
    if (m == next_to_fill) {
      sol[next_to_fill++] = h[i].r;
    }
    i++;
  }
}

void write_answers() {
  FILE* f = fopen("balama.out", "w");
  for (int i = k - 1; i >= 0; i--) {
    fprintf(f, "%d ", sol[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  sort_by_r();
  scan_hinges();
  write_answers();

  return 0;
}
```

Anexa C

Arbori indexați binar

C.1 Problema The Permutation Game Again (SPOJ)

înapoi

```
#include <stdio.h>

const int MAX_N = 200'000;
const int MOD = 1'000'000'007;

struct fenwick_tree {
  int v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
    for (int i = 1; i <= n; i++) {
      v[i] = 0;
    }
  }

  void check(int pos) {
    do {
      v[pos]++;
      pos += pos & -pos;
    } while (pos <= n);
  }

  int prefix_sum(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
```

```
  }
};

fenwick_tree fen;

void solve_test() {
  int n, x;
  scanf("%d", &n);
  fen.init(n);
  long long rank = 0;

  for (int place = n; place; place--) {
    scanf("%d", &x);
    int not_seen_before = x - 1 - fen.prefix_sum(x);
    rank = (rank * place + not_seen_before) % MOD;
    fen.check(x);
  }

  rank++; // Noi calculăm rangul începînd cu 0.

  printf("%lld\n", rank);
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    solve_test();
  }

  return 0;
}
```

C.2 Problema Multiset (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 1'000'000;

struct fenwick_tree {
  int v[MAX_N + 1];
  int n, max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
```

```
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  // Returnează poziția unde suma parțială atinge sau depășește sum.
  int bin_search(int sum) {
    int pos = 0;

    for (int interval = max_p2; interval; interval >>= 1) {
      if ((pos + interval <= n) && (v[pos + interval] < sum)) {
        sum -= v[pos + interval];
        pos += interval;
      }
    }

    return pos + 1;
  }

  void remove_kth_element(int k) {
    int pos = bin_search(k);
    add(pos, -1);
  }

  int get_smallest() {
    int pos = bin_search(1);
    return (pos <= n) ? pos : 0;
  }
};

fenwick_tree fen;
int n, num_ops;

void read_initial_multiset() {
  scanf("%d %d", &n, &num_ops);
  fen.init(n);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    fen.add(x, 1);
  }
}

void process_queries() {
  while (num_ops--) {
    int x;
```

```
    scanf("%d", &x);
    if (x > 0) {
      fen.add(x, 1);
    } else {
      fen.remove_kth_element(-x);
    }
  }
}

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_initial_multiset();
  process_queries();
  int answer = fen.get_smallest();
  write_answer(answer);

  return 0;
}
```

C.3 Problema Hanoi Factory (Codeforces)

‹ înapoi • versiune online

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

const int MAX_N = 100'000;

struct ring {
  int in, out, h;
};

long long max(long long x, long long y) {
  return (x > y) ? x : y;
}

struct max_fenwick_tree {
  long long* v;
  int n;

  void init(long long* v, int n) {
    this->v = v;
    this->n = n;
    for (int i = 0; i <= n; i++) {
      v[i] = 0;
```

```
    }
  }

  void improve(int pos, long long val) {
    do {
      v[pos] = max(v[pos], val);
      pos += pos & -pos;
    } while (pos <= n);
  }

  long long prefix_max(int pos) {
    long long m = 0;
    while (pos) {
      m = max(m, v[pos]);
      pos &= pos - 1;
    }
    return m;
  }
};

ring r[MAX_N];
// folosit și pentru arborele fenwick, și pentru normalizarea mărimilor
long long v[2 * MAX_N + 1];
max_fenwick_tree fen;
int n;

void read_rings() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d %d", &r[i].in, &r[i].out, &r[i].h);
  }
}

void sort_rings() {
  std::sort(r, r + n, [](ring& a, ring& b) {
    return (a.out > b.out) || ((a.out == b.out) && (a.in > b.in));
  });
}

int bin_search(int val) {
  int l = 0, r = 2 * n;
  while (v[l] != val) { // valoarea există garantat în v
    int m = (l + r) / 2;
    if (v[m] > val) {
      r = m;
    } else {
      l = m;
    }
  }
  return l;
```

```
}

void normalize_diameters() {
  for (int i = 0; i < n; i++) {
    v[2 * i] = r[i].in;
    v[2 * i + 1] = r[i].out;
  }
  std::sort(v, v + 2 * n);

  for (int i = 0; i < n; i++) {
    r[i].in = 1 + bin_search(r[i].in);
    r[i].out = 1 + bin_search(r[i].out);
  }
}

long long solve_recurrence() {
  fen.init(v, 2 * n);

  long long max_height = 0;
  for (int i = 0; i < n; i++) {
    long long best = r[i].h + fen.prefix_max(r[i].out - 1);
    fen.improve(r[i].in, best);
    max_height = max(max_height, best);
  }

  return max_height;
}

void write_answer(long long answer) {
  printf("%lld\n", answer);
}

int main() {
  read_rings();
  sort_rings();
  normalize_diameters();
  long long answer = solve_recurrence();
  write_answer(answer);

  return 0;
}
```

C.4 Problema Subsequences (Codeforces)

‹ înapoi • versiune online

```
// Folosim un singur vector pentru cnt, înlocuind cîte un element pe rînd.
#include <stdio.h>
```

```
const int MAX_N = 100'000;

struct fenwick_tree {
  long long v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
    for (int i = 1; i <= n; i++) {
      v[i] = 0;
    }
  }

  void add(int pos, long long val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  long long prefix_sum(int pos) {
    long long s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

fenwick_tree fen;
int a[MAX_N];
long long cnt[MAX_N];
int n, k;

void read_data() {
  scanf("%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
}

void iterate_lengths() {
  for (int i = 0; i < n; i++) {
    cnt[i] = 1;
  }

  while (k--) {
    fen.init(n);
```

```
    for (int i = 0; i < n; i++) {
      long long old_cnt = cnt[i];
      cnt[i] = fen.prefix_sum(a[i] - 1);
      fen.add(a[i], old_cnt);
    }
  }
}

long long array_sum(long long* v, int n) {
  long long sum = 0;
  for (int i = 0; i < n; i++) {
    sum += v[i];
  }
  return sum;
}

void write_answer(long long answer) {
  printf("%lld\n", answer);
}

int main() {
  read_data();
  iterate_lengths();
  long long total = array_sum(cnt, n);
  write_answer(total);

  return 0;
}
```

C.5 Problema D-query (SPOJ)

‹ înapoi

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

const int MAX_N = 30000;
const int MAX_QUERIES = 200000;
const int MAX_VAL = 1000000;

struct query {
  short l, r;
  int orig_index;
  short answer;
};

struct fenwick_tree {
```

```
  short v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  short prefix_sum(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }

  short range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};

int a[MAX_N + 1];
query q[MAX_QUERIES];
short prev_occur[MAX_VAL + 1];
fenwick_tree fen;
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &a[i]);
  }
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%hd %hd", &q[i].l, &q[i].r);
    q[i].orig_index = i;
  }
}

void sort_queries_by_right_end() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.r < b.r;
  });
```

```
}

void update_last_occurrence(int pos) {
  if (prev_occur[a[pos]]) {
    fen.add(prev_occur[a[pos]], -1);
  }
  prev_occur[a[pos]] = pos;
  fen.add(pos, +1);
}

int answer_queries_ending_at(int right, int q_index) {
  while ((q_index < num_queries) && (q[q_index].r == right)) {
    q[q_index].answer = fen.range_sum(q[q_index].l, q[q_index].r);
    q_index++;
  }

  return q_index;
}

void scan_array() {
  fen.init(n);
  int q_index = 0;

  for (int i = 1; i <= n; i++) {
    update_last_occurrence(i);
    q_index = answer_queries_ending_at(i, q_index);
  }
}

void sort_queries_by_orig_index() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.orig_index < b.orig_index;
  });
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", q[i].answer);
  }
}

int main() {
  read_data();
  sort_queries_by_right_end();
  scan_array();
  sort_queries_by_orig_index();
  write_answers();

  return 0;
```

```
}
```

C.6 Problema Magic Board (CodeChef)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_SIZE = 500'000;
const int MAX_TIME = 500'000;
const int MAX_WORD_LENGTH = 8;
enum op_type { OP_ROW_SET, OP_COL_SET, OP_ROW_QUERY, OP_COL_QUERY };

struct operation {
  op_type type;
  int index, value;
};

struct fenwick_tree {
  int v[MAX_TIME + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void set(int pos) {
    add(pos, +1);
  }

  void unset(int pos) {
    add(pos, -1);
  }

  int count(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
```

```
  }
};

struct line_info {
  int time[MAX_SIZE + 1];
  bool value[MAX_SIZE + 1];
  fenwick_tree change[2];
  int size, num_ops;

  void init(int size, int num_ops) {
    this->size = size;
    this->num_ops = num_ops;
    change[0].init(num_ops);
    change[1].init(num_ops);
  }

  void update(int index, int now, int new_value) {
    int old_value = value[index];
    int old_time = time[index];

    if (old_time) {
      change[old_value].unset(old_time);
    }

    change[new_value].set(now);
    time[index] = now;
    value[index] = new_value;
  }

  int count_zeroes(int index, int now, line_info& other) {
    int last_reset = time[index];
    if (!last_reset) {
      last_reset = num_ops;
    }
    int last_value = value[index];
    int changes = other.change[1 - last_value].count(last_reset);

    return (last_value == 0)
      ? (size - changes)
      : changes;
  }

  void query(int index, int now, line_info& other) {
    int num_zeroes = count_zeroes(index, now, other);
    printf("%d\n", num_zeroes);
  }

};

line_info rows, cols;
```

```
operation read_op() {
  char word[MAX_WORD_LENGTH + 1];
  operation op;

  scanf("%s", word);
  if (word[3] == 'Q') {
    op.type = (word[0] == 'R') ? OP_ROW_QUERY : OP_COL_QUERY;
    scanf("%d", &op.index);
  } else {
    op.type = (word[0] == 'R') ? OP_ROW_SET : OP_COL_SET;
    scanf("%d %d", &op.index, &op.value);
  }

  return op;
}

void process_op(operation op, int time) {
  if (op.type == OP_ROW_SET) {
    rows.update(op.index, time, op.value);
  } else if (op.type == OP_COL_SET) {
    cols.update(op.index, time, op.value);
  } else if (op.type == OP_ROW_QUERY) {
    rows.query(op.index, time, cols);
  } else { // OP_COL_QUERY
    cols.query(op.index, time, rows);
  }
}

void process_ops() {
  int size, num_ops;
  scanf("%d %d", &size, &num_ops);
  rows.init(size, num_ops);
  cols.init(size, num_ops);

  // Inversăm direcţia timpului astfel încît operaţiile din AIB-uri să fie pe
  // prefix, nu pe sufix.
  for (int time = num_ops; time; time--) {
    operation op = read_op();
    process_op(op, time);
  }
}

int main() {
  process_ops();

  return 0;
}
```

C.7 Problema Ball (Codeforces)

‹ înapoi • versiune online

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

const int MAX_LADIES = 500'000;

struct lady {
  int x, y, z;
};

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct suffix_max_fenwick_tree {
  int v[MAX_LADIES + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void update(int pos, int val) {
    pos = n + 1 - pos;
    do {
      v[pos] = max(v[pos], val);
      pos += pos & -pos;
    } while (pos <= n);
  }

  int suffix_max(int pos) {
    pos = n + 1 - pos;
    int result = 0;
    while (pos) {
      result = max(result, v[pos]);
      pos &= pos - 1;
    }
    return result;
  }
};

lady l[MAX_LADIES];
suffix_max_fenwick_tree fen;
int pos[MAX_LADIES]; // folosit pentru normalizare
int n;
```

```
void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &l[i].x);
  }
  for (int i = 0; i < n; i++) {
    scanf("%d", &l[i].y);
  }
  for (int i = 0; i < n; i++) {
    scanf("%d", &l[i].z);
  }
}

void normalize_x() {
  for (int i = 0; i < n; i++) {
    pos[i] = i;
  }

  std::sort(pos, pos + n, [](int a, int b) {
    return l[a].x < l[b].x;
  });

  int old_x = l[pos[0]].x, new_x = 1;
  for (int i = 0; i < n; i++) {
    if (l[pos[i]].x != old_x) {
      old_x = l[pos[i]].x;
      new_x++;
    }
    l[pos[i]].x = new_x;
  }
}

void sort_ladies_by_z() {
  std::sort(l, l + n, [](lady& a, lady& b) {
    return a.z > b.z;
  });
}

int process_equal_z_batch(int start, int end) {
  int result = 0;

  for (int i = start; i < end; i++) {
    int prev_max_y = fen.suffix_max(l[i].x + 1);
    result += (prev_max_y > l[i].y);
  }
  for (int i = start; i < end; i++) {
    fen.update(l[i].x, l[i].y);
  }

  return result;
```

```
}

int count_self_murderers() {
  fen.init(n);

  int result = 0;

  // Procesează calupuri de valori z egale. Aceste doamne nu se domină una pe
  // alta.
  int i = 0;
  while (i < n) {
    int j = i;
    while ((j < n) && (l[j].z == l[i].z)) {
      j++;
    }
    result += process_equal_z_batch(i, j);
    i = j;
  }

  return result;
}

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_data();
  normalize_x();
  sort_ladies_by_z();
  int answer = count_self_murderers();
  write_answer(answer);

  return 0;
}
```

C.8 Problema Medwalk revizitată (Lot 2025)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>
#include <vector>

const int MAX_N = 100'000;
const int MAX_QUERIES = 100'000;
const int MAX_VALUE = 300'000;
const int MAX_SEGTREE_NODES = 1 << 18;
```

```
const int INF = 1'000'000;
const int OP_UPDATE = 1;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct matrix {
  int v[2][MAX_N + 1];

  void set(int row, int col, int val) {
    v[row][col] = val;
  }

  int get_min(int col) {
    return min(v[0][col], v[1][col]);
  }

  int get_max(int col) {
    return max(v[0][col], v[1][col]);
  }
};

struct query {
  int type;
  // Syntactic sugar ca să putem folosi <row, col, val> sau <left, right>, nu
  // <x, y, z>.
  union { int row; int left; };
  union { int col; int right; };
  int val;
};

matrix mat;
query q[MAX_QUERIES];
int n, num_queries;

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

// Admite actualizări punctuale și interogări de minim pe interval.
struct min_segment_tree {
  int v[MAX_SEGTREE_NODES];
  int n;

  void init(int size) {
```

```
    n = next_power_of_2(size);
  }

  void update(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = min(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int range_min(int l, int r) {
    l += n;
    r += n;
    int result = INF;

    while (l <= r)  {
      if (l & 1) {
        result = min(result, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = min(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

// Kudos https://usaco.guide/plat/2DRQ?lang=cpp#offline-2d-bit și
// https://kilonova.ro/submissions/752782
//
// Arbore Fenwick 2D offline.
struct fenwick_2d {
  // Valorile distincte pe fiecare coloană.
  std::vector<int> col_rows[MAX_VALUE + 1];

  // Arborele 1D pe fiecare coloană, peste valorile existente.
  std::vector<int> col_fen[MAX_VALUE + 1];

  int n, max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
    for (int i = 0; i <= n; i++) {
      col_rows[i].push_back(0);
```

```
  }
}

// Notează faptul că rîndul row va fi folosit cel puțin o dată pe coloana
// col.
void reserve(int row, int col) {
  do {
    col_rows[col].push_back(row);
    col += col & -col;
  } while (col <= n);
}

void build() {
  for (int col = 1; col <= n; col++) {
    std::vector<int>& v = col_rows[col]; // syntactic sugar
    std::sort(v.begin(), v.end());
    v.erase(std::unique(v.begin(), v.end()), v.end());
    col_fen[col].resize(v.size() + 1);
  }
}

int leftmost_gte(int row, int col) {
  std::vector<int>& v = col_rows[col];
              return std::lower_bound(v.begin(), v.end(), row) - v.begin();
}

// Adaugă delta (bifează / debifează) pentru rîndul row și toți succesorii
// săi în fiecare AIB 1D, atît pe coloana col cît și pe toate coloanele
// succesoare în AIB-ul 2D.
void add(int row, int col, int delta) {
  do {
    int pos = leftmost_gte(row, col);
    do {
      col_fen[col][pos] += delta;
      pos += pos & -pos;
    } while (pos <= (int)col_fen[col].size());
    col += col & -col;
  } while (col <= n);
}

int prefix_sum(int pos, int col) {
  int s = 0;
  while (pos) {
    s += col_fen[col][pos];
    pos &= pos - 1;
  }
  return s;
}

int count_less_than(int val, int col) {
```

```
    // Reminder: AIB-ul 1D este normalizat. Află pe ce rînd se află val.
    int pos = leftmost_gte(val, col);
    return prefix_sum(pos - 1, col);
  }

  int count_in_range(int col, int l, int r) {
    return count_less_than(r + 1, col) - count_less_than(l, col);
  }

  // Contract: k este 0-based, iar [l, r] este un interval închis de valori
  // (rînduri).
  int kth_element(int k, int l, int r) {
    int col = 0;

    for (int interval = max_p2; interval; interval >>= 1) {
      if (col + interval <= n) {
        int cnt = count_in_range(col + interval, l, r);
        if (cnt <= k) {
          k -= cnt;
          col += interval;
        }
      }
    }

    return col + 1;
  }
};

min_segment_tree maxima;
fenwick_2d minima;

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int r = 0; r < 2; r++) {
    for (int c = 1; c <= n; c++) {
      int x;
      scanf("%d", &x);
      mat.set(r, c, x);
    }
  }

  for (int i = 0; i < num_queries; i++) {
    query& z = q[i];
    scanf("%d", &z.type);
    if (z.type == OP_UPDATE) {
      scanf("%d %d %d", &z.row, &z.col, &z.val);
      z.row--;
    } else {
      scanf("%d %d", &z.left, &z.right);
    }
```

```
  }
}

void build_maxima_segtree() {
  maxima.init(n + 1);
  for (int i = 1; i <= n; i++) {
    maxima.update(i, mat.get_max(i));
  }
}

void simulate() {
  // Rezervă minimele inițiale.
  for (int c = 1; c <= n; c++) {
    minima.reserve(c, mat.get_min(c));
  }

  // Rezervă minimele care iau naștere prin actualizări.
  matrix mat_copy = mat;
  for (int i = 0; i < num_queries; i++) {
    if (q[i].type == OP_UPDATE) {
      mat_copy.set(q[i].row, q[i].col, q[i].val);
      minima.reserve(q[i].col, mat_copy.get_min(q[i].col));
    }
  }
}

void build_fenwick() {
  minima.init(MAX_VALUE);
  simulate();
  minima.build();

  for (int c = 1; c <= n; c++) {
    minima.add(c, mat.get_min(c), +1);
  }
}

void update(int row, int col, int val) {
  int old_min = mat.get_min(col);
  mat.set(row, col, val);
  int new_min = mat.get_min(col);

  maxima.update(col, mat.get_max(col));

  if (new_min != old_min) {
    minima.add(col, old_min, -1);
    minima.add(col, new_min, +1);
  }
}

int query(int left, int right) {
```

```
  int len = right - left + 2;
  int median_pos = (len - 1) / 2;
  int median = minima.kth_element(median_pos, left, right);
  int best_max = maxima.range_min(left, right);

  if (best_max >= median) {
    return median;
  } else {
    // best_max trebuie inserat înainte de median_pos. Răspunsul poate fi la
    // poziția median_pos - 1 sau poate fi chiar best_max.
    int prev = minima.kth_element(median_pos - 1, left, right);
    return max(prev, best_max);
  }
}

void process_queries() {
  for (int i = 0; i < num_queries; i++) {
    if (q[i].type == OP_UPDATE) {
      update(q[i].row, q[i].col, q[i].val);
    } else {
      printf("%d\n", query(q[i].left, q[i].right));
    }
  }
}

int main() {
  read_data();
  build_maxima_segtree();
  build_fenwick();
  process_queries();

  return 0;
}
```

Anexa D

Descompunere în radical

D.1 Problema Mexitate (ONI 2018 clasa a 9-a)

înapoi

Sursă naivă (versiune online).

```
#include <stdio.h>

const int MAX_ELEMS = 400'000;
const int MOD = 1'000'000'007;

// Costul calculării funcției mex() este mare. Versiunea 2 va folosi o
// structură mai eficientă pentru frequency_tracker.
struct frequency_tracker {
  int f[MAX_ELEMS + 2]; // rows * cols + 1 în cel mai rău caz

  void add(int x) {
    f[x]++;
  }

  void remove(int x) {
    f[x]--;
  }

  int mex() {
    int x = 1;
    while (f[x]) {
      x++;
    }
    return x;
  }
};

int mat[MAX_ELEMS];
```

```
frequency_tracker ft;
int rows, cols, k, l;

void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void read_right_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[r * cols + c]);
    }
  }
}

void read_transposed_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[c * rows + r]);
    }
  }
}

void read_data() {
  FILE* f = fopen("mexitate.in", "r");
  fscanf(f, "%d %d %d %d", &rows, &cols, &k, &l);
  if (k <= l) {
    read_right_matrix(f);
  } else {
    read_transposed_matrix(f);
    swap(rows, cols);
    swap(k, l);
  }
  fclose(f);
}

int get(int r, int c) {
  return mat[r * cols + c];
}

int north_west_corner() {
  for (int r = 0; r < k; r++) {
    for (int c = 0; c < l; c++) {
      ft.add(get(r, c));
    }
  }
  return ft.mex();
}
```

```
void move_right(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c));
    ft.add(get(i, c + l));
  }
}

void move_left(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c + l - 1));
    ft.add(get(i, c - 1));
  }
}

void move_down(int r, int c) {
  for (int j = c; j < c + l; j++) {
    ft.remove(get(r, j));
    ft.add(get(r + k, j));
  }
}


int left_right_snake() {
  north_west_corner();
  long long result = ft.mex();
  int r = 0, c = 0;
  int final_r = rows - k;
  int final_c = (final_r % 2) ? 0 : (cols - l);

  while ((r != final_r) || (c != final_c)) {
    if ((r % 2 == 0) && (c < cols - l)) {
      move_right(r, c++);
    } else if ((r % 2 == 1) && (c > 0)) {
      move_left(r, c--);
    } else {
      move_down(r++, c);
    }
    result = result * ft.mex() % MOD;
  }

  return result;
}

void write_answer(int answer) {
  FILE* f = fopen("mexitate.out", "w");
  fprintf(f, "%d\n", answer);
  fclose(f);
}
```

```
int main() {
  read_data();
  int answer = left_right_snake();
  write_answer(answer);

  return 0;
}
```

Sursă eficientă (versiune online).

```
#include <stdio.h>

const int MAX_ELEMS = 400'000;
const int BLOCK_SIZE = 400;
const int MAX_BLOCKS = (MAX_ELEMS - 1) / BLOCK_SIZE + 1;
const int MOD = 1'000'000'007;

struct frequency_tracker {
  int f[MAX_ELEMS + 2]; // rows * cols + 1 în cel mai rău caz
  int block_nonzero[MAX_BLOCKS];

  void init() {
    add(0); // Ca să nu-l returnăm niciodată.
  }

  void add(int x) {
    if (++f[x] == 1) {
      block_nonzero[x / BLOCK_SIZE]++;
    }
  }

  void remove(int x) {
    if (--f[x] == 0) {
      block_nonzero[x / BLOCK_SIZE]--;
    }
  }

  int mex() {
    int b = 0;
    while (block_nonzero[b] == BLOCK_SIZE) {
      b++;
    }
    int x = b * BLOCK_SIZE;
    while (f[x]) {
      x++;
    }
    return x;
  }
};
```

```
int mat[MAX_ELEMS];
frequency_tracker ft;
int rows, cols, k, l;

void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void read_right_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[r * cols + c]);
    }
  }
}

void read_transposed_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[c * rows + r]);
    }
  }
}

void read_data() {
  FILE* f = fopen("mexitate.in", "r");
  fscanf(f, "%d %d %d %d", &rows, &cols, &k, &l);
  if (k <= l) {
    read_right_matrix(f);
  } else {
    read_transposed_matrix(f);
    swap(rows, cols);
    swap(k, l);
  }
  fclose(f);
}

int get(int r, int c) {
  return mat[r * cols + c];
}

int north_west_corner() {
  for (int r = 0; r < k; r++) {
    for (int c = 0; c < l; c++) {
      ft.add(get(r, c));
    }
  }
```

```
  return ft.mex();
}

void move_right(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c));
    ft.add(get(i, c + l));
  }
}

void move_left(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c + l - 1));
    ft.add(get(i, c - 1));
  }
}

void move_down(int r, int c) {
  for (int j = c; j < c + l; j++) {
    ft.remove(get(r, j));
    ft.add(get(r + k, j));
  }
}


int left_right_snake() {
  ft.init();
  north_west_corner();
  long long result = ft.mex();
  int r = 0, c = 0;
  int final_r = rows - k;
  int final_c = (final_r % 2) ? 0 : (cols - l);

  while ((r != final_r) || (c != final_c)) {
    if ((r % 2 == 0) && (c < cols - l)) {
      move_right(r, c++);
    } else if ((r % 2 == 1) && (c > 0)) {
      move_left(r, c--);
    } else {
      move_down(r++, c);
    }
    result = result * ft.mex() % MOD;
  }

  return result;
}

void write_answer(int answer) {
  FILE* f = fopen("mexitate.out", "w");
  fprintf(f, "%d\n", answer);
```

```
  fclose(f);
}

int main() {
  read_data();
  int answer = left_right_snake();
  write_answer(answer);

  return 0;
}
```

D.2 Problema Give Away (SPOJ)

înapoi

Sursă cu multiseturi PBDS.

```
// Complexitate: O(Q sqrt N log N).
//
// Metodă: Descompunere în radical. Pe fiecare bloc reține vectorul naiv și un
// set de valori pentru căutări în timp logaritmic. Avem nevoie de multisets
// pentru că pot exista duplicate și avem nevoie de PBDS pentru funcția
// order_of_key.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <stdio.h>

const int MAX_N = 500000;
const int BLOCK_SIZE = 10000;
const int MAX_BLOCKS = (MAX_N - 1) / BLOCK_SIZE + 1;
const int T_QUERY = 0;

// Multiseturile PBDS sînt obscene, dar par să meargă. Ștergerile necesită cod
// extra. Vezi https://stackoverflow.com/q/59731946/6022817
typedef __gnu_pbds::tree<
  int,
  __gnu_pbds::null_type,
  std::less_equal<int>,
  __gnu_pbds::rb_tree_tag,
  __gnu_pbds::tree_order_statistics_node_update
  > ordered_set;

// Toate intervalele sînt [închis, deschis).
struct block {
  int v[BLOCK_SIZE];
  ordered_set s;

  void set(int pos, int val) {
```

```
    if (v[pos]) {
      int rank = s.order_of_key(v[pos]);
      ordered_set::iterator it = s.find_by_order(rank);
      s.erase(it);
    }
    v[pos] = val;
    s.insert(v[pos]);
  }

  int partial_count(int l, int r, int val) {
    int cnt = 0;
    while (l < r) {
      cnt += (v[l++] >= val);
    }
    return cnt;
  }

  int prefix_count(int end, int val) {
    return partial_count(0, end, val);
  }

  int suffix_count(int start, int val) {
    return partial_count(start, BLOCK_SIZE, val);
  }

  int whole_count(int val) {
    return s.size() - s.order_of_key(val);
  }
};

block b[MAX_BLOCKS];
int n;

void read_array() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    b[i / BLOCK_SIZE].set(i % BLOCK_SIZE, x);
  }
}

int process_query(int l, int r, int val) {
  int bl = l / BLOCK_SIZE, offset_l = l % BLOCK_SIZE;
  int br = r / BLOCK_SIZE, offset_r = r % BLOCK_SIZE;
  if (bl == br) {
    return b[bl].partial_count(offset_l, offset_r, val);
  } else {
    int cnt = b[bl].suffix_count(offset_l, val)
```

```
      + b[br].prefix_count(offset_r, val);
    for (int i = bl + 1; i < br; i++) {
      cnt += b[i].whole_count(val);
    }
    return cnt;
  }
}

void process_update(int pos, int val) {
  b[pos / BLOCK_SIZE].set(pos % BLOCK_SIZE, val);
}

void process_ops() {
  int num_ops, type, pos1, pos2, val;
  scanf("%d", &num_ops);

  while (num_ops--) {
    scanf("%d", &type);
    if (type == T_QUERY) {
      scanf("%d %d %d", &pos1, &pos2, &val);
      int count = process_query(pos1 - 1, pos2, val);
      printf("%d\n", count);
    } else {
      scanf("%d %d", &pos1, &val);
      process_update(pos1 - 1, val);
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

Sursă elementară.

```
// Complexitate: O(Q sqrt N log N).
//
// Metodă: Descompunere în radical. Pe fiecare bloc reține vectorul naiv și un
// vector sortat pentru căutări în timp logaritmic.
#include <algorithm>
#include <stdio.h>

const int MAX_N = 500'000;
const int BLOCK_SIZE = 3'000;
const int MAX_BLOCKS = (MAX_N - 1) / BLOCK_SIZE + 1;
const int T_QUERY = 0;
```

```
// Toate intervalele sînt [închis, deschis).
struct block {
  int v[BLOCK_SIZE];
  int s[BLOCK_SIZE];
  int size;

  void push(int val) {
    v[size] = s[size] = val;
    size++;
  }

  void sort() {
    std::sort(s, s + size);
  }

  // Returnează cea mai din stînga poziție a unui element >= val.
  int bin_search(int val) {
    if (val < s[0]) {
      return 0;
    } else if (val > s[size - 1]) {
      return size;
    }
    int l = -1, r = size - 1; // (l, r]

    while (r - l > 1) {
      int mid = (l + r) >> 1;
      if (s[mid] < val) {
        l = mid;
      } else {
        r = mid;
      }
    }

    return r;
  }

  void migrate_left(int pos) {
    int save = s[pos];
    while (pos && (s[pos - 1] > save)) {
      s[pos] = s[pos - 1];
      pos--;
    }
    s[pos] = save;
  }

  void migrate_right(int pos) {
    int save = s[pos];
    while ((pos < size - 1) && (s[pos + 1] < save)) {
      s[pos] = s[pos + 1];
```

```
      pos++;
    }
    s[pos] = save;
  }

  void set(int pos, int val) {
    int sorted_pos = bin_search(v[pos]);
    s[sorted_pos] = val;
    migrate_left(sorted_pos);
    migrate_right(sorted_pos);
    v[pos] = val;
  }

  int partial_count(int l, int r, int val) {
    int cnt = 0;
    while (l < r) {
      cnt += (v[l++] >= val);
    }
    return cnt;
  }

  int prefix_count(int end, int val) {
    return partial_count(0, end, val);
  }

  int suffix_count(int start, int val) {
    return partial_count(start, BLOCK_SIZE, val);
  }

  int whole_count(int val) {
    return size - bin_search(val);
  }
};

block b[MAX_BLOCKS];
int n;

void read_array() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    b[i / BLOCK_SIZE].push(x);
  }

  int end_block = (n - 1) / BLOCK_SIZE + 1;
  for (int i = 0; i < end_block; i++) {
    b[i].sort();
  }
```

```
}

int process_query(int l, int r, int val) {
  int bl = l / BLOCK_SIZE, offset_l = l % BLOCK_SIZE;
  int br = r / BLOCK_SIZE, offset_r = r % BLOCK_SIZE;
  if (bl == br) {
    return b[bl].partial_count(offset_l, offset_r, val);
  } else {
    int cnt = b[bl].suffix_count(offset_l, val)
      + b[br].prefix_count(offset_r, val);
    for (int i = bl + 1; i < br; i++) {
      cnt += b[i].whole_count(val);
    }
    return cnt;
  }
}

void process_update(int pos, int val) {
  b[pos / BLOCK_SIZE].set(pos % BLOCK_SIZE, val);
}

void process_ops() {
  int num_ops, type, pos1, pos2, val;
  scanf("%d", &num_ops);

  while (num_ops--) {
    scanf("%d", &type);
    if (type == T_QUERY) {
      scanf("%d %d %d", &pos1, &pos2, &val);
      int count = process_query(pos1 - 1, pos2, val);
      printf("%d\n", count);
    } else {
      scanf("%d %d", &pos1, &val);
      process_update(pos1 - 1, val);
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

D.3 Problema Holes (Codeforces)

‹ înapoi • versiune online

```cpp
#include <stdio.h>

const int MAX_N = 100'000;
// Preferăm blocuri puțin mai mari deoarece actualizările au *cache locality*,
// pe cînd interogările sar mai mult.
const int BUCKET_SIZE = 1'000;
const int OP_UPDATE = 0;

struct hole {
  int power;
  int last;  // ultima destinație din același bloc
  int jumps; // numărul de salturi pînă la last
};

hole h[MAX_N + 1];
int n, num_queries;

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int i = 0; i < n; i++) {
    scanf("%d", &h[i].power);
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

// Actualizează găurile de la începutul blocului pînă la pos invlusiv.
void update_bucket_of(int pos) {
  int start = pos / BUCKET_SIZE * BUCKET_SIZE;
  int end = min(start + BUCKET_SIZE, n);
  for (int i = pos; i >= start; i--) {
    int dest = i + h[i].power;
    if (dest < end) {
      h[i].last = h[dest].last;
      h[i].jumps = h[dest].jumps + 1;
    } else {
      h[i].last = i;
      h[i].jumps = 0;
    }
  }
}

void init_buckets() {
  for (int start = 0; start < n; start += BUCKET_SIZE) {
    int end = min(start + BUCKET_SIZE - 1, n - 1);
    update_bucket_of(end);
  }
```

```
}

void query(int pos, int* last, int* count) {
  *count = 0;

  do {
    *count += h[pos].jumps + 1;
    *last = h[pos].last;
    pos = *last + h[*last].power;
  } while (pos < n);
}

void process_queries() {
  while (num_queries--) {
    int type, a;
    scanf("%d %d", &type, &a);
    a--;

    if (type == OP_UPDATE) {
      int power;
      scanf("%d", &power);
      h[a].power = power;
      update_bucket_of(a);
    } else {
      int last, num_jumps;
      query(a, &last, &num_jumps);
      printf("%d %d\n", 1 + last, num_jumps);
    }
  }
}

int main() {
  read_data();
  init_buckets();
  process_queries();

  return 0;
}
```

D.4 Problema Piezișă (Baraj ONI 2022)

‹ înapoi

Sursă forță brută (versiune online).

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

```cpp
#define MAX_N 500000
#define INFINITY (MAX_N + 1)
#define NONE -1

int v[MAX_N + 1]; // partial xor's
int pos[MAX_N + 1];
int first[MAX_N + 2];
int n, distinct;

int max(int x, int y) {
  return (x > y) ? x : y;
}

// Returnează cea mai din dreapta apariție a lui val pe poziția p sau înainte,
// sau NONE dacă val nu apare pe poziția p sau înainte.
int rightmost(int val, int p) {
  int l = first[val], r = first[val + 1]; // [l, r)
  while (r - l > 1) {
    int m = (l + r) >> 1;
    if (pos[m] > p) {
      r = m;
    } else {
      l = m;
    }
  }

  // Caz special: p < orice poziție unde apare val. Căutarea binară normală ar
  // returna pos[l].
  return (pos[l] <= p) ? pos[l] : NONE;
}

int main() {
  FILE* fin = fopen("piezisa.in", "r");
  FILE* fout = fopen("piezisa.out", "w");
  fscanf(fin, "%d", &n);

  // Citește datele și calculează xor-uri parțiale.
  v[0] = 0;
  for (int i = 1; i <= n; i++) {
    int x;
    fscanf(fin, "%d", &x);
    v[i] = v[i - 1] ^ x;
  }

  // Sortează pozițiile după valoarea xor parțială, apoi după poziție.
  for (int i = 0; i <= n; i++) {
    pos[i] = i;
  }
  std::sort(pos, pos + n + 1, [](int a, int b) {
    return (v[a] < v[b]) || ((v[a] == v[b]) && (a < b));
```

```
  });

  // Renumerotează valorile începînd cu n; colectează pozițiile.
  int from = -1;
  distinct = 0;
  for (int i = 0; i <= n; i++) {
    if (v[pos[i]] != from) {
      from = v[pos[i]];
      first[distinct++] = i;
    }
    v[pos[i]] = distinct - 1;
  }
  first[distinct] = n + 1;
  // Acum pos[first[i]...first[i+1]] conține o listă ordonată cu pozițiile
  // aparițiilor valorii i.

  int num_queries;
  fscanf(fin, "%d", &num_queries);
  while (num_queries--) {
    int l, r;
    fscanf(fin, "%d %d", &l, &r);
    r++;

    int end = r, best = INFINITY;
    // cît timp avem loc să avansăm și sperăm să îmbunătățim soluția existentă
    while ((end <= n) && (end - l < best)) {
      int start = rightmost(v[end], l);
      if ((start != NONE) && (end - start < best)) {
        best = end - start;
      }
      end++;
    }

    fprintf(fout, "%d\n", (best == INFINITY) ? NONE : best);
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Sursă cu metoda 1 (versiune online).

```
#include <algorithm>
#include <math.h>
#include <stdio.h>

const int MAX_BUCKETS = 354;
```

```
const int MAX_BUCKET_SIZE = 1416;
const int MAX_N = MAX_BUCKETS * MAX_BUCKET_SIZE;
const int MAX_Q = 500000;
const int INF = MAX_N + 1;
const int NONE = -1;

struct query {
  int l, r, orig_index;
};

int v[MAX_N + 1]; // xor-uri parțiale
int pos[MAX_N + 1];
int ptr[MAX_N + 1];
int last[MAX_N + 1];
int best[MAX_BUCKETS];

query q[MAX_Q];
// Logic ar trebui stocat în query.answer, dar astfel evităm o sortare.
int answer[MAX_Q];

int n, num_queries;
int bs, nb;

void read_array() {
  scanf("%d", &n);

  v[0] = 0;
  for (int i = 1; i <= n; i++) {
    scanf("%d", &v[i]);
    v[i] ^= v[i - 1];
  }
}

void read_queries() {
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].l, &q[i].r);
    q[i].r++;
    q[i].orig_index = i;
  }
}

void sort_queries() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return (a.l < b.l);
  });
}

void sort_positions() {
  for (int i = 0; i <= n; i++) {
```

```
    pos[i] = i;
  }
  std::sort(pos, pos + n + 1, [](int a, int b) {
    return (v[a] < v[b]) || ((v[a] == v[b]) && (a > b));
  });
}

void preprocess_values() {
  nb = 0.5 * sqrt(n + 1); // determinat experimental
  bs = n / nb + 1;

  sort_positions();

  // renumerotează valorile de la 0; creează pointeri
  int prev = -1, distinct = 0;
  for (int i = 0; i <= n; i++) {
    if (v[pos[i]] == prev) {
      v[pos[i]] = distinct - 1;
      bool same_bucket = (pos[i] / bs == pos[i - 1] / bs);
      ptr[pos[i]] = same_bucket
        ? ptr[pos[i - 1]]
        : pos[i - 1];
    } else {
      // începi o serie nouă de la sfîrșitul vectorului
      prev = v[pos[i]];
      v[pos[i]] = distinct++;
      ptr[pos[i]] = NONE;
    }
  }
}

inline int min(int x, int y) {
  return (x < y) ? x : y;
}

int answer_query(int l, int r) {
  int result = INF;

  // procedează incremental pînă la blocul următor
  int b = r / bs + 1, boundary = min(b * bs, n + 1);
  while (r < boundary) {
    result = min(result, r - last[v[r]]);
    r++;
  }

  // procedează bloc cu bloc restul vectorului
  while (b < nb) {
    result = min(result, best[b++]);
  }
```

```
  return (result == INF) ? NONE : result;
}

void scan() {
  for (int i = 0; i <= n; i++) {
    last[i] = -INF;
  }
  for (int i = 0; i < nb; i++) {
    best[i] = INF;
  }

  int qi = 0;
  for (int l = 0; l <= n; l++) {
    // actualizează-l pe last
    last[v[l]] = l;

    // actualizează-l pe best
    for (int i = ptr[l]; i != NONE; i = ptr[i]) {
      int b = i / bs;
      best[b] = min(best[b], i - l);
    }

    // răspunde la interogările care încep la l
    while (qi < num_queries && q[qi].l == l) {
      answer[q[qi].orig_index] = answer_query(q[qi].l, q[qi].r);
      qi++;
    }
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_array();
  read_queries();
  sort_queries();
  preprocess_values();
  scan();
  write_answers();

  return 0;
}
```

Sursă cu metoda 2 (versiune online).

```cpp
// Rescrisă după https://infoarena.ro/job_detail/3032904
//
// Complexitate: O((N + Q) sqrt(N)).
#include <algorithm>
#include <stdio.h>

const int MAX_N = 500'000;
const int BLOCK_SIZE = 700;
const int NUM_BLOCKS = MAX_N / BLOCK_SIZE + 1;
const int MAX_Q = 500'000;
const int INFINITY = 1'000'000;
const int NONE = -1;

struct query {
  int l, r;
  int orig_index;
  int block;
};

int v[MAX_N + 1];
query q[MAX_Q];
int left[MAX_N + 1], right[MAX_N + 1];
int* pos = left; // folosit inițial pentru normalizare;

// Logic ar trebui stocat în query.answer, dar astfel evităm o sortare.
int answer[MAX_Q];

int n, num_queries, max_value;

void read_array() {
  scanf("%d", &n);

  v[0] = 0;
  for (int i = 1; i <= n; i++) {
    scanf("%d", &v[i]);
    v[i] ^= v[i - 1];
  }
}

void read_queries() {
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].l, &q[i].r);
    q[i].l++;
    q[i].r++; // inteval închis, indexat de la 1
    q[i].orig_index = i;
    q[i].block = q[i].l / BLOCK_SIZE;
  }
}
```

```
void normalize_array() {
  for (int i = 1; i <= n; i++) {
    pos[i] = i;
  }
  std::sort(pos + 1, pos + n + 1, [](int a, int b) {
    return v[a] < v[b];
  });

  int prev = NONE;
  max_value = NONE;
  for (int i = 0; i <= n; i++) {
    if (v[pos[i]] != prev) {
      max_value++;
    }
    prev = v[pos[i]];
    v[pos[i]] = max_value;
  }
}

void sort_queries_by_left_block() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return (a.block < b.block) ||
      ((a.block == b.block) && (a.r > b.r));
  });
}

void init_left() {
  for (int i = 0; i <= max_value; i++) {
    left[i] = -INFINITY;
  }
}

void init_right() {
  for (int i = 0; i <= max_value; i++) {
    right[i] = +INFINITY;
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

void answer_query(query q, int closest_left_right) {
  int best = closest_left_right;
  for (int i = q.block * BLOCK_SIZE; i < q.l; i++) {
    best = min(best, right[v[i]] - i);
  }

  answer[q.orig_index] = (best > n) ? NONE : best;
```

```
}

int answer_queries_in_block(int block, int q_index) {
  init_right();
  int ptr = n + 1;
  int closest_left_right = INFINITY;
  while ((q_index < num_queries) && (q[q_index].block == block)) {
    while (ptr > q[q_index].r) {
      --ptr;
      right[v[ptr]] = ptr;
      int dist = ptr - left[v[ptr]];
      closest_left_right = min(closest_left_right, dist);
    }
    answer_query(q[q_index], closest_left_right);
    q_index++;
  }

  return q_index;
}

void update_left(int block) {
  int start = block * BLOCK_SIZE;
  int end = min(start + BLOCK_SIZE - 1, n);
  for (int i = start; i <= end; i++) {
    left[v[i]] = i;
  }
}

void scan_blocks() {
  init_left();
  int q_index = 0;

  int last_block = n / BLOCK_SIZE;
  for (int b = 0; b <= last_block; b++) {
    q_index = answer_queries_in_block(b, q_index);
    update_left(b);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_array();
  read_queries();
  normalize_array();
  sort_queries_by_left_block();
```

```
  scan_blocks();

  write_answers();

  return 0;
}
```

D.5 Problema Serega and Fun (Codeforces)

înapoi

Sursă cu `deque` (versiune online).

```
#include <deque>
#include <stdio.h>
#include <unordered_map>

const int MAX_N = 100'000;
const int BUCKET_SIZE = 2'000;
const int MAX_BUCKETS = (MAX_N - 1) / BUCKET_SIZE + 1;
const int TYPE_ROTATE = 1;
const int TYPE_COUNT = 2;

struct freq {
  std::unordered_map<int, short> map;

  void add(int val) {
    map[val]++;
  }

  void remove(int val) {
    auto it = map.find(val);
    if (it->second == 1) {
      map.erase(it);
    } else {
      it->second--;
    }
  }

  int count(int val) {
    auto it = map.find(val);
    return (it == map.end()) ? 0 : it->second;
  }
};

struct bucket {
  freq f;
```

```
std::deque<int> deq;

void push(int val) {
  deq.push_back(val);
  f.add(val);
}

void set(int pos, int val) {
  f.remove(deq[pos]);
  deq[pos] = val;
  f.add(val);
}

int shift(int val) {
  deq.push_front(val);
  f.add(val);
  int last = deq.back();
  deq.pop_back();
  f.remove(last);
  return last;
}

void rotate(int l, int r) {
  int val = deq[r];
  deq.erase(deq.begin() + r);
  deq.insert(deq.begin() + l, val);
}

int remove(int pos) {
  int val = deq[pos];
  f.remove(val);
  deq.erase(deq.begin() + pos);
  return val;
}

int remove_last() {
  return remove(BUCKET_SIZE - 1);
}

void insert(int pos, int val) {
  deq.insert(deq.begin() + pos, val);
  f.add(val);
}

void insert_first(int val) {
  insert(0, val);
}

int partial_count(int l, int r, int k) {
  int result = 0;
```

```
    while (l <= r) {
      result += (deq[l++] == k);
    }
    return result;
  }

  int prefix_count(int pos, int k) {
    return partial_count(0, pos, k);
  }

  int suffix_count(int pos, int k) {
    return partial_count(pos, BUCKET_SIZE - 1, k);
  }

};

bucket buck[MAX_BUCKETS];
int n, num_ops;
int last_answer;

void read_data() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    buck[i / BUCKET_SIZE].push(x);
  }

  scanf("%d", &num_ops);
}

int bucket_count(int l, int r, int k) {
  int result = 0;
  while (l < r) {
    result += buck[l++].f.count(k);
  }
  return result;
}

void process_count_op(int l, int r, int k) {
  int bl = l / BUCKET_SIZE, offset_l = l % BUCKET_SIZE;
  int br = r / BUCKET_SIZE, offset_r = r % BUCKET_SIZE;
  if (bl == br) {
    last_answer = buck[bl].partial_count(offset_l, offset_r, k);
  } else {
    last_answer =
      buck[bl].suffix_count(offset_l, k) +
      bucket_count(bl + 1, br, k) +
      buck[br].prefix_count(offset_r, k);
```

```
  }

  printf("%d\n", last_answer);
}

void process_rotate_op(int l, int r) {
  int bl = l / BUCKET_SIZE, offset_l = l % BUCKET_SIZE;
  int br = r / BUCKET_SIZE, offset_r = r % BUCKET_SIZE;
  if (bl == br) {
    buck[bl].rotate(offset_l, offset_r);
  } else {
    int from_left = buck[bl].remove_last();
    for (int b = bl + 1; b < br; b++) {
      from_left = buck[b].shift(from_left);
    }
    int to_right = buck[br].remove(offset_r);
    buck[br].insert_first(from_left);
    buck[bl].insert(offset_l, to_right);
  }
}

int transform(int x) {
  return last_answer
    ? (x + last_answer - 1) % n + 1
    : x;
}

void process_ops() {
  while (num_ops--) {
    int type, l, r, k;
    scanf("%d %d %d", &type, &l, &r);
    l = transform(l) - 1;
    r = transform(r) - 1;
    if (l > r) {
      int tmp = l; l = r; r = tmp;
    }

    if (type == TYPE_COUNT) {
      scanf("%d", &k);
      k = transform(k);
      process_count_op(l, r, k);
    } else {
      process_rotate_op(l, r);
    }
  }
}

int main() {
  read_data();
  process_ops();
```

```
  return 0;
}
```

Sursă cu descompunere după poziții (versiune online).

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

const int MAX_BUCKETS = 160;
const int MAX_BUCKET_SIZE = 634;
const int MAX_N = MAX_BUCKETS * MAX_BUCKET_SIZE;
const int TYPE_ROTATE = 1;
const int TYPE_COUNT = 2;

typedef struct {
  short freq[MAX_N];
  int circ[MAX_BUCKET_SIZE];
  int start;
} bucket;

bucket buck[MAX_BUCKETS];
int modulo[2 * MAX_BUCKET_SIZE];
int bs; // bucket size
int n, numOps;
int lastAnswer;

void initBuckets() {
  bs = 2 * sqrt(n);

  for (int i = 0; i < 2 * bs; i++) {
    modulo[i] = i % bs;
  }
}

void bucketSetInitial(int pos, int val) {
  buck[pos / bs].circ[pos % bs] = val;
  buck[pos / bs].freq[val]++;
}

void readInputData() {
  scanf("%d", &n);
  initBuckets();

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    bucketSetInitial(i, x);
  }
```

```
  scanf("%d", &numOps);
}

int realPos(int b, int pos) {
  return modulo[pos + buck[b].start];
}

int next(int pos) {
  return modulo[pos + 1];
}

int prev(int pos) {
  return modulo[pos + bs - 1];
}

void bucketSet(int b, int pos, int val) {
  bucket& g = buck[b];
  int realP = realPos(b, pos);
  int oldVal = g.circ[realP];
  g.circ[realP] = val;
  g.freq[oldVal]--;
  g.freq[val]++;
}

int partialCount(int b, int l, int r, int k) {
  int realL = realPos(b, l);
  int realR = realPos(b, r);

  // Numără-l separat pe realR ca să îl putem folosi ca terminator de buclă.
  int result = (buck[b].circ[realR] == k);

  while (realL != realR) {
    result += (buck[b].circ[realL] == k);
    realL = next(realL);
  }
  return result;
}

int bucketCount(int l, int r, int k) {
  int result = 0;
  while (l < r) {
    result += buck[l++].freq[k];
  }
  return result;
}

void processCountOp(int l, int r, int k) {
  int bl = l / bs, br = r / bs;
  int lpos = l - bl * bs, rpos = r - br * bs;
```

```
  if (bl == br) {
    lastAnswer = partialCount(bl, lpos, rpos, k);
  } else {
    lastAnswer =
      partialCount(bl, lpos, bs - 1, k) +
      partialCount(br, 0, rpos, k) +
      bucketCount(bl + 1, br, k);
  }

  printf("%d\n", lastAnswer);
}

int partialRotate(int b, int l, int r) {
  int *v = buck[b].circ;
  int realL = realPos(b, l);
  int realR = realPos(b, r);
  int prevR = prev(realR);
  int save = v[realR];

  while (realR != realL) {
    v[realR] = v[prevR];
    realR = prevR;
    prevR = prev(realR);
  }

  v[realL] = save;
  return save;
}

int bucketRotate(int b) {
  bucket& g = buck[b];
  g.start = prev(g.start);
  return g.circ[g.start];
}

void processRotateOp(int l, int r) {
  int bl = l / bs, br = r / bs;
  int lpos = l - bl * bs, rpos = r - br * bs;
  if (bl == br) {
    partialRotate(bl, lpos, rpos);
  } else {
    int fromLeft = partialRotate(bl, lpos, bs - 1);
    for (int b = bl + 1; b < br; b++) {
      int toRight = bucketRotate(b);
      bucketSet(b, 0, fromLeft);
      fromLeft = toRight;
    }
    int toRight = partialRotate(br, 0, rpos);
    bucketSet(br, 0, fromLeft);
    bucketSet(bl, lpos, toRight);
```

```
  }
}

int transform(int x) {
  return lastAnswer
    ? (x + lastAnswer - 1) % n + 1
    : x;
}

void processOps() {
  while (numOps--) {
    int type, l, r, k;
    scanf("%d %d %d", &type, &l, &r);
    l = transform(l) - 1;
    r = transform(r) - 1;
    if (l > r) {
      int tmp = l; l = r; r = tmp;
    }

    if (type == TYPE_COUNT) {
      scanf("%d", &k);
      k = transform(k);
      processCountOp(l, r, k);
    } else {
      processRotateOp(l, r);
    }
  }
}

int main() {
  readInputData();
  processOps();

  return 0;
}
```

Sursă cu descompunere după operații (versiune online).

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

const int MAX_BUCKETS = 160;
const int MAX_BUCKET_SIZE = 634;
const int OVERFLOW_FACTOR = 2;
const int MAX_N = MAX_BUCKETS * MAX_BUCKET_SIZE;
const int TYPE_ROTATE = 1;
const int TYPE_COUNT = 2;

struct bucket {
```

```
short freq[MAX_N];
int data[OVERFLOW_FACTOR * MAX_BUCKET_SIZE];
int size;

void append(int x) {
  data[size++] = x;
  freq[x]++;
}

// Returnează numărul de elemente vărsate. Golește data, size și freq.
int empty(int* dest) {
  for (int i = 0; i < size; i++) {
    dest[i] = data[i];
    freq[data[i]]--;
  }
  int old_size = size;
  size = 0;

  return old_size;
}

int partial_count(int l, int r, int val) {
  int cnt = 0;
  while (l <= r) {
    cnt += (data[l++] == val);
  }

  return cnt;
}

int prefix_count(int pos, int val) {
  return partial_count(0, pos, val);
}

int suffix_count(int pos, int val) {
  return partial_count(pos, size - 1, val);
}

void insert(int pos, int val) {
  freq[val]++;
  size++;
  for (int i = size - 1; i > pos; i--) {
    data[i] = data[i - 1];
  }
  data[pos] = val;
}

int remove(int pos) {
  int result = data[pos];
  freq[result]--;
```

```
    for (int i = pos; i < size - 1; i++) {
      data[i] = data[i + 1];
    }
    size--;

    return result;
  }
};

bucket buck[MAX_BUCKETS];
int naive[MAX_N];
int bs, nb; // mărimea blocului, numărul de blocuri
int n;
int last_answer;

// Stochează informații despre blocul în care se află un indice real.
struct bucket_info {
  int b;      // numărul blocului
  int start;  // indicele primului element din bloc
  int offset; // offset-ul indicelui dat față de start

  void find_next_bucket(int index) {
    while (start + buck[b].size <= index) {
      start += buck[b++].size;
    }
    offset = index - start;
  }
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

void distribute_buckets() {
  // Evită O(n) împărțiri deoarece vom rula acest cod de O(sqrt(n)) ori.
  for (int b = 0; b < nb; b++) {
    int start = b * bs;
    int end = min(start + bs, n);
    for (int i = start; i < end; i++) {
      buck[b].append(naive[i]);
    }
  }
}

void collect_buckets() {
  int ptr = 0;
  for (int i = 0; i < nb; i++) {
    ptr += buck[i].empty(&naive[ptr]);
  }
}
```

```
void read_data() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    scanf("%d", &naive[i]);
  }
}

void init_buckets() {
  bs = 2 * sqrt(n);
  nb = (n - 1) / bs + 1;
  distribute_buckets();
}

int cross_bucket_count(int l, int r, int k) {
  int result = 0;
  while (l < r) {
    result += buck[l++].freq[k];
  }
  return result;
}

// [l, r] inclusiv
void process_count_op(int l, int r, int k) {
  bucket_info bl = { 0, 0, 0 };
  bl.find_next_bucket(l);
  bucket_info br = bl;
  br.find_next_bucket(r);

  if (bl.b == br.b) {
    last_answer = buck[bl.b].partial_count(bl.offset, br.offset, k);
  } else {
    last_answer =
      buck[bl.b].suffix_count(bl.offset, k) +
      buck[br.b].prefix_count(br.offset, k) +
      cross_bucket_count(bl.b + 1, br.b, k);
  }

  printf("%d\n", last_answer);
}

void process_rotate_op(int l, int r) {
  bucket_info bl = { 0, 0, 0 };
  bl.find_next_bucket(l);
  bucket_info br = bl;
  br.find_next_bucket(r);

  int x = buck[br.b].remove(br.offset);
  buck[bl.b].insert(bl.offset, x);
```

```
  if (buck[bl.b].size == OVERFLOW_FACTOR * bs) {
    collect_buckets();
    distribute_buckets();
  }
}

int transform(int x) {
  return last_answer
    ? (x + last_answer - 1) % n + 1
    : x;
}

void process_ops() {
  int num_ops;
  scanf("%d", &num_ops);

  while (num_ops--) {
    int type, l, r, k;
    scanf("%d %d %d", &type, &l, &r);
    l = transform(l) - 1;
    r = transform(r) - 1;
    if (l > r) {
      int tmp = l; l = r; r = tmp;
    }

    if (type == TYPE_COUNT) {
      scanf("%d", &k);
      k = transform(k);
      process_count_op(l, r, k);
    } else {
      process_rotate_op(l, r);
    }
  }
}

int main() {
  read_data();
  init_buckets();
  process_ops();

  return 0;
}
```

D.6 Problema Time to Raid Cowavans (Codeforces)

‹ înapoi • versiune online

```cpp
// Complexitate: O((q + n) sqrt n).
#include <algorithm>
#include <stdio.h>

const int MAX_N = 300'000;
const int MAX_Q = 300'000;
const int PREPROCESS_LIMIT = 547;

struct query {
  int id, first, step;
};

int w[MAX_N + 1];
long long prep[MAX_N + 1];
query q[MAX_Q];
long long answer[MAX_Q];
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &w[i]);
  }
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].first, &q[i].step);
    q[i].id = i;
  }
}

void sort_queries() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return a.step < b.step;
  });
}

long long naive_prog_sum(int first, int step) {
  long long sum = 0;
  for (int i = first; i <= n; i += step) {
    sum += w[i];
  }
  return sum;
}

void preprocess(int step) {
  for (int i = n; (i > n - step) && (i >= 1); i--) {
    prep[i] = w[i];
  }
  for (int i = n - step; i >= 1; i--) {
```

```
    prep[i] = w[i] + prep[i + step];
  }
}

void process_queries() {
  sort_queries();
  for (int i = 0; i < num_queries; i++) {
    if (q[i].step > PREPROCESS_LIMIT) {
      answer[q[i].id] = naive_prog_sum(q[i].first, q[i].step);
    } else {
      if (!i || (q[i].step != q[i - 1].step)) {
        preprocess(q[i].step);
      }
      answer[q[i].id] = prep[q[i].first];
    }
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%lld\n", answer[i]);
  }
}

int main() {
  read_data();
  process_queries();
  write_answers();

  return 0;
}
```

D.7 Problema Sandor (Baraj ONI 2025)

‹ înapoi • versiune online

```
// Complexitate O(N sqrt N), provenită din 3 for-uri imbricate a cîte
// O(sqrt N) iterații.
#include <stdio.h>

const int MAX_N = 400'000;
const int MAX_WEIGHT = 800'000;

struct run {
  int val, cnt;
};

run r[MAX_N];
```

```cpp
// jump[w] este obiectul maxim care nu depășește greutatea w. Mai exact,
// jump[w] este cel mai mic i a.î r[i].val ≤ w
int jump[MAX_WEIGHT + 1];
long long sol[MAX_WEIGHT + 1];
int task, n, num_runs, capacity;

int min(int x, int y) {
  return (x < y) ? x : y;
}

void read_data() {
  FILE* f = fopen("sandor.in", "r");
  int x;

  fscanf(f, "%d %d %d", &task, &n, &capacity);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &x);
    if (num_runs && (x == r[num_runs - 1].val)) {
      r[num_runs - 1].cnt++;
    } else {
      r[num_runs++] = { x, 1 };
    }
  }

  fclose(f);
}

void compute_jumps() {
  int i = 0;
  for (int w = MAX_WEIGHT; w >= 0; w--) {
    while ((i < num_runs) && (r[i].val > w)) {
      i++;
    }
    jump[w] = i;
  }
}

// Rulează algoritmul lui Sandor pentru capacitatea dată. Returnează greutatea
// acumulată.
int do_the_sandor(int start, int capacity) {
  int c = capacity;
  int i = start;

  while (i < num_runs) {
    int taken = min(c / r[i].val, r[i].cnt);
    c -= taken * r[i].val;
    // Dacă din c = 100 am luat 2 * r[i].val = 20, fiindcă atîtea erau, nu are
    // sens să continui de la 80. Următorul număr poate fi doar 9 sau mai mic.
    i = jump[min(c, r[i].val - 1)];
  }
```

```
  return capacity - c;
}

// Presupunînd că am tăiat deja două obiecte înainte de start și am obținut
// greutatea weight_so_far, rulează Sandor simplu pe restul vectorului și
// adaugă rezultatul la soluție.
void cut_0(int start, int weight_so_far, long long multiplier) {
  int s = do_the_sandor(start, capacity - weight_so_far);
  sol[weight_so_far + s] += multiplier;
}

// Presupunînd că am tăiat deja un obiect înainte de start și am obținut
// greutatea weight_so_far, și că de la poziția start începînd mai am num_num
// obiecte, încearcă să elimini cîte un obiect în toate modurile posibile.
//
// Observăm că, dacă tăiem alt obiect decît cele pe care le-ar pune Sandor în
// rucsac, vom obține aceeași sumă ca și pe vectorul nemodificat.
void cut_1(int start, int weight_so_far, int num_num, long long multiplier) {
  int c = capacity - weight_so_far;
  int i = start;

  while (i < num_runs) {
    int taken = min(c / r[i].val, r[i].cnt);
    if (taken == r[i].cnt) {
      num_num -= taken;
      int all_but_one = (taken - 1) * r[i].val;
      cut_0(i + 1, weight_so_far + all_but_one, multiplier * taken);
    }
    weight_so_far += taken * r[i].val;
    c -= taken * r[i].val;
    i = jump[min(c, r[i].val - 1)];
  }

  // Toate numerele pe care nu le-am tăiat explicit merg pe soluția originală,
  // care duce la sumă weight_so_far.
  sol[weight_so_far] += multiplier * num_num;
}

// Pentru bucla exterioară este important să obținem tot complexitate
// O(sqrt N). De aceea, considerăm simultan fiecare grupă de obiecte
// consecutive NEincluse în rucsac.
void cut_2() {
  long long multiplier = 0;
  int weight_so_far = 0;
  int c = capacity;
  int num_right = n;

  for (int i = 0; i < num_runs; i++) {
    if (r[i].val > c) {
```

```
      multiplier += r[i].cnt;
    } else {
      int cnt = r[i].cnt;
      // Taie două obiecte dinainte de i. Rămîn 0 de tăiat.
      cut_0(i, weight_so_far, multiplier * (multiplier - 1) / 2);

      // Taie un obiect dinainte de i. Rămîne unul de tăiat.
      cut_1(i, weight_so_far, num_right, multiplier);

      // Taie două obiecte din grupa i. În rest, pune în rucsac ce se poate.
      if (r[i].cnt >= 2) {
        int taken = min(c / r[i].val, r[i].cnt - 2);
        cut_0(i + 1, weight_so_far + taken * r[i].val, cnt * (cnt - 1) / 2);
      }

      // Taie un obiect din grupa i. În rest, pune în rucsac ce se poate.
      int taken = min(c / r[i].val, r[i].cnt - 1);
      cut_1(i + 1, weight_so_far + taken * r[i].val, num_right - r[i].cnt, cnt);

      // Nu tăia nimic, bagă în rucsac.
      taken = min(c / r[i].val, r[i].cnt);
      c -= taken * r[i].val;
      weight_so_far += taken * r[i].val;

      multiplier = 0;
    }

    num_right -= r[i].cnt;
  }

  // Nu mai putem băga nimic în rucsac, dar din ultimele @multiplier obiecte
  // trebuie să tăiem două.
  sol[weight_so_far] += multiplier * (multiplier - 1) / 2;
}

void write_solution() {
  FILE* f = fopen("sandor.out", "w");
  for (int i = 0; i <= capacity; i++) {
    fprintf(f, "%lld ", sol[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  compute_jumps();

  if (task == 1) {
    cut_1(0, 0, n, 1);
```

```
  } else {
    cut_2();
  }

  write_solution();

  return 0;
}
```

D.8 Problema Puzzle-bila (Lot 2025)

‹ înapoi • versiune online

```
// Complexitate: O(n·m·sqrt(m) + n·m·log(m)).
#include <stdio.h>

const int MAX_COLS = 50'000;
const int MAX_LOG = 16;
const int MAX_DISTINCT_LENGTHS = 320;
const int INFTY = 2'000'000;
const int NONE = -1;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int log2(int x) {
  return 31 - __builtin_clz(x);
}

struct sparse_table {
  int v[MAX_LOG][MAX_COLS];
  int n;

  void build(int* src, int n, bool with_shifts) {
    for (int i = 0; i < n; i++) {
      v[0][i] = with_shifts ? (src[i] - i) : src[i];
    }
    for (int p = 1; (1 << p) <= n; p++) {
      for (int i = 0; i <= n - (1 << p); i++) {
        v[p][i] = min(v[p - 1][i], v[p - 1][i + (1 << (p - 1))]);
      }
    }
  }

  int range_min(int l, int r) {
    l = (l < 0) ? 0 : l;
    if (l > r) {
```

```
      return INFTY;
    }
    int row = log2(r - l + 1);
    return  min(v[row][l], v[row][r - (1 << row) + 1]);
  }
};

sparse_table st, st_shift;
bool row[MAX_COLS];
int prev1[MAX_COLS];
int distinct_len[MAX_DISTINCT_LENGTHS], num_lengths;
int dp[MAX_COLS];
int end[MAX_COLS + 1]; // coloana pe care se termină ultima fereastră de lățime l
int num_rows, num_cols;

void read_size() {
  scanf("%d %d ", &num_rows, &num_cols);
}

void read_row() {
  for (int c = 0; c < num_cols; c++) {
    row[c] = getchar() - '0';
  }
  getchar();
}

void init_all() {
  dp[0] = 0;
  for (int c = 1; c < num_cols; c++) {
    dp[c] = INFTY;
  }

  for (int l = 0; l <= num_cols; l++) {
    end[l] = NONE;
  }
}

void compute_prev1() {
  prev1[0] = row[0] ? 0 : -1;
  for (int c = 1; c < num_cols; c++) {
    prev1[c] = row[c] ? c : prev1[c - 1];
  }
}

void add_length(int len, int end_col) {
  if (len && (end[len] == NONE)) {
    distinct_len[num_lengths++] = len;
  }
  end[len] = end_col;
}
```

```
void reset_distinct_lengths() {
  while (num_lengths) {
    end[distinct_len[--num_lengths]] = NONE;
  }
}

// Adu ferestre de zerouri din stînga, deja închise, aliniindu-le cu col.
void slide_windows_right(int col) {
  dp[col] = INFTY;
  for (int i = 0; i < num_lengths; i++) {
    int len = distinct_len[i];
    int cost = st.range_min(col - len + 1, col) + (col - end[len]);
    dp[col] = min(dp[col], cost);
  }
}

// Adu ferestre de zerouri din dreapta, deja închise, aliniindu-le cu col.
void slide_windows_left(int col) {
  for (int i = 0; i < num_lengths; i++) {
    int len = distinct_len[i];
    int cost = st_shift.range_min(col - len + 1, col) + end[len] - len + 1;
    dp[col] = min(dp[col], cost);
  }
}

void slide_current_window(int col, int r_len) {
  if (!row[col]) {
    int l = 1 + prev1[col];
    int r = col + r_len - 1;
    int len = r - l + 1;
    dp[col] = min(dp[col], st.range_min(l, col));
    int cost_r = st_shift.range_min(col - len + 1, l - 1) + l;
    dp[col] = min(dp[col], cost_r);
  }
}

void scan_left_to_right() {
  reset_distinct_lengths();

  int cur_len = 0;
  for (int c = 0; c < num_cols; c++) {
    if (row[c]) {
      add_length(cur_len, c - 1);
      cur_len = 0;
    } else {
      cur_len++;
    }
    slide_windows_right(c);
  }
```

```
}

void scan_right_to_left() {
  reset_distinct_lengths();

  int cur_len = 0;
  for (int c = num_cols - 1; c >= 0; c--) {
    if (row[c]) {
      add_length(cur_len, c + cur_len);
      cur_len = 0;
    } else {
      cur_len++;
    }
    slide_windows_left(c);
    slide_current_window(c, cur_len);
  }
}

void process_rows() {
  init_all();
  for (int r = 0; r < num_rows; r++) {
    read_row();
    compute_prev1();
    st.build(dp, num_cols, false);
    st_shift.build(dp, num_cols, true);
    scan_left_to_right();
    scan_right_to_left();
  }
}

void write_result() {
  int res = dp[num_cols - 1];
  printf("%d\n", (res == INFTY) ? NONE : res);
}

int main() {
  read_size();
  process_rows();
  write_result();

  return 0;
}
```

Anexa E

Algoritmul lui Mo

E.1 Problema Powerful Array (Codeforces)

‹ înapoi • versiune online

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

const int MAX_N = 200'000;
const int BLOCK_SIZE = 300;
const int MAX_QUERIES = 200'000;
const int MAX_VAL = 1'000'000;

typedef unsigned long long u64;

struct query {
  int l, r;
  int orig_index;
};

int v[MAX_N];
int f[MAX_VAL + 1];
u64 power;
query q[MAX_QUERIES];
u64 answer[MAX_QUERIES]; // separat de q[] ca să evităm o sortare la final
int n, num_queries;

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int i = 0; i < n; i++) {
    scanf("%d", &v[i]);
  }
  for (int i = 0; i < num_queries; i++) {
    int l, r;
    scanf("%d %d", &l, &r);
```

```
    q[i].l = l - 1;
    q[i].r = r - 1;
    q[i].orig_index = i;
  }
}

void sort_queries_by_left_block() {
  std::sort(q, q + num_queries, [](query a, query b) {
    int x = a.l / BLOCK_SIZE, y = b.l / BLOCK_SIZE;
    if (x != y) {
      return (x < y);
    } else if (x % 2) {
      return a.r > b.r;
    } else {
      return a.r < b.r;
    }
  });
}

void add(int pos) {
  f[v[pos]]++;
  power += (u64)v[pos] * (2 * f[v[pos]] - 1);
}

void remove(int pos) {
  f[v[pos]]--;
  power -= (u64)v[pos] * (2 * f[v[pos]] + 1);
}

void mo() {
  int l = 0, r = -1; // vid
  for (int i = 0; i < num_queries; i++) {
    while (l > q[i].l) {
      add(--l);
    }
    while (r < q[i].r) {
      add(++r);
    }
    while (l < q[i].l) {
      remove(l++);
    }
    while (r > q[i].r) {
      remove(r--);
    }
    answer[q[i].orig_index] = power;
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
```

```
    printf("%llu\n", answer[i]);
  }
}

int main() {
  read_data();
  sort_queries_by_left_block();
  mo();

  write_answers();

  return 0;
}
```

E.2 Problema Most Frequent Value (SPOJ)

înapoi

```
// Complexitate: O((n + q) sqrt n).
#include <algorithm>
#include <stdio.h>

const int MAX_N = 100'000;
const int BLOCK_SIZE = 316;
const int MAX_QUERIES = 100'000;
const int MAX_VAL = 100'000;

struct query {
  int l, r;
  int orig_index;
};

int v[MAX_N];
int f[MAX_VAL + 1];
int num_having_f[MAX_N + 1], max_f;
query q[MAX_QUERIES];
int answer[MAX_QUERIES];
int n, num_queries;

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int i = 0; i < n; i++) {
    scanf("%d", &v[i]);
  }
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].l, &q[i].r);
    q[i].orig_index = i;
  }
```

```
}

void sort_queries_by_left_block() {
  std::sort(q, q + num_queries, [](query a, query b) {
    int x = a.l / BLOCK_SIZE, y = b.l / BLOCK_SIZE;
    if (x != y) {
      return (x < y);
    } else if (x % 2) {
      return a.r > b.r;
    } else {
      return a.r < b.r;
    }
  });
}

void add(int pos) {
  int x = ++f[v[pos]];
  num_having_f[x - 1]--;
  num_having_f[x]++;
  if (x > max_f) {
    max_f = f[v[pos]];
  }
}

void remove(int pos) {
  int x = --f[v[pos]];
  num_having_f[x + 1]--;
  num_having_f[x]++;
  if (num_having_f[max_f] == 0) {
    max_f--;
  }
}

void mo() {
  num_having_f[0] = n;
  int l = 0, r = -1; // vid
  for (int i = 0; i < num_queries; i++) {
    while (l > q[i].l) {
      add(--l);
    }
    while (r < q[i].r) {
      add(++r);
    }
    while (l < q[i].l) {
      remove(l++);
    }
    while (r > q[i].r) {
      remove(r--);
    }
    answer[q[i].orig_index] = max_f;
```

```
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_data();
  sort_queries_by_left_block();
  mo();

  write_answers();

  return 0;
}
```

E.3 Problema RangeMode (Infoarena Cup 2013)

‹ înapoi • versiune online

```
// Complexitate: O((n + q) sqrt n).
#include <algorithm>
#include <stdio.h>

const int MAX_N = 100000;
const int BLOCK_SIZE = 316;
const int MAX_QUERIES = 100000;
const int MAX_VAL = 100000;

struct query {
  int l, r;
  int orig_index;
};

int v[MAX_N];
int f[MAX_VAL + 1]; // frecvențele elementelor
int right_ptr;      // poziția ultimului element adăugat în f
int right_best;     // răspunsul considerînd strict dreapta blocului curent
query q[MAX_QUERIES];
int answer[MAX_QUERIES];
int n, num_queries;

void read_data() {
  FILE* f = fopen("rangemode.in", "r");
  fscanf(f, "%d %d", &n, &num_queries);
```

```
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &v[i]);
  }
  for (int i = 0; i < num_queries; i++) {
    int l, r;
    fscanf(f, "%d %d", &l, &r);
    q[i].l = l - 1;
    q[i].r = r - 1;
    q[i].orig_index = i;
  }
  fclose(f);
}

void sort_queries_by_left_block() {
  std::sort(q, q + num_queries, [](query a, query b) {
    int x = a.l / BLOCK_SIZE, y = b.l / BLOCK_SIZE;
    return (x < y) || ((x == y) && (a.r < b.r));
  });
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

void clear_f() {
  for (int i = 0; i <= MAX_VAL; i++) {
    f[i] = 0;
  }
}

void optimize(int& best, int candidate) {
  if ((f[candidate] > f[best]) ||
      ((f[candidate] == f[best]) && (candidate < best))) {
    best = candidate;
  }
}

void extend_right(int until) {
  while (right_ptr < until) {
    int val = v[++right_ptr];
    f[val]++;
    optimize(right_best, val);
  }
}

int add_current_block(int l, int r) {
  int best = right_best;
  for (int i = l; i <= r; i++) {
    f[v[i]]++;
    optimize(best, v[i]);
```

```
  }
  return best;
}

void remove_current_block(int l, int r) {
  for (int i = l; i <= r; i++) {
    f[v[i]]--;
  }
}

void process_query(query q, int last_element_in_block) {
  extend_right(q.r);

  // Interogarea nu trece neapărat în blocul următor. Poate fi complet
  // cuprinsă în blocul curent.
  int right = min(q.r,last_element_in_block);
  int best = add_current_block(q.l, right);
  answer[q.orig_index] = best;
  remove_current_block(q.l, right);
}

void scan_blocks() {
  int qi = 0;
  for (int start = 0; start < n; start += BLOCK_SIZE) {
    int end = start + BLOCK_SIZE;
    right_ptr = end - 1;
    right_best = 0;
    while ((qi < num_queries) && (q[qi].l < end)) {
      process_query(q[qi++], end - 1);
    }
    clear_f();
  }
}

void write_answers() {
  FILE* f = fopen("rangemode.out", "w");
  for (int i = 0; i < num_queries; i++) {
    fprintf(f, "%d\n", answer[i]);
  }
  fclose(f);
}

int main() {
  read_data();
  sort_queries_by_left_block();
  scan_blocks();

  write_answers();

  return 0;
```

```
}
```

E.4 Problema Machine Learning (Codeforces)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>
#include <unordered_map>

const int MAX_N = 100'000;
const int BLOCK_SIZE = 2'154; // n^{2/3}
const int MAX_OPS = 100'000;

const int T_QUERY = 1;
const int T_UPDATE = 2;

struct query {
  int l, r, time, answer;
};

struct update {
  int pos, old_val, val, time;
};

int a[MAX_N];
query q[MAX_OPS];
update u[MAX_OPS];
int tmp[MAX_N + MAX_OPS];
int n, num_ops, num_queries, num_updates;

// Date specifice pentru Mo.
int f[MAX_N + MAX_OPS];
int num_having_f[MAX_N + MAX_OPS];

// Renumerotează valorile din vector și actualizările.
struct normalizer {
  std::unordered_map<int, int> map;

  int normalize(int x) {
    auto it = map.find(x);
    if (it == map.end()) {
      int result = 1 + map.size();
      map[x] = result;
      return result;
    } else {
      return it->second;
    }
```

```
  }
};

normalizer norm;

void read_array() {
  scanf("%d %d", &n, &num_ops);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    a[i] = norm.normalize(x);
  }
}

void read_ops() {
  for (int i = 0; i < n; i++) {
    tmp[i] = a[i];
  }

  u[num_updates++] = { 0, 0, 0, 0 }; // santinelă stînga (t = 0)
  for (int t = 1; t <= num_ops; t++) {
    int type;
    scanf("%d", &type);
    if (type == T_QUERY) {
      int l, r;
      scanf("%d %d", &l, &r);
      q[num_queries++] = { l - 1, r - 1, t };
    } else {
      int pos, val;
      scanf("%d %d", &pos, &val);
      pos--;
      val = norm.normalize(val);
      u[num_updates++] = { pos, tmp[pos], val, t };
      tmp[pos] = val;
    }
  }
  u[num_updates++] = { 0, 0, 0, n + num_ops }; // santinelă dreapta (t = ∞)
}

void sort_queries() {
  std::sort(q, q + num_queries, [](query a, query b) {
    int x = a.l / BLOCK_SIZE, y = b.l / BLOCK_SIZE;
    if (x != y) {
      return (x < y);
    }

    x = a.r / BLOCK_SIZE, y = b.r / BLOCK_SIZE;
    if (x != y) {
      return (x < y);
    }
```

```
    return a.time < b.time;
  });
}

void change_frequency(int val, int delta) {
  num_having_f[f[val]]--;
  f[val] += delta;
  num_having_f[f[val]]++;
}

void add(int pos) {
  change_frequency(a[pos], +1);
}

void remove(int pos) {
  change_frequency(a[pos], -1);
}

void change_value(int l, int r, int pos, int val) {
  if ((pos >= l) && (pos <= r)) {
    change_frequency(a[pos], -1);
    change_frequency(val, +1);
  }
  a[pos] = val;
}

int get_mex() {
  int mex = 1;
  while (num_having_f[mex]) {
    mex++;
  }
  return mex;
}

void mo() {
  num_having_f[0] = n + num_updates; // ceva infinit
  int l = 0, r = -1, u_index = 1;
  for (int i = 0; i < num_queries; i++) {
    while (l > q[i].l) {
      add(--l);
    }
    while (r < q[i].r) {
      add(++r);
    }
    while (l < q[i].l) {
      remove(l++);
    }
    while (r > q[i].r) {
      remove(r--);
```

```
    }
    while (u[u_index].time < q[i].time) {
      change_value(l, r, u[u_index].pos, u[u_index].val);
      u_index++;
    }
    while (u[u_index - 1].time > q[i].time) {
      u_index--;
      change_value(l, r, u[u_index].pos, u[u_index].old_val);
    }
    q[i].answer = get_mex();
  }
}

void sort_queries_by_time() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return a.time < b.time;
  });
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", q[i].answer);
  }
}

int main() {
  read_array();
  read_ops();
  sort_queries();
  mo();
  sort_queries_by_time();
  write_answers();

  return 0;
}
```

Anexa F

Arbori - probleme esențiale

F.1 Generator simplu de arbori aleatorii

înapoi

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int n, k;

void usage() {
  fprintf(stderr, "Apel: ./generator <n> <k>\n");
  fprintf(stderr, "\n");
  fprintf(stderr, "Generează un arbore cu n noduri și k căi.\n");
  exit(1);
}

void parse_command_line_args(int argc, char** argv) {
  if (argc != 3) {
    usage();
  }

  n = atoi(argv[1]);
  k = atoi(argv[2]);
}

void init_rng() {
  struct timeval time;
  gettimeofday(&time, NULL);
  srand(time.tv_usec);
}

int rand2(int min, int max) {
  return min + rand() % (max - min + 1);
```

```
}

int main(int argc, char** argv) {
  parse_command_line_args(argc, argv);
  init_rng();

  printf("%d %d\n", n, k);
  for (int i = 2; i <= n; i++) {
    printf("%d %d\n", i, rand2(1, i - 1));
  }
  for (int i = 0; i < k; i++) {
    printf("%d %d\n", rand2(1, n), rand2(1, n));
  }

  return 0;
}
```

F.2 Generator avansat de arbori aleatorii

înapoi

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

#define MAX_N 1'000'000
#define NIL -1

int v[MAX_N], p[MAX_N];

void usage() {
  fprintf(stderr, "Apel: gen <n> <chain_length> <dense_node> <op_type> [...]\n");
  fprintf(stderr, "  op_type = 0:\n");
  fprintf(stderr, "    fără valori sau actualizări\n");
  fprintf(stderr, "  op_type = 1 <num_ops> <max_value>:\n");
  fprintf(stderr, "    valori inițiale, actualizări pe nod, interogări pe nod\n");
  fprintf(stderr, "  op_type = 2 <num_ops>:\n");
  fprintf(stderr, "    fără valori sau actualizări, cu interogări pe cale\n");
  _exit(1);
}

void shuffle(int* v, int n) {
  for (int i = 0; i < n; i++) {
    int j = rand() % (i + 1);
    int tmp = v[i];
    v[i] = v[j];
```

```
    v[j] = tmp;
  }
}

void generate_tree(int n, int chain_length, int dense_node) {
  for (int i = 0; i < n; i++) {
    v[i] = i;
    p[i] = NIL;
  }

  shuffle(v, n);

  // Înlănțuie nodurile [0, chain_length).
  for (int i = 1; i < chain_length; i++) {
    p[v[i]] = v[i - 1];
  }

  // Conectează nodurile [chain_length, n - dense_node) la noduri aleatorii
  // dinaintea lor.
  for (int i = chain_length; i < n - dense_node; i++) {
    p[v[i]] = v[rand() % i];
  }

  // Conectează nodurile [n - dense_node, n) de un același părinte.
  int parent = rand() % (n - dense_node);
  for (int i = n - dense_node; i < n; i++) {
    p[v[i]] = v[parent];
  }

  // Nu tipări muchiile chiar în ordinea asta.
  shuffle(v, n);

  printf("%d\n", n);
  for (int i = 0; i < n; i++) {
    int x = v[i], y = p[v[i]];
    if (y != NIL) {
      // Interschimbă aleatoriu fiul și părintele.
      if (rand() % 2) {
        int tmp = x;
        x = y;
        y = tmp;
      }
      printf("%d %d\n", 1 + x, 1 + y);
    }
  }
}

void generate_ops(int n, int num_ops, int max_value) {
  // Valorile inițiale.
  for (int i = 0; i < n; i++) {
```

```
    v[i] = rand() % (max_value + 1);
    printf("%d ", v[i]);
  }
  printf("\n");

  printf("%d\n", num_ops);
  while (num_ops--) {
    int u = rand() % n;
    if (rand() % 2) {
      // actualizare -- asigură-te că nu depășim max_value
      int delta = rand() % (max_value + 1) - v[u];
      v[u] += delta;
      printf("1 %d %d\n", 1 + u, delta);
    } else {
      // interogare
      printf("2 %d\n", 1 + u);
    }
  }
}

void generate_lca_queries(int n, int num_ops) {
  // Generează perechi de noduri distincte.
  printf("%d\n", num_ops);
  while (num_ops--) {
    int u = rand() % n, v;
    do {
      v = rand() % n;
    } while (u == v);
    printf("%d %d\n", 1 + u, 1 + v);
  }
}

int main(int argc, char** argv) {
  if (argc < 5) {
    usage();
  }

  int n = atoi(argv[1]);
  int chain_length = atoi(argv[2]);
  int dense_node = atoi(argv[3]);
  int op_type = atoi(argv[4]);

  assert(chain_length + dense_node <= n);

  struct timeval time;
  gettimeofday(&time, NULL);
  srand(time.tv_usec);

  generate_tree(n, chain_length, dense_node);
```

```
  int num_ops, max_value;
  switch (op_type) {
    case 0:
      // Nimic altceva de făcut.
      break;

    case 1:
      if (argc != 7) {
        usage();
      }
      num_ops = atoi(argv[5]);
      max_value = atoi(argv[6]);
      generate_ops(n, num_ops, max_value);
      break;

    case 2:
      if (argc != 6) {
        usage();
      }
      num_ops = atoi(argv[5]);
      generate_lca_queries(n, num_ops);
      break;

    default:
      assert(false);
  }

  return 0;
}
```

F.3 Problema Subordinates (CSES)

‹ înapoi

Sursă cu vectori STL.

```
#include <iostream>
#include <vector>

const int MAX_N = 200'000;

int s[MAX_N + 1]; // mărimea subarborelui, inclusiv nodul însuși
std::vector<int> c[MAX_N + 1]; // c[u] = fiii lui u
int n;

void read_data() {
  std::cin >> n;
  for (int u = 2; u <= n; u++) {
```

```
    int v;
    std::cin >> v;
    c[v].push_back(u);
  }
}

void depth_first_search(int u) {
  s[u] = 1;
  for (int v: c[u]) {
    depth_first_search(v);
    s[u] += s[v];
  }
}

void write_answers() {
  for (int u = 1; u <= n; u++) {
    std::cout << (s[u] - 1) << ' ';
  }
  std::cout << '\n';
}

int main() {
  read_data();
  depth_first_search(1);
  write_answers();

  return 0;
}
```

Sursă cu liste înlănțuite.

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int adj;   // începutul listei de adiacență
  int size;  // mărimea subarborelui, inclusiv nodul însuși
};

cell list[MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_child(int u, int v) {
```

```
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_data() {
  scanf("%d",  &n);
  for (int u = 2; u <= n; u++) {
    int par;
    scanf("%d", &par);
    add_child(par, u);
  }
}

void depth_first_search(int u) {
  nd[u].size = 1;
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    depth_first_search(v);
    nd[u].size += nd[v].size;
  }
}

void write_answers() {
  for (int u = 1; u <= n; u++) {
    printf("%d ", nd[u].size - 1);
  }
  printf("\n");
}

int main() {
  read_data();
  depth_first_search(1);
  write_answers();

  return 0;
}
```

F.4 Problema Tree Matching (CSES)

‹ înapoi

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
```

```
};

struct node {
  int adj;
  bool matched;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int max_match;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_data() {
  int n, u, v;

  scanf("%d",  &n);
  for (int i = 1; i < n; i++) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

void dfs(int u, int parent) {
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      dfs(v, u);
      if (!nd[u].matched && !nd[v].matched) {
        max_match++;
        nd[u].matched = nd[v].matched = true;
      }
    }
  }
}

void write_answer() {
  printf("%d\n", max_match);
}

int main() {
  read_data();
  dfs(1, 0);
  write_answer();
```

```
  return 0;
}
```

F.5 Problema Tree Diameter (CSES)

înapoi

Sursă cu două parcurgeri DFS.

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct rec {
  int node, dist;

  void improve(rec child) {
    if (child.dist + 1 > dist) {
      dist = child.dist + 1;
      node = child.node;
    }
  }
};

cell list[2 * MAX_NODES];
int adj[MAX_NODES + 1];

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, adj[u] };
  adj[u] = pos++;
}

void read_data() {
  int n, u, v;

  scanf("%d",  &n);
  for (int i = 1; i < n; i++) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}
```

```
rec dfs(int u, int parent) {
  rec result = { u, 0 }; // nodul însuși
  for (int ptr = adj[u]; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      result.improve(dfs(v, u));
    }
  }
  return result;
}

int main() {
  read_data();
  rec farthest = dfs(1, 0);
  rec farthest2 = dfs(farthest.node, 0);
  int diam = farthest2.dist;
  printf("%d\n", diam);

  return 0;
}
```

Sursă cu o singură parcurgere DFS.

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

cell list[2 * MAX_NODES];
int adj[MAX_NODES + 1];
int diam;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, adj[u] };
  adj[u] = pos++;
}

void read_data() {
  int n, u, v;

  scanf("%d",  &n);
  for (int i = 1; i < n; i++) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
```

```
  }
}

void maximize(int& x, int y) {
  x = (x > y) ? x : y;
}

// Returnează lungimea în muchii a căii celei mai lungi de la acest nod la
// orice frunză din subarbore.
int dfs(int u, int parent) {
  int dist = 0; // self
  for (int ptr = adj[u]; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      int l = 1 + dfs(v, u);
      maximize(diam, dist + l); // combină cu fiii anteriori
      maximize(dist, l);        // actualizează calea cea mai lungă
    }
  }
  return dist;
}

int main() {
  read_data();
  dfs(1, 0);
  printf("%d\n", diam);

  return 0;
}
```

F.6 Problema Tree Distances II (CSES)

‹ înapoi

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int adj;            // începutul listei de adiacență
  int size;           // mărimea subarborelui, inclusiv nodul însuși
  long long sum_dist; // suma distanțelor la toate celelalte noduri
};
```

```
cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_data() {
  scanf("%d",  &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

void size_dfs(int u, int parent, int depth) {
  nd[1].sum_dist += depth;
  nd[u].size = 1;
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      size_dfs(v, u, depth + 1);
      nd[u].size += nd[v].size;
    }
  }
}

void reroot_dfs(int u, int parent) {
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      nd[v].sum_dist = nd[u].sum_dist + n - 2 * nd[v].size;
      reroot_dfs(v, u);
    }
  }
}

void write_answers() {
  for (int u = 1; u <= n; u++) {
    printf("%lld ", nd[u].sum_dist);
  }
  printf("\n");
}

int main() {
```

```
  read_data();
  size_dfs(1, 0, 0);
  reroot_dfs(1, 0);
  write_answers();

  return 0;
}
```

F.7 Problema White-Black Balanced Subtrees (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 4'000;
const int WHITE = 0;
const int BLACK = 1;

struct node {
  short parent;
  short num_children; // fii care încă nu ne-au dat raportul
  short cnt[2];
  unsigned char color;
};

node nd[MAX_NODES + 1];
int n, answer;

void read_data() {
  scanf("%d", &n);
  for (int i = 2; i <= n; i++) {
    scanf("%hd ", &nd[i].parent);
    nd[nd[i].parent].num_children++;
  }

  for (int i = 1; i <= n; i++) {
    nd[i].color = (getchar() == 'B') ? BLACK : WHITE;
  }
}

// Toți fiii lui u i-au dat raportul. Este momentul ca u să își numere propria
// contribuție, apoi să i-o raporteze părintelui.
void process(node& u) {
  u.cnt[u.color]++; // numără-te pe tine însuți
  answer += (u.cnt[WHITE] == u.cnt[BLACK]);

  nd[u.parent].cnt[WHITE] += u.cnt[WHITE];
  nd[u.parent].cnt[BLACK] += u.cnt[BLACK];
```

```
}

void traverse_tree() {
  for (int u = 1; u <= n; u++) {
    int s = u;
    while (s && !nd[s].num_children) {
      process(nd[s]);
      s = nd[s].parent;
      nd[s].num_children--;
    }
  }
}

void reset() {
  for (int i = 1; i <= n; i++) {
    nd[i] = { 0, 0, 0, 0, 0 };
  }
  answer = 0;
}

void solve_test() {
  read_data();
  traverse_tree();
  printf("%d\n", answer);
  reset();
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    solve_test();
  }

  return 0;
}
```

F.8 Problema Blood Cousins (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(n + q).
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_QUERIES = 100'000;

struct cell {
```

```
  int v, next;
};

struct query_cell {
  int v, p, answer, next;
};

struct node {
  int adj;
  int qptr; // pointer în lista de interogări
};

cell list[MAX_NODES + 1];
query_cell q[MAX_QUERIES + 1];
node nd[MAX_NODES + 1];
int stack[MAX_NODES];
int n, num_queries;

void read_input_data() {
  int list_ptr = 1;

  scanf("%d", &n);

  // Pseudonodul 0 va avea toate rădăcinile drept fii.
  for (int u = 1; u <= n; u++) {
    int p;
    scanf("%d", &p);
    list[list_ptr] = { u, nd[p].adj };
    nd[p].adj = list_ptr++;
  }

  scanf("%d", &num_queries);
  for (int i = 1; i <= num_queries; i++) {
    scanf("%d %d", &q[i].v, &q[i].p);
  }
}

void distribute_queries() {
  for (int u = 1; u <= n; u++) {
    nd[u].qptr = 0;
  }
  for (int i = 1; i <= num_queries; i++) {
    if (q[i].v) { // poate fi 0 după primul DFS
      q[i].next = nd[q[i].v].qptr;
      nd[q[i].v].qptr = i;
    }
  }
}

void replace_query_nodes_dfs(int u, int depth) {
```

```
  stack[depth] = u;

  // Scanează toate interogările despre u și înlocuiește nodul cu al p-lea său
  // strămoș.
  for (int ptr = nd[u].qptr; ptr; ptr = q[ptr].next) {
    q[ptr].v = (depth > q[ptr].p)
      ? stack[depth - q[ptr].p]
      : 0;
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    replace_query_nodes_dfs(list[ptr].v, depth + 1);
  }

  stack[depth] = 0; // curățenie pentru al doilea DFS
}

void answer_queries_dfs(int u, int depth) {
  stack[depth]++;

  // Scanează toate interogările despre u și reține contorul inițial.
  for (int ptr = nd[u].qptr; ptr; ptr = q[ptr].next) {
    q[ptr].answer = stack[depth + q[ptr].p];
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    answer_queries_dfs(list[ptr].v, depth + 1);
  }

  // Rescanează interogările și reține diferența între valorile contorului.
  for (int ptr = nd[u].qptr; ptr; ptr = q[ptr].next) {
    q[ptr].answer = stack[depth + q[ptr].p] - q[ptr].answer - 1;
  }
}

void write_output_data() {
  for (int i = 1; i <= num_queries; i++) {
    printf("%d ", q[i].answer);
  }
  printf("\n");
}

int main() {
  read_input_data();
  distribute_queries();
  replace_query_nodes_dfs(0, 0);
  distribute_queries();
  answer_queries_dfs(0, 0);
  write_output_data();
```

```
  return 0;
}
```

Anexa G

Arbori - liniarizare

G.1 Problema Tree Queries (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 200'000;
const int NIL = -1;

struct list {
  int v, next;
};

struct node {
  int parent;
  int ptr; // pointer în lista de adiacență
  int time_in, time_out;
};

list adj[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n, num_queries;

void add_neighbor(int u, int v, int pos) {
  adj[pos] = { v, nd[u].ptr };
  nd[u].ptr = pos;
}

void read_tree() {
  scanf("%d %d", &n, &num_queries);
  for (int u = 1; u <= n; u++) {
    nd[u].ptr = NIL;
  }
```

```
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v, 2 * i);
    add_neighbor(v, u, 2 * i + 1);
  }
}

void euler_tour(int u, int parent) {
  static int time = 0;

  nd[u].parent = parent;
  nd[u].time_in = ++time;
  for (int ptr = nd[u].ptr; ptr != NIL; ptr = adj[ptr].next) {
    int v = adj[ptr].v;
    if (v != parent) {
      euler_tour(v, u);
    }
  }
  nd[u].time_out = ++time;
}

bool is_ancestor(int u, int v) {
  return
    (nd[u].time_in <= nd[v].time_in) &&
    (nd[u].time_out >= nd[v].time_out);
}

void process_queries() {
  while (num_queries--) {
    int size, u, lowest;
    bool has_path = true;
    scanf("%d %d", &size, &lowest);
    lowest = nd[lowest].parent;

    while (--size) {
      scanf("%d", &u);
      u = nd[u].parent;
      if (is_ancestor(lowest, u)) {
        lowest = u;
      } else if (!is_ancestor(u, lowest)) {
        has_path = false;
      }
    }

    printf(has_path ? "YES\n" : "NO\n");
  }
}

int main() {
```

```
  read_tree();
  euler_tour(1, 1);
  process_queries();

  return 0;
}
```

G.2 Problema Subtree Queries (CSES)

‹ înapoi

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct list {
  int val, next;
};

struct node {
  int val;
  int start, finish;
  int ptr; // pointer în lista de adiacență
};

struct fenwick_tree {
  long long v[MAX_NODES + 1];
  int n;

  void build(int n) {
    this->n = n;
    for (int i = 1; i <= n; i++) {
      int j = i + (i & -i);
      if (j <= n) {
        v[j] += v[i];
      }
    }
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  long long sum(int pos) {
    long long s = 0;
```

```
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  long long range_sum(int x, int y) {
    return sum(y) - sum(x - 1);
  }
};

list adj[2 * MAX_NODES];
node nd[MAX_NODES + 1];
fenwick_tree fen;
int n, num_queries;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  adj[ptr] = { v, nd[u].ptr };
  nd[u].ptr = ptr++;
}

void read_input_data() {
  scanf("%d %d", &n, &num_queries);
  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].val);
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void flatten(int u) {
  static int time = 0;

  nd[u].start = ++time;

  for (int ptr = nd[u].ptr; ptr; ptr = adj[ptr].next) {
    int v = adj[ptr].val;
    if (!nd[v].start) {
      flatten(v);
    }
  }

  nd[u].finish = time;
```

```
}

void build_fenwick_tree() {
  for (int u = 1; u <= n; u++) {
    fen.v[nd[u].start] = nd[u].val;
  }

  fen.build(n);
}

void process_queries() {
  while (num_queries--) {
    int type, u;
    scanf("%d %d", &type, &u);
    if (type == 1) {
      int val;
      scanf("%d", &val);
      fen.add(nd[u].start, val - nd[u].val);
      nd[u].val = val;
    } else {
      printf("%lld\n", fen.range_sum(nd[u].start, nd[u].finish));
    }
  }
}

int main() {
  read_input_data();
  flatten(1);
  build_fenwick_tree();
  process_queries();

  return 0;
}
```

G.3 Problema Path Queries (CSES)

‹ înapoi

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct list {
  int val, next;
};

struct node {
  int val;
```

```
  int time_in, time_out;
  int ptr; // pointer în lista de adiacență
};

struct fenwick_tree {
  long long v[2 * MAX_NODES + 1];
  int n;

  void build(int n) {
    this->n = n;
    for (int i = 1; i <= n; i++) {
      int j = i + (i & -i);
      if (j <= n) {
        v[j] += v[i];
      }
    }
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  long long sum(int pos) {
    long long s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

list adj[2 * MAX_NODES];
node nd[MAX_NODES + 1];
fenwick_tree fen;
int n, num_queries;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  adj[ptr] = { v, nd[u].ptr };
  nd[u].ptr = ptr++;
}

void read_input_data() {
  scanf("%d %d", &n, &num_queries);
  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].val);
```

```
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void flatten(int u) {
  static int time = 0;

  nd[u].time_in = ++time;

  for (int ptr = nd[u].ptr; ptr; ptr = adj[ptr].next) {
    int v = adj[ptr].val;
    if (!nd[v].time_in) {
      flatten(v);
    }
  }

  nd[u].time_out = ++time;
}

void build_fenwick_tree() {
  for (int u = 1; u <= n; u++) {
    fen.v[nd[u].time_in] = nd[u].val;
    fen.v[nd[u].time_out] = -nd[u].val;
  }

  fen.build(2 * n);
}

void process_queries() {
  while (num_queries--) {
    int type, u;
    scanf("%d %d", &type, &u);
    if (type == 1) {
      int val;
      scanf("%d", &val);
      int delta = val - nd[u].val;
      fen.add(nd[u].time_in, delta);
      fen.add(nd[u].time_out, -delta);
      nd[u].val = val;
    } else {
      printf("%lld\n", fen.sum(nd[u].time_in));
    }
  }
}
```

```
int main() {
  read_input_data();
  flatten(1);
  build_fenwick_tree();
  process_queries();

  return 0;
}
```

G.4 Problema New Year Tree (Codeforces)

înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 400'000;
const int MAX_NODES_ROUNDED = 1 << 19;
const int T_PAINT = 1;

typedef unsigned long long u64;

// Un arbore de segmente care admite operațiile:
//
// * range_set(int l, int r, int val)
//   Atribuie valoarea val pe [l, r]. Presupune că 1 ≤ val ≤ 60.
// * range_count_distinct(int l, int r)
//   Returnează numărul de valori distincte din [l, r].
//
// Contract: mask[x] este masca valorilor din subarborele lui x. Dacă dirty[x]
// este nonzero, atunci dirty[x] trebuie scris peste tot în subarborele lui x
// și mask[x] deja respectă dirty[x].
struct segment_tree {
  u64 mask[2 * MAX_NODES_ROUNDED];
  char dirty[2 * MAX_NODES_ROUNDED];
  int n, bits;

  int next_power_of_2(int n) {
    return 1 << (32 - __builtin_clz(n - 1));
  }

  void init(int size) {
    n = next_power_of_2(size);
    bits = __builtin_popcount(n - 1);
  }

  void raw_set(int pos, char val) {
    mask[pos + n] = 1ull << val;
```

```
}

void build() {
  for (int i = n - 1; i; i--) {
    mask[i] = mask[2 * i] | mask[2 * i + 1];
  }
}

void push(int x) {
  if (dirty[x]) {
    dirty[2 * x] = dirty[2 * x + 1] = dirty[x];
    mask[2 * x] = mask[2 * x + 1] = mask[x];
    dirty[x] = 0;
  }
}

void push_path(int x) {
  for (int b = bits; b; b--) {
    push(x >> b);
  }
}

void pull_path(int x) {
  for (x /= 2; x; x /= 2) {
    if (!dirty[x]) {
      mask[x] = mask[2 * x] | mask[2 * x + 1];
    }
  }
}

void range_set(int l, int r, int val) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;

  push_path(l);
  push_path(r);

  while (l <= r)  {
    if (l & 1) {
      mask[l] = 1ull << val;
      dirty[l++] = val;
    }
    l >>= 1;

    if (!(r & 1)) {
      mask[r] = 1ull << val;
      dirty[r--] = val;
    }
    r >>= 1;
```

```
    }

    pull_path(orig_l);
    pull_path(orig_r);
  }

  int range_count_distinct(int l, int r) {
    u64 result = 0;

    l += n;
    r += n;
    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        result |= mask[l++];
      }
      l >>= 1;

      if (!(r & 1)) {
        result |= mask[r--];
      }
      r >>= 1;
    }

    return __builtin_popcountll(result);
  }

};

struct cell {
  int v, next;
};

struct node {
  int adj; // lista de adiacență
  int time_in, time_out;
  unsigned char color;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
segment_tree segtree;
int n, num_ops;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
```

```
}

void read_tree() {
  scanf("%d %d", &n, &num_ops);
  for (int u = 1; u <= n; u++) {
    scanf("%hhd", &nd[u].color);
  }
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void flatten_tree(int u) {
  static int time = 0;

  nd[u].time_in = time++;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].time_in && (v != 1)) { // nu revizita rădăcina
      flatten_tree(v);
    }
  }

  nd[u].time_out = time - 1;
}

void make_segtree() {
  segtree.init(n);
  for (int u = 1; u <= n; u++) {
    segtree.raw_set(nd[u].time_in, nd[u].color);
  }
  segtree.build();
}

void process_ops() {
  while (num_ops--) {
    int type, u, color;
    scanf("%d %d", &type, &u);
    int l = nd[u].time_in, r = nd[u].time_out;

    if (type == T_PAINT) {
      scanf("%d", &color);
      segtree.range_set(l, r, color);
    } else {
      int colors = segtree.range_count_distinct(l, r);
      printf("%d\n", colors);
```

```
    }
  }
}

int main() {
  read_tree();
  flatten_tree(1);
  make_segtree();
  process_ops();

  return 0;
}
```

G.5 Problema Max Flow (USACO)

înapoi

Sursă cu vectori de diferențe pe arbore.

```
#include <stdio.h>

const int MAX_NODES = 50'000;
const int MAX_PATHS = 100'000;
const int NIL = -1;

struct list {
  int val, next;
};

struct node {
  int parent;
  int ptr; // pointer în lista de adiacență
  int path_ptr; // pointer în lista de căi
  int load;
};

struct path {
  int u, v, lca;
};

list adj[2 * MAX_NODES];
list path_list[2 * MAX_PATHS];
node nd[MAX_NODES + 1];
path p[MAX_PATHS];
int ds_parent[MAX_NODES + 1];
int n, num_paths;

/*******************************************************************/
```

```
/*                  Implementare de mulțimi disjuncte.                    */
/************************************************************************/

void ds_init() {
  for (int u = 1; u <= n; u++) {
    ds_parent[u] = u;
  }
}

int ds_find(int u) {
  return (ds_parent[u] == u)
    ? u
    : (ds_parent[u] = ds_find(ds_parent[u]));
}

// Întotdeauna unifică v în u. Fără unificare după rang.
void ds_union(int u, int v) {
  ds_parent[ds_find(v)] = ds_find(u);
}

/************************************************************************/
/*      Implementarea algoritmului lui Tarjan pentru LCA offline.       */
/************************************************************************/

void offline_lca_dfs(int u, int parent) {
  nd[u].parent = parent;

  for (int ptr = nd[u].ptr; ptr != NIL; ptr = adj[ptr].next) {
    int v = adj[ptr].val;
    if (v != parent) {
      offline_lca_dfs(v, u);
      ds_union(u, v);
    }
  }

  for (int ptr = nd[u].path_ptr; ptr != NIL; ptr = path_list[ptr].next) {
    int i = path_list[ptr].val;
    int v = (p[i].u == u) ? p[i].v : p[i].u;
    if (nd[v].parent) {
      p[i].lca = ds_find(v);
    }
  }
}

void offline_lca() {
  ds_init();
  offline_lca_dfs(1, 1);
}

/************************************************************************/
```

```
/*                          Codul principal.                                  */
/****************************************************************************/

void add_neighbor(int u, int v) {
  static int ptr = 0;
  adj[ptr] = { v, nd[u].ptr };
  nd[u].ptr = ptr++;
}

void add_path(int ind, int u) {
  static int ptr = 0;
  path_list[ptr] = { ind, nd[u].path_ptr };
  nd[u].path_ptr = ptr++;
}

void read_input_data() {
  FILE* f = fopen("maxflow.in", "r");
  fscanf(f, "%d %d", &n, &num_paths);
  for (int u = 1; u <= n; u++) {
    nd[u].ptr = nd[u].path_ptr = NIL;
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(f, "%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }

  for (int i = 0; i < num_paths; i++) {
    fscanf(f, "%d %d", &p[i].u, &p[i].v);
    add_path(i, p[i].u);
    add_path(i, p[i].v);
  }
  fclose(f);
}

void mark_differences() {
  for (int i = 0; i < num_paths; i++) {
    nd[p[i].u].load++;
    nd[p[i].v].load++;
    nd[p[i].lca].load--;
    if (p[i].lca != 1) {
      int lca_parent = nd[p[i].lca].parent;
      nd[lca_parent].load--;
    }
  }
}

void sum_differences(int u) {
```

```
  for (int ptr = nd[u].ptr; ptr != NIL; ptr = adj[ptr].next) {
    int v = adj[ptr].val;
    if (v != nd[u].parent) {
      sum_differences(v);
      nd[u].load += nd[v].load;
    }
  }
}

int get_max_load() {
  int result = 0;
  for (int u = 1; u <= n; u++) {
    if (nd[u].load > result) {
      result = nd[u].load;
    }
  }
  return result;
}

void write_answer(int answer) {
  FILE* f = fopen("maxflow.out", "w");
  fprintf(f, "%d\n", answer);
  fclose(f);
}

int main() {
  read_input_data();
  offline_lca();
  mark_differences();
  sum_differences(1);
  int answer = get_max_load();
  write_answer(answer);

  return 0;
}
```

Sursă cu liniarizare + AIB.

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

const int MAX_NODES = 50'000;
const int MAX_PATHS = 100'000;

struct cell {
  int v, next;
};

// Stochează două liste de noduri în fiecare nod u: una pentru căile care
```

```
// încep în u și alta pentru căile care se termină în u.
struct node {
  int adj;
  int path_begin, path_end; // pointers to list of nodes
  int start, finish;
};

struct fenwick_tree {
  unsigned short v[MAX_NODES + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  int count(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  int range_count(int x, int y) {
    return count(y) - count(x - 1);
  }
};

cell list[2 * MAX_NODES + 2 * MAX_PATHS + 1];
node nd[MAX_NODES + 1];
fenwick_tree active_start, active_finish;
int n, num_paths;
FILE* fin;

void add_to_list(int& head, int u) {
  static int ptr = 1;
  list[ptr] = { u, head };
  head = ptr++;
}

void read_tree() {
  fscanf(fin, "%d %d", &n, &num_paths);
```

```
  for (int i = 1; i < n; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    add_to_list(nd[u].adj, v);
    add_to_list(nd[v].adj, u);
  }
}

void read_paths() {
  for (int i = 0; i < num_paths; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    if (nd[u].start > nd[v].start) {
      int tmp = u; u = v; v = tmp;
    }
    add_to_list(nd[u].path_begin, v);
    add_to_list(nd[v].path_end, u);
  }
}

void compute_ranges(int u) {
  static int time = 0;

  nd[u].start = ++time;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].start) {
      compute_ranges(v);
    }
  }

  nd[u].finish = time;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int process_paths_starting_at(int u) {
  int cnt = 0;

  for (int ptr = nd[u].path_begin; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    active_finish.add(nd[v].start, +1);
    cnt++;
  }

  active_start.add(nd[u].start, +cnt);
```

```
  return cnt;
}

void process_paths_ending_at(int v) {
  int cnt = 0;

  for (int ptr = nd[v].path_end; ptr; ptr = list[ptr].next) {
    int u = list[ptr].v;
    active_start.add(nd[u].start, -1);
    cnt++;
  }

  active_finish.add(nd[v].start, -cnt);
}

int max_load = 0;

void compute_load(int u) {
  // Căi care se termină în subarborele lui u.
  int load = active_finish.range_count(nd[u].start, nd[u].finish);

  // Căi care încep în u.
  load += process_paths_starting_at(u);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (nd[v].start > nd[u].start) {
      compute_load(v);
      // Căi care au început într-un descendent al lui u și se vor termina fie
      // în alt fiu al lui u, fie în afara subarborelui lui u.
      load += active_start.range_count(nd[v].start, nd[v].finish);
    }
  }

  process_paths_ending_at(u);
  max_load = max(max_load, load);
}

void write_answer() {
  FILE* f = fopen("maxflow.out", "w");
  fprintf(f, "%d\n", max_load);
  fclose(f);
}

int main() {
  fin = fopen("maxflow.in", "r");
  read_tree();
  compute_ranges(1);
  read_paths();
  fclose(fin);
```

```
  active_start.init(n);
  active_finish.init(n);

  compute_load(1);
  write_answer();

  return 0;
}
```

G.6 Problema Distinct Colors (CSES)

înapoi

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

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int color, distinct;
  int adj; // lista de adiacență
};

// Un arbore Fenwick extins cu culori. Când colorăm un bit cu o culoare,
// arborele șterge automat apariția anterioară a culorii.
struct fenwick_tree {
  int v[MAX_NODES + 1];
  std::unordered_map<int, int> last_pos;
  int n, total;

  void init(int n) {
    this->n = n;
    total = 0;
  }

  void add(int pos, int val) {
    total += val;
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }
```

```
  void colorize(int pos, int color) {
    auto last = last_pos.find(color);
    if (last != last_pos.end()) {
      add(last->second, -1);
      last->second = pos;
    } else {
      last_pos[color] = pos;
    }
    add(pos, +1);
  }

  // count[pos..n] = total - count[1..pos-1]
  int suffix_count(int pos) {
    int s = total;
    pos--;
    while (pos) {
      s -= v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
fenwick_tree fen;
int n;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].color);
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void dfs(int u) {
```

```
  static int time = 0;
  int entry_time = ++time;

  fen.colorize(time, nd[u].color);
  nd[u].color = 0; // previne revizitarea

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (nd[v].color) {
      dfs(v);
    }
  }

  nd[u].distinct = fen.suffix_count(entry_time);
}

void write_output_data() {
  for (int u = 1; u <= n; u++) {
    printf("%d ", nd[u].distinct);
  }
  printf("\n");
}

int main() {
  read_input_data();
  fen.init(n);
  dfs(1);
  write_output_data();

  return 0;
}
```

G.7 Problema Disconnect (Infoarena)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_NODES_ROUNDED = 1 << 17;
const int T_REMOVE = 1;

// Un arbore de intervale cu operațiile:
//
// update(l, r, val): scrie val peste tot în [l, r].
// query(pos): returnează prima valoare găsită călătorind în sus de la pos.
//
// Presupune că actualizările sînt fie disjuncte, fie imbricate. Cu alte
```

```
// cuvinte, nu există suprapuneri parțiale. Actualizările mai înguste au
// prioritate în fața celor mai late.
struct segment_tree_node {
  int val;
  int priority; // cu cît mai mică, cu atît mai importantă

  void update(int new_val, int new_priority) {
    if (new_priority < priority) {
      val = new_val;
      priority = new_priority;
    }
  }
};

struct segment_tree {
  segment_tree_node v[2 * MAX_NODES_ROUNDED];
  int n;

  int next_power_of_2(int x) {
    return 1 << (32 - __builtin_clz(x - 1));
  }

  void init(int n) {
    this->n = next_power_of_2(n);
    for (int i = 1; i < 2 * this->n; i++) {
      v[i].priority = n + 1; // cea mai neimportantă
    }
  }

  int query(int pos) {
    for (pos += n; pos; pos >>= 1) {
      if (v[pos].val) {
        return v[pos].val;
      }
    }
    return 0;
  }

  void update(int l, int r, int val) {
    int priority = r - l + 1;
    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        v[l++].update(val, priority);
      }
      l >>= 1;

      if (!(r & 1)) {
```

```
        v[r--].update(val, priority);
      }
      r >>= 1;
    }
  }
};

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int start, finish;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
segment_tree segtree;
int n, num_ops;
FILE *fin, *fout;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  fscanf(fin, "%d %d", &n, &num_ops);
  for (int i = 1; i < n; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void flatten_tree(int u) {
  static int time = 0;

  nd[u].start = time++;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].start && (v != 1)) { // nu revizita rădăcina
      nd[v].parent = u;
      flatten_tree(v);
    }
```

```
  }

  nd[u].finish = time - 1;
}

void mark_node(int u) {
  segtree.update(nd[u].start, nd[u].finish, u);
}

bool bad_node(int u) {
  return (u < 1) || (u > n);
}

void remove_edge(int u, int v) {
  if (bad_node(u) || bad_node(v)) {
    return;
  }
  if (nd[u].parent == v) {
    mark_node(u);
  } else if (nd[v].parent == u) {
    mark_node(v);
  }
}

bool path_exists(int u, int v) {
  if (bad_node(u) || bad_node(v)) {
    return false;
  }

  u = segtree.query(nd[u].start);
  v = segtree.query(nd[v].start);

  return (u == v);
}

void process_ops() {
  int v = 0;

  while (num_ops--) {
    int type, x, y;
    fscanf(fin, "%d %d %d", &type, &x, &y);

    x ^= v;
    y ^= v;

    if (type == T_REMOVE) {
      remove_edge(x, y);
    } else {
      if (path_exists(x, y)) {
        fprintf(fout, "YES\n");
```

```
        v = x;
      } else {
        fprintf(fout, "NO\n");
        v = y;
      }
    }
  }
}

int main() {
  fin = fopen("disconnect.in", "r");
  fout = fopen("disconnect.out", "w");

  read_tree();
  flatten_tree(1);
  segtree.init(n);
  process_ops();

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Anexa H

Arbori - small-to-large

H.1 Problema Fixed-Length Paths I (CSES)

înapoi

```
#include <deque>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 200'000;

typedef std::deque<int> deque;

std::vector<int> adj[MAX_NODES + 1];
int n, k;
long long answer;

void read_input_data() {
  scanf("%d %d", &n, &k);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
}

// Presupune că src.size() <= dest.size().
void merge_into(deque& src, deque&dest) {
  int min_i = std::max(k - (int)dest.size() + 1, 0);
  int max_i = std::min((int)src.size() - 1, k);
  for (int i = min_i; i <= max_i; i++) {
    answer += (long long)src[i] * dest[k - i];
  }
```

```
  for (int i = 0; i < (int)src.size(); i++) {
    dest[i] += src[i];
  }
}

deque dfs(int u, int parent) {
  deque result = { 1 }; // nodul însuși
  for (int v: adj[u]) {
    if (v != parent) {
      deque d = dfs(v, u);
      d.push_front(0);
      if (d.size() > result.size()) {
        // Întotdeauna folosește swap(), care schimbă pointeri. Niciodată nu
        // folosi atribuirea, care copiază date.
        result.swap(d);
      }
      merge_into(d, result);
    }
  }

  if ((int)result.size() > k) {
    // Nu are rost să ținem statistici peste distanța k.
    result.pop_back();
  }
  return result;
}

void write_answer() {
  printf("%lld\n", answer);
}

int main() {
  read_input_data();
  dfs(1, 0);
  write_answer();

  return 0;
}
```

H.2 Problema Distinct Colors (CSES) (din nou)

înapoi

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

const int MAX_NODES = 200'000;
```

```cpp
typedef std::unordered_set<int> set;

struct cell {
  int v, next;
};

struct node {
  int color, distinct;
  int adj;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_data() {
  scanf("%d", &n);

  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].color);
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void merge_into(set& src, set& dest) {
  dest.merge(src);
  src.clear();
}

set dfs(int u) {
  // marchează-l ca vizitat ca să prevenim recursivitatea infinită
  nd[u].distinct = 1;
  set result = { nd[u].color };
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].distinct) {
      set s = dfs(v);
      if (s.size() > result.size()) {
```

```
        s.swap(result);
      }
      merge_into(s, result);
    }
  }

  nd[u].distinct = result.size();
  return result;
}

void write_answer() {
  for (int u = 1; u <= n; u++) {
    printf("%d ", nd[u].distinct);
  }
  printf("\n");
}

int main() {
  read_data();
  dfs(1);
  write_answer();

  return 0;
}
```

H.3 Problema Lomsat Gelral (Codeforces)

‹ înapoi

Sursă cu tehnica *small-to-large* (versiune online).

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

const int MAX_NODES = 100'000;

struct freq_info {
  std::unordered_map<int, int> map; // culori => frecvențe
  int max_f;                        // cea mai mare frecvență din map
  long long sum;                    // suma culorilor având max_f

  freq_info(int color) {
    map[color] = 1;
    max_f = 1;
    sum = color;
  }

  void swap(freq_info& other) {
```

```
    map.swap(other.map);
    max_f = other.max_f;
    sum = other.sum;
    // Tehnic, ar trebui să copiem și valorile noastre în other, dar nu este
    // necesar.
  }

  void absorb(freq_info& src) {
    if (src.map.size() > map.size()) {
      swap(src);
    }

    for (auto [color, f]: src.map) {
      // Salvează noua frecvență ca să nu o tot recăutăm în map.
      int new_f = (map[color] += f);
      if (new_f > max_f) {
        max_f = new_f;
        sum = color;
      } else if (new_f == max_f) {
        sum += color;
      }
    }

    src.map.clear();
  }
};

struct cell {
  int val, next;
};

struct node {
  int color;
  int adj;
  long long sum;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_input_data() {
  scanf("%d", &n);
```

```
  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].color);
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

freq_info dfs(int u) {
  freq_info result(nd[u].color);
  nd[u].color = 0; // previne recursia infinită

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].val;
    if (nd[v].color) {
      freq_info f = dfs(v);
      result.absorb(f);
    }
  }

  nd[u].sum = result.sum;
  return result;
}

void write_output_data() {
  for (int u = 1; u <= n; u++) {
    printf("%lld ", nd[u].sum);
  }
  printf("\n");
}

int main() {
  read_input_data();
  dfs(1);
  write_output_data();

  return 0;
}
```

Sursă cu algoritmul lui Mo (versiune online).

```
#include <algorithm>
#include <math.h>
#include <stdio.h>
```

```
const int MAX_NODES = 100'000;

struct list {
  int val, next;
};

struct node {
  int color, orig_pos;
  int start, finish;
  int ptr; // lista de adiacență
};

struct accountant {
  int* color;
  int f[MAX_NODES + 1];       // frecvența fiecărei culori
  int c[MAX_NODES + 1];       // numărul de culori având fiecare frecvență
  long long s[MAX_NODES + 1]; // suma culorilor având fiecare frecvență
  int left, right, max_f;

  void init(int* color) {
    this->color = color;
    left = 1; right = 0; // gol
  }

  void change_frequency(int x, int delta) {
    c[f[x]]--;
    s[f[x]] -= x;
    f[x] += delta;
    c[f[x]]++;
    s[f[x]] += x;
  }

  void include(int x) {
    change_frequency(x, +1);
    if (f[x] > max_f) {
      max_f++;
    }
  }

  void exclude(int x) {
    change_frequency(x, -1);
    if (!c[max_f]) {
      max_f--;
    }
  }

  long long query(int l, int r) {
    while (right < r) {
      include(color[++right]);
    }
```

```
    while (right > r) {
      exclude(color[right--]);
    }
    while (left < l) {
      exclude(color[left++]);
    }
    while (left > l) {
      include(color[--left]);
    }
    return s[max_f];
  }
};

list adj[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int color[MAX_NODES + 1];
long long answer[MAX_NODES + 1];
accountant acc;
int n, block_size;

void add_neighbor(int u, int v) {
  static int ptr = 1;
  adj[ptr] = { v, nd[u].ptr };
  nd[u].ptr = ptr++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].color);
    nd[u].orig_pos = u;
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_neighbor(u, v);
    add_neighbor(v, u);
  }
}

void dfs(int u) {
  static int time = 0;

  nd[u].start = ++time;
  color[time] = nd[u].color;

  for (int ptr = nd[u].ptr; ptr; ptr = adj[ptr].next) {
    int v = adj[ptr].val;
```

```
    if (!nd[v].start) {
      dfs(v);
    }
  }

  nd[u].finish = time;
}

void sort_in_mo_order() {
  std::sort(nd + 1, nd + n + 1, [](node& u, node& v) {
    // TODO fewer divisions
    int bu = u.start / block_size;
    int bv = v.start / block_size;
    if (bu != bv) {
      return bu < bv;
    } else if (bu % 2) {
      return u.finish < v.finish;
    } else {
      return u.finish > v.finish;
    }
  });
}

void process_queries() {
  block_size = sqrt(n);
  sort_in_mo_order();
  acc.init(color);
  for (int u = 1; u <= n; u++) {
    answer[nd[u].orig_pos] = acc.query(nd[u].start, nd[u].finish);
  }
}

void write_output_data() {
  for (int u = 1; u <= n; u++) {
    printf("%lld ", answer[u]);
  }
  printf("\n");
}

int main() {
  read_input_data();
  dfs(1);

  // acum fiecare nod este și o interogare [start, finish]
  process_queries();
  write_output_data();

  return 0;
}
```

H.4 Problema Tokens on a Tree (CodeChef)

‹ înapoi

Sursă cu tehnica *small-to-large* și liste proprii (versiune online).

```
#include <stdio.h>

const int MAX_NODES = 1'000'000;

struct cell {
  int f, prev, next;
};

cell list[MAX_NODES + 1];
int list_ptr;
long long total_moves;

void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

struct node {
  int parent, num_children;
  int head, tail, length;
  bool has_coin;

  void reset(bool has_coin) {
    this->has_coin = has_coin;
    num_children = head = tail = length = 0;
  }

  // Procesează un nod după ce toți fiii săi i-au transmis datele lor.
  void process(node& par) {
    prepend_self();
    add_or_move_coin();
    trim();
    spill_into(par);
  }

  void prepend_self() {
    list[list_ptr].f = 0;
    list[list_ptr].prev = 0;
    list[list_ptr].next = head;
    if (head) {
      list[head].prev = list_ptr;
    } else {
      tail = list_ptr;
```

```
    }
    head = list_ptr++;
    length++;
  }

  void add_or_move_coin() {
    if (has_coin) {
      list[head].f = 1;
    } else if (list[tail].f) {
      list[head].f = 1;
      list[tail].f--;
      total_moves += length - 1;
    }
  }

  void trim() {
    while ((tail != head) && !list[tail].f) {
      tail = list[tail].prev;
      list[tail].next = 0;
      length--;
    }
  }

  void spill_into(node& other) {
    if (length > other.length) {
      swap_lists(other);
    }

    for (int p = head, q = other.head;
         p;
         p = list[p].next, q = list[q].next)  {
      list[q].f += list[p].f;
    }
  }

  void swap_lists(node& other) {
    swap(head, other.head);
    swap(tail, other.tail);
    swap(length, other.length);
  }
};

node nd[MAX_NODES + 1];
int n;

void read_input_data() {
  list_ptr = 1;

  scanf("%d ", &n);
```

```
  for (int u = 1; u <= n; u++) {
    int c = getchar();
    nd[u].reset(c == '1');
  }

  for (int u = 2; u <= n; u++) {
    scanf("%d", &nd[u].parent);
    nd[nd[u].parent].num_children++;
  }
}

void traverse() {
  for (int u = 1; u <= n; u++) {
    int p = u;
    while (p && !nd[p].num_children) {
      nd[p].process(nd[nd[p].parent]);
      p = nd[p].parent;
      nd[p].num_children--;
    }
  }
}

void run_test() {
  read_input_data();
  total_moves = 0;
  traverse();
  printf("%lld\n", total_moves);
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    run_test();
  }

  return 0;
}
```

Sursă cu tehnica *small-to-large* și liste STL (versiune online).

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

const int MAX_NODES = 1'000'000;

long long total_moves;

void swap(int& x, int& y) {
```

```
  int tmp = x;
  x = y;
  y = tmp;
}

struct node {
  int parent, num_children;
  std::list<int> f; // numărul de monede la adîncimile 0, 1, ...
  bool has_coin;

  void reset(bool has_coin) {
    this->has_coin = has_coin;
    f.clear();
  }

  // Procesează un nod după ce toți fiii săi i-au transmis datele lor.
  void process(node& par) {
    prepend_self();
    trim();
    spill_into(par);
  }

  void prepend_self() {
    if (has_coin) {
      f.push_front(1);
    } else if (!f.empty() && f.back()) {
      f.push_front(1);
      f.back()--;
      total_moves += f.size() - 1;
    }
    // altfel nu există monete în subarbore și nu adăugăm un element 0
  }

  void trim() {
    while (!f.empty() && !f.back()) {
      f.pop_back();
    }
  }

  void spill_into(node& other) {
    if (f.size() > other.f.size()) {
      f.swap(other.f);
    }

    for (auto it = f.begin(), other_it = other.f.begin();
         it != f.end();
         it++, other_it++) {
      *other_it += *it;
    }
```

```
    f.clear();
  }
};

node nd[MAX_NODES + 1];
int n;

void read_input_data() {
  scanf("%d ", &n);

  for (int u = 1; u <= n; u++) {
    int c = getchar();
    nd[u].reset(c == '1');
  }

  for (int u = 2; u <= n; u++) {
    scanf("%d", &nd[u].parent);
    nd[nd[u].parent].num_children++;
  }
}

void traverse() {
  for (int u = 1; u <= n; u++) {
    int p = u;
    while (p && !nd[p].num_children) {
      nd[p].process(nd[nd[p].parent]);
      p = nd[p].parent;
      nd[p].num_children--;
    }
  }
}

void run_test() {
  read_input_data();
  total_moves = 0;
  traverse();
  printf("%lld\n", total_moves);
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    run_test();
  }

  return 0;
}
```

Sursă cu liniarizare și RMQ (versiune online).

```
#include <stdio.h>

const int MAX_NODES = 1'000'000;
const int MAX_SEGTREE_NODES = 1 << 21;

struct cell {
  int v, next;
};

struct node {
  int adj;
  bool has_coin;
};

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

struct segment_tree {
  int v[MAX_SEGTREE_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
    for (int i = 1; i < 2 * this->n; i++) {
      v[i] = 0;
    }
  }

  void set(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
    }
  }

  void optimize(int pos, int& max, int& best_pos) {
    if (v[pos] > max) {
      max = v[pos];
      best_pos = pos;
    }
  }
```

```
  void descend_and_erase(int pos) {
    while (pos < n) {
      pos = (v[pos] == v[2 * pos])
        ? (2 * pos)
        : (2 * pos + 1);
    }
    set(pos - n, 0);
  }

  // Returnează maximul din [l, r] și îl înlocuiește cu 0.
  int erase_rmq(int l, int r) {
    int result = -1, pos = 0;

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        optimize(l++, result, pos);
      }
      l >>= 1;

      if (!(r & 1)) {
        optimize(r--, result, pos);
      }
      r >>= 1;
    }

    // Coboară din pos într-o frunză care are aceeași valoare și șterge
    // valoarea.
    descend_and_erase(pos);

    return result;
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
segment_tree segtree;
int n, list_ptr, dfs_time;

void add_child(int u, int v) {
  list[list_ptr] = { v, nd[u].adj };
  nd[u].adj = list_ptr++;
}

void read_input_data() {
  scanf("%d ", &n);
```

```
  for (int u = 1; u <= n; u++) {
    int c = getchar();
    nd[u].has_coin = (c == '1');
    nd[u].adj = 0; // null
  }

  list_ptr = 1;
  for (int u = 2; u <= n; u++) {
    int p;
    scanf("%d", &p);
    add_child(p, u);
  }
}

long long dfs(int u, int depth) {

  long long result = 0;
  int start = dfs_time++;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    result += dfs(list[ptr].v, depth + 1);
  }

  if (nd[u].has_coin) {
    segtree.set(start, depth);
  } else {
    int lowest_coin = segtree.erase_rmq(start, dfs_time - 1);
    if (lowest_coin) {
      segtree.set(start, depth);
      result += lowest_coin - depth;
    }
  }

  return result;
}

void run_test() {
  read_input_data();
  segtree.init(n);
  dfs_time = 0;
  long long moves = dfs(1, 1);
  printf("%lld\n", moves);
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    run_test();
  }
```

```
  return 0;
}
```

H.5 Problema Blood Cousins Return (Codeforces)

înapoi

Sursă cu vectori de mulțimi (versiune online).

```
// Complexitate: O(n log n).
//
// Fiecare nod calculează un set de nume distincte la fiecare adîncime în
// subarborele său.
#include <iostream>
#include <unordered_map>
#include <set>
#include <string>
#include <vector>

const int MAX_NODES = 100'000;
const int MAX_QUERIES = 100'000;

struct query {
  int orig_index;
  int depth;
};

struct node {
  std::vector<int> adj;
  std::vector<query> queries;
  int parent;
  int name;
};

struct name_map {
  std::unordered_map<std::string, int> map;

  int insert_and_get_number(std::string s) {
    auto it = map.find(s);
    if (it == map.end()) {
      int result = map.size();
      map[s] = result;
      return result;
    } else {
      return it->second;
    }
  }
```

```
};

// Urmărește elementele distincte la fiecare adîncime. Adîncimea 0 (care
// conține doar nodul însuși) este ultimul element din vector.
struct dist {
  std::vector<std::set<int>> d;

  void absorb(dist other) {
    if (other.d.size() > d.size()) {
      d.swap(other.d);
    }
    int offset = d.size() - other.d.size();
    for (unsigned i = 0; i < other.d.size(); i++) {
      std::set<int>& us = d[i + offset];
      std::set<int>& them = other.d[i];
      if (them.size() > us.size()) {
        us.swap(them);
      }
      us.merge(them);
      them.clear();
    }
  }

  void prepend(int name) {
    d.push_back({name});
  }

  int count_distinct(unsigned depth) {
    if (depth < d.size()) {
      return d[d.size() - 1 - depth].size();
    } else {
      return 0;
    }
  }
};

node nd[MAX_NODES + 1]; // Vom folosi nodul 0 ca pe un strămoș comun fals.
int sol[MAX_QUERIES];
int n, q;

void read_tree() {
  name_map map;

  std::cin >> n;
  for (int u = 1; u <= n; u++) {
    std::string name;
    std::cin >> name >> nd[u].parent;
    nd[nd[u].parent].adj.push_back(u);
    nd[u].name = map.insert_and_get_number(name);
  }
```

```
}

void read_queries() {
  std::cin >> q;
  for (int i = 0; i < q; i++) {
    int u, depth;
    std::cin >> u >> depth;
    nd[u].queries.push_back({i, depth});
  }
}

dist dfs(int u) {
  dist result;
  for (int v: nd[u].adj) {
    result.absorb(dfs(v));
  }
  result.prepend(nd[u].name);

  for (query q: nd[u].queries) {
    sol[q.orig_index] = result.count_distinct(q.depth);
  }
  return result;
}

void write_answers() {
  for (int i = 0; i < q; i++) {
    std::cout << sol[i] << '\n';
  }
}

int main() {
  read_tree();
  read_queries();
  dfs(0);
  write_answers();

  return 0;
}
```

Sursă cu liniarizare (versiune online).

```
// 1. Mapează numele la numere ca de obicei.
// 2. Rulează un DFS pentru a calcula adîncimile și timpii de intrare/ieșire.
// 3. Rescrie fiecare interogare ca <u,d> ca „interoghează nodurile de nivel
//    depth[u] + d descoperite de DFS între timpii [t_i[u], t_o[u]]”.
// 4. Ordonează interogările după adîncime, apoi după t_o.
// 5. Calculează ordinea BFS. Fiecare interogare corespunde unui interval
//    contiguu în această ordonare.
// 6. Răspunde la intrerogări folosind un AIB ca în problema D-query.
```

```cpp
#include <algorithm>
#include <stdio.h>
#include <unordered_map>
#include <string>
#include <vector>

const int MAX_NODES = 100'000;
const int MAX_QUERIES = 100'000;
const int MAX_NAME_LENGTH = 20;

struct node {
  std::vector<int> children;
  int name;
  int tin, tout;
  int depth;
};

struct name_map {
  std::unordered_map<std::string, int> map;

  int insert_and_get_number(char* s) {
    std::string str(s);
    auto it = map.find(str);
    if (it == map.end()) {
      int result = map.size();
      map[str] = result;
      return result;
    } else {
      return it->second;
    }
  }
};

struct query {
  int orig_index;
  int depth;
  int tin, tout;
};

struct fenwick_tree {
  int v[MAX_NODES + 1];
  int last[MAX_NODES + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int delta) {
    do {
```

```
      v[pos] += delta;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void set(int pos, int name) {
    if (last[name]) {
      add(last[name], -1);
    }
    add(pos, +1);
    last[name] = pos;
  }

  int prefix_sum(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }

  int range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};

node nd[MAX_NODES + 1];
int bfs[MAX_NODES + 1];
query q[MAX_QUERIES];
int sol[MAX_QUERIES];
fenwick_tree fen;
int n, num_queries;

void read_tree() {
  name_map map;
  char name[MAX_NAME_LENGTH + 1];
  int parent;

  scanf("%d", &n);
  for (int u = 1; u <= n; u++) {
    scanf("%s %d", name, &parent);
    nd[u].name = map.insert_and_get_number(name);
    nd[parent].children.push_back(u);
  }
}

void dfs(int u) {
  static int time = 0;
  nd[u].tin = time++;
```

```
  for (int v: nd[u].children) {
    nd[v].depth = 1 + nd[u].depth;
    dfs(v);
  }

  nd[u].tout = time - 1;
}

void read_queries() {
  scanf("%d", &num_queries);

  for (int i = 0; i < num_queries; i++) {
    int u, depth;
    scanf("%d %d", &u, &depth);
    q[i] = { i, nd[u].depth + depth, nd[u].tin, nd[u].tout };
  }
}

void sort_queries() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return (a.depth < b.depth) ||
      ((a.depth == b.depth) && (a.tout < b.tout));
  });
}

void breadth_first_search() {
  int head = 0, tail = 0;
  bfs[tail++] = 0; // pune rădăcina în coadă

  while (head != tail) {
    int u = bfs[head++];
    for (int v: nd[u].children) {
      bfs[tail++] = v;
    }
  }
}

bool query_ends_at(query& q, int k) {
  if (k == n) {
    return true;
  }

  node& u = nd[bfs[k + 1]]; // următorul în BFS
  return (u.depth > q.depth) || ((u.depth == q.depth) && (u.tout > q.tout));
}

// Unde începe această interogare? Caută cel mai din stînga nod din BFS la
// adîncimea q.depth și cu timpul DFS cel puțin q.tin.
int bin_search(query& q, int end) {
```

```
  int l = 0, r = end; // (r, l]
  while (r - l > 1) {
    int mid = (r + l) / 2;
    int v = bfs[mid];
    if ((nd[v].depth < q.depth) ||
        (nd[v].tin < q.tin)) {
      l = mid;
    } else {
      r = mid;
    }
  }

  int v = bfs[r];
  return (nd[v].depth == q.depth) && (nd[v].tin >= q.tin)
    ? r
    : (end + 1);
}

void answer_queries_ending_at(int k) {
  static int i = 0;
  while ((i < num_queries) && query_ends_at(q[i], k)) {
    int start = bin_search(q[i], k);
    sol[q[i].orig_index] = fen.range_sum(start, k);
    i++;
  }
}

void answer_queries() {
  fen.init(n);

  for (int i = 1; i <= n; i++) {
    int v = bfs[i];
    fen.set(i, nd[v].name);
    answer_queries_ending_at(i);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", sol[i]);
  }
}

int main() {
  read_tree();
  dfs(0);
  breadth_first_search();
  read_queries();
  sort_queries();
  breadth_first_search();
```

```
  answer_queries();
  write_answers();

  return 0;
}
```

H.6 Problema Tree and Queries (Codeforces)

înapoi

Sursă cu PBDS (versiune online).

```
// Small-to-large relativ brut. Fiecare nod stochează informații despre
// subarbore:
//
// 1. Un map de culoare => frecvență.
// 2. Un multiset doar cu frecvențele.
//
// Folosim STL cu larghețe pentru un cod cît mai lent.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <iostream>
#include <map>
#include <vector>

// Multiseturile PBDS sînt obscene, dar par să meargă. Ștergerile necesită cod
// extra. Vezi https://stackoverflow.com/q/59731946/6022817
typedef __gnu_pbds::tree<
  int,
  __gnu_pbds::null_type,
  std::less_equal<int>,
  __gnu_pbds::rb_tree_tag,
  __gnu_pbds::tree_order_statistics_node_update
  > multiset;

const int MAX_NODES = 100'000;
const int MAX_QUERIES = 100'000;

struct query {
  int orig_index;
  int min_freq;
};

struct node {
  int color;
  std::vector<int> adj;
  std::vector<query> q;
};
```

```
struct freq_info {
  std::map<int, int> f;
  multiset sorted_f;

  freq_info(int color) {
    f.insert({color, 1});
    sorted_f.insert(1);
  }

  void update_frequency(int color, int cnt) {
    int new_freq;
    auto it = f.find(color);
    if (it != f.end()) {
      int old_freq = it->second;
      int rank = sorted_f.order_of_key(old_freq);
      multiset::iterator to_erase = sorted_f.find_by_order(rank);
      sorted_f.erase(to_erase);
      it->second += cnt;
      new_freq = it->second;
    } else {
      f.insert({color, cnt});
      new_freq = cnt;
    }
    sorted_f.insert(new_freq);
  }

  void absorb(freq_info& other) {
    if (other.f.size() > f.size()) {
      f.swap(other.f);
      sorted_f.swap(other.sorted_f);
    }
    for (auto [color, cnt]: other.f) {
      update_frequency(color, cnt);
    }
    other.f.clear();
    other.sorted_f.clear();
  }

  int count_freq_gte(int min_freq) {
    return f.size() - sorted_f.order_of_key(min_freq);
  }
};

node nd[MAX_NODES + 1];
int sol[MAX_QUERIES];
int n, q;

void read_data() {
  std::cin >> n >> q;
```

```
  for (int u = 1; u <= n; u++) {
    std::cin >> nd[u].color;
  }
  for (int i = 1; i < n; i++) {
    int u, v;
    std::cin >> u >> v;
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
  for (int i = 0; i < q; i++) {
    int u, min_freq;
    std::cin >> u >> min_freq;
    nd[u].q.push_back({i, min_freq});
  }
}

freq_info dfs(int u) {
  freq_info result(nd[u].color);
  nd[u].color = 0; // previne revizitarea fără a necesita un argument în plus

  for (int v: nd[u].adj) {
    if (nd[v].color) {
      freq_info fi = dfs(v);
      result.absorb(fi);
    }
  }

  for (query q: nd[u].q) {
    sol[q.orig_index] = result.count_freq_gte(q.min_freq);
  }

  return result;
}

void write_solution() {
  for (int i = 0; i < q; i++) {
    std::cout << sol[i] << '\n';
  }
}

int main() {
  read_data();
  dfs(1);
  write_solution();

  return 0;
}
```

Sursă cu DFS exclusiv (versiune online).

```cpp
// Small-to-large cu DFS exclusiv. Rescris după
// https://codeforces.com/contest/375/submission/5508178
//
// Folosește o singură structură de date globală conținînd:
//
// 1. Frecvența fiecărei culori.
// 2. Frecvențele frecvențelor („cîte culori au frecvența f?”).
// 3. Un AIB peste (2) care permite sume pe sufix.
//
// Funcționarea DFS-ului îi garantează fiecărui nod un moment cînd structura
// conține doar informații despre subarborele acelui nod.
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;
const int MAX_COLORS = 100'000;
const int MAX_QUERIES = 100'000;

struct fenwick_tree {
  int v[MAX_NODES + 1];
  int total;
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int val) {
    if (pos) {
      total += val;
      do {
        v[pos] += val;
        pos += pos & -pos;
      } while (pos <= n);
    }
  }

  int sum(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  int suffix_sum(int pos) {
    return (pos > n)
      ? 0
```

```
      : (total - sum(pos - 1));
  }
};

struct query {
  int orig_index;
  int min_freq;
};

struct node {
  int color;
  int heavy;
  std::vector<int> adj;
  std::vector<query> q;
};

node nd[MAX_NODES + 1];
int freq[MAX_COLORS + 1];
fenwick_tree fen;
int sol[MAX_QUERIES];
int n, q;

void read_data() {
  scanf("%d %d", &n, &q);
  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].color);
  }
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
  for (int i = 0; i < q; i++) {
    int u, min_freq;
    scanf("%d %d", &u, &min_freq);
    nd[u].q.push_back({i, min_freq});
  }
}

// Calculează fiul heavy al fiecărui nod. Șterge părintele nodului din lista
// de adiacență. Returnează mărimea subarborelui.
int size_dfs(int u, int parent) {
  int size = 1, max_c_size = 0;

  unsigned i = 0;
  while (i < nd[u].adj.size()) {
    int v = nd[u].adj[i];
    if (v == parent) {
      nd[u].adj[i] = nd[u].adj.back();
```

```
      nd[u].adj.pop_back();
    } else {
      int c = size_dfs(v, u);
      size += c;
      if (!nd[u].heavy || (c > max_c_size)) {
        nd[u].heavy = v;
        max_c_size = c;
      }
      i++;
    }
  }

  return size;
}

void change_freq(int color, int delta) {
  fen.add(freq[color], -1);
  freq[color] += delta;
  fen.add(freq[color], +1);
}

void toggle_subtree(int u, int delta) {
  change_freq(nd[u].color, delta);
  for (int v: nd[u].adj) {
    toggle_subtree(v, delta);
  }
}

void dfs(int u) {
  for (int v: nd[u].adj) {
    if (v != nd[u].heavy) {
      dfs(v);
      toggle_subtree(v, -1);
    }
  }

  if (nd[u].heavy) {
    dfs(nd[u].heavy);
  }

  for (int v: nd[u].adj) {
    if (v != nd[u].heavy) {
      toggle_subtree(v, +1);
    }
  }

  change_freq(nd[u].color, +1);

  for (query q: nd[u].q) {
    sol[q.orig_index] = fen.suffix_sum(q.min_freq);
```

```
  }
}

void write_solution() {
  for (int i = 0; i < q; i++) {
    printf("%d\n", sol[i]);
  }
}

int main() {
  read_data();
  size_dfs(1, 0);
  fen.init(n);
  dfs(1);
  write_solution();

  return 0;
}
```

Anexa I

Arbori - cel mai apropiat strămoș comun

I.1 LCA cu descompunere în radical

‹ înapoi

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

const int MAX_NODES = 500'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth;
  // Fiecare nod stochează doi pointeri la strămoși: unul la părinte, altul cu
  // sqrt(n) niveluri mai sus.
  int parent;
  int jump;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int st[MAX_NODES + 1];    // stivă DFS pentru construcția pointerilor
int n, jump_distance;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
```

```
}

void read_input_data() {
  scanf("%d", &n);
  jump_distance = sqrt(n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează părinții, adîncimile și jump pointers.
void dfs(int u) {
  st[nd[u].depth] = u;  // Stiva reține nodurile gri.
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].depth) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].jump = (nd[v].depth > jump_distance)
        ? st[nd[v].depth - jump_distance]
        : 0;
      dfs(v);
    }
  }
}

int sqrt_lca(int u, int v) {
  if (nd[v].depth > nd[u].depth) {
    int tmp = u; u = v; v = tmp;
  }

  // Întîi adu-le la același nivel.
  while (nd[u].depth - jump_distance >= nd[v].depth) {
    u = nd[u].jump;
  }
  while (nd[u].depth > nd[v].depth) {
    u = nd[u].parent;
  }

  // Urcă blocuri întregi.
  while ((nd[u].depth >= jump_distance) && (nd[u].jump != nd[v].jump)) {
    u = nd[u].jump;
    v = nd[v].jump;
  }

  // Urcă nivel cu nivel.
```

```
  while (u != v) {
    u = nd[u].parent;
    v = nd[v].parent;
  }
  return u;
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", sqrt_lca(u, v));
  }
}

int main() {
  read_input_data();
  nd[1].depth = 1;
  dfs(1);
  answer_queries();

  return 0;
}
```

I.2 LCA cu binary lifting ($\log n$ pointeri per nod)

‹ înapoi

Implementare cu urcare în paralel.

```
#include <stdio.h>

const int MAX_NODES = 500'000;
const int MAX_LOG = 19;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth;
  int jump[MAX_LOG];
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
```

```
int n, log_2;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);
  log_2 = 31 - __builtin_clz(n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează adîncimile și jump pointers.
void dfs(int u, int parent) {
  nd[u].depth = 1 + nd[parent].depth;
  nd[u].jump[0] = parent;
  for (int i = 0; i < log_2; i++) {
    nd[u].jump[i + 1] = nd[nd[u].jump[i]].jump[i];
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      dfs(v, u);
    }
  }
}

int lca(int u, int v) {
  if (nd[v].depth > nd[u].depth) {
    int tmp = u; u = v; v = tmp;
  }

  // Adu u și v la aceeași adîncime. Compară d[u] și d[v] bit cu bit.
  int pow = 0;
  while (nd[u].depth > nd[v].depth) {
    if ((nd[u].depth & (1 << pow)) != (nd[v].depth & (1 << pow))) {
      u = nd[u].jump[pow];
    }
    pow++;
  }
```

```
  if (u == v) {
    return u;
  } else {
    for (int i = log_2; i >= 0; i--) {
      if (nd[u].jump[i] != nd[v].jump[i]) {
        u = nd[u].jump[i];
        v = nd[v].jump[i];
      }
    }

    return nd[u].jump[0];
  }
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", lca(u, v));
  }
}

int main() {
  read_input_data();
  dfs(1, 0);
  answer_queries();

  return 0;
}
```

Implementare cu test de strămoș.

```
#include <stdio.h>

const int MAX_NODES = 500'000;
const int MAX_LOG = 19;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int time_in, time_out;
  int jump[MAX_LOG];
};

cell list[2 * MAX_NODES];
```

```
node nd[MAX_NODES + 1];
int n, log_2;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);
  log_2 = 31 - __builtin_clz(n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează timpii de intrare în noduri și jump
// pointers.
void dfs(int u, int parent) {
  static int time = 0;
  nd[u].time_in = time++;

  nd[u].jump[0] = parent;
  for (int i = 0; i < log_2; i++) {
    nd[u].jump[i + 1] = nd[nd[u].jump[i]].jump[i];
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      dfs(v, u);
    }
  }

  nd[u].time_out = time++;
}

bool is_ancestor(int u, int v) {
  return
    (nd[u].time_in <= nd[v].time_in) &&
    (nd[u].time_out >= nd[v].time_out);
}

int lca(int u, int v) {
  if (is_ancestor(u, v)) {
```

```
    return u;
  }

  // Găsește cel mai de sus strămoș al lui u care *nu* este strămoș al lui v.
  for (int i = log_2; i >= 0; i--) {
    if (nd[u].jump[i] && !is_ancestor(nd[u].jump[i], v)) {
      u = nd[u].jump[i];
    }
  }

  // Acum părintele lui u este strămoș al lui v.
  return nd[u].jump[0];
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", lca(u, v));
  }
}

int main() {
  read_input_data();
  dfs(1, 0);
  answer_queries();

  return 0;
}
```

I.3 LCA cu binary lifting (2 pointeri per nod)

‹ înapoi

Implementare cu urcare în paralel.

```
#include <stdio.h>

const int MAX_NODES = 500'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth;
```

```
  int parent;
  int jump;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează părinții, adîncimile și jump pointers.
void dfs(int u) {

  int u2 = nd[u].jump, u3 = nd[u2].jump;
  bool equal = (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].depth) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].jump = equal ? u3 : u;
      dfs(v);
    }
  }
}

int two_ptr_lca(int u, int v) {
  if (nd[v].depth > nd[u].depth) {
    int tmp = u; u = v; v = tmp;
  }

  // Întîi adu-le la același nivel.
  while (nd[u].depth > nd[v].depth) {
    u = (nd[nd[u].jump].depth >= nd[v].depth) ? nd[u].jump : nd[u].parent;
```

```
  }

  while (u != v) {
    if (nd[u].jump != nd[v].jump) {
      u = nd[u].jump;
      v = nd[v].jump;
    } else {
      u = nd[u].parent;
      v = nd[v].parent;
    }
  }
  return u;
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", two_ptr_lca(u, v));
  }
}

int main() {
  read_input_data();
  nd[1].depth = 1;
  dfs(1);
  answer_queries();

  return 0;
}
```

Implementare cu test de strămoș.

```
#include <stdio.h>

const int MAX_NODES = 500'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth;
  int time_in, time_out;
  int parent;
  int jump;
};
```

```
cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează părinții, adîncimile și jump pointers.
void dfs(int u) {
  static int time = 1;
  nd[u].time_in = time++;

  int u2 = nd[u].jump, u3 = nd[u2].jump;
  bool equal = (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].depth) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].jump = equal ? u3 : u;
      dfs(v);
    }
  }

  nd[u].time_out = time++;
}

bool is_ancestor(int u, int v) {
  return
    (nd[u].time_in <= nd[v].time_in) &&
    (nd[u].time_out >= nd[v].time_out);
}

int two_ptr_lca(int u, int v) {
```

```
  // Găsește cel mai jos strămoș al lui u care este și strămoș al lui v.
  while (!is_ancestor(u, v)) {
    if (nd[u].jump && !is_ancestor(nd[u].jump, v)) {
      u = nd[u].jump;
    } else {
      u = nd[u].parent;
    }
  }

  return u;
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", two_ptr_lca(u, v));
  }
}

int main() {
  read_input_data();
  nd[1].depth = 1;
  dfs(1);
  answer_queries();

  return 0;
}
```

Implementare cu formula LSB și urcare în paralel.

```
#include <stdio.h>

const int MAX_NODES = 500'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth;
  // Fiecare nod stochează doi pointeri la strămoși: unul la părinte, altul cu
  // LSB(adîncime) niveluri mai sus, similar unui arbore Fenwick.
  int parent;
  int jump;
};
```

```
cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int st[MAX_NODES + 1];    // stivă DFS pentru construcția pointerilor
int n;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează părinții, adâncimile și jump pointers.
void dfs(int u) {
  st[nd[u].depth] = u;  // Stiva reține nodurile gri.

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].depth) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].jump = st[nd[v].depth & (nd[v].depth - 1)];
      dfs(v);
    }
  }
}

int two_ptr_lca(int u, int v) {
  if (nd[v].depth > nd[u].depth) {
    int tmp = u; u = v; v = tmp;
  }

  // Întîi adu-le la același nivel.
  while (nd[u].depth > nd[v].depth) {
    u = (nd[nd[u].jump].depth >= nd[v].depth) ? nd[u].jump : nd[u].parent;
  }

  while (u != v) {
    if (nd[u].jump != nd[v].jump) {
      u = nd[u].jump;
```

```
      v = nd[v].jump;
    } else {
      u = nd[u].parent;
      v = nd[v].parent;
    }
  }
  return u;
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", two_ptr_lca(u, v));
  }
}

int main() {
  read_input_data();
  nd[1].depth = 1;
  dfs(1);
  answer_queries();

  return 0;
}
```

I.4 LCA cu algoritmul lui Tarjan (offline)

‹ înapoi

Implementarea folosește STL, ca să ne putem concentra pe algoritm. Vă reamintesc însă că arborii scriși cu STL sînt de două ori mai lenți și consumă dublul memoriei.

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

const int MAX_NODES = 500'000;
const int MAX_QUERIES = 500'000;

struct disjoint_set_forest {
  int p[MAX_NODES + 1];

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
    }
  }
```

```
  int find(int u) {
    return (p[u] == u)
      ? u
      : (p[u] = find(p[u]));
  }

  // Întotdeauna îl leagă pe v de u. Fără *union by rank*.
  void unite(int u, int v) {
    p[find(v)] = find(u);
  }
};

struct node {
  std::vector<int> adj, queries;
  bool visited;
};

struct query {
  int u, v, lca;
};

node nd[MAX_NODES + 1];
query q[MAX_QUERIES];
disjoint_set_forest  dsf;
int n, num_queries;

void read_input_data() {
  scanf("%d", &n);
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }

  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].u, &q[i].v);
    nd[q[i].u].queries.push_back(i);
    nd[q[i].v].queries.push_back(i);
  }
}

void offline_lca(int u) {
  nd[u].visited = true;

  for (int v: nd[u].adj) {
    if (!nd[v].visited) {
      offline_lca(v);
```

```
      dsf.unite(u, v);
    }
  }

  for (int i: nd[u].queries) {
    int v = (q[i].u == u) ? q[i].v : q[i].u;
    if (nd[v].visited) {
      q[i].lca = dsf.find(v);
    }
  }
}

void answer_queries() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", q[i].lca);
  }
}

int main() {
  read_input_data();
  dsf.init(n);
  offline_lca(1);
  answer_queries();

  return 0;
}
```

I.5 Problema Gold Transfer (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 300'001;
const int OP_ADD_NODE = 1;
const int OP_BUY_GOLD = 2;

struct node {
  int depth;
  int supply;
  int cost;
  int parent;
  int jump;
};

struct purchase {
  int amount;
  long long cost;
```

```
};

node nd[MAX_NODES];
int stack[MAX_NODES];
int num_queries;

int min(int x, int y) {
  return (x < y) ? x : y;
}

void create_node_zero() {
  scanf("%d %d %d", &num_queries, &nd[0].supply, &nd[0].cost);
  nd[0].depth = 1;
}

void read_node(int u) {
  scanf("%d %d %d", &nd[u].parent, &nd[u].supply, &nd[u].cost);
  int p = nd[u].parent, p2 = nd[p].jump, p3 = nd[p2].jump;
  bool equal = (nd[p2].depth - nd[p].depth == nd[p3].depth - nd[p2].depth);
  nd[u].depth = 1 + nd[p].depth;
  nd[u].jump = equal ? p3 : p;
}

void buy_from_node(int u, int& demand, purchase& p) {
  int bought = min(demand, nd[u].supply);
  demand -= bought;
  nd[u].supply -= bought;
  p.amount += bought;
  p.cost += (long long)bought * nd[u].cost;
}

purchase buy_from_path(int path_end, int demand) {
  purchase result = { 0, 0 };
  int ptr = 1;
  stack[0] = path_end;

  // Binary lifting după caz. Păstrează rezultatele pe stivă pentru a obține
  // coborîrea în O(1) amortizat.
  while (ptr && demand) {
    int u = stack[ptr - 1];
    if (u && nd[nd[u].jump].supply) {
      stack[ptr++] = nd[u].jump;
    } else if (u && nd[nd[u].parent].supply) {
      stack[ptr++] = nd[u].parent;
    } else {
      buy_from_node(u, demand, result);
      if (!nd[u].supply) {
        ptr--;
      }
    }
```

```
  }

  return result;
}

void buy_gold() {
  int u, demand;
  scanf("%d %d", &u, &demand);
  purchase p = buy_from_path(u, demand);
  printf("%d %lld\n", p.amount, p.cost);
  fflush(stdout);
}

void process_queries() {
  for (int q = 1; q <= num_queries; q++) {
    int type;
    scanf("%d", &type);
    if (type == OP_ADD_NODE) {
      read_node(q);
    } else {
      buy_gold();
    }
  }
}

int main() {
  create_node_zero();
  process_queries();

  return 0;
}
```

I.6 Problema A and B and Lecture Rooms (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth;
  int time_in, time_out;
```

```
  int parent;
  int jump;

  int subtree_size() {
    return time_out - time_in + 1;
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Traversează arborele și calculează părinții, adîncimile și *jump pointers*.
void dfs(int u) {
  static int time = 1;
  nd[u].time_in = time++;

  int u2 = nd[u].jump, u3 = nd[u2].jump;
  bool equal = (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth);
  int dest = equal ? u3 : u;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].depth) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].jump = dest;
      dfs(v);
    }
  }

  nd[u].time_out = time - 1;
}
```

```
bool is_ancestor(int u, int v) {
  return
    (nd[u].time_in <= nd[v].time_in) &&
    (nd[u].time_out >= nd[v].time_out);
}

int two_ptr_lca(int u, int v) {
  // Găsește cel mai jos strămoș al lui u care este și strămoș al lui v.
  while (!is_ancestor(u, v)) {
    if (nd[u].jump && !is_ancestor(nd[u].jump, v)) {
      u = nd[u].jump;
    } else {
      u = nd[u].parent;
    }
  }

  return u;
}

int get_ancestor_at_depth(int u, int d) {
  while (nd[u].depth > d) {
    u = (nd[nd[u].jump].depth >= d) ? nd[u].jump : nd[u].parent;
  }
  return u;
}

int query(int u, int v) {
  if ((nd[u].depth + nd[v].depth) % 2) {
    return 0;
  }

  if (u == v) {
    return n;
  }

  int l = two_ptr_lca(u, v);
  if (nd[u].depth == nd[v].depth) {
    int u2 = get_ancestor_at_depth(u, nd[l].depth + 1);
    int v2 = get_ancestor_at_depth(v, nd[l].depth + 1);
    return n - nd[u2].subtree_size() - nd[v2].subtree_size();
  }

  if (nd[u].depth < nd[v].depth) {
    int tmp = u; u = v; v = tmp;
  }

  int mid_level = nd[l].depth + (nd[u].depth - nd[v].depth) / 2;
  int u2 = get_ancestor_at_depth(u, mid_level + 1);
  int mid = nd[u2].parent;
```

```
  return nd[mid].subtree_size() - nd[u2].subtree_size();
}

void answer_queries() {
  int num_queries, u, v;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &u, &v);
    printf("%d\n", query(u, v));
  }
}

int main() {
  read_input_data();
  nd[1].depth = 1;
  dfs(1);
  answer_queries();

  return 0;
}
```

I.7 Problema Company (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_SEGTREE_NODES = 256 * 1024;
const int INF = 1'000'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int jump;
  int depth;
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
```

```
}

struct segment_tree_node {
  int min1, min2, max1, max2;

  void set(int val) {
    min1 = max1 = val;
    min2 = INF;
    max2 = -INF;
  }

  void combine(segment_tree_node& other) {
    if (min1 < other.min1) {
      // păstrăm min1
      min2 = min(min2, other.min1);
    } else {
      min2 = min(min1, other.min2);
      min1 = other.min1;
    }
    if (max1 > other.max1) {
      // păstrăm max1
      max2 = max(max2, other.max1);
    } else {
      max2 = max(max1, other.max2);
      max1 = other.max1;
    }
  }
};

// Un arbore de intervale care stochează primul și al doilea minim și
// maxim. După construcția inițială, admite interogări pe interval.
struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n;

  int next_power_of_2(int x) {
    return 1 << (32 - __builtin_clz(x - 1));
  }

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos, int val) {
    v[n + pos].set(val);
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i];
```

```
      v[i].combine(v[2 * i + 1]);
    }
  }

  segment_tree_node query(int l, int r) {
    segment_tree_node result = { +INF, +INF, -INF, -INF };

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        result.combine(v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result.combine(v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

cell list[MAX_NODES];
node nd[MAX_NODES + 1];
int dfs_order[MAX_NODES];
segment_tree st;
int n, num_queries;

void add_child(int p, int c) {
  static int ptr = 1;
  list[ptr] = { c, nd[p].adj };
  nd[p].adj = ptr++;
}

void read_tree() {
  scanf("%d %d", &n, &num_queries);

  for (int u = 2; u <= n; u++) {
    scanf("%d", &nd[u].parent);
    add_child(nd[u].parent, u);
  }
}

void dfs(int u) {
  static int time = 0;
  dfs_order[time] = u;
```

```
  st.set(u, time++);

  int u2 = nd[u].jump, u3 = nd[u2].jump;
  bool equal = (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    nd[v].depth = 1 + nd[u].depth;
    nd[v].parent = u;
    nd[v].jump = equal ? u3 : u;
    dfs(v);
  }
}

int lca(int u, int v) {
  if (nd[v].depth > nd[u].depth) {
    int tmp = u; u = v; v = tmp;
  }

  // Întîi adu-le la același nivel.
  while (nd[u].depth > nd[v].depth) {
    u = (nd[nd[u].jump].depth >= nd[v].depth) ? nd[u].jump : nd[u].parent;
  }

  while (u != v) {
    if (nd[u].jump != nd[v].jump) {
      u = nd[u].jump;
      v = nd[v].jump;
    } else {
      u = nd[u].parent;
      v = nd[v].parent;
    }
  }
  return u;
}

void query(int l, int r, int& kick, int& depth) {
  segment_tree_node m = st.query(l, r);
  int u = dfs_order[m.min1];
  int v = dfs_order[m.min2];
  int w = dfs_order[m.max2];
  int x = dfs_order[m.max1];

  kick = x;
  depth = nd[lca(u, w)].depth;

  int cand = nd[lca(v, x)].depth;
  if (cand > depth) {
    kick = u;
    depth = cand;
```

```
  }
}

void process_queries() {
  while (num_queries--) {
    int l, r, kick, depth;
    scanf("%d %d", &l, &r);
    query(l, r, kick, depth);
    printf("%d %d\n", kick, depth);
  }
}

int main() {
  read_tree();
  st.init(n + 1);
  dfs(1);
  st.build();
  process_queries();

  return 0;
}
```

I.8 Problema Duff in the Army (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_LENGTH = 10;
const int SENTINEL = 100'001; // populația maximă + 1

int min(int x, int y) {
  return (x < y) ? x : y;
}

int rbuf[2 * MAX_LENGTH]; // pentru combinarea a două liste

// Un *roster* este o listă ordonată de ID-uri de persoane.
struct roster {
  int n;
  int v[MAX_LENGTH + 1];

  void clear() {
    n = 0;
  }

  void add(int id) {
```

```
    int k = n;
    while (k && (id < v[k - 1])) {
      v[k] = v[k - 1];
      k--;
    }
    v[k] = id;
    if (n < MAX_LENGTH) {
      n++;
    }
  }

  void copy_from(roster& other) {
    n = other.n;
    for (int i = 0; i <= n; i++) {
      v[i] = other.v[i];
    }
  }

  void merge_from(roster& other, int limit) {
    v[n] = other.v[other.n] = SENTINEL;
    n = min(n + other.n, limit);
    int i = 0, j = 0;

    for (int k = 0; k < n; k++) {
      if (v[i] < other.v[j]) {
        rbuf[k] = v[i++];
      } else {
        rbuf[k] = other.v[j++];
      }
    }

    for (int i = 0; i < n; i++) {
      v[i] = rbuf[i];
    }
  }

  void print() {
    printf("%d", n);
    for (int i = 0; i < n; i++) {
      printf(" %d", v[i]);
    }
    printf("\n");
  }
};

struct cell {
  int v, next;
};

struct node {
```

```
  int adj;
  int depth;
  int time_in, time_out;
  int parent;
  int jump;

  // populația proprie
  roster ids;

  // populația până la destinația saltului, exclusiv
  roster jids;

  void compute_jump();
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int num_queries, num_people;

void node::compute_jump() {
  jids.copy_from(ids);

  int u = parent, u2 = nd[u].jump, u3 = nd[u2].jump;
  if (u3 && (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth)) {
    jump = u3;
    jids.merge_from(nd[u].jids, MAX_LENGTH);
    jids.merge_from(nd[u2].jids, MAX_LENGTH);
  } else {
    jump = u;
  }
}

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  int n, u, v;
  scanf("%d %d %d", &n, &num_people, &num_queries);

  for (int i = 0; i < n - 1; i++) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }

  for (int id = 1; id <= num_people; id++) {
    scanf("%d", &u);
```

```
    nd[u].ids.add(id);
  }
}

// Traversează arborele și calculează părinții, adîncimile, *jump pointers* și
// listele subîntinse de toți pointerii.
void dfs(int u) {
  static int time = 1;
  nd[u].time_in = time++;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].depth) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].compute_jump();
      dfs(v);
    }
  }

  nd[u].time_out = time++;
}

bool is_ancestor(int u, int v) {
  return
    (nd[u].time_in <= nd[v].time_in) &&
    (nd[u].time_out >= nd[v].time_out);
}

int climb_until_ancestor(int u, int v, int limit, roster& dest) {
  while (!is_ancestor(u, v)) {
    if (nd[u].jump && !is_ancestor(nd[u].jump, v)) {
      dest.merge_from(nd[u].jids, limit);
      u = nd[u].jump;
    } else {
      dest.merge_from(nd[u].ids, limit);
      u = nd[u].parent;
    }
  }

  return u;
}

// Urcă cu u și v pînă la LCA și colectează listele de pe drumuri.
void two_ptr_lca(int u, int v, int limit, roster& dest) {
  dest.clear();
  u = climb_until_ancestor(u, v, limit, dest);
  v = climb_until_ancestor(v, u, limit, dest);
  dest.merge_from(nd[u].ids, limit);
}
```

```
void answer_queries() {
  int u, v, limit;
  roster r;
  while (num_queries--) {
    scanf("%d %d %d", &u, &v, &limit);
    two_ptr_lca(u, v, limit, r);
    r.print();
  }
}

int main() {
  read_input_data();
  nd[1].depth = 1;
  dfs(1);
  answer_queries();

  return 0;
}
```

Anexa J

Arbori - algoritmul lui Mo pe arbore

J.1 Problema Dating (Codeforces)

înapoi

Sursă cu LCA implementat cu doi pointeri (versiune online).

```
#include <algorithm>
#include <stdio.h>
#include <unordered_map>
#include <vector>

const int MAX_NODES = 100'000;
const int MAX_QUERIES = 100'000;
const int BLOCK_SIZE = 775; // de ordinul a sqrt(2n)

struct normalizer {
  std::unordered_map<int, int> map;

  int normalize(int x) {
    auto it = map.find(x);
    if (it == map.end()) {
      int result = 1 + map.size();
      map[x] = result;
      return result;
    } else {
      return it->second;
    }
  }
};

struct node {
  std::vector<int> adj; // listă de adiacență
  int tin, tout;        // timpi de intrare și ieșire din DFS
  int parent, jump;     // pointeri pentru LCA
```

```
  int depth;
  int fav;
  bool gender;
};

struct query {
  int l, r;
  int lca;
  int orig_index;
};

node nd[MAX_NODES + 1];
int euler[2 * MAX_NODES + 1];
query q[MAX_QUERIES];
unsigned answer[MAX_QUERIES];
int n, num_queries;

void read_tree() {
  scanf("%d ", &n);
  for (int u = 1; u <= n; u++) {
    nd[u].gender = (getchar() == '1');
    getchar();
  }

  normalizer norm;
  for (int u = 1; u <= n; u++) {
    int x;
    scanf("%d", &x);
    nd[u].fav = norm.normalize(x);
  }

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
}

void dfs(int u) {
  static int time = 0;
  nd[u].tin = ++time;
  euler[time] = u;

  int u2 = nd[u].jump, u3 = nd[u2].jump;
  bool equal = (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth);

  for (auto v: nd[u].adj) {
    if (!nd[v].tin) {
      nd[v].depth = 1 + nd[u].depth;
```

```
      nd[v].parent = u;
      nd[v].jump = equal ? u3 : u;
      dfs(v);
    }
  }

  nd[u].tout = ++time;
  euler[time] = u;
}

bool is_ancestor(int u, int v) {
  return
    (nd[u].tin <= nd[v].tin) &&
    (nd[u].tout >= nd[v].tout);
}

int lca(int u, int v) {
  // Găsește cel mai jos strămoș al lui u care este și strămoș al lui v.
  while (!is_ancestor(u, v)) {
    if (nd[u].jump && !is_ancestor(nd[u].jump, v)) {
      u = nd[u].jump;
    } else {
      u = nd[u].parent;
    }
  }

  return u;
}

void read_queries() {
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    int l = lca(u, v);
    if (nd[u].tin > nd[v].tin) {
      std::swap(u, v);
    }
    if (l == u) {
      q[i] = { nd[u].tin, nd[v].tin, 0, i };
    } else {
      q[i] = { nd[u].tout, nd[v].tin, l, i };
    }
  }
}

void sort_queries_in_mo_order() {
  std::sort(q, q + num_queries, [](query a, query b) {
    int x = a.l / BLOCK_SIZE, y = b.l / BLOCK_SIZE;
    if (x != y) {
```

```
      return (x < y);
    } else if (x % 2) {
      return a.r > b.r;
    } else {
      return a.r < b.r;
    }
  });
}

struct mo_tracker {
  bool on[MAX_NODES + 1];  // nodurile care apar exact o dată în interval
  int f[MAX_NODES + 1][2]; // frecvența numerelor favorite pentru fiecare gen
  int l, r;
  unsigned num_pairs;

  void init() {
    l = 1;
    r = 0;
  }

  void toggle(int pos) {
    int u = euler[pos];
    on[u] = !on[u];
    int sign = on[u] ? +1 : -1;
    f[nd[u].fav][nd[u].gender] += sign;
    num_pairs += sign * f[nd[u].fav][!nd[u].gender];
  }

  unsigned query(int target_l, int target_r, int extra_fav, bool extra_gender) {
    while (l > target_l) {
      toggle(--l);
    }
    while (r < target_r) {
      toggle(++r);
    }
    while (l < target_l) {
      toggle(l++);
    }
    while (r > target_r) {
      toggle(r--);
    }

    if (extra_fav) {
      return num_pairs + f[extra_fav][!extra_gender];
    } else {
      return num_pairs;
    }
  }
};
```

```
mo_tracker tracker;

void answer_queries() {
  tracker.init();
  for (int i = 0; i < num_queries; i++) {
    int extra_fav = nd[q[i].lca].fav;
    int extra_gender = nd[q[i].lca].gender;
    unsigned result = tracker.query(q[i].l, q[i].r, extra_fav, extra_gender);
    answer[q[i].orig_index] = result;
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%u\n", answer[i]);
  }
}

int main() {
  read_tree();
  nd[1].depth = 1;
  dfs(1);
  read_queries();
  sort_queries_in_mo_order();
  answer_queries();
  write_answers();

  return 0;
}
```

Sursă cu LCA implementat cu algoritmul lui Tarjan (versiune online).

```
#include <algorithm>
#include <stdio.h>
#include <unordered_map>

const int MAX_NODES = 100'000;
const int MAX_QUERIES = 100'000;
const int BLOCK_SIZE = 775; // de ordinul a sqrt(2n)

struct disjoint_set_forest {
  int p[MAX_NODES + 1];

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
    }
  }
```

```
  int find(int u) {
    return (p[u] == u)
      ? u
      : (p[u] = find(p[u]));
  }

  // Întotdeauna îl alipește pe v la u. Fără *union by rank*.
  void unite(int u, int v) {
    p[find(v)] = find(u);
  }
};

struct normalizer {
  std::unordered_map<int, int> map;

  int normalize(int x) {
    auto it = map.find(x);
    if (it == map.end()) {
      int result = 1 + map.size();
      map[x] = result;
      return result;
    } else {
      return it->second;
    }
  }
};

struct cell {
  int val, next;
};

struct node {
  int adj;       // listă de adiacență
  int queries;   // listă de interogări pentru acest nod
  int tin, tout; // timpi de intrare și ieșire din DFS
  int fav;
  bool gender;
};

struct query {
  // Redenumește și refolosește cîmpurile pentru a economisi memorie.
  union { int u; int l; };
  union { int v; int r; };
  int lca;
  int orig_index;
};

cell list[2 * MAX_NODES];
cell qlist[2 * MAX_QUERIES + 1];
node nd[MAX_NODES + 1];
```

```
int euler[2 * MAX_NODES + 1];
disjoint_set_forest dsf;
query q[MAX_QUERIES];
unsigned answer[MAX_QUERIES];
int n, num_queries;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void add_query(int ind, int u) {
  static int pos = 1;
  qlist[pos] = { ind, nd[u].queries };
  nd[u].queries = pos++;
}

void read_tree() {
  scanf("%d ", &n);
  for (int u = 1; u <= n; u++) {
    nd[u].gender = (getchar() == '1');
    getchar();
  }

  normalizer norm;
  for (int u = 1; u <= n; u++) {
    int x;
    scanf("%d", &x);
    nd[u].fav = norm.normalize(x);
  }

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

void read_queries() {
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].u, &q[i].v);
    add_query(i, q[i].u);
    add_query(i, q[i].v);
    q[i].orig_index = i;
  }
}
```

```
void process_lca_queries(int u) {
  for (int ptr = nd[u].queries; ptr; ptr = qlist[ptr].next) {
    int i = qlist[ptr].val;
    int v = (q[i].u == u) ? q[i].v : q[i].u;
    if (nd[v].tin) {
      q[i].lca = dsf.find(v);
    }
  }
}

void dfs(int u) {
  static int time = 0;
  nd[u].tin = ++time;
  euler[time] = u;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].val;
    if (!nd[v].tin) {
      dfs(v);
      dsf.unite(u, v);
    }
  }

  process_lca_queries(u);

  nd[u].tout = ++time;
  euler[time] = u;
}

// Acum că știm LCA-urile și timpii Euler, calculează intervalele.
void rewrite_queries() {
  for (int i = 0; i < num_queries; i++) {
    if (nd[q[i].u].tin > nd[q[i].v].tin) {
      int tmp = q[i].u;
      q[i].u = q[i].v;
      q[i].v = tmp;
    }
    if (q[i].lca == q[i].u) {
      q[i].l = nd[q[i].u].tin;
      q[i].r = nd[q[i].v].tin;
      q[i].lca = 0;
    } else {
      q[i].l = nd[q[i].u].tout;
      q[i].r = nd[q[i].v].tin;
    }
  }
}

void sort_queries_in_mo_order() {
  std::sort(q, q + num_queries, [](query a, query b) {
```

```
    int x = a.l / BLOCK_SIZE, y = b.l / BLOCK_SIZE;
    if (x != y) {
      return (x < y);
    } else if (x % 2) {
      return a.r > b.r;
    } else {
      return a.r < b.r;
    }
  });
}

struct mo_tracker {
  bool on[MAX_NODES + 1];  // nodurile care apar exact o dată în interval
  int f[MAX_NODES + 1][2]; // frecvența numerelor favorite pentru fiecare gen
  int l, r;
  unsigned num_pairs;

  void init() {
    l = 1;
    r = 0;
  }

  void toggle(int pos) {
    int u = euler[pos];
    on[u] = !on[u];
    int sign = on[u] ? +1 : -1;
    f[nd[u].fav][nd[u].gender] += sign;
    num_pairs += sign * f[nd[u].fav][!nd[u].gender];
  }

  unsigned query(int target_l, int target_r, int extra_fav, bool extra_gender) {
    while (l > target_l) {
      toggle(--l);
    }
    while (r < target_r) {
      toggle(++r);
    }
    while (l < target_l) {
      toggle(l++);
    }
    while (r > target_r) {
      toggle(r--);
    }

    if (extra_fav) {
      return num_pairs + f[extra_fav][!extra_gender];
    } else {
      return num_pairs;
    }
  }
```

```
};

mo_tracker tracker;

void answer_queries() {
  tracker.init();
  for (int i = 0; i < num_queries; i++) {
    int extra_fav = nd[q[i].lca].fav;
    int extra_gender = nd[q[i].lca].gender;
    unsigned result = tracker.query(q[i].l, q[i].r, extra_fav, extra_gender);
    answer[q[i].orig_index] = result;
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%u\n", answer[i]);
  }
}

int main() {
  read_tree();
  read_queries();
  dsf.init(n);
  dfs(1);
  rewrite_queries();
  sort_queries_in_mo_order();
  answer_queries();
  write_answers();

  return 0;
}
```

Anexa K

Arbori - descompunere *heavy-light*

K.1 Problema Heavy Path Decomposition (Infoarena)

înapoi

Versiunea 1.

```
#include <stdio.h>

const int MAX_NODES = 1 << 17;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int val;
  int depth;
  int parent;
  int heavy;  // fiul cu subarborele maxim
  int head;   // vîrful lanțului nostru
  int tin;    // momentul descoperirii în DFS
};

node nd[MAX_NODES + 1];

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

int max(int x, int y) {
  return (x > y) ? x : y;
}
```

```
struct segment_tree {
  int v[2 * MAX_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
    for (int u = 1; u <= n; u++) {
      v[nd[u].tin + this->n] = nd[u].val;
    }
    for (int pos = this->n - 1; pos; pos--) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
    }
  }

  void update(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int rmq(int l, int r) {
    int result = -1;

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        result = max(result, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = max(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

cell list[2 * MAX_NODES];
segment_tree segtree;
int n, num_queries;
FILE *fin, *fout;

void add_edge(int u, int v) {
```

```
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_data() {
  fscanf(fin, "%d %d", &n, &num_queries);
  for (int u = 1; u <= n; u++) {
    fscanf(fin, "%d", &nd[u].val);
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Returnează mărimea subarborelui lui u.
int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      nd[v].depth = 1 + nd[u].depth;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

void decompose_dfs(int u, int head) {
  static int time = 0;

  nd[u].head = head;
  nd[u].tin = time++;

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
```

```
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v); // începe un lanț nou
    }
  }
}

int query(int u, int v) {
  int result = 0;
  while (nd[u].head != nd[v].head) {
    // Interogare de prefix pe lanțul de jos.
    if (nd[nd[v].head].depth > nd[nd[u].head].depth) {
      int tmp = u; u = v; v = tmp;
    }
    result = max(result, segtree.rmq(nd[nd[u].head].tin, nd[u].tin));
    // Saltul la părintele capului de lanț ne plasează pe un lanț nou.
    u = nd[nd[u].head].parent;
  }

  // Ultima interogare are loc pe lanțul comun.
  if (nd[u].depth > nd[v].depth) {
    int tmp = u; u = v; v = tmp;
  }
  result = max(result, segtree.rmq(nd[u].tin, nd[v].tin));

  return result;
}

void process_queries() {
  while (num_queries--) {
    int t, x, y;
    fscanf(fin, "%d %d %d", &t, &x, &y);
    if (t) {
      fprintf(fout, "%d\n", query(x, y));
    } else {
      segtree.update(nd[x].tin, y);
    }
  }
}

int main() {
  fin = fopen("heavypath.in", "r");
  fout = fopen("heavypath.out", "w");

  read_data();
  heavy_dfs(1);
  decompose_dfs(1, 1);
```

```
  segtree.init(n);
  process_queries();

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Versiunea 2.

```
#include <stdio.h>

const int MAX_NODES = 1 << 17;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int val;
  int parent;
  int heavy;  // fiul cu subarborele maxim
  int head;   // vîrful lanțului nostru
  int tin;    // momentul descoperirii în DFS
};

node nd[MAX_NODES + 1];

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct segment_tree {
  int v[2 * MAX_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
    for (int u = 1; u <= n; u++) {
      v[nd[u].tin + this->n] = nd[u].val;
    }
    for (int pos = this->n - 1; pos; pos--) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
```

```
    }
  }

  void update(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int rmq(int l, int r) {
    int result = -1;

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        result = max(result, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = max(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

cell list[2 * MAX_NODES];
segment_tree segtree;
int n, num_queries;
FILE *fin, *fout;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_data() {
  fscanf(fin, "%d %d", &n, &num_queries);
  for (int u = 1; u <= n; u++) {
    fscanf(fin, "%d", &nd[u].val);
  }
```

```
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Returnează mărimea subarborelui lui u.
int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

void decompose_dfs(int u, int head) {
  static int time = 0;

  nd[u].head = head;
  nd[u].tin = time++;

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v); // începe un lanț nou
    }
  }
}

int query(int u, int v) {
  int result = 0;
```

```
  while (nd[u].head != nd[v].head) {
    // Interogare de prefix pe lanțul de jos.
    if (nd[v].tin > nd[u].tin) {
      int tmp = u; u = v; v = tmp;
    }
    result = max(result, segtree.rmq(nd[nd[u].head].tin, nd[u].tin));
    // Saltul la părintele capului de lanț ne plasează pe un lanț nou.
    u = nd[nd[u].head].parent;
  }

  // Ultima interogare are loc pe lanțul comun.
  if (nd[u].tin > nd[v].tin) {
    int tmp = u; u = v; v = tmp;
  }
  result = max(result, segtree.rmq(nd[u].tin, nd[v].tin));

  return result;
}

void process_queries() {
  while (num_queries--) {
    int t, x, y;
    fscanf(fin, "%d %d %d", &t, &x, &y);
    if (t) {
      fprintf(fout, "%d\n", query(x, y));
    } else {
      segtree.update(nd[x].tin, y);
    }
  }
}

int main() {
  fin = fopen("heavypath.in", "r");
  fout = fopen("heavypath.out", "w");

  read_data();
  heavy_dfs(1);
  decompose_dfs(1, 1);
  segtree.init(n);
  process_queries();

  fclose(fin);
  fclose(fout);

  return 0;
}
```

K.2 Problema Disruption (USACO)

‹ înapoi

Soluție cu descompunere *heavy-light*.

```
#include <stdio.h>

const int MAX_NODES = 50'000;
const int MAX_SEGTREE_NODES = 1 << 17;
const int INFINITY = 2'000'000'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int heavy;
  int head;
  int pos;
};

struct edge {
  int u, v;
};

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

// Un arbore de intervale care suportă operațiile:
//
// 1. set(l, r, m): atribuie v[i] = min(v[i], m) pe toate pozițiile l ≤ i ≤ r.
// 2. get(i): returnează v[i].
//
// În plus, toate operațiile get() vin după toate operațiile set().
struct segment_tree {
  int v[MAX_SEGTREE_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
    for (int i = 1; i < 2 * this->n; i++) {
      v[i] = INFINITY;
```

```
    }
  }

  void set(int l, int r, int val) {
    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        v[l] = min(v[l], val);
        l++;
      }
      l >>= 1;

      if (!(r & 1)) {
        v[r] = min(v[r], val);
        r--;
      }
      r >>= 1;
    }
  }

  void push_all() {
    for (int i = 1; i < n; i++) {
      v[2 * i] = min(v[2 * i], v[i]);
      v[2 * i + 1] = min(v[2 * i + 1], v[i]);
    }
  }

  int get(int pos) {
    return v[pos + n];
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
edge e[MAX_NODES];
segment_tree segtree;
FILE* fin;
int n, num_extra_paths;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  fscanf(fin, "%d %d", &n, &num_extra_paths);
  for (int i = 0; i < n - 1; i++) {
```

```
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    e[i] = { u, v };
    add_edge(u, v);
    add_edge(v, u);
  }
}

int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

void decompose_dfs(int u, int head) {
  static int time = 0;

  nd[u].head = head;
  nd[u].pos = time++;

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v);
    }
  }
}

void set_cost_on_path(int u, int v, int cost) {
  while (nd[u].head != nd[v].head) {
    if (nd[v].pos > nd[u].pos) {
```

```
      int tmp = u; u = v; v = tmp;
    }
    segtree.set(nd[nd[u].head].pos, nd[u].pos, cost);
    u = nd[nd[u].head].parent;
  }

  if (nd[u].pos > nd[v].pos) {
    int tmp = u; u = v; v = tmp;
  }

  // Nu scrie nimic pe nodul LCA însuși.
  segtree.set(nd[u].pos + 1, nd[v].pos, cost);
}

void read_extra_paths() {
  while (num_extra_paths--) {
    int u, v, cost;
    fscanf(fin, "%d %d %d\n", &u, &v, &cost);
    set_cost_on_path(u, v, cost);
  }
}

void compute_answers() {
  FILE* fout = fopen("disrupt.out", "w");

  segtree.push_all();
  for (int i = 0; i < n - 1; i++) {
    int child = (nd[e[i].u].parent == e[i].v) ? e[i].u : e[i].v;
    int cost = segtree.get(nd[child].pos);
    cost = (cost == INFINITY) ? -1 : cost;
    fprintf(fout, "%d\n", cost);
  }

  fclose(fout);
}

int main() {
  fin = fopen("disrupt.in", "r");
  read_tree();
  heavy_dfs(1);
  decompose_dfs(1, 1);
  segtree.init(n);
  read_extra_paths();
  compute_answers();
  fclose(fin);

  return 0;
}
```

Soluție cu tehnica *small-to-large*.

```
#include <algorithm>
#include <set>
#include <stdio.h>
#include <vector>

typedef unsigned short u16;
typedef std::set<u16> set;

const int MAX_NODES = 50'000;
const int MAX_REPL_PATHS = 50'000;

struct node {
  std::vector<u16> adj;
  std::vector<u16> repl_ids;
  u16 parent;
  int min_cost;
};

struct edge {
  u16 u, v;
};

struct repl_path {
  u16 u, v;
  int cost;
};

node nd[MAX_NODES + 1];
edge e[MAX_NODES];
repl_path repl[MAX_REPL_PATHS];
int n, num_repl_paths;

void read_data() {
  FILE* f = fopen("disrupt.in", "r");

  fscanf(f, "%d %d", &n, &num_repl_paths);
  for (int i = 0; i < n - 1; i++) {
    u16 u, v;
    fscanf(f, "%hd %hd", &u, &v);
    e[i] = { u, v };
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }

  for (int i = 0; i < num_repl_paths; i++) {
    fscanf(f, "%hd %hd %d\n", &repl[i].u, &repl[i].v, &repl[i].cost);
  }
```

```
  fclose(f);
}

void sort_repl_paths() {
  std::sort(repl, repl + num_repl_paths, [](repl_path& a, repl_path& b) {
    return a.cost < b.cost;
  });
}

void distribute_repl_paths() {
  for (int i = 0; i < num_repl_paths; i++) {
    nd[repl[i].u].repl_ids.push_back(i);
    nd[repl[i].v].repl_ids.push_back(i);
  }
}

void merge_sets(set& parent, set& child) {
  if (child.size() > parent.size()) {
    parent.swap(child);
  }
  for (u16 id: child) {
    std::pair<set::iterator, bool> p = parent.insert(id);
    if (!p.second) {
      // Inserarea a eșuat deoarece ID-ul este deja în set; șterge-l.
      parent.erase(p.first);
    }
  }
  child.clear();
}

set dfs(int u) {
  set s;
  for (u16 id: nd[u].repl_ids) {
    s.insert(id);
  }

  for (u16 v: nd[u].adj) {
    if (v != nd[u].parent) {
      nd[v].parent = u;
      set s2 = dfs(v);
      merge_sets(s, s2);
    }
  }

  if (s.empty()) {
    nd[u].min_cost = -1;
  } else {
    u16 first = *s.begin();
    nd[u].min_cost = repl[first].cost;
```

```
  }

  return s;
}

void compute_answers() {
  FILE* fout = fopen("disrupt.out", "w");

  for (int i = 0; i < n - 1; i++) {
    int child = (nd[e[i].u].parent == e[i].v) ? e[i].u : e[i].v;
    int cost = nd[child].min_cost;
    fprintf(fout, "%d\n", cost);
  }

  fclose(fout);
}

int main() {
  read_data();
  sort_repl_paths();
  distribute_repl_paths();
  dfs(1);
  compute_answers();

  return 0;
}
```

Soluție cu tehnica *small-to-large* cu DFS exclusiv.

```
#include <algorithm>
#include <stdio.h>
#include <vector>

typedef unsigned short u16;

const int MAX_NODES = 50'000;
const int MAX_REPL_PATHS = 50'000;

struct node {
  std::vector<u16> adj;
  std::vector<u16> repl_ids;
  u16 parent;
  u16 heavy;
  u16 min_id;
};

struct edge {
  u16 u, v;
};
```

```
struct repl_path {
  u16 u, v;
  int cost;
};

// Un arbore Fenwick de booleeni. Admite operația „cel mai mic bit setat”.
struct fenwick_tree {
  u16 v[MAX_NODES + 1];
  bool raw[MAX_NODES + 1];
  int n;
  int max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
  }

  void add(int pos, int delta) {
    do {
      v[pos] += delta;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void toggle(int pos) {
    add(pos, raw[pos] ? -1 : +1);
    raw[pos] = !raw[pos];
  }

  // Returnează poziția primului bit setat sau n + 1 dacă arborele este gol.
  int get_first_set_bit() {
    int pos = 0;

    for (int interval = max_p2; interval; interval >>= 1) {
      if ((pos + interval <= n) && (v[pos + interval] == 0)) {
        pos += interval;
      }
    }

    return pos + 1;
  }
};

node nd[MAX_NODES + 1];
edge e[MAX_NODES];
repl_path repl[MAX_REPL_PATHS + 1];
fenwick_tree fen;
int n, num_repl;
```

```
void read_data() {
  FILE* f = fopen("disrupt.in", "r");

  fscanf(f, "%d %d", &n, &num_repl);
  for (int i = 0; i < n - 1; i++) {
    u16 u, v;
    fscanf(f, "%hd %hd", &u, &v);
    e[i] = { u, v };
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }

  for (int i = 1; i <= num_repl; i++) {
    fscanf(f, "%hd %hd %d\n", &repl[i].u, &repl[i].v, &repl[i].cost);
  }

  fclose(f);
}

void sort_repl_paths() {
  std::sort(repl + 1, repl + num_repl + 1, [](repl_path& a, repl_path& b) {
    return a.cost < b.cost;
  });
}

void distribute_repl_paths() {
  for (int i = 1; i <= num_repl; i++) {
    nd[repl[i].u].repl_ids.push_back(i);
    nd[repl[i].v].repl_ids.push_back(i);
  }
}

int size_dfs(int u) {
  int size = 1, max_c_size = 0;

  for (u16 v: nd[u].adj) {
    if (v != nd[u].parent) {
      nd[v].parent = u;
      int c = size_dfs(v);
      size += c;
      if (!nd[u].heavy || (c > max_c_size)) {
        nd[u].heavy = v;
        max_c_size = c;
      }
    }
  }

  return size;
}
```

```
void toggle_node(int u) {
  for (u16 id: nd[u].repl_ids) {
    fen.toggle(id);
  }
}

void toggle_subtree(int u) {
  toggle_node(u);
  for (int v: nd[u].adj) {
    if (v != nd[u].parent) {
      toggle_subtree(v);
    }
  }
}

void dfs(int u) {
  for (u16 v: nd[u].adj) {
    if ((v != nd[u].parent) && (v != nd[u].heavy)) {
      dfs(v);
      toggle_subtree(v);
    }
  }

  if (nd[u].heavy) {
    dfs(nd[u].heavy);
  }

  for (u16 v: nd[u].adj) {
    if ((v != nd[u].parent) && (v != nd[u].heavy)) {
      toggle_subtree(v);
    }
  }

  toggle_node(u);
  nd[u].min_id = fen.get_first_set_bit();
}

void compute_answers() {
  FILE* fout = fopen("disrupt.out", "w");

  for (int i = 0; i < n - 1; i++) {
    int child = (nd[e[i].u].parent == e[i].v) ? e[i].u : e[i].v;
    int id = nd[child].min_id;
    int cost = (id <= num_repl) ? repl[id].cost : -1;
    fprintf(fout, "%d\n", cost);
  }

  fclose(fout);
}
```

```
int main() {
  read_data();
  sort_repl_paths();
  distribute_repl_paths();
  fen.init(num_repl);
  size_dfs(1);
  dfs(1);
  compute_answers();

  return 0;
}
```

K.3 Problema Rafaela (Lot 2014)

înapoi

Soluție cu HLD (versiune online).

```
// Complexitate: O(q log^2 n).
// Metodă: descompunere heavy-light.
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_SEGTREE_NODES = 1 << 19;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int heavy;      // fiul cu cel mai mare subarbore
  int head;       // începutul lanțului nostru
  int tin, tout;  // momentele intrării și ieșirii din DFS
  int init_pop;   // populația inițială
};

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct segtree_node {
  int m;    // maximul din intervalul subîntins
```

```
  int lazy; // sumă de adăugat la tot intervalul
  // Invariant: pentru nodul curent, m include lazy
};

// Arbore de segmente cu adăugare pe interval și maxim pe interval.
struct segment_tree {
  segtree_node v[MAX_SEGTREE_NODES];
  int n, bits;

  void init(int size) {
    n = next_power_of_2(size);
    bits = __builtin_popcount(n - 1);
  }

  void raw_set(int pos, int val) {
    v[pos + n].m = val;
  }

  void build() {
    for (int pos = n - 1; pos; pos--) {
      v[pos].m = max(v[2 * pos].m, v[2 * pos + 1].m);
    }
  }

  void push(int pos) {
    if (v[pos].lazy) {
      v[2 * pos].m += v[pos].lazy;
      v[2 * pos].lazy += v[pos].lazy;
      v[2 * pos + 1].m += v[pos].lazy;
      v[2 * pos + 1].lazy += v[pos].lazy;
      v[pos].lazy = 0;
    }
  }

  void push_path(int pos) {
    for (int b = bits; b; b--) {
      push(pos >> b);
    }
  }

  void pull_path(int pos) {
    for (pos /= 2; pos; pos /= 2) {
      if (!v[pos].lazy) {
        v[pos].m = max(v[2 * pos].m, v[2 * pos + 1].m);
      }
    }
  }

  void range_add(int l, int r, int val) {
    l += n;
```

```
    r += n;
    int orig_l = l, orig_r = r;

    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        v[l].m += val;
        v[l++].lazy += val;
      }
      l >>= 1;

      if (!(r & 1)) {
        v[r].m += val;
        v[r--].lazy += val;
      }
      r >>= 1;
    }

    pull_path(orig_l);
    pull_path(orig_r);
  }

  int range_max(int l, int r) {
    int result = 0;

    l += n;
    r += n;
    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        result = max(result, v[l++].m);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = max(result, v[r--].m);
      }
      r >>= 1;
    }

    return result;
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
```

```
segment_tree segtree;
FILE *fin, *fout;
int n, num_queries, total_pop;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  fscanf(fin, "%d %d", &n, &num_queries);
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }

  for (int u = 1; u <= n; u++) {
    fscanf(fin, "%d", &nd[u].init_pop);
    total_pop += nd[u].init_pop;
  }
}

int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      nd[u].init_pop += nd[v].init_pop;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

void decompose_dfs(int u, int head) {
  static int time = 0;
```

```
  nd[u].head = head;
  nd[u].tin = time++;
  segtree.raw_set(nd[u].tin, nd[u].init_pop);

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v); // la v începe un lanț nou
    }
  }

  nd[u].tout = time - 1;
}

void update(int u, int delta) {
  total_pop += delta;
  while (u) {
    int h = nd[u].head;
    segtree.range_add(nd[h].tin, nd[u].tin, delta);
    u = nd[h].parent;
  }
}

int query(int u) {
  int size_of_u = segtree.range_max(nd[u].tin, nd[u].tin);
  int from_parent = total_pop - size_of_u;
  int from_any_child = segtree.range_max(nd[u].tin + 1, nd[u].tout);
  return max(from_parent, from_any_child);
}

void process_ops() {
  char type;
  int u, delta;

  while (num_queries--) {
    fscanf(fin, " %c", &type);
    if (type == 'U') {
      fscanf(fin, "%d %d ", &delta, &u);
      update(u, delta);
    } else {
      fscanf(fin, "%d ", &u);
      fprintf(fout, "%d\n", query(u));
    }
  }
}
```

```
int main() {
  fin = fopen("rafaela.in", "r");
  fout = fopen("rafaela.out", "w");

  read_tree();
  segtree.init(n);
  heavy_dfs(1);
  decompose_dfs(1, 1);
  segtree.build();
  process_ops();

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Soluție cu HLD și `multiset` (versiune online).

```
// Complexitate: O(q log^2 n).
// Metodă: descompunere heavy-light + multiset.
#include <set>
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int heavy;      // fiul cu cel mai mare subarbore
  int head;       // începutul lanțului nostru
  int pos;        // momentul descoperirii în dfs
  int init_pop;   // populația inițială
  std::multiset<int> set;
};

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct fenwick_tree {
  int v[MAX_NODES + 1];
  int n;

  void init(int n) {
```

```
    this->n = n;
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void range_add(int l, int r, int val) {
    add(l, val);
    add(r + 1, -val);
  }

  void point_add(int pos, int val) {
    range_add(pos, pos, val);
  }

  int get(int pos) {
    long long s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
fenwick_tree fen;
FILE *fin, *fout;
int n, num_queries, total_pop;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  fscanf(fin, "%d %d", &n, &num_queries);
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
```

```
  for (int u = 1; u <= n; u++) {
    fscanf(fin, "%d", &nd[u].init_pop);
    total_pop += nd[u].init_pop;
  }
}

int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      nd[u].init_pop += nd[v].init_pop;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

void decompose_dfs(int u, int head) {
  static int time = 1;

  nd[u].head = head;
  nd[u].pos = time++;
  fen.point_add(nd[u].pos, nd[u].init_pop);

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v); // la v începe un lanț nou
    }
  }
}

void update_parent_set(int h, int delta) {
  int p = nd[h].parent;
  std::multiset<int>& s = nd[p].set; // syntactic sugar
```

```
  int old = fen.get(nd[h].pos);
  auto it = s.find(old);
  if (it != s.end()) {
    s.erase(it);
  }
  s.insert(old + delta);
}

void update(int u, int delta) {
  total_pop += delta;
  while (u) {
    int h = nd[u].head;
    update_parent_set(h, delta);
    fen.range_add(nd[h].pos, nd[u].pos, delta);
    u = nd[h].parent;
  }
}

int query(int u) {
  int size_of_u = fen.get(nd[u].pos);
  int max_flow = total_pop - size_of_u;

  if (nd[u].heavy) {
    int from_heavy_child = fen.get(nd[nd[u].heavy].pos);
    max_flow = max(max_flow, from_heavy_child);
  }

  if (!nd[u].set.empty()) {
    max_flow = max(max_flow, *nd[u].set.rbegin());
  }

  return max_flow;
}

void process_ops() {
  char type;
  int u, delta;

  while (num_queries--) {
    fscanf(fin, " %c", &type);
    if (type == 'U') {
      fscanf(fin, "%d %d ", &delta, &u);
      update(u, delta);
    } else {
      fscanf(fin, "%d ", &u);
      fprintf(fout, "%d\n", query(u));
    }
  }
}
```

```
int main() {
  fin = fopen("rafaela.in", "r");
  fout = fopen("rafaela.out", "w");

  read_tree();
  fen.init(n);
  heavy_dfs(1);
  decompose_dfs(1, 1);
  process_ops();

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Soluție cu descompunere în radical după interogări (versiune online).

```
// Metodă: descompunere în radical după interogări.
#include <algorithm>
#include <stdio.h>
#include <unordered_set>
#include <vector>

const int MAX_NODES = 200'000;
const int MAX_QUERIES = 200'000;
const int BLOCK_SIZE = 3'000;

struct query {
  int id;             // ordinea de la intrare
  int outside_delta;  // modificări în afara subarborelui
  int child_delta;    // modificări la fiul curent
  int max_touched;    // maximul dintre fiii anteriori

  query(int id) {
    this->id = id;
    outside_delta = 0;
    child_delta = 0;
    max_touched = 0;
  }
};

struct node {
  std::vector<int> adj;
  std::vector<query> q; // interogări (din blocul curent)
  int parent;
  int tin, tout;        // timpii de intrare/ieșire din DFS
  int pop, spop;        // populația din nod, respectiv din subarbore
};
```

```
struct update {
  int id, u, delta;
};

node nd[MAX_NODES + 1];
update updates[BLOCK_SIZE];
int answer[MAX_QUERIES + 1];
std::unordered_set<int> nodes_with_queries;
int n, num_ops, num_updates;
FILE* fin;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

void read_data() {
  fscanf(fin, "%d %d", &n, &num_ops);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }

  for (int u = 1; u <= n; u++) {
    fscanf(fin, "%d", &nd[u].pop);
  }
}

// Calculează timpii DFS și părinții
void initial_dfs(int u) {
  static int time = 0;
  nd[u].tin = time++;

  for (int v: nd[u].adj) {
    if (v != nd[u].parent) {
      nd[v].parent = u;
      initial_dfs(v);
    }
  }

  nd[u].tout = time - 1;
}
```

```
void read_ops(int start, int end) {
  char type;
  int u, delta;

  num_updates = 0;
  for (int id = start; id < end; id++) {
    fscanf(fin, " %c", &type);
    if (type == 'U') {
      fscanf(fin, "%d %d ", &delta, &u);
      updates[num_updates++] = { id, u, delta };
      // vor intra în vigoare în următorul bloc
      nd[u].pop += delta;
    } else {
      fscanf(fin, "%d ", &u);
      nd[u].q.push_back(query(id));
      nodes_with_queries.insert(u);
    }
  }
}

void sort_updates_by_dfs_time() {
  std::sort(updates, updates + num_updates, [](update& a, update& b) {
    return nd[a.u].tin < nd[b.u].tin;
  });
}

void dfs(int u) {
  nd[u].spop = nd[u].pop;

  for (int v: nd[u].adj) {
    if (v != nd[u].parent) {
      dfs(v);
      nd[u].spop += nd[v].spop;
    }
  }
}

// Notifică-l pe u că există actualizarea upd în afara subarborelui său.
void process_outside_update(int u, update& upd) {
  for (query& q: nd[u].q) {
    if (upd.id < q.id) {
      q.outside_delta += upd.delta;
    }
  }
}

// Notifică-l pe u că există actualizarea upd în fiul său curent.
void process_child_update(int u, update& upd) {
  for (query& q: nd[u].q) {
    if (upd.id < q.id) {
```

```
      q.child_delta += upd.delta;
    }
  }
}

int merge_init(int u) {
  int i = 0;

  while ((i < num_updates) && (nd[updates[i].u].tin < nd[u].tin)) {
    process_outside_update(u, updates[i++]);
  }

  while ((i < num_updates) && (updates[i].u == u)) {
    // Doar le sărim. Populația din u nu influențează interogările din u.
    i++;
  }

  return i;
}

int merge_advance(int u, int v, int i) {
  while ((i < num_updates) && (nd[updates[i].u].tout <= nd[v].tout)) {
    process_child_update(u, updates[i++]);
  }

  for (query& q: nd[u].q) {
    q.max_touched = max(q.max_touched, nd[v].spop + q.child_delta);
    q.child_delta = 0;
  }

  return i;
}

void merge_finish(int u, int i) {
  while (i < num_updates) {
    process_outside_update(u, updates[i++]);
  }
}

void answer_queries(int u, int max_untouched) {
  for (query& q: nd[u].q) {
    // fluxul pe muchia dinspre părinte
    answer[q.id] = nd[1].spop + q.outside_delta - nd[u].spop;

    // maximul fiilor afectați de actualizări
    answer[q.id] = max(answer[q.id], q.max_touched);

    // maximul fiilor neafectați de actualizări
    answer[q.id] = max(answer[q.id], max_untouched);
  }
```

```
  nd[u].q.clear();
}

void process_queries(int u) {
  int max_untouched = 0;
  int i = merge_init(u);

  for (int v: nd[u].adj) {
    if (v != nd[u].parent) {
      int old_i = i;
      i = merge_advance(u, v, i);

      if (i == old_i) {
        max_untouched = max(max_untouched, nd[v].spop);
      }
    }
  }

  merge_finish(u, i);
  answer_queries(u, max_untouched);
}

void process_nodes_with_queries() {
  for (int u: nodes_with_queries) {
    process_queries(u);
  }
  nodes_with_queries.clear();
}

void process_ops() {
  for (int start = 0; start < num_ops; start += BLOCK_SIZE) {
    int end = min(start + BLOCK_SIZE, num_ops);
    dfs(1);
    read_ops(start, end);
    sort_updates_by_dfs_time();
    process_nodes_with_queries();
  }
}

void write_answers() {
  FILE* f = fopen("rafaela.out", "w");

  for (int i = 0; i < num_ops; i++) {
    if (answer[i]) {
      fprintf(f, "%d\n", answer[i]);
    }
  }

  fclose(f);
```

```
}

int main() {
  fin = fopen("rafaela.in", "r");

  read_data();
  initial_dfs(1);
  process_ops();

  fclose(fin);

  write_answers();

  return 0;
}
```

K.4 Problema Doi arbori (Lot 2025)

înapoi

Soluție cu descompunere *heavy-light* (versiune online).

```
// Complexitate: O(q log^2 n).
// Metodă: Descompunere heavy-light.
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_SEGTREE_NODES = 1 << 19;
const int INFINITY = 3 * MAX_NODES;
const int OP_TOGGLE = 1;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int heavy;       // fiul cu subarborele maxim
  int head;        // începutul lanțului nostru
  int light_start; // începutul restului subarborelui, după lanțul greu
  int d;           // adîncimea
  int tin, tout;   // timpii DFS
};

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}
```

```
int min(int x, int y) {
  return (x < y) ? x : y;
}

struct segment_tree {
  int v[MAX_SEGTREE_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
    for (int i = 1; i < 2 * this->n; i++) {
      v[i] = INFINITY;
    }
  }

  void set(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos >>= 1; pos; pos >>= 1) {
      v[pos] = min(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int get(int pos) {
    return v[pos + n];
  }

  int get_min(int l, int r) {
    int result = INFINITY;
    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        result = min(result, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = min(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }

  int get_min_plus(int l, int r, int delta) {
    int x = get_min(l, r);
```

```
    return (x == INFINITY) ? INFINITY : (x + delta);
  }
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
segment_tree seg, seg_diff, seg_2diff;
int n, num_ops;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  scanf("%d %d", &n, &num_ops);
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      nd[v].d = 1 + nd[u].d;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

void decompose_dfs(int u, int head) {
  static int time = 0;
```

```
  nd[u].head = head;
  nd[u].tin = time++;

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
  }
  nd[u].light_start = time;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v); // începi un lanț nou
    }
  }

  nd[u].tout = time - 1;
}

void init_segment_trees() {
  seg.init(n);
  seg_diff.init(n);
  seg_2diff.init(n);
}

void toggle(int u) {
  int was_off = seg.get(nd[u].tin) == INFINITY;
  int val = was_off ? nd[u].d : INFINITY;
  seg.set(nd[u].tin, val);

  while (u) {
    seg_diff.set(nd[u].tin, (val == INFINITY) ? INFINITY : (val - nd[u].d));
    seg_2diff.set(nd[u].tin, (val == INFINITY) ? INFINITY : (val - 2 * nd[u].d));
    u = nd[nd[u].head].parent;
    if (u) {
      val = seg.get_min(nd[u].light_start, nd[u].tout);
    }
  }
}

// Găsește distanța minimă până la o frunză urcînd din u.
int up_query(int u) {
  int leaf_dist = INFINITY;
  int orig_u = u;

  while (u) {
    int h = nd[u].head;
    int on_path_h_u = seg_2diff.get_min_plus(nd[h].tin, nd[u].tin, nd[orig_u].d);
    int on_path_below_u = seg.get_min_plus(nd[u].tin, nd[u].tout, nd[orig_u].d - 2 * nd[u].d);
    leaf_dist = min(leaf_dist, on_path_h_u);
    leaf_dist = min(leaf_dist, on_path_below_u);
```

```
    u = nd[h].parent;
  }

  return leaf_dist;
}

// Găsește distanța în jos pînă la o frunză de la orice nod de pe lanțul greu
// u-v.
int path_score(int u, int v) {
  int on_path_u_v = seg_diff.get_min(nd[u].tin, nd[v].tin);
  int in_subtree_of_v = seg.get_min_plus(nd[v].tin, nd[v].tout, -nd[v].d);
  return min(on_path_u_v, in_subtree_of_v);
}

int query(int u, int v) {
  int orig_u = u, orig_v = v;

  int leaf_dist = INFINITY;
  // Urcă-l pe u sau v cît timp sînt pe lanțuri diferite.
  while (nd[u].head != nd[v].head) {
    if (nd[v].tin > nd[u].tin) {
      int tmp = u; u = v; v = tmp;
    }
    int h = nd[u].head;
    leaf_dist = min(leaf_dist, path_score(h, u));
    u = nd[h].parent;
  }

  // O ultimă operație cînd u și v se află pe același lanț.
  if (nd[u].tin > nd[v].tin) {
    int tmp = u; u = v; v = tmp;
  }
  leaf_dist = min(leaf_dist, path_score(u, v));

  leaf_dist = min(leaf_dist, up_query(u));

  if (leaf_dist == INFINITY) {
    return -1;
  } else {
    int dist_uv = nd[orig_u].d + nd[orig_v].d - 2 * nd[u].d;
    return dist_uv + 2 * leaf_dist;
  }
}

void process_ops() {
  int type, u, v;

  while (num_ops--) {
    scanf("%d %d", &type, &u);
    if (type == OP_TOGGLE) {
```

```
      toggle(u);
    } else {
      scanf("%d", &v);
      printf("%d\n", query(u, v));
    }
  }
}

int main() {
  read_tree();
  heavy_dfs(1);
  decompose_dfs(1, 1);
  init_segment_trees();
  process_ops();

  return 0;
}
```

Soluție parțială cu descompunere în radical (versiune online).

```
// Complexitate: O((n + q) sqrt q).
// Metodă: Descompunere în radical după interogări.
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_OPS = 200'000;
const int MAX_LOG = 19;
const int BLOCK_SIZE = 2'000; // determinat experimental
const int INFINITY = 2 * MAX_NODES;

const int OP_TOGGLE = 1;
const int OP_QUERY = 2;

struct cell {
  int v, next;
};

cell list[2 * MAX_NODES];

struct node {
  int adj;
  int depth;

  // binary lifting cu doi pointeri per nod
  int parent, jump;

  int dist; // distanța până la cea mai apropiată frunză safe

  // distanța până la cea mai apropiată frunză safe de la orice nod subîntins
```

```
  // de jump pointer (excluzînd nodul destinație)
  int jdist;

  bool active;
  bool safe;
};

struct operation {
  int type, u, v;
};

// O simplă coadă.
struct queue {
  int v[MAX_NODES];
  int head, tail;

  void init() {
    head = tail = 0;
  }

  void enqueue(int u) {
    v[tail++] = u;
  }

  int dequeue() {
    return v[head++];
  }

  bool is_empty() {
    return head == tail;
  }
};

// O colecție care menține valori distincte între 1 și MAX_NODES și admite
// adăugări și ștergeri în O(1) și iterarea prin valori în O(nr. de valori).
struct tracker {
  int size;
  int t[BLOCK_SIZE];
  int pos[MAX_NODES + 1]; // BLOCK_SIZE = absent

  void init(int max_val) {
    for (int val = 1; val <= max_val; val++) {
      pos[val] = BLOCK_SIZE;
    }
    size = 0;
  }

  void clear() {
    for (int i = 0; i < size; i++) {
      pos[t[i]] = BLOCK_SIZE;
```

```
    }
    size = 0;
  }

  void add(int val) {
    pos[val] = size;
    t[size++] = val;
  }

  void remove(int val) {
    int p = pos[val];
    t[p] = t[size - 1]; // move last element in place of val
    pos[t[p]] = p;
    pos[val] = BLOCK_SIZE;
    size--;
  }

  void toggle(int val) {
    if (pos[val] == BLOCK_SIZE) {
      add(val);
    } else {
      remove(val);
    }
  }
};

node nd[MAX_NODES + 1];
operation op[MAX_OPS];
tracker unsafe;
queue q;
int n, num_ops;

int log(int x) {
  return 31 - __builtin_clz(x);
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

// Liniarizare cu RMQ, cu care putem răspunde la interogări de LCA în O(1).
struct lca_info {
  int d[MAX_LOG][2 * MAX_NODES];
  int tout[MAX_NODES + 1];
  int n; // lungimea liniarizării

  void append(int u) {
    tout[u] = n;
    d[0][n++] = u;
  }
```

```
  void build_rmq_table() {
    for (int l = 1; (1 << l) <= n; l++) {
      for (int i = 0; i <= n - (1 << l); i++) {
        int r = i + (1 << (l - 1));
        d[l][i] = (nd[d[l - 1][i]].depth < nd[d[l - 1][r]].depth)
          ? d[l - 1][i]
          : d[l - 1][r];
      }
    }
  }

  int lca(int u, int v) {
    int left = min(tout[u], tout[v]);
    int right = tout[u] + tout[v] - left;
    int bits = log(right - left + 1);
    int x = d[bits][left];
    int y = d[bits][right - (1 << bits) + 1];
    return (nd[x].depth < nd[y].depth) ? x : y;
  }

  int dist(int u, int v) {
    int l = lca(u, v);
    return nd[u].depth + nd[v].depth - 2 * nd[l].depth;
  }
};

lca_info l;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_data() {
  scanf("%d %d", &n, &num_ops);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }

  for (int i = 0; i < num_ops; i++) {
    scanf("%d %d", &op[i].type, &op[i].u);
    if (op[i].type == OP_QUERY) {
      scanf("%d", &op[i].v);
    }
```

```
  }
}

// Calculează părinți, adîncimi, jump pointers și liniarizarea.
void initial_dfs(int u) {
  int u2 = nd[u].jump, u3 = nd[u2].jump;
  bool equal = (nd[u2].depth - nd[u].depth == nd[u3].depth - nd[u2].depth);
  l.append(u);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {
      nd[v].depth = 1 + nd[u].depth;
      nd[v].parent = u;
      nd[v].jump = equal ? u3 : u;
      initial_dfs(v);
      l.append(u);
    }
  }
}

// Colecteaza un bloc cu tel mult BLOCK_SIZE operații toggle(). Marchează ca
// safe frunzele care sînt active acum și care nu vor fi modificate în bloc.
// Inițializează trackerul cu frunzele unsafe, dar care încep ca active.
// Returnează finalul blocului.
int classify_leaves(int start) {
  int num_toggles = 0;
  int end = start;

  unsafe.clear();
  for (int u = 1; u <= n; u++) {
    nd[u].safe = nd[u].active;
  }

  while ((end < num_ops) && (num_toggles < BLOCK_SIZE)) {
    if (op[end].type == OP_TOGGLE) {
      int u = op[end].u;
      if (nd[u].safe) {
        // Frunza este unsafe, dar începe ca activă.
        unsafe.add(u);
        nd[u].safe = false;
      }
      num_toggles++;
    }
    end++;
  }

  return end;
}
```

```
// Recalculează câmpul dist pentru toate nodurile.
void bfs_from_safe_leaves() {
  q.init();

  for (int u = 1; u <= n; u++) {
    if (nd[u].safe) {
      nd[u].dist = 0;
      q.enqueue(u);
    } else {
      nd[u].dist = INFINITY;
    }
  }

  while (!q.is_empty()) {
    int u = q.dequeue();
    for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
      int v = list[ptr].v;
      if (nd[v].dist == INFINITY) {
        nd[v].dist = nd[u].dist + 1;
        q.enqueue(v);
      }
    }
  }
}

// Recalculează câmpul jdist pentru toate nodurile.
void jdist_dfs(int u) {
  nd[u].jdist = nd[u].dist;

  int p = nd[u].parent;
  if (nd[u].jump != p) {
    int p2 = nd[p].jump;
    nd[u].jdist = min(nd[u].jdist, nd[p].jdist);
    nd[u].jdist = min(nd[u].jdist, nd[p2].jdist);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != p) {
      jdist_dfs(v);
    }
  }
}

// Returnează finalul blocului.
int preprocess_block(int start) {
  int end = classify_leaves(start);
  bfs_from_safe_leaves();
  jdist_dfs(1);
```

```
  return end;
}

// Returnează distanța minimă la o frunză safe de la orice nod de pe calea [u,
// ancestor].
int upwards_path_min(int u, int ancestor) {
  int result = nd[ancestor].dist;

  while (u != ancestor) {
    if (nd[u].jump && (nd[nd[u].jump].depth >= nd[ancestor].depth)) {
      result = min(result, nd[u].jdist);
      u = nd[u].jump;
    } else {
      result = min(result, nd[u].dist);
      u = nd[u].parent;
    }
  }

  return result;
}

int query(int u, int v) {
  // Rezultatul vizitării unei frunze safe.
  int w = l.lca(u, v);
  int leaf_dist = min(upwards_path_min(u, w), upwards_path_min(v, w));
  int dist_uv = nd[u].depth + nd[v].depth - 2 * nd[w].depth;
  int result = 2 * leaf_dist + dist_uv;

  // Rezultatul vizitării unei frunze unsafe.
  for (int i = 0; i < unsafe.size; i++) {
    int leaf = unsafe.t[i];
    result = min(result, l.dist(u, leaf) + l.dist(leaf, v));
  }

  return (result < INFINITY) ? result : -1;
}

void toggle(int u) {
  unsafe.toggle(u);
  nd[u].active = !nd[u].active;
}

void process_ops() {
  int block_end = 0;

  for (int i = 0; i < num_ops; i++) {
    if (i == block_end) {
      block_end = preprocess_block(i);
    }
    if (op[i].type == OP_TOGGLE) {
```

```
      toggle(op[i].u);
    } else {
      printf("%d\n", query(op[i].u, op[i].v));
    }
  }
}

int main() {
  read_data();
  initial_dfs(1);
  l.build_rmq_table();
  unsafe.init(n);
  process_ops();

  return 0;
}
```

K.5 Problema Query on a Tree VI (CodeChef)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;
const int BLACK = 0;
const int WHITE = 1;
const int T_QUERY = 0;
const int T_TOGGLE = 1;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int heavy;  // fiul cu cel mai mare subarbore
  int head;   // începutul lanțului nostru
  int pos;    // timpul de descoperire în DFS / poziția în liniarizare
  int size;   // mărimea subarborelui (indiferent de culoare)
  bool color;
};

// Un AIB cu adăugare pe interval și interogare punctuală.
struct range_add_fenwick_tree {
  int v[MAX_NODES + 2];
  int n;
```

```
  void init(int n) {
    this->n = n;
  }

  void add(int pos, int delta) {
    do {
      v[pos] += delta;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void range_add(int l, int r, int delta) {
    add(l, delta);
    add(r + 1, -delta);
  }

  void point_add(int pos, int delta) {
    range_add(pos, pos, delta);
  }

  int point_query(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }
};

// Un AIB de booleeni cu suport pentru căutări binare.
struct bool_fenwick_tree {
  int v[MAX_NODES + 1];
  int n;
  int max_p2;

  void init(int n, bool val) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));

    for (int i = 1; i <= n; i++) {
      v[i] += val;
      int j = i + (i & -i);
      if (j <= n) {
        v[j] += v[i];
      }
    }
  }

  void add(int pos, int delta) {
```

```
    do {
      v[pos] += delta;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void set(int pos) {
    add(pos, +1);
  }

  void clear(int pos) {
    add(pos, -1);
  }

  int prefix_sum(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }

  int last_occurrence_of_partial_sum(int sum) {
    int pos = 0;

    for (int interval = max_p2; interval; interval >>= 1) {
      if ((pos + interval <= n) && (v[pos + interval] < sum)) {
        sum -= v[pos + interval];
        pos += interval;
      }
    }

    return pos;
  }

  // Returnează poziția ultimului 1 pe o poziție ≤ pos.
  int last_one_before(int pos) {
    int s = prefix_sum(pos);
    if (s) {
      return 1 + last_occurrence_of_partial_sum(s);
    } else {
      return 0;
    }
  }
};

cell list[2 * MAX_NODES];
range_add_fenwick_tree size[2];
bool_fenwick_tree color[2];
```

```
node nd[MAX_NODES + 1];
int hld_order[MAX_NODES + 1];
int n;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  scanf("%d", &n);
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

void init_fenwick_trees() {
  size[BLACK].init(n);
  size[WHITE].init(n);
  color[BLACK].init(n, 1);
  color[WHITE].init(n, 0);
}

void heavy_dfs(int u) {
  nd[u].size = 1;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      heavy_dfs(v);
      nd[u].size += nd[v].size;
      if (!nd[u].heavy || (nd[v].size > nd[nd[u].heavy].size)) {
        nd[u].heavy = v;
      }

    }
  }
}

void decompose_dfs(int u, int head) {
  static int time = 1;

  nd[u].head = head;
  size[BLACK].point_add(time, nd[u].size);
```

```
  size[WHITE].point_add(time, 1);
  hld_order[time] = u;
  nd[u].pos = time++;

  if (nd[u].heavy) {
    decompose_dfs(nd[u].heavy, head);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && v != nd[u].heavy) {
      decompose_dfs(v, v); // începe un lanț nou
    }
  }
}

// Returnează cel mai depărtat strămoș w pe același lanț greu cu u astfel
// încît (1) toate nodurile de la w la u inclusiv au aceeași culoare și (2)
// părintele lui w are o culoare diferită. Notă: părintele lui w poate fi pe
// lanțul următor. Returnează 0 dacă w nu există.
int get_top_same_color(int u) {
  int start = nd[nd[u].head].pos;
  int pos_1 = color[!nd[u].color].last_one_before(nd[u].pos);
  if (pos_1 >= start) {
    // Există un nod de culoare opusă pe lanț.
    return hld_order[pos_1 + 1];
  }

  int next_path = nd[nd[u].head].parent;
  if (nd[next_path].color != nd[u].color) {
    // Părintele capului de lanț are culoarea opusă.
    return nd[u].head;
  }

  return 0;
}

int query(int u) {
  nd[0].color = !nd[u].color; // santinelă

  int t;
  while (u && (t = get_top_same_color(u)) == 0) {
    u = nd[nd[u].head].parent;
  }

  return size[nd[t].color].point_query(nd[t].pos);
}

void path_update(int u, int delta1, int delta2) {
  int c = nd[u].color;
```

```
  size[!c].point_add(nd[u].pos, delta1);

  // Toți strămoșii câștigă/pierd cîtă vreme au culoarea părintelui.
  int t;
  while (u && (t = get_top_same_color(u)) == 0) {
    size[c].range_add(nd[nd[u].head].pos, nd[u].pos, delta2);
    u = nd[nd[u].head].parent;
  }

  if (u) {
    size[c].range_add(nd[t].pos, nd[u].pos, delta2);

    // Și primul strămoș de culoarea opusă câștigă/pierde.
    t = nd[t].parent;
    if (t) {
      size[c].point_add(nd[t].pos, delta2);
    }
  }
}

void toggle(int u) {
  int old = nd[u].color;
  nd[u].color = !old;

  color[old].clear(nd[u].pos);
  color[!old].set(nd[u].pos);

  int p = nd[u].parent;
  if (p) {
    int lose = size[old].point_query(nd[u].pos);
    int gain = size[!old].point_query(nd[u].pos);
    if (nd[u].color == nd[p].color) {
      path_update(p, -lose, gain);
    } else {
      path_update(p, gain, -lose);
    }
  }
}

void process_ops() {
  int num_queries, type, u;
  scanf("%d", &num_queries);
  while (num_queries--) {
    scanf("%d %d", &type, &u);
    if (type == T_QUERY) {
      printf("%d\n", query(u));
    } else {
      toggle(u);
    }
  }
```

```
}

int main() {
  read_tree();
  init_fenwick_trees();
  heavy_dfs(1);
  decompose_dfs(1, 1);
  process_ops();

  return 0;
}
```

K.6 Problema Adă caii (Lot 2025)

‹ înapoi • versiune online

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

const int MAX_NODES = 100'000;
const int MAX_PATH = 1'800; // determinat experimental

typedef std::unordered_set<int> hash_set;

// Un buffer circular de n elemente care operează într-o zonă de memorie
// alocată extern.
struct circ_buf {
  int* v;
  int n;
  int offset; // Conținutul real al vectorului este v[offset]...v[offset-1].
  int num_nonzero;

  void init(int* _v, int _n) {
    v = _v;
    n = _n;
    offset = 0;
    num_nonzero = 0;

    for (int i = 0; i < n; i++) {
      v[i] = 0;
    }
  }

  bool is_empty() {
    return num_nonzero == 0;
  }

  int get(int pos) {
```

```
  return v[(pos + offset) % n];
}

void set(int pos, int val) {
  pos = (pos + offset) % n;
  num_nonzero -= (v[pos] != 0);
  v[pos] = val;
  num_nonzero += (v[pos] != 0);
}

void add(int pos, int delta) {
  set(pos, get(pos) + delta);
}

// Shiftează (conceptual) toate elementele spre stînga. Inserează un zero la
// coadă. Returnează elementul care a ieșit din buffer.
int shift_left() {
  int leftmost = v[offset];
  num_nonzero -= (leftmost != 0);
  v[offset] = 0;
  offset = (offset + 1) % n;
  return leftmost;
}

int shift_right() {
  offset = (offset + n - 1) % n;
  int rightmost = v[offset];
  num_nonzero -= (rightmost != 0);
  v[offset] = 0;
  return rightmost;
}

// Mută naiv elementele din stînga lui pos cu 1 mai la dreapta. Mută
// eficient elementele din dreapta lui pos cu 1 mai la stînga. Returnează
// valoarea inițială de pe poziția pos.
int attract_left(int pos) {
  if (pos == 0) {
    return shift_left();
  }

  int extracted = get(pos);

  add(pos + 1, get(pos - 1));
  for (int i = pos; i >= 2; i--) {
    set(i, get(i - 2));
  }
  set(1, 0);
  shift_left();

  return extracted;
```

```
  }

  int attract_right(int pos) {
    if (pos == n - 1) {
      return shift_right();
    }

    int extracted = get(pos);

    add(pos - 1, get(pos + 1));
    for (int i = pos; i + 2 < n; i++) {
      set(i, get(i + 2));
    }
    set(n - 2, 0);
    shift_right();

    return extracted;
  }

  int attract(int pos) {
    return (2 * pos < n)
      ? attract_left(pos)
      : attract_right(pos);
  }
};

struct cell {
  int v, next;
};

struct node {
  int adj;
  int parent;
  int heavy;   // fiul cu cel mai mare subarbore
  int head;    // capătul lanțului nostru
  int pos;     // momentul descoperirii în dfs
  int tmp_pop; // populația din nod, doar temporar pînă construim HLD

  circ_buf pop; // populația pe lanț, folosită doar în capetele de lanț
};

// Nu putem muta instantaneu caii de pe un lanț L1 pe altul L2, căci dacă încă
// nu l-am procesat pe L2, vom muta ulterior caii încă odată. Deci ținem o
// listă de postprocesări de făcut după ce procesăm toate lanțurile.
struct postprocess {
  int node, pop;
};

cell list[2 * MAX_NODES];
int circ_buf_space[MAX_NODES];
```

```
node nd[MAX_NODES + 1];
hash_set nonempty_paths; // Doar prin aceste lanţuri vom itera.
postprocess postproc[MAX_NODES];
int n, num_queries, num_postproc;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  scanf("%d %d", &n, &num_queries);

  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].tmp_pop);
  }

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

int heavy_dfs(int u) {
  int my_size = 1, max_child_size = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent) {

      nd[v].parent = u;
      int child_size = heavy_dfs(v);
      my_size += child_size;
      if (child_size > max_child_size) {
        max_child_size = child_size;
        nd[u].heavy = v;
      }

    }
  }

  return my_size;
}

// Descompunere heavy-light cu o mică modificare: impunem o limită de
// MAX_PATH pe lungimea oricărui lanţ.
void decompose_dfs(int u, int head) {
```

```
  static int time = 0;

  nd[u].head = head;
  nd[u].pos = time++;
  int path_size = nd[u].pos - nd[head].pos + 1;
  bool can_fit_heavy = (path_size < MAX_PATH);

  if (nd[u].heavy && can_fit_heavy) {
    decompose_dfs(nd[u].heavy, head);
  } else {
    // Aici se termină lanțul head-u.
    int* start_pos = circ_buf_space + nd[head].pos;
    nd[head].pop.init(start_pos, path_size);
  }

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != nd[u].parent && ((v != nd[u].heavy) || !can_fit_heavy)) {
      decompose_dfs(v, v); // la v începe un lanț nou
    }
  }
}

void populate_circular_buffers() {
  for (int u = 1; u <= n; u++) {
    int h = nd[u].head;
    int pos = nd[u].pos - nd[h].pos;
    nd[h].pop.add(pos, nd[u].tmp_pop);
  }

  for (int u = 1; u <= n; u++) {
    if ((nd[u].head == u) && !nd[u].pop.is_empty()) {
      nonempty_paths.insert(u);
    }
  }
}

void migrate_paths_to_root(int u, hash_set& paths_to_root) {
  int prev_h = 0;
  while (u) {
    int h = nd[u].head;
    if (!nd[h].pop.is_empty()) {
      int pos = nd[u].pos - nd[h].pos;
      int extracted = nd[h].pop.attract(pos);
      if (prev_h && extracted) {
        // Caii din centrul migrației coboară la începutul următorului lanț.
        postproc[num_postproc++] = { prev_h, extracted };
      } else {
        // Caii din centrul migrației nu pleacă nicăieri.
        nd[h].pop.add(pos, extracted);
```

```
      }
    }

    paths_to_root.insert(h);
    u = nd[h].parent;
    prev_h = h;
  }
}

// Migrează caii de pe celelalte lanțuri nevide. Evită lanțurile deja migrate
// în migrate_paths_to_root.
void migrate_other_paths(int u, hash_set paths_to_root) {
  for (int h: nonempty_paths) {
    if (!paths_to_root.contains(h)) {
      int extracted = nd[h].pop.shift_left(); // spre rădăcină
      if (extracted) {
        postproc[num_postproc++] = { nd[h].parent, extracted };
      }
    }
  }
}

void post_migration() {
  // Ștergem din set în timp ce iterăm prin set.
  for (auto it = nonempty_paths.begin(); it != nonempty_paths.end(); ) {
    if (nd[*it].pop.is_empty()) {
      it = nonempty_paths.erase(it);
    } else {
      it++;
    }
  }

  while (num_postproc) {
    postprocess& p = postproc[--num_postproc];
    int h = nd[p.node].head;
    int pos = nd[p.node].pos - nd[h].pos;
    nd[h].pop.add(pos, p.pop);
    nonempty_paths.insert(h);
  }
}

void migrate(int u) {
  hash_set paths_to_root;
  migrate_paths_to_root(u, paths_to_root);
  migrate_other_paths(u, paths_to_root);
  post_migration();
}

int query(int u) {
  int h = nd[u].head;
```

```
  int pos = nd[u].pos - nd[h].pos;
  return nd[h].pop.get(pos);
}

void process_ops() {
  char type1, type2;
  int u;

  while (num_queries--) {
    scanf(" %c%c %d", &type1, &type2, &u);
    if (type1 == 'C') {
      migrate(u);
    } else {
      int claimed_pop;
      scanf("%d", &claimed_pop);
      printf("%d\n", claimed_pop == query(u));
    }
  }
}

int main() {
  read_tree();
  heavy_dfs(1);
  decompose_dfs(1, 1);
  populate_circular_buffers();
  process_ops();

  return 0;
}
```

Anexa L

Arbori - descompunere în centroizi

L.1 Problema Finding a Centroid (CSES)

înapoi

Implementare recursivă și cu STL.

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

const int MAX_NODES = 200'000;

struct node {
  std::vector<int> adj;
  int size;
};

node nd[MAX_NODES + 1];
int n;

void read_input_data() {
  scanf("%d", &n);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
}

void size_dfs(int u) {
  nd[u].size = 1;

  for (int v: nd[u].adj) {
```

```
    if (!nd[v].size) {
      size_dfs(v);
      nd[u].size += nd[v].size;
    }
  }
}

int find_centroid(int u) {
  for (int v: nd[u].adj) {
    if ((nd[v].size < nd[u].size) && (nd[v].size > n / 2)) {
      return find_centroid(v);
    }
  }

  return u;
}

int main() {
  read_input_data();
  size_dfs(1);
  int c = find_centroid(1);
  printf("%d\n", c);

  return 0;
}
```

Implementare iterativă și cu liste proprii.

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int size;
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_child(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
```

```
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_child(u, v);
    add_child(v, u);
  }
}

void size_dfs(int u) {
  nd[u].size = 1;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].size) {
      size_dfs(v);
      nd[u].size += nd[v].size;
    }
  }
}

int get_heavy_child(int u) {
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if ((nd[v].size < nd[u].size) && (nd[v].size > n / 2)) {
      return v;
    }
  }
  return 0;
}

void find_centroid() {
  int u = 1, child;

  while ((child = get_heavy_child(u)) != 0) {
    u = child;
  }

  printf("%d\n", u);
}

int main() {
  read_input_data();
  size_dfs(1);
  find_centroid();
```

```
  return 0;
}
```

L.2 Problema Mystery Tree (CodeChef)

înapoi

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

const int MAX_NODES = 200'000;

struct node {
  std::vector<int> adj;
  int size;
  bool dead;
};

node nd[MAX_NODES + 1];
int n;

void read_tree() {
  scanf("%d", &n);

  // șterge listele de adiacență
  for (int u = 1; u <= n; u++) {
    nd[u].adj.clear();
    nd[u].dead = false;
  }

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
}

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (int v: nd[u].adj) {
    if ((v != parent) && !nd[v].dead) {
      size_dfs(v, u);
      nd[u].size += nd[v].size;
    }
  }
}
```

```
int find_centroid(int u, int limit) {
  for (int v: nd[u].adj) {
    if ((nd[v].size < nd[u].size) && (nd[v].size > limit)) {
      return find_centroid(v, limit);
    }
  }

  return u;
}

void solve(int u) {
  int prev = 0;

  while (u != -1) {
    size_dfs(u, 0);
    u = find_centroid(u, nd[u].size / 2);

    printf("1 %d\n", u);
    fflush(stdout);

    nd[u].dead = true;
    prev = u;
    scanf("%d", &u);
  }

  printf("2 %d\n", prev);
}

int main() {
  int num_tests, check = 1;
  scanf("%d", &num_tests);

  while (num_tests-- && (check == 1)) {
    read_tree();
    solve(1);
    scanf("%d", &check);
  }

  return 0;
}
```

L.3 Problema Ciel the Commander (Codeforces)

‹ înapoi

Soluție cu descompunere în centroizi.

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int size;
  bool dead;
  char rank; // răspunsul
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_child(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_child(u, v);
    add_child(v, u);
  }
}

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if ((v != parent) && !nd[v].dead) {
      size_dfs(v, u);
      nd[u].size += nd[v].size;
    }
  }
}

int find_centroid(int u, int limit) {
```

```
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if ((nd[v].size < nd[u].size) && !nd[v].dead && (nd[v].size > limit)) {
      return find_centroid(v, limit);
    }
  }

  return u;
}

void decompose(int u, int rank) {
  size_dfs(u, 0);
  u = find_centroid(u, nd[u].size / 2);

  // Aceasta este problema pe care o rezolvăm. Restul codului este șablonul
  // pentru descompunerea în centroizi.
  nd[u].rank = rank;

  nd[u].dead = true;
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (!nd[v].dead) {
      decompose(v, rank + 1);
    }
  }
}

void write_ranks() {
  for (int u = 1; u <= n; u++) {
    putchar(nd[u].rank);
    putchar((u == n) ? '\n' : ' ');
  }
}

int main() {
  read_input_data();
  decompose(1, 'A');
  write_ranks();

  return 0;
}
```

Soluție elementară.

```
#include <stdio.h>

const int MAX_NODES = 200'000;

struct cell {
```

```
  int v, next;
};

struct node {
  int adj;
  char rank; // răspunsul
};

cell list[2 * MAX_NODES];
node nd[MAX_NODES + 1];
int n;

void add_child(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_input_data() {
  scanf("%d", &n);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_child(u, v);
    add_child(v, u);
  }
}

// Returnează indicele celui mai din dreapta bit care respectă regulile de
// combinare.
int get_valid_bit(unsigned once, unsigned twice) {
  // Ia cel mai mare rang care apare de două ori.
  int msb = twice
    ? (31 - __builtin_clz(twice))
    : -1;

  // Fiecare bit în stînga lui msb este OK.
  unsigned legal_twice = ~((1 << (msb + 1)) - 1);

  // Fiecare bit văzut o singură dată nu este OK.
  unsigned legal_once = ~once;

  // Trebuie să respectăm ambele criterii.
  unsigned both = legal_once & legal_twice;

  // Returnează indicele celui mai din dreapta bit.
  return __builtin_ctz(both);
}
```

```
// Returnează o mască pe 26 de biți a rangurilor vizibile (MSB este A, LSB
// este Z).
unsigned dfs(int u, int parent) {
  unsigned once = 0, twice = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      unsigned vis_mask = dfs(v, u);
      // Noduri văzute o dată înainte și a doua oară în subarborele lui v.
      twice |= vis_mask & once;
      once |= vis_mask;
    }
  }

  int bit = get_valid_bit(once, twice);
  nd[u].rank = 'Z' - bit;

  // Acest rang este vizibil și maschează toate rangurile mai mici.
  return (1 << bit) | (once & ~((1 << bit) - 1));
}

void write_ranks() {
  for (int u = 1; u <= n; u++) {
    putchar(nd[u].rank);
    putchar((u == n) ? '\n' : ' ');
  }
}

int main() {
  read_input_data();
  dfs(1, 0);
  write_ranks();

  return 0;
}
```

L.4 Problema Fixed-Length Paths I (CSES) (din nou)

‹ înapoi

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

const int MAX_NODES = 200'000;

struct node {
  std::vector<int> adj;
```

```
  int size;
  bool dead;
};

node nd[MAX_NODES + 1];
int freq[MAX_NODES];
int length;
long long answer;

void read_input_data() {
  int n;
  scanf("%d %d", &n, &length);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
}

/**
 * Aceasta este problema pe care o rezolvăm efectiv.
 **/

int max(int x, int y) {
  return (x > y) ? x : y;
}

int max_depth;

// Numără căile de lungime k pornind din u, mergînd în sus pînă la centroid și
// în jos în alt subarbore.
void count_dfs(int u, int parent, int depth) {
  max_depth = max(max_depth, depth);
  answer += freq[length - depth];

  for (int v: nd[u].adj) {
    if ((!nd[v].dead) && (v != parent) && (depth < length)) {
      count_dfs(v, u, depth + 1);
    }
  }
}

// Marchează adîncimile fiecărui nod.
void mark_dfs(int u, int parent, int depth) {
  freq[depth]++;

  for (int v: nd[u].adj) {
    if ((!nd[v].dead) && (v != parent) && (depth < length)) {
```

```
      mark_dfs(v, u, depth + 1);
    }
  }
}

// Numără căile de lungime k care trec prin u.
void count_paths_through(int u) {
  freq[0] = 1; // nodul u însuși
  max_depth = 0;

  for (int v: nd[u].adj) {
    if (!nd[v].dead) {
      count_dfs(v, u, 1);
      mark_dfs(v, u, 1);
    }
  }

  for (int d = 0; d <= max_depth; d++) {
    freq[d] = 0;
  }
}

/**
 * Restul codului (decompose(), size_dfs() și find_centroid()) este șablon. Îl
 * folosim deoarece ne permite să rulăm un DFS din fiecare centroid și să
 * obținem timp O(n log n).
 **/

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (int v: nd[u].adj) {
    if ((v != parent) && !nd[v].dead) {
      size_dfs(v, u);
      nd[u].size += nd[v].size;
    }
  }
}

int find_centroid(int u, int limit) {
  for (int v: nd[u].adj) {
    if ((nd[v].size < nd[u].size) && (nd[v].size > limit)) {
      return find_centroid(v, limit);
    }
  }

  return u;
}

void decompose(int u) {
```

```
  size_dfs(u, 0);
  u = find_centroid(u, nd[u].size / 2);

  // Aceasta este ce încercăm să rezolvăm.
  count_paths_through(u);

  nd[u].dead = true;
  for (int v: nd[u].adj) {
    if (!nd[v].dead) {
      decompose(v);
    }
  }
}

void write_answer() {
  printf("%lld\n", answer);
}

int main() {
  read_input_data();
  decompose(1);
  write_answer();

  return 0;
}
```

L.5 Problema Xenia and Tree (Codeforces)

‹ înapoi

Soluție cu descompunere în radical după operații.

```
// Complexitate: O((n + q) sqrt q).
// Metodă: Descompunere în radical după operații.
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_LOG = 18;
const int BLOCK_SIZE = 1'000; // determinat experimental
const int INFINITY = MAX_NODES;

const int OP_PAINT = 1;

struct cell {
  int v, next;
};

cell list[2 * MAX_NODES];
```

```
struct node {
  int adj;
  int depth;
  int dist; // distanța până la cel mai apropiat nod roșu
  bool red;
};

struct queue {
  int v[MAX_NODES];
  int head, tail;

  void init() {
    head = tail = 0;
  }

  void enqueue(int u) {
    v[tail++] = u;
  }

  int dequeue() {
    return v[head++];
  }

  bool is_empty() {
    return head == tail;
  }
};

node nd[MAX_NODES + 1];
int fresh[BLOCK_SIZE];
queue q;
int n, num_ops, num_fresh;

int log(int x) {
  return 31 - __builtin_clz(x);
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

// O liniarizare DFS cu repetiție și informații pentru interogări de RMQ și
// LCA în O(1).
struct lca_info {
  int d[MAX_LOG][2 * MAX_NODES];
  int tout[MAX_NODES + 1];
  int n; // lungimea liniarizării

  void append(int u) {
```

```
    tout[u] = n;
    d[0][n++] = u;
  }

  void build_rmq_table() {
    for (int l = 1; (1 << l) <= n; l++) {
      for (int i = 0; i <= n - (1 << l); i++) {
        int r = i + (1 << (l - 1));
        d[l][i] = (nd[d[l - 1][i]].depth < nd[d[l - 1][r]].depth)
          ? d[l - 1][i]
          : d[l - 1][r];
      }
    }
  }

  int lca(int u, int v) {
    int left = min(tout[u], tout[v]);
    int right = tout[u] + tout[v] - left;
    int bits = log(right - left + 1);
    int x = d[bits][left];
    int y = d[bits][right - (1 << bits) + 1];
    return (nd[x].depth < nd[y].depth) ? x : y;
  }

  int dist(int u, int v) {
    int l = lca(u, v);
    return nd[u].depth + nd[v].depth - 2 * nd[l].depth;
  }
};

lca_info l;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
}

void read_tree() {
  scanf("%d %d", &n, &num_ops);

  for (int i = 0; i < n - 1; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

// Calculează adîncimile și liniarizarea.
```

```
void initial_dfs(int u, int parent) {
  l.append(u);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      nd[v].depth = 1 + nd[u].depth;
      initial_dfs(v, u);
      l.append(u);
    }
  }
}

void bfs_from_red_nodes() {
  q.init();

  for (int u = 1; u <= n; u++) {
    if (nd[u].red) {
      nd[u].dist = 0;
      q.enqueue(u);
    } else {
      nd[u].dist = INFINITY;
    }
  }

  while (!q.is_empty()) {
    int u = q.dequeue();
    for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
      int v = list[ptr].v;
      if (nd[v].dist == INFINITY) {
        nd[v].dist = nd[u].dist + 1;
        q.enqueue(v);
      }
    }
  }
}

void preprocess_block() {
  bfs_from_red_nodes();
  num_fresh = 0;
}

void paint(int u) {
  nd[u].red = true;
  fresh[num_fresh++] = u;
}

int query(int u) {
  int result = nd[u].dist;
  for (int i = 0; i < num_fresh; i++) {
```

```
    result = min(result, l.dist(u, fresh[i]));
  }

  return result;
}

void process_ops() {
  nd[1].red = true;

  for (int i = 0; i < num_ops; i++) {
    if (i % BLOCK_SIZE == 0) {
      preprocess_block();
    }
    int type, u;
    scanf("%d %d", &type, &u);
    if (type == OP_PAINT) {
      paint(u);
    } else {
      printf("%d\n", query(u));
    }
  }
}

int main() {
  read_tree();
  initial_dfs(1, 0);
  l.build_rmq_table();
  process_ops();

  return 0;
}
```

Soluție cu descompunere în centroizi.

```
// Complexitate: O((n + q) log n).
// Metodă: Descompunere în centroizi.
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;
const int MAX_LOG = 17;

struct node {
  std::vector<int> adj;
  int size;
  int level;
  int closest_red;
  int cparent; // părintele în arborele de centroizi
  int cdist[MAX_LOG]; // distanța până la părintele centroid de nivel k
```

```
  bool dead;
};

node nd[MAX_NODES + 1];
int freq[MAX_NODES];
int n, num_queries;

void read_input_data() {
  scanf("%d %d", &n, &num_queries);

  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }
}

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (int v: nd[u].adj) {
    if ((v != parent) && !nd[v].dead) {
      size_dfs(v, u);
      nd[u].size += nd[v].size;
    }
  }
}

void dist_dfs(int u, int parent, int level) {
  for (int v: nd[u].adj) {
    if ((v != parent) && !nd[v].dead) {
      nd[v].cdist[level] = 1 + nd[u].cdist[level];
      dist_dfs(v, u, level);
    }
  }
}

int find_centroid(int u, int limit) {
  for (int v: nd[u].adj) {
    if ((nd[v].size < nd[u].size) && (nd[v].size > limit)) {
      return find_centroid(v, limit);
    }
  }

  return u;
}

void decompose(int u, int parent, int level) {
  size_dfs(u, 0);
```

```
  u = find_centroid(u, nd[u].size / 2);

  nd[u].dead = true;
  nd[u].level = level;
  nd[u].cparent = parent;

  dist_dfs(u, 0, level);

  for (int v: nd[u].adj) {
    if (!nd[v].dead) {
      decompose(v, u, level + 1);
    }
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

void paint_red(int u) {
  int center = u;
  for (int l = nd[u].level; l >= 0; l--) {
    nd[center].closest_red = min(nd[center].closest_red, nd[u].cdist[l]);
    center = nd[center].cparent;
  }
}

int find_closest_red(int u) {
  int result = n; // infinit

  int center = u;
  for (int l = nd[u].level; l >= 0; l--) {
    int dist = nd[u].cdist[l] + nd[center].closest_red;
    result = min(result, dist);
    center = nd[center].cparent;
  }
  return result;
}

void init_distances() {
  for (int u = 1; u <= n; u++) {
    nd[u].closest_red = n; // infinit
  }
  paint_red(1);
}

void process_ops() {
  while (num_queries--) {
    int type, u;
    scanf("%d %d", &type, &u);
```

```
    if (type == 1) {
      paint_red(u);
    } else {
      printf("%d\n", find_closest_red(u));
    }
  }
}

int main() {
  read_input_data();
  decompose(1, 0, 0);
  init_distances();
  process_ops();

  return 0;
}
```

L.6 Problema Flareon (Lot 2017)

înapoi

```
#include "flareon.h"
#include <vector>

const int MAX_NODES = 200'000;

struct node {
  std::vector<int> adj;
  std::vector<int> pow; // puterile flăcărilor cu originea aici
  int size;
};

static node nd[MAX_NODES + 1];
static long long ans[MAX_NODES + 1];

struct plume_tracker {
  // Date din punctul de vedere al centroidului, pe măsură ce flăcările urcă
  // în subarbore și ajung la centroid.
  //
  // Frecvențele flăcărilor indexate după puterea rămasă. Nu ținem evidența
  // puterilor mai mari decît n, deoarece acelea nu se vor stinge niciodată.
  int freq[MAX_NODES];
  // Numărul de flăcări.
  int cnt;
  // Puterea totală.
  long long sum;

  // Mărimea subarborelui curent. Ne trebuie ca să ne încadrăm în efort
```

```
// O(mărimea_subarborelui), nu O(n).
int sts;

void init(int sts) {
  this->sts = sts;
  sum = cnt = 0;
  for (int i = 0; i < sts; i++) {
    freq[i] = 0;
  }
}

void add_plume(int power, int depth, int sign) {
  if (power > depth) {
    if (power - depth < sts)  {
      freq[power - depth] += sign;
    }
    cnt += sign;
    sum += sign * (power - depth);
  }
  // Altfel nu face nimic: flacăra nu va ajunge la centroid.
}

// Colectează flăcările din nodul u și notează contribuția lor la centroid.
// Cînd sign = +1 includem contribuțiile, iar cînd sign = -1, le excludem.
void collect(int u, int parent, int depth, int sign) {
  for (int p: nd[u].pow) {
    add_plume(p, depth, sign);
  }

  for (int v: nd[u].adj) {
    if (v != parent) {
      collect(v, u, depth + 1, sign);
    }
  }
}

void distribute(int u, int parent, int depth, int cnt, long long sum) {
  // Stinge unele flăcări și redu puterile celorlalte cu cîte 1.
  sum -= cnt;
  cnt -= freq[depth];
  ans[u] += sum;

  for (int v: nd[u].adj) {
    if (v != parent) {
      distribute(v, u, depth + 1, cnt, sum);
    }
  }
}

void process_plumes_through(int u) {
```

```
    init(nd[u].size);
    collect(u, 0, 0, +1);
    ans[u] +=  sum;

    for (int v: nd[u].adj) {
      // Cînd numărăm efectele flăcărilor într-un subarbore, mai întîi
      // excludem flăcările care provin tocmai din acel subarbore.
      collect(v, u, 1, -1);
      distribute(v, u, 1, cnt, sum);
      collect(v, u, 1, +1);
    }
  }
};

plume_tracker pt;

/**
 * Restul codului (decompose(), size_dfs() și find_centroid()) este șablon. Îl
 * folosim deoarece ne permite să rulăm un DFS din fiecare centroid și să
 * obținem timp O(n log n).
 **/

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (int v: nd[u].adj) {
    if (v != parent) {
      size_dfs(v, u);
      nd[u].size += nd[v].size;
    }
  }
}

int find_centroid(int u, int limit) {
  for (int v: nd[u].adj) {
    if ((nd[v].size < nd[u].size) && (nd[v].size > limit)) {
      return find_centroid(v, limit);
    }
  }

  return u;
}

void delete_edge(int u, int v) {
  int i = 0;
  while (nd[u].adj[i] != v) {
    i++;
  }
  nd[u].adj[i] = nd[u].adj.back();
  nd[u].adj.pop_back();
```

```
}

void delete_node(int u) {
  for (int v: nd[u].adj) {
    delete_edge(v, u);
  }
}

void decompose(int u) {
  size_dfs(u, 0);
  u = find_centroid(u, nd[u].size / 2);

  // Aceasta este problema pe care încercăm să o rezolvăm.
  pt.process_plumes_through(u);

  delete_node(u);
  for (int v: nd[u].adj) {
    decompose(v);
  }
}

void solve(int n, int m, int* v, int* pos, int* power) {
  for (int u = 2; u <= n; u++) {
    nd[u].adj.push_back(v[u - 2]);
    nd[v[u - 2]].adj.push_back(u);
  }

  for (int i = 0; i < m; i++) {
    nd[pos[i]].pow.push_back(power[i]);
  }

  decompose(1);
  answer(ans + 1); // shiftează vectorul cu 1
}
```

L.7 Problema Digit Tree (Codeforces)

înapoi

Soluție cu descompunere în centroizi și `std::map`.

```
// Method: Centroid decomposition.
// Complexity: O(n log^2 n).
#include <map>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;
```

```
struct edge {
  int v, digit;
};

struct node {
  std::vector<edge> adj;
  int size;
  bool dead;
};

node nd[MAX_NODES];
int pow10[MAX_NODES + 1];
int inv_pow10[MAX_NODES + 1];
int n, mod;
long long num_pairs;

void read_tree() {
  scanf("%d %d", &n, &mod);

  for (int i = 1; i < n; i++) {
    int u, v, digit;
    scanf("%d %d %d", &u, &v, &digit);
    nd[u].adj.push_back({v, digit % mod});
    nd[v].adj.push_back({u, digit % mod});
  }
}

void extended_euclid(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
  } else {
    int xp, yp;
    extended_euclid(b, a % b, xp, yp);
    x = yp;
    y = xp - (a / b) * yp;
  }
}

int inverse(int a, int mod) {
  int x, y;
  extended_euclid(a, mod, x, y);
  return (x + mod) % mod;
}

void precompute_pow10() {
  pow10[0] = 1;
  for (int i = 1; i <= n; i++) {
    pow10[i] = 10ll * pow10[i - 1] % mod;
```

```
  }

  long long inv = inverse(10, mod);
  inv_pow10[0] = 1;
  for (int i = 1; i <= n; i++) {
    inv_pow10[i] = inv * inv_pow10[i - 1] % mod;
  }
}

struct pair_counter {
  std::map<int, int> freq;

  void head_dfs(int u, int parent, int depth, int rem, int sign) {
    freq[rem] += sign;

    for (edge e: nd[u].adj) {
      if ((e.v != parent) && !nd[e.v].dead) {
        int new_rem = ((long long)pow10[depth] * e.digit + rem) % mod;
        head_dfs(e.v, u, depth + 1, new_rem, sign);
      }
    }
  }

  void tail_dfs(int u, int parent, int depth, int rem) {
    int h = (long long)inv_pow10[depth] * (mod - rem) % mod;

    // Numărul de moduri de a prefixa această coadă cu un început.
    num_pairs += freq[h];

    // Are coada curentă un rest egal cu zero?
    num_pairs += (rem == 0);

    for (edge e: nd[u].adj) {
      if ((e.v != parent) && !nd[e.v].dead) {
        int new_rem = ((long long)rem * 10 + e.digit) % mod;
        tail_dfs(e.v, u, depth + 1, new_rem);
      }
    }
  }

  void count_pairs_through(int u) {
    freq.clear();

    for (edge e: nd[u].adj) {
      if (!nd[e.v].dead) {
        head_dfs(e.v, u, 1, e.digit, +1);
      }
    }

    for (edge e: nd[u].adj) {
```

```
      if (!nd[e.v].dead) {
        head_dfs(e.v, u, 1, e.digit, -1);
        tail_dfs(e.v, u, 1, e.digit);
        head_dfs(e.v, u, 1, e.digit, +1);
      }
    }

    // Numărul de căi de început care au un rest 0.
    num_pairs += freq[0];
  }
};

pair_counter pc;

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (edge e: nd[u].adj) {
    if ((e.v != parent) && !nd[e.v].dead) {
      size_dfs(e.v, u);
      nd[u].size += nd[e.v].size;
    }
  }
}

int find_centroid(int u, int limit) {
  for (edge e: nd[u].adj) {
    if ((nd[e.v].size < nd[u].size) && !nd[e.v].dead && (nd[e.v].size > limit)) {
      return find_centroid(e.v, limit);
    }
  }

  return u;
}

void decompose(int u) {
  size_dfs(u, -1);
  u = find_centroid(u, nd[u].size / 2);

  pc.count_pairs_through(u);

  nd[u].dead = true;
  for (edge e: nd[u].adj) {
    if (!nd[e.v].dead) {
      decompose(e.v);
    }
  }
}

void write_answer() {
```

```
  printf("%lld\n", num_pairs);
}

int main() {
  read_tree();
  precompute_pow10();
  decompose(0);
  write_answer();

  return 0;
}
```

Soluție cu descompunere în centroizi și căutări binare.

```
#include <algorithm>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;

struct edge {
  int v, digit;
};

struct node {
  std::vector<edge> adj;
  int size;
};

// Stochează informații despre frecvența unui rest modulo M. Putem interoga
// aceste informații pentru un fiu dat al rădăcinii (0-based) sau pentru toți
// fiii.
struct rem_info {
  struct elem {
    int child, rem, freq;

    bool operator ==(elem other) {
      return (child == other.child) && (rem == other.rem);
    }

    bool operator <(elem other) {
      return (child < other.child) ||
        ((child == other.child) && (rem < other.rem));
    }

    bool operator >(elem other) {
      return (child > other.child) ||
        ((child == other.child) && (rem > other.rem));
    }
```

```
};

elem c[MAX_NODES], t[MAX_NODES];
int nc, nt;

void init() {
  nc = nt = 0;
}

void insert(int child, int rem) {
  c[nc++] = { child, rem, 1 };
  t[nt++] = { 0, rem, 1 };
}

int merge_duplicates(elem* v, int n) {
  if (n) {
    int j = 1;
    for (int i = 1; i < n; i++) {
      if (v[i] == v[j - 1]) {
        v[j - 1].freq++;
      } else {
        v[j++] = v[i];
      }
    }
    n = j;
  }
  return n;
}

void sort() {
  std::sort(c, c + nc);
  std::sort(t, t + nt);

  nc = merge_duplicates(c, nc);
  nt = merge_duplicates(t, nt);
}

int count_rem_0() {
  return (nt && (t[0].rem == 0))
    ? t[0].freq
    : 0;
}

int find(elem* v, int n, int child, int rem) {
  elem el = { child, rem, 0 };
  int pos = std::lower_bound(v, v + n, el) - v;
  return ((pos < n) && (v[pos] == el)) ? v[pos].freq : 0;
}

int count_not_in_child(int child, int rem) {
```

```
    return find(t, nt, 0, rem) - find(c, nc, child, rem);
  }
};

node nd[MAX_NODES];
int pow10[MAX_NODES + 1];
int inv_pow10[MAX_NODES + 1];
rem_info r;
int n, mod;
long long num_pairs;

void read_tree() {
  scanf("%d %d", &n, &mod);

  for (int i = 1; i < n; i++) {
    int u, v, digit;
    scanf("%d %d %d", &u, &v, &digit);
    nd[u].adj.push_back({v, digit});
    nd[v].adj.push_back({u, digit});
  }
}

void extended_euclid(int a, int b, int& x, int& y) {
  if (!b) {
    x = 1;
    y = 0;
  } else {
    int xp, yp;
    extended_euclid(b, a % b, xp, yp);
    x = yp;
    y = xp - (a / b) * yp;
  }
}

int inverse(int a, int mod) {
  int x, y;
  extended_euclid(a, mod, x, y);
  return (x + mod) % mod;
}

void precompute_pow10() {
  pow10[0] = 1;
  for (int i = 1; i <= n; i++) {
    pow10[i] = 10ll * pow10[i - 1] % mod;
  }

  long long inv = inverse(10, mod);
  inv_pow10[0] = 1;
  for (int i = 1; i <= n; i++) {
    inv_pow10[i] = inv * inv_pow10[i - 1] % mod;
```

```
  }
}

struct pair_counter {
  int child;

  void head_dfs(int u, int parent, int depth, int rem) {
    r.insert(child, rem);

    for (edge e: nd[u].adj) {
      if (e.v != parent) {
        int new_rem = ((long long)pow10[depth] * e.digit + rem) % mod;
        head_dfs(e.v, u, depth + 1, new_rem);
      }
    }
  }

  void tail_dfs(int u, int parent, int depth, int rem) {
    int h = (long long)inv_pow10[depth] * (mod - rem) % mod;

    // Numărul de moduri de a prefixa această coadă cu un început.
    num_pairs += r.count_not_in_child(child, h);

    // Are coada curentă un rest egal cu zero?
    num_pairs += (rem == 0);

    for (edge e: nd[u].adj) {
      if (e.v != parent) {
        int new_rem = ((long long)rem * 10 + e.digit) % mod;
        tail_dfs(e.v, u, depth + 1, new_rem);
      }
    }
  }

  void count_pairs_through(int u) {
    r.init();

    for (edge e: nd[u].adj) {
      child = e.v;
      head_dfs(e.v, u, 1, e.digit % mod);
    }

    r.sort();

    for (edge e: nd[u].adj) {
      child = e.v;
      tail_dfs(e.v, u, 1, e.digit % mod);
    }

    // Numărul de căi de început care au un rest 0.
```

```
    num_pairs += r.count_rem_0();
  }
};

pair_counter pc;

void size_dfs(int u, int parent) {
  nd[u].size = 1;

  for (edge e: nd[u].adj) {
    if (e.v != parent) {
      size_dfs(e.v, u);
      nd[u].size += nd[e.v].size;
    }
  }
}

int find_centroid(int u, int limit) {
  for (edge e: nd[u].adj) {
    if ((nd[e.v].size < nd[u].size) && (nd[e.v].size > limit)) {
      return find_centroid(e.v, limit);
    }
  }

  return u;
}

void delete_edge(int u, int v) {
  int i = 0;
  while (nd[u].adj[i].v != v) {
    i++;
  }
  nd[u].adj[i] = nd[u].adj.back();
  nd[u].adj.pop_back();
}

void delete_node(int u) {
  for (edge e: nd[u].adj) {
    delete_edge(e.v, u);
  }
}

void decompose(int u) {
  size_dfs(u, -1);
  u = find_centroid(u, nd[u].size / 2);

  pc.count_pairs_through(u);

  delete_node(u);
  for (edge e: nd[u].adj) {
```

```
    decompose(e.v);
  }
}

void write_answer() {
  printf("%lld\n", num_pairs);
}

int main() {
  read_tree();
  precompute_pow10();
  decompose(0);
  write_answer();

  return 0;
}
```

Anexa M

Combinatorică - Elemente de bază

M.1 Problema Binomial Coefficients (CSES)

înapoi

```cpp
#include <stdio.h>

const int MAX_N = 1'000'000;
const int MOD = 1'000'000'007;

int fact[MAX_N + 1], inv_fact[MAX_N + 1];

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int comb(int n, int k) {
  return (long long)fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD;
}
```

```
void precompute_factorials() {
  fact[0] = 1;
  for (int i = 1; i <= MAX_N; i++) {
    fact[i] = (long long)fact[i - 1] * i % MOD;
  }

  inv_fact[MAX_N] = inverse(fact[MAX_N]);
  for (int i = MAX_N - 1; i >= 0; i--) {
    inv_fact[i] = (long long)inv_fact[i + 1] * (i + 1) % MOD;
  }
}

void process_queries() {
  int num_tests, n, k;
  scanf("%d", &num_tests);
  while (num_tests--) {
    scanf("%d %d", &n, &k);
    printf("%d\n", comb(n, k));
  }
}

int main() {
  precompute_factorials();
  process_queries();

  return 0;
}
```

M.2 Problema Creating Strings II (CSES)

‹ înapoi

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

const int MAX_N = 1'000'000;
const int MOD = 1'000'000'007;
const int SIGMA = 26;

int fact[MAX_N + 1], inv_fact[MAX_N + 1];
int freq[SIGMA];
int n;

void read_string() {
  int c;
  while (islower(c = getchar())) {
    freq[c - 'a']++;
    n++;
```

```
  }
}

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

void precompute_factorials() {
  fact[0] = 1;
  for (int i = 1; i <= n; i++) {
    fact[i] = (long long)fact[i - 1] * i % MOD;
  }

  inv_fact[n] = inverse(fact[n]);
  for (int i = n - 1; i >= 0; i--) {
    inv_fact[i] = (long long)inv_fact[i + 1] * (i + 1) % MOD;
  }
}

void count_permutations() {
  long long answer = fact[n];
  for (int c = 0; c < SIGMA; c++) {
    answer = answer * inv_fact[freq[c]] % MOD;
  }
  printf("%lld\n", answer);
}

int main() {
  read_string();
  precompute_factorials();
  count_permutations();

  return 0;
}
```

M.3 Problema Distributing Apples (CSES)

înapoi

```cpp
#include <stdio.h>

const int MAX_VAL = 2'000'000;
const int MOD = 1'000'000'007;

int fact[MAX_VAL + 1], inv_fact[MAX_VAL + 1];

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

void precompute_factorials() {
  fact[0] = 1;
  for (int i = 1; i <= MAX_VAL; i++) {
    fact[i] = (long long)fact[i - 1] * i % MOD;
  }

  inv_fact[MAX_VAL] = inverse(fact[MAX_VAL]);
  for (int i = MAX_VAL - 1; i >= 0; i--) {
    inv_fact[i] = (long long)inv_fact[i + 1] * (i + 1) % MOD;
  }
}

int comb(int n, int k) {
  return (long long)fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD;
}

void process_query() {
  int n, k;
  scanf("%d %d", &n, &k);
```

```
  printf("%d\n", comb(n + k - 1, k));
}

int main() {
  precompute_factorials();
  process_query();

  return 0;
}
```

M.4 Problema Christmas Party (CSES)

înapoi

Sursă cu principiul includerii și excluderii, varianta 1.

```
#include <stdio.h>

const int MOD = 1'000'000'007;

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int count_derangements(int n) {
  long long fact = 1;
  for (int i = 1; i <= n; i++) {
    fact = fact * i % MOD;
  }

  long long inv_fact = inverse(fact);
  int sign = (n % 2) ? -1 : 1;
  long long sum = 0;
```

```
  for (int i = n; i >= 0; i--) {
    // Invarianți:
    //   sign = (-1)^i
    //   inv_fact = 1/i!
    sum = (sum + fact * sign * inv_fact) % MOD;
    sign = -sign;
    inv_fact = inv_fact * i % MOD;
  }

  return sum;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", count_derangements(n));

  return 0;
}
```

Sursă cu principiul includerii și excluderii, varianta 2.

```
#include <stdio.h>

const int MOD = 1'000'000'007;

int count_derangements(int n) {
  long long n_fact_over_i_fact = 1;
  int sign = (n % 2) ? -1 : 1;
  long long sum = 0;

  for (int i = n; i >= 0; i--) {
    sum = (sum + sign * n_fact_over_i_fact) % MOD;
    sign = -sign;
    n_fact_over_i_fact = n_fact_over_i_fact * i % MOD;
  }

  return (sum + MOD) % MOD;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", count_derangements(n));

  return 0;
}
```

Sursă cu formulă de recurență.

```
#include <stdio.h>

const int MOD = 1'000'000'007;

int count_derangements(int n) {
  if (n == 0) {
    return 1;
  } else if (n == 1) {
    return 0;
  }

  int a, b = 0, c = 1; // termenii 1 și 2
  for (int i = 3; i <= n; i++) {
    a = b;
    b = c;
    c = (long long)(i - 1) * (a + b) % MOD;
  }
  return c;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", count_derangements(n));

  return 0;
}
```

M.5 Problema Bracket Sequences I (CSES)

‹ înapoi

```
//   comb(2n, n) / (n + 1) =
//   (2n)! / (n! * n! * (n + 1)) =
//   (n + 2) * (n + 3) * ... * (2n) / n!
#include <stdio.h>

const int MOD = 1'000'000'007;

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
```

```
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int catalan(int n) {
  long long fact = 1;
  for (int i = 1; i <= n; i++) {
    fact = fact * i % MOD;
  }

  long long result = inverse(fact);
  for (int i = n + 2; i <= 2 * n; i++) {
    result = result * i % MOD;
  }

  return result;
}

int num_strings_of_length(int n) {
  return (n % 2)
    ? 0
    : catalan(n / 2);
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", num_strings_of_length(n));

  return 0;
}
```

M.6 Problema Bracket Sequences II (CSES)

‹ înapoi

```
// (k + 1) / (n + 1) * comb(2n - k, n)
#include <ctype.h>
#include <stdio.h>
```

```
const int MOD = 1'000'000'007;
int n, k;
bool feasible;

void read_string() {
  scanf("%d ", &n);
  int excess = 0;
  int c;

  while ((excess >= 0) && ispunct(c = getchar())) {
    k++;
    excess += (c == '(') ? +1 : -1;
  }

  feasible = (n % 2 == 0) && (excess >= 0);

  // Redu problema la lungime n, începînd cu k paranteze deschise.
  n = (n - k + excess) / 2;
  k = excess;
}

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int extended_catalan() {
  if (!feasible) {
    return 0;
  }

  long long fact = 1;
  for (int i = 1; i <= n + 1; i++) {
    fact = fact * i % MOD;
  }
  long long result = (long long)(k + 1) * inverse(fact) % MOD;
```

```
  for (int i = n - k + 1; i <= 2 * n - k; i++) {
    result = result * i % MOD;
  }

  return result;
}

int main() {
  read_string();
  printf("%d\n", extended_catalan());

  return 0;
}
```

M.7 Problema Counting Necklaces (CSES)

înapoi

```
#include <stdio.h>

const int MAX_LEN = 1'000'000;
const int MOD = 1'000'000'007;

int p[MAX_LEN + 1];

void precompute_powers(int base, int up_to) {
  p[0] = 1;
  for (int i = 1; i <= up_to; i++) {
    p[i] = (long long)p[i - 1] * base % MOD;
  }
}

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
```

```
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int count_necklaces(int len, int colors) {
  long long sum = 0;
  for (int i = 0; i < len; i++) {
    int d, x, y;
    extended_euclid_iterative(i, len, d, x, y);
    sum += p[d];
  }

  return sum % MOD * inverse(len) % MOD;
}

int main() {
  int len, colors;
  scanf("%d %d", &len, &colors);
  precompute_powers(colors, len);
  printf("%d\n", count_necklaces(len, colors));

  return 0;
}
```

M.8 Problema Counting Grids (CSES)

înapoi

```
#include <stdio.h>

const int MOD = 1'000'000'007;
const int QUARTER = 250'000'002;

long long pow2(long long e) {
  long long result = 1;
  long long b = 2;

  while (e) {
    if (e & 1) {
      result = result * b % MOD;
    }
    b = b * b % MOD;
    e >>= 1;
  }

  return result;
}
```

```
int count_grids(long long n) {
  long long sum;
  if (n % 2 == 0) {
    sum =
      pow2(n * n) +          // nicio rotație
      2 * pow2(n * n / 4) +  // rotație cu 90/270°
      pow2(n * n / 2);       // rotație cu 180°
  } else {
    sum =
      pow2(n * n) +
      2 * pow2((n / 2) * (n / 2) + (n / 2) + 1) +
      pow2((n /2) * n + (n / 2) + 1);
  }

  return sum * QUARTER % MOD;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", count_grids(n));

  return 0;
}
```

M.9 Problema Number of Permutations (Codeforces)

‹ înapoi • versiune online

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

const int MAX_N = 300'000;
const int MOD = 998'244'353;

struct pair {
  int a, b;
};

typedef bool (*cmpFunc)(pair, pair);
typedef bool (*equalsFunc)(pair, pair);

pair v[MAX_N + 1];
int fact[MAX_N + 1];
int n;

bool cmpA(pair p, pair q) {
  return p.a < q.a;
```

```
}

bool cmpB(pair p, pair q) {
  return p.b < q.b;
}

bool cmpBoth(pair p, pair q) {
  return (p.a < q.a) || ((p.a == q.a) && (p.b < q.b));
}

bool equalsA(pair p, pair q) {
  return p.a == q.a;
}

bool equalsB(pair p, pair q) {
  return p.b == q.b;
}

bool equalsBoth(pair p, pair q) {
  return (p.a == q.a) && (p.b == q.b);
}

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &v[i].a, &v[i].b);
  }
  v[n].a = v[n].b = n + 1; // santinele infinite
}

void compute_fact() {
  fact[0] = 1;
  for (int i = 1; i <= n; i++) {
    fact[i] = (long long)i * fact[i - 1] % MOD;
  }
}

int count(cmpFunc cmp, equalsFunc equals) {
  std::sort(v, v + n, cmp);

  long long result = 1;
  int run = 1;
  for (int i = 1; i <= n; i++) {
    if (equals(v[i], v[i - 1])) {
      run++;
    } else {
      result = result * fact[run] % MOD;
      run = 1;
    }
  }
```

```
  return result;
}

bool is_sorted_by_b() {
  for (int i = 1; i < n; i++) {
    if (v[i].b < v[i - 1].b) {
      return false;
    }
  }
  return true;
}

// Dintre cele n! permutări, scade-le pe cele sortate după a sau după b și
// adună-le la loc pe cele sortate după a și după b.
int inclusion_exclusion() {
  long long result = (long long)fact[n]
    + (MOD - count(cmpA, equalsA))
    + (MOD - count(cmpB, equalsB));

  int both = count(cmpBoth, equalsBoth);
  if (is_sorted_by_b()) {
    result += both;
  }
  return result % MOD;
}

int main() {
  read_data();
  compute_fact();
  int result = inclusion_exclusion();
  printf("%d\n", result);

  return 0;
}
```

M.10 Problema Counting Factorizations (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(n^2).
#include <stdio.h>

const int MAX_N = 2'022;
const int MAX_VAL = 1'000'000;
const int MOD = 998'244'353;

short freq[MAX_VAL + 1];
```

```
short pfreq[2 * MAX_N]; // frecvența numerelor prime
int fact[2 * MAX_N + 1], inv_fact[2 * MAX_N + 1];
int d[2 * MAX_N + 1];   // spațiu pentru programarea dinamică
int n;
int num_primes;         // distincte
long long comp_contrib; // contribuția de la numere compuse

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < 2 * n; i++) {
    int x;
    scanf("%d", &x);
    freq[x]++;
  }
}

int bin_exp(int b, int e) {
  long long result = 1;

  while (e) {
    if (e & 1) {
      result = result * b % MOD;
    }
    b = (long long)b * b % MOD;
    e >>= 1;
  }

  return result;
}

void compute_fact() {
  fact[0] = 1;
  for (int i = 1; i <= 2 * n; i++) {
    fact[i] = (long long)i * fact[i - 1] % MOD;
  }

  inv_fact[2 * n] = bin_exp(fact[2 * n], MOD - 2);
  for (int i = 2 * n - 1; i >= 0; i--) {
    inv_fact[i] = (long long)(i + 1) * inv_fact[i + 1] % MOD;
  }
}

bool is_prime(int x) {
  for (int d = 2; d * d <= x; d++) {
    if (x % d == 0) {
      return false;
    }
  }

  return (x != 1);
```

```
}

void collect_primes() {
  comp_contrib = fact[n];
  for (int i = 1; i <= MAX_VAL; i++) {
    if (freq[i]) {
      if (is_prime(i)) {
        pfreq[num_primes++] = freq[i];
      } else {
        comp_contrib = comp_contrib * inv_fact[freq[i]] % MOD;
      }
    }
  }
}

// d[i][j] = Suma contribuțiilor dacă alegem j prime până la poziția i.
void build_dp() {
  d[0] = 1;

  for (int i = 0; i < num_primes; i++) {
    for (int j = i + 1; j > 0; j--) {
      int no_take = (long long)inv_fact[pfreq[i]] * d[j] % MOD;
      int take = (long long)inv_fact[pfreq[i] - 1] * d[j - 1] % MOD;
      d[j] = (take + no_take) % MOD;
    }

    d[0] = (long long)inv_fact[pfreq[i]] * d[0] % MOD;
  }
}

void wrap_it_up() {
  int answer = comp_contrib * d[n] % MOD;
  printf("%d\n", answer);
}

int main() {
  read_data();
  compute_fact();
  collect_primes();
  build_dp();
  wrap_it_up();

  return 0;
}
```

M.11 Problema Monocarp and the Set (Codeforces)

‹ înapoi • versiune online

```cpp
#include <stdio.h>

const int MAX_N = 300'000;
const int MOD = 998'244'353;

char s[MAX_N];
int n, num_ops;
long long answer;

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

void read_data() {
  scanf("%d %d %s", &n, &num_ops, s);
}

void compute_first_answer() {
  answer = 1;
  for (int i = 1; i < n - 1; i++) {
    if (s[i] == '?') {
      answer = answer * i % MOD;
    }
  }
}

void update_string() {
  int pos;
  char c;
  scanf("%d %c", &pos, &c);
  pos--;

  if (s[pos] != c) { // optimizare de viteză
    if ((pos > 0) && (s[pos] == '?')) {
```

```
      answer = answer * inverse(pos) % MOD;
    }
    s[pos] = c;
    if ((pos > 0) && (s[pos] == '?')) {
      answer = answer * pos % MOD;
    }
  }
}

void write_answer() {
  int real_answer = (s[0] == '?') ? 0 : answer;
  printf("%d\n", real_answer);
}

int main() {
  read_data();
  compute_first_answer();
  write_answer();

  while (num_ops--) {
    update_string();
    write_answer();
  }

  return 0;
}
```

M.12 Problema Devu and Flowers (Codeforces)

‹ înapoi

Sursă cu calcularea inverselor (versiune online).

```
#include <stdio.h>

const int MAX_BOXES = 20;
const int MOD = 1'000'000'007;

long long cap[MAX_BOXES], num_objects;
int num_boxes, denominator;

// Calculează x^{-1} % MOD ca x^{MOD-2} % MOD.
int inverse(int x) {
  long long result = 1;
  int e = MOD - 2;

  while (e) {
    if (e & 1) {
```

```
      result = result * x % MOD;
    }
    x = (long long)x * x % MOD;
    e >>= 1;
  }

  return result;
}

void precompute_denominator() {
  denominator = 1;
  for (int i = 1; i < num_boxes; i++) {
    denominator = (long long)denominator * inverse(i) % MOD;
  }
}

// Returnează combinări(n, num_boxes - 1).
int comb(long long n) {
  n %= MOD; // altfel result * n poate depăși
  n += MOD; // altfel n + 1 - i poate trece pe negativ

  long long result = 1;
  for (int i = 1; i < num_boxes; i++) {
    result = result * (n + 1 - i) % MOD;
  }
  return result * denominator % MOD;
}

void read_data() {
  scanf("%d %lld", &num_boxes, &num_objects);
  for (int i = 0; i < num_boxes; i++) {
    scanf("%lld", &cap[i]);
  }
}

int pie(int box, long long num_objects, int sign) {
  if (box == num_boxes) {
    int c = comb(num_objects + num_boxes - 1);
    return (sign == 1) ? c : (MOD - c);
  }

  int sum = pie(box + 1, num_objects, sign);
  if (num_objects > cap[box]) {
    sum += pie(box + 1, num_objects - cap[box] - 1, -sign);
  }

  return sum % MOD;
}
// Mmmmm, pie.
```

```
void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_data();
  precompute_denominator();
  int answer = pie(0, num_objects, +1);
  write_answer(answer);

  return 0;
}
```

Sursă cu inverse precalculate (versiune online).

```
// v3: Hardcodăm 1/(num_boxes-1)!, doar ca distracție.
#include <stdio.h>

const int MAX_BOXES = 20;
const int MOD = 1'000'000'007;

// DENOM[i] = 1 / (i-1)!
int DENOM[MAX_BOXES + 1] = {
  0, 1, 1, 500000004, 166666668, 41666667, 808333339, 301388891, 900198419,
  487524805, 831947206, 283194722, 571199524, 380933296, 490841026, 320774361,
  821384963, 738836565, 514049213, 639669405, 402087866,
};

long long cap[MAX_BOXES], num_objects;
int num_boxes;

// Returnează combinări(n, num_boxes - 1).
int comb(long long n) {
  n %= MOD; // altfel result * n poate depăși
  n += MOD; // altfel n + 1 - i poate trece pe negativ

  long long result = DENOM[num_boxes];
  for (int i = 1; i < num_boxes; i++) {
    result = result * (n + 1 - i) % MOD;
  }
  return result;
}

void read_data() {
  scanf("%d %lld", &num_boxes, &num_objects);
  for (int i = 0; i < num_boxes; i++) {
    scanf("%lld", &cap[i]);
  }
}
```

```
int pie(int box, long long num_objects, int sign) {
  if (box == num_boxes) {
    int c = comb(num_objects + num_boxes - 1);
    return (sign == 1) ? c : (MOD - c);
  }

  int sum = pie(box + 1, num_objects, sign);
  if (num_objects > cap[box]) {
    sum += pie(box + 1, num_objects - cap[box] - 1, -sign);
  }

  return sum % MOD;
}
// Mmmmm, pie.

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_data();
  int answer = pie(0, num_objects, +1);
  write_answer(answer);

  return 0;
}
```

M.13 Problema Lengthening Sticks (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

long long count_all(int a, int b, int c, int l) {
  long long result = 0;
  int start = max(0, b + c - a); // Mai jos ne asigurăm că h ≥ 0.
  for (int x = start; x <= l; x++) {
    long long h = min(l - x, a + x - b - c);
    result += (h + 1) * (h + 2) / 2;
```

```
  }

  return result;
}

int main() {
  int a, b, c, l;
  scanf("%d %d %d %d\n", &a, &b, &c, &l);

  long long result =
    (long long)(l + 3) * (l + 2) * (l + 1) / 6
    - count_all(a, b, c, l)
    - count_all(b, c, a, l)
    - count_all(c, a, b, l);

  printf("%lld\n", result);

  return 0;
}
```

M.14 Problema Shuffle (Codeforces)

‹ înapoi

Sursă în $\mathcal{O}(n^2)$ (versiune online).

```
#include <stdio.h>

const int MAX_N = 5'000;
const int MOD = 998'244'353;

int fact[MAX_N + 1], inv_fact[MAX_N + 1];
char s[MAX_N + 1];
int n, k, total_ones;

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
  }
  d = a;
}
```

```
int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int comb(int n, int k) {
  return (long long)fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD;
}

void precompute_factorials() {
  fact[0] = 1;
  for (int i = 1; i <= n; i++) {
    fact[i] = (long long)fact[i - 1] * i % MOD;
  }

  inv_fact[n] = inverse(fact[n]);
  for (int i = n - 1; i >= 0; i--) {
    inv_fact[i] = (long long)inv_fact[i + 1] * (i + 1) % MOD;
  }
}

void read_data() {
  scanf("%d %d ", &n, &k);
  for (int i = 0; i < n; i++) {
    s[i] = getchar() - '0';
    total_ones += (s[i] == 1);
  }
}

int count_shuffles(int l, int r, int num_ones) {
  if (num_ones > k) {
    return 0;
  }

  int num_zeroes = r - l + 1 - num_ones;

  int outer_zeroes = (s[l] == 0) + (s[r] == 0);
  int outer_ones = 2 - outer_zeroes;

  int inner_zeroes = num_zeroes - outer_ones;
  int inner_ones = num_ones - outer_zeroes;

  return ((inner_zeroes < 0) || (inner_ones < 0))
    ? 0
    : comb(inner_zeroes + inner_ones, inner_ones);
}

int try_all_pairs() {
  long long result = 0;
```

```
  if (total_ones >= k) {
    for (int l = 0; l < n; l++) {
      int num_ones = (s[l] == 1);
      for (int r = l + 1; r < n; r++) {
        num_ones += (s[r] == 1);
        result += count_shuffles(l, r, num_ones);
      }
    }
  }

  result++; // neschimbat

  return result % MOD;
}

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_data();
  precompute_factorials();
  int answer = try_all_pairs();
  write_answer(answer);

  return 0;
}
```

Sursă în $\mathcal{O}(n)$ (versiune online).

```
#include <stdio.h>

const int MAX_N = 5'000;
const int MOD = 998'244'353;

int fact[MAX_N + 1], inv_fact[MAX_N + 1];
char s[MAX_N + 1];
int n, k, total_ones;

void extended_euclid_iterative(int a, int b, int& d, int& x, int& y) {
  x = 1;
  y = 0;
  int xp = 0, yp = 1;
  while (b) {
    int q = a / b;
    int tmp = b; b = a - q * b; a = tmp;
    tmp = xp; xp = x - q * xp; x = tmp;
    tmp = yp; yp = y - q * yp; y = tmp;
```

```
  }
  d = a;
}

int inverse(int x) {
  int y, k, d;
  extended_euclid_iterative(x, MOD, d, y, k);
  return (y >= 0) ? y : (y + MOD);
}

int comb(int n, int k) {
  return (long long)fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD;
}

void precompute_factorials() {
  fact[0] = 1;
  for (int i = 1; i <= n; i++) {
    fact[i] = (long long)fact[i - 1] * i % MOD;
  }

  inv_fact[n] = inverse(fact[n]);
  for (int i = n - 1; i >= 0; i--) {
    inv_fact[i] = (long long)inv_fact[i + 1] * (i + 1) % MOD;
  }
}

void read_data() {
  scanf("%d %d ", &n, &k);
  for (int i = 0; i < n; i++) {
    s[i] = getchar() - '0';
    total_ones += (s[i] == 1);
  }
}

void extend_to_k_ones(int& r, int& ones) {
  while ((r < n - 1) && (ones < k)) {
    r++;
    ones += (s[r] == 1);
  }

  // Consumă și zerourile.
  while ((r < n - 1) && (s[r + 1] == 0)) {
    r++;
  }
}

void contract(int& l, int& ones) {
  ones -= (s[l] == 1);
}
```

```
int count_shuffles(int l, int r, int ones) {
  int zeroes = (r - l + 1) - ones;
  if (!ones || !zeroes) {
    // Fereastra are toate valorile egale. Nu putem schimba capătul stîng.
    return 0;
  }

  ones -= (s[l] == 0);
  zeroes -= (s[l] == 1);
  return comb(zeroes + ones, ones);
}

int try_all_left_ends() {
  long long result = 1; // neschimbat

  int r = -1, ones = 0;
  for (int l = 0; l < n; l++) {
    extend_to_k_ones(r, ones);
    result += count_shuffles(l, r, ones);
    contract(l, ones);
  }

  return result % MOD;
}

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_data();
  precompute_factorials();
  int answer = ((k == 0) || (total_ones < k))
    ? 1
    : try_all_left_ends();
  write_answer(answer);

  return 0;
}
```

Anexa N

Combinatorică - rangul permutărilor și al combinărilor

N.1 Rangul permutărilor

înapoi

```
// Notă: _pdep_u32 necesită argumentul -march=native la compilare.
//
// Ranking și unranking de permutări prin mai multe metode. Nu sînt ieșite din
// comun. Ne interesează să fie ușor de codat.
#include <assert.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

// Vom face ranking și unranking pe NUM_PASSES * NUM_PERMS permutări aleatorii
// de mărime n.
const int NUM_PERMS = 1'000;
const int NUM_PASSES = 10'000;
const int N = 20;
const int MAX_P2 = 16; // pentru fenwick_bin_search()

typedef unsigned long long u64;

typedef struct {
  u64 rank, witness;
  int v[N];
} permutation;

permutation p[NUM_PERMS], q[NUM_PERMS];
u64 fact[N];
char kth[1 << N][N]; // kth[x][k] = al k-lea bit zero în x (0-based)
```

```
void generate_permutations() {
  for (int i = 0; i < NUM_PERMS; i++) {
    for (int j = 0; j < N; j++) {
      int k = rand() % (j + 1);
      p[i].v[j] = p[i].v[k];
      p[i].v[k] = j;
    }
  }
}

void precompute_factorials() {
  fact[0] = 1;
  for (int i = 1; i < N; i++) {
    fact[i] = fact[i - 1] * i;
  }
}

void precompute_kth() {
  for (int i = 0; i < (1 << N); i++) {
    int cnt = 0;
    for (int k = 0; k < N; k++) {
      if ((i & (1 << k)) == 0) {
        kth[i][cnt++] = k;
      }
    }
    // Pozițiile de la cnt la N sînt irelevante pentru că nu există destui
    // biți setați. Fie le atribuim o valoare specială, fie ne asigurăm că
    // codul nu citește acele poziții.
  }
}

void verify_rank() {
  for (int i = 0; i < NUM_PERMS; i++) {
    assert(p[i].rank == p[i].witness);
  }
}

void verify_unrank() {
  for (int i = 0; i < NUM_PERMS; i++) {
    for (int j = 0; j < N; j++) {
      assert(p[i].v[j] == q[i].v[j]);
    }
  }
}

/*****************************************************************
 *                    Gestiunea timpului                         *
 *****************************************************************/
long long t0;
```

```
long long get_time() {
  struct timeval time;
  gettimeofday(&time, NULL);
  return 1000ll * time.tv_sec + time.tv_usec / 1000;
}

void mark_time() {
  t0 = get_time();
}

void report_time(const char* msg) {
  int millis = get_time() - t0;
  int ops = NUM_PASSES * NUM_PERMS;

  printf("%s: %d ms pentru %d operații (%d Mops/sec)\n",
         msg, millis, ops, ops / millis / 1000);
}

/*****************************************************************
 *                        Metoda naivă                           *
 *****************************************************************/

bool used[N];

void rank_naive() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      u64 rank = 0;
      for (int i = 0; i < N; i++) {
        // Numără elementele mai mici decît p[i].
        int cnt = 0;
        for (int j = 0; j < i; j++) {
          cnt += (p[k].v[j] < p[k].v[i]);
        }
        rank += fact[N - 1 - i] * (p[k].v[i] - cnt);
      }
      p[k].rank = p[k].witness = rank;
    }
  }
}

void unrank_naive() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      for (int i = 0; i < N; i++) {
        used[i] = false;
      }
      u64 rank = p[k].rank;
      for (int i = 0; i < N; i++) {
```

```
        int quot = rank / fact[N - 1 - i];
        rank %= fact[N - 1 - i];
        // Caută a quot-a valoare nefolosită.
        int j = 0;
        while (quot || used[j]) {
          quot -= !used[j++];
        }
        q[k].v[i] = j;
        used[j] = true;
      }
    }
  }
}

/*****************************************************************
 *                       Arbori Fenwick                          *
 *****************************************************************/

// Reminder: AIB-urile sînt indexate de la 1.
int fen[N + 1];

void fenwick_clear() {
  for (int i = 1; i <= N; i++) {
    fen[i] = 0;
  }
}

// Pune toate valorile pe 1 și apoi construiește AIB-ul.
// https://stackoverflow.com/a/31070683/6022817
void fenwick_set_1() {
  for (int i = 1; i <= N; i++) {
    fen[i] = 1;
  }
  for (int i = 1; i <= N; i++) {
    int j = i + (i & -i);
    if (j <= N) {
      fen[j] += fen[i];
    }
  }
}

void fenwick_add(int pos, int val) {
  do {
    fen[pos] += val;
    pos += pos & -pos;
  } while (pos <= N);
}

int fenwick_sum(int pos) {
  int s = 0;
```

```
  while (pos) {
    s += fen[pos];
    pos &= pos - 1;
  }
  return s;
}

// Returnează cea mai mare poziție pentru care fen[pos] < val.
int fenwick_bin_search(int val) {
  int pos = 0;

  for (int interval = MAX_P2; interval; interval >>= 1) {
    if ((pos + interval <= N) && (fen[pos + interval] < val)) {
      val -= fen[pos + interval];
      pos += interval;
    }
  }

  return pos;
}

void rank_fenwick() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      fenwick_clear();
      u64 rank = 0;
      for (int i = 0; i < N; i++) {
        // Numără elementele mai mici decît p[i] -- ca în versiunea naivă.
        int cnt = fenwick_sum(p[k].v[i] + 1);
        rank += fact[N - 1 - i] * (p[k].v[i] - cnt);
        fenwick_add(p[k].v[i] + 1, 1);
      }
      p[k].rank = rank;
    }
  }
}

void unrank_fenwick() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      u64 rank = p[k].rank;
      // Valorile 1 din AIB indică elemente încă folosibile.
      fenwick_set_1();
      for (int i = 0; i < N; i++) {
        int quot = rank / fact[N - 1 - i];
        rank %= fact[N - 1 - i];
        // Caută a quot-a valoare folosibilă.
        int j = fenwick_bin_search(quot + 1);
        q[k].v[i] = j;
        fenwick_add(j + 1, -1);
```

```
      }
    }
  }
}

/*****************************************************************
 *                           Popcount                            *
 *****************************************************************/

void rank_popcount() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      u64 rank = 0;
      int mask = 0;
      for (int i = 0; i < N; i++) {
        // Numără biții setați pe pozițiile [0...p[i]).
        int cnt = __builtin_popcount(mask & ((1 << p[k].v[i]) - 1));
        rank += fact[N - 1 - i] * (p[k].v[i] - cnt);
        mask |= (1 << p[k].v[i]);
      }
      p[k].rank = rank;
    }
  }
}

void unrank_popcount() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      u64 rank = p[k].rank;
      int mask = 0;
      for (int i = 0; i < N; i++) {
        int quot = rank / fact[N - 1 - i];
        rank %= fact[N - 1 - i];
        // Caută a quot-a valoare nefolosită.
        int j = kth[mask][quot];
        q[k].v[i] = j;
        mask |= (1 << j);
      }
    }
  }
}

void unrank_popcount2() {
  for (int pass = 0; pass < NUM_PASSES; pass++) {
    for (int k = 0; k < NUM_PERMS; k++) {
      u64 rank = p[k].rank;
      int mask = 0;
      for (int i = 0; i < N; i++) {
        int quot = rank / fact[N - 1 - i];
        rank %= fact[N - 1 - i];
```

```
        // Caută a quot-a valoare nefolosită.
        int zero_bit_mask = _pdep_u32(1 << quot, ~mask);
        int j = _tzcnt_u32(zero_bit_mask);

        q[k].v[i] = j;
        mask |= (1 << j);
      }
    }
  }
}

int main() {
  struct timeval time;
  gettimeofday(&time, NULL);
  srand(time.tv_usec);

  mark_time();
  precompute_factorials();
  precompute_kth();
  generate_permutations();
  rank_naive();
  report_time("inițializare");
  printf("--\n");

  mark_time();
  rank_naive();
  report_time("rank naiv");
  verify_rank();

  mark_time();
  rank_fenwick();
  report_time("rank cu AIB");
  verify_rank();

  mark_time();
  rank_popcount();
  report_time("rank cu popcount");
  verify_rank();
  printf("--\n");

  mark_time();
  unrank_naive();
  report_time("unrank naiv");
  verify_unrank();

  mark_time();
  unrank_fenwick();
  report_time("unrank cu AIB");
  verify_unrank();
```

```
  mark_time();
  unrank_popcount();
  report_time("unrank cu popcount și tabel");
  verify_unrank();

  mark_time();
  unrank_popcount2();
  report_time("unrank cu popcount și pdep");
  verify_unrank();

  return 0;
}
```

N.2 Rangul combinărilor

‹ înapoi

```
// Ranking și unranking de combinări în ordine colexicografică.
#include <algorithm>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

// Vom face ranking și unranking TRIALS combinări aleatorii de N luate cîte K.
#define TRIALS 10'000'000
#define N 60
#define K 30
#define COLEX_UNRANK_STEP 3 // de ordinul a sqrt(N/K)

typedef unsigned long long u64;

typedef struct {
  u64 rank;
  int v[K];
} combination;

combination c[TRIALS];
bool seen[N]; // folosit pentru generarea combinăriilor aleatorii
u64 choose[N + 1][K + 1];

/*****************************************************************
 *                    Gestiunea timpului                         *
 *****************************************************************/
long long __time1;

long long get_time() {
```

```
  struct timeval time;
  gettimeofday(&time, NULL);
  return 1000ll * time.tv_sec + time.tv_usec / 1000;
}

void mark_time() {
  __time1 = get_time();
}

void report_time(const char* msg) {
  long long time2 = get_time();
  fprintf(stderr, "%s: %lld milisecunde\n", msg, time2 - __time1);
}

/*****************************************************************
 *   Ordine colexicografică: ranking în O(k), unranking în O(n)   *
 *****************************************************************/

u64 rank_colex(int* c) {
  u64 result = 0;
  for (int i = 0; i < K; i++) {
    result += choose[(int)c[i]][i + 1];
  }
  return result;
}

void unrank_colex(u64 rank, int* witness) {
  int n = N - 1;
  for (int k = K; k; k--) {
    while (choose[n][k] > rank) {
      n--;
    }
    rank -= choose[n][k];
    assert(n == witness[k - 1]);
  }
}

// ca și unrank_colex, dar cu o optimizare (sare mai mulți pași deodată)
void unrank_colex2(u64 rank, int* witness) {
  int n = N - 1;
  for (int k = K; k; k--) {
    while (n >= COLEX_UNRANK_STEP && choose[n - COLEX_UNRANK_STEP][k] > rank) {
      n -= COLEX_UNRANK_STEP;
    }
    while (choose[n][k] > rank) {
      n--;
    }
    rank -= choose[n][k];
    assert(n == witness[k - 1]);
  }
```

```
}

int main() {
  struct timeval time;
  gettimeofday(&time, NULL);
  srand(time.tv_usec);

  // precalculează combinările
  mark_time();
  for (int n = 0; n <= N; n++) {
    for (int k = 0; k <= K; k++) {
      if (k == 0) {
        choose[n][k] = 1;
      } else if (n == 0) {
        choose[n][k] = 0;
      } else {
        choose[n][k] = choose[n - 1][k] + choose[n - 1][k - 1];
      }
    }
  }
  printf("Benchmark C(%d,%d) = %.3e, pas unranking=%d, %dM experimente\n",
         N, K, (double)choose[N][K], COLEX_UNRANK_STEP, TRIALS / 1'000'000);
  report_time("Precalcularea combinărilor");

  // generează experimentele
  mark_time();
  for (int i = 0; i < TRIALS; i++) {
    for (int j = 0; j < K; j++) {
      int val;
      do {
        val = rand() % N;
      } while (seen[val]);
      seen[val] = true;
      c[i].v[j] = val;
    }
    std::sort(c[i].v, c[i].v + K);

    for (int j = 0; j < K; j++) {
      seen[(int)c[i].v[j]] = false;
    }
  }
  report_time("Generarea experimentelor");
  printf("--\n");

  // benchmarking
  mark_time();
  for (int i = 0; i < TRIALS; i++) {
    c[i].rank = rank_colex(c[i].v);
  }
  report_time("Ranking colexicografic");
```

```
  mark_time();
  for (int i = 0; i < TRIALS; i++) {
    unrank_colex(c[i].rank, c[i].v);
  }
  report_time("Unranking colexicografic");
  mark_time();

  for (int i = 0; i < TRIALS; i++) {
    unrank_colex2(c[i].rank, c[i].v);
  }
  report_time("Unranking colexicografic cu optimizare");
  return 0;
}
```

N.3 Problema Misha and Permutation Summation (Codeforces)

‹ înapoi • versiune online

```
// Metodă calculăm rangul lui P și Q în baza factorial. Adunăm rangurile și
// ignorăm ultimul transport (peste cifra a N-a). Calculăm permutarea pentru
// rangul-sumă.
#include <stdio.h>

const int MAX_N  = 200'000;

int r[MAX_N], n; // r[] acumulează rangurile lui P și Q

struct fenwick_tree {
  int v[MAX_N + 1];
  int n, max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
    for (int i = 1; i <= n; i++) {
      v[i] = 0;
    }
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }
```

```
  int prefix_sum(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  int bin_search(int val) {
    int pos = 0;

    for (int interval = max_p2; interval; interval >>= 1) {
      if ((pos + interval <= n) && (v[pos + interval] < val)) {
        val -= v[pos + interval];
        pos += interval;
      }
    }

    return pos;
  }
};

fenwick_tree fen;

// Citește o permutare și îi adaugă rangul factorial în r[].
void read_and_rank() {
  fen.init(n);

  for (int i = n - 1; i >= 0; i--) {
    int x;
    scanf("%d", &x);
    x++;
    int cnt = fen.prefix_sum(x);
    r[i] += (x - 1) - cnt;
    fen.add(x, 1);
  }
}

void carry() {
  int t = 0;
  for (int i = 0; i < n; i++) {
    r[i] += t;
    t = (r[i] > i);
    r[i] -= t * (i + 1);
  }
}

// Face unrank la r[] și tipărește permutarea.
```

```
void unrank_and_write() {
  // Valorile de 1 din AIB indică elemente folosibile.
  // Avem deja toate valorile pe 1 ca urmare a lui read_and_rank().
  for (int i = n - 1; i >= 0; i--) {
    // Găsește cea de-a r[i]-a valoare nefolosită.
    int x = fen.bin_search(r[i] + 1);
    printf("%d ", x);
    fen.add(x + 1, -1);
  }
  printf("\n");
}

int main() {
  scanf("%d", &n);
  read_and_rank();
  read_and_rank();
  carry();
  unrank_and_write();

  return 0;
}
```

N.4 Problema Inversion Sort (SPOJ)

înapoi

```
// 1. Calculează rangurile permutărilor și le mapează la intervalul [0...10!).
// 2. Rulează BFS ca să calculeze distanțele.
// 3. Pentru o interogare, întîi remapeaz-o ca să pornească de la abcdefghij.
//
// v2: Generează mutările schimbînd cîte o singură pereche.
// v3: Folosește popcount-ul propriu.
#include <stdio.h>
#include <string.h>

const int LENGTH = 10;
const int NUM_PERMS = 3'628'800;
const int INFINITY = 0xff;
const char* START = "abcdefghij";
const int FACT[LENGTH] = { 1, 1, 2, 6, 24, 120, 720, 5'040, 40'320, 362'880 };

struct string {
  char s[LENGTH + 1];
  int rank;

  void set(const char* s, int rank) {
    strcpy(this->s, s);
    this->rank = rank;
```

```
  }

  void flip(int l, int r) {
    while (l < r) {
      swap(l++, r--);
    }
  }

  void swap(int i, int j) {
    int tmp = s[i]; s[i] = s[j]; s[j] = tmp;
  }
};

string q[NUM_PERMS];
unsigned char dist[NUM_PERMS];
int pop[1 << LENGTH];

void precompute() {
  for (int i = 1; i < (1 << LENGTH); i++) {
    pop[i] = pop[i >> 1] + (i & 1);
  }
}

int rank(const char* s) {
  int result = 0, mask = 0;

  for (int i = 0; i < LENGTH; i++) {
    int bit = 1 << (s[i] - 'a');
    int cnt = pop[mask & (bit - 1)];
    mask |= bit;
    result += cnt * FACT[i];
  }

  return result;
}

void bfs() {
  for (int i = 0; i < NUM_PERMS; i++) {
    dist[i] = INFINITY;
  }

  int head = 0, tail = 0;
  int rank_start = rank(START);
  q[tail++].set(START, rank_start);
  dist[rank_start] = 0;

  while (head != tail) {
    string str = q[head++];

    for (int l = 1; l <= 2; l++) {
```

```
      for (int i = 0; i + l < LENGTH; i++) {
        int x = i, y = i + l;
        do {
          str.swap(x--, y++);
          int r2 = rank(str.s);
          if (dist[r2] == INFINITY) {
            dist[r2] = 1 + dist[str.rank];
            q[tail++].set(str.s, r2);
          }

        } while (x >= 0 && y < LENGTH);
        str.flip(x + 1, y - 1);
      }
    }
  }
}

// Aplică aceeași transformare pe dest care ar transforma src în START.
void remap(char* src, char* dest) {
  for (int i = 0; i < LENGTH; i++) {
    int j = 0;
    while (src[j] != dest[i]) {
      j++;
    }
    dest[i] = START[j];
  }
}

void process_queries() {
  char src[LENGTH + 1], dest[LENGTH + 1];
  while ((scanf("%s %s", src, dest) == 2) && strcmp(src, "*")) {
    remap(src, dest);
    printf("%d\n", dist[rank(dest)]);
  }
}

int main() {
  precompute();
  bfs();
  process_queries();

  return 0;
}
```

N.5 Problema Four Chips (SPOJ)

‹ înapoi

```
#include <stdio.h>

typedef unsigned char byte;

const int SIZE = 70;
const int NUM_BOARDS = 916'895; // comb(SIZE, 4)
const byte INFINITY = 0xff;
const int NUM_MOVES = 20; // 4 piese, fiecare cu 2 deplasări și 3 salturi
const int JUMP_TARGET[4][3] = { {1, 2, 3}, {0, 2, 3}, {0, 1, 3}, {0, 1, 2} };

int comb[SIZE + 1][5];

struct board {
  byte p[4];
  int rank;

  board operator=(const board& other) {
    for (int i = 0; i < 4; i++) {
      p[i] = other.p[i];
    }
    rank = other.rank;
    return *this;
  }

  void compute_rank() {
    rank =
      (comb[SIZE][4] - comb[SIZE - p[0]][4]) +
      (comb[SIZE - 1 - p[0]][3] - comb[SIZE - p[1]][3]) +
      (comb[SIZE - 1 - p[1]][2] - comb[SIZE - p[2]][2]) +
      (p[3] - p[2] - 1);
  }

  void sort(int i) {
    while ((i > 0) && (p[i] < p[i - 1])) {
      int tmp = p[i]; p[i] = p[i - 1]; p[i - 1] = tmp;
      i--;
    }
    while ((i < 3) && (p[i] > p[i + 1])) {
      int tmp = p[i]; p[i] = p[i + 1]; p[i + 1] = tmp;
      i++;
    }
  }

  bool empty(int pos) {
    return (pos >= 0) && (pos < SIZE) && (p[0] != pos) &&
      (p[1] != pos) && (p[2] != pos) && (p[3] != pos);
  }

  bool make_slide_move(int i, int delta) {
```

```
    if (empty(p[i] + delta)) {
      p[i] += delta;
      // nu este nevoie de resortare
      return true;
    } else {
      return false;
    }
  }

  bool make_jump_move(int i, int j) {
    int x = 2 * p[j] - p[i];
    if (empty(x)) {
      p[i] = x;
      sort(i);
      return true;
    } else {
      return false;
    }
  }

  bool make_move(int index) {
    int i = index / 5, j;
    int type = index % 5;

    bool success;
    switch (type) {
      case 0:
      case 1:
      case 2:
        j = JUMP_TARGET[i][type];
        success = make_jump_move(i, j);
        break;

      case 3:
        success = make_slide_move(i, +1);
        break;

      case 4:
        success = make_slide_move(i, -1);
        break;
    }

    if (success) {
      compute_rank();
    }

    return success;
  }
};
```

```
const board START = { { 0, 1, 2, 3 }, 0 };

board q[NUM_BOARDS];
byte dist[NUM_BOARDS];

void precompute_combinations() {
  comb[0][0] = 1;
  for (int n = 1; n <= SIZE; n++) {
    comb[n][0] = 1;
    for (int k = 1; k <= 4; k++) {
      comb[n][k] = comb[n - 1][k] + comb[n - 1][k - 1];
    }
  }
}

void bfs() {
  for (int i = 0; i < NUM_BOARDS; i++) {
    dist[i] = INFINITY;
  }
  dist[0] = 0;

  int head = 0, tail = 0;
  q[tail++] = START;

  while (head != tail) {
    board& b = q[head++];
    board b2;

    for (int m = 0; m < NUM_MOVES; m++) {
      b2 = b;
      if (b2.make_move(m) && (dist[b2.rank] == INFINITY)) {
        dist[b2.rank] = 1 + dist[b.rank];
        q[tail++] = b2;
      }
    }
  }
}

void process_queries() {
  int num_tests;
  board b;

  scanf("%d", &num_tests);

  while (num_tests--) {
    scanf("%hhd %hhd %hhd %hhd", &b.p[0], &b.p[1], &b.p[2], &b.p[3]);
    b.p[0]--;
    b.p[1]--;
    b.p[2]--;
    b.p[3]--;
```

```
    b.compute_rank();
    printf("%d\n", dist[b.rank]);
  }
}

int main() {
  precompute_combinations();
  bfs();
  process_queries();

  return 0;
}
```

N.6 Problema Long Permutation (Codeforces)

înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 14;
const int T_RANGE_SUM = 1;
const long long FACT[MAX_N] = {
  1, 1, 2, 6, 24, 120, 720, 5'040, 40'320, 362'880, 3'628'800, 39'916'800,
  479'001'600, 6'227'020'800,
};

// O permutare normală a mulțimii [0...n-1].
struct permutation {
  int v[MAX_N], psum[MAX_N];
  int n;

  void init(int n) {
    this->n = n;
  }

  int get_kth_free(int mask, int k) {
    int pos = 0;
    while (k || (mask & (1 << pos))) {
      k -= !(mask & (1 << pos));
      pos++;
    }
    return pos;
  }

  void unrank(long long r) {
    int occupancy = 0;
    for (int i = 0; i < n; i++) {
      int d = r / FACT[n - 1 - i];
```

```
      d = get_kth_free(occupancy, d);
      v[i] = d;
      psum[i] = v[i] + (i ? psum[i - 1] : 0);
      occupancy |= 1 << d;
      r %= FACT[n - 1 - i];
    }
  }
};

// O permutare care este sortată pe prefix și permutată doar pe sufix.
struct permutation_with_cherry_on_top {
  permutation p;
  int n, head, tail;
  long long rank;

  void init(int n) {
    this->n = n;
    rank = 0;
    tail = (n < MAX_N) ? n : MAX_N;
    head = n - tail;
    p.init(tail);
    p.unrank(0);
  }

  void advance(int x) {
    rank += x;
    p.unrank(rank);
  }

  long long prefix_sum(int k) {
    if (k <= head) {
      // Sumă pe interval în progresia sortată.
      return (long long)k * (k + 1) / 2;
    } else {
      // Toată partea sortată...
      return (long long)head * (head + 1) / 2
        // ...plus suma cozii permutate...
        + p.psum[k - head - 1]
        // ...plus faptul că elementele cozii încep de fapt la head+1
        + (long long)(k - head) * (head + 1);
    }
  }

  long long range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};

permutation_with_cherry_on_top pc;
```

```
int main() {
  int n, q, type, l, r, x;
  scanf("%d %d", &n, &q);
  pc.init(n);

  while (q--) {
    scanf("%d", &type);
    if (type == T_RANGE_SUM) {
      scanf("%d %d", &l, &r);
      printf("%lld\n", pc.range_sum(l, r));
    } else {
      scanf("%d", &x);
      pc.advance(x);
    }
  }

  return 0;
}
```

N.7 Problema Arbperm2 (NerdArena)

‹ înapoi • versiune online

```
// Complexitate: O(n log n).
#include <stdio.h>

const int MAX_N = 100'000;

struct fenwick_tree {
  int v[MAX_N + 1];
  int n, max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
    for (int i = 1; i <= n; i++) {
      v[i] = 0;
    }
  }

  void set(int pos) {
    do {
      v[pos]++;
      pos += pos & -pos;
    } while (pos <= n);
  }

  int prefix_sum(int pos) {
```

```
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  // Returnează ultima poziție unde numărul de zerouri ≤ k, datorită faptului
  // că AIB-ul este 1-based, pe cînd logica programului este 0-based.
  int get_kth_zero(int k) {
    int pos = 0;

    for (int size = max_p2; size; size >>= 1) {
      if ((pos + size <= n) && (size - v[pos + size] <= k)) {
        k -= size - v[pos + size];
        pos += size;
      }
    }

    return pos;
  }
};

int p[MAX_N];       // permutarea originală
int ord[MAX_N + 1]; // ord[val] = ordinea lui val printre elemente < val
fenwick_tree fen;
int n, k;

void read_data() {
  FILE* f = fopen("arbperm2.in", "r");
  fscanf(f, "%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &p[i]);
  }
  fclose(f);
}

void init_ord() {
  fen.init(n);
  for (int i = 0; i < n; i++) {
    ord[p[i]] = fen.prefix_sum(p[i]);
    fen.set(p[i]);
  }
}

void advance_ord() {
  for (int i = n; i >= 1; i--) {
    int old_k = k;
    k = (ord[i] + k) / i;
```

```
    ord[i] = (ord[i] + old_k) % i;
  }
}

void rebuild_p() {
  fen.init(n);
  for (int i = n; i >= 1; i--) {
    int pos = fen.get_kth_zero(ord[i]);
    p[pos] = i;
    fen.set(pos + 1);
  }
}

void write_answer() {
  FILE* f = fopen("arbperm2.out", "w");
  for (int i = 0; i < n; i++) {
    fprintf(f, "%d ", p[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  init_ord();
  advance_ord();
  rebuild_p();
  write_answer();

  return 0;
}
```

N.8 Problema Hipersimetrie (ONI 2019 clasele 11-12)

‹ înapoi • versiune online

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

const int MAX_K = 1'000'000;
const int MAX_LOG = 30;
const int MOD = 1'000'000'007;
const int ROOT_N = 32'768;

typedef unsigned long long u64;

char k[MAX_K + 1];
```

```
int n, len_k;

struct powers_of_2 {
  // p: 2^0, 2^1, 2^2, ..., 2^32767
  // q: 2^0, 2^32768, 2^65536, 2^98304, ... 2^(32768*32767)
  int p[ROOT_N], q[ROOT_N];

  void precompute() {
    p[0] = 1;
    for (int i = 1; i < ROOT_N; i++) {
      p[i] = p[i - 1] * 2 % MOD;
    }

    q[0] = 1;
    q[1] = (2 * p[ROOT_N - 1]) % MOD;
    for (int i = 2; i < ROOT_N; i++) {
      q[i] = (u64)q[i - 1] * q[1] % MOD;
    }
  }

  int get(u64 e) {
    e %= (MOD - 1); // Mica teoremă a lui Fermat.
    u64 large = q[e / ROOT_N];
    u64 small = p[e % ROOT_N];
    return small * large % MOD;
  }
};

powers_of_2 pow2;

struct bit_stream {
  std::vector<bool> col[MAX_LOG];
  int height[MAX_LOG];
  int num_cols;

  void collect_heights(int n) {
    num_cols = 0;
    while (n) {
      if (n & 1) {
        height[num_cols++] = (n + 1) / 2;
      }
      n /= 2;
    }
  }

  void distribute_bits() {
    int c = 0;
    for (int b = len_k - 1; b >= 0; b--) {
      col[c].push_back(k[b] - '0');
      height[c]--;
```

```
      if (height[c] < height[c + 1]) {
        c++;
      } else {
        c = 0;
      }
    }
  }

  void init(int n) {
    collect_heights(n);
    distribute_bits();
  }

  std::vector<bool> get_next_column() {
    static int pos = 0;
    return col[pos++];
  }
};

bit_stream bs;

int compute_cross(int size) {
  std::vector<bool> col = bs.get_next_column();
  u64 pos = (u64)(size / 2) * (n + 1);

  u64 result = 0;

  if (col.size() && col[0]) {
    result += pow2.get(pos);
  }

  for (int i = 1; i < (int)col.size(); i++) {
    if (col[i]) {
      result += (u64)pow2.get(pos - (u64)n * i)
        + pow2.get(pos + (u64)n * i)
        + pow2.get(pos - i)
        + pow2.get(pos + i);
    }
  }

  return result % MOD;
}

int compute_matrix(int size) {
  if (!size) {
    return 0;
  }

  u64 cross = (size & 1) ? compute_cross(size) : 0;
```

```
  int shift = (1 + size) / 2;
  u64 small = compute_matrix(size / 2);
  u64 mult = (u64)(pow2.get(shift) + 1) * (pow2.get((u64)shift * n) + 1) % MOD;
  u64 corners = small * mult % MOD;

  return (corners + cross) % MOD;
}

void read_data() {
  FILE* f = fopen("hipersimetrie.in", "r");
  fscanf(f, "%d %s", &n, k);
  fclose(f);

  len_k = strlen(k);
}

void decrement_k() {
  char* s = k + len_k - 1;
  while (*s == '0') {
    *s = '1';
    s--;
  }
  *s = '0';
}

void write_result(int result) {
  FILE* f = fopen("hipersimetrie.out", "w");
  fprintf(f, "%d\n", result);
  fclose(f);
}

int main() {
  read_data();
  decrement_k();
  pow2.precompute();
  bs.init(n);

  int result = compute_matrix(n);
  write_result(result);

  return 0;
}
```

N.9 Problema Trim (Baraj ONI 2023)

‹ înapoi • versiune online

```
#include <algorithm>
```

```
#include <stdio.h>

const int MAX_BITS = 100;
const int MAX_QUERIES = 100'000;

struct query {
  long long pos;
  int orig_ind;
};

query q[MAX_QUERIES];
bool ans[MAX_QUERIES];
long long choose[MAX_BITS + 1][MAX_BITS + 1];
int n, num_queries;

void read_input_data() {
  FILE* f = fopen("trim.in", "r");
  fscanf(f, "%d %d", &n, &num_queries);
  for (int i = 0; i < num_queries; i++) {
    fscanf(f, "%lld", &q[i].pos);
    q[i].pos--;
    q[i].orig_ind = i;
  }
  fclose(f);
}

void precompute_combinations() {
  choose[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    choose[i][0] = 1;
    for (int j = 1; j <= i; j++) {
      choose[i][j] = choose[i - 1][j] + choose[i - 1][j - 1];
    }
  }
}

void sort_queries() {
  std::sort(q, q + num_queries, [] (query a, query b) {
    return a.pos < b.pos;
  });
}

// Consideră toate măștile de n biți cu k biți de 1, unde biții 0 și n-1 sînt
// 1. Există C(n-2, k-2) astfel de măști. Din combinarea #comb (0-based)
// returnează bitul #pos (0-based, numărînd de la MSB spre LSB).
bool find_bit(int n, int k, long long comb, int pos) {
  if (pos == 0 || pos == n - 1) {
    return true;
  }
```

```
  n -= 2;
  k -= 2;
  pos--;

  while (pos--) {
    if (comb >= choose[n - 1][k]) {
      comb -= choose[n - 1][k];
      k--;
    }
    n--;
  }

  return (comb >= choose[n - 1][k]);
}

struct bit_stream {
  int qind;          // indicele interogării curente
  long long emitted; // numărul de biți emiși până acum

  // Grupul curent. De exemplu, pentru n = 5, vom emite grupurile:
  //
  // * numere cu 1 bit de 1 și de lățime 1
  // * numere cu 2 biți de 1 și de lățimi 2, 3, 4, 5
  // * numere cu 3 biți de 1 și de lățimi 3, 4, 5
  // * numere cu 4 biți de 1 și de lățimi 4, 5
  // * numere cu 5 biți de 1 și de lățime 5
  int num_bits, width;

  // Procesează separat numerele pe 1 bit. Există n astfel de numere.
  void answer_width_1() {
    while (qind < num_queries && q[qind].pos < n) {
      ans[q[qind++].orig_ind] = true;
    }
    emitted = n;
  }

  void advance_group() {
    if (width == n) {
      width = ++num_bits;
    } else {
      width++;
    }
  }

  void answer_queries_for_group() {
    long long combinations = choose[width - 2][num_bits - 2];
    int repetitions = n + 1 - width;
    long long comb_size = combinations * repetitions * width;

    while ((qind < num_queries) && (q[qind].pos < emitted + comb_size)) {
```

```
      long long offset = q[qind].pos - emitted;
      long long combination = offset / (repetitions * width);
      offset %= width;
      ans[q[qind++].orig_ind] = find_bit(width, num_bits, combination, offset);
    }

    emitted += comb_size;
  }

  // Emite biții în grupuri și răspunde la întrebări în ordine crescătoare.
  void emit() {
    qind = 0;
    num_bits = 1;
    width = n;

    answer_width_1();

    while (qind < num_queries) {
      advance_group();
      answer_queries_for_group();
    }
  }
};

bit_stream bs;

void write_answers() {
  FILE* f = fopen("trim.out", "w");
  for (int i = 0; i < num_queries; i++) {
    fputc(ans[i] + '0', f);
  }
  fputc('\n', f);
  fclose(f);
}

int main() {
  read_input_data();
  precompute_combinations();
  sort_queries();
  bs.emit();
  write_answers();

  return 0;
}
```

Anexa O

Geometrie - Elemente de bază

O.1 Problema Cuiburi (Baraj ONI 2010)

‹ înapoi • versiune online

```cpp
#include <algorithm>
#include <math.h>
#include <stdio.h>

const int MAX_NESTS = 2'000;
const int T_RECTANGLE = 0;
const int T_CIRCLE = 1;

bool closer(int x1, int y1, int x2, int y2, int dist) {
  return (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1) <= dist * dist;
}

struct nest {
  double area;
  union { short x1; short x; };
  union { short y1; short y; };
  union { short x2; short r; };
  union { short y2; short unused; };
  unsigned char type;

  void read(FILE* f) {
    fscanf(f, "%hhu", &type);
    if (type == T_RECTANGLE) {
      fscanf(f, "%hd %hd %hd %hd", &x1, &y1, &x2, &y2);
    } else {
      fscanf(f, "%hd %hd %hd", &x, &y, &r);
    }

    compute_area();
  }
```

```
  void compute_area() {
    if (type == T_RECTANGLE) {
      area = ((int)x2 - x1) * ((int)y2 - y1);
    } else {
      area = M_PI * r * r;
    }
  }

  bool fits_in(nest& other) {
    if (type == T_RECTANGLE) {
      if (other.type == T_RECTANGLE) {
        return
          (x1 >= other.x1) && (y1 >= other.y1) &&
          (x2 <= other.x2) && (y2 <= other.y2);
      } else {
        return
          closer(x1, y1, other.x, other.y, other.r) &&
          closer(x1, y2, other.x, other.y, other.r) &&
          closer(x2, y1, other.x, other.y, other.r) &&
          closer(x2, y2, other.x, other.y, other.r);
      }
    } else {
      if (other.type == T_RECTANGLE) {
        return
          (x >= other.x1 + r) &&
          (x <= other.x2 - r) &&
          (y >= other.y1 + r) &&
          (y <= other.y2 - r);
      } else {
        return closer(x, y, other.x, other.y, other.r - r);
      }
    }
  }
};

nest a[MAX_NESTS];
short len[MAX_NESTS];
int n;

void read_nests() {
  FILE* f = fopen("cuiburi.in", "r");
  fscanf(f, "%d", &n);
  for (int i = 0; i < n; i++) {
    a[i].read(f);
  }
  fclose(f);
}

void sort_nests() {
```

```
  std::sort(a, a + n, [](nest& p, nest& q) {
    return p.area < q.area;
  });
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int find_longest_subsequence() {
  short max_len = 0;

  for (int i = 0; i < n; i++) {
    len[i] = 1;
    for (int j = 0; j < i; j++) {
      if (a[j].fits_in(a[i])) {
        len[i] = max(len[i], 1 + len[j]);
      }
    }

    max_len = max(max_len, len[i]);
  }

  return max_len;
}

void write_result(int result) {
  FILE* f = fopen("cuiburi.out", "w");
  fprintf(f, "%d\n", result);
  fclose(f);
}

int main() {
  read_nests();
  sort_nests();
  int result = find_longest_subsequence();
  write_result(result);

  return 0;
}
```

O.2 Problema Ace (OJI 2017, clasa a 9-a)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_ROWS = 1'000;
```

```
short a[MAX_ROWS][MAX_ROWS];
int task, rows, cols, result;

void read_matrix() {
  FILE* f = fopen("ace.in", "r");
  fscanf(f, "%d %d %d", &task, &rows, &cols);

  for (int r = rows - 1; r >= 0; r--) {
    for (int c = cols - 1; c >= 0; c--) {
      fscanf(f, "%hd", &a[r][c]);
    }
  }

  fclose(f);
}

// Numără de la (dr,dc) și avansînd cu (dr,dc). Pentru distanța pe orizontală
// ar trebui să folosim numărul iterației, dar orice combinație liniară de r
// și c este bună. Deci folosim (r+c) ca să eliminăm o variabilă.
void count_direction(int dr, int dc) {
  int max_h = 0, max_g = 1;

  for (int r = dr, c = dc;
       r < rows && c < cols;
       r += dr, c += dc) {
    if (a[r][c] * max_g > max_h * (r + c)) {
      result++;
      max_h = a[r][c];
      max_g = r + c;
    }
    a[r][c] = 0;
  }
}

void count_visible() {
  if (task == 1) {
    count_direction(0, 1);
    count_direction(1, 0);
  } else {
    for (int dr = 0; dr < rows; dr++) {
      for (int dc = 0; dc < cols; dc++) {
        if (a[dr][dc]) {
          count_direction(dr, dc);
        }
      }
    }
  }
}
```

```
void write_result() {
  FILE* f = fopen("ace.out", "w");
  fprintf(f, "%d\n", result);
  fclose(f);
}

int main() {
  read_matrix();
  count_visible();
  write_result();
  return 0;
}
```

O.3 Problema Baba Oarba (Lot 2016)

‹ înapoi • versiune online

```
#include "babaoarba.h"
#include <math.h>

const int NUM_JIGGLES = 25;
const int CLOSER = 1;
const int FURTHER = 0;
const int FOUND = -1;

void play() {
  double l = 0.0, r = 2 * M_PI;
  bool found = false;

  int iter = 0;
  do {
    double mid = (l + r) / 2;
    int answer = makeStep(mid + M_PI / 2);
    if (answer == CLOSER) {
      l = mid;
      // There and back again.
      makeStep(mid - M_PI / 2);
    } else if (answer == FURTHER) {
      r = mid;
      makeStep(mid - M_PI / 2);
    } else {
      found = true;
    }
  } while (!found && (++iter < NUM_JIGGLES));

  double angle = (l + r) / 2;

  while (!found) {
```

```
    found = (makeStep(angle) == FOUND);
  }
}
```

Un fișier `babaoarba.h`, util pentru testarea offline:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

const int MAX_COORD = 70'000;

extern void play();

struct point {
  double x, y;

  double dist2() {
    return x * x + y * y;
  }

};

point tassadar;

void init_rng() {
  struct timeval time;
  gettimeofday(&time, NULL);
  int micros = time.tv_sec * 1'000'000 + time.tv_usec;
  srand(micros);
}

int makeStep(double angle) {
  printf("Got called with angle %.6lf\n", angle);
  point new_t = {
    tassadar.x + cos(angle),
    tassadar.y + sin(angle),
  };

  int result;
  if (new_t.dist2() <= 1) {
    result = -1;
  } else if (new_t.dist2() < tassadar.dist2()) {
    result = 1;
  } else {
    result = 0;
  }
```

```
  tassadar = new_t;
  printf("New coords (%.6lf,%.6lf), dist %.6lf, result %d\n",
         tassadar.x, tassadar.y, sqrt(tassadar.dist2()), result);

  return result;
}

int main() {
  init_rng();

  tassadar.x = rand() % MAX_COORD + 2;
  tassadar.y = rand() % MAX_COORD + 2;
  printf("Chose %.0lf %.0lf\n", tassadar.x, tassadar.y);
  play();

  return 0;
}
```

O.4 Problema Arhitect (OJI 2023, clasa a 10-a)

‹ înapoi

Sursă cu perechi (versiune online).

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

const int MAX_N = 100'000;

struct slope {
  int dx, dy;

  // Ne interesează doar pante în [0°, 90°). Vom reformula o pantă de 110° ca
  // pe o pantă de 20° pentru a le număra împreună.
  void set(int dx, int dy) {
    // Rotește cu 90° până când ajungem la x > 0 și y >= 0.
    while ((dx <= 0) || (dy < 0)) {
      int tmp = dy;
      dy = -dx;
      dx = tmp;
    }
    this->dx = dx;
    this->dy = dy;
  }

  bool operator ==(const slope& o) const {
    return (long long)dx * o.dy == (long long)dy * o.dx;
  }
```

```
  bool operator <(const slope& o) const {
    return (long long)dx * o.dy < (long long)dy * o.dx;
  }
};

slope s[MAX_N];
int n;

void read_slopes() {
  FILE* f = fopen("arhitect.in", "r");
  fscanf(f, "%d", &n);
  for (int i = 0; i < n; i++) {
    int x1, y1, x2, y2;
    fscanf(f, "%d %d %d %d", &x1, &y1, &x2, &y2);
    s[i].set(x2 - x1, y2 - y1);
  }
  fclose(f);
}

int count_duplicates() {
  std::sort(s, s + n);
  int run = 1, max_run = 1;
  for (int i = 1; i < n; i++) {
    run = (s[i] == s[i - 1]) ? (run + 1) : 1;
    if (run > max_run) {
      max_run = run;
    }
  }
  return max_run;
}

void write_result(int result) {
  FILE* f = fopen("arhitect.out", "w");
  fprintf(f, "%d\n", result);
  fclose(f);
}

int main() {
  read_slopes();
  int result = count_duplicates();
  write_result(result);
  return 0;
}
```

Sursă cu numere reale (versiune online).

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

```cpp
const int MAX_N = 100'000;

double s[MAX_N];
int n;

// Ne interesează doar pante în [0°, 90°). Vom reformula o pantă de 110° ca pe
// o pantă de 20° pentru a le număra împreună.
double make_slope(int dx, int dy) {
  // Rotește cu 90° pînă cînd ajungem la x > 0 și y >= 0.
  while ((dx <= 0) || (dy < 0)) {
    int tmp = dy;
    dy = -dx;
    dx = tmp;
  }
  return (double)dy / dx;
}

void read_slopes() {
  FILE* f = fopen("arhitect.in", "r");
  fscanf(f, "%d", &n);
  for (int i = 0; i < n; i++) {
    int x1, y1, x2, y2;
    fscanf(f, "%d %d %d %d", &x1, &y1, &x2, &y2);
    s[i] = make_slope(x2 - x1, y2 - y1);
  }
  fclose(f);
}

int count_duplicates() {
  std::sort(s, s + n);
  int run = 1, max_run = 1;
  for (int i = 1; i < n; i++) {
    run = (s[i] == s[i - 1]) ? (run + 1) : 1;
    if (run > max_run) {
      max_run = run;
    }
  }
  return max_run;
}

void write_result(int result) {
  FILE* f = fopen("arhitect.out", "w");
  fprintf(f, "%d\n", result);
  fclose(f);
}

int main() {
  read_slopes();
  int result = count_duplicates();
```

```
  write_result(result);
  return 0;
}
```

O.5 Problema Elicoptere (OJI 2016, clasele 11-12)

înapoi • versiune online

```
#include <algorithm>
#include <math.h>
#include <stdio.h>

const int MAX_N = 100;
const int MAX_EDGES = MAX_N * (MAX_N - 1) / 2;

struct point {
  int x, y;
};

struct triangle {
  point a, b, c;
};

struct edge {
  int u, v;
  double d;
};

triangle t[MAX_N];
edge e[MAX_EDGES];
int comp[MAX_N];
int n, num_edges, k, task, num_selected_edges, num_connected_pairs;
double total_length;

void read_data() {
  FILE* f = fopen("elicoptere.in", "r");
  fscanf(f, "%d %d %d", &task, &n, &k);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d %d %d %d %d %d",
           &t[i].a.x, &t[i].a.y, &t[i].b.x, &t[i].b.y, &t[i].c.x, &t[i].c.y);
  }
  fclose(f);
}

double min(double a, double b) {
  return (a < b) ? a : b;
}
```

```
double min3(double a, double b, double c) {
  return min(min(a, b), c);
}

double dist_horiz(point p, point a, point b) {
  if ((a.y == p.y) && (p.y == b.y)) {
    // a, b, p coliniare pe orizontală
    return min(fabs(p.x - a.x), fabs(p.x - b.x));
  } else if ((long long)(p.y - a.y) * (p.y - b.y) <= 0) {
    // p.y se află între a.y și b.y
    double x = a.x + (double)(p.y - a.y) * (b.x - a.x) / (b.y - a.y);
    return fabs(p.x - x);
  } else {
    // orizontala din p *nu* întîlnește ab; returnează infinit
    return k + 1;
  }
}

double dist_point_segment(point p, point a, point b) {
  // Pentru a afla distanța pe verticală, oglindim punctele față de diagonală.
  return min(dist_horiz(p, a, b),
             dist_horiz({p.y, p.x}, {a.y, a.x}, {b.y, b.x}));
}

double dist_point_triangle(point p, triangle t) {
  return min3(dist_point_segment(p, t.a, t.b),
              dist_point_segment(p, t.b, t.c),
              dist_point_segment(p, t.c, t.a));
}

double dist_triangle_triangle(triangle& g, triangle& h) {
  return min(min3(dist_point_triangle(g.a, h),
                  dist_point_triangle(g.b, h),
                  dist_point_triangle(g.c, h)),
             min3(dist_point_triangle(h.a, g),
                  dist_point_triangle(h.b, g),
                  dist_point_triangle(h.c, g)));
}

void create_edges() {
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      double d = dist_triangle_triangle(t[i], t[j]);
      if (d <= k) {
        e[num_edges++] = { i, j, d };
      }
    }
  }
}
```

```
void init_comp() {
  for (int i = 0; i < n; i++) {
    comp[i] = i;
  }
}

void unite(int u, int v) {
  for (int i = 0; i < n; i++) {
    if (comp[i] == u) {
      comp[i] = v;
    }
  }
}

void kruskal() {
  std::sort(e, e + num_edges, [](edge a, edge b) {
    return a.d < b.d;
  });

  init_comp();

  for (int i = 0; i < num_edges; i++) {
    if (comp[e[i].u] != comp[e[i].v]) {
      unite(comp[e[i].u], comp[e[i].v]);
      num_selected_edges++;
      total_length += e[i].d;
    }
  }
}

void write_answer() {
  FILE* f = fopen("elicoptere.out", "w");
  if (task == 1) {
    fprintf(f, "%d\n", num_selected_edges);
  } else if (task == 2) {
    fprintf(f, "%d\n", num_connected_pairs);
  } else {
    fprintf(f, "%0.6lf\n", total_length);
  }
  fclose(f);
}

void count_connected_pairs() {
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      num_connected_pairs += (comp[i] == comp[j]);
    }
  }
}
```

```
int main() {
  read_data();
  create_edges();
  kruskal();
  count_connected_pairs();
  write_answer();
}
```

O.6 Problema Arpa and an Exam About Geometry (Codeforces)

‹ înapoi • versiune online

```
#include <iostream>

long long dist2(long long ax, long long ay, long long bx, long long by) {
  return ((ax - bx) * (ax - bx) + (ay - by) * (ay - by));
}

int main() {
  long long ax, ay, bx, by, cx, cy;
  std::cin >> ax >> ay >> bx >> by >> cx >> cy;

  bool noncollinear = ((cx - ax) * (by - ay) - (cy - ay) * (bx - ax) != 0);
  bool equidistant = dist2(ax, ay, bx, by) == dist2(bx, by, cx, cy);

  std::cout << ((noncollinear && equidistant) ? "Yes\n" : "No\n");

  return 0;
}
```

O.7 Problema Bear and Floodlight (Codeforces)

‹ înapoi • versiune online

```
#define _USE_MATH_DEFINES // https://codeforces.com/blog/entry/51835
#include <math.h>
#include <stdio.h>

const int MAX_N = 20;

double min(double x, double y) {
  return (x < y) ? x : y;
}
```

```
double max(double x, double y) {
  return (x > y) ? x : y;
}

struct floodlight {
  int xc, yc;
  double c, s;

  void setAngle(int angle) {
    double rad = M_PI * angle / 180;
    c = cos(rad);
    s = sin(rad);
  }

  // Extinde segmentul vizibil la dreapta de la x, folosind unghiul acestui
  // reflector.
  double extend(double x, int limit) {
    // Rotește (x, 0) cu <c,s> grade în jurul lui (xc, yc).
    double rotx = (x - xc) * c - (0 - yc)* s + xc;
    double roty = (x - xc) * s + (0 - yc)* c + yc;

    if (roty >= yc) {
      // Reflectorul bate la dreapta sau chiar mai sus, deci acoperă axa Ox
      // până la infinit (și dincolo!)
      return limit;
    }

    // Intersectează segmentul (xc, yc) - (rotx, roty) cu axa Ox.
    double newx = (xc * roty - yc * rotx) / (roty - yc);
    return min(newx, limit);
  }
};

floodlight f[MAX_N];
double bestx[1 << MAX_N];
int n, startx, endx;

void read_data() {
  scanf("%d %d %d", &n, &startx, &endx);
  for (int i = 0; i < n; i++) {
    int angle;
    scanf("%d %d %d", &f[i].xc, &f[i].yc, &angle);
    f[i].setAngle(angle);
  }
}

double dyn_prog() {
  bestx[0] = startx;
  for (int mask = 1; mask < (1 << n); mask++) {
    bestx[mask] = startx;
```

```
    for (int b = 0; b < n; b++) {
      if (mask & (1 << b)) {
        bestx[mask] = max(bestx[mask], f[b].extend(bestx[mask ^ (1 << b)], endx));
      }
    }
  }

  return bestx[(1 << n) - 1] - startx;
}

int main() {
  read_data();
  double result = dyn_prog();
  printf("%0.9lf\n", result);

  return 0;
}
```

O.8 Problema TrapEZZ (Moisil++ 2023, clasele 11-12)

înapoi • versiune online

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

const int MAX_POINTS = 1'000;
const int MAX_SEGMENTS = MAX_POINTS * (MAX_POINTS - 1) / 2;

struct point {
  int x, y;
};

int gcd(int a, int b) {
  while (b) {
    int tmp = a % b;
    a = b;
    b = tmp;
  }
  return a;
}

struct line {
  int a, b, c;

  void normalize() {
    int g = gcd(abs(c), gcd(abs(a), abs(b)));
    if ((a < 0) || ((a == 0) & (b < 0))) {
```

```
      g = -g;
    }
    a /= g;
    b /= g;
    c /= g;
  }

  bool operator ==(line& other) {
    return (a == other.a) && (b == other.b) && (c == other.c);
  }
};

// Pentru segmente cu o mediatoare comună, teoretic trebuie să cunoaștem
// ecuațiile dreptelor. Dar nu ne trebuie ecuația completă -- știm deja că au
// aceleași valori a și b, deoarece sînt paralele. De aceea, stocăm doar
// valorile c, care denotă pozițiile segmentelor pe mediatoare.
struct segment {
  int len2; // lungimea la pătrat
  line bis; // mediatoarea
  int pos;  // poziția pe mediatoare

  void from_points(point p, point q) {
    len2 = (q.x - p.x) * (q.x - p.x) + (q.y - p.y) * (q.y - p.y);
    bis = {
      .a = q.x - p.x,
      .b = q.y - p.y,
      .c = (q.x * q.x - p.x * p.x + q.y * q.y - p.y * p.y) / 2,
    };
    bis.normalize();

    line l = {
      .a = q.y - p.y,
      .b = p.x - q.x,
      .c = p.y * (q.x - p.x) - p.x * (q.y - p.y),
    };
    l.normalize();
    pos = l.c;
  }
};

point p[MAX_POINTS];
segment s[MAX_SEGMENTS + 1];
int num_points, num_segments;

void read_data() {
  FILE* f = fopen("trapezz.in", "r");
  fscanf(f, "%d", &num_points);
  for (int i = 0; i < num_points; i++) {
    fscanf(f, "%d %d", &p[i].x, &p[i].y);
    p[i].x *= 2;
```

```
    p[i].y *= 2; // astfel ca toate mijloacele să aibă coordonate întregi
  }
  fclose(f);
}

void create_segments() {
  for (int i = 0; i < num_points - 1; i++) {
    for (int j = i + 1; j < num_points; j++) {
      s[num_segments++].from_points(p[i], p[j]);
    }
  }
}

void sort_by_bisectors() {
  std::sort(s, s + num_segments, [](segment& s, segment& t) {
    if (s.bis.a != t.bis.a) {
      return s.bis.a < t.bis.a;
    } else if (s.bis.b != t.bis.b) {
      return s.bis.b < t.bis.b;
    } else if (s.bis.c != t.bis.c) {
      return s.bis.c < t.bis.c;
    } else {
      return s.pos < t.pos;
    }
  });
}

void sort_by_length(int start, int end) {
  std::sort(s + start, s + end, [](segment& s, segment& t) {
    return s.len2 < t.len2;
  });
}

// TODO: Codul următoarelor două funcții este aproape duplicat.
int count_collinear_pairs(int start, int end) {
  int result = 0, i = start;
  while (i < end) {
    int j = i + 1;
    while ((j < end) && (s[i].pos == s[j].pos)) {
      j++;
    }
    result += (j - i) * (j - i - 1) / 2;
    i = j;
  }
  return result;
}

int count_same_length_pairs(int start, int end) {
  int result = 0, i = start;
  while (i < end) {
```

```
    int j = i + 1;
    while ((j < end) && (s[i].len2 == s[j].len2)) {
      j++;
    }
    result += (j - i) * (j - i - 1) / 2;
    i = j;
  }
  return result;
}

// Acest grup de segmente arată ca o frigăruie. Toate au o mediatoare comună
// și sînt sortate după poziția pe această mediatoare.
int count_bisector(int start, int end) {
  int result = (end - start) * (end - start - 1) / 2;
  result -= count_collinear_pairs(start, end);
  sort_by_length(start, end);
  result -= count_same_length_pairs(start, end);
  // Notă: nu este nevoie de pinex. Două segmente nu pot fi și coliniare, și
  // de aceeași lungime. Atunci ar fi confundate, or enunțul promite că
  // punctele sînt distincte.
  return result;
}

int count_trapezoids() {
  s[num_segments].bis = { 0, 0, 0 }; // santinelă

  int i = 0, result = 0;
  while (i < num_segments) {
    int j = i + 1;
    while (s[i].bis == s[j].bis) {
      j++;
    }
    result += count_bisector(i, j);
    i = j;
  }
  return result;
}

void write_result(int result) {
  FILE* f = fopen("trapezz.out", "w");
  fprintf(f, "%d\n", result);
  fclose(f);
}

int main() {
  read_data();
  create_segments();
  sort_by_bisectors();
  int result = count_trapezoids();
  write_result(result);
```

```
  return 0;
}
```

O.9 Problema Terenuri (Baraj ONI 2011)

‹ înapoi • versiune online

```
#include <map>
#include <math.h>
#include <stdio.h>

struct point {
  double x, y;

  void translate(point other) {
    x -= other.x;
    y -= other.y;
  }

  double dist2() {
    return x * x + y * y;
  }
};

typedef std::map<double, point>::iterator iter;

std::map<double, point> hull;
point center;

// True dacă și numai dacă ∠ABC virează spre dreapta.
bool reflex_angle(point a, point b, point c) {
  return (b.x - a.x) * (c.y - b.y) - (c.x - b.x) * (b.y - a.y) < 0;
}

// Treci la punctul anterior, circular.
iter prev(iter x) {
  return (x == hull.begin())
    ? std::prev(hull.end())
    : std::prev(x);
}

// Treci la punctul următor, circular.
iter next(iter x) {
  x = std::next(x);
  return (x == hull.end())
    ? hull.begin()
    : x;
```

```
}

// Returnează hull.end() dacă avem deja un punct pe aceeași rază, dar mai
// depărtat de centru. În acest caz nu avem nimic de făcut.
iter insert(double angle, point p) {
  iter it = hull.lower_bound(angle);
  bool exists = (it != hull.end()) && (it->first == angle);

  if (exists && (p.dist2() <= it->second.dist2())) {
    // Exista deja un punct mai depărtat de centru, pe aceeași rază.
    return hull.end();
  }

  if (exists) {
    // Exista deja un punct, dar mai apropiat de centru.
    it->second = p;
  } else {
    // Inserează la poziția deja căutată.
    it = hull.insert(it, {angle, p});
  }

  return it;
}

void update_hull(point p) {
  p.translate(center);
  double angle = atan2(p.y, p.x);

  iter it = insert(angle, p);
  if (it == hull.end()) {
    return;
  }

  // Curățenie printre punctele anterioare.
  iter b = prev(it), a = prev(b);
  while ((hull.size() > 3) && reflex_angle(a->second, b->second, p)) {
    hull.erase(b);
    b = a;
    a = prev(a);
  }

  // Curățenie printre punctele următoare.
  iter c = next(it), d = next(c);
  while ((hull.size() > 3) && reflex_angle(p, c->second, d->second)) {
    hull.erase(c);
    c = d;
    d = next(d);
  }

  // Curăță și punctul însuși.
```

```
  if (reflex_angle(b->second, p, c->second)) {
    hull.erase(it);
  }
}

int main() {
  FILE* fin = fopen("terenuri.in", "r");
  FILE* fout = fopen("terenuri.out", "w");
  int n, m;
  point p, q;

  fscanf(fin, "%d %d", &n, &m);
  fscanf(fin, "%lf %lf %lf %lf", &p.x, &p.y, &q.x, &q.y);
  center = { (p.x + q.x) / 2, (p.y + q.y) / 2 };

  update_hull(p);
  update_hull(q);

  for (int i = 2; i < m + n; i++) {
    fscanf(fin, "%lf %lf", &p.x, &p.y);
    update_hull(p);
    if (i >= n - 1) {
      fprintf(fout, "%lu\n", hull.size());
    }
  }

  fclose(fin);
  fclose(fout);
  return 0;
}
```

O.10 Problema Metin2 (Finala IIOT 2021-22)

‹ înapoi • versiune online

```
#include <algorithm>
#include <deque>
#include <math.h>
#include <stdio.h>

const int MAX_STONES = 100'000;

struct ipoint {
  int x, y;
};

struct dpoint {
  double x, y;
```

```
};

struct line {
  long long a, b, c;
  ipoint p; // un punct cunoscut a fi pe linie
  double angle;

  void from_points(ipoint p, ipoint q) {
    a = p.y - q.y;
    b = q.x - p.x;
    c = (long long)p.x * q.y - (long long)q.x * p.y;
    this->p = p;
    angle = atan2(q.y - p.y, q.x - p.x);
  }

  bool has_left(ipoint p) {
    return a * p.x + b * p.y + c > 0;
  }

  bool has_right(dpoint p) {
    return a * p.x + b * p.y + c < 0;
  }

  dpoint intersect(line h) {
    return {
      (double)(b * h.c - c * h.b) / (a * h.b - b * h.a),
      (double)(c * h.a - a * h.c) / (a * h.b - b * h.a),
    };
  }
};

line l[4 * MAX_STONES];
std::deque<int> dl; // indicii liniilor
std::deque<dpoint> poly;
int num_lines;

void read_data() {
  int n;
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    ipoint p, q, r, s;
    scanf("%d %d %d %d %d %d %d %d",
          &p.x, &p.y, &q.x, &q.y, &r.x, &r.y, &s.x, &s.y);
    l[num_lines++].from_points(p, q);
    l[num_lines++].from_points(q, r);
    l[num_lines++].from_points(r, s);
    l[num_lines++].from_points(s, p);
  }
}
```

```
void sort_lines() {
  std::sort(l, l + num_lines, [](line& l1, line& l2) {
    return
      (l1.angle < l2.angle) ||
      ((l1.angle == l2.angle) && l2.has_left(l1.p));
  });
}

void unique_lines() {
  int j = 1;
  for (int i = 1; i < num_lines; i++) {
    if (l[i].angle != l[i - 1].angle) {
      l[j++] = l[i];
    }
  }
  num_lines = j;
}

void build_intersection() {
  dl.push_front(0);
  for (int i = 1; i < num_lines; i++) {
    while (!poly.empty() && (l[i].has_right(poly.front()))) {
      dl.pop_front();
      poly.pop_front();
    }
    poly.push_front(l[i].intersect(l[dl.front()]));
    dl.push_front(i);
  }

  unsigned old_size;
  do {
    old_size = dl.size();
    if (l[dl.front()].has_right(poly.back())) {
      dl.pop_back();
      poly.pop_back();
    }

    if (l[dl.back()].has_right(poly.front())) {
      dl.pop_front();
      poly.pop_front();
    }
  } while (dl.size() != old_size);

  poly.push_front(l[dl.front()].intersect(l[dl.back()]));
}

long double compute_area() {
  poly.push_back(poly.front());
  long double area = 0;
```

```
  for (unsigned i = 0; i < poly.size() - 1; i++) {
    area += (poly[i + 1].x - poly[i].x) * (poly[i].y + poly[i + 1].y);
  }

  return area / 2;
}

int main() {
  read_data();
  sort_lines();
  unique_lines();
  build_intersection();
  long double area = compute_area();

  printf("%.9Lf\n", area);
  return 0;
}
```

Anexa P

Geometrie - Algoritmi specifici

P.1 Problema Copaci (Infoarena)

‹ înapoi • versiune online

```cpp
// Teorema lui Pick: A = I + B/2 - 1, deci I = A + 1 - B/2. În practică:
// 2I = 2A + 1 - B, căci formula pentru arie calculează dublul ariei.
#include <stdio.h>
#include <stdlib.h>

struct point {
  long long x, y;
};

int gcd(point p, point q) {
  long long x = llabs(p.x - q.x);
  long long y = llabs(p.y - q.y);
  while (y) {
    int tmp = x % y;
    x = y;
    y = tmp;
  }
  return x;
}

long long vector(point p, point q) {
  return p.x * q.y - p.y * q.x;
}

int main() {
  point first, prev, p;
  long long boundary = 0, twice_area = 0;
  int n;

  FILE* f = fopen("copaci.in", "r");
```

```
  fscanf(f, "%d %lld %lld", &n, &first.x, &first.y);
  prev = first;

  for (int i = 1; i < n; i++) {
    fscanf(f, "%lld %lld", &p.x, &p.y);
    twice_area += vector(prev, p);
    boundary += gcd(prev, p);
    prev = p;
  }
  twice_area += vector(prev, first);
  boundary += gcd(prev, first);

  fclose(f);

  f = fopen("copaci.out", "w");
  fprintf(f, "%lld\n", (llabs(twice_area) + 2 - boundary) / 2);
  fclose(f);

  return 0;
}
```

P.2 Problema Emptri (Lot 2015)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 1'000'000;

int phi[MAX_N + 1];

int read_n() {
  int n;
  FILE* f = fopen("emptri.in", "r");
  fscanf(f, "%d", &n);
  fclose(f);
  return n;
}

void compute_phi(int n) {
  for (int i = 1; i <= n; i++) {
    phi[i] = i;
  }
  for (int i = 2; i <= n; i++) {
    if (phi[i] == i) {
      for (int j = i; j <= n; j += i) {
        phi[j] -= phi[j] / i;
      }
```

```
    }
  }
}

long long sum_phi(int n) {
  long long sum = 0;
  for (int i = 2; i <= n; i++) {
    sum += phi[i];
  }
  return sum * 2 + 1;
}

void write_answer(long long answer) {
  FILE* f = fopen("emptri.out", "w");
  fprintf(f, "%lld\n", answer);
  fclose(f);
}

int main() {
  int n = read_n();
  compute_phi(n);
  long long answer = sum_phi(n);
  write_answer(answer);

  return 0;
}
```

P.3 Problema Înfășurătoare convexă (Infoarena)

‹ înapoi • versiune online

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

const int MAX_N = 120'000;

struct point {
  double x, y;
};

point p[MAX_N];
int st[MAX_N], ss;  // stivă pentru Graham's scan
int n;

void read_data() {
  FILE *f = fopen("infasuratoare.in", "r");
  fscanf(f, "%d", &n);
  for (int i = 0; i < n; i++) {
```

```
    fscanf(f, "%lf %lf", &p[i].x, &p[i].y);
  }
  fclose(f);
}

void find_extreme() {
  int min = 0;
  for (int i = 1; i < n; i++) {
    if ((p[i].y < p[min].y) || (p[i].y == p[min].y && p[i].x < p[min].x)) {
      min = i;
    }
  }
  std::swap(p[0], p[min]);
}

double orientation(point a, point b, point c) {
  return (b.x - a.x) * (c.y - b.y) - (c.x - b.x) * (b.y - a.y);
}

double dist2(point a, point b) {
  return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
}

void polar_sort() {
  std::sort(p + 1, p + n, [](point a, point b) {
    return orientation(p[0], a, b) > 0;
  });
}

void graham_scan() {
  ss = 0;
  for (int i = 0; i < n; i++) {
    while ((ss >= 2) &&
           (orientation(p[st[ss - 2]], p[st[ss - 1]], p[i]) < 0)) {
      ss--;
    }
    st[ss++] = i;
  }
}

void write_hull() {
  FILE* f = fopen("infasuratoare.out", "w");
  fprintf(f, "%d\n", ss);
  for (int i = 0; i < ss; i++) {
    fprintf(f, "%.6lf %.6lf\n", p[st[i]].x, p[st[i]].y);
  }
  fclose(f);
}

int main(void) {
```

```
  read_data();
  find_extreme();
  polar_sort();
  graham_scan();
  write_hull();

  return 0;
}
```

P.4 Problema Înfășurătoare convexă (NerdArena)

‹ înapoi • versiune online

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

const int MAX_N = 100'000;

struct point {
  int x, y;
};

point p[MAX_N];
int st[MAX_N], ss;  // stivă cu indicii punctelor
int n;

void read_data() {
  FILE *f = fopen("infasuratoare.in", "r");
  fscanf(f, "%d", &n);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d %d", &p[i].x, &p[i].y);
  }
  fclose(f);
}

void find_extreme() {
  int min = 0;
  for (int i = 1; i < n; i++) {
    if ((p[i].y < p[min].y) || (p[i].y == p[min].y && p[i].x < p[min].x)) {
      min = i;
    }
  }
  std::swap(p[0], p[min]);
}

long long orientation(point a, point b, point c) {
  return ((long long)(b.x - a.x)) * (c.y - b.y)
    - ((long long)(c.x - b.x)) * (b.y - a.y);
```

```
}

long long dist2(point a, point b) {
  return ((long long)(a.x - b.x)) * (a.x - b.x)
    + ((long long)(a.y - b.y)) * (a.y - b.y);
}

void polar_sort() {
  std::sort(p + 1, p + n, [] (point a, point b) {
    long long disc = orientation(p[0], a, b);
    if (disc == 0) {
      // a și b sînt coliniare cu p[0]; primul vrem să fie cel mai apropiat
      return dist2(a, p[0]) < dist2(b, p[0]);
    } else {
      return disc > 0;
    }
  });
}

void graham_scan() {
  ss = 0;
  for (int i = 0; i < n; i++) {
    while ((ss >= 2) &&
           (orientation(p[st[ss - 2]], p[st[ss - 1]], p[i]) <= 0)) {
      ss--;
    }
    st[ss++] = i;
  }
}

void write_hull() {
  FILE* f = fopen("infasuratoare.out", "w");
  fprintf(f, "%d\n", ss);
  for (int i = 0; i < ss; i++) {
    fprintf(f, "%d %d\n", p[st[i]].x, p[st[i]].y);
  }
  fclose(f);
}

int main(void) {
  read_data();
  find_extreme();
  polar_sort();
  graham_scan();
  write_hull();

  return 0;
}
```

P.5 Problema Magic (JBOI 2023)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_WIZARDS = 200'000;

struct wizard {
  int x, e;
  int p, q;
};

wizard w[MAX_WIZARDS];
int st[MAX_WIZARDS], ss;
int n;

void read_input_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &w[i].x, &w[i].e);
  }
}

// Test standard de orientare trigonometrică.
bool is_reflex(int i, int j, int x, int e) {
  return
    (long long)(w[i].x - x) * (w[j].e - e) >
    (long long)(w[j].x - x) * (w[i].e - e);
}

// Elimină vrăjitorul anterior din stivă cît vreme vrăjitorul curent (A) este
// mai sus sau (B) face un unghi reflex cu cei doi vrăjitori anteriori.
void evict(int x, int e) {
  while ((ss && (e >= w[st[ss - 1]].e)) ||
         ((ss > 1) && is_reflex(st[ss - 2], st[ss - 1], x, e))) {
    ss--;
  }
}

// Vrăjitorul me are un nou mentor c. Vezi dacă este mai bun decît vechiul
// mentor.
void compare_mentors(int me, int c) {
  // Dacă q nu există, atunci evident alege p/q din c. Altfel compară vechea
  // și noua pantă.
  if (!w[me].q ||
      ((long long)w[c].e * w[me].q >
       (long long)w[me].p * (w[me].x - w[c].x))) {
    w[me].p = w[c].e;
```

```
    w[me].q = w[me].x - w[c].x;
  }
}

void iterate_wizards() {
  ss = 0; // mărimea stivei
  st[ss++] = 0;

  for (int i = 1; i < n; i++) {
    // Mai întîi privește de la nivelul solului și alege-ți un mentor.
    evict(w[i].x, 0);
    compare_mentors(i, st[ss - 1]);

    // Apoi privește mai de sus și continuă să elimini.
    evict(w[i].x, w[i].e);
    st[ss++] = i;
  }
}

// Oglindește coordonatele x. Astfel scăpăm de nevoia de a copia algoritmul cu
// stivă de la stînga la dreapta și de la dreapta la stînga.
void reverse_wizards() {
  int min_coord = w[0].x, max_coord = w[n - 1].x;

  for (int i = 0, j = n - 1; i < j; i++, j--) {
    wizard tmp = w[i];
    w[i] = w[j];
    w[j] = tmp;
  }

  for (int i = 0; i < n; i++) {
    w[i].x = min_coord + max_coord - w[i].x;
  }
}

void find_mentors() {
  iterate_wizards();
  reverse_wizards();
  iterate_wizards();
  reverse_wizards();
}

int gcd(int x, int y) {
  while (y) {
    int tmp = x;
    x = y;
    y = tmp % y;
  }
  return x;
}
```

```
void write_reduced_fractions() {
  for (int i = 0; i < n; i++) {
    int d = gcd(w[i].p, w[i].q);
    printf("%d %d\n", w[i].p / d, w[i].q / d);
  }
}

int main() {
  read_input_data();
  find_mentors();
  write_reduced_fractions();

  return 0;
}
```

P.6 Problema Triangular Queries (CodeChef)

‹ înapoi • versiune online

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

const int MAX_N = 300000;
const int MAX_Q = 200000;
const int MAX_COORD = 300000;

// Ordinea este importanta.  Punctul trebuie adăugat *după* ce procesăm orice
// bază care îl conține, ca sa nu îl scădem. În schimb, punctul trebuie
// adăugat *înainte* să procesăm orice vîrf de triunghi cu care se suprapune,
// ca să îl luăm în calcul.
const int T_BASE = 0;
const int T_POINT = 1;
const int T_TIP = 2;

struct point {
  int x, y;
};

struct triangle {
  int x, y, d;
  int answer;
};

struct event {
  int diag;
  int obj_id;
  unsigned char type;
```

```
};

struct fenwick {
  int v[MAX_COORD + 1];

  void inc(int pos) {
    do {
      v[pos]++;
      pos += pos & -pos;
    } while (pos <= MAX_COORD);
  }

  int partial_sum(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }
};

point p[MAX_N];
triangle t[MAX_Q];
event e[MAX_N + 2 * MAX_Q];
fenwick fx, fy;
int num_points, num_triangles, num_events;
int points_seen;

void read_data() {
  scanf("%d %d", &num_points, &num_triangles);
  for (int i = 0; i < num_points; i++) {
    scanf("%d %d", &p[i].x, &p[i].y);
  }
  for (int i = 0; i < num_triangles; i++) {
    scanf("%d %d %d", &t[i].x, &t[i].y, &t[i].d);
  }
}

void create_events() {
  for (int i = 0; i < num_points; i++) {
    e[num_events++] = {
      .diag = p[i].x + p[i].y,
      .obj_id = i,
      .type = T_POINT,
    };
  }
  for (int i = 0; i < num_triangles; i++) {
    e[num_events++] = {
      .diag = t[i].x + t[i].y + t[i].d,
```

```
      .obj_id = i,
      .type = T_BASE,
    };
    e[num_events++] = {
      .diag = t[i].x + t[i].y,
      .obj_id = i,
      .type = T_TIP,
    };
  }
}

void sort_events() {
  std::sort(e, e + num_events, [](event a, event b) {
    return (a.diag > b.diag) ||
      ((a.diag == b.diag) && (a.type < b.type));
  });
}

void base_event(int id) {
  t[id].answer = points_seen
    - fx.partial_sum(t[id].x - 1)
    - fy.partial_sum(t[id].y - 1);
}

void point_event(int id) {
  fx.inc(p[id].x);
  fy.inc(p[id].y);
  points_seen++;
}

void tip_event(int id) {
  int total = points_seen
    - fx.partial_sum(t[id].x - 1)
    - fy.partial_sum(t[id].y - 1);
  t[id].answer = total - t[id].answer;
}

void process_events() {
  for (int i = 0; i < num_events; i++) {
    switch (e[i].type) {
      case T_BASE: base_event(e[i].obj_id); break;
      case T_POINT: point_event(e[i].obj_id); break;
      case T_TIP: tip_event(e[i].obj_id); break;
    }
  }
}

void write_answers() {
  for (int i = 0; i < num_triangles; i++) {
    printf("%d\n", t[i].answer);
```

```
  }
}

int main() {
  read_data();
  create_events();
  sort_events();
  process_events();
  write_answers();

  return 0;
}
```

P.7 Problema Hill Walk (USACO Gold 2013)

înapoi

```
#include <algorithm>
#include <set>
#include <stdio.h>

const int MAX_N = 100'000;

// Returnează true dacă şi numai dacă (x1,y1), (x2,y2) şi (x3,y3) sînt în
// ordine trigonometrică.
long long orientation(int x1, int y1, int x2, int y2, int x3, int y3) {
  return ((long long)(x2 - x1)) * (y3 - y2)
    - ((long long)(x3 - x2)) * (y2 - y1);
}

struct segment {
  int x1, y1, x2, y2;
  int ind;

  // Returneaza true dacă şi numai dacă acest segment este „sub” o.
  bool operator<(segment const& o) const {
    if (x2 < o.x2) {
      // Acest segment *nu* este sub o dacă (x2,y2) cade pe o, adică dacă
      // (o.x1,o.y1), (o.x2,o.y2) şi (x2,y2) sînt în ordine
      // antitrigonometrică.
      return orientation(o.x1, o.y1, o.x2, o.y2, x2, y2) < 0;
                } else {
      // Acest segment *este* sub o dacă (o.x2,o.y2) cade pe acest segment,
      // adică dacă (x1,y1), (x2,y2) şi (o.x2,o.y2) sînt în ordine
      // trigonometrică.
      return orientation(x1, y1, x2, y2, o.x2, o.y2) > 0;
                }
  }
```

```
};

struct event {
  int ind;
  int x, y;
};

segment seg[MAX_N];
event evt[2 * MAX_N];
std::set<segment> set;
int n, bessie, hill_count;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d %d %d", &seg[i].x1, &seg[i].y1, &seg[i].x2, &seg[i].y2);
    seg[i].ind = i;
  }
}

void create_and_sort_events() {
  for (int i = 0; i < n; i++) {
    evt[2 * i] = { i, seg[i].x1, seg[i].y1 };
    evt[2 * i + 1] = { i, seg[i].x2, seg[i].y2 };
  }

  std::sort(evt, evt + 2 * n, [](event& a, event& b) {
    return (a.x < b.x) || ((a.x == b.x) && (a.y < b.y));
  });
}

void process_end(event& e) {
  std::set<segment>::iterator it = set.find(seg[e.ind]);

  if (e.ind == bessie) {
    if (it == set.begin()) {
      bessie = n;
    } else {
      std::set<segment>::iterator prev = std::prev(it);
      bessie = prev->ind;
      hill_count++;
    }
  }

  set.erase(it);
}

void sweep_line() {
  bessie = 0;
  hill_count = 1;
```

```
  for (int i = 0; i < 2 * n; i++) {
    event& e = evt[i];
    segment& s = seg[e.ind];
    if (e.x == s.x1) {
      set.insert(s);
    } else {
      process_end(e);
    }
  }
}

int main() {
  read_data();
  create_and_sort_events();
  sweep_line();
  printf("%d\n", hill_count);

  return 0;
}
```

P.8 Problema Ydist (Lot 2014)

‹ înapoi • versiune online

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

const int MAX_N = 100'000;
const int MAX_Q = 100'000;
const int NONE = -1;

struct point {
  int x, y;
  int qindex; // sau NONE dacă acest punct este o bilă, nu o interogare
};

point p[MAX_N + MAX_Q];
int st[MAX_N], ss; // stiva de puncte-candidat
double ans[MAX_Q];
int n, q;

void read_data() {
  FILE* f = fopen("ydist.in", "r");
  fscanf(f, "%d %d", &n, &q);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d %d", &p[i].x, &p[i].y);
    p[i].qindex = NONE;
```

```
  }
  for (int i = 0; i < q; i++) {
    fscanf(f, "%d %d", &p[i + n].x, &p[i + n].y);
    p[i + n].qindex = i;
  }
  fclose(f);
}

void polar_sort() {
  // Procesează bilele înaintea interogărilor pe o rază. Vrem să luăm bilele
  // în calcul pentru interogări (cu distanța 0).
  std::sort(p, p + n + q, [](point a, point b) {
    long long disc = (long long)a.y * b.x - (long long)a.x * b.y;
    return (disc > 0) ||
      ((disc == 0) & (a.qindex == NONE));
  });
}

long long orientation(point& a, point& b, point& c) {
  return (long long)(b.x - a.x) * (c.y - b.y)
    - (long long)(c.x - b.x) * (b.y - a.y);
}

void process_point(int i) {
  // Elimină punctele la NE de p[i].
  while (ss &&
         ((p[st[ss - 1]].x >= p[i].x) ||
          (p[st[ss - 1]].y >= p[i].y))) {
    ss--;
  }

  // Elimină punctele care încalcă concavitatea stivei.
  while ((ss >= 2) &&
         (orientation(p[st[ss - 2]], p[st[ss - 1]], p[i]) < 0)) {
    ss--;
  }
  st[ss++] = i;
}

// Calculează distanța verticală de la p la raza (0,0)-r.
double dist_to_ray(point p, point r) {
  return p.y - (double)r.y * p.x / r.x;
}

void process_query(int i) {
  // Dacă ultimul punct din stivă este mai departe de rază decît penultimul,
  // el poate fi șters (pentru razele viitoare el va fi și mai departe).
  double d1 = dist_to_ray(p[st[ss - 1]], p[i]), d2;
  while ((ss >= 2) &&
         ((d2 = dist_to_ray(p[st[ss - 2]], p[i])) < d1)) {
```

```
    ss--;
    d1 = d2;
  }
  ans[p[i].qindex] = d1;
}

void scan_points() {
  for (int i = 0; i < n + q; i++) {
    if (p[i].qindex == NONE) {
      process_point(i);
    } else {
      process_query(i);
    }
  }
}

void write_answers() {
  FILE* f = fopen("ydist.out", "w");
  for (int i = 0; i < q; i++) {
    fprintf(f, "%.7f\n", ans[i]);
  }
  fclose(f);
}

int main() {
  read_data();
  polar_sort();
  scan_points();
  write_answers();

  return 0;
}
```

P.9 Problema Fossil in the Ice (CodeChef)

‹ înapoi • versiune online

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

const int MAX_N = 100000;

typedef struct {
  int x, y;
} point;

point p[MAX_N];
int st[MAX_N];  // stivă pentru Graham's scan
```

```
int n;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &p[i].x, &p[i].y);
  }
}

long long area(point a, point b, point c) {
  return
    (long long)(b.x - a.x) * (c.y - b.y) -
    (long long)(b.y - a.y) * (c.x - b.x);
}

long long dist2(point a, point b) {
  return
    (long long)(a.x - b.x) * (a.x - b.x) +
    (long long)(a.y - b.y) * (a.y - b.y);
}

void find_min_y() {
  int min = 0;
  for (int i = 1; i < n; i++) {
    if ((p[i].y < p[min].y) || (p[i].y == p[min].y && p[i].x < p[min].x)) {
      min = i;
    }
  }
  std::swap(p[0], p[min]);
}

void polar_sort() {
  std::sort(p + 1, p + n, [](point a, point b) {
    long long d = area(p[0], a, b);
    return (d > 0) ||
      ((d == 0) && (dist2(a, p[0]) < dist2(b, p[0])));
  });
}

void graham_scan() {
  int ss = 0;
  for (int i = 0; i < n; i++) {
    while ((ss >= 2) &&
           (area(p[st[ss - 2]], p[st[ss - 1]], p[i]) <= 0)) {
      ss--;
    }
    st[ss++] = i;
  }

  // Colectează punctele de pe înfășurătoare în ordine trigonometrică.
```

```
  for (int i = 0; i < ss; i++) {
    p[i] = p[st[i]];
  }
  n = ss;
}

long long max(long long a, long long b) {
  return (a > b) ? a : b;
}

int next(int k) {
  return (k == n - 1) ? 0 : (k + 1);
}

long long rotating_calipers() {
  int j = 0, j2 = 1;
  long long result = 0;

  for (int i = 0; i < n; i++) {
    // Mică optimizare: calculează d(i, j) o singură dată.
    long long d = dist2(p[i], p[j]), d2;
    while ((d2 = dist2(p[i], p[j2])) > d) {
      d = d2;
      j = j2;
      j2 = next(j2);
    }
    result = max(result, d);
    // Invariant: j este cel mai depărtat punct de i.
  }

  return result;
};

long long find_diameter() {
  if (n <= 1) {
    return 0;
  } else if (n == 2)  {
    return dist2(p[0], p[1]);
  } else {
    find_min_y();
    polar_sort();
    graham_scan();
    return rotating_calipers();
  }
}

int main(void) {
  int num_tests;

  scanf("%d", &num_tests);
```

```
  while (num_tests--) {
    read_data();
    long long diameter = find_diameter();
    printf("%lld\n", diameter);
  }

  return 0;
}
```

P.10 Problema Rubarba (Infoarena)

‹ înapoi • versiune online

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

const int MAX_N = 100'000;

struct point {
  int x, y;

  point operator - (const point& other) {
    return { x - other.x, y - other.y };
  }
};

point p[MAX_N];
int st[MAX_N];  // stivă pentru Graham's scan
int n;

void read_data() {
  FILE* f = fopen("rubarba.in", "r");
  fscanf(f, "%d", &n);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d %d", &p[i].x, &p[i].y);
  }
  fclose(f);
}

void find_min_y() {
  int min = 0;
  for (int i = 1; i < n; i++) {
    if ((p[i].y < p[min].y) || (p[i].y == p[min].y && p[i].x < p[min].x)) {
      min = i;
    }
  }
  std::swap(p[0], p[min]);
```

```
}

long long dot(point a, point b) {
  return (long long)a.x * b.x + (long long)a.y * b.y;
}

long long area(point a, point b, point c) {
  return
    (long long)(b.x - a.x) * (c.y - b.y) -
    (long long)(b.y - a.y) * (c.x - b.x);
}

long long dist2(point a, point b) {
  return dot(b - a, b - a);
}

void polar_sort() {
  std::sort(p + 1, p + n, [](point a, point b) {
    long long d = area(p[0], a, b);
    return (d > 0) ||
      ((d == 0) && (dist2(a, p[0]) < dist2(b, p[0])));
  });
}

void graham_scan() {
  int ss = 0;
  for (int i = 0; i < n; i++) {
    while ((ss >= 2) &&
           (area(p[st[ss - 2]], p[st[ss - 1]], p[i]) <= 0)) {
      ss--;
    }
    st[ss++] = i;
  }

  // Colectează punctele de pe înfășurătoare în ordine trigonometrică.
  for (int i = 0; i < ss; i++) {
    p[i] = p[st[i]];
  }
  n = ss;
}

int next(int k) {
  return (k == n - 1) ? 0 : (k + 1);
}

// Returnează aria triunghiului sprijinit pe b1b2 cu extremele r (dreapta), a
// (opus) și l (stînga).
//
// Înălțimea dreptunghiului este distanța de la a la b1b2, pe care o putem
// exprima ca 2 * aria(Δab1b2) / |b1b2|. Lățimea dreptunghiului este proiecția
```

```
// vectorului r - l pe dreapta b1b2. Aceasta prin definiție este (r - l) ·
// b1b2 / |b1b2| (produsul scalar). Aria dreptunghiului este deci:
//
// (r - l) · b1b2 * aria(Δab1b2) / |b1b2|^2
long double rectangle_area(int b1, int b2, int r, int a, int l) {
  return (long double)dot(p[b2] - p[b1], p[r] - p[l])
    / dist2(p[b1], p[b2])
    * area(p[b1], p[b2], p[a]);
}

void find_extremes(int b1, int& b2, int& r, int& a, int& l) {
  b2 = next(b1);
  point base = p[b2] - p[b1];

  while (dot(base, p[next(r)] - p[b1]) > dot(base, p[r] - p[b1])) {
    r = next(r);
  }

  a = r;
  while (area(p[b1], p[b2], p[next(a)]) > area(p[b1], p[b2], p[a])) {
    a = next(a);
  }

  l = a;
  while (dot(base, p[next(l)] - p[b1]) < dot(base, p[l] - p[b1])) {
    l = next(l);
  }
}

// Pentru o explicație vizuală a funționării algoritmului, vezi
// https://www.youtube.com/watch?v=OaGvH450jRM
long double rotating_calipers() {
  int j, r = 1, a, l;
  long double min_area = 0;

  for (int i = 0; i < n; i++) {
    find_extremes(i, j, r, a, l);
    long double area = rectangle_area(i, j, r, a, l);
    if (!i || (area < min_area)) {
      min_area = area;
    }
  }

  return min_area;
}

long double find_min_area() {
  find_min_y();
  polar_sort();
  graham_scan();
```

```
  return (n <= 2) ? 0 : rotating_calipers();
}

void write_result(long double result) {
  FILE* f = fopen("rubarba.out", "w");
  fprintf(f, "%.2Lf\n", result);
  fclose(f);
}

int main() {
  read_data();
  long double result = find_min_area();
  write_result(result);

  return 0;
}
```

Anexa Q

Grafuri - Arbori parțiali minimi

Q.1 Problema Arbore parțial de cost minim (Infoarena)

înapoi

Sursă cu algoritmul lui Kruskal (versiune online).

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

const int MAX_NODES = 200'000;
const int MAX_EDGES = 400'000;

struct disjoint_set_forest {
  int p[MAX_NODES + 1];

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
    }
  }

  int find(int u) {
    return (p[u] == u)
      ? u
      : (p[u] = find(p[u]));
  }

  void unite(int u, int v) {
    p[find(v)] = find(u);
  }

  bool are_connected(int u, int v) {
    return find(u) == find(v);
  }
```

```
};

struct edge {
  int u, v;
  short cost;
  bool chosen;
};

edge e[MAX_EDGES];
disjoint_set_forest dsf;
int n, m, mst_cost;

void read_data() {
  FILE* f = fopen("apm.in", "r");

  fscanf(f, "%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    fscanf(f, "%d %d %hd", &e[i].u, &e[i].v, &e[i].cost);
  }

  fclose(f);
}

void kruskal() {
  std::sort(e, e + m, [](edge& a, edge& b) {
    return a.cost < b.cost;
  });
  dsf.init(n);

  for (int i = 0; i < m; i++) {
    if (!dsf.are_connected(e[i].u, e[i].v)) {
      e[i].chosen = true;
      mst_cost += e[i].cost;
      dsf.unite(e[i].u, e[i].v);
    }
  }
}

void write_mst() {
  FILE* f = fopen("apm.out", "w");

  fprintf(f, "%d\n", mst_cost);
  fprintf(f, "%d\n", n - 1);
  for (int i = 0; i < m; i++) {
    if (e[i].chosen) {
      fprintf(f, "%d %d\n", e[i].u, e[i].v);
    }
  }

  fclose(f);
```

```
}

int main() {
  read_data();
  kruskal();
  write_mst();

  return 0;
}
```

Sursă cu algoritmul lui Prim (versiune online).

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

const int MAX_NODES = 200'000;
const int MAX_EDGES = 400'000;
const int INFINITY = 1'001; // mai mare decît orice cost

struct edge {
  int v;
  short cost;

  // Vezi https://stackoverflow.com/a/3141107/6022817 pentru "const"
  bool operator<(edge other) const {
    return cost > other.cost;
  }
};

struct node {
  std::vector<edge> adj;
  int parent;
  short dist;
  bool visited;
};

node nd[MAX_NODES + 1];
int n, mst_cost;

void read_data() {
  int m;
  FILE* f = fopen("apm.in", "r");

  fscanf(f, "%d %d", &n, &m);
  while (m--) {
    int u, v;
    short cost;
    fscanf(f, "%d %d %hd", &u, &v, &cost);
    nd[u].adj.push_back({v, cost});
```

```
    nd[v].adj.push_back({u, cost});
  }

  fclose(f);
}

void prim() {
  // Priority queue cu reinserare. Un nod poate trece prin coadă de mai multe
  // ori, iar prin coadă pot trece în total m noduri.
  std::priority_queue<edge> q;
  for (int u = 2; u <= n; u++) {
    nd[u].dist = INFINITY;
  }
  q.push({1, 0});

  while (!q.empty()) {
    int u = q.top().v, cost = q.top().cost;
    q.pop();

    if (!nd[u].visited) {
      nd[u].visited = true;
      mst_cost += cost;
      for (edge e: nd[u].adj) {
        if (!nd[e.v].visited && (e.cost < nd[e.v].dist)) {
          nd[e.v].dist = e.cost;
          nd[e.v].parent = u;
          q.push({e.v, e.cost});
        }
      }
    }
  }
}

void write_mst() {
  FILE* f = fopen("apm.out", "w");

  fprintf(f, "%d\n", mst_cost);
  fprintf(f, "%d\n", n - 1);
  for (int u = 2; u <= n; u++) {
    fprintf(f, "%d %d\n", nd[u].parent, u);
  }

  fclose(f);
}

int main() {
  read_data();
  prim();
  write_mst();
```

```
  return 0;
}
```

Q.2 Problema Autobuze3 (AGM 2015)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 200'000;
const int MAX_EDGES = 400'000;

struct disjoint_set_forest {
  int p[MAX_NODES + 1];

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
    }
  }

  int find(int u) {
    return (p[u] == u)
      ? u
      : (p[u] = find(p[u]));
  }

  void unite(int u, int v) {
    p[find(v)] = find(u);
  }

  bool are_connected(int u, int v) {
    return find(u) == find(v);
  }
};

struct edge {
  int u, v;
  short cost;
};

struct node {
  std::vector<int> adj; // doar muchiile din APM
  std::vector<int> ppl; // ppl[0] este șoferul, care își conduce propriul autobuz
};
```

```
edge e[MAX_EDGES];
node nd[MAX_NODES + 1];
disjoint_set_forest dsf;
int n, m, mst_cost;
FILE *fin, *fout;

void read_data() {
  fscanf(fin, "%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    fscanf(fin, "%d %d %hd", &e[i].u, &e[i].v, &e[i].cost);
  }
}

void kruskal() {
  std::sort(e, e + m, [](edge& a, edge& b) {
    return a.cost < b.cost;
  });
  dsf.init(n);

  mst_cost = 0;
  for (int i = 0; i < m; i++) {
    if (!dsf.are_connected(e[i].u, e[i].v)) {
      mst_cost += e[i].cost;
      dsf.unite(e[i].u, e[i].v);
      nd[e[i].u].adj.push_back(e[i].v);
      nd[e[i].v].adj.push_back(e[i].u);
    }
  }
}

void merge(int u, int v) {
  // small to large
  if (nd[v].ppl.size() > nd[u].ppl.size()) {
    nd[u].ppl.swap(nd[v].ppl);
  }
  for (int p: nd[v].ppl) {
    nd[u].ppl.push_back(p);
    fprintf(fout, "Move %d %d %d\n", p, nd[v].ppl[0], nd[u].ppl[0]);
  }
  nd[v].ppl.clear();
}

void dfs(int u, int parent) {
  nd[u].ppl.push_back(u);

  for (int v: nd[u].adj) {
    if (v != parent) {
      dfs(v, u);
      fprintf(fout, "Drive %d %d %d\n", nd[v].ppl[0], v, u);
      merge(u, v);
```

```
    }
  }
}

void gen_ops() {
  fprintf(fout, "%d\n", mst_cost);
  dfs(1, 0);
  fprintf(fout, "Gata\n");
}

void clear() {
  for (int u = 1; u <= n; u++) {
    nd[u].adj.clear();
  }
  nd[1].ppl.clear();
}

int main() {
  fin = fopen("autobuze3.in", "r");
  fout = fopen("autobuze3.out", "w");

  int num_tests;
  fscanf(fin, "%d", &num_tests);
  while (num_tests--) {
    read_data();
    kruskal();
    gen_ops();
    clear();
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Q.3 Problema Rusuoaica (FMI No Stress 9 Warmup)

‹ înapoi • versiune online

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

const int MAX_NODES = 100'000;
const int MAX_EDGES = 400'000;

struct edge {
  int u, v, c;
```

```
};

struct disjoint_set_forest {
  int parent[MAX_NODES + 1];
  int n, num_trees;

  void init(int n) {
    this->n = n;
    for (int u = 1; u <= n; u++) {
      parent[u] = u;
    }
    num_trees = n;
  }

  int find(int u) {
    return (parent[u] == u)
      ? u
      : (parent[u] = find(parent[u]));
  }

  // Returnează true dacă chiar a unificat u cu v.
  bool merge(int u, int v) {
    u = find(u);
    v = find(v);
    if (u != v) {
      parent[u] = v;
      num_trees--;
      return true;
    } else {
      return false;
    }
  }

  int get_num_trees() {
    return num_trees;
  }
};

disjoint_set_forest dsf;
edge e[MAX_EDGES];
int n, m, build_cost, total_cost;

void read_and_filter_edges() {
  int real_m;
  FILE* f = fopen("rusuoaica.in", "r");

  fscanf(f, "%d %d %d", &n, &real_m, &build_cost);

  while (real_m--) {
    fscanf(f, "%d %d %d", &e[m].u, &e[m].v, &e[m].c);
```

```
    if (e[m].c <= build_cost) {
      m++;
    } else {
      total_cost -= e[m].c;
    }
  }

  fclose(f);
}

void sort_edges() {
  std::sort(e, e + m, [] (edge& a, edge& b) {
    return a.c < b.c;
  });
}

void kruskal() {
  dsf.init(n);
  sort_edges();

  for (int i = 0; i < m; i++) {
    if (dsf.merge(e[i].u, e[i].v)) {
      total_cost += e[i].c;
    } else {
      total_cost -= e[i].c;
    }
  }

  total_cost += build_cost * (dsf.get_num_trees() - 1);
}

void write_answer() {
  FILE* f = fopen("rusuoaica.out", "w");
  fprintf(f, "%d\n", total_cost);
  fclose(f);
}

int main() {
  read_and_filter_edges();
  kruskal();
  write_answer();

  return 0;
}
```

Q.4 Problema MinOr Tree (Codeforces)

❮ înapoi • versiune online

```cpp
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_EDGES = 200'000;
const int MAX_BIT = 29;

struct edge {
  int u, v, weight;
};

struct disjoint_set_forest {
  int parent[MAX_NODES + 1];
  int n, num_trees;

  void reset(int n) {
    this->n = n;
    for (int u = 1; u <= n; u++) {
      parent[u] = u;
    }
    num_trees = n;
  }

  int find(int u) {
    return (parent[u] == u)
      ? u
      : (parent[u] = find(parent[u]));
  }

  void merge(int u, int v) {
    u = find(u);
    v = find(v);
    if (u != v) {
      parent[u] = v;
      num_trees--;
    }
  }

  bool all_connected() {
    return num_trees == 1;
  }

};

edge e[MAX_EDGES];
disjoint_set_forest ds;
int num_nodes, num_edges, ority;

void read_graph() {
  ority = 0;
```

```
  scanf("%d %d", &num_nodes, &num_edges);
  for (int i = 0; i < num_edges; i++) {
    scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].weight);
    ority |= e[i].weight;
  }
}

bool is_connected(int mask) {
  ds.reset(num_nodes);
  int i = 0;
  while ((i < num_edges) && !ds.all_connected()) {
    if ((e[i].weight & mask) == e[i].weight) {
      ds.merge(e[i].u, e[i].v);
    }
    i++;
  }
  return ds.all_connected();
}

void find_ority()  {
  for (int b = MAX_BIT; b >= 0; b--) {
    if ((ority & (1 << b)) &&
        is_connected(ority ^ (1 << b))) {
      ority ^= 1 << b;
    }
  }
}

void write_ority() {
  printf("%d\n", ority);
}

void process_test() {
  read_graph();
  find_ority();
  write_ority();
}

int main() {
  int num_tests;

  scanf("%d", &num_tests);
  while (num_tests--) {
    process_test();
  }

  return 0;
}
```

Q.5 Problema 0-1 MST (Codeforces)

‹ înapoi

Sursă cu algoritmul lui Kruskal (versiune online).

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

const int MAX_NODES = 100'000;

// O structură de mulțimi disjuncte augmentată cu trei informații:
// * numărul de mulțimi;
// * mărimea fiecărei mulțimi;
// * lista de rădăcini.
struct disjoint_set_forest {
  int p[MAX_NODES + 1];
  int s[MAX_NODES + 1];
  int root_pos[MAX_NODES + 1]; // root_pos[u] = poziția lui u în roots[]
  int roots[MAX_NODES + 1];
  int n, num_trees;

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
      s[u] = 1;
      root_pos[u] = u;
      roots[u] = u;
    }
    num_trees = n;
  }

  int find(int u) {
    return (p[u] == u)
      ? u
      : (p[u] = find(p[u]));
  }

  void unite(int u, int v) {
    u = find(u);
    v = find(v);
    if (u != v) {
      s[u] += s[v];
      p[v] = u;
      // Înlocuiește-l pe v cu ultimul element din roots și actualizează
      // root_pos.
      int repl = roots[num_trees - 1];
      roots[root_pos[v]] = repl;
      root_pos[repl] = root_pos[v];
      num_trees--;
```

```
    }
  }

  void unite_incoming(int u, int* incoming) {
    u = find(u);
    int i = 0;
    while (i < num_trees) {
      if ((roots[i] == u) || (incoming[roots[i]] == s[roots[i]])) {
        i++;
      } else {
        unite(u, roots[i]);
      }
    }
  }
};

std::vector<int> unadj[MAX_NODES + 1];
int incoming[MAX_NODES + 1];
disjoint_set_forest dsf;
int n;

void read_data() {
  int m, u, v;
  scanf("%d %d", &n, &m);
  while (m--) {
    scanf("%d %d", &u, &v);
    unadj[u].push_back(v);
    unadj[v].push_back(u);
  }
}

void process_node(int u) {
  // Numără muchiile 1 care intră în fiecare componentă.
  for (int v: unadj[u]) {
    incoming[dsf.find(v)]++;
  }

  dsf.unite_incoming(u, incoming);

  // Curățenie.
  for (int v: unadj[u]) {
    incoming[dsf.find(v)] = 0;
  }
}

void connectivity() {
  dsf.init(n);
  for (int u = 1; u <= n; u++) {
    process_node(u);
  }
```

```
}

int main() {
  read_data();
  connectivity();
  printf("%d\n", dsf.num_trees - 1);

  return 0;
}
```

Sursă cu BFS (versiune online).

```
#include <unordered_set>
#include <queue>
#include <stdio.h>

const int MAX_NODES = 100'000;

std::unordered_set<int> adj[MAX_NODES + 1];
int unseen[MAX_NODES], num_unseen;
int n, answer;

void read_graph() {
  int m, u, v;
  scanf("%d %d", &n, &m);
  while (m--) {
    scanf("%d %d", &u, &v);
    adj[u].insert(v);
    adj[v].insert(u);
  }
}

void init_seen() {
  for (int u = 1; u <= n; u++) {
    unseen[num_unseen++] = u;
  }
}

void bfs(int u) {
  std::queue<int> q;
  q.push(u);
  while (!q.empty()) {
    int u = q.front();
    q.pop();

    int i = 0;
    while (i < num_unseen) {
      int v = unseen[i];
      if (adj[u].find(v) == adj[u].end()) {
```

```
        q.push(v);
        unseen[i] = unseen[--num_unseen];
      } else {
        i++;
      }
    }
  }
}

void bfs_wrapper() {
  int num_comps = 0;
  while (num_unseen) {
    num_comps++;
    int u = unseen[--num_unseen];
    bfs(u);
  }
  answer = num_comps - 1;
}

void write_answer() {
  printf("%d\n", answer);
}

int main() {
  read_graph();
  init_seen();
  bfs_wrapper();
  write_answer();

  return 0;
}
```

Q.6 Problema Minimum Spanning Tree for Each Edge (Codeforces)

‹ înapoi • versiune online

```
// Adaptată după https://codeforces.com/contest/609/submission/129056851
#include <algorithm>
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_EDGES = 200'000;

struct edge {
  int u, v;
  int weight;
```

```
  int orig;
};

struct disjoint_set_forest {
  int p[MAX_NODES + 1];
  int size[MAX_NODES + 1];
  // costul necesar pentru a părăsi componenta prin părinte
  int w[MAX_NODES + 1];

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
      size[u] = 1;
    }
  }

  // Returnează costul maxim pe calea u-v dacă u și v sînt conectate. Altfel
  // unifică u cu v și returnează 0.
  //
  // Trebuie să ne asigurăm că, pentru a părăsi subarborele lui v ca să
  // ajungem la restul arborelui, weight este cea mai scumpă muchie. Ne bazăm
  // pe faptul că procesăm muchiile în ordine crescătoare a costului.
  //
  // Totuși, asta înseamnă că nu putem folosi compresia căii. Dar folosim
  // unirea după mărime ca să ne asigurăm că înălțimea componentelor nu
  // depășește log n.
  int try_unite(int u, int v, int weight) {
    int maxw = 0;

    while ((u != v) && (p[u] != u || p[v] != v)) {
      if ((p[u] == u) || ((p[v] != v) && (size[v] < size[u]))) {
        int tmp = u; u = v; v = tmp;
      }
      maxw = (w[u] > maxw) ? w[u] : maxw;
      u = p[u];
    }

    if (u == v) {
      return maxw;
    } else {
      unite(u, v, weight);
      return 0;
    }
  }

  void unite(int u, int v, int weight) {
    if (size[u] < size[v]) {
      int tmp = u; u = v; v = tmp;
    }
```

```
    size[u] += size[v];
    p[v] = u;
    w[v] = weight;
  }
};

edge e[MAX_EDGES];
disjoint_set_forest dsf;
int diff[MAX_EDGES]; // răspunsurile, relative la APM-ul pe care îl vom afla
int n, m;
long long mst_weight;

void read_data() {
  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].weight);
    e[i].orig = i;
  }
}

void sort_edges_by_weight() {
  std::sort(e, e + m, [] (edge& a, edge& b) {
    return a.weight < b.weight;
  });
}

void kruskal() {
  dsf.init(n);
  for (int i = 0; i < m; i++) {
    edge& z = e[i]; // syntactic sugar
    int maxw = dsf.try_unite(z.u, z.v, z.weight);
    if (maxw) {
      // Ca să o introducem pe z în APM, vom scoate o muchie de cost maxw.
      diff[z.orig] = z.weight - maxw;
    } else {
      mst_weight += z.weight;
      diff[z.orig] = 0;
    }
  }
}

void write_answers() {
  for (int i = 0; i < m; i++) {
    printf("%lld\n", mst_weight + diff[i]);
  }
}

int main() {
  read_data();
  sort_edges_by_weight();
```

```
  kruskal();
  write_answers();

  return 0;
}
```

Q.7 Problema Apm2 (Infoarena)

‹ înapoi • versiune online

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

const int MAX_NODES = 10'000;
const int MAX_EDGES = 100'000;

struct edge {
  int u, v, c;
};

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct disjoint_set_forest {
  int parent[MAX_NODES + 1];
  int cost[MAX_NODES + 1]; // costul părăsirii subarborelui
  int size[MAX_NODES + 1];
  int n;

  void init(int n) {
    this->n = n;
    for (int u = 1; u <= n; u++) {
      parent[u] = u;
      size[u] = 1;
    }
  }

  int find(int u) {
    while (parent[u] != u) {
      u = parent[u];
    }
    return u;
  }

  void merge(int u, int v, int c) {
    u = find(u);
    v = find(v);
```

```
    if (u != v) {
      if (size[u] > size[v]) {
        int tmp = u; u = v; v = tmp;
      }
      parent[u] = v;
      cost[u] = c;
      size[v] += size[u];
    }
  }

  int get_max_cost(int u, int v) {
    int result = 0;

    while (u != v) {
      if (size[u] < size[v]) {
        result = max(result, cost[u]);
        u = parent[u];
      } else {
        result = max(result, cost[v]);
        v = parent[v];
      }
    }

    return result;
  }
};

edge e[MAX_EDGES];
disjoint_set_forest dsf;
int num_nodes, num_edges, num_queries;
FILE* fin;

void read_graph() {
  fin = fopen("apm2.in", "r");
  fscanf(fin, "%d %d %d", &num_nodes, &num_edges, &num_queries);
  for (int i = 0; i < num_edges; i++) {
    fscanf(fin, "%d %d %d", &e[i].u, &e[i].v, &e[i].c);
  }
}

void sort_edges_by_cost() {
  std::sort(e, e + num_edges, [] (edge& a, edge& b) {
    return a.c < b.c;
  });
}

void kruskal() {
  sort_edges_by_cost();
  dsf.init(num_nodes);
  for (int i = 0; i < num_edges; i++) {
```

```
    dsf.merge(e[i].u, e[i].v, e[i].c);
  }
}

void answer_queries() {
  FILE* fout = fopen("apm2.out", "w");

  while (num_queries--) {
    int u, v;
    fscanf(fin, "%d %d", &u, &v);
    int max_cost = dsf.get_max_cost(u, v);
    fprintf(fout, "%d\n", max_cost - 1);
  }

  fclose(fin);
  fclose(fout);
}

int main() {
  read_graph();
  kruskal();
  answer_queries();

  return 0;
}
```

Q.8 Problema Edges in MST (Codeforces)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;
const int MAX_EDGES = 100'000;

const int T_SOME = 0; // răspunsul implicit, dacă nu este suprascris
const int T_NONE = 1;
const int T_ALL = 2;

struct edge {
  int u, v;
  int cost;
  int orig;
};

struct disjoint_set_forest {
```

```
  int p[MAX_NODES + 1];

  void init(int n) {
    for (int u = 1; u <= n; u++) {
      p[u] = u;
    }
  }

  int find(int u) {
    return (p[u] == u)
      ? u
      : (p[u] = find(p[u]));
  }

  void unite(int u, int v) {
    p[find(v)] = find(u);
  }

  bool are_united(int u, int v) {
    return find(u) == find(v);
  }
};

struct reduced_edge {
  int v, orig; // O muchie u-v care are indexul „orig” în datele de intrare.
};

struct node {
  std::vector<reduced_edge> adj;
  int time, low;
};

edge e[MAX_EDGES];
disjoint_set_forest dsf;
node nd[MAX_NODES + 1];
int roots[2 * MAX_NODES], num_roots; // Rădăcini pentru grupa Kruskal curentă.
int answer[MAX_EDGES];
int n, m;

void read_data() {
  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].cost);
    e[i].orig = i;
  }
}

void sort_edges() {
  std::sort(e, e + m, [](edge& a, edge& b) {
    return a.cost < b.cost;
```

```
  });
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

int time;

void bridge_dfs(int u, int orig) {
  nd[u].time = nd[u].low = ++time;

  for (reduced_edge r: nd[u].adj) {
    if (r.orig != orig) {
      int v = r.v;
      if (!nd[v].time) {
        bridge_dfs(v, r.orig);
        nd[u].low = min(nd[u].low, nd[v].low);
        if (nd[v].low > nd[u].time) {
          answer[r.orig] = T_ALL;
        }

      } else {
        if (nd[v].time < nd[u].time) {
          nd[u].low = min(nd[u].low, nd[v].time);
        }
      }
    }
  }
}

void bridge_dfs_driver() {
  time = 0;
  for (int i = 0; i < num_roots; i++) {
    if (!nd[roots[i]].time) {
      bridge_dfs(roots[i], -1);
    }
  }
}

void dfs_cleanup() {
  for (int i = 0; i < num_roots; i++) {
    int u = roots[i];
    nd[u].adj.clear();
    nd[u].time = nd[u].low = 0;
  }
  num_roots = 0;
}

// Este important ca acest bloc să ruleze în O(batch_size). Dacă buclele scapă
```

```
// în O(n), vor duce la O(n²) global. De aceea, colectează capetele muchiilor
// și rulează DFS doar din acele noduri.
void process_batch(int start, int end) {
  num_roots = 0;

  for (int i = start; i < end; i++) {
    int u = dsf.find(e[i].u), v = dsf.find(e[i].v);
    if (u == v) {
      answer[e[i].orig] = T_NONE;
    } else {
      nd[u].adj.push_back({v, e[i].orig});
      nd[v].adj.push_back({u, e[i].orig});
      roots[num_roots++] = u;
      roots[num_roots++] = v;
    }
  }

  bridge_dfs_driver();
  dfs_cleanup();

  // Acum aplică modificările.
  for (int i = start; i < end; i++) {
    dsf.unite(e[i].u, e[i].v);
  }
}

void kruskal() {
  sort_edges();
  dsf.init(n);

  int i = 0;
  while (i < m) {
    int j = i + 1;
    while ((j < m) && (e[j].cost == e[i].cost)) {
      j++;
    }
    process_batch(i, j);
    i = j;
  }
}

void write_answers() {
  for (int i = 0; i < m; i++) {
    switch (answer[i]) {
      case T_SOME: printf("at least one\n"); break;
      case T_NONE: printf("none\n"); break;
      case T_ALL:  printf("any\n"); break;
    }
  }
}
```

```
int main() {
  read_data();
  kruskal();
  write_answers();

  return 0;
}
```

Q.9 Problema Envy (Codeforces)

‹ înapoi • versiune online

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

const int MAX_NODES = 500'000;
const int MAX_EDGES = 1'000'000;
const int MAX_QUERIES = 500'000;
const int NONE = MAX_QUERIES + 1;

struct edge {
  int u, v, w;
  int q;
};

// Pădure de mulțimi disjuncte augmentată. Când schimbă un părinte, îl ține
// minte pe cel vechi. Părinții salvați pot fi uitați sau restaurați în timp
// O(numărul_de_modificări).
struct disjoint_set_forest {
  int p[MAX_NODES + 1];
  int orig_p[MAX_NODES + 1];
  int save_pos[MAX_NODES + 1];
  int n, num_saved;

  void init(int n) {
    this->n = n;
    num_saved = 0;
    for (int u = 1; u <= n; u++) {
      p[u] = u;
      orig_p[u] = 0;
    }
  }

  void save(int u) {
    if (!orig_p[u]) {
      orig_p[u] = p[u];
      save_pos[num_saved++] = u;
```

```
    }
  }

  int find(int u) {
    if (p[u] == u) {
      return u;
    } else {
      save(u);
      p[u] = find(p[u]);
      return p[u];
    }
  }

  // Returnează true dacă și numai dacă u și v erau în arbori diferiți.
  bool unite(int u, int v) {
    u = find(u);
    v = find(v);
    if (u == v) {
      return false;
    } else {
      save(u);
      p[u] = v;
      return true;
    }
  }

  void wipe_saves() {
    for (int i = 0; i < num_saved; i++) {
      int u = save_pos[i];
      orig_p[u] = 0;
    };
    num_saved = 0;
  }

  void restore_saves() {
    for (int i = 0; i < num_saved; i++) {
      int u = save_pos[i];
      p[u] = orig_p[u];
      orig_p[u] = 0;
    };
    num_saved = 0;
  }
};

edge e[MAX_EDGES];
disjoint_set_forest dsf;
bool answer[MAX_QUERIES];
int n, m, num_queries;

void read_data() {
```

```
  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].w);
    e[i].q = NONE;
  }

  scanf("%d", &num_queries);
  for (int q = 0; q < num_queries; q++) {
    int qsize;
    scanf("%d", &qsize);
    while (qsize--) {
      int ind;
      scanf("%d", &ind);
      ind--;
      e[m++] = { e[ind].u, e[ind].v, e[ind].w, q };
    }
    answer[q] = true;
  }
}

void sort_edges() {
  std::sort(e, e + m, [] (edge& a, edge& b) {
    return (a.w < b.w) ||
      ((a.w == b.w) && (a.q < b.q));
  });
}

int find_batch(int start) {
  int end = start;
  while ((end < m) && (e[start].w == e[end].w) && (e[start].q == e[end].q)) {
    end++;
  }
  return end;
}

void process_equal_weights(int start, int end) {
  int k = start;
  while ((k < end) && answer[e[k].q]) {
    answer[e[k].q] = dsf.unite(e[k].u, e[k].v);
    k++;
  }
  dsf.restore_saves();
}

void process_edges() {
  dsf.init(n);
  int start = 0;
  while (start < m) {
    if (e[start].q == NONE) {
      dsf.unite(e[start].u, e[start].v);
```

```
      dsf.wipe_saves();
      start++;
    } else {
      int end = find_batch(start);
      process_equal_weights(start, end);
      start = end;
    }
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%s\n", answer[i] ? "YES" : "NO");
  }
}

int main() {
  read_data();
  sort_edges();
  process_edges();
  write_answers();

  return 0;
}
```

Q.10 Problema DFS Trees (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_EDGES = 200'000;
const int WHITE = 0;
const int BLACK = -1;

struct cell {
  int val, next;
};

struct node {
  int ptr, nptr; // liste de adiacență pentru muchii de arbore/nonarbore
  int state;     // 0 = alb; -1 = negru; d > 0 = gri la adîncime d
  int bad_count;
};

struct disjoint_set_forest {
  int parent[MAX_NODES + 1];
```

```
  int n;

  void init(int n) {
    this->n = n;
    for (int u = 1; u <= n; u++) {
      parent[u] = u;
    }
  }

  int find(int u) {
    return (parent[u] == u)
      ? u
      : (parent[u] = find(parent[u]));
  }

  // Unifică v în u. Fără *union by rank*.
  // Returnează true dacă și numai dacă unificarea chiar a reușit.
  bool merge(int u, int v) {
    u = find(u);
    v = find(v);
    if (u != v) {
      parent[v] = u;
      return true;
    } else {
      return false;
    }
  }
};

cell list[2 * MAX_EDGES + 1];
node nd[MAX_NODES + 1];
int stack[MAX_NODES];
disjoint_set_forest dsf;
int n, list_pos = 1;

void add_tree_edge(int u, int v) {
  list[list_pos] = { v, nd[u].ptr };
  nd[u].ptr = list_pos++;
}

void add_non_tree_edge(int u, int v) {
  list[list_pos] = { v, nd[u].nptr };
  nd[u].nptr = list_pos++;
}

void read_graph_and_kruskal() {
  int num_edges, u, v;

  scanf("%d %d", &n, &num_edges);
  dsf.init(n);
```

```
  while (num_edges--) {
    scanf("%d %d", &u, &v);
    if (dsf.merge(u, v)) {
      add_tree_edge(u, v);
      add_tree_edge(v, u);
    } else {
      add_non_tree_edge(u, v);
      add_non_tree_edge(v, u);
    }
  }
}

// Nodul u este nodul DFS activ. Nodul v este fie gri, fie negru.
void mark_subtrees(int u, int v) {
  if (nd[v].state == BLACK) {
    nd[1].bad_count++;
    nd[v].bad_count--;
    nd[u].bad_count--;
  } else {
    int w = stack[nd[v].state + 1];
    nd[w].bad_count++;
    nd[u].bad_count--;
  }
}

void dfs_mark_counts(int u, int depth) {
  nd[u].state = depth;
  stack[depth] = u;

  // Marchează muchiile de nonarbore.
  for (int ptr = nd[u].nptr; ptr; ptr = list[ptr].next) {
    int v = list[ptr].val;
    if (nd[v].state != WHITE) {
      mark_subtrees(u, v);
    }
  }

  // Traversează muchiile de arbore.
  for (int ptr = nd[u].ptr; ptr; ptr = list[ptr].next) {
    int v = list[ptr].val;
    if (nd[v].state == WHITE) {
      dfs_mark_counts(v, depth + 1);
    }
  }

  nd[u].state = BLACK;

}

void dfs_propagate_counts(int u, int parent) {
```

```
  for (int ptr = nd[u].ptr; ptr; ptr = list[ptr].next) {
    int v = list[ptr].val;
    if (v != parent) {
      nd[v].bad_count += nd[u].bad_count;
      dfs_propagate_counts(v, u);
    }
  }
}

void write_answers() {
  for (int u = 1; u <= n; u++) {
    char answer = (nd[u].bad_count == 0) ? '1' : '0';
    putchar(answer);
  }
  putchar('\n');
}

int main() {
  read_graph_and_kruskal();

  dfs_mark_counts(1, 1);
  dfs_propagate_counts(1, 0);

  write_answers();

  return 0;
}
```

Anexa R

Grafuri - Conectivitate

R.1 Problema Course Schedule (CSES)

înapoi

Sursă cu BFS (algoritmul lui Kahn).

```
#include <queue>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;

struct node {
  std::vector<int> adj;
  int in_degree;
};

std::vector<int> order;
node nd[MAX_NODES + 1];
int n;

void read_graph() {
  int num_edges, u, v;

  scanf("%d %d", &n, &num_edges);
  while (num_edges--) {
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].in_degree++;
  }
}

void bfs() {
  std::queue<int> q;
```

```
  for (int u = 1; u <= n; u++) {
    if (!nd[u].in_degree) {
      q.push(u);
    }
  }

  while (!q.empty()) {
    int u = q.front();
    q.pop();
    order.push_back(u);
    for (auto v: nd[u].adj) {
      nd[v].in_degree--;
      if (!nd[v].in_degree) {
        q.push(v);
      }
    }
  }
}

void write_order() {
  if ((int)order.size() == n) {
    for (int u: order) {
      printf("%d ", u);
    }
    printf("\n");
  } else {
    printf("IMPOSSIBLE\n");
  }
}

int main() {
  read_graph();
  bfs();
  write_order();

  return 0;
}
```

Sursă cu DFS.

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

const int MAX_NODES = 100'000;
const int WHITE = 0;
const int GRAY = 1;
const int BLACK = 2;

struct node {
```

```
  std::vector<int> adj;
  char color;
};

node nd[MAX_NODES + 1];
std::vector<int> order;
bool has_cycle;
int n;

void read_graph() {
  int num_edges, u, v;

  scanf("%d %d", &n, &num_edges);
  while (num_edges--) {
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
  }
}

void dfs(int u) {
  has_cycle |= (nd[u].color == GRAY);

  if (nd[u].color == WHITE) {
    nd[u].color = GRAY;
    for (auto v: nd[u].adj) {
      dfs(v);
    }
    nd[u].color = BLACK;
    order.push_back(u);
  }
}

void dfs_driver() {
  for (int u = 1; u <= n; u++) {
    dfs(u);
  }
}

void write_order() {
  if (has_cycle) {
    printf("IMPOSSIBLE\n");
  } else {
    for (int i = n - 1; i >= 0; i--) {
      printf("%d ", order[i]);
    }
    printf("\n");
  }
}

int main() {
```

```
  read_graph();
  dfs_driver();
  write_order();

  return 0;
}
```

R.2 Problema Book (Codeforces)

înapoi

Sursă cu BFS (algoritmul lui Kahn).

```
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_EDGES = 200'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int in_degree;
  int pass;
};

struct queue {
  int a[MAX_NODES];
  int head, tail;

  void init() {
    head = tail = 0;
  }

  void push(int u) {
    a[tail++] = u;
  }

  int pop() {
    return a[head++];
  }

  bool is_empty() {
    return head == tail;
  }
};
```

```
cell list[MAX_EDGES + 1];
node nd[MAX_NODES + 1];
queue q;
int n, num_edges, num_learned;

void clear_graph() {
  num_edges = 0;
  num_learned = 0;
  for (int u = 1; u <= n; u++) {
    nd[u].adj = 0;
    nd[u].pass = 0;
  }
}

void add_edge(int u, int v) {
  list[++num_edges] = { v, nd[u].adj };
  nd[u].adj = num_edges;
}

void read_graph() {
  scanf("%d", &n);

  clear_graph();
  for (int u = 1; u <= n; u++) {
    scanf("%d", &nd[u].in_degree);
    for (int i = 0; i < nd[u].in_degree; i++) {
      int v;
      scanf("%d", &v);
      add_edge(v, u);
    }
  }
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

void update_passes(int u, int v) {
  int pass_v = (u < v) ? nd[u].pass : (nd[u].pass + 1);
  nd[v].pass = max(nd[v].pass, pass_v);
}

void bfs() {
  q.init();

  for (int u = 1; u <= n; u++) {
    if (!nd[u].in_degree) {
      nd[u].pass = 1;
      q.push(u);
```

```
    }
  }

  while (!q.is_empty()) {
    num_learned++;
    int u = q.pop();
    for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
      int v = list[ptr].v;
      update_passes(u, v);
      nd[v].in_degree--;
      if (!nd[v].in_degree) {
        q.push(v);
      }
    }
  }
}

int get_max_passes() {
  if (num_learned < n) {
    return -1;
  }

  int m = 0;
  for (int u = 1; u <= n; u++) {
    m = max(m, nd[u].pass);
  }
  return m;
}

void solve_test() {
  read_graph();
  bfs();
  int answer = get_max_passes();
  printf("%d\n", answer);
}

int main() {

  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    solve_test();
  }

  return 0;
}
```

Sursă cu DFS.

```c
#include <stdio.h>

const int MAX_NODES = 200'000;
const int MAX_EDGES = 200'000;

const int WHITE = 0;
const int GRAY = 1;
const int BLACK = 2;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int pass;
  char color;
};

cell list[MAX_EDGES + 1];
node nd[MAX_NODES + 1];
int n, m, max_pass;
bool has_cycle;

void clear_graph() {
  m = 0;
  max_pass = 1;
  has_cycle = false;
  for (int u = 1; u <= n; u++) {
    nd[u].adj = 0;
    nd[u].pass = 1;
    nd[u].color = WHITE;
  }
}

void add_edge(int u, int v) {
  list[++m] = { v, nd[u].adj };
  nd[u].adj = m;
}

void read_graph() {
  scanf("%d", &n);

  clear_graph();
  for (int u = 1; u <= n; u++) {
    int in_degree, v;
    scanf("%d", &in_degree);
    while (in_degree--) {
      scanf("%d", &v);
```

```
      add_edge(v, u);
    }
  }
}

void maximize(int& x, int y) {
  if (y > x) {
    x = y;
  }
}

void top_sort(int u) {
  has_cycle |= (nd[u].color == GRAY);
  if (nd[u].color != WHITE) {
    return;
  }

  nd[u].color = GRAY;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    top_sort(v);
    maximize(nd[u].pass, nd[v].pass + (u > v));
    maximize(max_pass, nd[u].pass);
  }

  nd[u].color = BLACK;
}

void solve_test() {
  read_graph();

  for (int u = 1; u <= n; u++) {
    top_sort(u);
  }
  printf("%d\n", has_cycle ? -1 : max_pass);
}

int main() {

  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    solve_test();
  }

  return 0;
}
```

R.3 Problema Componente tare conexe (Infoarena)

‹ înapoi

Sursă cu algoritmul lui Kosaraju și STL (versiune online).

```cpp
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;

struct node {
  std::vector<int> adj, adjt;
  bool mark;
};

node n[MAX_NODES + 1];
std::vector<int> order;
std::vector<std::vector<int>> comps;
int num_nodes;

void read_graph() {
  FILE* f = fopen("ctc.in", "r");
  int num_edges, u, v;
  fscanf(f, "%d %d", &num_nodes, &num_edges);
  for (int i = 1; i <= num_edges; i++) {
    fscanf(f, "%d %d", &u, &v);
    n[u].adj.push_back(v);
    n[v].adjt.push_back(u);
  }
  fclose(f);
}

void dfs(int u) {
  n[u].mark = true;
  for (auto v: n[u].adj) {
    if (!n[v].mark) {
      dfs(v);
    }
  }

  order.push_back(u);
}

void dfs_driver() {
  for (int u = 1; u <= num_nodes; u++) {
    if (!n[u].mark) {
      dfs(u);
    }
  }
```

```
}

void tdfs(int u) {
  n[u].mark = false;
  for (auto v: n[u].adjt) {
    if (n[v].mark) {
      tdfs(v);
    }
  }

  comps.back().push_back(u);
}

void transpose_dfs_driver() {
  for (int i = num_nodes - 1; i >= 0; i--) {
    int u = order[i];
    if (n[u].mark) {
      comps.push_back({});
      tdfs(u);
    }
  }
}

void write_components() {
  FILE* f = fopen("ctc.out", "w");
  fprintf(f, "%lu\n", comps.size());
  for (auto vec: comps)  {
    for (auto u: vec) {
      fprintf(f, "%d ", u);
    }
    fprintf(f, "\n");
  }
  fclose(f);
}

int main() {
  read_graph();
  dfs_driver();
  transpose_dfs_driver();
  write_components();

  return 0;
}
```

Sursă cu algoritmul lui Kosaraju și structuri proprii (versiune online).

```
#include <stdio.h>

const int MAX_NODES = 100'000;
```

```
const int MAX_EDGES = 200'000;

struct edge {
  int v, next;
};

struct node {
  int adj;
  bool mark;
};

node n[MAX_NODES + 1];
edge e[MAX_EDGES + 1];
int order[MAX_NODES], order_size;
int comp[MAX_NODES + 1], comp_size;
int comp_start[MAX_NODES + 1], num_comps;
int num_nodes;

void read_graph() {
  FILE* f = fopen("ctc.in", "r");
  int num_edges, u, v;
  fscanf(f, "%d %d", &num_nodes, &num_edges);
  for (int i = 1; i <= num_edges; i++) {
    fscanf(f, "%d %d", &u, &v);
    e[i] = { v, n[u].adj };
    n[u].adj = i;
  }
  fclose(f);
}

void dfs(int u) {
  n[u].mark = true;
  for (int ptr = n[u].adj; ptr; ptr = e[ptr].next) {
    int v = e[ptr].v;
    if (!n[v].mark) {
      dfs(v);
    }
  }

  order[order_size++] = u;
}

void dfs_driver() {
  for (int u = 1; u <= num_nodes; u++) {
    if (!n[u].mark) {
      dfs(u);
    }
  }
}
```

```
// Transpune graful, reutilizînd vectorul de muchii.
void transpose() {
  int *old_adj = comp; // syntactic sugar
  for (int u = 1; u <= num_nodes; u++) {
    old_adj[u] = n[u].adj;
    n[u].adj = 0;
  }
  for (int u = 1; u <= num_nodes; u++) {
    int ptr = old_adj[u];
    while (ptr) {
      // mută e[ptr] din lista de adiacență a lui u în a lui v.
      int v = e[ptr].v;
      int next = e[ptr].next;
      e[ptr] = { u, n[v].adj };
      n[v].adj = ptr;
      ptr = next;
    }
  }
}

void tdfs(int u) {
  n[u].mark = false;
  for (int ptr = n[u].adj; ptr; ptr = e[ptr].next) {
    int v = e[ptr].v;
    if (n[v].mark) {
      tdfs(v);
    }
  }

  comp[comp_size++] = u;
}

void transpose_dfs_driver() {
  for (int i = num_nodes - 1; i >= 0; i--) {
    int u = order[i];
    if (n[u].mark) {
      comp_start[num_comps++] = comp_size;
      tdfs(u);
    }
  }
  comp_start[num_comps] = num_nodes;
}

void write_components() {
  FILE* f = fopen("ctc.out", "w");
  fprintf(f, "%d\n", num_comps);
  for (int i = 0; i < num_comps; i++) {
    for (int j = comp_start[i]; j < comp_start[i + 1]; j++) {
      fprintf(f, "%d ", comp[j]);
    }
```

```
    fprintf(f, "\n");
  }
  fclose(f);
}

int main() {
  read_graph();
  dfs_driver();
  transpose();
  transpose_dfs_driver();
  write_components();

  return 0;
}
```

Sursă cu algoritmul lui Tarjan și STL (versiune online).

```
#include <algorithm>
#include <stack>
#include <stdio.h>
#include <vector>

const int MAX_NODES = 100'000;

struct node {
  std::vector<int> adj, adjt;
  int time_in;
  int low;
  bool on_stack;
};

node n[MAX_NODES + 1];
std::stack<int> st;  // stiva DFS
std::vector<std::vector<int>> comps;
int num_nodes, time;

void read_graph() {
  FILE* f = fopen("ctc.in", "r");
  int num_edges, u, v;
  fscanf(f, "%d %d", &num_nodes, &num_edges);
  for (int i = 1; i <= num_edges; i++) {
    fscanf(f, "%d %d", &u, &v);
    n[u].adj.push_back(v);
  }
  fclose(f);
}

void pop_scc(int last) {
  std::vector<int> c;
```

```
  int v;
  do {
    v = st.top();
    st.pop();
    c.push_back(v);
    n[v].on_stack = false;
  } while (v != last);
  comps.push_back(c);
}

void dfs(int u) {
  n[u].time_in = n[u].low = ++time;
  n[u].on_stack = true;
  st.push(u);

  for (auto v: n[u].adj) {
    if (!n[v].time_in) {
      // Traversează fiul alb și notează cît poate să urce.
      dfs(v);
      n[u].low = std::min(n[u].low, n[v].low);
    } else if (n[v].on_stack) {
      // Putem urca pînă la nivelul lui v
      n[u].low = std::min(n[u].low, n[v].time_in);
    }
  }

  // Dacă niciun nod din subarborele lui u nu poate urca mai sus decît u,
  // atunci stiva de la u pînă la vîrf este o CTC.
  if (n[u].low == n[u].time_in) {
    pop_scc(u);
  }
}

void dfs_driver() {
  for (int u = 1; u <= num_nodes; u++) {
    if (!n[u].time_in) {
      dfs(u);
    }
  }
}

void write_components() {
  FILE* f = fopen("ctc.out", "w");
  fprintf(f, "%lu\n", comps.size());
  for (auto vec: comps)  {
    for (auto u: vec) {
      fprintf(f, "%d ", u);
    }
    fprintf(f, "\n");
  }
```

```
  fclose(f);
}

int main() {
  read_graph();
  dfs_driver();
  write_components();

  return 0;
}
```

R.4 Problema Mr. Kitayuta's Technology (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_EDGES = 100'000;
const int GRAY = 1;
const int BLACK = 2;

struct edge {
  int v, next;
};

struct node {
  int adj;
  char color;
};

// O pădure de mulțimi disjuncte augmentată cu mărimea și nodurile fiecărei
// rădăcini.
struct disjoint_set_forest {
  int parent[MAX_NODES + 1];
  int size[MAX_NODES + 1];
  int next[MAX_NODES + 1];
  int last[MAX_NODES + 1];
  int n;

  void init(int n) {
    this->n = n;
    for (int u = 1; u <= n; u++) {
      parent[u] = u;
      size[u] = 1;
      next[u] = 0;
      last[u] = u;
    }
```

```
  }

  int find(int u) {
    return (parent[u] == u)
      ? u
      : (parent[u] = find(parent[u]));
  }

  void merge(int u, int v) {
    u = find(u);
    v = find(v);
    if (u != v) {
      parent[v] = u;
      size[u] += size[v];
      next[last[u]] = v;
      last[u] = last[v];
    }
  }
};

node nd[MAX_NODES + 1];
edge e[MAX_EDGES + 1];
disjoint_set_forest dsf;
int n;

void add_edge(int u, int v) {
  static int pos = 1;
  e[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_graph_and_wcc() {
  int num_edges, u, v;
  scanf("%d %d", &n, &num_edges);
  dsf.init(n);

  while (num_edges--) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    dsf.merge(u, v);
  }
}

bool find_cycle(int u) {
  if (nd[u].color == GRAY) {
    return true;
  } else if (nd[u].color == BLACK) {
    return false;
  }
```

```
  nd[u].color = GRAY;
  int ptr = nd[u].adj;
  while (ptr && !find_cycle(e[ptr].v)) {
    ptr = e[ptr].next;
  }
  nd[u].color = BLACK;

  return ptr;
}

int satisfy_wcc(int u) {
  int size = dsf.size[u];
  while (u && !find_cycle(u)) {
    u = dsf.next[u];
  }
  return u ? size : (size - 1);
}

void satisfy_all_wccs() {
  int answer = 0;
  for (int u = 1; u <= n; u++) {
    if (dsf.parent[u] == u) {
      answer += satisfy_wcc(u);
    }
  }
  printf("%d\n", answer);
}

int main() {
  read_graph_and_wcc();
  satisfy_all_wccs();

  return 0;
}
```

R.5 Problema Ralph and Mushrooms (Codeforces)

❮ înapoi • versiune online

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

const int MAX_NODES = 1'000'000;
const int MAX_EDGES = 1'000'000;

typedef unsigned long long u64;

struct edge {
```

```
  int v, mushrooms, next;
};

struct node {
  u64 inc_profit, exc_profit; // profitul cu/fără CTC-ul lui u
  int adj;
  int time_in;
  int low;
  bool on_stack;
};

node nd[MAX_NODES + 1];
edge e[MAX_EDGES + 1];
int st[MAX_NODES], ss;  // stiva DFS
int n, start_node;

void read_graph() {
  int num_edges, u;
  scanf("%d %d", &n, &num_edges);
  for (int i = 1; i <= num_edges; i++) {
    scanf("%d %d %d", &u, &e[i].v, &e[i].mushrooms);
    e[i].next = nd[u].adj;
    nd[u].adj = i;
  }
  scanf("%d", &start_node);
}

u64 max(u64 x, u64 y) {
  return (x > y) ? x : y;
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

u64 get_mushroom_profit(int m) {
  // Implementează https://oeis.org/A060432. Remarcăm că diferențele sînt 1,
  // 2, 2, 3, 3, 3, 4, 4, 4, 4 etc. De aceea, rezolvă t * (t + 1) / 2 ≤ m.
  // Ultimul termen cu un t întreg este o sumă de pătrate.
  int t = (sqrtl(8 * m + 1) - 1) * 0.5;
  u64 base = t * (t + 1) / 2;
  u64 sum = base * (2 * t + 1) / 3;
  sum += (u64)(m - base) * (t + 1);
  return sum;
}

void update_profits(int u, int v, int mushrooms) {
  if (nd[v].on_stack) {
    // muchie internă
    u64 p = get_mushroom_profit(mushrooms);
```

```
    nd[u].exc_profit = max(nd[u].exc_profit, nd[v].exc_profit);
    if (u == v) {
      nd[u].inc_profit += p;
    } else {
      nd[u].inc_profit += p + nd[v].inc_profit;
      nd[v].inc_profit = 0;
    }
  } else {
    // muchie externă
    nd[u].exc_profit = max(nd[u].exc_profit, nd[v].inc_profit + mushrooms);
  }
}

void pop_scc(int root) {
  u64 scc_profit = nd[root].inc_profit + nd[root].exc_profit;

  int v;
  do {
    v = st[--ss];
    nd[v].on_stack = false;
    nd[v].inc_profit = scc_profit;
  } while (v != root);
}

void tarjan(int u) {
  static int time = 0;

  nd[u].time_in = nd[u].low = ++time;
  nd[u].on_stack = true;
  st[ss++] = u;

  for (int ptr = nd[u].adj; ptr; ptr = e[ptr].next) {
    int v = e[ptr].v;

    if (!nd[v].time_in) {
      tarjan(v);
      nd[u].low = min(nd[u].low, nd[v].low);
    } else if (nd[v].on_stack) {
      nd[u].low = min(nd[u].low, nd[v].time_in);
    }

    update_profits(u, v, e[ptr].mushrooms);
  }

  if (nd[u].low == nd[u].time_in) {
    pop_scc(u);
  }
}

void write_answer() {
```

```
  printf("%llu\n", nd[start_node].inc_profit);
}

int main() {
  read_graph();
  tarjan(start_node);
  write_answer();

  return 0;
}
```

R.6 Problema Bertown Roads (Codeforces)

înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 100'000;
const int MAX_EDGES = 300'000;

struct edge {
  int u, v;
};

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth, low;
};

node nd[MAX_NODES + 1];
cell list[2 * MAX_EDGES + 1];
edge e[MAX_EDGES];
int n, sol_edges;
bool has_bridges;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_graph() {
  int num_edges, u, v;
  scanf("%d %d", &n, &num_edges);
```

```
  while (num_edges--) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

void dfs(int u, int parent) {
  nd[u].depth = nd[u].low = nd[parent].depth + 1;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;

    if (!nd[v].depth) {
      // muchie de arbore
      e[sol_edges++] = { u, v };
      dfs(v, u);
      nd[u].low = min(nd[u].low, nd[v].low);
      has_bridges |= (nd[v].low > nd[u].depth);

    } else if ((v != parent) && (nd[v].depth < nd[u].depth)) {
      // muchie înapoi
      e[sol_edges++] = { u, v };
      nd[u].low = min(nd[u].low, nd[v].depth);

    }
  }
}

void write_answer() {
  if (has_bridges) {
    printf("0\n");
  } else {
    for (int i = 0; i < sol_edges; i++) {
      printf("%d %d\n", e[i].u, e[i].v);
    }
  }
}

int main() {
  read_graph();
  dfs(1, 0);
  write_answer();

  return 0;
```

```
}
```

R.7 Problema Tourist Reform (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 400'000;
const int MAX_EDGES = 400'000;

struct edge {
  int u, v;
};

struct cell {
  int edge_index, next;
};

struct node {
  int adj;
  int depth, low;
};

node nd[MAX_NODES + 1];
cell list[2 * MAX_EDGES + 1];
edge e[MAX_EDGES + 1];
int n, m, stack_size, max_comp_size, max_comp_node;

void add_edge(int u, int edge_index) {
  static int pos = 1;
  list[pos] = { edge_index, nd[u].adj };
  nd[u].adj = pos++;
}

void read_graph() {
  scanf("%d %d", &n, &m);

  for (int i = 1; i <= m; i++) {
    scanf("%d %d", &e[i].u, &e[i].v);
    add_edge(e[i].u, i);
    add_edge(e[i].v, i);
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}
```

```
void pop_bcc(int root, int root_stack_pos) {
  if (stack_size - root_stack_pos > max_comp_size) {
    max_comp_size = stack_size - root_stack_pos;
    max_comp_node = root;
  }
  stack_size = root_stack_pos;
}

void dfs(int u, int parent) {
  nd[u].depth = nd[u].low = nd[parent].depth + 1;
  int stack_pos = stack_size++;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    edge& f = e[list[ptr].edge_index];
    int v = (f.u == u) ? f.v : f.u;

    if (!nd[v].depth) {
      dfs(v, u);
      nd[u].low = min(nd[u].low, nd[v].low);
    } else if ((v != parent) && (nd[v].depth < nd[u].depth)) {
      nd[u].low = min(nd[u].low, nd[v].depth);
    }
  }

  if (nd[u].low > nd[parent].depth) {
    pop_bcc(u, stack_pos);
  }
}

void wipe_depths() {
  for (int u = 1; u <= n; u++) {
    nd[u].depth = 0;
  }
}

void orient_edges(int u, int parent) {
  nd[u].depth =  nd[parent].depth + 1;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    edge& f = e[list[ptr].edge_index];
    int v = (f.u == u) ? f.v : f.u;

    if (!nd[v].depth) {
      f = { v, u };
      orient_edges(v, u);
    } else if ((v != parent) && (nd[v].depth < nd[u].depth)) {
      f = { v, u };
    }
  }
```

```
}

void write_answer() {
  printf("%d\n", max_comp_size);
  for (int i = 1; i <= m; i++) {
    printf("%d %d\n", e[i].u, e[i].v);
  }
}

int main() {
  read_graph();
  dfs(1, 0);
  wipe_depths();
  orient_edges(max_comp_node, 0);
  write_answer();

  return 0;
}
```

R.8 Problema We Need More Bosses (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 300'000;
const int MAX_EDGES = 300'000;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int depth, low;
};

node nd[MAX_NODES + 1];
cell list[2 * MAX_EDGES + 1];
int n, diam;

void add_edge(int u, int v) {
  static int pos = 1;
  list[pos] = { v, nd[u].adj };
  nd[u].adj = pos++;
}

void read_data() {
```

```
  int num_edges, u, v;
  scanf("%d %d", &n, &num_edges);

  while (num_edges--) {
    scanf("%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
}

void minimize(int& x, int y) {
  x = (y < x) ? y : x;
}

void maximize(int& x, int y) {
  x = (y > x) ? y : x;
}

// Returnează distanța maximă în punți de la u la orice frunză.
int dfs(int u, int parent) {
  nd[u].depth = nd[u].low = nd[parent].depth + 1;
  int dist = 0; // distanța maximă văzută în fiii anteriori

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;

    if (!nd[v].depth) {
      int dist2 = dfs(v, u); // distanța maximă văzută în v
      if (nd[v].low > nd[u].depth) {
        dist2++; // u-v este punte
      }
      minimize(nd[u].low, nd[v].low);
      maximize(diam, dist2 + dist);
      maximize(dist, dist2);
    } else if (v != parent) {
      minimize(nd[u].low, nd[v].depth);
    }
  }

  return dist;
}

void write_answer() {
  printf("%d\n", diam);
}

int main() {
  read_data();
  dfs(1, 0);
  write_answer();
```

```
  return 0;
}
```

R.9 Problema Forbidden Cities (CSES)

înapoi

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

const int MAX_NODES = 100'000;
const int MAX_EDGES = 200'000;
const int MAX_QUERIES = 100'000;

struct node {
  std::vector<int> adj;           // lista de adiacență
  std::vector<int> queries;       // interogări în care eu sînt capătul
  std::vector<int> notifications; // interogări în care eu sînt centrul
  int tin, tout;
  int low;
  bool notify_me;
};

struct query {
  int a, b, c;
  bool red_flag;
};

node nd[MAX_NODES + 1];
query q[MAX_QUERIES + 1];
int n, num_queries;

void read_data() {
  int num_edges, u, v;
  scanf("%d %d %d", &n, &num_edges, &num_queries);

  while (num_edges--) {
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }

  for (int i = 1; i <= num_queries; i++) {
    scanf("%d %d %d", &q[i].a, &q[i].b, &q[i].c);
    nd[q[i].a].queries.push_back(i);
    nd[q[i].b].queries.push_back(i);
  }
```

```
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

bool is_ancestor(int u, int v) {
  return (nd[v].tin >= nd[u].tin) && (nd[v].tout <= nd[u].tout);
}

void send_notifications(int u) {
  for (int ind: nd[u].queries) {
    int c = q[ind].c;
    if (nd[c].notify_me) {
      nd[c].notifications.push_back(ind);
    }
  }
}

void receive_notifications(int u, int child) {
  for (int ind: nd[u].notifications) {
    if (is_ancestor(child, q[ind].a) ^ is_ancestor(child, q[ind].b)) {
      q[ind].red_flag = true;
    }
  }
}

void clear_notifications(int u) {
  nd[u].notifications.clear();
}

void dfs(int u, int parent) {
  static int time = 0;
  nd[u].tin = nd[u].low = ++time;
  nd[u].notify_me = true;

  for (int v: nd[u].adj) {
    if (!nd[v].tin) {
      dfs(v, u);
      nd[u].low = min(nd[u].low, nd[v].low);

      if (nd[v].low >= nd[u].tin) {
        // v nu îl poate ocoli pe u: u este punct de articulație
        receive_notifications(u, v);
      }
      clear_notifications(u);

    } else if (v != parent) {
      nd[u].low = min(nd[u].low, nd[v].tin);
    }
```

```
  }

  send_notifications(u);
  nd[u].notify_me = false;
  nd[u].tout = ++time;
}

void write_answers() {
  for (int i = 1; i <= num_queries; i++) {
    q[i].red_flag |= (q[i].a == q[i].c) || (q[i].b == q[i].c);
    printf(q[i].red_flag ? "NO\n" : "YES\n");
  }
}

int main() {
  read_data();
  dfs(1, 0);
  write_answers();

  return 0;
}
```

R.10 Problema Case of Computer Network (Codeforces)

‹ înapoi • versiune online

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

const int MAX_NODES = 200'000;

struct node {
  // vecinii
  std::vector<int> adj;
  // celelalte capete ale mesajelor care încep / se termină aici
  std::vector<int> pairs;
  // adîncimea mea și cea mai joasă adîncime accesibilă din subarborele meu
  int depth, low;
  // surse și destinații deschise în subarborele meu
  int num_src, num_dest;
};

struct disjoint_set_forest {
  int parent[MAX_NODES + 1];
  int n;

  void init(int n) {
    this->n = n;
```

```
    for (int u = 1; u <= n; u++) {
      parent[u] = u;
    }
  }

  int find(int u) {
    return (parent[u] == u)
      ? u
      : (parent[u] = find(parent[u]));
  }

  void merge(int u, int v) {
    parent[find(v)] = find(u);
  }
};

node nd[MAX_NODES + 1];
disjoint_set_forest ds;
int n;
bool red_flag;

void read_data() {
  int num_edges, num_queries, u, v;
  scanf("%d %d %d", &n, &num_edges, &num_queries);

  while (num_edges--) {
    scanf("%d %d", &u, &v);
    nd[u].adj.push_back(v);
    nd[v].adj.push_back(u);
  }

  for (int i = 1; i <= num_queries; i++) {
    scanf("%d %d", &u, &v);
    nd[u].num_src++;
    nd[v].num_dest++;
    nd[u].pairs.push_back(v);
    nd[v].pairs.push_back(u);
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

void endpoint_action(int u) {
  for (auto v: nd[u].pairs) {
    if (nd[v].depth) {
      int lca = ds.find(v);
      // Spune-i LCA-ului să închida o pereche (sursă, destinație).
      nd[lca].num_src--;
```

```
      nd[lca].num_dest--;
    }
  }
}

void descendant_action(int u, int parent) {
  // Propagă numerele de capete la rădăcina 2ECC.
  nd[parent].num_src += nd[u].num_src;
  nd[parent].num_dest += nd[u].num_dest;
  nd[u].num_src = 0;
  nd[u].num_dest = 0;
}

void bridge_action(int u, int parent) {
  if (nd[u].low > nd[parent].depth) {
    if (nd[u].num_src && nd[u].num_dest) {
      red_flag = true;
    }
  }
}

void dfs(int u, int parent) {
  nd[u].depth = nd[u].low = nd[parent].depth + 1;

  endpoint_action(u);

  int num_parent_edges = 0;
  for (auto v: nd[u].adj) {
    num_parent_edges += (v == parent);

    if (!nd[v].depth) {
      dfs(v, u);
      ds.merge(u, v);
      nd[u].low = min(nd[u].low, nd[v].low);
      descendant_action(v, u);
    } else if ((v != parent) || ((v == parent) && (num_parent_edges > 1))) {
      nd[u].low = min(nd[u].low, nd[v].depth);
    }
  }

  bridge_action(u, parent);
}

void dfs_driver() {
  for (int u = 1; u <= n; u++) {
    if (!nd[u].depth) {
      dfs(u, 0);
      // Asigură-te că niciun mesaj nu circulă între două componente.
      if (nd[u].num_src || nd[u].num_dest) {
        red_flag = true;
```

```
      }
    }
  }
}

void write_answer() {
  printf(red_flag ? "No\n" : "Yes\n");
}

int main() {
  read_data();
  ds.init(n);
  dfs_driver();
  write_answer();

  return 0;
}

// If it's worth doing, it's worth doing well.
```

R.11 Problema Dog Trick Competition 2 (IIOT 2023-2024 runda 4)

‹ înapoi • versiune online

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

const int MAX_NODES = 250'000;
const int MAX_EDGES = 250'000;
const int MAX_TRICKS = 250'000;
const int INFINITY = 2 * MAX_TRICKS;

struct edge_set {
  std::unordered_set<long long> set;

  void insert(int u, int v) {
    set.insert((long long)u * (MAX_NODES + 1) + v);
  }

  bool contains(int u, int v) {
    return set.contains((long long)u * (MAX_NODES + 1) + v);
  }
};

struct cell {
  int v, next;
```

```
};

struct node {
  int adj;       // lista de adiacență
  int first_pos; // prima apariție a acestui nod în t[]
  int comp;      // numărul CTC a acestui nod

  // cîmpuri pentru algoritmul lui Tarjan pentru CTC
  int time_in;
  int low;
  bool on_stack;
};

// Un interval de indici din t[] pe care o CTC îi poate servi.
struct range {
  int l, r;

  void minimize(range other) {
    if (other.l < l) {
      *this = other;
    }
  }

  // this = intervalul subarborelui nodului curent
  // child = cel mai timpuriu interval al oricărei CTC descendente
  void combine(range child) {
    if (child.l < l) {
      *this = child;
    } else if (child.l == r + 1) {
      r = child.r;
    }
  }
};

node nd[MAX_NODES + 1];
cell list[MAX_EDGES + 1];
range scc[MAX_NODES + 1];
int st[MAX_NODES], ss; // stiva DFS
int t[MAX_TRICKS + 1]; // santinelă la t[num_tricks]
edge_set eset;
int num_comps;

void minimize(int& x, int y) {
  if (y < x) {
    x = y;
  }
}

void read_data() {
  int num_tricks, n, num_edges, u, v;
```

```
  scanf("%d %d", &num_tricks, &n);
  for (int u = 1; u <= n; u++) {
    nd[u].first_pos = INFINITY;
  }
  for (int i = 1; i <= num_tricks; i++) {
    scanf("%d", &t[i]);
    minimize(nd[t[i]].first_pos, i);
  }

  scanf("%d", &num_edges);
  for (int i = 1; i <= num_edges; i++) {
    scanf("%d %d", &u, &v);
    eset.insert(u, v);
    list[i] = { v, nd[u].adj };
    nd[u].adj = i;
  }
}

void process_scc(int last) {
  int v;
  num_comps++;
  range& c = scc[num_comps]; // syntactic sugar
  c.l = INFINITY;

  // Stiva conține nodurile din CTC. Ia prima apariție în t[].
  do {
    v = st[--ss];
    nd[v].comp = num_comps;
    nd[v].on_stack = false;
    minimize(c.l, nd[v].first_pos);
  } while (v != last);

  // Extinde intervalul la dreapta.
  c.r = c.l;
  while (nd[t[c.r + 1]].comp == num_comps) {
    c.r++;
  }
}

range dfs(int u) {
  static int time = 0;
  nd[u].time_in = nd[u].low = ++time;
  nd[u].on_stack = true;
  st[ss++] = u;

  range result = { INFINITY, INFINITY };

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
```

```
    if (!nd[v].time_in) {
      // Explorează un nod nou.
      result.minimize(dfs(v));
      minimize(nd[u].low, nd[v].low);
    } else if (nd[v].on_stack) {
      // Muchie înapoi.
      minimize(nd[u].low, nd[v].time_in);
    } else {
      // Muchie la o CTC explorată anterior. Acesta este singurul moment cînd
      // schimbăm rezultatul pentru CTC curentă. Operațiile return propagă
      // aceste valori pînă la rădăcina CTC.
      result.minimize(scc[nd[v].comp]);
    }
  }

  if (nd[u].low == nd[u].time_in) {
    // u este rădăcina unei CTC.
    process_scc(u);
    scc[num_comps].combine(result);
    return scc[num_comps];
  } else {
    return result;
  }
}

void compute_score() {
  // Nodul de start este t[1] și aparține lui scc[num_comps].
  int score = 2;
  for (int i = 2; i <= scc[num_comps].r; i++) {
    score += 1 + eset.contains(t[i - 1], t[i]);
  }
  printf("%d\n", score);
}

int main() {
  read_data();
  dfs(t[1]);
  compute_score();

  return 0;
}
```

Anexa S

Probleme diverse

S.1 Problema Liars (Baraj ONI 2025)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_NODES = 5'000;
const int MOD = 666'013;

const int ANY = -1;
const int LIAR = 0;
const int TRUTH_TELLER = 1;

// Combinații posibile pentru nodul curent × părintele
const int LIAR_P_LIAR = 0;
const int LIAR_P_TT = 1;
const int TT_P_LIAR = 2;
const int TT_P_TT = 3;

struct cell {
  int v, next;
};

struct node {
  int adj;
  int num_children;
  int resp;       // numărul de vecini mincinoși declarați de u
  int assignment; // valoarea asignată lui u în soluția construită
  int ways[4];    // În câte moduri putem fixa calitatea nodului și a părintelui?
};

node nd[MAX_NODES + 1];
cell list[2 * MAX_NODES];
int n;
```

```
// Vectorul pentru combinatorică. Ne ajunge unul singur, căci îl completăm în
// fiecare nod după ce procesăm fiii. a[i] = numărul de moduri de a avea i fii
// mincinoși. Calitatea părintelui este fixată în momentul apelului. Din
// fiecare fiu vom folosi cîmpurile ways[LIAR_P_LIAR] și ways[TT_P_LIAR]
// pentru a fixa părintele ca mincinos, respectiv ways[LIAR_P_TT] și
// ways[TT_P_TT] pentru a fixa părintele ca sincer.
//
// Îl îmbrăcăm într-un struct ca să oferim funcții getter pentru indici
// incorecți.
struct combo {
  long long a[MAX_NODES + 1];
  int curr_node;

  void merge_children(int u, int parent, int c_liar, int c_tt) {
    curr_node = u;

    // Înainte de a procesa fiii, avem un singur mod de a avea 0 mincinoși.
    a[0] = 1;
    int i = 0;

    for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
      int v = list[ptr].v;
      if (v != parent) {
        i++;
        // Toți fiii sînt mincinoși.
        a[i] = a[i - 1] * nd[v].ways[c_liar] % MOD;

        for (int j = i - 1; j; j--) {
          // Aveam deja j mincinoși și adăugăm un sincer...
          long long gain_tt = a[j] * nd[v].ways[c_tt];
          // ... sau aveam j - 1 mincinoși și adăugăm un mincinos.
          long long gain_liar = a[j - 1] * nd[v].ways[c_liar];
          a[j] = (gain_tt + gain_liar) % MOD;
        }

        // Toți fiii sînt sinceri.
        a[0] = a[0] * nd[v].ways[c_tt] % MOD;
      }
    }
  }

  int get_sum() {
    long long result = 0;
    for (int i = 0; i <= nd[curr_node].num_children; i++) {
      result += a[i];
    }
    return result % MOD;
  }
```

```
  int get(int k) {
    if ((k >= 0) && (k <= nd[curr_node].num_children)) {
      return a[k];
    } else {
      return 0;
    }
  }
};

combo c;

void add_edge(int u, int v) {
  static int ptr = 1;
  list[ptr] = { v, nd[u].adj };
  nd[u].adj = ptr++;
  nd[u].num_children++;
}

void read_data() {
  FILE* f = fopen("liars.in", "r");
  fscanf(f, "%d", &n);
  for (int u = 1; u <= n; u++) {
    fscanf(f, "%d", &nd[u].resp);
  }
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    fscanf(f, "%d %d", &u, &v);
    add_edge(u, v);
    add_edge(v, u);
  }
  fclose(f);

  for (int u = 2; u <= n; u++) {
    nd[u].num_children--; // toate nodurile în afară de 1 au un părinte
  }
}

void count_dfs(int u, int parent) {
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      count_dfs(v, u);
    }
  }

  int r = nd[u].resp;
  // Cazul I: u este sincer.
  c.merge_children(u, parent, LIAR_P_TT, TT_P_TT);

  // Dacă și părintele este sincer, cei r mincinoși se află printre
```

```
  // fii.
  nd[u].ways[TT_P_TT] = c.get(r);

  // Dacă părintele este mincinos, ceilalți r - 1 mincinoși se află
  // printre fii.
  nd[u].ways[TT_P_LIAR] = c.get(r - 1);

  // Cazul II: u este mincinos.
  c.merge_children(u, parent, LIAR_P_LIAR, TT_P_LIAR);
  int sum = c.get_sum();

  // Dacă părintele este sincer, u trebuie să aibă orice în afară de r
  // mincinoși printre fii.
  nd[u].ways[LIAR_P_TT] = (sum + MOD - c.get(r)) % MOD;

  // Dacă și părintele este mincinos, u trebuie să aibă orice în afară de
  // r - 1 mincinoși printre fii.
  nd[u].ways[LIAR_P_LIAR] = (sum + MOD - c.get(r - 1)) % MOD;
}

// Atribuie toți fiii impuși.
int assign_forced(int u, int parent) {
  int c_liar = (nd[u].assignment == LIAR) ? LIAR_P_LIAR : LIAR_P_TT;
  int c_tt = (nd[u].assignment == LIAR) ? TT_P_LIAR : TT_P_TT;
  int num_assigned = 0;

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      if (!nd[v].ways[c_liar]) {
        nd[v].assignment = TRUTH_TELLER;
      } else if (!nd[v].ways[c_tt]) {
        nd[v].assignment = LIAR;
        num_assigned++;
      } else {
        nd[v].assignment = ANY;
      }
    }
  }

  return num_assigned;
}

void assign_optional(int u, int parent, int liars_left) {
  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (nd[v].assignment == ANY) {
      if (liars_left) {
        nd[v].assignment = LIAR;
        liars_left--;
```

```
      } else {
        nd[v].assignment = TRUTH_TELLER;
      }
    }
  }
}

// Atribuie valori fiilor lui u. Nodul lui u are deja valoare. Bug: dacă
// numărul de posibilități este 0 modulo MOD, el va fi considerat incorect ca
// fiind 0.
void assign_dfs(int u, int parent) {
  int assigned_liars = assign_forced(u, parent);
  // Și părintele poate minți (ce familie de cacao).
  assigned_liars += parent && (nd[parent].assignment == LIAR);

  int liars_left;
  if (nd[u].assignment == TRUTH_TELLER) {
    liars_left = nd[u].resp - assigned_liars;
  } else if (assigned_liars == nd[u].resp) {
    liars_left = 1; // Nu ne putem opri acum, căci u spune adevărul.
  } else {
    liars_left = 0;
  }

  assign_optional(u, parent, liars_left);

  for (int ptr = nd[u].adj; ptr; ptr = list[ptr].next) {
    int v = list[ptr].v;
    if (v != parent) {
      assign_dfs(v, u);
    }
  }
}

void write_answer() {
  FILE* f = fopen("liars.out", "w");
  int ans = (nd[1].ways[LIAR_P_TT] + nd[1].ways[TT_P_TT]) % MOD;
  fprintf(f, "%d\n", ans);
  for (int u = 1; u <= n; u++) {
    fprintf(f, "%d ", nd[u].assignment);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  count_dfs(1, 0);
  nd[1].assignment = nd[1].ways[TT_P_TT] ? TRUTH_TELLER : LIAR;
  assign_dfs(1, 0);
```

```
  write_answer();

  return 0;
}
```