2. Problemele concursului de programare "Bursele Agora"

2.1. Problemele propuse în ediția 2000-2001

2.1.1. Triangularizare

Se consideră un poligon convex cu n laturi; pentru acest poligon trebuie să se determine în câte moduri poate fi acesta triangularizat. Prin triangularizare se înțelege împărțirea poligonului în $n - 2$ triunghiuri sau trasarea a $n - 3$ diagonale care nu se intersectează decât în vârfurile poligonului.

Date de intrare

Fişierul de intrare **`POLIGON.IN`** conține un singur număr întreg n, reprezentând numărul de laturi ale poligonului convex.

Date de ieşire

Fişierul de ieşire **`POLIGON.OUT`** trebuie să conțină o singură linie pe care se va afla numărul triangularizărilor posibile.

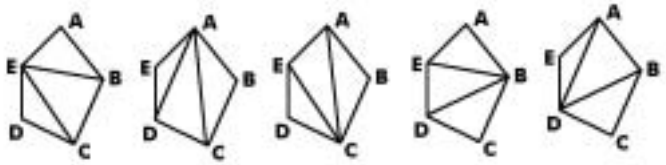
Restricție

- $1 \leq n \leq 2457$.

Exemplu

`POLIGON.IN`
```
5
```

`POLIGON.OUT`
```
5
```



Timp de execuție: 1 secundă/test

2.1.2. Divizori

Se consideră un număr natural nenul n; să se determine care este cel mai mic număr natural care are exact n divizori.

Date de intrare
Fişierul de intrare **DIV.IN** conţine un singur număr întreg *n*, reprezentând numărul de divizori pe care trebuie să-i aibă numărul furnizat la ieşire.

Date de ieşire
Fişierul de ieşire **DIV.OUT** trebuie să conţină o singură linie pe care se va afla un număr *d* care are exact *n* divizori.

Restricţie
- $1 \le n \le 2457$.

Exemplu

DIV.IN	DIV.OUT
6	12

Timp de execuţie: 1 secundă/test

2.1.3. Premiere

La un concurs trebuie premiaţi *n* participanţi. Sponsorii au pus la dispoziţia organizatorilor *k* categorii de premii, din fiecare categorie fiind disponibile oricâte premii. Va trebui să determinaţi numărul modalităţilor de premiere a câştigătorilor, astfel încât fiecare să primească un singur premiu şi să fie oferit cel puţin un premiu din fiecare categorie.

Date de intrare
Fişierul de intrare **PREMII.IN** conţine două numere întregi *n* şi *k*, reprezentând numărul de premianţi, respectiv numărul de categorii de premii. Cele două numere sunt separate printr-un singur spaţiu.

Date de ieşire
Fişierul de ieşire **PREMII.OUT** trebuie să conţină o singură linie pe care se va afla numărul posibilităţilor de premiere.

Restricţie
- $1 \le k \le n \le 100$.

Exemplu

PREMII.IN	PREMII.OUT
5 3	25

Timp de execuţie: 1 secundă/test

2.1.4. Acadele

O fabrică de dulciuri produce zilnic N acadele care trebuie ambalate în B cutii, fiecare cutie conținând L acadele. Ciclul de producție durează D zile şi satisface următoarele criterii:

- parametrii de producție N, B ($B > 1$) şi L sunt aceiaşi pe toată durata unui ciclu de producție;
- acadelele produse în aceeaşi zi au ambalaje diferite;
- există exact N tipuri de ambalaje folosite pe toată durata unui ciclu de producție;
- la sfârşitul unui ciclu de producție, pentru fiecare pereche de ambalaje distincte există o singură cutie care conține exact două acadele care au aceste ambalaje, celelalte acadele fiind ambalate diferit față de cele două.

De exemplu, dacă se produc zilnic 9 acadele, ciclul de producție este de 4 zile, atunci în fiecare zi se pot folosi 3 cutii care conțin câte 3 acadele. Cele 9 tipuri de ambalaje pot fi folosite astfel:

```
            cutia 1     cutia 2     cutia 3
ziua 1:      123         468         579
ziua 2:      145         278         369
ziua 3:      167         249         358
ziua 4:      189         256         347
```

Scrieți un program care pentru un număr N dat determină valoarea L, astfel încât valoarea $D - B$ să fie minimă. Dacă există mai multe soluții se va alege cea pentru care valoarea L este maximă.

Date de intrare

Fişierul de intrare **ACADELE.IN** conține un singur număr întreg N, reprezentând numărul de acadele produse zilnic.

Date de ieşire

Fişierul de ieşire **ACADELE.OUT** va conține, pe o singură linie, valorile L, B şi D separate printr-un singur spațiu. Dacă nu există soluție se va scrie 0 0 0.

Restricție

- $1 \le N \le 2000000000$.

Exemplu

ACADELE.IN	ACADELE.OUT
5 3	25

Timp de execuție: 1 secundă/test

2.1.5. Galbeni

Se consideră n grămezi de galbeni. Dacă o grămadă A conține a galbeni, o grămadă B conține b galbeni şi $b \le a$, atunci din grămada A pot fi mutați b galbeni în grămada B. Efectuând mai multe operații, trebuie să obținem cel mult două grămezi de galbeni (restul de $n - 2$ grămezi nu vor mai conține nici un galben). Grămezile sunt etichetate cu numere de la 1 la n.

Date de intrare

Fişierul de intrare **GALBENI.IN** va conține pe prima linie numărul n, iar pe a doua linie n numere reprezentând numărul de galbeni din cele n grămezi.

Date de ieşire

Fişierul de ieşire **GALBENI.OUT** va conține câte o linie pentru fiecare operație efectuată. Fiecare linie va conține două numere; primul va indica grămada din care se iau galbeni, iar al doilea grămada în care se pun galbeni.

Restricții şi precizări

- $1 \le n \le 1000$;
- numărul total al galbenilor nu poate depăşi valoarea 2000000000;
- numărul de operații efectuate nu trebuie să fie neapărat minim.

Exemplu

GALBENI.IN	GALBENI.OUT
4	3 1
1 2 3 4	1 2
	2 4

Timp de execuție: 1 secundă/test

2.1.6. Fantome

Un grup de n vânători luptă cu n fantome. Vânătorii şi fantomele se află în puncte având coordonate reale. Nu există trei puncte coliniare în care să se afle vânători sau fantome. Vânătorii au arme fotonice cu ajutorul cărora pot trimite raze care distrug fantomele. Fiecare vânător va trebui să distrugă câte o fantomă. Armele sunt descărcate în acelaşi timp şi razele nu se intersectează, deoarece efectele nu mai pot fi controlate. Scrieți un program care determină care fantomă trebuie distrusă de fiecare vânător.

Date de intrare

Fişierul de intrare **FANTOME.IN** conține pe prima linie numărul n. Următoarele n linii conțin coordonatele punctelor în care se află vânătorii şi următoarele n linii conțin coordonatele punctelor în care se află fantomele.

Date de ieşire
Fişierul de ieşire **FANTOME.OUT** va conține n numere, un număr pe câte o linie, a i-a linie indicând fantoma distrusă de al i-lea vânător. Fantomele sunt etichetate cu numere de la 1 la n.

Restricții şi precizări
- $1 \le n \le 500$;
- punctele au coordonatele din intervalul [0, 1000] cu trei zecimale exacte.

Exemplu

FANTOME.IN	FANTOME.OUT
3 0 0 4 3 4 0 2 1 2 5 5 5	2 1 3

Timp de execuție: 1 secundă/test

2.1.7. Unitate

Se consideră o matrice de dimensiuni $m \times n$ care conține doar elemente din mulțimea {0, 1}. Fiecare linie şi coloană reprezintă scrierea binară a unui număr natural (în cadrul unei linii numărul se citeşte de la dreapta la stânga, iar în cadrul unei coloane numărul se citeşte de jos în sus). Singura operație admisă este adunarea binară a unei unități la numărul de pe o linie sau de pe o coloană. Determinați numărul de unități care trebuie adunate pentru a obține o matrice ale cărei elemente au toate valoarea 1.

Date de intrare
Prima linie a fişierului de intrare **UNITATE.IN** conține valorile m şi n, separate printr-un spațiu. Următoarele m linii conțin câte n numere reprezentând valorile elementelor matricei. Prima dintre aceste linii corespunde primei linii a matricei, cea de-a doua dintre linii corespunde celei de-a doua linii a matricei etc. Primul element de pe o linie corespunde elementului aflat pe prima coloană a matricei, cel de-al doilea element de pe linie corespunde elementului de pe a doua coloană etc. Numerele de pe o linie sunt separate prin spații.

Date de ieşire
Fişierul de ieşire **UNITATE.OUT** va conține o singură linie pe care se va afla numărul operațiilor care trebuie efectuate.

Restricție

- $1 \le m, n \le 100$.

Exemplu

UNITATE.IN	UNITATE.OUT
2 1 0 0 1	2

Timp de execuție: 1 secundă/test

2.1.8. Sfere

Dacă avem o sferă, atunci spațiul este împărțit în două zone: cea din interiorul sferei şi cea din exterior. Dacă avem două sfere, atunci spațiul poate fi împărțit în trei sau patru zone. Dacă sferele nu se intersectează vom avea trei zone: interiorul primei sfere, interiorul celei de-a doua sfere şi zona exterioară sferelor. Dacă sferele se intersectează atunci vor fi patru zone: zona comună celor două sfere, zona din interiorul primei sfere (fără zona comună), zona din interiorul celei de-a doua sfere (fără zona comună) şi zona din exteriorul sferelor. Dându-se *n* sfere, determinați care este numărul maxim de zone în care poate fi împărțit spațiul aranjând sferele într-un mod convenabil.

Date de intrare

Fişierul de intrare **SFERE.IN** va conține pe prima linie numărul *n* al sferelor.

Date de ieşire

Fişierul de ieşire **SFERE.OUT** va conține, pe o singură linie, numărul maxim de zone în care poate fi împărțit spațiul.

Restricție

- $2 \le n \le 1000$.

Exemplu

SFERE.IN	SFERE.OUT
2	4

Timp de execuție: 1 secundă/test

2.1.9. Asemănare

Se dau două poligoane în plan, fiecare având *n* vârfuri. Poligoanele sunt date prin coordonatele vârfurilor lor, în ordine trigonometrică, sau invers trigonometrică. Se cere să se verifice dacă cele două poligoane sunt asemenea.

Date de intrare
Prima linie a fişierului de intrare **POLIGON.IN** conţine numărul *n* al vârfurilor poligoanelor. Următoarele *n* linii conţin coordonatele vârfurilor primului poligon. Următoarele *n* linii conţin coordonatele vârfurilor celui de-al doilea poligon.

Date de ieşire
Datele de ieşire vor fi scrise în fişierul **POLIGON.OUT**. Dacă poligoanele sunt asemenea, fişierul de ieşire va conţine *n* numere (un număr pe linie), a *i*-a linie indicând vârful din al doilea poligon care corespunde celui de-al *i*-lea vârf din primul poligon (unghiurile corespunzătoare acestor două vârfuri trebuie să fie egale). Dacă poligoanele nu sunt asemenea, fişierul de ieşire va conţine doar cifra 0.

Restricţii şi precizări
- $3 \le n \le 100$;
- coordonatele vârfurilor poligoanelor sunt numere reale cu o precizie de trei zecimale exacte din intervalul [0, 1000].

Exemplu

POLIGON.IN	POLIGON.OUT
5	2
0.000 0.000	1
0.000 4.000	5
2.000 6.000	4
4.000 4.000	3
4.000 0.000	
0.000 1.000	
0.000 3.000	
2.000 3.000	
2.000 1.000	
1.000 0.000	

Timp de execuţie: 1 secundă/test

2.1.10. Dreptunghi
Se consideră un dreptunghi ale cărui dimensiuni sunt numere naturale. Dreptunghiul trebuie descompus într-un număr minim de pătrate ale căror laturi sunt paralele cu laturile dreptunghiului. Într-un dreptunghi se poate efectua o tăietură completă (de la o margine la alta) paralelă cu o latură.

Date de intrare
Prima linie a fişierului de intrare **DREPT.IN** conţine valorile *m* şi *n*, separate printr-un spaţiu, reprezentând dimensiunile dreptunghiului.

Date de ieşire
Fişierul de ieşire **DREPT.OUT** va conține numărul minim de pătrate în care poate fi împărțit dreptunghiul, respectând regula de efectuare a tăieturilor.

Restricție
- $1 \le m, n \le 100$.

Exemplu

DREPT.IN	**DREPT.OUT**
2 4	2

Timp de execuție: 1 secundă/test

2.1.11. Alb şi negru

Pe o tablă liniară cu $2 \cdot n + 1$ căsuțe se află n bile albe (pe primele n poziții) şi n bile negre (pe ultimele n poziții), căsuța din mijloc fiind liberă. O bilă albă poate fi deplasată în căsuța din dreapta ei dacă aceasta este liberă (operația `W`) sau poate sări peste bila din dreapta ei, dacă a doua căsuță din dreapta este liberă (operația `w`). O bilă neagră poate fi deplasată în căsuța din stânga ei dacă aceasta este liberă (operația `B`) sau poate sări peste bila din stânga ei, dacă a doua căsuță din stânga este liberă (operația `b`). După un număr oarecare de astfel de operații bilele albe trebuie să ajungă pe ultimele n poziții, iar cele negre pe primele n poziții.

Date de intrare
Fişierul de intrare **ALBNEGRU.IN** va conține pe prima linie valoarea n.

Date de ieşire
Fişierul de ieşire **ALBNEGRU.OUT** va conține, pe un singur rând şi fără spații, codificarea operațiilor efectuate.

Restricție
- $1 \le n \le 1000$.

Exemplu

ALBNEGRU.IN	**ALBNEGRU.OUT**
2	WbBwwBbW

Timp de execuție: 1 secundă/test

2.1.12. Coeficient

Se consideră n oraşe care sunt legate prin m străzi cu dublu sens. Fiecare stradă are ataşat un coeficient întreg c. Între două oraşe poate exista cel mult o stradă. Coeficientul unui traseu este dat de produsul coeficienţilor străzilor care îl formează. Se cere să se determine coeficientul minim al unui traseu de la oraşul 1 la oraşul n. Se ştie că există întotdeauna un astfel de traseu.

Date de intrare

Prima linie a fişierului de intrare **COEF.IN** conţine numărul n al oraşelor şi numărul m al străzilor separate printr-un singur spaţiu. Următoarele m linii conţin câte trei numere x, y şi z (separate prin spaţii) cu semnificaţia: între oraşele x şi y există o stradă de coeficient z.

Date de ieşire

Fişierul **COEF.OUT** va conţine coeficientul minim al unui traseu între oraşele 1 şi n.

Restricţii şi precizări

- $2 \le n \le 1000$;
- $2 \le m \le 5000$;
- coeficienţii sunt numere întregi cuprinse între 1 şi 1000;
- coeficientul minim al traseului va fi un număr care va conţine cel mult 5000 de cifre.

Exemplu

COEF.IN	COEF.OUT
5 6 1 2 2 1 4 2 2 3 5 3 4 2 3 5 4 4 5 9	16

Timp de execuţie: 1 secundă/test

2.1.13. Matrice

Se consideră o matrice având dimensiunile $m \times n$ care are toate elementele egale cu 1. O operaţie constă în schimbarea semnului tuturor elementelor de pe o linie sau de pe o coloană. În urma mai multor astfel de operaţii trebuie obţinute exact k valori -1 în întreaga matrice.

Date de intrare
Prima linie a fişierului de intrare **MATRICE.IN** conţine valorile *m* şi *n*, separate printr-un spaţiu, reprezentând dimensiunile matricei. A doua linie conţine numărul *k* de valori -1 care trebuie obţinute.

Date de ieşire
În cazul în care nu există soluţie, fişierul de ieşire **MATRICE.OUT** va conţine doar valoarea 0. Dacă există soluţie, atunci fişierul va conţine câte o linie pentru fiecare operaţie efectuată. O operaţie este descrisă printr-un caracter *c*, urmat de un număr *x* (fără ca acestea să fie separate prin spaţii). Caracterul *c* va avea valoarea 'L' dacă operaţia este efectuată asupra unei linii şi valoarea 'C' dacă este efectuată asupra unei coloane. Valoarea *x* va indica numărul de ordine a liniei sau coloanei asupra căreia este efectuată operaţia (numerotarea începe de la 1).

Restricţii
- $1 \le m, n \le 500$;
- $1 \le k \le 250.000$.

Exemplu

MATRICE.IN	**MATRICE.OUT**
3 3 4	L2
	C2

Timp de execuţie: 1 secundă/test

2.1.14. Inversiuni

Se consideră un vector *a* care conţine *n* elemente şi se cere determinarea numărului de perechi (i, j) cu proprietatea $i < j$ şi $a_i > a_j$.

Date de intrare
Fişierul de intrare **INVERS.IN** va conţine pe prima linie numărul *n* al elementelor vectorului. Pe următoarea linie se află cele *n* elemente ale vectorului, separate prin spaţii.

Date de ieşire
Fişierul de ieşire **INVERS.OUT** va conţine numărul de perechi (i, j) care au proprietatea cerută.

Restricţie
- $2 \le n \le 100$.

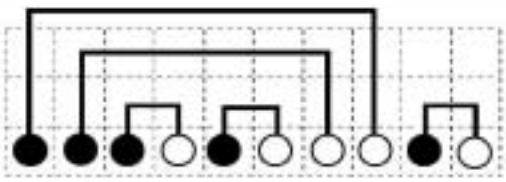
Exemplu

INVERS.IN	INVERS.OUT
4	5
4 2 3 1	

Timp de execuţie: 1 secundă/test

2.1.15. Bile

Se consideră *n* bile negre şi *n* bile albe înşirate pe o bară dreaptă. Aceste bile urmează să fie legate între ele două câte două prin alte bare care nu pot să se intersecteze şi să se suprapună. În acest scop se vor folosi, în cazul



fiecărei perechi, trei bucăţi de bare: două verticale şi una orizontală, conform figurii. O pereche este formată dintr-o bilă albă şi una neagră, sau dintr-una neagră şi una albă.

Date de intrare

Fişierul de intrare **BILE.IN** va conţine pe prima linie numărul *n* al bilelor albe (şi care este totodată şi numărul bilelor negre). Pe următoarea linie se află o succesiune de cifre egale cu 0 şi 1, neseparate prin spaţii, reprezentând bilele fixate pe bară; valoarea 0 corespunde bilelor albe, iar valoarea 1 corespunde bilelor negre.

Date de ieşire

Fişierul de ieşire **BILE.OUT** va conţine pe prima linie un număr natural nenul, reprezentând lungimea minimă totală a barelor folosite, iar fiecare dintre următoarele *n* linii va conţine două numere naturale nenule *x* şi *y*, separate printr-un spaţiu, reprezentând numărul de ordine a două bile legate în pereche; perechile care se vor scrie în fişier vor fi ordonate crescător în funcţie de valoarea *x*.

Restricţie

- $2 \le n \le 100$.

Exemplu

BILE.IN	BILE.OUT
10	31
1110100010	1 8
	2 7
	3 4
	5 6
	9 10

Timp de execuţie: 1 secundă/test

2.1.16. Robot

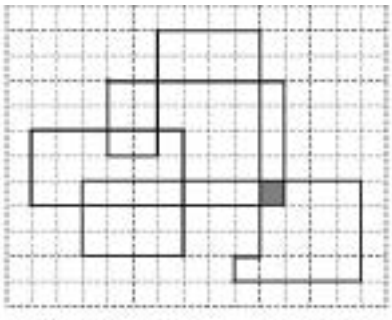
Un robot se poate mişca în plan doar pe direcţiile verticală, notate cu *up* ('U') şi *down* ('D') respectiv orizontală, notate cu *left* ('L') şi *right* ('R'). Robotul porneşte dintr-un anumit punct şi după ce îşi îndeplineşte sarcinile, se întoarce tot în acest punct. Mişcarea robotului se descrie printr-o secvenţă de comenzi, conform căreia acesta se va deplasa de-a lungul unor linii care formează un caroiaj în care pătrăţelele au laturile egale cu 1.

Robotul, în timp ce îşi parcurge drumul, trasează o linie în urma lui. Datorită faptului că traseul se sfârşeşte în acelaşi punct din care a pornit, se pot forma figuri geometrice, printre care şi dreptunghiuri.

Vom numi dreptunghi vid acel dreptunghi care se formează din segmente aparţinând traseului şi care nu este intersectat de alte segmente din traseu.

Determinaţi cel mai mic dreptunghi vid care se formează din traseul robotului.

În imaginea următoare este prezentat un posibil traseu al robotului. Cel mai mic dreptunghi vid care se formează este haşurat.



Date de intrare

Fişierul de intrare **ROBOT.IN** va conţine pe prima linie două numere naturale *x* şi *y* separate printr-un singur spaţiu, reprezentând coordonatele punctului de pornire. Pe următoarea linie se află un număr natural nenul *c*, reprezentând numărul de comenzi pe care le execută robotul, iar fiecare dintre următoarele *c* linii conţine o pereche formată dintr-un caracter din mulţimea {'U', 'D', 'L', 'R'}, având semnificaţia de mai sus, şi un număr care reprezintă numărul de pătrăţele cu care se deplasează robotul în direcţia specificată de caracter. Caracterul este separat de număr printr-un spaţiu.

Date de ieşire

Fişierul de ieşire **ROBOT.OUT** va conţine pe prima linie o pereche de numere naturale, separate printr-un spaţiu, reprezentând coordonatele colţului stânga-jos al dreptunghiului vid de arie minimă. A doua linie conţine două numere naturale separate printr-un spaţiu, reprezentând coordonatele colţului dreapta-sus al dreptunghiului vid de arie minimă.

Restricție

- coordonatele în care se va afla robotul vor fi din intervalul [0, 100].

Exemplu

ROBOT.IN	ROBOT.OUT
4 7 8 R 6 D 4 L 3 U 2 R 5 D 4 L 8 U 6	7 3 10 5

Timp de execuție: 1 secundă/test

2.1.17. Longhorn

Viitorul sistem de operare Microsoft Windows, cunoscut acum sub numele de cod Longhorn, va fi lansat, dacă nu se întâmplă nimic neprevăzut, în anul 2004. După achiziționarea produsului, mulți utilizatori vor dori să instaleze noul sistem de operare "pe curat".

Ei vor formata discul hard, vor instala noul sistem de operare, apoi vor dori să instaleze aplicațiile preferate. După cum ştiți, instalarea unei aplicații necesită, uneori, un anumit număr de resetări ale sistemului. Sarcina voastră este de a-i ajuta pe utilizatori să instaleze toate aplicațiile, resetând calculatorul de cât mai puține ori.

Pentru fiecare program de instalare este cunoscut numărul x de resetări necesare, precum şi lista aplicațiilor care trebuie să fie deja instalate pentru ca acesta să poată fi lansat în execuție.

Un program de instalare funcționează astfel:

- este lansat în execuție;
- se execută primul pas al instalării (eventual, în paralel cu execuția altor programe de instalare);
- se aşteaptă finalizarea execuției pasului curent de către toate programele de instalare aflate în execuție;
- se resetează sistemul;
- se trece la al doilea pas etc.

La un moment dat se pot afla în execuție oricâte programe de instalare, dacă aplicațiile necesare pentru execuția lor au fost deja instalate.

Să presupunem că dorim să instalăm aplicațiile *Microsoft Office*, *Microsoft Visual Studio.NET*, *Microsoft DirectX* şi *Microsoft Internet Explorer*. Pentru a instala *Office* este nevoie de patru resetări, pentru Visual Studio este nevoie de trei resetări, pentru *DirectX* de două resetări şi pentru *Internet Explorer* de o resetare. *Internet Explorer* şi *DirectX* pot fi instalate fără să fie instalată nici o altă aplicație. *Visual Studio* poate fi instalat numai dacă sunt instalate *DirectX* şi *Internet Explorer*, iar *Office* dacă a fost instalat *Internet Explorer*.

La primul pas se vor lansa programele de instalare pentru *Internet Explorer* şi *DirectX*. După resetare, *Internet Explorer* este instalat, iar pentru *DirectX* se trece la al doilea pas al instalării.

Datorită faptului că *Internet Explorer* este instalat, se poate lansa în execuție şi programul de instalare pentru *Office*. La acest al doilea pas se află în execuție programele de instalare pentru *DirectX* (al doilea pas) şi *Office* (primul pas).

După resetare, *DirectX* este instalat, iar pentru *Office* se trece la al doilea pas. Datorită faptului că *DirectX* este instalat, se poate lansa în execuție şi programul de instalare pentru *Visual Studio*. La acest al treilea pas se află în execuție programele de instalare pentru *Office* (al doilea pas) şi *Visual Studio* (primul pas).

După resetare se vor afla în execuție programele de instalare pentru *Office* (al treilea pas) şi *Visual Studio* (al doilea pas). După o nouă resetare se vor afla în execuție programele de instalare pentru *Office* (al patrulea pas) şi *Visual Studio* (al treilea pas). Va avea loc o ultimă resetare, după care şi aceste două programe sunt instalate. În total au avut loc cinci resetări.

Date de intrare

Fişierul de intrare **`LONGHORN.IN`** va conține pe prima linie numărul n al aplicațiilor care trebuie instalate. Pe fiecare a i-a linie dintre următoarele n linii se află mai multe numere separate prin spații. Primul număr reprezintă numărul de resetări pentru aplicația i, al doilea număr reprezintă numărul na de aplicații care trebuie deja să fie instalate, iar restul de na numere reprezintă numerele de ordine ale aplicațiilor care trebuie să fie deja instalate (dacă este cazul).

Date de ieşire

Fişierul de ieşire **`LONGHORN.OUT`** va conține pe prima linie numărul total de resetări r, iar fiecare dintre următoarele r linii conțin un număr oarecare de numere naturale nenule separate între ele prin spații, reprezentând numerele de ordine ale aplicațiilor a căror programe de instalare se află în execuție la pasul respectiv.

Restricție

- $2 \leq n \leq 10.000$.

Exemplu

LONGHORN.IN	LONGHORN.OUT
4	5
4 1 4	3 4
3 2 3 4	1 4
2 0	1 2
1 0	1 2
	1 2

Timp de execuție: 1 secundă/test

2.2. Problemele propuse în ediția 2001-2002

2.2.1. Dune

După moartea tatălui său, *Paul Atreides* s-a refugiat în deşert printre fremeni. Aici el a primit numele *Muad'Dib* şi a început să îi organizeze pe oamenii deşertului pentru a lupta împotriva *Harkonnienilor* în vederea redobândirii de către *Casa Atreides* a controlului asupra mirodeniei, bogăția inestimabilă a planetei-deşert *Arrakis*. Doar fremenii cunoşteau adevărata identitate al lui *Muad'Dib*, dar faima sa de profet a ajuns până pe *Giedi Prim*, planeta de reşedință a *Casei Harkonnen*. Baronul *Vladimir Harkonnen* şi-a dat seama că ar fi bine să cunoască acțiunile celui care a devenit într-un timp foarte scurt simbolul fremenilor. În acest scop, el a reuşit să infiltreze printre fremeni un spion care avea asupra sa un dispozitiv de urmărire. *Paul Muad'Dib* şi-a dat seama că este urmărit şi s-a gândit să plece în deşert. Totuşi, spionul a reuşit să îi ascundă în interiorul distraiului un aparat de urmărire rudimentar. Acest aparat este capabil de a detecta doar direcția de mers, dar nu şi distanțele parcurse.

Planeta *Dune* (denumirea dată de fremeni *Arrakisului*) este descrisă printr-un tablou bidimensional de dimensiuni $m \times n$. O regiune stâncoasă este reprezentată de caracterul '+', o regiune deşertică prin caracterul '.', iar *sietch*-ul din care pleacă *Muad'Dib* prin caracterul '*'.

Se ştie că pericolul principal al planetei *Dune* îl reprezintă uriaşii viermi de nisip. Doar cei foarte bine antrenați pot supraviețui întâlnirii cu aceşti monştri care pot avea lungimi de câțiva kilometri. Totuşi, trăind în deşert, fremenii au descoperit cum pot să folosească aceşti viermi în avantajul lor. Ținuturile stâncoase sunt controlate de către *Harkonnieni* deoarece aici nu există pericolul viermilor. Din aceste motive tânărul *Atreides* se deplasează doar prin deşert. Baronul a aflat acest lucru şi încearcă să folosească informația în avantajul său.

Pentru aceasta îl cheamă pe mentatul *Thufir Hawat* şi îi cere să determine şansele pe care le are de a-l găsi pe profetul fremenilor. În acest scop mentatul trebuie să determine numărul pozițiilor în care se poate afla *Muad'Dib*. *Hawat* ştie că fremenul se

poate deplasa doar pe orizontală şi pe verticală şi că dispozitivul de urmărire va detecta orice schimbare de direcţie. După schimbarea unei direcţii *Muad'Dib* se deplasează cu cel puţin o poziţie în acea direcţie, şi modificările de direcţie au loc întotdeauna cu 90°.

Date de intrare
Prima linie a fişierului de intrare **DUNE.IN** conţine numerele *m* şi *n*, separate printr-un singur spaţiu. Următoarele *m* linii conţin câte *n* caractere care descriu suprafaţa planetei (caracterele admise sunt '.', '+' şi '*' ; ele au semnificaţia prezentată anterior). Următoarea linie va descrie informaţiile furnizate de dispozitivul de urmărire, neseparate prin spaţii. Litera 'N' indică o deplasare spre nord (spre prima linie a matricei), litera 'V' o deplasare spre vest (spre prima coloană), litera 'S' o deplasare spre sud (spre ultima linie), iar litera 'E' o deplasare spre est (spre ultima coloană).

Date de ieşire
Fişierul de ieşire **DUNE.OUT** va conţine un singur număr care va indica numărul poziţiilor în care se poate afla *Muad'Dib*.

Restricţii şi precizări
- $1 \le m, n \le 50$;
- numărul schimbărilor de direcţie este cel mult 1000;
- *sietch-ul* din care porneşte *Paul* este considerat a fi o regiune stâncoasă;
- *Paul* nu poate ieşi în afara matricei.

Exemplu

DUNE.IN
```
4 5
.....
.+.+.
+....
.++.*
NVS
```

DUNE.OUT
```
4
```

Timp de execuţie: 1 secundă/test

2.2.2. Matrice

Se consideră două matrice pătratice binare (valorile elementelor pot fi 0 sau 1) *A* şi *B*, care au *N* linii şi *N* coloane şi matricea $M_0 = [1]$, o matrice de dimensiuni 1×1 al cărei unic element are valoarea 1.

Regula matricei pentru o matrice *M* de dimensiuni $k \times k$ se aplică astfel:

- fiecare element din matricea M care are valoarea 1 este înlocuit cu matricea A;
- fiecare element din matricea M care are valoarea 0 este înlocuit cu matricea B.

Astfel se obține o matrice de dimensiuni $k \cdot N \times k \cdot N$.
Regula matricei va fi folosită de 7 ori astfel:

- la început, ea este aplicată asupra matricei M_0 şi duce la obținerea matricei $M_1 = A$;
- la al doilea pas, ea este aplicată asupra matricei $M_1 = A$ şi se obține matricea M_2;
- la al treilea pas, este aplicată asupra matricei M_2 şi se obține matricea M_3;
- la următorul pas, este aplicată asupra matricei M_3 şi se obține matricea M_4;
- urmează aplicarea regulii asupra matricei M_4 şi obținerea matricei M_5;
- după aplicarea regulii asupra matricei M_5 se obține matricea M_6;
- la ultimul pas, regula matricei este aplicată asupra matricei M_6 şi se obține matricea M_7.

Sarcina voastră este de a determina poziția celui de-al X-lea element cu valoarea 1 din matricea M_7. Pentru a determina poziția celui de-al X-lea element cu valoarea 1, veți parcurge matricea M_7 linie cu linie (de la prima până la ultima) şi veți citi fiecare linie de la stânga la dreapta.

Date de intrare
Prima linie a fişierului de intrare **`MATRICE.IN`** va conține un număr întreg N, care va indica dimensiunile matricelor A şi B.

Fiecare dintre următoarele N linii va conține câte N valori binare (0 sau 1) separate printr-un singur spațiu, reprezentând cele $N \cdot N$ elemente ale matricei A. Pe prima dintre aceste linii se vor afla elementele corespunzătoare primei linii a matricei, pe cea de-a doua linie cele corespunzătoare celei de-a doua linii a matricei etc. Primul element dintr-o linie va corespunde elementului de pe prima coloană a liniei respective, al doilea va corespunde elementului de pe a doua coloană etc.

Fiecare dintre următoarele N linii va conține câte N valori binare (0 sau 1) separate printr-un singur spațiu, reprezentând cele $N \cdot N$ elemente ale matricei B. Pozițiile elementelor sunt identice cu cele precizate pentru matricea A.

Pe ultima linie a fişierului de intrare se va afla numărul natural X.

Date de ieşire
Fişierul de ieşire **`MATRICE.OUT`** va conține o singură linie pe care se vor afla două numere întregi, separate printr-un spațiu, care vor indica linia şi coloana în care se află cel de-al X-lea element cu valoarea 1 din matricea M_7.

Restricții și precizări

- $2 \le N \le 5$;
- matricea are cel puțin X elemente cu valoarea 1.

Exemplu

MATRICE.IN
```
2
1 1
1 1
0 1
1 0
131
```

MATRICE.OUT
```
2 3
```

Timp de execuție: 1 secundă/test

2.2.3. Templu

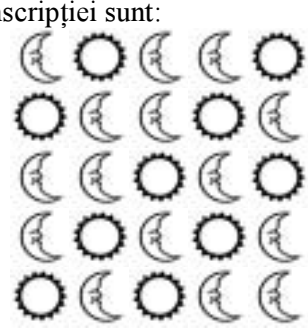
Arheologii au descoperit ruinele unui vechi templu. Pe un perete se află o inscripție foarte interesantă, formată din simboluri ale lunii şi ale soarelui. Inscripția apare sub forma unei matrice pătratice cu N linii şi N coloane.

Din nefericire, trecerea timpului a dus la deteriorarea peretelui şi acum numai ultima coloană mai este vizibilă. Până la urmă, arheologii au reuşit să descopere o regulă interesantă, chiar dacă simbolurile nu mai sunt integral vizibile.

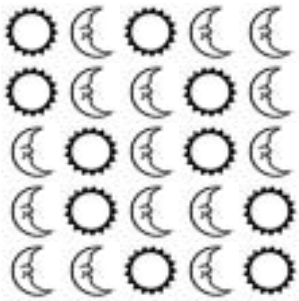
Se consideră simbolurile lunii şi ale soarelui de pe o linie care sunt dispuse într-o ordine aparent aleatoare. Totuşi, în continuare se observă o regulă foarte clară. O altă linie a matricei se obține din prima, efectuând o permutare cu o poziție, adică primul element ajunge pe ultima coloană, toate celelalte sunt mutate la stânga cu o poziție. O altă linie se obține permutând din nou cu o poziție şi aşa mai departe, până la obținerea tuturor rândurilor.

De exemplu, în situația ipotetică în care un rând ar fi 

atunci cele cinci rânduri ale inscripției sunt:



Este evident că toate aceste rânduri apar pe inscripție, dar ele nu se află în această ordine. S-a observat că există o regulă de ordonare lexicografică a acestora. Singura regulă care stă la baza acestei ordonări este aceea că soarele se află, din punct de vedere lexicografic, înaintea lunii. Aşadar, fiecare rând poate fi privit ca un cuvânt şi aceste cuvinte sunt ordonate folosind această regulă. În acest caz, pentru linia considerată, configurația inscripției ar fi:



Folosind regulile prezentate, va trebui să determinați prima linie a inscripției (celelalte sunt uşor de aflat dacă este cunoscută aceasta), având în vedere faptul că aveți la dispoziție doar simbolurile de pe ultima coloană.

Date de intrare
Prima linie a fişierului de intrare **TEMPLU.IN** va conține un număr întreg N, care va indica dimensiunea inscripției. Următoarea linie va conține N numere întregi, separate prin câte un spațiu, care vor indica simbolurile de pe ultima coloană a inscripției. Pentru a simboliza luna se va folosi valoarea 1, iar pentru a simboliza soarele se va folosi valoarea 0.

Date de ieşire
Fişierul de ieşire **TEMPLU.OUT** va conține o singură linie pe care se vor afla N numere întregi, separate prin câte un spațiu, care vor reprezenta simbolurile de pe prima linie a inscripției. Se va folosi valoarea 0 pentru soare şi valoarea 1 pentru lună.

Restricții şi precizări
- $1 \leq N \leq 50000$;
- există întotdeauna soluție.

Exemplu

TEMPLU.IN	**TEMPLU.OUT**
5 1 1 1 0 0	0 1 0 1 1

Timp de execuție: 1 secundă/test

2.2.4. Cub

Se consideră un cub de dimensiune N. Acesta poate fi împărțit în N^3 cubulețe cu latura egală cu unitatea. În fiecare dintre aceste cubulețe se află câte o literă mică a alfabetului latin.

Sarcina voastră este de a determina numărul de apariții ale unui cuvânt S de lungime K, format din litere mici distincte. Cuvântul apare în cub dacă prima sa literă apare într-un anumit cubuleț, cea de-a doua literă apare în unul dintre cele cel mult 26 de cubulețe învecinate, a treia literă apare în unul dintre cele cel mult 26 de cubulețe vecine cu cubulețul în care se află a doua literă şi aşa mai departe.

Pentru un cubuleț de coordonate (x, y, z), cele 26 de cubulețe învecinate au coordonatele $(x + d_x, y + d_y, z + d_z)$, unde d_x, d_y şi d_z fac parte din mulțimea $\{-1, 0, 1\}$ şi $|d_x| + |d_y| + |d_z| > 0$.

Aparițiile cuvântului se pot suprapune parțial, în sensul că un anumit cubuleț poate apărea în două sau mai multe apariții ale cuvântului dat.

Date de intrare

Pe prima linie a fişierului de intrare `CUB.IN` se află cuvântul S. Pe a doua linie se află un număr întreg N care reprezintă dimensiunea cubului.

Pe următoarele N^2 linii se află câte N litere ale alfabetului latin neseparate prin spații.

Considerăm că cele N^3 cubulețe sunt identificate prin coordonatele (x, y, z), unde x, y şi z variază între 1 şi N. Pe primele N linii (dintre cele N^2) se vor afla literele corespunzătoare cubulețelor pentru care $z = 1$. Pe prima dintre acestea se vor afla literele corespunzătoare cubulețelor pentru care $y = 1$. Prima dintre aceste litere va corespunde cubulețului pentru care $x = 1$, a doua va corespunde cubulețului pentru care $x = 2$ şi aşa mai departe. Pe a doua dintre cele N linii se vor afla literele corespunzătoare cubulețelor pentru care $y = 2$ şi aşa mai departe. Pe următoarele N linii se vor afla literele corespunzătoare cubulețelor pentru care $z = 2$ şi aşa mai departe.

Date de ieşire

Fişierul de ieşire `CUB.OUT` va conține o singură linie pe care se va afla un singur număr, reprezentând numărul de apariții ale cuvântului S în cub.

Restricții şi precizări

- $1 \le N < 40$;
- $1 \le K \le 26$;
- numărul soluțiilor nu depăşeşte 2000000000.

Exemplu

CUB.IN	CUB.OUT
ab 2 aa aa bb bc	12

Timp de execuţie: 1 secundă/test

2.2.5. Arbori

În curtea unei case se află *M* "arbori binari". Într-o zi, proprietarul a primit de la un prieten o fotografie cu *N* arbori binari. Poza i-a plăcut atât de mult, încât a decis să transforme *N* dintre arborii din grădină în cei *N* arbori din poză, dacă este posibil.

Dacă nu, el doreşte să transforme un număr *K* dintre arborii din grădină în *K* arbori (cu numere de ordine diferite) din poză, maximizând valoarea *K*.

Proprietarul poate tăia crengi ale arborilor din grădină. În final, doi arbori (unul din gradină, unul din poză) sunt identici dacă au aceeaşi configuraţie a crengilor (muchiilor); numerele care identifică nodurile sunt ignorate. Mai multe detalii reies din studierea exemplului.

Trebuie să scrieţi un program pentru a-l ajuta pe proprietar să minimizeze numărul crengilor tăiate, în condiţiile în care numărul *K*, definit anterior, este maxim.

Date de intrare

Pe prima linie a fişierului de intrare **ARBORI.IN** se află valorile *M* şi *N*, separate printr-un singur spaţiu.

În continuare sunt descrişi cei *M* arbori din curte. Pe prima linie corespunzătoare unui arbore se află numărul *X* al nodurilor arborelui. Fiecare dintre următoarele linii descriu fiii fiecărui nod din arbore. Prima linie corespunde primului nod (care este întotdeauna rădăcina arborelui), a doua celui de-al doilea nod etc. Fiecare linie conţine două numere cuprinse între 0 şi *X* şi indică numărul de ordine al celor doi fii ai nodului corespunzător liniei. Valoarea 0 indică faptul că nodul respectiv nu există.

Urmează descrierile celor *N* arbori din poză. Aceştia sunt descrişi în acelaşi mod ca şi cei *M* arbori din curte.

Datele de intrare sunt corecte; se garantează că fiecare descriere corespunde unui arbore binar.

Date de ieşire

Pe prima linie a fişierului de ieşire **ARBORI.OUT** se va afla numărul *K* al arborilor care pot fi obţinuţi. Pe a doua linie se află numărul *MIN* al crengilor tăiate.

Pe fiecare dintre următoarele *K* linii se va afla o pereche de numere: *număr_arbore_curte*, *număr_arbore_poză* care indică faptul că arborele *număr_arbore_poză* se obține tăind crengi din arborele *număr_arbore_curte*. Numerele de ordine ale arborilor din curte care apar în fişierul de ieşire trebuie să fie distincte. Similar, numerele de ordine ale arborilor din poză care apar în fişierul de ieşire trebuie să fie distincte. Nu contează ordinea în care sunt scrise perechile în fişierul de ieşire, iar dacă există mai multe soluții se va afişa una singură.

Restricții şi precizări

- $1 \le M, N \le 40$;
- doi arbori nu se consideră identici decât dacă subarborii lor drepți sunt identici şi subarborii lor stângi sunt identici; dacă cei doi arbori sunt simetrici (subarborele stâng al unuia este identic cu subarborele drept al celuilalt şi invers), atunci ei nu sunt considerați identici;
- fiecare arbore are cel mult 100 de noduri.

Exemplu

ARBORI.IN	**ARBORI.OUT**
3 3 2 2 0 0 0 3 2 3 0 0 0 0 5 5 3 0 0 0 0 0 0 4 2 3 3 2 0 0 0 0 2 0 2 0 0 3 2 0 0 3 0 0	2 1 2 1 3 2

Timp de execuție: 1 secundă/test

2.2.6. Atlantis

În Anul Domnului Două Mii Patru Sute Cincizeci şi şapte au fost, în sfârşit, descoperite vechile ruine ale civilizaţiei *Atlantidei*.

Expediţia a fost condusă de marele explorator al oceanelor *Iahim Uratrox*. Acesta a ajuns în faţa porţii miticului oraş *Atlantis*, dar nu a reuşit să o deschidă deoarece este protejată de un vechi cifru.

Pe poartă sunt inscripţionate zece simboluri distincte. Oamenii de ştiinţă au descoperit că acestea reprezintă cele 10 cifre zecimale, prima dintre ele corespunde cifrei 0, a doua corespunde cifre 1 şi aşa mai departe.

Pe mecanismul cifrului sunt reprezentate patru dintre cele zece simboluri. Există posibilitatea ca două sau mai multe dintre cele patru simboluri să fie identice. Mecanismul cifrului constă din *N* dispozitive care sunt activate prin simpla atingere. Pe fiecare dintre aceste *N* dispozitive este gravată o inscripţie corespunzătoare unei cifre.

Oamenii de ştiinţă au mai descoperit că cifrul este dat de cel mai mic multiplu strict pozitiv al numărului format din cele patru simboluri desenate pe mecanism. Datorită faptului că pentru formarea multiplului nu sunt disponibile decât cifrele corespunzătoare celor *N* dispozitive, doar acestea pot fi folosite pentru formarea multiplului.

Operaţia de deschidere a porţii este acum foarte simplă. Trebuie activate, succesiv, dispozitivele corespunzătoare cifrelor care formează multiplul. După activarea unui dispozitiv se aprinde o lumină care indică faptul că cifra a fost acceptată, iar apoi lumina se stinge pentru a permite repetarea cifrei.

Sarcina voastră este de a-l ajuta pe conducătorul expediţiei să găsească cifrul. Veţi avea la dispoziţie cele patru cifre ale numărului de la care se porneşte şi cifrele pe care le aveţi la dispoziţie pentru a determina multiplul.

Date de intrare

Pe prima linie a fişierului de intrare **`ATLANTIS.IN`** se află cele patru cifre ale numărului gravat pe mecanismul cifrului, neseparate prin spaţii. Există posibilitatea ca prima, primele două sau primele trei cifre să fie 0. Totuşi, cel puţin una dintre cele patru cifre este diferită de zero.

Pe a doua linie a fişierului se află numărul *N* al cifrelor pe care le aveţi la dispoziţie pentru a forma multiplul.

Următoarele *N* linii conţin, fiecare, câte o cifră pe care o aveţi la dispoziţie.

Date de ieşire

Fişierul de ieşire **`ATLANTIS.OUT`** va conţine o singură linie pe care se vor afla cifrele multiplului neseparate prin spaţii.

În cazul în care un astfel de multiplu nu poate fi determinat, *Iahim Uratrox* va folosi o încărcătură explozivă pentru a deschide poarta, iar dumneavostră veţi scrie în fişierul de ieşire valoarea 0.

Restricții și precizări

- $1 \le N \le 10$;
- cele N cifre pe care le aveți la dispoziție sunt distincte;
- valoarea numărului inscripționat pe mecanismul cifrului este cuprinsă între 1 şi 9999;
- în cazul în care aveți la dispoziție cifra 0, aceasta nu poate fi prima cifră a multiplului pe care îl veți afișa.

Exemplu

ATLANTIS.IN	**ATLANTIS.OUT**
0012 5 5 4 1 7 9	144

Timp de execuție: 1 secundă/test

2.2.7. Celule

Un număr de n celule sensibile sunt aşezate în cerc, fiecare comunicând cu cele două celule vecine. O celulă se poate găsi în două stări: excitată sau liniştită. Dacă o celulă este excitată la un moment dat t, atunci ea emite un semnal care ajunge la cele două celule vecine la momentul $t + 1$. Fiecare celulă este excitată atunci şi numai atunci, când la ea ajunge un semnal de la una din celulele vecine. Dacă semnalele ajung deodată din ambele părți, atunci ele se anulează.

Dându-se o configurație inițială de celule excitate şi liniştite, se cere să se determine dacă excitația se menține oricât de mult, sau se va linişti.

Date de intrare

Prima linie a fişierului de intrare **CELULE.IN** conține un număr natural m, reprezentând numărul configurațiilor din fişier.

Fiecare din configurații se precizează pe două linii: pe prima linie se află un număr natural n, reprezentând numărul celulelor din configurație, iar pe a doua se află configurația propriu-zisă.

O configurație este reprezentată printr-o succesiune de cifre (nedespărțite prin spații) din mulțimea {0, 1}, unde 1 indică faptul că celula este excitată, iar 0 indică faptul că celula este liniştită.

Date de ieşire
Fişierul de ieşire **CELULE.OUT** va conține câte o linie pentru fiecare configurație descrisă în fişierul de intrare. În cazul în care excitația se menține oricât de mult, pe linia corespunzătoare se va scrie şirul de caractere DA. În caz contrar, linia va conține şirul de caractere NU.

Restricții şi precizări
- $1 \le m \le 30$;
- $1 \le n \le 10000$;
- nu se va acorda punctajul corespunzător unui test decât dacă toate cele m răspunsuri sunt corecte; nu se acordă punctaje parțiale.

Exemplu

CELULE.IN	CELULE.OUT
3 12 111111111111 3 110 6 101010	NU DA NU

Timp de execuție: 1 secundă/test

2.2.8. Fibonacci
Se consideră celebrul şir al lui Fibonacci, definit astfel:
- $F_0 = 0$;
- $F_1 = 1$;
- $F_i = F_{i-1} + F_{i-2}$ pentru $i > 1$.

Sarcina voastră este de a determina indicele unui element al acestui şir astfel încât elementul să fie divizibil cu un număr de forma 2^n.

Valoarea indicelui trebuie să fie mai mică decât 2^n. De exemplu, pentru $n = 3$, unul dintre elementele divizibile cu 2^3 este 8 (are indicele 6 şi $6 < 8$).

Date de intrare
Fişierul de intrare **FIBO.IN** conține o singură linie pe care se află un singur număr care reprezintă valoarea n.

Date de ieşire
Fişierul de ieşire **FIBO.OUT** va conține o singură linie pe care se va afla indicele elementului divizibil cu 2^n.

În cazul în care nu există nici un element divizibil cu 2^n a cărui indice să aibă o valoare mai mică decât 2^n, atunci în fişierul de ieşire se va scrie valoarea -1.

Restricție
- $1 \le n \le 10000$.

Exemple

FIBO.IN	**FIBO.OUT**
3	6

FIBO.IN	**FIBO.OUT**
1	-1

FIBO.IN	**FIBO.OUT**
5	24

FIBO.IN	**FIBO.OUT**
2	-1

Timp de execuție: 1 secundă/test

2.2.9. Atlantis2

După ce a reuşit să treacă de poarta cetății *Atlantis*, *Iahim Uratrox*, marele explorator al *Atlantidei*, a descoperit o mulțime dintre vestigiile vechii civilizații.

Singura clădire în care nu a putut intra este palatul foştilor suverani atlanți. Se presupune că în interiorul acestui palat se pot găsi documentele care să explice motivul scufundării miticului oraş. Din nefericire, intrarea în palat este protejată şi ea de un cifru.

Mecanismul este unul mult mai simplu, el constând din doar două dispozitive de introducere a codului. Pe mecanism este gravat, la fel ca şi la intrare, un număr format din şase cifre.

Bineînțeles, savanții au reuşit imediat să descopere algoritmul de funcționare al mecanismului. Cele două dispozitive corespund semnelor '+' şi '-'. Fiecare atingere a unui dispozitiv duce la adunarea sau scăderea unui număr. Acest număr variază în funcție de numărul de ordine al atingerii. La a i-a atingere, numărul care se adună sau se scade este i^2. Se porneşte cu valoarea 0.

Aşadar, la fiecare atingere i valoarea corespunzătoare mecanismului scade sau creşte cu i^2. În final, pentru a deschide poarta, trebuie să se obțină valoarea gravată pe mecanism.

Date de intrare
Fişierul de intrare **ATLANTIS.IN** conţine o singură linie pe care se află cele şase cifre ale numărului gravat pe mecanismul cifrului, neseparate prin spaţii. Există posibilitatea ca prima, primele două, primele trei, primele patru sau primele cinci cifre să fie 0. Totuşi, cel puţin una dintre cele şase cifre este diferită de zero.

Date de ieşire
Fişierul de ieşire **ATLANTIS.OUT** va conţine o singură linie pe care se vor afla semnele corespunzătoare atingerilor mecanismului. Al *i*-lea semn de pe această linie va corespunde celei de-a *i*-a atingeri. Dacă prin această atingere se adună valoarea i^2, atunci semnul va fi '+'. Dacă atingerea duce la scăderea valorii i^2, atunci semnul va fi '–'.

Restricţii şi precizări
- Valoarea numărului inscripţionat pe mecanismul cifrului este cuprinsă între 1 şi 999999.
- În cazul în care nu poate fi descoperită o secvenţă de atingeri, *Iahim Uratrox* nu va mai putea deschide poarta folosind o încărcătură explozivă aşa cum ar fi putut face atunci când se afla în faţa porţii oraşului, deoarece ar risca să distrugă prea multe vestigii importante; astfel, cel mai mare secret al Atlantidei va fi păstrat pentru totdeauna. Din fericire, după câteva luni, exploratorul s-a întors şi a prezentat motivele scufundării *Continentului Pierdut*; se pare că explicaţia era atât de îngrijorătoare încât s-a decis ca motivele să nu fie cunoscute publicului. Totuşi, concluzia evidentă care poate fi trasă este că există întotdeauna o secvenţă corectă de atingeri care duc la deschiderea porţii.
- Poarta se deschide chiar dacă numărul atingerilor nu este minim.
- Dacă există mai multe soluţii, doar una trebuie scrisă în fişierul de ieşire.

Exemple

ATLANTIS.IN	**ATLANTIS.OUT**
000012	++-+

ATLANTIS.IN	**ATLANTIS.OUT**
000055	+++++

Timp de execuţie: 5 secunde/test

2.2.10. Conducte
Într-o ţară există *k* rafinării, unde din ţiţei se prepară benzină şi *n* benzinării, unde trebuie să ajungă prin conducte benzina preparată de rafinării. Se pune problema amplasării conductelor care leagă rafinăriile de benzinării, astfel încât lungimea totală a conductelor să fie minimă.

Stabiliţi care sunt rafinăriile şi benzinăriile între care se vor amplasa conductele, astfel încât lungimea lor totală să fie minimă.

Date de intrare
Pe prima linie a fişierului de intrare **CONDUCTE.IN** se află două numere naturale k şi n, reprezentând numărul rafinăriilor, respectiv numărul benzinăriilor, separate printr-un spaţiu. Următoarele k linii conţin câte două numere întregi, separate printr-un spaţiu, reprezentând coordonatele rafinăriilor; următoarele n linii conţin, de asemenea, câte două numere întregi, separate printr-un spaţiu, reprezentând coordonatele benzinăriilor. Coordonatele corespund unor indici de linii, respectiv de coloană într-un caroiaj imaginar care conţine punctele corespunzătoare rafinăriilor şi benzinăriilor.

Date de ieşire
În fişierul de ieşire **CONDUCTE.OUT** se vor scrie acele locuri care se vor lega prin conductele reţelei, având lungimea totală minimă. În fiecare linie se vor scrie două numere întregi, separate printr-un spaţiu, reprezentând numerele de ordine a două puncte legate. Indicele rafinăriei se va preceda de litera 'R', iar indicele benzinăriei de litera 'B'.

Restricţii şi precizări
- $1 \le k \le 10$;
- $1 \le n \le 1000$;
- coordonatele rafinăriilor şi benzinăriilor sunt numere întregi din intervalul [-100, 100];
- conductele se pot ramifica doar în dreptul benzinăriilor sau în dreptul rafinăriilor;
- este posibil ca două benzinării să fie legate prin conducte (altfel spus, nu este obligatoriu ca într-o benzinărie benzina să sosească direct dintr-o rafinărie).

Exemplu

CONDUCTE.IN	**CONDUCTE.OUT**
2 4	B2 B4
100 100	R2 B3
0 100	B2 B3
-100 100	R2 B1
100 0	
40 40	
100 -50	

Timp de execuţie: 1 secundă/test

2.2.11. Dune2

Pe *Câmpia Arakeenului* are loc bătălia finală pentru controlul planetei *Dune*. Câmpia este reprezentată printr-o matrice având şapte coloane şi N linii.

În fiecare zonă (celulă a matricei) se află câte un contingent de trupe ale *atreizilor* şi ale *harkonnienilor*. Valoarea corespunzătoare unei zone este dată de diferența dintre numărul soldaților celor două case. Valoarea este pozitivă dacă *harkonnienii* sunt mai numeroşi, negativă dacă sunt mai multe trupe *Atreides* şi 0 dacă cele două contingente sunt egale.

Paul Muad'Dib are la dispoziție k bombe nucleare (*Arde-Piatră*) pe care le-a achiziționat la un preț foarte mare de la *ixieni*. Fiecare bombă poate fi lansată asupra unei secțiuni pătratice având latura 2. Aşadar, patru zone vor fi afectate de un *Arde-Piatră*. Evident, toți soldații aflați în acele zone vor fi nimiciți.

Bombele trebuie lansate complet în interiorul câmpiei. *Paul* doreşte ca pagubele provocate de cele k bombe să fie maxime (diferența dintre numărul *harkonnenilor* şi cel al *atreizilor* din zonele afectate să fie cât mai mare). Pentru aceasta el poate folosi nici una, una, mai multe sau toate bombele pe care le-a achiziționat.

Date de intrare

Prima linie a fişierului **DUNE.IN** va conține două numere întregi, separate printr-un spațiu. Primul va indica numărul N al liniilor matricei, iar al doilea va indica numărul K al bombelor pe care le are la dispoziție *Paul*. Fiecare dintre următoarele N linii va conține câte şapte numere întregi separate prin câte un spațiu, reprezentând cele $7 \cdot N$ elemente ale matricei. Pe prima dintre aceste linii se vor afla elementele corespunzătoare primei linii a matricei, pe cea de-a doua linie cele corespunzătoare celei de-a doua linii etc. Primul element dintr-o linie va corespunde elementului de pe prima coloană a liniei respective, al doilea va corespunde elementului de pe a doua coloană etc.

Date de ieşire

În fişierul de ieşire **DUNE.OUT** se va scrie un număr întreg care va indica diferența maximă dintre numărul *harkonnienilor* şi numărul *atreizilor* afectați de bombe.

Restricții

- $2 \le n \le 100$;
- $1 \le k \le 100$.

Exemplu

DUNE.IN	DUNE.OUT
3 4 1 1 0 0 0 0 0 1 1 1 0 0 -3 0 -4 1 1 0 0 1 1	7

Timp de execuție: 1 secundă/test

2.2.12. Frăția Inelului

După ce a primit *Inelul Puterii* de la *Bilbo*, *Frodo* a fost nevoit să fugă din *Comitat* deoarece era urmărit de duhurile *Inelului*. A reuşit să ajungă în *Vâlceaua Despicată*, loc în care s-a ținut marele sfat din casa lui *Elrond*.

Aici s-a hotărât ca inelul să fie distrus. Pentru aceasta el trebuia transportat la *Muntele Osândei*, singurul loc în care acesta putea fi distrus. *Frodo* a fost ales ca *Purtător al Inelului* şi s-a decis ca opt reprezentanți ai semințiilor puse în pericol de *Umbra* din *Mordor* să îl însoțească. Astfel s-a format *Frăția Inelului*.

Din păcate au izbucnit certuri cu privire la traseul care urma să fie parcurs. Pentru a rezolva această problemă, *Elrond* şi-a dat seama că, mai întâi, trebuie cunoscut numărul traseelor posibile. Sarcina calculării acestui număr i-a revenit lui *Sam cel Înțelept*. El a primit cea mai nouă hartă a *Pământului de Mijloc* pe care erau marcate cele mai importante puncte de reper, precum şi traseele care le unesc.

Drumurile care unesc două repere sunt cu sens unic şi nu există posibilitatea ca, dacă s-a ajuns la un anumit punct de reper, să existe o cale spre un punct de reper anterior. Şi acum, va trebui să vă închipuiți că sunteți *Sam*, şi că de dumneavoastră depinde succesul expediției.

Date de intrare

Fişierul de intrare **`SAM.IN`** conține, pe prima linie, numărul *M* al drumurilor care leagă două repere. Pe fiecare dintre următoarele *M* linii se vor afla câte două cuvinte (formate doar din litere mari ale alfabetului latin), care vor reprezenta reperele unite de un drum (sensul este de la primul la al doilea reper).

Date de ieşire

Fişierul de ieşire **`SAM.OUT`** va conține o singură linie pe care se va afla numărul traseelor.

Restricții şi precizări

- există cel mult 1000 de drumuri şi 100 de repere;
- numărul total al traseelor este întotdeauna cuprins între 1 şi 2000000000;
- se pleacă întotdeauna din `VALCEAUADESPICATA`;
- se ajunge întotdeauna la `MUNTELEOSANDEI`.

Exemplu

`SAM.IN`

```
9
ROHAN GONDOR
VALCEAUADESPICATA COMITAT
VALCEAUADESPICATA ROHAN
VALCEAUADESPICATA MORIA
```

`SAM.OUT`

```
5
```

```
MORIA ROHAN
MORIA GONDOR
MORIA MORDOR
GONDOR MUNTELEOSANDEI
ROHAN MUNTELEOSANDEI
```

Timp de execuție: 1 secundă/test

2.2.13. Litere

Se consideră un şir de litere mari ale alfabetului latin. Acest şir se află pe prima linie a unei matrice pătratice *A*, având dimensiunea egală cu lungimea şirului. Următoarea linie a matricei *A* se obține prin permutarea la stânga cu o poziție a elementelor de pe prima linie. Aceeaşi operație se aplică şi pentru următoarele linii: linia *i* se obține prin permutarea la stânga cu o poziție a elementelor de pe linia *i* – 1.

În continuare se generează o matrice *B* care conține liniile matricei *A* ordonate lexicografic. De exemplu, dacă şirul de caractere considerat este `GINFO`, atunci matricea *A* va avea următoarea structură:

```
GINFO
INFOG
NFOGI
FOGIN
OGINF
```

Ordonând lexicografic liniile matricei *A*, se obține matricea *B*:

```
FOGIN
GINFO
INFOG
NFOGI
OGINF
```

Se cunosc caracterele de pe ultima coloană a matricei *B* şi se cere determinarea caracterelor de pe prima linie a acesteia.

Date de intrare

Fişierul de intrare **`LITERE.IN`** conține o singură linie pe care se află caracterele de pe ultima coloană a matricei *B*.

Date de ieşire

Fişierul de ieşire **`LITERE.OUT`** va conține o singură linie pe care se vor afla caracterele de pe prima linie a matricei *B*.

Restricții şi precizări

- şirul inițial conține cel mult 10000 de caractere;
- pe baza ultimei coloane se va putea determina întotdeauna prima linie.

Exemplu

LITERE.IN	LITERE.OUT
NOGIF	FOGIN

Timp de execuție: 1 secundă/test

2.2.14. Zăvor

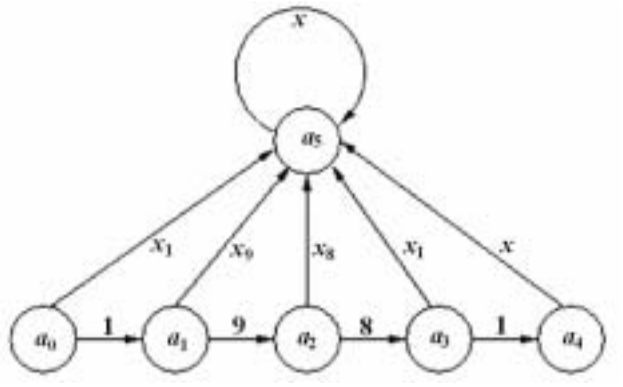
O închizătoare cu cifru este un instrument cu zece butoane pentru tastarea a câte unei cifre şi un buton de reset care foloseşte la aducerea închizătorii în starea inițială.

Închizătoarea mai dispune de un buton cu care aceasta poate fi deschisă, cu condiția să fi tastat în prealabil parola corectă.

În închizătoare se încorporează un cip pentru o parolă dată, care funcționează ca un automat finit (rezultă că dacă parola este formată din N cifre, atunci automatul are exact $N + 2$ stări).

Dacă automatul se află într-o stare a şi recepționează un semnal x (s-a apăsat butonul corespunzător cifrei x), atunci ajunge într-o stare nouă b, determinată în mod univoc de a şi x.

De exemplu, dacă parola este 1981, atunci un automat care funcționează corect, trebuie să-şi schimbe stările conform figurii următoare. De asemenea, automatul are voie să deschidă închizătoarea doar în starea a_4. Pe figură x_i reprezintă toate cifrele diferite de i, iar x reprezintă toate cifrele.



Verificați dacă cipul dat funcționează corect sau nu. Se va verifica doar faptul că pe baza semnalelor se va deschide sau nu închizătoarea; dacă se recepționează parola corectă, închizătoarea se deschide întotdeauna.

Pentru această testare aveți la dispoziție trei operații; pentru programatorii în *Pascal*, acestea sunt conținute în *unit*-ul `ZAVOR.TPU`:

- **procedure** ResetA: automatul este adus în starea de bază;
- **procedure** Schimba(x:Char): automatul trece din starea curentă în starea corespunzătoare semnalului *x* ('0' ≤ *x* ≤ '9');
- **function** Deschide:Char: dacă starea curentă este cea de deschidere, atunci valoarea returnată este 'D', altfel se returnează valoarea 'N'.

Programatorii în *C/C++* vor avea la dispoziție fișierul *header* ZAVOR.H care va conține funcțiile:

- **void** ResetA(**void**): automatul este adus în starea de bază;
- **void** Schimba(**char** x): automatul trece din starea curentă în starea corespunzătoare semnalului *x* ('0' ≤ x ≤ '9');
- **char** Deschide(**void**): dacă starea curentă este cea de deschidere, atunci valoarea returnată este 'D', altfel se returnează valoarea 'N'.

Date de intrare
Singura linie a fișierului de intrare **TEST.IN** conține parola care este formată din cel mult 100 cifre zecimale (caractere cuprinse între '0' și '9') nedespărțite prin spații. Ultimul caracter este marcajul de sfârșit de linie.

Date de ieșire
În fișierul de ieșire **TEST.OUT** se va scrie rezultatul pe o singură linie.

Dacă cifrul se deschide doar pentru parola dată în fișierul de intrare, atunci în fișier se scrie cuvântul CORECT. Dacă cifrul se deschide și pentru alte chei decât parola dată, în fișierul de ieșire se va scrie un șir de caractere care deschid cifrul.

Timp de execuție: 1 secundă/test

2.3. Soluțiile problemelor propuse în ediția 2000-2001

2.3.1. Triangularizare

Chiar dacă problema nu este foarte cunoscută în această formă, ea se reduce la a determina numărul de șiruri care pot avea $2 \cdot n$ elemente și care satisfac criteriile:

- Exact *n* elemente au valoarea 1 și exact *n* elemente au valoarea 0.
- Orice prefix al acestor șiruri conține mai multe elemente care au valoarea 1 decât elemente care au valoarea 0; prin prefix înțelegem un subșir care conține primele elemente ale unui șir.

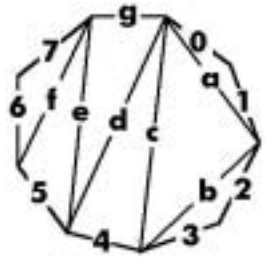
O altă variantă echivalentă a problemei este următoarea: *Să se determine numărul șirurilor de n perechi de paranteze care se deschid și se închid corect.*

Aceasta este cea mai cunoscută formă; există şi alte enunţuri echivalente ale aceleiaşi probleme.

Vom demonstra acum echivalenţa dintre o triangularizarea unui poligon cu n laturi şi determinarea unui şir de n - 2 perechi de paranteze care se deschid şi se închid corect.

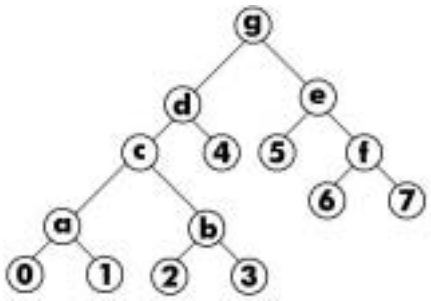
Dându-se o triangularizare oarecare a poligonului, considerăm una dintre laturi ca fiind baza triangulaţiei. Celelalte laturi sunt etichetate cu numere cuprinse între 0 şi n – 2. Construim şirul de paranteze aşa cum se descrie în continuare.

Baza şi diagonalele pot fi etichetate dacă fac parte dintr-un triunghi care are celelalte două laturi etichetate. În acest caz, eticheta ei va conţine o paranteză dreaptă deschisă, eticheta primei laturi, eticheta celei de-a doua laturi şi o paranteză dreaptă închisă. Să considerăm următoarea triangulaţie a unui poligon cu 9 laturi:



La primul pas vom eticheta diagonala a cu [01], diagonala b cu [23] şi diagonala f cu [67]. La al doilea pas diagonala c este etichetată cu [[01][23]], iar diagonala e este etichetată cu [5[67]]. La al treilea pas etichetăm diagonala d cu [[01][23]4], iar la ultimul, baza este etichetată cu [[[01][23]4][5[67]]].

Această etichetare poate fi considerată ca fiind descrierea unui arbore binar în care numerele reprezintă frunzele, iar parantezele descriu modul în care sunt realizate legăturile. Astfel, fiecărei triangularizări îi va corespunde un arbore binar unic. Pentru triangularizarea anterioară avem următorul arbore:



Vom eticheta cu 0 muchiile spre fiii drepţi şi cu 1 muchiile spre fiii stângi. Efectuând apoi o parcurgere a arborelui, obţinem un şir de numere 1 şi 0, care satisface cele

două criterii amintite anterior. Înlocuind cifrele 1 cu o paranteză deschisă şi cifrele 0 cu o paranteză închisă, obținem o parantezare. Astfel am demonstrat că fiecărei triangularizări a unui poligon cu n laturi îi corespunde o parantezare cu $n - 2$ perechi de paranteze.

Se ştie că numărul de parantezări cu n perechi de paranteze este numărul lui *Catalan* care are valoarea:

$$\frac{C_{2n}^{n}}{n+1} = \frac{2n!}{n!\cdot(n+1)!}.$$

Deoarece numărul triangularizărilor unui poligon cu n laturi corespunde unei parantezări cu n - 2 perechi de paranteze, formula pe care o vom folosi este:

$$\frac{C_{2n-4}^{n-2}}{n-1} = \frac{(2n-4)!}{(n-2)!\cdot(n-1)!}$$

Datorită faptului că rezultatul poate avea aproape 1500 cifre, trebuie implementate operații cu numere mari.

Vom construi un vector care va conține valorile $n - 1$, n, $n + 1$, ..., $2 \cdot n - 4$. Produsul P al acestor numere reprezintă valoarea:

$$\frac{(2n-4)!}{(n-2)!}$$

Pentru a obține rezultatul final va trebui să împărțim acest produs la $(n - 1)!$. Pentru fiecare număr cuprins între 2 şi $n - 1$ vom încerca să împărțim produsul P la numărul respectiv. Să considerăm un număr k. Pentru a împărți P la k vom efectua următoarele operații:

```
cât timp k > 1 execută
  d ← cmmdc(k, a_i)
  a_i ← a_i / d
  k ← k / d
  i ← i + 1
sfârşit cât timp
```

După efectuarea tuturor împărțirilor, rezultatul final va fi obținut înmulțind elementele vectorului a. Pentru a micşora timpul de execuție vom lucra în baza 10000. De asemenea, nu vom efectua operații cu numere mari atâta timp cât putem realiza înmulțiri parțiale, astfel încât rezultatul să nu depăşească valoarea 10000. Dacă printr-o astfel de înmulțire parțială s-ar depăşi valoarea 10000, atunci se efectuează o înmulțire folosind numerele mari.

Analiza complexității

Citirea datelor de intrare se realizează în timp constant, deoarece acestea constau într-un singur număr; aşada,r ordinul de complexitate al acestei operații este $O(1)$.

Pentru a determina şirul valorilor care vor fi înmulțite pentru a obține rezultatul, va trebui, mai întâi să creăm şirul care conține valorile $n - 1$, n, $n + 1$, ..., $2 \cdot n - 4$, operație al cărei ordin de complexitate este $O(n)$.

După crearea acestui şir, vom lua în considerare toate numerele cuprinse între 2 şi $n - 1$. Pentru realizarea împărțirii va trebui, în cel mai defavorabil caz, să parcurgem întreg şirul. Pentru fiecare element al şirului vom determina un cel mai mare divizor comun, operație al cărei ordin de complexitate este $O(\log n)$. Aşadar, pentru a realiza o împărțire avem nevoie de un timp de ordinul $O(n \cdot \log n)$, iar pentru realizarea tuturor împărțirilor ordinul de complexitate este $O(n^2 \cdot \log n)$.

Datorită faptului că numărul de cifre (în baza 10000) al rezultatului este limitat, putem considera că o înmulțire se realizează în timp constant. Numărul total al înmulțirilor are ordinul $O(n)$, deci operația de determinare a rezultatului final va avea ordinul de complexitate $O(n)$.

Afişarea numărului de triangularizări este realizată în timp constant, deoarece se afişează un singur număr.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(n) + O(n^2 \cdot \log n) + O(n) + O(1) = O(n^2 \cdot \log n)$.

Observație

Ordinul de complexitate ar fi fost $O(n)$ dacă nu am fi folosit artificiul descris pentru calcularea rezultatului. Totuşi, ar fi fost efectuate foarte multe operații cu numere mari şi, deşi am considerat constant timpul de execuție al unei astfel de operații, această constantă este, de fapt, foarte mare. Ca urmare, timpul real de execuție ar fi fost mult mai mare dacă am fi realizat înmulțiri şi împărțiri cu numere mari.

2.3.2. Divizori

Pentru rezolvarea acestei probleme vom aplica metoda *programării dinamice*. Vom începe cu demonstrarea unui adevăr matematic: dacă n este un număr natural şi a un şir care conține puterile la care apar factorii primi din descompunerea lui n, atunci numărul divizorilor lui n este:

$$\prod_{i=0}^{\infty} (a_i + 1)$$

Demonstrația acestei teoreme este foarte simplă. Numărului n i se asociază şirul a_i al puterilor la care apar factorii primi din descompunerea sa. Astfel, fiecărui divizor îi va corespunde un element al produsului cartezian $A_1 \times A_2 \times \ ... \times A_n$, unde $A_i = \{0, 1, ..., a_i\}$, iar numărul elementelor acestui produs cartezian este:

$$\prod_{i=0}^{\infty}(a_i+1)$$

De fapt, nu trebuie să luăm în considerare toate numerele prime, fiind suficiente doar cele mai mici.

În continuare vom presupune că am ales primele 15 numere prime. La început determinăm vectorul d al divizorilor numărului k. Considerăm că acest vector are m elemente. Construim o matrice A cu 15 linii şi m coloane, A_{ij} având valoarea egală cu cel mai mic număr care are ca factori primi primele numere prime şi având d_j divizori. Rezultatul cerut va fi valoarea minimă de pe ultima coloană a matricei.

Primul element al fiecărei linii ($A_{i,1}$) va avea valoarea 1, deoarece 1 este cel mai mic număr cu un singur divizor. Vom avea $A_{0,j} = \infty$ deoarece nu există numere cu i divizori şi nici un factor prim.

Elementul $A_{i,j}$ poate fi calculat folosind valorile $A_{i\text{-}1,k}$ pentru $k < j$ şi d_k este divizor al lui d_j. $A_{i\text{-}1,k}$ are d_k divizori, deci o valoare A_{ij} cu d_j divizori poate fi obținută calculând valoarea:

$$A_{i-1,k} \cdot p_i^{d_j / d_k + 1},$$

unde p_i este al i-lea număr prim. Valoarea A_{ij} va fi aleasă ca fiind minimul acestor valori. Dacă această valoare minimă este mai mare decât $A_{i\text{-}1,j}$, atunci A_{ij} va primi valoarea $A_{i\text{-}1,j}$.

Deoarece ar trebui să lucrăm cu numere foarte mari, în program determinăm logaritmii naturali ai valorilor A_{ij}, iar în final calculăm valoarea reală a minimului de pe ultima coloană a matricei folosind operații cu numere mari.

Analiza complexității

Citirea datelor de intrare se realizează în timp constant deoarece acestea constau într-un singur număr; aşadar ordinul de complexitate al acestei operații este $O(1)$.

Operația de determinare a divizorilor numărului n are ordinul de $O(n)$ deoarece vom verifica toate numerele cuprinse între 1 şi n. Există şi variante mai rapide, dar această operație are un timp de execuție nesemnificativ față de restul operațiilor, motiv pentru care putem folosi un algoritm puțin mai lent.

Pentru a determina valorile elementelor matricei A va trebui ca, pentru fiecare element, să luăm în considerare toate numerele aflate deasupra sa, pe aceeaşi coloană. Aşadar, pentru un element ordinul de complexitate al operației va fi $O(n)$, deoarece numărul liniilor matricei este egal cu numărul divizorilor, iar acest număr variază liniar în funcție de n. Deoarece numărul coloanelor matricei este constant, ordinul de complexitate al operației de determinare a elementelor matricei va fi $O(n^2)$.

Va trebui acum să determinăm valoarea minimă de pe ultima coloană, operație al cărei ordin de complexitate este $O(n)$.

Această valoare reprezintă doar logaritmul rezultatului; pentru determinarea valorii reale sunt necesare operații cu numere mari. În timp liniar vom determina divizorii şi puterile acestora şi apoi, dacă presupunem că o operație cu numere mari se realizează în timp constant, tot în timp liniar vom determina valoarea exactă a rezultatului.

Afişarea celui mai mic număr cu n divizori este realizată în timp constant, deoarece se afişează un singur număr.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(n) + O(n^2) + O(n) + O(n) + O(1) = O(n^2)$.

Observație

Se observă imediat că, chiar dacă am fi folosit o metodă mai rapidă pentru determinarea divizorilor unui număr, ordinul de complexitate ar fi fost tot $O(n^2)$.

2.3.3. Premiere

Problema se reduce la determinarea numărului de partiții în k clase a unei mulțimi având n elemente.

Aceste numere poartă denumirea de numerele lui *Stirling* de speța a II-a şi se notează prin $S(n, k)$. Se observă că pentru $k > n$ avem $S(n, 1) = 1$ şi $S(n, k) = 0$.

Considerăm cele $S(n - 1, k - 1)$ partiții în $k - 1$ clase ale unei mulțimi având $n - 1$ elemente. Adăugând partiției o nouă clasă formată din al n-lea element, obținem $S(n - 1, k - 1)$ partiții în k clase ale unei mulțimi având n elemente.

Considerăm acum cele $S(n - 1, k)$ partiții în k clase ale unei mulțimi având n elemente.

Putem adăuga al n-lea la oricare dintre cele k clase, deci vom avea încă $k \cdot S(n - 1, k)$ partiții în k clase ale unei mulțimi având n elemente.

În concluzie, formula de calcul a numerelor lui Stirling de speța a II-a este:

$$S(n, k) = S(n - 1, k - 1) + k \cdot S(n - 1, k).$$

Datorită valorilor mari care se obțin, vor trebui simulate operațiile pe numere mari.

Analiza complexității

Citirea datelor de intrare se realizează în timp constant, deoarece acestea constau în două numere; aşadar ordinul de complexitate al acestei operații este $O(1)$.

În total vom determina $n \cdot k$ numere *Stirling*. Considerând, că o operație cu numere mari se realizează în timp constant, ordinul de complexitate al tuturor acestor operații este $O(n \cdot k)$

Afişarea rezultatului obținut este realizată în timp constant, deoarece se afişează un singur număr.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(n \cdot k) + O(1) = O(n \cdot k)$.

Observație
Datorită faptului că un număr $S(n, k)$ nu depinde decât de valorile $S(n-1, k-1)$ şi $S(n-1, k)$, nu este necesară păstrarea în memorie a tuturor numerelor calculate. Dacă determinăm numerele pentru o anumită valoare n, atunci avem nevoie doar de numerele pentru valoarea $n-1$.

2.3.4. Acadele

Cele N acadele produse zilnic sunt ambalate câte L în B cutii folosindu-se N ambalaje diferite, corespunzând celor N tipuri disponibile. Rezultă ecuația $B \cdot L = N$.

Restricția, potrivit căreia la sfârşitul fiecărui ciclu de producție, pentru fiecare pereche de ambalaje diferite există o singură cutie care conține exact două acadele cu cele două ambalaje, celelalte având ambalaje diferite față de cele două, poate fi reformulată astfel: pentru fiecare tip de ambalaj din cele N (fie el x) şi pentru fiecare orice alt tip y de ambalaj din cele $N-1$ rămase, există o singură cutie cu ambalajele x şi y, produsă în cele D zile de producție; într-o cutie x se află împreună alături de $L-1$ ambalaje diferite, iar x nu mai poate apărea în alte cutii produse în aceeaşi zi. Rezultă că după cele D zile, fiind zilnic alături de $L-1$ ambalaje diferite, x trebuie să acopere cele $N-1$ posibilități de asociere; se obține ecuația $D \cdot (L-1) = (N-1)$.

La cele două ecuații se adaugă inegalitatea $B > 1$, care este prezentă în enunț. Se obține următorul sistem nedeterminat:

$B \cdot L = N$ (*)
$D \cdot (L-1) = (N-1)$ (**)
$B > 1$ (***)

În acest sistem valorile L, B, N şi D sunt numere întregi pozitive. În continuare vom arăta că o soluție a acestui sistem este şi o soluție a problemei noastre.

Afirmăm că o valoare L_V a variabilei L care satisface ecuațiile (*) şi (**) satisface toate restricțiile problemei.

Pentru a demonstra această afirmație, considerăm valorile întregi B_V şi D_V corespunzătoare valorii L_V. Deoarece aceste valori sunt obținute din ecuația (*), atunci primele trei restricții din enunț sunt, evident, îndeplinite. Vom arăta acum că şi ultima restricție este satisfăcută.

Fie x un ambalaj, ales la întâmplare din cele N. Conform primei restricții, x apare într-o singură cutie produsă într-o anumită zi a ciclului de producție, iar pe întreg ciclul, x apare în D_V cutii. Să presupunem că x nu respectă ultima restricție. Atunci, există cel puțin două cutii care, în afară de x, conțin alte ambalaje comune şi, deci, numărul de ambalaje distincte din cele două cutii este mai mic decât $2 \cdot (L_V - 1)$. Înseamnă că cele D_V cutii în care se găseşte x conțin împreună $D_V \cdot (L_V - 1) < N - 1$ ambalaje, iar ecuația (**) nu este satisfăcută. Acest lucru este imposibil deoarece ipoteza ne asigură că L_V şi D_V satisfac această ecuație. Prin urmare, ultima restricție este satisfăcută pentru x şi, deoarece x a fost ales aleator, este satisfăcută pentru oricare dintre cele N ambalaje.

Această afirmație justifică un algoritm de rezolvare foarte simplu: dintre toate soluțiile sistemului ecuațiilor (*), (**), (***) se alege soluția pentru care valoarea $|D - B|$ este minimă.

```
Algoritm Acadele(N):
  min ← ∞
  pentru L ← 2, N - 1 execută:
    dacă rest[N/L] = 0 şi rest[(N - 1) / (L - 1)] = 0
    atunci                                          { o soluție posibilă }
      B ← N / L
      D ← (N - 1) / (L - 1)
      dacă |D - B| < min atunci
        Lmin ← L
        Bmin ← B
        Dmin ← D
        min ← |D - B|
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
  dacă min ≠ ∞ atunci
    returnează Lmin, Bmin, Dmin
  altfel
    returnează 0, 0, 0
  sfârşit dacă
sfârşit algoritm
```

Acest algoritm este liniar şi funcționează într-un timp acceptabil pentru valori medii ale lui N, dar pentru valori mari nu se va mai încadra în limita de timp admisă.

Din acest motiv algoritmul nu poate rămâne în această formă. Vom prezenta în continuare o adaptare care se va executa într-un timp mai mic decât unul liniar şi anume $O(\sqrt{N})$.

Problema are două proprietăți importante. Prima restrânge intervalul de căutare a soluției: este suficient ca valorile L care satisfac restricțiile problemei să fie căutate în intervalul $[2, \sqrt{N}]$.

A doua proprietate arată că diferența $D - B$ este pozitivă pentru L cuprins între 2 şi $\sqrt{N}$, iar valoarea maximă a diferenței se obține pentru valoarea maximă a lui L, care satisface restricțiile problemei. Vom demonstra în continuare cele două proprietăți.

Pentru început vom arăta că nu există nici o valoare $\sqrt{N} < L < N$ pentru care ecuațiile (*) şi (**) sunt satisfăcute.

Pentru aceasta vom rescrie ecuațiile (*) şi (**) în forma:

$N \bmod L = 0$ (*)

$(N - 1) \bmod (L - 1) = 0$ (**).

Să considerăm că există o valoare $\sqrt{N} < L < N$ care satisface ecuațiile (*) şi (**). Atunci, trebuie să existe doi întregi, $q_1 > 0$ şi $q_2 > 0$, astfel încât:

$N = q_2 \cdot L$

$N - 1 = q_1 \cdot (L - 1)$.

Din aceste ecuații obținem:

$$q_1 = \frac{q_2 \cdot L - 1}{L - 1} = \frac{q_2 \cdot (L-1) + q_2 - 1}{L - 1} = q_2 + \frac{q_2 - 1}{L - 1}.$$

Avem $\sqrt{N} < L$; din ecuația $N = q_2 \cdot L$ obținem $q_2 < \sqrt{N}$, deci $q_2 <$ L. Atunci, raportul dintre $q_2 - 1$ şi $L - 1$ este subunitar şi obținem $q_2 \le q_1 < q_2 + 1$.

Deoarece q_1 şi q_2 sunt valori întregi rezultă că $q_1 = q_2$. Obținem $N / L = (N - 1) / (L - 1)$, adică $N \cdot (L - 1) = L \cdot (N - 1)$. După efectuarea calculelor obținem $N \cdot L - N = N \cdot L - L$. Rezultă egalitatea $N = L$, ceea ce contravine ipotezei.

Vom demonstra acum faptul că valoarea minimă pentru $|D - B|$ se obține pentru cea mai mare valoare L din intervalul $(1, N)$ care satisface ecuațiile (*) şi (**).

Pentru aceasta, vom observa mai întâi că $D - B$ este o funcție care depinde de L:

$$f(L) = \frac{N - 1}{L - 1} - \frac{N}{L} = \frac{N - L}{L \cdot (L - 1)}.$$

Vom arăta că această funcție este strict descrescătoare pentru valorile L cuprinse în intervalul $[2, \sqrt{N}]$.

Vom considera două valori L_1 şi L_2 ($L_1 < L_2$) din acest interval şi vom demonstra că $f(L_1) > f(L_2)$. Practic, trebuie să demonstrăm că:

$$\frac{N - L_1}{L_1 \cdot (L_1 - 1)} > \frac{N - L_2}{L_2 \cdot (L_2 - 1)}.$$

Avem $(N - L_1) \cdot L_2 \cdot (L_2 - 1) > (N - L_2) \cdot L_1 \cdot (L_1 - 1)$ care se reduce, după efectuarea unor calcule la:

$$(L_2 - L_1) \cdot (N \cdot L_2 + N \cdot L_1 - N - L_1 \cdot L_2) > 0.$$

Primul termen al produsului este întotdeauna pozitiv deoarece $L_2 > L_1$. Al doilea termen poate fi scris sub forma:

$$N \cdot (L_2 - 1) + L_1 \cdot (N - L_2).$$

Datorită intervalului ales pentru valorile L_1 şi L_2, şi acest termen este pozitiv. În concluzie, inegalitatea inițială este adevărată pentru intervalele care ne interesează, deci funcția $f(L)$ este descrescătoare pe intervalul considerat.

Rezultă imediat că valoarea minimă posibilă a expresiei $|D - B|$ este obținută pentru cea mai mare valoare L din intervalul $(1, N)$ care satisface ecuațiile (*) şi (**).

Folosind toate aceste afirmații, rezultă că soluția problemei este unică şi corespunde valorii maxime a parametrului L care satisface ecuațiile (*) şi (**).

Se obține astfel următorul algoritm:

```
Algoritm Acadele2(N)
  pentru L ← [sqrt(N)], 2 execută:                    { pas −1}
    dacă rest[N / L] = 0 şi rest[(N - 1) / (L - 1)] = 0
      atunci
       returnează L, N / L, (N - 1) / (L - 1)
    sfârşit dacă
  sfârşit pentru
  returnează 0, 0, 0
sfârşit algoritm
```

Acest algoritm poate fi implementat foarte uşor chiar dacă demonstrarea corectitudinii sale a necesitat câteva observaţii matematice nu foarte uşor de demonstrat.

Analiza complexităţii

Citirea datelor de intrare se realizează în timp constant, deoarece acestea constau într-un singur număr; aşadar, ordinul de complexitate al acestei operaţii este $O(1)$.

Pentru a determina valoarea L, vom considera toate numerele întregi cuprinse între 2 şi $\sqrt{N}$. Pentru fiecare astfel de număr vom efectua câteva operaţii simple, care sunt realizate în timp constant. Aşadar, ordinul de complexitate al operaţiei de determinare a valorii L este $O(\sqrt{N})$.

Afişarea rezultatului obţinut este realizată în timp constant, deoarece se afişează întotdeauna trei numere.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(\sqrt{N}) + O(1) = O(\sqrt{N})$.

2.3.5. Galbeni

Putem considera primele trei grămezi şi folosi algoritmul pentru a rămâne cu două grămezi. La aceste două grămezi vom adăuga cea de-a patra grămadă şi apoi vom aplica acelaşi algoritm. Astfel, din primele patru grămezi am rămas cu două. La acestea vom adăuga cea de-a cincea grămadă şi vom folosi acelaşi algoritm. Continuând în acelaşi mod, vom ajunge la ultima grămadă şi vom rămâne, în final, cu numai două grămezi.

În continuare vom prezenta soluţia problemei cu trei grămezi. Să presupunem că, la un moment dat, numărul galbenilor din cele trei grămezi este a, b şi c. Vom încerca să determinăm o secvenţă de operaţii, astfel încât să obţinem în cele trei grămezi a', b' şi c' galbeni cu proprietatea $\min(a', b', c') < \min(a, b, c)$. Aplicând de mai multe ori această operaţie vom ajunge în situaţia în care minimul va ajunge la 0, deci vom rămâne cu numai două grămezi. Deoarece la fiecare pas minimul va scădea, numărul de galbeni dintr-o grămadă va ajunge după un număr finit de paşi la 0.

Să presupunem că ne aflăm în configurația (a, b, c). Fără a restrânge generalitatea, putem presupune că $a \le b \le c$. (Dacă nu ne aflăm în această situație putem să permutăm grămezile.)

Vom transforma această configurație în (a', b', c'), astfel încât b' = rest[b/a]. Este evident că b' este mai mic decât a (fiind restul unei împărțiri la a), deci $b' < \min(a, b, c)$ adică, $\min(a', b', c') < \min(a, b, c)$.

Pentru a realiza această transformare vom calcula x = [b/a]. Analizăm biții corespunzători reprezentării binare a lui x. Fiecare bit care are valoarea 1 poate fi "anulat", transformând configurația (a, b, c) în ($2 \cdot a$, $b - a$, c). Pentru un bit care are valoarea 0 nu vom mai modifica valoarea b, ci va trebui doar să deplasăm cu o poziție la stânga biții din reprezentarea binară a lui a. Acest lucru se realizează prin transformarea configurației a, b, c în configurația ($2 \cdot a$, b, $c - a$). După parcurgerea tuturor biților lui x vom obține în cea de-a doua grămadă rest[b/a] galbeni.

Vom demonstra în continuare că nu vom avea niciodată un număr negativ de galbeni în nici una dintre cele trei grămezi.

Este evident că numărul elementelor din prima grămadă se dublează la fiecare pas, deci aceasta nu va conține niciodată un număr negativ de galbeni.

Numărul elementelor din a doua grămadă scade permanent, dar în final are valoarea rest[b/a], deci este un număr pozitiv. Ca urmare, nici ce-a de-a doua grămadă nu va conține un număr negativ de galbeni.

Numărul elementelor din cea de-a treia grămadă scade şi el permanent. Vom calcula acum valoarea sa minimă. Este evident că avem $a + b + c = a' + b' + c'$. Mai ştim că b' = rest[b/a], deci avem $a + b' + a \cdot x + c = a' + b' + c'$, unde x = [b/a]. Scăzând valoarea b' din ambii membri obținem $a \cdot (1 + x) + c = a' + c'$. Pentru fiecare bit al lui x, valoarea a se dublează. Având în vedere că numărul biților lui x este $[\log_2 x] + 1$, a' va avea cel mult valoarea $2 \cdot x \cdot a$. Ajungem astfel la inegalitatea $a \cdot (x + 1) + c \le 2 \cdot x \cdot a + c'$ sau $c' \ge c - a \cdot (x - 1)$. Ştim că x = [b/a], deci $b \ge a \cdot x$. Deoarece $c \ge b$ avem $c \ge a \cdot x$. Astfel, inegalitatea devine $c' \ge a$. Deoarece valoarea a este pozitivă, rezultă că şi valoarea c' este pozitivă.

Deoarece numărul de elemente din nici una dintre cele trei grămezi nu poate ajunge negativ, este clar că algoritmul este corect şi poate fi aplicat pentru a rezolva problema în varianta cu trei grămezi.

Vom porni cu primele trei grămezi şi apoi, pe măsură ce numărul de elemente dintr-o grămadă va deveni zero, vom adăuga celelalte grămezi.

Analiza complexității

Citirea datelor de intrare se realizează în timp liniar deoarece acestea constau într-un şir de n numere; aşadar ordinul de complexitate al acestei operații este $O(n)$.

Pentru a elimina o grămadă, numărul operațiilor efectuate este proporțional cu numărul cifrelor egale cu 1 din reprezentarea binară a numărului de galbeni din una dintre grămezi. Datorită faptului că, în total, avem cel mult două miliarde de galbeni, numărul cifrelor binare va fi de cel mult 31, dintre care cel mult 30 vor avea valoarea 1.

Aşadar, putem considera că numărul operațiilor este constant. În total vom elimina $n - 2$ grămezi, ordinul de complexitate al acestei operații fiind $O(n)$.

Afişarea rezultatului obținut este realizată pe măsura efectuării operațiilor, motiv pentru care nu se consumă timp suplimentar pentru scrierea datelor de ieşire.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n) = O(n)$.

2.3.6. Fantome

Pentru simplitate, vom prezenta mai întâi un enunț echivalent al acestei probleme:

O placă dreptunghiulară de dimensiuni $L_1 \times L_2$ este împărțită într-o rețea ortogonală de pătrate de latură 1, în care sunt marcate N puncte albe şi N puncte negre, toate având coordonate întregi. Oricare trei dintre cele $2 \cdot N$ puncte nu sunt coliniare. În scopul obținerii unui circuit imprimat prin corodarea plăcii, fiecare punct alb urmează a fi unit printr-un segment de dreaptă cu un punct negru oarecare, dar astfel încât oricare două dintre cele N segmente să nu se intersecteze.

Din faptul că punctele au coordonate numere reale cu trei zecimale exacte, fiecare valoare se va înmulți cu 1000 şi coordonatele vor deveni numere întregi cuprinse între 1 şi 1000000. Aşadar, în această situație avem $L_1 = L_2 = 1000000$.

Există mai multe soluții pentru rezolvarea problemei. Prima se bazează pe metoda *backtracking*: se încearcă o conectare a punctului alb (vânătorului) curent cu un punct negru (fantomă) neconectat încă, astfel încât segmentul (raza) format(ă) să nu se intersecteze cu nici unul dintre segmentele (razele) deja formate. Este evident că o astfel de soluție nu se va încadra în limita de timp admisă, aşadar nu vom mai insista asupra ei.

Cea de-a doua soluție foloseşte un algoritm care evită metoda *backtracking*. Pentru început vom conecta punctele într-un anumit mod. De exemplu, am putea conecta punctele (vânătorii şi fantomele) în ordinea dată la intrare: primul punct alb (vânător) cu primul punct negru (fantomă) etc.

Vom identifica prin V_i punctele albe (vânătorii) şi cu F_i punctele negre (fantomele). Vom verifica apoi dacă există două segmente V_1F_1 şi V_2F_2 care se intersectează. În acest caz, conexiunile se vor modifica şi noile segmente vor fi V_1F_2 şi V_2F_1. Este evident că aceste segmente nu se vor mai intersecta.

Vom demonstra în continuare că, prin schimbarea conexiunilor, suma lungimilor celor două segmente se reduce. Vom nota cu O intersecția segmentelor V_1F_1 şi V_2F_2. Ştim că într-un triunghi lungimea unei laturi este întotdeauna mai mică decât suma lungimilor celorlalte două. Aşadar, în triunghiul V_1OF_2 avem $V_1F_2 < V_1O + OF_2$, iar în triunghiul V_2OF_1 avem $V_2F_1 < V_2O + OF_1$. Adunând cele două inegalități obținem $V_1F_2 + V_2F_1 < V_1O + OF_1 + V_2O + OF_2$, adică $V_1F_2 + V_2F_1 < V_1F_1 + V_2F_2$.

Aşadar, după fiecare modificare a conexiunilor suma totală a lungimilor segmentelor se reduce. Deoarece această sumă nu va fi niciodată negativă, înseamnă că după un număr finit de paşi nu vom mai avea intersecții.

Se conturează astfel următorul algoritm de rezolvare a problemei: atâta timp cât există două segmente care se intersectează, se modifică conexiunile pentru punctele corespunzătoare.

Ordinul de complexitate al algoritmului prezentat anterior este liniar în suma lungimilor segmentelor, dar nu este polinomial în numărul de puncte, deoarece după modificarea conexiunilor pentru două segmente, segmentele nou formate se pot intersecta cu alte segmente, deci numărul de intersecții nu se reduce neapărat, ci numai suma totală a lungimilor.

Există, totuşi, o modalitate de rezolvare a problemei care să ducă la obținerea unui timp de execuție polinomial în numărul de puncte. Vom încerca să demonstrăm că există o linie care trece printr-un vânător de fantome şi printr-o fantomă, astfel încât numărul vânătorilor de fantome aflați de o parte a liniei este egal cu numărul fantomelor aflate de aceeaşi parte.

Este relativ simplu să determinăm soluția dacă reuşim să găsim o astfel de dreaptă. Pentru rezolvare vom folosi metoda *divide et impera*. Dreapta va împărți vânătorii şi fantomele în două mulțimi fiecare dintre ele conținând un număr egal de vânători şi fantome. Aşadar, acelaşi algoritm poate fi aplicat pentru cele două mulțimi fără a exista pericolul ca vreun segment trasat în una dintre mulțimi să se intersecteze cu un segment trasat în cealaltă mulțime sau cu segmentul care uneşte cele două puncte care determină dreapta care separă mulțimile.

Problema mai dificilă este determinarea dreptei care are proprietatea cerută. Vom considera punctul P având coordonata y cea mai mică, indiferent dacă reprezintă poziția unui vânător sau a unei fantome. Vom sorta toate celelalte puncte în funcție de unghiul format de dreapta care uneşte P de acest punct şi dreapta orizontală care trece prin P (unghiul polar).

Vom avea doi indicatori V şi F, primul va indica numărul vânătorilor luați în considerare, iar al doilea numărul fantomelor. Dacă punctul P reprezintă o fantomă, atunci inițializăm V cu 0 şi F cu 1, iar dacă reprezintă un vânător inițializăm V cu 1 şi F cu 0.

Vom considera, pe rând, punctele sortate; dacă punctul curent reprezintă un vânător incrementăm indicatorul V, iar dacă reprezintă o fantomă incrementăm indicatorul F. În momentul în care vom avea $V = F$ vom şti că sub dreapta care uneşte punctul P cu punctul curent se află un număr egal de vânători şi fantome.

Vom demonstra în continuare că, la un moment dat, vom ajunge în situația în care $V = F$. Presupunem, fără a restrânge generalitatea, că în punctul P se află un vânător.

Dacă în primul punct se află o fantomă atunci, ajungem în situația $V = F = 1$, deci am determinat deja o dreaptă cu proprietatea cerută. Dacă în ultimul punct se află o fantomă atunci, fie am găsit o dreaptă anterior, fie acest ultim punct şi punctul P determină dreapta pe care o căutăm. Aşadar, cazul în care primul sau ultimul punct reprezintă o fantomă este rezolvat.

Să presupunem acum că atât primul, cât şi ultimul punct reprezintă un vânător. În acest caz inițial vom avea $V = 1$ şi $F = 0$ ($V - F = 1$), iar înaintea considerării ultimului punct $V = n - 1$ şi $F = n$ ($V - F = -1$).

Deoarece valoarea $V - F$ nu se poate modifica decât cu 1 la fiecare pas, pentru a ajunge de la 1 la –1 ea trebuie neapărat să treacă şi prin 0. Aşadar, la un moment dat ne vom afla în situația pentru care $V = F$, deci vom avea dreapta cerută.

Vom descrie acum o metodă pe baza căreia punctele se pot sorta în funcție de unghiul polar pe care îl formează cu punctul P. Să presupunem că, în sistemul de coordonate cu originea în P, avem două puncte A şi B de coordonate (x_1, y_1) şi (x_2, y_2).

Pentru a sorta punctele în funcție de unghi avem trei cazuri. Dacă punctele A şi B se află în primul cadran al sistemului de coordonate cu originea în P, atunci punctul A se află înaintea punctului B în şirul sortat dacă şi numai dacă $y_1 / x_1 < y_2 / x_2$, adică $y_1 \cdot x_2 < y_2 \cdot x_1$, deoarece coordonatele orizontale sunt pozitive. (Am înlocuit împărțirile cu înmulțiri pentru a mări viteza de execuție şi a evita împărțirea cu zero.) Dacă punctele A şi B se află în cadrane diferite, atunci punctul din primul cadran îl va precede pe cel din al doilea cadran. Dacă punctele A şi B se află în al doilea cadran al sistemului de coordonate cu originea în P, atunci punctul A se află înaintea punctului B în şirul sortat dacă şi numai dacă $y_1 / x_1 < y_2 / x_2$, adică $y_1 \cdot x_2 > y_2 \cdot x_1$, deoarece coordonatele orizontale sunt negative. Datorită faptului că am ales punctul P având coordonata y minimă, punctele A şi B nu se pot afla în al treilea sau al patrulea cadran.

Analiza complexității

Datele de intrare constau în coordonatele a $2 \cdot n$ puncte; operația de citire a acestora are ordinul de complexitate $O(n)$.

Pentru determinarea unei drepte va trebui să realizăm o sortare a punctelor şi o parcurgere a punctelor sortate. Ordinul de complexitate al operației de sortare este $O(n \cdot \log n)$, iar cel al operației de parcurgere este $O(n)$. Ca urmare ordinul de complexitate al operației de determinare a unei drepte este $O(n \cdot \log n)$. În total vom determina n astfel de drepte, operația de determinare a tuturor având, ca urmare, ordinul de complexitate $O(n^2 \cdot \log n)$.

Datele de ieşire constau în afişarea a n perechi de numere, operație al cărei ordin de complexitate este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n^2 \cdot \log n) + O(n) = O(n^2 \cdot \log n)$.

2.3.7. Unitate

O primă metodă de rezolvare a problemei se bazează pe utilizarea *programării dinamice*. Se construieşte o matrice B de dimensiuni $m \times n$ în care elementul b_{ij} reprezintă numărul minim de unități care trebuie adăugate pentru ca toate elementele matricei inițiale care se află pe primele i linii şi primele j coloane să aibă valoarea 1.

Cu alte cuvinte, elementul b_{ij} conține numărul de unități care trebuie adăugate pentru ca submatricea care are colțul stânga-sus în poziția (1, 1) şi colțul dreapta-jos în poziția (i, j) să conțină numai elemente cu valoarea 1.

Modul de calcul al elementelor matricei B este următorul:

- Elementele de pe prima linie se calculează pe baza formulelor: $b_{11} = 1 - a_{11}$ şi $b_{1,i} = 1 - a_{1,i} + b_{1,i-1}$ pentru $i > 1$.
- Elementele de pe prima coloană se calculează pe baza unor formule asemănătoare: $b_{11} = 1 - a_{11}$ şi $b_{i,1} = 1 - a_{i,1} + b_{i-1,1}$ pentru $i > 1$.
- Dacă $a_{ij} = 1$, atunci nu mai trebuie adăugată nici o unitate pentru a obține cifra 1 pe poziția (i, j), deci vom avea:
$$b_{i,j} = b_{i,j-1} + b_{i-1,j} - b_{i-1,j-1}.$$
- Dacă $a_{ij} = 0$, atunci trebuie adăugate un număr minim de unități pentru a obține cifra 1 pe poziția (i, j). Dacă notăm cu k minimul dintre i şi j, vom adăuga valoarea 2^{k-1}, deci formula va deveni:
$$b_{i,j} = b_{i,j-1} + b_{i-1,j} + b_{i-1,j-1} + 2^{k-1}.$$

Analizând această soluție observăm că nu se adaugă unități decât în cazul în care întâlnim un element a_{ij} cu valoarea 0, iar numărul de unități adăugate în acest caz este $2^{\min(i, j)-1}$.

Astfel algoritmul de programare dinamică poate fi înlocuit cu următorul:

```
Algoritm Unitate:
  total ← 0
  pentru i ← 1, m execută:
    pentru i ← 1, n execută:
      dacă a[i,j] = 0 atunci
         total ← total + 2^(min(i,j)-1)
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
  returnează total
sfârşit algoritm
```

Datorită faptului că matricea poate avea până la 100 de linii sau coloane, s-ar putea să trebuiască să adunăm valori foarte mari cum ar fi 2^{99}. Datorită acestui fapt va trebui să implementăm operațiile de adunare şi înmulțire pentru numere întregi foarte mari.

Pentru a micşora timpul de execuție nu vom efectua adunările în momentul în care vom întâlni un element cu valoarea 0, ci vom păstra un vector v care să indice de câte ori va trebui să adunăm valoarea 2^i. O putere a lui 2 poate fi foarte uşor determinată pe baza puterii anterioare şi pentru fiecare putere vom înmulți rezultatul cu valoarea corespunzătoare din vectorul v, apoi îl vom aduna la valoarea care va indica în final numărul de unități care trebuie adăugate.

Analiza complexității

Datele de intrare constau într-o matrice cu m linii şi n coloane, deci operația de citire a acestora are ordinul de complexitate $O(m \cdot n)$.

Pentru a determina valorile care trebuie adunate este suficient să parcurgem elementele matricei, operație care are ordinul de complexitate $O(m \cdot n)$.

Vom avea cel mult $\min(m, n)$ puteri distincte ale lui 2 care vor trebui adunate. Datorită faptului că numărul cifrelor este limitat putem considera că o operație cu numere mari se va realiza în timp constant. Pentru fiecare putere vom efectua cel mult o înmulțire şi o adunare; în total vom avea cel mult $\min(m, n)$ înmulțiri şi cel mult $\min(m, n)$ adunări. Ca urmare, ordinul de complexitate al operației de determinare a rezultatului este $O(\min(m, n))$.

Operația de afişare a rezultatului obținut este realizată în timp constant, deoarece acesta constă într-un singur număr. Ca urmare, ordinul de complexitate al acestei operații este $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m \cdot n) + O(m \cdot n) + O(\min(m, n)) + O(1) = O(m \cdot n)$.

2.3.8. Sfere

Să presupunem că am ales k sfere; vom încerca să determinăm cu cât creşte numărul de zone în care este împărțit spațiul după alegerea celei de-a $(k + 1)$-a sfere.

Numărul maxim de zone se obține dacă oricare două sfere se intersectează, adică nu există nici o pereche de sfere tangente, nu există nici o sferă care să se afle complet în interiorul unei alte sfere, nu există trei sfere care să aibă un cerc comun şi oricare patru sfere nu au nici un punct comun.

Sfera considerată se va intersecta cu celelalte k sfere după k cercuri. Vom încerca să determinăm acum numărul maxim de zone în care aceste k cercuri pot împărți suprafața sferei.

Deoarece dimensiunile sferelor nu au nici o importanță, putem considera că noua sferă are o rază mult mai mare decât cea a celorlalte k sfere. La limită, vom considera că raza acestei sfere este infinită. Datorită faptului că, în acest caz, sfera degenerează într-un plan (un plan poate fi considerat ca fiind o sferă de rază infinită) problema în spațiu se reduce la o problemă în plan.

Enunțul acestei noi probleme este următorul: *determinați numărul maxim de zone în care poate fi împărțit planul prin intermediul a n cercuri*.

Cercurile împart planul într-un număr maxim de zone dacă oricare două cercuri se intersectează, adică nu există nici o pereche de cercuri tangente, nu există nici un cerc care să se afle complet în interiorul unui alt cerc şi nu există nici un grup de trei cercuri care să aibă un punct comun.

Considerăm că am ales k cercuri şi încercăm să determinăm numărul de zone suplimentare care apar dacă alegem cel de-al $(k + 1)$-lea cerc. Deoarece cercul se intersectează cu fiecare dintre celelalte k cercuri în câte două puncte, rezultă că circumfe-

rința sa va fi împărțită în $2 \cdot k$ regiuni. Ca urmare, numărul de zone suplimentare care apar după considerarea unui nou cerc este egal cu dublul numărului cercurilor considerate anterior.

Aşadar, relația de recurență, cu ajutorul căreia să se poată determina numărul de zone în care poate fi împărțit un plan, este $c_{k+1} = c_k + 2 \cdot k$. Deoarece un cerc împarte planul în două zone (interiorul cercului şi exteriorul său), avem $c_1 = 2$.

Astfel obținem:

$$\begin{aligned} c_n &= 2+2+4+6+8+\ldots+2\cdot(n-1) \\ &= 2+2\cdot(1+2+3+\ldots+n-1) \\ &= 2+2\cdot\frac{n\cdot(n-1)}{2} \\ &= n^2-n+2. \end{aligned}$$

Aşadar, numărul maxim de zone în care k cercuri pot împărți suprafața unei sfere este de $k^2 - k + 2$. Ca urmare, prin considerarea celei de-a $(k + 1)$-a sfere numărul de zone este mărit cu $k^2 - k + 2$. Relația de recurență, cu ajutorul căreia se poate determina numărul de zone în care poate fi împărțit spațiul, este $s_{k+1} = s_k + k^2 - k + 2$.

Deoarece o sferă împarte spațiul în două zone (interiorul şi exteriorul ei), $c_1 = 2$.

Efectuând calculele obținem:

$$\begin{aligned} s_n &= 2+(1^2-1+2)+(2^2-2+2)+(3^2-3+3)+\ldots+[(n-1)^2-(n-1)+2] \\ &= 2+[1^2+2^2+3^2+\ldots+(n-1)^2]-[1+2+3+\ldots+(n-1)]+2\cdot(n-1) \\ &= 2+\frac{n\cdot(n-1)\cdot(2\cdot n-1)}{6}-\frac{n\cdot(n-1)}{2}+2\cdot(n-1) \\ &= \frac{12+2\cdot n^3-3\cdot n^2+n-3\cdot n^2+3\cdot n+12\cdot n-12}{6} \\ &= \frac{2\cdot n^3-6\cdot n^2+16\cdot n}{6} \\ &= \frac{n\cdot(n^2-3\cdot n+8)}{3}. \end{aligned}$$

Aşadar, numărul maxim de zone în care poate fi împărțit spațiul cu ajutorul a n sfere este: $\frac{n\cdot(n^2-3\cdot n+8)}{3}$.

Analiza complexității

Citirea numărului n se realizează în timp constant, deci are ordinul de complexitate $O(1)$, iar operația de determinare a numărului maxim de zone în care poate fi împărțit spațiul are ordinul de complexitate tot $O(1)$, deoarece este utilizată o simplă formulă matematică. Operația de scriere a rezultatului are, de asemenea, ordinul de complexitate $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(1) + O(1) = O(1)$.

2.3.9. Asemănare

Potrivit definiţiei asemănării poligoanelor, două poligoane se vor numi asemenea dacă şi numai dacă au unghiurile respectiv congruente şi laturile respectiv proporţionale.

Ştim că în cazul particular al triunghiurilor, pentru a demonstra asemănarea este suficient să arătăm că una dintre următoarele afirmaţii este adevărată:

- triunghiurile au unghiurile respectiv congruente;
- triunghiurile au laturile respectiv proporţionale;
- triunghiurile au două laturi respectiv proporţionale şi unghiurile formate de aceste laturi în cele două triunghiuri sunt congruente.

Un poligon cu n laturi poate fi triangularizat în $n - 2$ triunghiuri. Dacă pentru două poligoane cele $n - 2$ triunghiuri sunt respectiv asemenea, atunci putem afirma că poligoanele sunt asemenea.

De aici putem deduce că două poligoane sunt asemenea dacă au laturile şi diagonalele care pleacă dintr-un anumit vârf respectiv proporţionale.

Vom rezolva problema folosind această proprietate. Vom considera vârfurile primului poligon în ordinea dată şi vom încerca să verificăm dacă putem determina o ordine a vârfurilor celui de-al doilea poligon pentru care să obţinem asemănarea.

Vom începe pe rând cu fiecare dintre vârfurile 1, ..., n. Pentru fiecare vârf k, vom considera pentru început ordinea k, $k + 1$, ..., n, 1, 2, ... $k - 1$ şi apoi ordinea k, $k - 1$, ..., 1, n, $n - 1$, ..., $k + 1$.

Pentru a reduce puţin numărul calculelor vom lucra cu pătratele distanţelor în loc de distanţe propriu-zise. Acest lucru este posibil deoarece următoarea relaţie este întotdeauna adevărată:

$$\frac{a}{b} = \frac{c}{d} \Leftrightarrow \frac{a^2}{b^2} = \frac{c^2}{d^2}.$$

De asemenea, în loc de a testa o relaţie de tipul $a / b = c / d$, vom testa relaţia echivalentă $a \cdot d = b \cdot c$.

Analiza complexităţii

Datele de intrare constau în coordonatele celor n puncte ale fiecărui poligon, operaţia de citire a acestora având ordinul de complexitate $O(n)$.

Pentru fiecare posibilă ordine a vârfurilor din al doilea poligon, va trebui să verificăm dacă laturile şi diagonalele care pleacă dintr-un vârf sunt proporţionale. O astfel de verificare se realizează în timp liniar şi, datorită faptului că vom avea cel mult $2 \cdot n$ astfel de verificări, ordinul de complexitate al operaţiei de verificare a asemănării dintre poligoane este $O(n^2)$.

Datele de ieşire constau fie într-o ordine a celor n vârfuri ale celui de-al doilea poligon, fie doar din numărul 0. Aşadar, în cazul cel mai defavorabil, ordinul de complexitate al operației de scriere a acestora este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n^2) + O(n) = O(n^2)$.

2.3.10. Dreptunghi

Pentru a rezolva această problemă vom folosi metoda *programării dinamice*.

Vom construi o matrice A de dimensiuni $m \times n$, elementul a_{ij} indicând numărul minim de pătrate în care poate fi împărțit un dreptunghi care are lungimile laturilor i şi j. Este evident faptul că elementele a_{ii} au valoarea 1.

Pentru a determina valorile elementelor a_{ij} va trebui să determinăm poziția în care ar trebui să efectuăm tăietura pentru a minimiza numărul pătratelor în care va fi împărțit dreptunghiul care are laturi de lungimi i şi j.

Pentru aceasta vom determina numărul de pătrate corespunzător fiecărei tăieturi verticale şi fiecărei tăieturi orizontale. Formula pe care o vom folosi este:

$$a_{ij} = \min\left(\min_{k=1,i} \left(a_{kj} + a_{i-k,j}\right), \min_{k=1,j} \left(a_{ik} + a_{i,j-k}\right) \right).$$

Se observă că numărul minim de pătrate în care poate fi împărțit un dreptunghi cu laturile i şi j este egal cu numărul minim de pătrate în care poate fi împărțit orice dreptunghi cu laturile $k \cdot i$ şi $k \cdot j$, pentru orice număr natural $k > 0$.

De aceea, pentru fiecare element al matricei A, am putea verifica dacă nu am determinat deja valoarea corespunzătoare. Valoarea a fost deja determinată numai dacă $\text{cmmdc}(i, j) > 1$. În acest caz avem $a_{ij} = a_{i/\text{cmmdc}(i,j),\, j/\text{cmmdc}(i,j)}$.

Prin aceasta timpul de execuție se reduce semnificativ. Totuşi, o variantă de rezolvare care nu foloseşte această observație se încadrează fără probleme în timpul de execuție admis.

Analiza complexității

Datele de intrare constau în dimensiunile dreptunghiului, operația de citire a acestora având ordinul de complexitate $O(1)$.

Pentru a determina un element al matricei a va trebui să parcurgem elementele aflate pe aceeaşi linie, în stânga şi elementele aflate pe aceeaşi coloană deasupra. Aşadar, pentru un element de pe poziția a_{ij} vom parcurge $i - 1 + j - 1$ elemente.

Vom determina acum numărul total de elemente care vor fi parcurse păstrându-se coloana. Pentru cele n elemente ale unei linii i, numărul total va fi $n \cdot (i - 1)$. Pentru toate cele m linii vom avea:

$$\sum_{i=1}^{m} n \cdot (i-1) = \frac{n \cdot m \cdot (m-1)}{2}.$$

În mod analog, numărul total de elemente care vor fi parcurse păstrându-se linia, pentru o anumită coloană j va fi $m \cdot (j - 1)$. Pentru toate cele n coloane vom avea:

$$\sum_{i=1}^{n} m \cdot (i-1) = \frac{m \cdot n \cdot (n-1)}{2}.$$

Ca urmare, numărul total de elemente va fi:

$$\frac{n \cdot m \cdot (m-1)}{2} + \frac{m \cdot n \cdot (n-1)}{2} = \frac{m \cdot n \cdot (m+n-2)}{2}.$$

Aşadar, ordinul de complexitate al operației de determinare a elementelor matricei A este $O(m \cdot n \cdot (m + n))$.

Datele de ieşire constau într-un singur număr, ordinul de complexitate al operației de scriere a acestora fiind $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(m \cdot n \cdot (m + n)) + O(1) = O(m \cdot n \cdot (m + n))$.

2.3.11. Alb şi negru

Metoda de rezolvare poate fi găsită foarte uşor folosind inducția matematică. Pentru început trebuie observat faptul că prima mutare admisă este fie deplasarea unei bile albe cu o poziție spre stânga, fie deplasarea unei bile negre cu o poziție spre dreapta. Vom presupune acum că vom muta bila albă; soluția corespunzătoare mutării unei bile negre la început poate fi găsită efectuând mutările simetric – în locul efectuării unei mutări cu o bilă albă vom efectua mutarea corespunzătoare cu o bilă neagră şi invers. Fiecărei alternative îi corespunde o singură soluție.

Vom nota bilele albe cu `A` şi bilele negre cu `B`. Configurația inițială este:

`|A...AAAA BBBB...B|`.

La primul pas vom muta bila albă; astfel vom ajunge în configurația:

`|A...AAA ABBBB...B|`.

Vom muta acum bila neagră pentru a ajunge în configurația `|A...AAABA BBB...B|`.

Urmează o nouă mutare cu o bilă neagră. Configurația în care se ajunge este `|A...AAABAB BB...B|`.

În acest moment vom efectua două mutări cu bilele albe. Configurația obținută este `|A...AA BABABB...B|`.

Urmează o deplasare cu o poziție a unei bile albe şi apoi trei deplasări ale bilelor negre, cu două poziții. Configurația este `|A...ABABABA B...B|`.

Va urma o deplasare a unei bile negre cu o poziție şi patru deplasări ale bilelor albe cu două poziții, o deplasare a unei bile albe cu o poziție şi cinci deplasări ale bilelor negre cu două poziții etc.

După ce vom ajunge la pasul în care am efectuat n deplasări vom deplasa din nou o bilă cu o poziție şi apoi vom efectua $n - 1$ deplasări cu două poziții etc. În final vom obține configurația cerută `|A...AAABA BBB...B|`.

Vom exemplifica algoritmul descris pentru $n = 4$:

Inițial: |AAAA BBBB|;
Pasul 1: |AAA ABBBB|;
Pasul 2: |AAABA BBB|;
Pasul 3: |AAABAB BB|;
Pasul 4: |AAAB BABB|;
Pasul 5: |AA BABABB|;
Pasul 6: |A ABABABB|;
Pasul 7: |ABA ABABB|;
Pasul 8: |ABABA ABB|;
Pasul 9: |ABABABA B|;
Pasul 10: |ABABABAB |;
Pasul 11: |ABABAB BA|;
Pasul 12: |ABAB BABA|;
Pasul 13: |AB BABABA|;
Pasul 14: | BABABABA|;
Pasul 15: |B ABABABA|;
Pasul 16: |BBA ABABA|;
Pasul 17: |BBABA ABA|;
Pasul 18: |BBABABA A|;
Pasul 19: |BBABAB AA|;
Pasul 20: |BBAB BAAA|;
Pasul 21: |BB BABAAA|;
Pasul 22: |BBB ABAAA|;
Pasul 23: |BBBBA AAA|;
Pasul 24: |BBBB AAAA|.

Analiza complexității

Datele de intrare constau în numărul bilelor albe (care este egal cu cel al bilelor negre), operația de citire a acestuia având ordinul de complexitate $O(1)$.

Imediat după citire, putem trece direct la scrierea soluției. Se observă că numărul deplasărilor creşte cu 1 la fiecare pas, până în momentul în care ajunge la n, după care scade cu 1 la fiecare pas, până în momentul în care ajunge la 1. Aşadar, numărul caracterelor scrise va depinde de pătratul valorii n, ordinul de complexitate al operației de scriere fiind $O(n^2)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(n^2) = O(n^2)$.

2.3.12. Coeficient

Transpusă în termeni de teoria grafurilor, problema cere determinarea drumului minim între două vârfuri ale unui graf, dar costul unui drum nu este dat de suma costurilor muchiilor componente, ci de produsul lor.

Totuşi, ea poate fi foarte uşor transformată în problema clasică a drumului minim, dacă transformăm operația de înmulțire în operație de adunare. Acest lucru poate fi realizat foarte uşor dacă logaritmăm costurile muchiilor şi determinăm logaritmul costului drumului minim, după care calculăm şi costul real al acestui drum. Deoarece costul real poate fi foarte mare, trebuie să implementăm operații cu numere mari.

Datorită limitei specificate de 1000 de noduri, este suficient să implementăm versiunea algoritmului lui *Dijkstra* cu complexitatea $O(n^2)$.

Analiza complexității

Datele de intrare constau în descrierea muchiilor grafului care reprezintă oraşele şi străzile dintre ele. Ordinul de complexitate al operației de citire a acestora este $O(m)$.

Pentru acest graf va trebui să aplicăm algoritmul lui *Dijkstra*, al cărui ordin de complexitate este $O(n^2)$.

Datele de ieşire constau într-un singur număr care este calculat folosind operații cu numere mari (înmulțiri) al căror timp de execuție poate fi considerat constant. Vor fi cel mult n înmulțiri, deci ordinul de complexitate al operației de determinare şi afişare a rezultatului final este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m) + O(n^2) + O(n) = O(n^2 + m)$.

2.3.13. Matrice

Este evident că efectuarea a două operații asupra aceleiaşi linii sau a aceleiaşi coloane nu are nici un efect.

O valoare -1 poate părea pe poziția (i, j) dacă s-a efectuat o operație asupra liniei i şi nu s-a efectuat o operație asupra liniei j sau invers.

Dacă notăm cu x numărul operațiilor efectuate asupra liniilor şi cu y al celor efectuate asupra coloanelor, atunci numărul total de valori -1 va fi $x \cdot (m - y) + y \cdot (n - x)$. Această valoare trebuie să fie k, deci va trebui să rezolvăm ecuația $x \cdot (m - y) + y \cdot (n - x) = k$.

Vom rezolva ecuația considerând toate perechile (x, y) posibile. În cazul în care aceasta are soluție, putem alege oricare x linii şi oricare y coloane.

Analiza complexității

Datele de intrare constau în dimensiunile matricei şi numărul elementelor cu valoarea –1 care pot fi obținute, deci operația de citire a acestora are ordinul de complexitate $O(1)$.

Pentru a rezolva ecuația va trebui să verificăm, în cel mai defavorabil caz, toate perechile (x, y) care sunt în număr de $m \cdot n$. Aşadar, această operație va avea ordinul de complexitate $O(m \cdot n)$.

După determinarea valorilor x şi y va trebui să scriem în fişierul de ieşire datele corespunzătoare celor y coloane şi x linii. În cazurile defavorabile x va avea o valoare apropiată de m, iar y va avea o valoare apropiată de n. Ca urmare, ordinul de complexitate al operației de scriere a datelor de ieşire este $O(m + n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(m \cdot n) + O(m + n) = O(m \cdot n)$.

2.3.14. Inversiuni

Cea mai simplă idee de rezolvare este de a lua în considerare toate perechile (i, j) pentru care $i < j$ şi a verifica dacă $a_i > a_j$.

Este evident că această metodă foloseşte $n \cdot (n - 1) / 2$ comparări, deci ordinul de complexitate al algoritmului este $O(n^2)$.

Chiar şi un program care implementează un astfel de algoritm s-ar încadra în limita de timp admisă dacă nu efectuează operații suplimentare inutile.

Există şi un algoritm de rezolvare, bazat pe *metoda divide et impera*, care are ordinul de complexitate $O(n \cdot \log n)$.

Etapa divide

Se împarte şirul de numere în două subşiruri de lungimi aproximativ egale. Primul subşir va conține prima jumătate a numerelor, iar al doilea cea de-a doua jumătate.

Etapa stăpâneşte

Se apelează recursiv algoritmul pentru cele două subşiruri formate şi se determină numărul de inversiuni din fiecare dintre ele.

Etape combină

Numărul de inversiuni din şir va fi egal cu suma numărului de inversiuni din cele două subşiruri la care se adaugă numărul de inversiuni formate de elemente din primul subşir cu elemente din al doilea subşir.

Pentru fiecare element din al doilea subşir va trebui să determinăm numărul de elemente din primul subşir care sunt mai mari decât el. Pentru aceasta vom ordona cele două subşiruri (această operație poate fi realizată pe parcursul apelurilor recursive succesive) şi vom realiza o simplă operație de interclasare. În momentul în care vom adăuga la şirul interclasat un element din al doilea subşir, vom şti câte elemente mai mari decât el se află în primul subşir.

Practic, folosim algoritmul *mergesort* de ordonare a unui şir de numere naturale. Singura operație suplimentară pe care o efectuăm este numărarea inversiunilor.

Analiza complexității

Datele de intrare constau în cele n elemente ale şirului, ordinul de complexitate al operației de citire a acestora fiind $O(n)$.

Pentru acest şir vom aplica o variantă a algoritmului de sortare prin interclasare, algoritm al cărui ordin de complexitate este $O(n \cdot \log n)$.

Datele de ieşire constau într-un singur număr, deci ordinul de complexitate al operației de scriere a rezultatului final este $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n \cdot \log n) + O(1) = O(n \cdot \log n)$.

2.3.15. Bile

Problema poate fi rezolvată folosind metoda *programării dinamice*. Vom construi o matrice cost; elementul $cost_{ij}$ va indica lungimea minimă totală a barelor folosite pentru a lega bilele cuprinse între pozițiile i şi j. În final, elementul $cost_{1,2n}$ va conține lungimea totală a barelor necesare pentru a lega toate bilele.

Este evident că toate elementele de pe diagonala principală a matricei vor avea valoarea 0. Inițial vom considera că celelalte elemente (doar cele aflate deasupra diagonalei principale sunt semnificative) au valoarea ∞.

Pentru două bile vecine (aflate pe pozițiile i şi $i + 1$) care au culori diferite putem afirma că valoarea $cost_{i,i+1}$ este 3, deoarece avem nevoie de două bare verticale şi una orizontală, toate având lungimea 1.

Nu are rost să determinăm valorile corespunzătoare pozițiilor i şi $i + k$, pentru care k este par, deoarece nu avem posibilitatea de a lega un număr impar de bile. Aşadar, la paşii următori vom determina valorile corespunzătoare pozițiilor i şi $i + k$, pentru care k este impar.

Există două posibilități de a determina valoarea minimă corespunzătoare pozițiilor i şi j.

Dacă bilele au culori diferite, atunci putem să legăm cele două bile cu o bară orizontală de lungime $j - i$ şi două verticale care au o înălțime cu 1 mai mare decât cea mai mare înălțime a barelor verticale de pe pozițiile cuprinse între $i + 1$ şi $j - 1$.

Se observă că mai avem nevoie de o matrice (*height*), elementul $height_{ij}$ indicând înălțimea maximă a barelor verticale cuprinse între pozițiile i şi j.

O altă posibilitate de obținere a unei valori pentru $cost_{ij}$ (indiferent de culoarea bilelor de pe pozițiile i şi j) este de a alege o poziție intermediară k şi a aduna valorile $cost_{ik}$ şi $cost_{kj}$. Valoarea $height_{ij}$ va fi egală cu maximul valorilor $height_{ik}$ şi $height_{kj}$. Vom alege toate valorile k pentru care $k - i$ este un număr par.

Evident, vom alege cea mai mică valoare pe care o determinăm folosind cele două posibilități.

Folosind această metodă vom completa toate elementele $cost_{ij}$ pentru care există o posibilitate de a lega toate bilele cuprinse între i şi j. Pentru toate celelalte elemente se va păstra valoarea ∞.

După determinarea valorii minime trebuie reconstituite legăturile realizate. Pentru aceasta trebuie să determinăm modul în care au fost calculate valorile $cost_{ij}$. Va trebui să ştim dacă am folosit prima sau a doua metodă şi, în cazul celei de-a doua metode, care este valoarea k aleasă. Vom păstra încă o matrice (*inter*), iar elementele $inter_{ij}$ vor avea valoarea -1 dacă a fost folosită prima posibilitate sau valoarea k dacă a fost folosită cea de-a doua.

Legăturile între poziţiile i şi j se vor afişa printr-o rutină recursivă; iniţial ea va fi apelată pentru poziţiile 1 şi $2 \cdot n$.

În cazul în care se foloseşte prima posibilitate pentru poziţiile i şi j, vom "tipări" legătura dintre i şi j şi apoi vom apela recursiv rutina pentru poziţiile $i + 1$ şi $j - 1$.

În cazul în care se foloseşte cea de-a doua posibilitate pentru poziţiile i şi j, nu vom tipări nimic şi vom apela recursiv rutina pentru poziţiile i şi k şi pentru poziţiile $k + 1$ şi j.

Analiza complexităţii

Datele de intrare constau în culorile celor $2 \cdot n$ bile, ordinul de complexitate al operaţiei de citire a acestora fiind $O(n)$.

Pentru fiecare pereche de bile (de culori diferite), va trebui să luăm în considerare toate poziţiile dintre ele. Vom avea cel mult n^2 astfel de perechi, iar numărul poziţiilor este mai mic decât $2 \cdot n$ (cel mult $2 \cdot n - 2$). Ca urmare, ordinul de complexitate al operaţiei de determinare a rezultatului va fi $O(n^3)$.

Deşi metoda prin care se realizează scrierea datelor de ieşire pare destul de complicată, la fiecare pas vom scrie o linie care corespunde unei perechi de bile. Aşadar, ordinul de complexitate al operaţiei de scriere a datelor de ieşire este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n^3) + O(n) = O(n^3)$.

2.3.16. Robot

Pentru a păstra traseul robotului vom folosi o matrice ale cărei elemente vor reprezenta pătrăţelele caroiajului. Fiecare element va păstra informaţii referitoare la laturile pătrăţelului corespunzător: existenţa unei linii deasupra, sub, în stânga sau în dreapta pătrăţelului. Cea mai simplă posibilitate de implementare este folosirea a patru biţi, fiecare reprezentând una dintre cele patru laturi şi având valoarea 1 dacă există linie pe acea latură şi 0 în caz contrar.

După construirea matricei, trebuie determinat dreptunghiul vid de arie minimă care se formează. Pentru simplificare, vom trata patru cazuri:

1. dreptunghiuri de lungime şi înălţime 1.
2. dreptunghiuri de lungime 1 şi înălţime diferită de 1.
3. dreptunghiuri de lungime diferită de 1 şi înălţime 1.
4. dreptunghiuri de lungime şi înălţime diferite de 1.

Pentru primul caz va trebui doar să căutăm pătrăţele care au linii pe toate cele patru laturi ale lor. Dacă găsim un astfel de pătrăţel, atunci soluţia problemei este un dreptunghi de arie 1.

Pentru al doilea caz va trebui să căutăm pătrăţele care au linii pe trei dintre laturi (deasupra, sub şi la stânga). Vom parcurge apoi pătrăţelele din dreapta sa care au linii deasupra şi dedesubt.

Când întâlnim primul pătrăţel care nu respectă această condiţie, e posibil să fi ajuns la ultimul pătrăţel al dreptunghiului. Dacă acesta are linii deasupra, sub şi la dreapta, atunci am găsit un nou dreptunghi. Dacă aria acestuia este mai mică decât cea a celui mai mic dreptunghi determinat anterior, vom păstra noua soluţie.

Pentru al treilea caz va trebui să căutăm pătrăţele care au linii pe trei dintre laturi (deasupra, la stânga şi la dreapta). Vom parcurge apoi pătrăţelele de sub el, care au linii la stânga şi la dreapta.

Când întâlnim primul pătrăţel care nu respectă această condiţie, e posibil să fi ajuns la ultimul pătrăţel al dreptunghiului. Dacă acesta are linii dedesubt, la stânga şi la dreapta, atunci am găsit un nou dreptunghi. Dacă aria acestuia este mai mică decât cea a celui mai mic dreptunghi determinat anterior, vom păstra noua soluţie.

Pentru al patrulea caz vom căuta pătrăţele care au linii pe două dintre laturi (la stânga şi deasupra). Vom parcurge apoi pătrăţelele din dreapta sa care au linii numai deasupra.

Când întâlnim un pătrăţel care nu respectă această condiţie, e posibil să fi ajuns la latura din dreapta a dreptunghiului. Acest pătrăţel trebuie să aibă linii pe latura de deasupra şi pe cea din dreapta. Vom parcurge acum pătrăţelele aflate sub pătrăţelul iniţial, care au linii numai la stânga.

Când întâlnim un pătrăţel care nu respectă această condiţie, e posibil să fi ajuns la latura de jos a dreptunghiului. Acest pătrăţel trebuie să aibă linii pe latura de dedesubt şi pe cea din stânga. Dacă toate condiţiile au fost îndeplinite, am reuşit să determinăm un posibil dreptunghi.

În plus, trebuie să verificăm dacă acesta este unul valid. Pentru aceasta vom parcurge pătrăţelele de pe latura de jos şi vom verifica dacă nu au şi alte linii cu excepţia celei de dedesubt.

Analog, pătrăţelele de pe latura din dreapta trebuie să aibă linii numai pe latura din dreapta. În sfârşit, pătrăţelul din colţul dreapta-jos trebuie să aibă linii pe latura din dreapta şi pe cea de dedesubt.

Dacă şi aceste condiţii sunt îndeplinite, atunci am găsit un nou dreptunghi. Dacă aria acestuia este mai mică decât cea a celui mai mic dreptunghi determinat anterior, vom păstra noua soluţie.

O observaţie importantă este că nu trebuie să verificăm decât pătrăţelele de pe laturi, nu şi pe cele din interior. Aceasta se datorează continuităţii liniei care formează traseul robotului. Pentru ca o linie să se afle în interiorul unui dreptunghi, aceasta ar fi

trebuit să pornească din exteriorul acestuia, deci s-ar fi intersectat cu cel puțin una dintre laturi şi dreptunghiul nu ar mai fi fost valid.

Analiza complexității

Datele de intrare constau în cele c comenzi ale robotului, ordinul de complexitate al operației de citire a acestora fiind $O(c)$.

Datorită faptului că robotul trebuie să aibă întotdeauna coordonatele cuprinse între 1 şi 100, operațiile se vor efectua doar asupra acestei regiuni. Din această cauză se poate considera că timpul de execuție al acestora este constant. Deşi, din punct de vedere teoretic, aceste operații se execută într-un timp de ordinul O(1), ele durează foarte mult. Practic, dacă am fi avut coordonate care ar putea varia între 1 şi n, ordinul de complexitate al algoritmului ar fi fost $O(n^3)$ deoarece, pentru fiecare pereche de coordonate, în cazul cel mai defavorabil, trebuie să parcurgem un număr de pătrățele care depinde de valoarea n.

Datele de ieşire constau în coordonatele a două colțuri opuse ale dreptunghiului, operație de scriere a acestora efectuându-se în timp constant.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(c) + O(1) + O(1) = O(c)$, dar acest rezultat nu este relevant datorită faptului că "ascunde" o constantă foarte mare.

2.3.17. Longhorn

Această problemă poate fi rezolvată folosind o strategie de tip *greedy*.

Pentru început vom crea o "listă de aşteptare" pentru fiecare aplicație, listă care va conține toate aplicațiile care trebuie instalate în prealabil pentru ca programul de instalare să poată fi lansat în execuție.

Este evident că vom începe instalarea tuturor aplicațiilor care nu necesită instalarea prealabilă a altor aplicații.

Apoi, la fiecare pas, vom determina toate aplicațiile al căror proces de instalare s-a încheiat şi le vom elimina din listele de aşteptare ale aplicațiilor care nu au fost încă instalate.

După aceea vom verifica dacă există aplicații pentru care listele de aşteptare sunt vide. Dacă găsim astfel de aplicații putem să lansăm în execuție programul de instalare corespunzător. La fiecare pas vom memora aplicațiile ale căror programe de instalare se află în execuție. În final vom afişa numărul de paşi parcurşi şi, corespunzător fiecărui pas, lista aplicațiilor al căror proces de instalare este în desfăşurare.

Analiza complexității

Datele de intrare constau în citirea, pentru fiecare aplicație, a aplicațiilor care trebuie să fie instalate în prealabil. În cazul cel mai defavorabil, prima aplicație nu va depinde de nici una, a doua va depinde de prima, a treia de primele două etc. Ca urmare, dimensiunea datelor de intrare poate creşte după o funcție pătratică, ordinul de complexitate al operației de citire a datelor fiind, aşadar, $O(n^2)$.

În continuare, la fiecare pas va fi instalată cel puţin o aplicaţie, deci vom avea cel mult *n* paşi. La fiecare pas va trebui să verificăm listele aplicaţiilor care trebuie să fie instalate în prealabil, liste a căror dimensiune totală poate avea ordinul $O(n^2)$ din motivele invocate şi în cazul operaţiei de citire. Ca urmare, operaţia de determinare a ordinii de instalare a aplicaţiilor are ordinul de complexitate $O(n^3)$.

Datele de ieşire sunt scrise pe măsura determinării aplicaţiilor care pot fi instalate, motiv pentru care nu se consumă timp suplimentar pentru scrierea lor.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n^2) + O(n^3) = O(n^3)$, dar acest rezultat nu este relevant datorită faptului că "ascunde" o constantă foarte mare.

2.4. Soluţiile problemelor propuse în ediţia 2001-2002

2.4.1. Dune

Problema poate fi rezolvată foarte uşor folosind algoritmul lui *Lee*. La fiecare pas vom memora toate poziţiile în care poate ajunge *Muad'Dib*.

Iniţial *Muad'Dib* se află în poziţia marcată cu '*'. Vom marca cu '#' toate regiunile în care se poate ajunge din poziţia iniţială mergând în direcţia indicată de primul element al şirului de caractere dat.

La fiecare dintre următorii paşi vom efectua următoarele operaţii:

- vom marca cu '.' toate poziţiile care nu sunt marcate cu '+' sau cu '#';
- vom marca cu '*' toate poziţiile care sunt marcate cu '#';
- vom marca cu '+' poziţia în care se află *sietch*-ul de plecare:
- pentru fiecare poziţie marcată cu '*':
  - vom marca cu '#' toate regiunile în care se poate ajunge din poziţia considerată mergând în direcţia indicată de elementul curent al şirului de caractere dat;
  - există posibilitatea să marcăm cu '#' poziţii care au fost marcate cu '*'; acest lucru nu afectează corectitudinea algoritmului, deoarece din poziţiile care au fost marcate cu '*' nu se poate ajunge în alte poziţii decât cele deja marcate cu '#'.

În final, soluţia va fi dată de numărul poziţiilor marcate cu '#'.

Aceasta este o prezentare teoretică a rezolvării. Practic, pentru simplitate, nu se mai face diferenţă între primul şi ceilalţi paşi ai algoritmului. Astfel, se iau în considerare toate poziţiile marcate cu '*', se marchează cu '#' toate poziţiile în care se poate ajunge, se marchează cu '.' toate poziţiile marcate cu '*', se marchează cu '+' poziţia *sietch*-ului de pornire şi se repetă aceşti paşi până când nu mai are loc nici o schimbare de direcţie. În final, se numără poziţiile marcate cu '*'.

Trebuie luat în considerare faptul că *sietch*-ul de pornire reprezintă o regiune stâncoasă.

Analiza complexității

Datele de intrare constau dintr-o matrice cu m linii şi n coloane, precum şi dintr-un şir al direcțiilor a cărui lungime o vom nota prin s. Ca urmare, ordinul de complexitate al operației de citire a datelor are ordinul de complexitatea $O(m \cdot n + s)$.

În continuare, la fiecare pas va trebui să parcurgem întreaga matrice pentru a determina pozițiile la care ar fi putut ajunge *Paul* la pasul anterior. La un astfel de pas vor fi marcate noile poziții în care poate ajunge *Paul*. În cel mai defavorabil caz vor fi marcate, în total, toate pozițiile. Ca urmare, ordinul de complexitate al operațiilor executate la un pas este $O(m \cdot n)$. Numărul paşilor va fi s, deci ordinul de complexitate al operației de determinare a configurației finale a matricei este $O(m \cdot n \cdot s)$.

Pentru determinarea şi afişarea rezultatului final va trebui să parcurgem configurația finală, operație al cărei ordin de complexitate este $O(m \cdot n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m \cdot n + s) + O(m \cdot n \cdot s) + O(m \cdot n) = O(m \cdot n \cdot s)$.

2.2.2. Matrice

Problema se rezolvă folosind metoda programării dinamice.

Evident, determinarea fiecărui element din matrice consumă prea mult timp, deci nu este o soluție viabilă. Rezolvarea propusă se bazează pe o tehnică recursivă.

Vom încerca să determinăm, pentru fiecare linie din M_7, numărul de elemente cu valoarea 1 aflate pe linia respectivă. Folosind această informație, linia pe care se află al X-lea element poate fi se găsită imediat (se parcurg liniile matricei M_7, se însumează valorile corespunzătoare primelor K linii şi, în cazul în care această sumă este mai mică decât X se trece la linia $K + 1$; dacă suma este mai mare sau egală cu X, atunci al X-lea element se află pe linia K).

Se pune problema determinării numărului de elemente cu valoarea 1 de pe o linie a matricei M_7. Linia respectivă este obținută prin expandarea unei linii a matricei M_6 (de fapt, prin această expandare se obțin N linii, linia respectivă fiind una dintre ele). Dacă se cunoaşte numărul de elemente cu valoarea 1 de pe linia corespunzătoare a matricei M_6, rezultatul cerut se determină uşor.

Pentru simplificarea formulelor vom numerota liniile şi coloanele începând cu 0; în final se vor incrementa cu 1 valorile obținute în urma calculelor.

Aşadar, dorim să determinăm numărul elementelor cu valoarea 1 de pe o linie L a matricei M_7. Pentru aceasta, trebuie să cunoaştem numărul elementelor cu valoarea 1 de pe linia $[L / N]$ a matricei M_6. Fiecare dintre aceste elemente se transformă în matricea A şi "contribuie" la construirea liniei L din M_7 prin intermediul liniei rest$[L / N]$ a matricei A. Similar, fiecare element cu valoarea 0 din M_6 se transformă în matricea B. Aşadar, rezultatul cerut este:

*numărul de elemente cu valoarea 1 de pe linia [L / N] a matricei M_6 **
numărul de elemente cu valoarea 1 de pe linia rest[L / N] din matricea A +
*numărul de elemente cu valoarea 0 de pe linia [L / N] a matricei M_6 **
numărul de elemente cu valoarea 1 de pe linia rest[L / N] din matricea B.

Pentru a determina numărul de elemente cu valoarea 1 de pe linia [*L* / *N*] a matricei M_6 vom aplica recursiv acelaşi procedeu, până ajungem la $M_1 = A$. Din nefericire, această metodă duce la repetarea unor calcule de foarte multe ori, prin apelarea funcţiei recursive cu aceiaşi parametri.

Pentru a rezolva acest impediment se memorează rezultatele într-o matrice *NR* de dimensiuni $6 \times N^6$ în care elementul NR_{ij} reprezintă numărul de elemente cu valoarea 1 de pe linia *j* a matricei M_i.

Rezultatele pentru M_7 nu sunt memorate, deoarece fiecare dintre acestea sunt calculate o singură dată. Fără această optimizare, matricea *NR* nu ar fi putut fi memorată în memoria convenţională.

După determinarea liniei pe care se află al *X*-lea element cu valoarea 1, determinarea coloanei este simplă şi poate fi realizată fie într-o manieră similară, fie pur şi simplu determinând pe rând fiecare element de pe linia respectivă a matricei M_7. Pe linia respectivă se află cel mult $5^7 = 78125$ elemente, deci numărul iteraţiilor nu ar fi foarte mare.

Analiza complexităţii

Datele de intrare constau într-o matrice cu *N* linii şi *N* coloane, aşadar ordinul de complexitate al operaţiei de citire a datelor are ordinul de complexitatea $O(N^2)$.

În continuare, la fiecare pas *i* va trebui să determinăm şi să memorăm elementele unei matrice cu N^6 linii şi şase coloane, operaţie al cărei ordin de complexitate este $O(N^6)$. După efectuarea acestei operaţii va trebui să calculăm şi valorile corespunzătoare pentru matricea M_7, operaţie al cărei ordin de complexitate este $O(N^7)$. Aşadar, operaţia de identificare a rezultatului problemei are ordinul de complexitate $O(N^7)$.

Datele de ieşire constau din scrierea a doar două valori, operaţie al cărei ordin de complexitate este $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N^2) + O(N^7) + O(1) = O(N^7)$.

2.4.3. Templu

Pentru simplitate, vom prezenta un enunţ echivalent al problemei: *se consideră un şir de numere binare; folosind acest şir se construieşte o matrice în care prima linie este dată de şirul considerat, iar fiecare dintre următoarele linii sunt obţinute prin permutarea la stânga cu o poziţie a elementelor din linia precedentă; în continuare, liniile matricei obţinute sunt sortate lexicografic (valoarea 0 precede valoarea 1); dându-se elementele de pe ultima coloană a matricei obţinute după sortare, să se determine elementele de pe prima linie a acestei matrice.*

Algoritmul constă în trei paşi:

- se numără elementele cu valoarea 0 şi elementele cu valoarea 1 de pe ultima coloană şi, pe baza rezultatului numărării, se construieşte prima coloană a matricei; în continuare vom presupune că avem p elemente cu valoarea 0 şi q elemente cu valoarea 1.
- se creează un şir a astfel:
  - pentru $i \le p$, elementul a_i va avea ca valoare linia pe care se află cel de-al i-lea element cu valoarea 0 de pe ultima coloană;
  - pentru $j \le q$, elementul a_{p+j} va avea ca valoare linia pe care se află cel de-al j-lea element cu valoarea 1 de pe ultima coloană;
- începând cu prima linie, se parcurg liniile folosind vectorul construit a (de la linia k se trece la linia a_k) şi sunt reconstituite elementele de pe prima linie pe baza valorilor de pe ultima coloană.

În cele ce urmează vom demonstra corectitudinea algoritmului prezentat. Numim *succesor* al unui şir, şirul obținut printr-o permutare cu o poziție la stânga a elementelor şirului.

Pentru a demonstra corectitudinea algoritmului, va trebui să arătăm că şirul a reprezintă exact această relație de succesiune între şiruri. După demonstrarea acestui fapt, este evident că al treilea pas al algoritmului reconstituie corect elementele de pe prima linie.

Considerăm liniile (din matricea obținută după sortare) care încep cu un element care are valoarea 0. Deoarece, aceste şiruri sunt ordonate în matricea sortată, după efectuarea unei permutări la stânga cu o poziție, şirurile obținute (care se termină cu un element care are valoarea 0) sunt şi ele sortate.

Aceeaşi proprietate este valabilă şi pentru liniile care încep cu un element care are valoarea 1.

Deoarece aceste proprietăți sunt echivalente cu relația de succesiune, corectitudinea algoritmului este demonstrată.

Analiza complexității

Vom demonstra în continuare că timpul de execuție al algoritmului este unul liniar. Pentru aceasta vom arăta că fiecare dintre cei trei paşi se efectuează în timp liniar.

Numărarea elementelor de pe ultima coloană se realizează printr-o singură parcurgere, deci liniaritatea este evidentă.

Inițializarea tabloului a necesită doar o singură parcurgere a acestuia şi a ultimei coloane, deci timpul este liniar şi în acest caz.

Cel de-al treilea pas se execută şi el în timp liniar deoarece la fiecare tranziție se determină câte un element de pe prima linie a matricei sortate.

Evident, operațiile de citire şi de scriere se efectuează, ambele, în timp liniar.

2.4.4. Cub

Considerăm, pe rând, literele cuvântului care trebuie căutat (fie acesta *s*). La fiecare pas *x* vom putea determina numărul de apariții al cuvântului format din primele *x* caractere ale cuvântului dat.

Pentru aceasta, la fiecare pas vom calcula valorile elementelor a_{ijk}, ale unui tablou tridimensional *A*, unde a_{ijk} indică numărul de apariții ale cuvântului format din primele *x* litere ale şirului dat, astfel încât elementul din cubulețul de coordonate (*i*, *j*, *k*) a cubului să fie s_x.

Pentru toate căsuțele care nu conțin litera s_x valoarea a_{ijk} va fi 0. Evident, la primul pas vom avea $a_{ijk} = 1$ pentru toate pozițiile pe care se găseşte valoarea s_1 şi $a_{ijk} = 0$ pentru celelalte poziții. La următorul pas, pentru fiecare element care conține valoarea s_2 vom număra elementele vecine care au valoarea 1 şi vom atribui elementului corespunzător valoarea obținută. Practic, se poate spune că vom aduna toate valorile elementelor vecine. La al treilea pas, pentru fiecare element care conține valoarea s_3 vom calcula valoarea corespunzătoare însumând valorile elementelor vecine care corespund unor cubulețe în care se află litera s_2.

Se observă că, practic, şi la al doilea pas se poate realiza o operație similară: pentru toate elementele corespunzătoare unui cubuleț în care se află litera s_2, se însumează valorile elementelor vecine care corespund unor cubulețe care conțin litera s_1.

În general, pentru toate elementele corespunzătoare unui cubuleț în care se află litera s_i, se însumează valorile elementelor vecine care corespund unor cubulețe care conțin litera s_{i-1} (pentru $i > 1$).

Soluția este dată de suma elementelor corespunzătoare cubulețelor în care se află litera s_n, unde *n* este lungimea cuvântului dat.

Datorită faptului că literele cuvântului sunt distincte, nu apar probleme suplimentare la calcularea valorilor elementelor. Dacă literele s-ar fi putut repeta, în cazul a două litere consecutive identice, ar fi putut apărea erori. Soluția ar fi fost dată de folosirea unui tablou tridimensional suplimentar care ar fi păstrat valorile calculate la iterația anterioară.

Datorită faptului că numărul de apariții ale cuvântului este de cel mult 2000000000, fiecare element al tabloului tridimensional necesită patru bytes pentru a fi reprezentat. Dimensiunea maximă a laturii cubului este 39, deci cantitatea de memorie necesară pentru a păstra elementele este de $39^3 \cdot 4 = 237276$ bytes ≈ 232 KB.

Pentru a păstra literele din cubulețe avem nevoie de alți $39^3 = 59319$ bytes ≈ 58 KB. Aşadar, dimensiunea totală a structurilor de date pe care le folosim este de aproximativ 290 KB.

Analiza complexității

Datele de intrare constau dintr-un tablou tridimensional care are N^3 elemente şi dintr-un cuvânt a cărui lungime este notată prin *S*. Ca urmare, ordinul de complexitate al operației de citire a acestora este $O(N^3 + S)$.

Numărul de parcurgeri ale tabloului tridimensional este egal cu lungimea S a cuvântului dat. Ordinul de complexitate al unei parcurgeri este $O(N^3)$ deoarece se parcurge întregul tablou tridimensional. Ca urmare, ordinul de complexitate al operației de determinare a configurației finale a tabloului A este $O(S \cdot N^3)$.

Pentru a afişa rezultatul va trebui să însumăm valorile corespunzătoare din tabloul A, operație care implică din nou parcurgerea întregului tablou, deci are ordinul de complexitate $O(N^3)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N^3 + S) + O(N^3 \cdot S) + O(N^3) = O(N^3 \cdot S)$.

2.4.5. Arbori

Algoritmul care trebuie aplicat pentru rezolvarea problemei constă din doi paşi. La început se va determina costul transformării unui arbore din curte în fiecare dintre arborii din poză. Acest algoritm este destul de simplu, el neridicând dificultăți deosebite. La al doilea pas se va aplica un algoritm de determinare a unui cuplaj maxim de cost minim pentru un graf construit pe baza costurilor determinate la primul pas.

Primul pas care trebuie realizat în rezolvarea problemei este calcularea, pentru fiecare pereche de forma *arbore_din_curte – arbore_din_poză*, a costului (numărul de crengi tăiate) necesar transformării.

Pentru aceasta se poate folosi o funcție recursivă care returnează valoarea -1 dacă transformarea nu este posibilă (arborele din poză conține crengi în anumite poziții în care arborele din grădină nu conține crengi) sau costul cerut, calculat în funcție de costurile necesare transformării subarborilor stâng şi drept (dacă oricare dintre ei este vid pentru arborele din poză şi nevid pentru cel din curte, se adaugă valoarea 1 la costul corespunzător celuilalt subarbore şi nu se mai efectuează apelul recursiv pe ramura respectivă). Este esențial să începem cu această operație, deoarece nu se doreşte recalcularea acestor costuri de fiecare dată când avem nevoie de ele.

Rezultatul va fi o matrice C; aceasta poate fi privită ca fiind un graf bipartit; partițiile nodurilor sunt "arborii din curte" şi "arborii din poză". Problema cere să asociem în mod biunivoc cât mai multor noduri din prima partiție, câte un nod din a doua partiție, astfel încât costul total să fie minim.

Aşadar, problema se reduce la aplicarea unui algoritm pentru determinarea unui cuplaj maximal de cost minim, care va fi rezolvată conform algoritmilor cunoscuți.

Graful se poate transforma într-o rețea de transport prin introducerea unui nod sursă (conectat cu arborii din curte) şi a unui nod destinație (conectat cu arborii din poză).

Toate arcele din graful bipartit inițial au capacitatea 1, iar sensul este dinspre sursă spre destinație. Costul arcelor introduse de nodurile sursă şi destinație este 0.

Trebuie să determinăm fluxul maxim de cost minim în această rețea. După cum se ştie, fluxul maxim se determină găsind repetat drumuri de creştere, de la sursă la destinație. Un drum de creştere poate să conțină un arc (i, j), dacă nu există flux pe el, sau dacă există flux pe arcul (j, i) - deci se parcurge arcul (j, i) în sens invers.

Se caută un astfel de drum de la sursă la destinație, se actualizează fluxul pe arcele care formează drumul (devine 1 dacă valoarea a fost 0 şi 0 dacă valoarea a fost 1), se caută un nou drum de creştere etc.

Operația se repetă până în momentul în care nu mai există nici un drum de creştere. Inițial se va considera că fluxurile corespunzătoare tuturor arcelor din graf au valoarea 0.

Pot exista mai multe drumuri de creştere la un moment dat. În această situație va trebui ales drumul de creştere de cost minim.

Costul unui drum de creştere este dat de suma costurilor arcelor componente. Costul unui arc este pozitiv dacă valoarea fluxului de pe acel arc este 0 şi negativ în cazul în care fluxul de pe acel arc are valoarea 1. Valoarea absolută a acestui flux este dată de costul determinat la pasul anterior.

În continuare se va construi graful format din arcele corespunzătoare (cele pentru care valoarea fluxului este 1 sunt inversate şi vor avea cost negativ; celelalte arce sunt păstrate fără a fi modificate). În acest graf, pe baza algoritmului *Belmann-Ford*, se va determina un astfel de drum de cost minim. Algoritmul *Belmann-Ford* va funcționa corect, deoarece graful nu va conține cicluri cu cost negativ. În final se obține un flux maxim de cost minim; arcele pe care fluxul are valoarea 1 constituie soluția problemei.

Analiza complexității

Pentru a analiza complexitatea algoritmului vom stabili, mai întâi, faptul că numărul nodurilor unui arbore poate fi considerat constant, deoarece este cel mult 100.

În aceste condiții, datele de intrare conțin informații referitoare la $M + N$ arbori, deci operația de citire a acestora are ordinul de complexitate $O(M + N)$.

În continuare vor fi stabilite costurile muchiilor grafului bipartit. Pentru fiecare muchie vom realiza parcurgeri ale arborilor corespunzători, operații care se realizează în timp constant. Vom determina cel mult $M \cdot N$ muchii, aşadar ordinul de complexitate va fi $O(M \cdot N)$.

Vom determina apoi un cuplaj bipartit folosind un algoritm de determinarea a unui flux maxim de cost minim. Avem o rețea cu $M + N + 2$ noduri (N pentru arborii din poză, M pentru arborii din grădină, sursa şi destinația) şi cel mult $M \cdot N + M + N$ arce ($M \cdot N$ muchii determinate şi $M + N$ muchii introduse pentru a lega nodurile corespunzătoare arborilor de sursă sau destinație).

Valoarea fluxului maxim va fi cel mult egal cu minimul valorilor M şi N. Dacă utilizăm algoritmul *Ford-Fulkerson* pentru a determina cuplajul, împreună cu algoritmul *Bellman-Ford* pentru determinarea drumurilor de creştere, ordinul de complexitate al operației este $O(\min(M, N) \cdot (M + N + 2) \cdot (M \cdot N + M + N)) = O(\min(M, N) \cdot (M + N) \cdot M \cdot N)$.

Pentru a identifica muchiile pe care fluxul are valoarea 1 (şi a scrie, în acelaşi timp, datele de ieşire) va trebuie să parcurgem toate muchiile grafului bipartit, operație al cărei ordin de complexitate este $O(M \cdot N)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(M + N) + O(M \cdot N) + O(\min(M, N) \cdot (M + N) \cdot M \cdot N) + O(M \cdot N) = O(\min(M, N) \cdot (M + N) \cdot M \cdot N)$.

Observație
Constanta care se ascunde în spatele ordinelor de complexitate ale operațiilor de citire a datelor şi determinare a grafului bipartit este foarte mare, deoarece arborii pot avea până la 100 de noduri. Cu toate acestea, termenii corespunzători acestor operații în formula pentru stabilirea complexității globale sunt nesemnificativi față de termenul corespunzător operației de determinare a cuplajului.

2.4.6. Atlantis

În principiu, problema ar putea fi enunțată astfel: *dându-se un număr întreg să se determine cel mai mic multiplu al său care este format doar din anumite cifre date*.

Soluția acestei probleme este destul de simplă şi se bazează pe câteva observații matematice.

Pentru început vom demonstra că, dacă există un multiplu al numărului *Nr* care este format doar din cifrele $x_1, x_2, ..., x_m$, atunci cel mai mic astfel de multiplu are cel mult *Nr* cifre.

Vom presupune, prin absurd, că cel mai mic multiplu *M* are *Nr* + 1 cifre. Eliminând ultima cifră, vom obține un număr care, împărțit la *Nr*, duce la obținerea unui rest R_1. Repetând procesul de eliminare a cifrelor, se obțin resturile $R_2, ..., R_{Nr}$.

Dacă cele *Nr* resturi sunt distincte, atunci unul dintre ele este 0, deci există un multiplu care conține mai puțin de *Nr* + 1 cifre (cel mult *Nr* cifre).

Dacă resturile nu sunt distincte, atunci cel puțin două dintre ele (fie acestea R_i şi R_j, $i < j$) sunt egale. Multiplul *M* are forma $m_1...m_i...m_j...m_{Nr+1}$; datorită faptului că resturile R_i şi R_j sunt egale, înseamnă că prin eliminarea cifrelor $m_{i+1}, ..., m_j$ se obține un număr care este, la rândul său, multiplu al numărului *Nr*.

Acest număr are forma $m_1...m_i m_{j+1}...m_{Nr+1}$, deci are cel mult *Nr* cifre.

Putem trage concluzia că ipoteza inițială este falsă; aşadar, dacă există un multiplu al numărului *Nr* care să fie format din anumite cifre date, atunci cel mai mic astfel de multiplu conține cel mult *Nr* cifre.

Ipoteza potrivit căreia multiplul ar fi format din *Nr* + 1 cifre nu reduce generalitatea deoarece, dacă presupunem că numărul ar fi format din *Nr* + *k* cifre, folosind acelaşi procedeu putem obține, pe rând, multiplii formați din cel mult *Nr* + *k* – 1 cifre, *Nr* + *k* – 2 cifre şi aşa mai departe până se ajunge la un multiplu format din cel mult *Nr* + 1 cifre.

În continuare vom descrie modul în care poate fi determinat un multiplu format din cel mult *Nr* cifre. Vom crea o coadă care va conține numere prin împărțirea cărora la *Nr* se obțin resturi distincte. Corespunzător fiecărui rest care poate fi obținut, în coadă se va afla cel mai mic număr care duce la obținerea restului respectiv. Inițial vom inse-

ra în coadă numere formate dintr-o singură cifră (cifrele care le avem la dispoziție), ordonate crescător. La fiecare pas, vom alege primul element din coadă şi vom încerca să adăugăm la sfârşitul său, pe rând, una dintre cifrele disponibile. Pentru a obține cele mai mici numere, cifrele vor fi considerate în ordine crescătoare.

În cazul în care un număr astfel determinat duce, prin împărțirea la *Nr*, la obținerea unui rest care nu a mai fost obținut anterior, atunci numărul determinat este adăugat în coadă.

După considerarea tuturor cifrelor, elementul curent este eliminat din coadă. Datorită acestor eliminări trebuie păstrat un vector de valori booleene care să indice dacă un rest a fost sau nu obținut anterior.

În momentul în care se obține restul 0, cel mai mic multiplu este determinat şi execuția algoritmului se opreşte.

Noile resturi pot fi determinate foarte repede folosind o altă observație matematică: dacă restul împărțirii unui număr *a* la *Nr* este *r*, atunci restul împărțirii la *Nr* a numărului obținut prin adăugarea cifrei *x* la sfârşitul numărului *a* este rest[$(r \cdot 10 + x) / Nr$].

În cazul în care nu am reuşit să obținem restul 0 considerând numere formate din cel mult *Nr* cifre, atunci suntem siguri că nu există nici un multiplu format doar din cifrele date deoarece, dacă astfel de multiplii ar fi existat, atunci cel puțin unul dintre ei ar fi avut cel mult *Nr* cifre.

Analiza complexității

Datele de intrare constau din numărul *Nr* de pe mecanism (care are întotdeauna patru cifre, chiar dacă este posibil ca primele să fie 0) şi cele *N* cifre pe care le avem la dispoziție. Ca urmare, ordinul de complexitate al operației de citire a datelor este $O(N)$.

În continuare, pentru fiecare rest vom încerca să adăugăm cele *N* cifre. Aşadar, avem ordinul de complexitate $O(N)$ pentru fiecare rest. Datorită faptului că numărul resturilor considerate este de cel mult *Nr*, ordinul de complexitate al operației de determinare a multiplului este $O(Nr \cdot N)$.

Datele de ieşire constau în cele cel mult *Nr* cifre ale multiplului sau doar în cifra 0. Ca urmare, ordinul de complexitate al operației de scriere a datelor de ieşire este $O(Nr)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N) + O(Nr \cdot N) + O(Nr) = O(Nr \cdot N)$.

2.4.7. Celule

Pentru început, vom studia modul în care se propagă semnalul în cazul în care şirul are o lungime infinită şi, inițial, avem o singură celulă excitată. Vom considera că celula excitată se află pe poziția 0.

```
-9 -8 -7 -6 -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5 +6 +7 +8 +9
 0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0  1  0  1  0  1  0  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0
 0  0  0  0  1  0  1  0  0  0  0  0  1  0  1  0  0  0  0
 0  0  0  1  0  0  0  1  0  0  0  1  0  0  0  1  0  0  0
 0  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  0
 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0
 1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1
```

Se observă că, dacă momentul inițial este considerat a fi $t = 0$, atunci la momentele de timp de forma $t = 2^k$ vom avea celule excitate doar pe pozițiile 2^{k+1} şi -2^{k+1}.

Vom considera acum un şir de celule excitate şi liniştite; prima celulă a şirului se va afla pe poziția 0, a doua pe poziția 1 etc. Se observă că, la fiecare pas, configurația se obține aplicând operatorul **`xor`** (sau exclusiv) între configurațiile care ar fi fost obținute pentru fiecare celulă în parte, luând în considerare poziția inițială a acesteia. Astfel, dacă inițial celula excitată se află pe poziția n, atunci la momente de timp de forma $t = 2^k$ vom avea celule excitate doar pe pozițiile $n + 2^{k+1}$ şi $n - 2^{k+1}$.

Observăm că, la momente de timp de forma $t = 2^k$ vom obține configurația inițială pe şirurile de celule care încep din pozițiile 2^{k+1} şi -2^{k+1}.

De exemplu, pentru şirul de celule 111 vom obține următoarele configurații:

```
-9 -8 -7 -6 -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5 +6 +7 +8 +9
 0  0  0  0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  1  0  1  1  0  0  0  0  0  0
 0  0  0  0  0  0  0  1  1  1  0  1  1  1  0  0  0  0  0
 0  0  0  0  0  0  1  1  0  1  0  1  0  1  1  0  0  0  0
 0  0  0  0  0  1  1  1  0  0  0  0  0  1  1  1  0  0  0
 0  0  0  0  1  1  0  1  1  0  0  0  1  1  0  1  1  0  0
```

Datorită faptului că lungimea L a şirului de celule este finită, rezultă că o anumită configurație dată va ajunge la momente de timp de forma $t = 2^k$ pe poziții de forma rest$[2^{k+1} / L]$ şi rest$[-2^{k+1} / L]$.

Evident, pentru ca celulele să nu se liniştească, trebuie să existe o perioadă de repetare a configurațiilor şi aceasta nu poate fi mai lungă decât 2^L.

Aşadar, este suficient să verificăm dacă la momentul de timp $t = 2^L$ mai există celule excitate.

La acest moment de timp, şirul inițial se va regăsi în poziții de forma rest$[2^{L+1} / L]$ şi rest$[-2^{L+1} / L]$. Aplicând operatorul **`xor`** asupra celor două configurații vom obține un şir care ar putea să conțină celule excitate.

În cazul în care avem celule excitate, rezultă că excitația se va menține la infinit. Dacă celulele s-au liniştit, atunci este evident faptul că nu se va menține la infinit excitația.

O altă modalitate (mai puțin eficientă) de abordare a problemei era generarea configurațiilor la fiecare pas. În cazul în care celulele se liniştesc este evident că excitația nu se menține. Astfel, există şanse destul de mari ca, dacă celulele nu se liniştesc după un anumit timp, excitația să se păstreze la infinit. O altă abordare ar fi fost căutarea perioadei după care se repetă configurațiile. Astfel, pentru fiecare configurație nouă se va verifica dacă există o configurație identică obținută anterior. În caz afirmativ, se poate trage concluzia că excitația se menține la infinit.

Analiza complexității

Datele de intrare constau din numărul M al configurațiilor şi stările celulelor care fac parte dintr-o configurație. Vom nota cu S suma lungimilor tuturor configurațiilor. În acest caz ordinul de complexitate al operației de citire va fi $O(S)$.

Vom analiza acum ordinul de complexitate al operației de verificare pentru o configurație de lungime L. Va trebuie doar să aplicăm operatorul `xor` asupra a două configurații de lungime L şi apoi să verificăm dacă au rămas celule excitate. Aşadar, ordinul de complexitate al operației de verificare este $O(L)$.

Datorită faptului că vom verifica toate cele M configurații, iar lungimea totală a acestora este S, operația de verificare a tuturor celor M configurații are ordinul de complexitate $O(S)$.

Datele de ieşire pot fi scrise pe parcursul efectuării verificărilor, deci nu se consumă timp suplimentar pentru această operație.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(L) + O(S) = O(S)$.

2.4.8. Fibonacci

Această problemă nu ridică dificultăți deosebite; ea poate fi rezolvată folosind o simplă formulă matematică.

Se observă că $F_6 = 8$ se divide cu 2^3, $F_{12} = 144$ se divide cu 2^4 şi $F_{24} = 46368$ se divide cu 2^5. De aici rezultă o regulă destul de clară: indicele elementului şirului lui *Fibonacci* divizibil cu 2^3 este $6 = 3 \cdot 2$, cel al elementului divizibil cu 24 este $12 = 3 \cdot 2^2$, iar cel al elementului divizibil cu 2^5 este $24 = 3 \cdot 23$. Putem presupune că elementul şirului lui *Fibonacci* divizibil cu 2^n are indicele $3 \cdot 2^{n-2}$.

În cele ce urmează vom demonstra că această afirmație este adevărată. Pentru aceasta vom folosi următoarea proprietate a şirului lui *Fibonacci*: elementul $F_{k \cdot n}$ este divizibil cu elementul F_n. Această proprietate poate fi demonstrată foarte uşor, folosind expresia generală a termenilor şirului:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

Pe baza unor calcule rezultă imediat proprietatea amintită. Utilizând acest rezultat matematic, vom aplica metoda inducției matematice pentru a demonstra că elementul şirului lui *Fibonacci* divizibil cu 2^n are indicele $3 \cdot 2^{n-2}$.

Ipoteza de inducție va fi: elementul şirului lui *Fibonacci* divizibil cu 2^{n-1} are indicele $3 \cdot 2^{n-3}$.

Pentru $n = 4$ ipoteza de inducție poate fi verificată foarte uşor, deoarece elementul cu indicele $3 \cdot 2^{4-3} = 6$ este 8 şi acesta este divizibil cu $2^{4-1} = 8$.

Folosind ipoteza de inducție vom demonstra afirmația inițială. Datorită faptului că $F_{2 \cdot n}$ este divizibil cu elementul F_n, elementul cu indicele $3 \cdot 2^{n-2}$ va fi divizibil cu 2^{n-1}.

Mai mult, raportul dintre elementul cu indicele $3 \cdot 2^n$ şi cel cu indicele $3 \cdot 2^{n-1}$ este întotdeauna un număr par (proprietatea se poate verifica tot pe baza expresiei generale a termenilor şirului), deci elementul cu indicele $3 \cdot 2^{n-2}$ va fi divizibil cu $2 \cdot 2^{n-1} = 2^n$.

Aşadar, am arătat că există un element al şirului lui *Fibonacci* divizibil cu 2^n care are indicele $3 \cdot 2^{n-2}$. Mai trebuie arătat că acest indice este mai mic decât 2^n. Este evident că $3 \cdot 2^{n-2} < 4 \cdot 2^{n-2} = 2^n$ pentru $n > 2$. Aşadar, pentru $n > 2$, soluția problemei este dată de expresia $3 \cdot 2^{n-2}$.

Datorită faptului că valoarea n poate ajunge până la 10000, va trebui să implementăm operații cu numere mari pentru a rezolva această problemă.

Analiza complexității

Din fişierul de intrare se va citi doar valoarea n, ordinul de complexitate al acestei operații fiind $O(1)$.

Pentru a calcula valoarea $3 \cdot 2^{n-2}$ va trebui să efectuăm $n - 1$ înmulțiri. Am putea să limităm numărul acestora folosind metoda de ridicare la putere care are ordinul de complexitate $O(\log n)$ dar, în acest caz ar fi trebuit să realizăm înmulțiri între numere mari. Dacă se utilizează metoda clasică, nu vom avea decât înmulțiri cu 2 (dacă alegem valoarea inițială 3), care sunt efectuate mult mai rapid. Obținem astfel ordinul de complexitate $O(n)$. Practic, datorită faptului că valoarea $O(\log n)$ ascunde o constantă foarte mare nu are rost să ne complicăm cu implementarea operației de înmulțire a două numere mari.

Datele de ieşire constau într-un singur număr, ordinul de complexitate al operației de scriere a acestuia fiind $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(n) + O(1) = O(n)$.

2.4.9. Atlantis2

Soluția acestei probleme este foarte uşor de găsit, fiind suficientă o simplă observație matematică.

Pentru început, vom arăta că există o secvență de patru semne a cărei prezență va duce la creşterea rezultatului cu 4. Vom presupune că există o secvență de m semne care duce la obținerea rezultatului R. Dacă adăugăm cele patru semne, vom obține o secvență de $m + 4$ semne care va duce la obținerea rezultatului $R + 4$. Secvența respectivă este:

+−−+

Se observă că, prin adăugarea acestei secvențe, valoarea rezultatului va creşte cu $m2 – (m + 1)^2 – (m + 2)^2 + (m + 3)^2$.

Efectuând calculele obținem:

$$m^2 – (m^2 + 2 \cdot m + 1) – (m^2 + 4 \cdot m + 4) + (m^2 + 6 \cdot m + 9) =$$
$$m^2 – m^2 – 2 \cdot m – 1 – m^2 – 4 \cdot m – 4 + m^2 + 6 \cdot m + 9 =$$
$$= –1 – 4 + 9 = 4.$$

Aşadar, prin adăugarea secvenței +−−+ la sfârşitul unei secvențe date, valoarea rezultatului creşte cu 4.

Se observă că, dacă dorim să obținem o valoare care este multiplu de 4 (are forma $4 \cdot k$), va trebui să folosim k secvențe de această formă.

În cele ce urmează vom arăta modul în care pot fi obținute secvențe de semne care să ducă la rezultate de forma $4 \cdot k + 1$, $4 \cdot k + 2$ şi $4 \cdot k + 3$.

Se observă că, pentru a obține secvențe care au forma $4 \cdot k$ + p, va trebui să adăugăm k secvențe +−−+ la secvențele care duc la obținerea rezultatului p.

Ca urmare, va trebui să căutăm secvențe de semne care să ducă la obținerea rezultatelor 1, 2 şi 3.

Pentru a obține rezultatul 1, secvența constă dintr-un singur semn +. O secvență formată doar din acest semn duce la obținerea rezultatului $0 + 1^2 = 1$. Aşadar, dacă se caută rezultate de forma $4 \cdot k + 1$, secvențele de semne au următoarea structură: ++−−+...+−−+.

Pentru a obține rezultatul 2, secvența constă din patru semne: trei semne −, urmate de un semn +. O secvență formată doar din aceste semne duce la obținerea rezultatului $0 – 1^2 – 2^2 – 3^2 + 4^2 = 0 – 1 – 4 – 9 + 16 = 2$. Aşadar, dacă se caută rezultate de forma $4 \cdot k + 2$, secvențele de semne au următoarea structură: −−−++−−+...+−−+.

Pentru a obține rezultatul 3, secvența constă din două semne: un semn −, urmat de un semn +. O secvență formată doar din aceste semne duce la obținerea rezultatului $0 - 1^2 + 2^2 = 0 – 1 + 4 = 3$. Aşadar, dacă se caută rezultate de forma $4 \cdot k + 3$, secvențele de semne au următoarea structură: −++−−+...+−−+.

Analiza complexității

Datele de intrare constau dintr-un singur număr, ordinul de complexitate al operației de citire a acestora fiind $O(1)$.

După citire vom trece direct la scrierea rezultatului. Vom determina restul împărțirii la 4 a valorii *n* şi vom scrie secvența de început corespunzătoare, operația care se realizează în timp constant. În continuare vom scrie secvențele +−−+ care vor duce la obținerea rezultatului corect. Ordinul de complexitate al acestei operații este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(n) = O(n)$.

2.4.10. Conducte

Putem privi sistemul de rafinării şi benzinării ca fiind un graf neorientat complet ale cărui noduri reprezintă obiectivele (benzinării sau rafinării), iar costurile muchiilor sunt date de distanțele (lungimile conductelor) care leagă două obiective. Evident, nu putem uni printr-o conductă două rafinării, aşadar vom considera că muchiile dintre rafinării au costul 0 (nu sunt necesare conducte pentru ca benzina să ajungă de la o rafinărie la alta).

Acum, problema se reduce la a determina un arbore parțial de cost minim în graful construit. Pentru aceasta vom folosi algoritmul lui *Prim*; în final vom afişa muchiile care fac parte din arborele parțial minim, dar nu au ca extremități două noduri care corespund unor rafinării.

Datorită faptului că putem avea până la 1000 de benzinării nu vom putea reprezenta în memorie graful folosind o matrice de adiacență. De asemenea, nu este posibilă păstrarea unei liste de muchii deoarece numărul acestora este prea mare. De aceea, vom păstra doar coordonatele benzinăriilor şi rafinăriilor şi vom calcula costurile muchiilor (distanțele dintre obiective) în momentul în care avem nevoie de aceste informații. Din acest motiv, nu poate fi folosit algoritmul lui *Kruskal* de determinare a arborelui parțial minim pentru că acesta implică sortarea tuturor muchiilor, deci păstrarea acestora în memorie.

Analiza complexității

Datele de intrare constau în coordonatele rafinăriilor şi benzinăriilor, deci operația de citire a acestora are ordinul de complexitate $O(n) + O(k) = O(n + k)$.

Este obligatoriu ca nodurile corespunzătoare rafinăriilor să fie legate între ele în arborele parțial minim folosind muchii de cost 0, operație care se realizează în timp liniar (ordinul de complexitate este $O(k)$). Pentru celelalte noduri vom folosi algoritmul lui *Prim*, ordinul de complexitate al acestuia fiind $O(n^2)$. Aşadar, determinarea arborelui parțial minim are ordinul de complexitate $O(n^2)+ O(k) = O(n^2 + k)$.

Datele de ieşire constau în cele *n* muchii care leagă benzinăriile între ele sau de rafinării, deci ordinul de complexitate al operației de scriere a acestora este $O(n)$.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(n + k) + O(n^2 + k) + O(n) = O(n^2 + k)$.

2.4.11. Dune2

Pentru rezolvarea problemei vom folosi metoda *programării dinamice*. Considerăm matricea de intrare *X*. Se observă că există cel mult 64 de modalități de a plasa bombe (mai exact colțurile din stânga-sus ale secțiunilor pătratice afectate de acestea) pe o linie a matricei. Valoarea 64 se obține datorită faptului că secțiunile trebuie să fie incluse integral în matrice (bombele nu pot atinge zone din afara *Câmpiei Arakeenului*), deci colțul din stânga-sus al unui pătrat poate fi plasat numai pe una dintre primele şase coloane. De fapt, acest număr poate fi redus la 37, deoarece unele posibilități de plasare sunt inutile.

De exemplu, se observă că patru bombe sunt suficiente pentru a acoperi integral o linie, deci nu are sens să plasăm cinci bombe pe o linie, deoarece cel puțin una dintre ele va afecta poziții atinse şi de alte bombe, deci bomba poate fi folosită în alte zone, valoarea diferenței dintre numărul *harkonnenilor* şi cel al *atreizilor* ucişi rămânând aceeaşi. De asemenea, nu are rost să plasăm colțurile din stânga-sus a zonelor afectate de trei bombe în trei coloane consecutive, deoarece pozițiile afectate de bomba din mijloc sunt afectate de celelalte două bombe.

Presupunem că ştim, pentru o anumită linie, sumele maxime care pot fi obținute dacă plasăm 0, 1, 2, ..., *K* bombe pe primele *i* linii, folosind pe linia *i* o anumită configurație. Apare întrebarea dacă putem determina aceleaşi informații pentru linia *i* + 1.

Răspunsul este afirmativ şi ne oferă rezolvarea problemei. Construim (cel puțin teoretic) tabloul tridimensional *M*[1 .. *N* - 1, 0 .. *K*, 1 .. 37]. Un element *M*[*i*, *j*, l] reprezintă suma maximă obținută dacă plasăm *j* bombe pe primele *i* linii, folosind configurația l pe linia *i*.

Elementele matricei *M*[*i* + 1] pot fi determinate folosind matricea *M*[*i*]. Pentru a calcula un element al matricei *M*[*i* + 1] avem nevoie de următoarele informații:

- suma valorilor de pe linia *i* + 2 corespunzătoare unor zone afectate de bombe care vor fi plasate pe linia *i* + 1, folosind configurația l;
- numărul *nr*[l] al bombelor care sunt folosite pentru obținerea configurației l;
- suma valorilor de pe linia *i* + 1 care corespund unor zone afectate dacă se foloseşte configurația l, dar nu şi folosind o configurație *c* (determinăm valorile sumei pentru orice configurație *c*);
- valoarea maximă din tabloul *M* pentru fiecare configurație *c* plasată pe linia *i* care conține *j* – *nr*[l] bombe.

Programul este construit astfel:

- configurațiile posibile de pe o linie sunt păstrate în matricea de constante *p*, iar numărul de bombe folosit pentru obținerea fiecărei configurații este păstrat în vectorul *nr*;
- tabloul tridimensional *M* este simulat cu ajutorul a două matrice *A* şi *B* (adică *A* = *M*[*i*], se calculează *B* = *M*[*i* + 1], apoi *A* devine *B* etc.)

- pentru fiecare rând unde se pot plasa bombe:
  - se calculează suma adăugată pe rândul respectiv dacă plasăm o anumită configurație, în funcție de configurația plasată pe rândul anterior;
  - se calculează suma adăugată pe rândul următor dacă plasăm o anumită configurație pe rândul curent;
  - se inițializează matricea B;
  - se realizează calculul valorilor din B (datorită faptului că operațiile din această zonă sunt executate de foarte multe ori, codul trebuie optimizat cât mai mult posibil;
  - A devine B.

Soluția este dată de elementul maxim obținut în final în matricea A.

Analiza complexității

Citirea unei linii a matricei care reprezintă *Câmpia Arakeenului* se realizează în timp constant, deoarece matricea are întotdeauna şapte coloane. Datorită faptului că trebuie citite n linii, ordinul de complexitate al operației de citire este $O(1) \cdot O(n) = O(n)$.

Matricele A şi B au, fiecare, k linii şi 37 de coloane. Operația de determinare a elementelor matricei B, folosind matricea A, are ordinul de complexitate $O(k)$, deoarece numărul operațiilor efectuate pentru determinarea uneia dintre linii este $O(37) \cdot O(37) \cdot O(7) = 37 \cdot O(1) \cdot 37 \cdot O(1) \cdot 7 \cdot O(1) = 9583 \cdot O(1) = O(1)$. Pentru fiecare dintre cele n linii ale matricei care caracterizează câmpia, se vor calcula elementele matricei B, deci ordinul de complexitate al operației de determinare a valorilor finale ale elementelor matricei B este $O(n) \cdot O(k) = O(n \cdot k)$.

Pentru afişarea diferenței maxime trebuie parcursă matricea finală, deci această operație are ordinul de complexitate $O(k)$.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(n) + O(n \cdot k) + O(k) = O(n \cdot k)$.

2.4.12. Frăția Inelului

Vom construi un graf orientat ale cărui noduri reprezintă reperele şi ale cărui arce reprezintă drumurile care leagă două repere. Sensul unui arc va fi de la reperul de la care porneşte drumul la reperul la care ajunge drumul. Datorită faptului că nu se poate ajunge de la un reper la unul anterior, deducem că graful este aciclic.

Acum, problema se reduce la determinarea numărului de drumuri din acest graf de la nodul care reprezintă reperul `VALCEAUADESPICATA` la nodul care reprezintă reperul `MUNTELEOSANDEI`.

Pentru simplitate vom codifica reperele prin numere cuprinse între 0 şi $N - 1$, unde N reprezintă numărul total al reperelor.

Pentru a determina numărul drumurilor dintre două noduri ale unui graf orientat vom folosi metoda *programării dinamice*. Vom construi un şir care reprezintă numărul de posibilități de a ajunge la un anumit reper (numărul de drumuri care pornesc din VALCEAUADESPICATA şi ajung la reperul respectiv).

Pentru un anumit nod vom putea determina numărul de posibilități dacă se cunoaşte numărul de posibilități în care se poate ajunge la fiecare dintre predecesorii săi din graf. Dacă toate aceste informații sunt cunoscute, atunci numărul posibilităților de a ajunge la acest reper va fi dat de suma numerelor posibilităților de a ajunge la reperele corespunzătoare predecesorilor.

Inițial considerăm că există o singură posibilitate de a ajunge la VALCEAUADESPICATA. Datorită aciclicității grafului suntem siguri că, la fiecare pas, vom determina cel puțin un nod pentru care cunoaştem valorile corespunzătoare tuturor predecesorilor. Aşadar, la fiecare pas, vom verifica pentru toate nodurile pentru care nu cunoaştem numărul posibilităților, dacă sunt cunoscute numerele posibilităților corespunzătoare predecesorilor. Pentru fiecare nod pentru care sunt cunoscute aceste informații vom calcula valoarea corespunzătoare. Datorită faptului că la fiecare pas vom determina cel puțin un astfel de nod, suntem siguri că algoritmul se va termina după un număr finit de paşi (cel mult $N - 1$).

În final, vom afişa valoarea corespunzătoare nodului care reprezintă destinația inelului (MUNTELEOSANDEI).

Analiza complexității

Datele de intrare constau în drumurile dintre repere, deci ordinul de complexitate al operației de citire a acestora este $O(M)$.

Pentru fiecare reper citit se determină codul acestuia, deci ordinul de complexitate al operației de determinare a codurilor este $O(M) \cdot O(N) = O(M \cdot N)$ datorită faptului că acest cod este căutat într-un şir care conține cel mult N elemente.

Crearea listelor predecesorilor se realizează pe parcursul citirii datelor de intrare, operația având ordinul de complexitate $O(M)$.

Stabilirea faptului că există o posibilitate de a se ajunge în VALCEAUADESPICATA implică determinarea codului acestui reper, deci are ordinul de complexitate $O(N)$.

La fiecare pas al algoritmului vom verifica, pentru fiecare nod, dacă au fost determinate informațiile pentru predecesori. Deoarece un nod poate avea, teoretic, până la $N - 1$ predecesori, operațiile efectuate la fiecare pas au ordinul de complexitate $O(N) \cdot O(N) = O(N^2)$. Deoarece sunt efectuați cel mult $N - 1$ paşi, operația de determinare a numărului de posibilități de a ajunge la MUNTELEOSANDEI are ordinul de complexitate $O(N) \cdot O(N^2) = O(N^3)$.

Scrierea datelor de ieşire implică determinarea codului pentru MUNTELEOSANDEI, deci are ordinul de complexitate $O(N)$.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(M) + O(M \cdot N) + O(M) + O(N) + O(N^3) + O(N) = O(N^3 + M \cdot N) = O(N \cdot (N^2 + M))$.

2.4.13. Litere

Problema se reduce la determinarea unui cuvânt care a fost codificat folosind transformarea *Burrows-Wheeler*, dacă se cunoaşte rezultatul codificării. Modul în care se realizează codificarea este prezentat în enunțul problemei.

Decodificarea cuvântului se realizează folosind următorul algoritm:

- se determină frecvența de apariție a literelor, ceea ce permite obținerea primei coloane a matricei (prima coloană conține întotdeauna literele ordonate lexicografic).
- se construieşte un şir de corespondențe A între literele de pe prima coloană şi cele de pe ultima, ținând cont de faptul că celei de-a i-a apariții a caracterului c de pe prima coloană îi corespunde cea de-a i-a apariție a caracterului c de pe ultima.
- pentru determinarea cuvântului se scriu literele corespunzătoare pozițiilor $A[1]$, $A[A[1]]$, $A[A[A[1]]]$ etc; pentru aceasta se foloseşte un indice i care, după fiecare pas va primi valoarea $A[i]$.

Acest algoritm a fost demonstrat matematic de către cei doi cercetători a căror nume au fost date acestei metode.

Analiza complexității

Datele de intrare constau în citirea şirului literelor de pe ultima coloană; deoarece avem N litere, ordinul de complexitate al operației este $O(N)$.

Determinarea frecvențelor aparițiilor literelor se realizează în timp liniar prin simpla parcurgere a şirului care conține literele.

Construirea şirului de corespondențe se realizează tot în timp liniar, deoarece la fiecare pas este realizată câte o corespondență.

Pentru afişarea datelor de ieşire (a cuvântului căutat) se realizează o simplă parcurgere a şirului de corespondențe, deci această operație are ordinul de complexitate $O(N)$.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(N) + O(N) + O(N) + O(N) = O(N)$.

2.4.14. Zăvor

În cele ce urmează vom folosi următoarele denumiri pentru a ne referi la anumite stări ale automatului:

- *stare inițială* (stare de bază) – starea în care se află automatul la început sau după resetare (prima stare);
- *stare finală* (stare de deschidere) – starea în care se află automatul atunci când se deschide cifrul;

- *stare intermediară* – una dintre stările în care se află automatul pe parcursul introducerii parolei corecte (una dintre cele $N - 1$ stări corespunzătoare primelor $N - 1$ cifre ale parolei);
- *stare de eroare* – starea în care se ajunge dacă se recepționează un semnal incorect (a ($N + 2$)-a stare a automatului).

Vom descrie în continuare cele patru situații în care putem detecta că cipul nu funcționează corect:

- cipul poate fi deschis şi în una dintre stările intermediare, nu numai în starea finală;
- există posibilitatea de a "sări" peste anumite stări intermediare;
- există posibilitatea de a ajunge într-o anumită stare în mai multe moduri;
- există posibilitatea de a reveni din starea de eroare în una dintre celelalte stări.

În cele ce urmează vom explica pe larg caracteristicile fiecărei situații şi vom arăta modul în care poate fi găsită o modalitate alternativă de a deschide cifrul dacă a fost detectată o astfel de situație.

Pentru prima situație vom folosi cifrele din parola corectă şi, la fiecare pas, vom verifica dacă una dintre stările intermediare i este stare de deschidere.

Parola alternativă va fi formată din cifrele de pe pozițiile 1, 2, ..., i ale parolei corecte.

Pentru a doua situație vom lua în considerare toate perechile i, j ($i < j$) şi vom verifica dacă cifrul se deschide folosind cifrele de pe pozițiile 1, 2, ..., i, $j + 1$, ..., n, unde n este lungimea parolei corecte.

Parola alternativă va fi formată din cifrele de pe aceste poziții.

Pentru a treia situație vom lua în considerare toate tripletele i, j, k şi vom verifica dacă din starea i se poate ajunge în starea j folosind semnalul k, unde semnalul k nu este cifra care duce la trecerea din starea i în starea $i + 1$.

Parola alternativă va fi formată din cifrele de pe pozițiile 1, 2, ..., i ale parolei corecte, semnalul k şi cifrele de pe pozițiile j, $j + 1$, ..., n ale parolei corecte.

Pentru a patra situație vom lua toate perechile i, j şi vom verifica dacă din starea de eroare putem ajunge în starea i folosind semnalul j. Pentru a ajunge în starea de eroare vom "trimite" la început un semnal z diferit de prima cifră a parolei.

Parola alternativă va fi dată de semnalul z, semnalul j şi cifrele de pe pozițiile i, $i + 1$, ..., n ale parolei corecte.

Dacă nu s-a detectat nici o parolă alternativă, aceasta este scrisă în fişierul de ieşire. În caz contrar, în fişierul de ieşire se va scrie şirul de caractere `CORECT`.

Analiza complexității

Trebuie remarcat faptul că operațiile efectuate de bibliotecă se realizează în timp constant.

Datele de intrare constau în cele n cifre ale parolei corecte, deci citirea acestora are ordinul de complexitate $O(n)$.

Pentru prima situație vom trece pe rând dintr-o stare în alta (pornim din starea inițială şi parcurgem stările intermediare) şi vom testa la fiecare pas dacă s-a ajuns într-o stare de deschidere. Aşadar, ordinul de complexitate al acestei operații este $O(n)$.

Pentru cea de-a doua situație există $n \cdot (n - 1) / 2$ perechi (i, j) pe care le vom lua în considerare. Pentru fiecare astfel de pereche vom parcurge stările intermediare corespunzătoare (vor exista cel mult $n - 1$ astfel de stări), deci ordinul de complexitate al operațiilor efectuate pentru fiecare pereche este $O(n)$. Aşadar, pentru cea de-a doua situație ordinul de complexitate al tuturor operațiilor este $O(n^2) \cdot O(n) = O(n^3)$.

Pentru cea de-a treia situație există n^2 perechi (i, j) şi $9 \cdot n^2$ triplete (i, j, k). Pentru fiecare triplet vom parcurge stările intermediare corespunzătoare care vor fi în număr de cel mult $2 \cdot n + 1$, deci ordinul de complexitate al operațiilor efectuate pentru fiecare triplet este $O(n)$. Aşadar, pentru cea de-a treia situație ordinul de complexitate al tuturor operațiilor este $O(9) \cdot O(n^2) \cdot O(n) = 9 \cdot O(1) \cdot O(n^2) \cdot O(n) = O(1) \cdot O(n^2) \cdot O(n) = O(n^3)$.

Pentru cea de-a patra situație există $10 \cdot n$ perechi (i, j) şi pentru fiecare dintre acestea vom parcurge stările intermediare corespunzătoare (cel mult $n + 1$ astfel de stări), deci ordinul de complexitate al operațiilor efectuate pentru fiecare pereche este $O(n)$. Aşadar, pentru cea de-a patra situație ordinul de complexitate al tuturor operațiilor este $O(10) \cdot O(n) \cdot O(n) = 10 \cdot O(1) \cdot O(n) \cdot O(n) = O(1) \cdot O(n) \cdot O(n) = O(n^2)$.

În momentul în care detectăm o posibilitate alternativă de deschidere a cifrului, va trebui să construim noua parolă. Operația se realizează în timp liniar deoarece, noua parolă va conține cel mult $2 \cdot n + 1$ cifre.

În final, vom scrie parola găsită sau cuvântul `CORECT`, operație realizabilă în timp liniar (dacă a fost găsită o parolă alternativă) sau constant (dacă nu a fost găsită o astfel de parolă).

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(n) + O(n) + O(n^3) + O(n^3) + O(n^2) + O(n) + O(n) = O(n^3)$.