1. Problemele Olimpiadelor de Informatică

1.1. Faza Locală Cluj-Napoca, 2003

1.1.1. Numere

Se consideră un număr natural N cuprins între 1 şi 16. Va trebui să construiți un şir care conține toate numerele cuprinse între 1 şi 2^N astfel:

- se porneşte cu şirul care conține numerele 1 şi 2;
- înaintea acestui şir este inserat numărul 3, iar la sfârşitul şirului este inserat numărul 4, obținându-se şirul 3, 1, 2, 4;
- înaintea acestui şir sunt inserate numerele 5 şi 6, iar la sfârşitul şirului sunt inserate numerele 7 şi 8, obținându-se şirul 5, 6, 3, 1, 2, 4, 7, 8;
- procedeul continuă până la inserarea tuturor celor 2^N numere.

Aşadar, la fiecare pas i şirul va conține primele 2^i numere şi înaintea sa vor fi inserate următoarele 2^{i-1} numere, iar la sfârşitul său vor fi inserate următoarele 2^{i-1} numere.

Date de intrare

Fişierul de intrare **NUMERE.IN** va conține o singură linie pe care se va afla valoarea N.

Date de ieşire

Fişierul de ieşire **NUMERE.OUT** va conține o singură linie pe care se vor afla cele 2^N elemente ale şirului construit potrivit regulilor prezentate anterior. Elementele şirului vor fi separate prin câte un spațiu.

Exemplu

NUMERE.IN	NUMERE.OUT
4	9 10 11 12 5 6 3 1 2 4 7 8 13 14 15 16

Timp de execuție: 1 secundă/test

1.1.2. Triunghi

Se consideră un triunghi (determinat de trei puncte necoliniare din plan) şi alte N puncte din plan.

Să se determine câte dintre aceste N puncte se află în interiorul triunghiului. Un punct este considerat a fi în interiorul triunghiului şi în cazurile în care se află pe una dintre cele trei laturi sau este unul dintre cele trei vârfuri ale triunghiului.

Date de intrare
Primele trei linii ale fişierului de intrare **TRIUNGHI.IN** conţin câte două numere întregi reprezentând coordonatele vârfurilor triunghiului. Următoarea linie conţine numărul N al celorlalte puncte. Fiecare dintre următoarele N linii va conţine câte două numere întregi reprezentând coordonatele unui punct.

Date de ieşire
Fişierul de ieşire **TRIUNGHI.OUT** trebuie să conţină o singură linie pe care se va afla numărul punctelor din interiorul triunghiului.

Restricţii şi precizări
- $1 \le N \le 100000$;
- coordonatele punctelor sunt numere naturale cuprinse între 0 şi 255;
- coordonatele tuturor punctelor sunt date în ordinea: coordonata pe axa Ox, coordonata pe axa Oy.

Exemple

TRIUNGHI.IN	**TRIUNGHI.OUT**
0 0 10 0 0 10 3 1 1 5 5 10 1	2

TRIUNGHI.IN	**TRIUNGHI.OUT**
0 0 1 0 0 1 3 1 1 2 2 3 3	0

Timp de execuţie: 1 secundă/test

1.1.3. Găuri

Se consideră o matrice ale cărei elemente fac parte din mulțimea {0, 1}. O *regiune* este o parte a matricei care conține elemente cu aceeaşi valoare, astfel încât se poate ajunge din orice element al regiunii în oricare altul prin deplasări pe orizontală sau verticală, trecând doar prin elemente ale regiunii. O *gaură* este o regiune de zerouri care nu "atinge" marginile matricei. Va trebui să determinați numărul de elemente pe care le conține cea mai mare gaură (cea care conține un număr maxim de elemente).

Date de intrare

Prima linie a fişierului de intrare **`GAURI.IN`** conține două numere întregi M şi N care reprezintă numărul liniilor, respectiv numărul coloanelor matricei. Fiecare dintre următoarele M linii conțin câte N cifre din mulțimea {0, 1}. Acestea reprezintă valorile elementelor matricei. Valorile de pe o linie nu vor fi separate prin spații.

Date de ieşire

Fişierul de ieşire **`GAURI.OUT`** trebuie să conțină o singură linie pe care se va afla numărul elementelor din cea mai mare gaură.

Restricții şi precizări

- $1 \le M, N \le 100$;
- dacă nu există nici o gaură, se consideră că cea mai mare gaură are 0 elemente;
- pot exista mai multe găuri care au acelaşi număr maxim de elemente.

Exemplu

`GAURI.IN`	**`GAURI.OUT`**

`8 8`
`00000000`
`00100111`
`01110101`
`01011011`
`01001010`
`01101001`
`00111111`
`00000000`
	`4`

Timp de execuție: 1 secundă/test

1.1.4. Exponent

Se consideră două numere întregi m şi n. Să se determine cea mai mare valoare k, astfel încât numărul $m!$ să fie multiplu al numărului n^k.

Date de intrare
Fişierul de intrare **EXPONENT.IN** conţine o singură linie pe care se află două numere întregi, separate printr-un spaţiu, reprezentând valorile *m* şi *n*.

Date de ieşire
Fişierul **EXPONENT.OUT** va conţine o singură linie pe care se va afla valoarea *k*.

Restricţie
- $2 \le m, n \le 2000000000$ (două miliarde).

Exemplu

EXPONENT.IN	EXPONENT.OUT
20 24	6

Timp de execuţie: 1 secundă/test

1.1.5. Soldaţi

Se consideră *n* soldaţi care fac parte dintr-un detaşament şi *n* valori care reprezintă numărul de pachete de hrană pe care le poate duce fiecare dintre soldaţi (prima valoare corespunde primului soldat, a doua valoare celui de-al doilea soldat etc.). Soldaţii trebuie dispuşi într-un număr nenul *b* de bastioane, respectând condiţiile:

- nu pot exista bastioane fără soldaţi;
- bastioanele sunt identificate prin numere cuprinse între 1 şi *b* şi pot fi distruse, dar numai unul după altul, începând cu primul (cu alte cuvinte, un bastion nu poate fi distrus dacă nu sunt distruse toate bastioanele din faţa lui);
- pentru orice număr *d* de bastioane ($0 \le d \le b - 1$) care pot fi distruse, pachetele de hrană purtate de soldaţii din bastionul $d + 1$ trebuie să poată fi împărţite exact (fără fracţiuni de pachet) la toţi soldaţii rămaşi în bastioanele identificate prin numere cuprinse între $d + 1$ şi *b* (cele care nu au fost distruse).

Date de intrare
Prima linie a fişierului de intrare **SOLDATI.IN** va conţine numărul *n* al soldaţilor. Cea de-a doua linie va conţine cele *n* valori care reprezintă numărul de pachete de hrană pe care le poate duce fiecare soldat. Aceste valori vor fi separate prin câte un spaţiu.

Date de ieşire
Fişierul de ieşire **SOLDATI.OUT** va conţine un număr de linii egal cu numărul bastioanelor. Fiecare linie va conţine numărul pachetelor pe care le pot duce soldaţii din bastionul respectiv. Aceste numere vor fi separate prin câte un spaţiu.

Restricții și precizări

- $2 \leq n \leq 1000$;
- un soldat va duce cel puțin unul şi cel mult 1000 de pachete;
- dacă există mai multe soluții, trebuie generată doar una dintre ele;
- pentru datele de test folosite va exista întotdeauna cel puțin o soluție.

Exemplu

SOLDATI.IN	**SOLDATI.OUT**
6	2 4
4 10 3 9 2 3	3 9
	10
	3

Explicație

Detaşamentul este format din şase soldați. Soldații de la primul bastion pot duce 6 pachete care pot fi împărțite la cei şase soldați (cazul în care nu cade nici un bastion).

Soldații de la al doilea bastion pot duce 12 pachete care pot fi împărțite la cei patru soldați rămaşi (cazul în care nu cade primul bastion).

Soldatul de la al treilea bastion poate duce 10 pachete care pot fi împărțite la cei doi soldați rămaşi (cazul în care cad primele două bastioane).

Soldatul de la ultimul bastion poate duce 3 pachete. Acestea nu mai trebuie împărțite deoarece a rămas un singur soldat. *Dacă la ultimul bastion rămas ar fi fost plasați mai mulți soldați, atunci numărul total de pachete pe care le duc soldații de la acest bastion trebuie să poată fi împărțit la numărul soldaților de la acest bastion.*

Dispunerea tuturor soldaților într-un singur bastion nu este posibilă, deoarece numărul total al pachetelor de hrană este 31, care nu este multiplu de 6.

Timp de execuție: 1 secundă/test

1.1.6. Graf

Se consideră un graf conex cu n vârfuri şi m muchii. Nodurile sunt identificate prin numere cuprinse între 1 şi n. Să se determine nodul aflat la cea mai mare distanță față de nodul identificat prin numărul 1. Distanța este dată de lungimea minimă a unui drum de la nodul respectiv la nodul identificat prin numărul 1.

Date de intrare

Prima linie a fişierului de intrare **GRAF.IN** va conține numărul n al nodurilor şi numărul m al muchiilor, separate printr-un spațiu. Fiecare dintre următoarele m linii va descrie o muchie a grafului; ea va conține două numere x şi y, separate printr-un spațiu, cu semnificația: există o muchie între nodul identificat prin numărul x şi nodul identificat prin numărul y.

Date de ieşire
Fişierul de ieşire **GRAF.OUT** va conține o singură linie pe care se va afla numărul prin care este identificat nodul aflat la cea mai mare distanță față de nodul identificat prin numărul 1.

Restricții şi precizări
- $1 \le n \le 100$;
- $n \le m \le 1000$;
- nu pot exista două sau mai multe muchii între aceleaşi două noduri ale grafului;
- dacă există mai multe noduri aflate la distanța maximă, poate fi ales oricare dintre ele.

Exemple

GRAF.IN
```
4 4
1 2
1 3
1 4
2 4
```

GRAF.OUT
```
2
```

GRAF.IN
```
7 8
1 2
1 3
2 3
2 4
3 4
1 5
1 6
1 7
```

GRAF.OUT
```
4
```

Timp de execuție: 1 secundă/test

1.2. Olimpiada Națională de Informatică, 2002

1.2.1. Reconstrucție

În urma unor atentate cu bombe, unul dintre pereții unei clădiri în formă de pentagonu a suferit daune majore. Imaginea codificată a peretelui avariat se reprezintă sub forma unei matrice cu m linii şi n coloane ale cărei elemente sunt numere întregi din mulțimea {0, 1}. Valoarea 1 are semnificația "zid intact", în timp ce semnificația valorii 0 este "zid avariat".

Sumele alocate de guvern pentru refacerea clădirii vor fi donate celor care îi vor ajuta pe constructori să refacă această clădire prin plasarea, pe verticală, a unor blocuri de diverse înălțimi şi lățime 1, în zonele care au fost avariate.

Pentru o structură dată a unui perete din clădire, determinați numărul minim de blocuri necesare pentru refacerea construcției.

Date de intrare

Fişierul de intrare **RECONSTR.IN** conține, pe prima linie, dimensiunile *m* şi *n* ale peretelui clădirii, iar pe următoarele *m* linii, câte *n* valori din mulțimea {0, 1}, neseparate prin spații.

Date de ieşire

Fişierul de ieşire **RECONSTR.OUT** va conține, pe câte o linie, secvențe de forma *k nr*, unde *k* indică înălțimea unui bloc, iar *nr* este numărul de blocuri de înălțime *k*. Cele două numere vor fi separate prin câte un spațiu, iar liniile vor fi ordonate crescător, în funcție de valoarea *k*.

Restricții şi precizări

- $1 \le m, n \le 200$;
- în fişierul de ieşire nu vor apărea linii de forma *k* 0.

Exemplu

RECONSTR.IN	RECONSTR.OUT
5 10	1 7
1110000111	2 1
1100001111	3 2
1000000011	5 1
1111101111	
1110000111	

Timp de execuție: 1 secundă/test

1.2.2. Pod

Între două maluri ale unei văi adânci s-a construit un pod suspendat format din *N* bucăți de scândură, legate cu liane. Vom considera că scândurile sunt numerotate de la 1 la *N*, începând de pe malul pe care ne aflăm. În timp, unele bucăți de scândură s-au deteriorat, iar altele chiar au dispărut. Pentru traversarea podului se ştie că:

- se pot face paşi doar de lungime 1, 2 sau 3;
- scândurile deteriorate sunt nesigure, deci pe ele şi de pe ele se pot face doar paşi de lungime 1;
- evident, nu se poate păşi pe o scândură care lipseşte.

Scrieți un program care să determine numărul de modalități de traversare a podului (mai exact, de a ajunge pe celălalt mal), precum şi o soluție de traversare, dacă o astfel de soluție există.

Date de intrare
Prima linie a fişierului de intrare **POD.IN** conține numărul n al scândurilor. Primul număr de pe a doua linie este k şi indică numărul scândurilor care lipsesc. Această linie mai conține numerele de ordine ale celor k scânduri. Primul număr de pe a treia linie este h şi indică numărul scândurilor deteriorate. Linia mai conține numerele de ordine ale celor h scânduri. Numerele de pe o linie sunt separate prin spații.

Date de ieşire
Prima linie a fişierului de ieşire **POD.OUT** va conține numărul de posibilități de a traversa podul. În cazul în care podul nu poate fi traversat, linia va conține valoarea -1. Dacă podul poate fi traversat, pe a doua linie va fi descrisă o modalitate de traversare. Pe această linie se vor afla numerele de ordine ale scândurilor pe care se păşeşte, în ordinea în care acestea sunt parcurse.

Restricții şi precizări
- $3 \le n \le 300$;
- $0 \le k, h \le n$;
- prima scândură nu poate lipsi;
- numerele de ordine ale scândurilor deteriorate şi numerele de ordine ale scândurilor care lipsesc formează două mulțimi disjuncte;
- numărul posibilităților de traversare conține cel mult 80 de cifre.

Exemple

POD.IN	**POD.OUT**
5 0 0	24 3

POD.IN	**POD.OUT**
10 2 2 7 1 5	48 3 6 8

POD.IN	**POD.OUT**
6 2 2 4 1 3	-1

Timp de execuție: 1 secundă/test

1.2.3. Suma

Considerăm şirul format din primele N numere naturale: $a_i = i$ pentru $i = 1, ..., N$. Fiecărui element al acestui şir i se asociază un semn (+ sau –).

Pentru o valoare S dată se cere să se determine cel mai mic număr N astfel încât, după asocierea semnelor, elementele obținute să aibă suma S.

Date de intrare

Fişierul **SUMA.IN** va conține un singur număr întreg care reprezintă suma care trebuie obținută după asocierea semnelor.

Date de ieşire

Fişierul de ieşire **SUMA.OUT** va conține, pe prima linie, numărul N al elementelor şirului. Pe următoarele linii se vor afla indicii elementelor care au semn negativ, câte unul pe o linie.

Restricție

- $0 < S \leq 100000$.

Exemple

SUMA.IN	SUMA.OUT
12	7 1 7

Timp de execuție: 1 secundă/test

1.2.4. Becuri

Un panou publicitar, de formă dreptunghiulară conține becuri, unul lângă altul, aliniate pe linii şi coloane. Fiecare linie şi fiecare coloană are un comutator care schimbă starea tuturor becurilor de pe acea linie sau coloană, din starea în care se află în starea opusă. Inițial panoul are toate becurile stinse. Să se realizeze un program care, acționând asupra unui număr minim de linii şi coloane, aduce panoul din starea inițială, la o configurație dată, dacă acest lucru este posibil.

Date de intrare

Pe prima linie a fişierului de intrare **BECURI.IN** se află două numere naturale, m şi n, separate printr-un spațiu, reprezentând numărul de linii, respectiv coloane, ale panoului. Pe următoarele m linii se află câte n numere, separate prin câte un spațiu, reprezentând configurația finală a panoului. Un bec aprins este codificat prin 1, în timp ce un bec stins este codificat prin 0.

Date de ieşire
Pe prima linie a fişierului de ieşire **BECURI.OUT** se vor afla indicii coloanelor asupra cărora s-a acționat. Cea de-a doua linie va conține indicii liniilor asupra cărora s-a acționat. Numerele de pe o linie vor fi separate prin câte un spațiu. Dacă nu se acționează asupra nici unei linii sau asupra nici unei coloane, pe linia corespunzătoare va fi scrisă doar valoarea 0. Numerotarea liniilor şi a coloanelor începe de la 1. În cazul în care nu există nici o soluție, fişierul de ieşire va conține o singură linie pe care se va afla valoarea -1.

Restricție
- $1 \le m, n \le 100$.

Exemple

BECURI.IN	BECURI.OUT
5 6 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0	2 5 1 2 3

Timp de execuție: 1 secundă/test

1.2.5. Discuri
Se dau N numere reale care reprezintă razele a N discuri. Considerăm că aşezăm un disc în sistemul xOy dacă îl plasăm la o coordonată x pozitivă suficient de mare, tangent cu axa Ox şi deasupra ei, apoi îl împingem spre Oy până când devine tangent cu Oy sau cu primul disc întâlnit, aşezat anterior. În figura rezultată, după aşezarea tuturor discurilor în ordinea dată, unele dintre ele pot fi considerate *dispensabile* deoarece, prin eliminarea lor, nu se modifică lățimea totală a figurii rezultate, adică nici un disc nu se mai poate deplasa spre stânga. Sarcina voastră este de a scrie un program care identifică toate discurile dispensabile dintr-o configurație dată.

Date de intrare
Fişierul de intrare **DISCURI.IN** conține pe prima linie numărul N al discurilor, iar pe următoarele N linii câte un număr real care indică raza unui disc. Razele apar în ordinea amplasării discurilor.

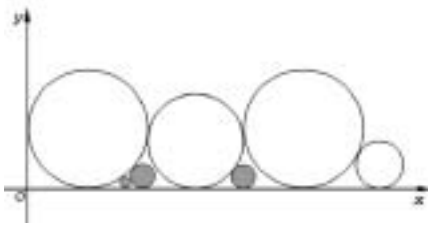
Date de ieşire
Prima linie a fişierului **DISCURI.OUT** va conține numărul K al discurilor dispensabile. Pe următoarele K linii se vor afla numerele de ordine ale discurilor dispensabile.

Restricție

- $1 \le N \le 1000$.

Exemplu

DISCURI.IN	DISCURI.OUT
7	3
4	2
0.1	3
0.5	5
3	
0.5	
4	
1	



cercurile haşurate sunt dispensabile

Timp de execuție: 1 secundă/test

1.2.6. Cod

Transmiterea şi memorarea informațiilor necesită diverse sisteme de codificare în vederea utilizării optime a spațiilor disponibile. Un sistem foarte des întâlnit este acela prin care unei secvențe de caractere i se asociază un număr.

Se consideră cuvintele formate numai cu literele mici ale alfabetului englez. Dintre toate aceste cuvinte le considerăm doar pe cele ale căror caractere sunt în ordine strict lexicografică (caracterul de pe orice poziție este strict mai mic decât un eventual succesor). Sistemul de codificare se obține astfel:

- se ordonează cuvintele în ordinea crescătoare a lungimilor lor;
- cuvintele de aceeaşi lungime se ordonează lexicografic;
- cuvinte sunt codificate prin numerotarea lor (în şirul obținut după ordonare):

```
1.      - a
2.      - b
...
26.     - z
27.     - ab
...
51.     - az
52.     - bc
...
83681.  - vwxyz
...
```

Dându-se un cuvânt, să se precizeze dacă poate fi codificat conform sistemului descris. În caz afirmativ, să se precizeze codul său.

Date de intrare
Fişierul de intrare **COD.IN** conține o singură linie pe care se află un cuvânt.

Date de ieşire
În cazul în care cuvântul poate fi codificat, fişierul de ieşire **COD.OUT** va conține o singură linie pe care se va afla codul acestuia. În caz contrar, fişierul va conține doar valoarea 0.

Restricție
- un cuvânt conține cel mult zece litere.

Exemplu

COD.IN	COD.OUT
bf	55

COD.IN	COD.OUT
aab	0

COD.IN	COD.OUT
vwxyz	83681

Timp de execuție: 1 secundă/test

1.2.7. Hotel

Departamentul administrativ al unui hotel are *n* angajați. Patronul hotelului hotărăşte să schimbe costumele personalului din acest departament, astfel încât angajații care lucrează la etaje diferite să fie îmbrăcați în haine colorate diferit, iar cei care lucrează la acelaşi etaj să fie îmbrăcați în haine colorate la fel. Angajații sunt identificați printr-un cod unic, care constă într-un număr natural format din cel mult patru cifre.

Să se determine numărul modalităților de alegere a costumelor, astfel încât să fie respectate condițiile descrise anterior. De asemenea, trebuie determinată şi una dintre aceste modalități.

Date de intrare
Prima linie a fişierului de intrare **HOTEL.IN** conține numărul *n* al angajaților şi numărul *k* al culorilor disponibile. Cele două numere sunt separate printr-un spațiu. Pe următoarele *n* linii se află câte două numere naturale, separate printr-un spațiu, primul reprezentând codul, iar al doilea etajul asociat unui angajat.

Date de ieşire
Prima linie a fişierului de ieşire **HOTEL.OUT** va conține numărul modalităților de alegere a costumelor. Ştiind că fiecare culoare este codificată printr-un număr natural nenul mai mic sau egal cu k şi că numerele asociate culorilor sunt distincte, în fişier se va scrie pe câte o linie (începând cu a doua) codul unei persoane şi culoarea costumului, valori separate prin câte un spațiu. Ordinea apariției în fişierul de ieşire va fi aceeaşi cu cea din fişierul de intrare. Dacă nu există nici o soluție, în fişier se va scrie o singură linie care va conține valoarea 0.

Restricții şi precizări
- $1 \le n \le 1000$;
- $1 \le k \le 200$;
- hotelul are cel mult 200 de etaje.

Exemple

HOTEL.IN
```
4 5
123 2
35 1
430 2
13 0
```

HOTEL.OUT
```
60
123 1
35 2
430 1
13 3
```

HOTEL.IN
```
5 2
12 1
13 0
14 1
10 2
11 0
```

HOTEL.OUT
```
0
```

Timp de execuție: 1 secundă/test

1.2.8. Lac

O zonă mlăştinoasă are forma dreptunghiulară, având *nl* linii şi *nc* coloane. Ea este formată din celule cu latura de o unitate. O parte din acestea reprezintă *uscat*, iar altele reprezintă *apă*, uscatul fiind codificat cu 0, iar apa cu 1. Se doreşte să se obțină un drum de pe malul de nord spre cel de sud, trecând doar pe uscat. Celulele cu apă pot fi transformate în uscat, paraşutând într-un loc cu apă câte un ponton (o plută) de dimensiunea unei celule. Deoarece paraşutarea este periculoasă, se doreşte minimizarea numărului de paraşutări. Pentru deplasare, dintr-o celulă se poate trece într-o celulă vecină pe linie, coloană sau diagonală.

Determinați numărul minim de pontoane şi coordonatele acestora.

Date de intrare
Fişierul de intrare **LAC.IN** conţine, pe prima linie, numerele naturale *nl* şi *nc*, separate printr-un singur spaţiu. Pe următoarele *nl* linii se află câte *nc* valori binare, separate prin câte un spaţiu, reprezentând configuraţia zonei (0 pentru uscat şi 1 pentru apă).

Date de ieşire
Prima linie a fişierului de ieşire **LAC.OUT** va conţine numărul minim *k* al pontoanelor necesare. Pe următoarele *k* linii se vor afla câte două numere naturale, separate prin câte un spaţiu, reprezentând linia, respectiv coloana, în care a fost amplasat unul dintre pontoane.

Restricţie
- $1 \le nl, nc \le 100$.

Exemplu

LAC.IN
```
8 9
0 1 1 1 1 1 1 1 1
0 1 1 1 1 1 1 1 1
1 0 1 1 1 0 1 1 1
1 1 0 0 1 1 0 1 1
1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 1 1
```

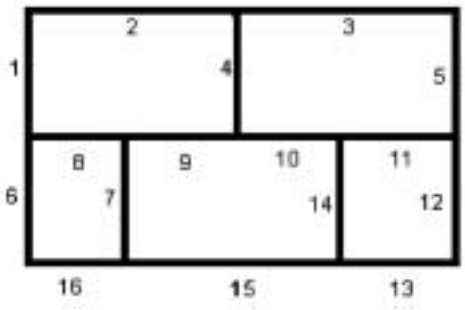
LAC.OUT
```
2
4 5
7 8
```

Timp de execuţie: 1 secundă/test

1.2.9. Logic

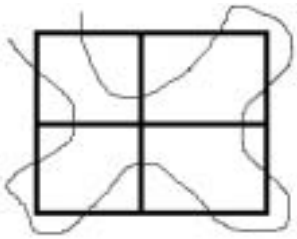
Într-o zi a venit la mine un coleg şi mi-a propus un joc de logică. Mi-a arătat următoarea figură:

Pe această figură au fost numerotate segmentele, pentru a fi mai clară noţiunea de segment. Având la dispoziţie un creion care se află pe hârtie în zona exterioară, trebuie să se traseze curbe pe foaie, fără să se ridice creionul, astfel încât linia să treacă prin toate segmentele (fără a atinge extremităţile) exact o dată.



La sfârşit trebuie să se ajungă tot în exterior. Liniile (curbele) se pot intersecta. Am început şi am încercat de mai multe ori, dar n-am reuşit.

Acum, când am mai crescut, am reuşit să demonstrez că nu se poate, dar am observat că pentru alte figuri este posibil. Un exemplu în care problema are soluție este:



Eu mi-am dat seama de ce uneori se poate şi alteori nu, dar vreau să văd dacă reuşiți şi voi. De aceea vă voi da câteva figuri şi, pentru fiecare, trebuie să răspundeți cu `DA` sau `NU` la întrebarea: *se poate sau nu trasa o curbă având tipul descris mai sus*?

Date de intrare

Fişierul de intrare **`LOGIC.IN`** are următoarea structură:

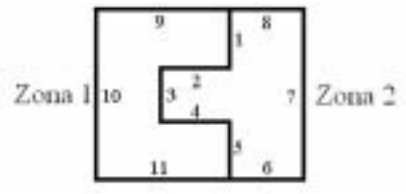
- pe prima linie se află numărul *T* al figurilor;
- pe următoarele *T* grupuri de linii se află datele corespunzătoare celor *T* figuri, după cum urmează:
  - pe prima linie a grupului se află valoarea *N* care reprezintă numărul de linii şi de coloane ale matricei pătratice care descrie figura;
  - pe următoarele *N* linii se află câte *N* numere, separate printr-un spațiu (numere întregi cuprinse între 0 şi 255), care reprezintă elementele matricei.

Această matrice codifică figura astfel: definim o zonă ca fiind o parte continuă a matricei care conține acelaşi număr şi este aria maximă cu această proprietate; altfel spus, două căsuțe alăturate (care diferă la o singură coordonată printr-o unitate) care conțin acelaşi număr, se lipesc. Cu alte cuvinte, în interiorul zonei nu vor exista segmente. La aceste zone se mai adaugă şi exteriorul matricei ca fiind o nouă zonă. În matrice pot fi zone cărora le corespunde acelaşi număr, dar care "nu se ating" şi se consideră a fi zone diferite. Segmentele sunt segmente de dreaptă care separă două zone şi au lungimea maximă.

De exemplu, figura definită prin:

```
3
1 1 2
1 2 2
1 1 2
```

este următoarea:



Se observă că între zonele 1 şi 2 există cinci segmente, între zona 1 şi exterior există trei segmente, iar între zona 2 şi exterior există tot trei segmente. Trebuie remarcat faptul că segmentul numerotat în figura anterioară cu 10 este compus din trei segmente mici, dar este considerat a fi un singur segment, conform definiției.

Date de ieşire
În fişierul **LOGIC.OUT** se vor scrie T linii, corespunzătoare celor T figuri. Pe fiecare linie va fi scris unul dintre mesajele DA sau NU, în funcție de răspunsul la întrebare.

Restricții
- $1 \le T \le 10$;
- $1 \le N \le 100$.

Exemplu

LOGIC.IN	LOGIC.OUT
2 2 1 2 3 4 4 1 1 2 2 1 1 2 2 3 4 4 1 3 4 4 1	DA NU

Timp de execuție: 1 secundă/test

1.2.10. Foto

Gigel, specialist în editare grafică pe calculator, se confruntă cu o problemă. El trebuie să aranjeze patru fotografii, disponibile în format electronic, într-o pagină de prezentare, astfel încât suprafața paginii să fie complet "acoperită" de cele patru fotografii şi fără ca acestea să se suprapună în vreun fel. Gigel poate modifica dimensiunile inițiale ale fotografiilor, însă fără a deforma imaginile. Pentru aceasta el trebuie să păstreze neschimbat raportul dintre lungimea şi înălțimea fotografiilor. Nu contează ordinea aşezării fotografiilor, ele putând fi translatate oriunde în cadrul paginii, însă operațiile de rotație nu sunt permise.

Determinați pentru fiecare fotografie dimensiunile finale, cunoscându-se dimensiunile paginii, precum şi dimensiunile inițiale ale fotografiilor.

Date de intrare
Prima linie a fişierului de intrare **FOTO.IN** conține două numere naturale l şi h, separate printr-un spațiu, reprezentând lungimea, respectiv înălțimea paginii. Pe următoarele patru linii se află câte două numere naturale, separate printr-un spațiu, reprezentând lungimea, respectiv înălțimea, uneia dintre cele patru fotografii care trebuie aranjate.

Date de ieşire
Fişierul de ieşire **FOTO.OUT** va conține patru linii pe care se vor afla câte două numere naturale care reprezintă dimensiunile (lățime şi înălțime) finale ale celor patru fotografii. Prima linie va corespunde primei fotografii din fişierul de intrare, a doua linie celei de-a doua fotografii etc.

Restricții şi precizări
- dimensiunile paginii şi ale fotografiilor sunt numere naturale cuprinse între 1 şi 2000;
- dacă există mai multe soluții va fi determinată doar una dintre ele;
- există întotdeauna cel puțin o posibilitate de amplasare a fotografiilor.

Exemplu

FOTO.IN	FOTO.OUT
140 140 24 12 4 13 10 14 4 2	20 10 40 130 100 140 20 10

Timp de execuție: 1 secundă/test

1.2.11. Balanță

Gigel are o "balanță" mai ciudată pe care vrea să o echilibreze. De fapt, aparatul este diferit de orice balanță pe care ați văzut-o până acum. Balanța lui Gigel dispune de două brațe de greutate neglijabilă, fiecare având lungimea 15. Din loc în loc, pe aceste brațe se află cârlige, de care Gigel poate atârna greutăți distincte din colecția sa de G greutăți (numere naturale cuprinse între 1 şi 25). Gigel poate atârna oricâte greutăți de orice cârlig, dar trebuie să folosească toate greutățile de care dispune.

Gigel a reuşit să echilibreze balanța relativ repede, dar acum doreşte să ştie în câte moduri poate fi ea echilibrată.

Cunoscând amplasamentul cârligelor şi greutățile pe care Gigel le are la dispoziție, scrieți un program care calculează în câte moduri se poate echilibra balanța.

Date de intrare
Fişierul de intrare **BALANTA.IN** are următoarea structură:
- pe prima linie se află numărul C al cârligelor şi numărul G al greutăților; cele două numere sunt separate printr-un spațiu;
- pe a doua linie se află C numere întregi distincte, separate prin spațiu, cu valori cuprinse între -15 şi 15 inclusiv, reprezentând amplasamentele cârligelor față de

centrul balanței; valoarea absolută a numerelor reprezintă distanța față de centrul balanței, iar semnul indică brațul balanței pe care se află cârligul, '–' pentru brațul stâng şi '+' pentru brațul drept;

- pe a treia linie se află G numere naturale distincte, cuprinse între 1 şi 25, reprezentând valorile greutăților pe care Gigel le va folosi pentru a echilibra balanța.

Date de ieşire

Fişierul de ieşire **BALANTA.OUT** conține o singură linie, pe care se află un număr natural M, numărul de variante de plasare a greutăților care duc la echilibrarea balanței.

Restricții şi precizări

- $2 \le C, G \le 20$;
- există întotdeauna cel puțin o variantă şi cel mult 100.000.000;
- balanța se echilibrează dacă suma produselor dintre greutăți şi coordonatele unde ele sunt plasate este 0 (suma momentelor greutăților față de centrul balanței este 0).

Exemplu

BALANTA.IN	BALANTA.OUT
2 4 -2 3 3 4 5 8	2

Timp de execuție: 1 secundă/test

1.2.12. Aliniere

În armată, o companie este alcătuită din n soldați. La inspecția de dimineață soldații stau aliniați în linie dreaptă în fața căpitanului. Acesta nu e mulțumit de ceea ce vede; e drept că soldații sunt aşezați în ordinea numerelor de cod din registru, dar nu în ordinea înălțimii. Căpitanul cere câtorva soldați să iasă din rând, astfel ca cei rămaşi, fără a-şi schimba locurile, doar apropiindu-se unul de altul (pentru a nu rămâne spații mari între ei) să formeze un şir în care fiecare soldat vede privind de-a lungul şirului, cel puțin una din extremități (stânga sau dreapta). Un soldat vede o extremitate dacă între el şi capătul respectiv nu există un alt soldat cu înălțimea mai mare sau egală cu a lui.

Scrieți un program care determină, cunoscând înălțimea fiecărui soldat, numărul minim de soldați care trebuie să părăsească formația, astfel ca şirul rămas să respecte condiția impusă de căpitan.

Date de intrare

Pe prima linie a fişierului de intrare **ALINIERE.IN** se află numărul n al soldaților din şir, iar pe linia următoare se află un şir de n numere reale, cu maximum cinci zecimale

fiecare, separate prin spații. Al *k*-lea număr de pe această linie reprezintă înălțimea soldatului având codul *k*.

Date de ieşire
Fişierul **ALINIERE.OUT** va conține pe prima linie numărul soldaților care trebuie să părăsească formația, iar pe linia următoare, codurile acestora în ordine crescătoare, separate două câte două printr-un spațiu.

Restricții şi precizări
- înălțimile soldaților sunt numere reale cuprinse între 0,5 şi 2,5;
- $2 \le n \le 1000$;
- dacă există mai multe soluții posibile, se va alege doar una dintre ele.

Exemplu

ALINIERE.IN	**ALINIERE.OUT**
8 1.86 1.86 1.30621 2 1.4 1 1.97 2.2	4 1 3 7 8

Timp de execuție: 1 secundă/test

1.2.13. Arbore
Se consideră un arbore cu *N* vârfuri, numerotate de la 1 la *N*. Scrieți un program care adaugă arborelui dat un număr minim de muchii, astfel încât fiecare vârf al grafului obținut să aparțină exact unui singur ciclu.

Date de intrare
Prima linie a fişierului de intrare **ARBORE.IN** conține numărul *N* al vârfurilor arborelui. Pe următoarele *N* – 1 linii se află câte două numere, separate printr-un spațiu, care reprezintă extremitățile uneia dintre muchiile arborelui.

Date de ieşire
Prima linie a fişierului de ieşire **ARBORE.OUT** conține numărul *Nr* al muchiilor adăugate. Pe următoarele *Nr* – 1 linii se află câte două numere, separate printr-un spațiu, care reprezintă extremitățile uneia dintre muchiile adăugate. În cazul în care problema nu are soluție, fişierul de ieşire va conține o singură linie pe care se va afla valoarea -1.

Restricție
- $3 \le N \le 1000$.

Exemple

ARBORE.IN
```
4
1 2
2 3
2 4
```

ARBORE.OUT
```
-1
```

ARBORE.IN
```
7
1 2
1 3
3 5
3 4
5 6
5 7
```

ARBORE.OUT
```
2
6 7
2 4
```

Timp de execuție: 1 secundă/test

1.2.14. Decodificare

Serviciul Român de Informații a dat de urma unei organizații teroriste care își are sediul pe teritoriul țării noastre. Folosind cei mai pricepuți şi mai bine antrenați spioni şi ofițeri, SRI a reuşit să identifice computerul principal al organizației teroriste. Dacă va reuşi să acceseze informațiile din acest computer, SRI îi va putea aresta pe toți membrii organizației şi va asigura menținerea păcii mondiale. Singura problemă este spargerea codului de acces. Tot ceea ce se ştie despre acest cod este că el este reprezentat de către o permutare de lungime *N*. Specialiştii SRI au încercat diverse metode de a descoperi codul, însă tot ceea ce au reuşit să obțină este un program care, transmițându-i-se ca parametru o permutare de lungime *N*, specifică în câte poziții codul de acces coincide cu aceasta.

Scrieți un program care, folosind programul-ajutător (reprezentat sub forma unui modul), determină codul de acces în computerul teroriştilor.

Date de intrare

Programul nu va citi date din nici un fişier de intrare. El va apela întâi funcția `GetN` a modulului `PROG`, care va returna valoarea *N* – numărul de elemente ale permutării care trebuie descoperită.

Apoi va apela numai funcția `Check`, căreia îi va fi transmisă ca parametru, de fiecare dată, o permutare de lungime *N*. Această funcție va returna numărul de poziții în care permutarea coincide cu permutarea care trebuie descoperită. Programul dumneavoastră trebuie ca, după un număr finit de apelări ale funcției `Check`, să descopere permutarea căutată.

Date de ieşire
Programul va trebui să tipărească în fişierul **DECOD.OUT** o permutare de lungime *N*. Toate cele *N* elemente vor fi tipărite pe prima linie a fişierului, fiind separate de câte un spațiu.

Restricție
- $5 \leq N \leq 256$.

Instrucțiuni pentru programatorii în C/C++
Programatorii în C/C++ au la dispoziție *header*-ul `PROG.H`. În acest fişier sunt declarate următoarele funcții:

```
int GetN(void)
int Check(int p[256])
```

Funcția `Check` are ca parametru un vector cu elemente întregi. Ea va returna una dintre următoarele valori:
- valoarea -1 dacă primele *N* elemente (începând cu poziția 0) ale vectorului transmis ca parametru nu constituie o permutare de lungime *N*;
- în cazul în care permutarea este corectă, se returnează un număr cuprins între 0 şi *N* care reprezintă numărul de poziții în care permutarea de lungime *N* transmisă ca parametru coincide cu permutarea care trebuie descoperită.

Instrucțiuni pentru programatorii în Pascal
Modulul extern este implementat sub forma *unit*-ului `PROG`. În acest *unit* sunt declarate următoarele tipuri şi funcții care vor fi folosite în program:

```
type perm = array[1..256] of Integer;
function GetN:Integer;
function Check(var permut:perm):Integer;
```

Funcția `Check` are ca parametru un vector de tipul `perm`. Ea va returna una dintre următoarele valori:
- valoarea -1 dacă primele *N* elemente (începând cu poziția 0) ale vectorului transmis ca parametru nu constituie o permutare de lungime *N*;
- în cazul în care permutarea este corectă, se returnează un număr cuprins între 0 şi *N* care reprezintă numărul de poziții în care permutarea de lungime *N* transmisă ca parametru coincide cu permutarea care trebuie descoperită.

Exemplu
Să presupunem că permutarea căutată este 2 1 3 4 5. La început va fi apelată funcția `GetN`; aceasta va returna valoarea 5. Vor urma apeluri succesive ale funcției `Check`, până în momentul în care aceasta returnează valoarea 5. O posibilă succesiune este:

- apel pentru permutarea 1 2 3 4 5 - se returnează 3;
- apel pentru permutarea 3 2 1 4 5 - se returnează 2;
- apel pentru permutarea 2 1 3 4 5 - se returnează 5.

Permutarea corectă este scrisă în fişierul **DECOD.OUT**.

Timp de execuţie: 1 secundă/test

1.2.15. Seti

Se pare că, în sfârşit, căutătorii vieţii extraterestre au descoperit ceva! În cursul proiectului *SETI@home* a fost izolată o secvenţă care ar putea reprezenta un semnal de la alte forme de viaţă inteligentă. Ca urmare, proiectul *SETI@ONI* îşi propune să verifice dacă acel semnal provine într-adevăr de la extratereştri sau doar de la nişte puşti care beau *Fanta*.

Pentru comoditate, porţiunea de semnal care trebuie analizată vi se pune la dispoziţie sub forma unei succesiuni de litere ale alfabetului latin. De asemenea, aveţi la dispoziţie şi un dicţionar de cuvinte extraterestre, codificate în acelaşi mod. Trebuie să numărați de câte ori apare fiecare dintre aceste cuvinte în posibilul mesaj extraterestru. Pornind de la aceste date, lingviştii pot să înceapă lucrul la traducerea mesajului.

Date de intrare

Pe prima linie a fişierului de intrare **SETI.IN** se află numărul N al liniilor mesajului. Urmează N linii, fiecare conţinând exact 64 de litere ale alfabetului latin, urmate de marcajul de sfârşit de linie. Prin alipirea acestor linii se obţine mesajul de analizat, format din $64 \cdot N$ litere.

Pe prima linie a celui de-al doilea fişier de intrare **DIC.IN** este scris numărul M al cuvintelor din dicţionar. Urmează M linii, fiecare conţinând un cuvânt din dicţionar, reprezentat ca o secvenţă de cel puţin una şi cel mult 16 litere. Cuvintele nu sunt neapărat distincte.

Date de ieşire

Fişierul de ieşire **SETI.OUT** va conţine exact M linii. Pe linia cu numărul i va fi scris numărul de apariţii ale celui de-al i-lea cuvânt din dicţionar. Orice apariţie a unui cuvânt trebuie numărată, chiar dacă se suprapune peste alte apariţii. Se va face distincţie între literele mari şi literele mici.

Restricţii şi precizări

- $0 \le N < 2048$;
- $5 \le M \le 32000$;
- un cuvânt poate avea cel mult 65535 de apariţii.

Exemplu
SETI.IN
```
2
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaBaba
babaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaBaB
```

DIC.IN
```
3
b
bab
b
```

SETI.OUT
```
3
2
3
```

Timp de execuție: 1 secundă/test

1.2.16. Suma divizorilor

Se consideră două numere naturale A şi B. Fie S suma tuturor divizorilor naturali ai lui A^B. Să se determine restul împărțirii lui S la 9901.

Date de intrare
Pe prima linie a fişierului de intrare **SUMDIV.IN** se află numerele A şi B, separate prin cel puțin un spațiu.

Date de ieşire
Fişierul de ieşire **SUMDIV.OUT** va conține o singură linie pe care se va afla restul împărțirii numărului S la 9901.

Restricție

- $0 \le A, B \le 50000000$.

Exemplu

SUMDIV.IN	SUMDIV.OUT
2 3	15

Timp de execuție: 0,5 secunde/test

1.2.17. Sistem

Într-un județ există N oraşe, numerotate de la 1 la N. Fiecare dintre cele N oraşe ale județului este legat de exact alte două oraşe, prin străzi bidirecționale. Şi mai ciudat este faptul că, în cadrul acestui sistem stradal, nu este întotdeauna posibil să ajungi din orice oraş în oricare alt oraş mergând pe străzi. Oricum, locuitorii județului sunt mândri de acest sistem al lor şi sunt de părere că nu mai există altul la fel. Trebuie să le demonstrați contrariul şi pentru aceasta trebuie să calculați câte sisteme stradale distincte cu proprietatea de mai sus există. Două sisteme sunt considerate distincte dacă există cel puțin o stradă între o pereche de oraşe i şi j în cadrul primului sistem, care nu există în cadrul celui de-al doilea.

Date de intrare

Din fişierul **SISTEM.IN** veți citi valoarea N, care reprezintă numărul de oraşe ale județului.

Date de ieşire

În fişierul **SISTEM.OUT** se va scrie numărul de sisteme stradale distincte, cu proprietatea că orice oraş este legat prin străzi directe de exact alte două oraşe.

Restricție

- $3 \le N \le 100$.

Exemple

SISTEM.IN	SISTEM.OUT
4	3

SISTEM.IN	SISTEM.OUT
6	70

Timp de execuție: 1 secundă/test

1.2.18. Comitat

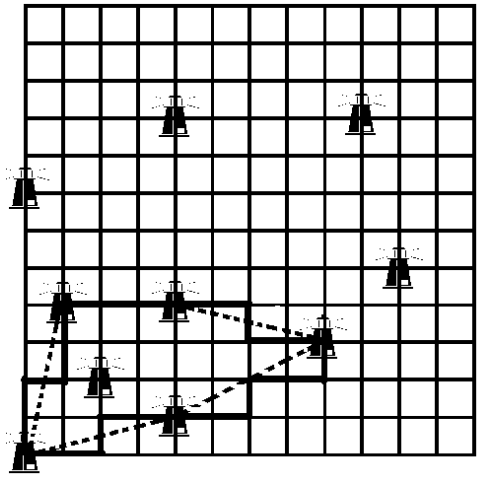
Toate semințiile convieţuitoare pe *Terra* au hotărât ca *hobbiții*, păstrătorii *Inelului Puterii*, să fie izolați într-o zonă a pământului numită *Comitat*. Hotarele *Comitatului* trebuie să fie reprezentate de un poligon convex cu câte un turn de pază în fiecare vârf.

Se cunosc pozițiile tuturor turnurilor din regiune (două numere naturale raportate la un sistem de axe rectangulare). Un paznic pe cal alb veghează hotarele *Comitatului* parcugând, pe rând, toate distanțele dintre două turnuri succesive, mergând pe drum minim, numai pe cărări paralele cu axele.

Se cunoaşte lungimea maximă a drumului pe care-l poate parcurge paznicul la un tur complet al hotarelor *Comitatului* şi se cere să se determine un poligon cu un număr

maxim de turnuri pe contur, poligon care poate constitui hotarul comitatului. În plus, hotarul trebuie să conțină turnul din *Mordor* (de coordonate 0 şi 0) într-un vârf, iar în fiecare dintre celelalte vârfuri se află, obligatoriu, unul dintre turnurile existente.

De exemplu, pentru amplasamentul turnurilor ilustrat mai jos şi pentru limita de 25 km a unui tur efectuat de paznic, hotarul *Comitatului* poate fi format, în această ordine, din turnurile de coordonate (0, 0), (4, 1), (8, 3), (4, 4), (1, 4), (0, 0). Se observă că poligonul determinat de aceste turnuri este un poligon convex cu cinci turnuri pe contur.



Poligonul cu vârfurile (0, 0), (4, 1), (4, 12), (0, 7), (0, 0) are tot cinci turnuri pe contur, dar un tur complet al acestui poligon depăşeşte 25 km.

Date de intrare

Pe prima linie a fişierului de intrare `COMITAT.IN` se află numărul *n* al turnurilor din ținut (turnul din *Mordor* nu este numărat). Pe fiecare dintre următoarele *n* linii se află câte două numere naturale, despărțite printr-un spațiu, care reprezintă coordonatele unuia dintre turnuri. Ultima linie a fişierului conține lungimea maximă *L* a unui tur complet al poligonului.

Date de ieşire

Prima linie a fişierului de ieşire `COMITAT.OUT` va conține numărul *v* al turnurilor de pe conturul poligonului (incluzând turnul din *Mordor*). Pe a doua linie se vor afla $v - 1$ numere întregi, reprezentând numerele de ordine ale turnurilor de pe contur, pornind de la turnul din *Mordor* (care nu este afişat) şi respectând succesiunea în sens trigonometric sau antitrigonometric a turnurilor de pe contur.

Restricții și precizări

- $0 < n \leq 50$;
- $0 < L < 1000$;
- coordonatele turnurilor sunt numere întregi cuprinse între 0 şi 200;
- pot exista turnuri în interiorul poligonului, dar acestea nu sunt luate în considerare pentru numărarea turnurilor corespunzătoare unui poligon;
- există posibilitatea ca soluția să fie dată de un poligon degenerat format dintr-un singur vârf (*Mordor*), din două vârfuri (*Mordor* şi un alt turn) sau din mai multe vârfuri coliniare;
- pe conturul poligonului determinat pot exista turnuri coliniare;
- dacă există mai multe soluții care respectă condițiile din enunț, se va furniza doar una dintre acestea;
- în fişierul de intrare nu există două turnuri ale căror poziții să coincidă şi singurul turn din poziția (0, 0) este cel din *Mordor*.

Exemplu

COMITAT.IN	COMITAT.OUT
9	5
0 7	4 7 5 2
1 4	
2 2	
4 1	
4 4	
4 9	
8 3	
9 9	
10 5	
25	

Timp de execuție: 1 secundă/test

1.3. Soluțiile problemelor propuse spre rezolvare la olimpiada de informatică, faza locală, Cluj, 2003

1.3.1. Numere

Definirea recursivă a problemei sugerează foarte clar modul de rezolvare a acesteia. Vom crea două rutine recursive, una care generează numerele din prima jumătate a şirului şi una care generează numerele din a doua jumătate a şirului.

Pentru prima parte a şirului, la fiecare pas i se vor genera numere cuprinse între 2^i şi $3 \cdot 2^{i-1} - 1$. Pentru a doua parte a şirului, la fiecare pas i se vor genera numere cuprinse între $3 \cdot 2^{i-1}$ şi $2^{i+1} - 1$.

Diferența principală a celor două rutine recursive este poziția apelului recursiv față de generarea elementelor. Pentru prima parte a şirului, apelul va apărea după generare, iar pentru a doua parte a şirului, apelul va apărea înaintea generării.

Analiza complexității

Datele de intrare constau într-un singur număr, deci ordinul de complexitate al operației de citire este $O(1)$.

Întotdeauna vor fi generate 2^N numere, deci ordinul de complexitate al operației de generare a acestora este $O(2^N)$.

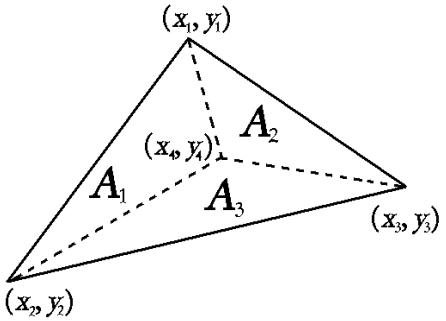
Scrierea datelor în fişierul de ieşire este realizată pe parcursul determinării acestora, deci operația de scriere nu consumă timp suplimentar.

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

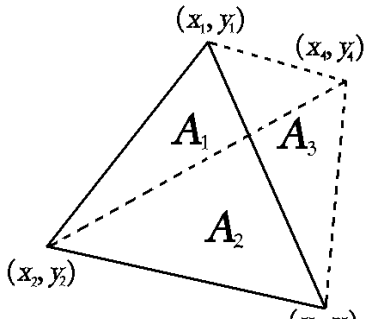
1.3.2. Triunghi

Pentru fiecare punct dat va trebui să verificăm dacă se află sau nu în interiorul triunghiului considerat şi să numărăm punctele din interiorul triunghiului.

O variantă mai simplă de rezolvare se bazează pe faptul că dacă un punct se află în interiorul triunghiului, pe una din laturi sau este un vârf, atunci suma ariilor celor trei triunghiuri determinate de punctul respectiv şi perechi de vârfuri ale triunghiului dat este egală cu aria triunghiului dat (vezi figura următoare).



$A = A_1 + A_2 + A_3$



$A < A_1 + A_2 + A_3$

Aşadar, pentru fiecare punct în parte vom determina ariile triunghiurilor determinate de punctul respectiv şi două vârfuri ale triunghiului. Dacă suma celor trei arii este egală cu aria triunghiului, atunci punctul se află în interiorul triunghiului.

Se observă că dacă punctul este un vârf al triunghiului, două dintre cele trei arii vor fi egale cu 0, iar cea de-a treia arie va fi chiar aria triunghiului dat.

În cazul în care vârful se află pe una dintre laturi (dar nu este un vârf al triunghiului), atunci una dintre arii va fi egală cu 0.

Dacă se cunosc coordonatele vârfurilor unui triunghi, atunci formula de calcul a ariei acestuia este:

$$\left| \frac{1}{2} \cdot \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \right|.$$

În final vom scrie în fişierul de ieşire numărul punctelor aflate în interiorul triunghiului.

Analiza complexității

Citirea coordonatelor vârfurilor triunghiului constă în citirea a şase numere întregi, deci se realizează în timp constant.

Verificarea faptului dacă un punct se află sau nu în interiorul triunghiului considerat se realizează cu ajutorul unui număr constant de operații aritmetice. Această operație poate fi realizată pe măsura citirii coordonatelor punctelor. Aşadar, ordinul de complexitate al operațiilor de citire a datelor şi determinare a soluției este $O(N)$.

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

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(1) + O(N) + O(1) = O(N)$.

1.3.3. Găuri

Vom determina toate regiunile formate din zerouri şi vom eticheta elementele acestor regiuni folosind numere naturale, începând cu 1.

Pentru fiecare regiune determinată vom păstra dimensiunile acestora (numărul de elemente) într-un vector ai cărui indici sunt numerele folosite pentru etichetarea dimensiunilor regiunilor. Pentru a determina aceste regiuni vom folosi un simplu algoritm de umplere.

O gaură poate fi privită ca fiind o regiune formată din zerouri care nu conține nici un element aflat pe marginea matricei. Vom parcurge cele patru margini ale matricei şi vom seta la 0 dimensiunile tuturor regiunilor care conțin elemente de pe aceste margini. Astfel, practic am eliminat toate regiunile care nu sunt găuri.

În final vom parcurge vectorul dimensiunilor şi vom determina elementul care are valoarea maximă. Valoarea acestui element va fi scrisă în fişierul de ieşire.

Analiza complexității

Datele de intrare constau în dimensiunile matricei şi valorile tuturor elementelor acesteia. Aşadar ordinul de complexitate al operației de citire este $O(M \cdot N)$.

Operația de determinare a regiunilor formate din zerouri folosind un algoritm de umplere are ordinul de complexitate $O(M \cdot N)$.

Pentru a elimina regiunile care nu sunt găuri vom parcurge cele patru margini ale matricei, aşadar această operație are ordinul de complexitate $O(M) + O(N) + O(M) + O(N) = O(M + N)$.

În total pot exista cel mult $[M \cdot N / 2] + 1$ regiuni (dacă matricea are aspectul unei table de şah). Aşadar, operația de determinare a dimensiunii celei mai mari regiuni are ordinul de complexitate $O(M \cdot N)$.

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

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(M \cdot N) + O(M \cdot N) + O(M + N) + O(M \cdot N) + O(1) = O(M \cdot N)$.

1.3.4. Exponent

Pentru a determina valoarea k, va trebui, mai întâi să descompunem numărul n în factori primi. Pentru fiecare factor p care apare în descompunerea lui n, va trebui să determinăm puterea la care apare acest factor în descompunerea în factori primi a valorii $m!$. Vom considera că p apare în descompunerea în factori primi a numărului n de u ori; cu alte cuvinte, n poate fi scris sub forma $n = p^u \cdot q$, unde q este un număr întreg. Vom determina acum puterea v la care apare p în descompunerea în factori primi a numărului $m!$. Valoarea v poate fi obținută astfel:

- pentru fiecare multiplu al lui p mai mic sau egal cu m, numărul p apare o dată în descompunerea lui $m!$;
- pentru fiecare multiplu al lui p^2 mai mic sau egal cu m, numărul p apare încă o dată în descompunerea lui $m!$;
- în general, pentru fiecare multiplu al lui p^i mai mic sau egal cu m, numărul p apare încă o dată în descompunerea lui $m!$.

Expresia pentru fiecare multiplu al lui p^2 mai mic sau egal cu m este echivalentă cu expresia pentru fiecare multiplu al lui p mai mic sau egal cu $[m / p]$.

În general, expresia pentru fiecare multiplu al lui p^i mai mic sau egal cu m este echivalentă cu expresia pentru fiecare multiplu al lui p mai mic sau egal cu $[m / p^{i-1}]$.

Ca urmare, valoarea v la care apare p în descompunerea în factori primi a numărului $m!$ poate fi determinată folosind următorul algoritm:

```
...
v ← 0
cât timp m > 0 execută
  v ← v + [m / p]
  m ← [m / p]
sfârşit cât timp
...
```

În aceste condiții, pentru ca $n^k = p^{u \cdot k} \cdot q^k$ să fie divizor al numărului $m!$, trebuie să avem $k \le [v / u]$.

Vom determina valorile maxime ale numărului k pentru toți divizorii p şi o vom alege pe cea mai mică dintre acestea.

Analiza complexității

Intrarea constă în citirea a două numere, aşadar ordinul de complexitate al operației de citire a datelor este $O(1)$.

Operația de descompunere în factori primi a unui număr natural n are ordinul de complexitate $O(\sqrt{n})$.

Numărul factorilor primi ai valorii n poate fi cel mult $[\log_2 n]$. Pentru un număr m dat, folosirea algoritmului prezentat pentru determinarea puterii la care apare p în descompunerea în factori primi a valorii $m!$ necesită parcurgerea a $[\log_p m]$ paşi.

Ca urmare, operația de determinare a valorii k are ordinul de complexitate $O(\log m) \cdot O(\log n) = O(\log m \cdot \log n)$.

Ieşirea constă într-un singur număr, aşadar ordinul de complexitate al operației de scriere a datelor este $O(1)$.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(1) + O(\sqrt{n}) + O(\log m \cdot \log n) + O(1) = O(\sqrt{n} + \log m \cdot \log n)$.

1.3.5. Soldați

Se observă că pentru primul bastion va trebui să alegem o submulțime a soldaților care pot duce un număr total de pachete divizibil cu valoarea n. Pentru al doilea bastion, dintre cei k soldați rămaşi va trebui să alegem o submulțime a soldaților care pot duce un număr total de pachete divizibil cu valoarea k. Aşadar, pentru al doilea bastion avem aceeaşi problemă pentru o valoare k strict mai mică decât n.

Practic, la fiecare pas vom rezolva aceeaşi problemă pentru valori din ce în ce mai mici până în momentul în care nu mai există nici un soldat.

Aşadar, la fiecare pas va trebui să determinăm o submulțime a unei mulțimi cu k elemente a cărei sumă să fie divizibilă cu k. În acest scop vom folosi *principiul cutiei lui Dirichlet*.

Presupunem că, la un moment dat, cei k soldați rămaşi pot duce a_1, a_2, ..., respectiv a_k pachete. Vom calcula sumele $S_i = a_1 + a_2 + ... + a_i$ folosind formula recurentă $S_1 = a_1$ şi $S_i = a_i + S_{i-1}$ pentru toate valorile i cuprinse între 2 şi k.

Potrivit principiului cutiei lui Dirichlet, dacă resturile împărțirii valorilor S_i la k sunt distincte, atunci unul dintre aceste resturi este 0.

Potrivit aceluiaşi principiu, dacă resturile nu sunt distincte, atunci două dintre acestea sunt egale. Dacă resturile împărțirii valorilor S_u şi S_v sunt egale, atunci restul împărțirii valorii $S_v - S_u$ la k este 0. Aşadar, valoarea $a_{u+1} + a_{u+2} + ... + a_v$ va fi divizibilă cu k.

În concluzie, va exista întotdeauna posibilitatea de a alege o submulțime a cărei sumă să fie divizibilă cu k. După alegerea acestei submulțimi, dacă ea nu conține toate cele k elemente, va trebui să rezolvăm aceeaşi problemă pentru o valoare strict mai mică decât k (elementele mulțimii respective sunt eliminate).

Analiza complexității

Intrarea constă în citirea valorilor corespunzătoare celor n soldați, deci ordinul de complexitate al operației de citire a datelor este $O(n)$.

La fiecare pas va trebui să determinăm sumele S_i, operație care are ordinul de complexitate $O(k)$, unde $k = O(n)$ este numărul soldaților rămaşi la pasul respectiv.

Pe parcursul determinării sumelor vom păstra şi resturile împărțirii lor la k, aşadar vom putea identifica elementele care vor fi eliminate pe parcursul determinării sumelor. Eliminarea propriu-zisă este realizată în timp liniar. În cazul cel mai defavorabil, la fiecare pas vom elimina un singur element, aşadar putem avea cel mult n paşi. Ca urmare, ordinul de complexitate al operației de repartizare a soldaților este $O(n) \cdot O(n) = O(n^2)$.

Datele de ieşire sunt scrise pe parcursul determinării soluțiilor, deci operația de scriere a acestora nu consumă timp suplimentar.

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

1.3.6. Graf

Pentru a determina cel mai îndepărtat nod față de cel identificat prin valoarea 1 va trebui să realizăm o parcurgere în lățime (***B**readth **F**irst – **BF***) a grafului.

O astfel de parcurgere implică atingerea nodurilor mai apropiate de nodul 1 înaintea atingerii nodurilor mai îndepărtate.

Ca urmare, ultimul nod atins de o astfel de parcurgere va fi întotdeauna unul dintre cele mai îndepărtate noduri, chiar dacă există şi alte noduri aflate la aceeaşi distanță.

După determinarea ultimului nod parcurs vom scrie în fişierul de ieşire numărul său de ordine.

Analiza complexității

Intrarea constă în citirea informațiilor referitoare la cele m muchii ale grafului, deci ordinul de complexitate al operației de citire a datelor este $O(m)$. Pe parcursul citirii este construită şi structura de date în care va fi păstrat graful, deci nu este consumat timp suplimentar pentru generarea acesteia.

Operația de parcurgerea în lățime a unui graf are ordinul de complexitate $O(m)$.

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

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(m) + O(m) + O(1) = O(m)$.

1.4. Soluțiile problemelor propuse spre rezolvare la Olimpiada Națională de Informatică, 2002

1.4.1. Reconstrucție

Rezolvarea acestei probleme este foarte simplă, ea neimplicând decât parcurgerea pe coloane a unei matrice, contorizarea unor secvențe de zerouri şi păstrarea unei statistici referitoare la aceste secvențe.

Pentru a determina blocurile necesare, vom determina blocurile pe fiecare coloană (datorită faptului că lățimea unui bloc este întotdeauna 1 şi toate blocurile sunt dispuse pe verticală, un bloc poate ocupa o singură coloană).

Pentru a determina blocurile de pe o coloană va trebui să determinăm secvențele de zerouri de pe coloana respectivă. Vom lua în considerare doar secvențele de lungime maximă pentru a minimiza numărul total de blocuri. De exemplu, dacă avem şase zerouri consecutive, am putea folosi un bloc de lungime 6, dar şi două blocuri de lungime 5 şi 1, 4 şi 2 etc. Evident, este obligatoriu să folosim un singur bloc pentru ca numărul total al blocurilor utilizate să fie minim.

Aşadar, pentru fiecare coloană vom determina lungimile secvențelor de zerouri. O secvență de zerouri poate începe fie pe prima linie a coloanei, fie în momentul în care întâlnim o linie pe care se află un element cu valoarea 0, în timp ce pe linia anterioară se află un element cu valoarea 1. Secvența se va termina fie la terminarea parcurgerii coloanei (se ajunge pe ultima linie a acesteia), fie în momentul în care întâlnim o linie pe care se află un element cu valoarea 1, în timp ce pe linia anterioară se află un element cu valoarea 0.

În momentul detectării terminării unei secvențe (presupunem că lungimea acesteia este x), numărul blocurilor de lungime x este incrementat. Pentru păstrarea numărului de blocuri se utilizează un şir a, unde a_x indică numărul blocurilor de lungime x.

La sfârşit, vom afişa statistica cerută pe baza datelor păstrate în şirul a. Vor fi afişate toate perechile de forma $i\ a_i$ care respectă condiția $a_i \neq 0$.

Analiza complexității

Datorită faptului că numărul de elemente care compun o secvență de zerouri poate fi determinat pe măsură ce se parcurge secvența, întregul algoritm constă într-o singură traversare a matricei. Ordinul de complexitate al unei astfel de traversări este $O(m \cdot n)$, unde m reprezintă numărul de linii ale matricei, iar n reprezintă numărul de coloane.

Pentru citirea datelor, matricea este parcursă o singură dată, deci ordinul de complexitate al acestei operații este tot $O(m \cdot n)$.

Pentru afişarea datelor se parcurge o singură dată şirul a; acesta nu poate conține mai mult de m elemente, deoarece nu pot fi folosite blocuri mai înalte decât înălțimea zidului. Aşadar, scrierea soluției are ordinul de complexitate $O(m)$.

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

1.4.2. Pod

Rezolvarea problemei se bazează pe o variantă simplă a metodei *programării dinamice*. Se observă foarte uşor că numărul de posibilități de a ajunge pe cea de-a i-a scândură depinde doar de numărul de posibilități de a ajunge pe cele trei scânduri aflate în fața ei. Vom nota cu t_i numărul de posibilități de a ajunge pe cea de-a i-a scândură. Vom considera malul opus ca fiind cea de-a $(N + 1)$-a scândură, unde N este numărul scândurilor care formează podul. Soluția problemei va fi dată de valoarea t_{N+1}.

În cazul în care cea de-a i-a scândură lipseşte, pe ea nu se poate ajunge, deci vom avea $t_i = 0$.

În cazul în care această scândură există, dar este deteriorată, pe ea se poate ajunge doar de pe scândura precedentă, deci vom avea $t_i = t_{i-1}$.

În cazul în care scândura există şi nu este deteriorată, pe ea se poate ajunge de pe scândura anterioară (chiar dacă este deteriorată) sau de pe oricare dintre precedentele două (dacă nu sunt deteriorate). Pentru a exprima relația matematică pentru t_i, vom folosi funcția şi definită prin $s_i = t_i$,dacă a i-a scândură există şi nu este deteriorată şi $s_i = 0$ în caz contrar. Folosind această funcție, relația este $t_i = s_{i-3} + s_{i-2} + t_{i-1}$.

Inițial, vom avea $t_0 = 1$, deoarece se poate spune că există o singură posibilitate de a ajunge pe malul pe care ne aflăm.

Datorită faptului că valorile t_i pot avea până la 80 de cifre, este necesară simularea operației de adunare pentru numere mari.

Determinarea unei soluții corecte se poate realiza destul de uşor dacă, la fiecare pas, păstrăm indicele unei scânduri anterioare de pe care se poate trece pe scândura curentă. Acest indice va fi fie cel al scândurii anterioare (dacă aceasta există şi numărul posibilităților de a ajunge la ea este nenul), fie al uneia dintre precedentele două (dacă există, nu este deteriorată şi numărul posibilităților de a ajunge la ea este nenul). Dacă valoarea t_{N+1} este nenulă, atunci există cu siguranță cel puțin o soluție. În final, modalitatea de traversare va fi generată cu ajutorul unei tehnici recursive foarte simple.

Analiza complexității

Operația de adunare a numerelor mari are ordinul de complexitate $O(NCif)$, unde $NCif$ este numărul de cifre al celui mai mare dintre numerele care se adună. Deoarece $NCif$ este cel mult 80, ordinul de complexitate al operației de adunare poate fi considerat a fi $O(80) = 80 \cdot O(1) = O(1)$.

Trebuie efectuate cel mult $2 \cdot N$ astfel de adunări, deci operația de determinare a valorilor t_i are ordinul de complexitate $O(N) \cdot O(1) = O(N)$.

Pentru reconstituirea drumului, la fiecare pas trebuie păstrat indicele unei scânduri precedente de pe care se poate ajunge pe scândura curentă. Există doar trei posibilități de a ajunge pe scândura curentă, deci ordinul de complexitate al acestei operații este $O(1)$. Pentru determinarea scândurii anterioare corespunzătoare fiecărei scânduri din componența podului sunt efectuate $O(N)$ astfel de operații, deci ordinul de complexitate al operației de determinare a unei modalități de traversare este $O(N)$.

Citirea datelor de intrare se realizează într-un timp cu ordinul de complexitate $O(N)$, deoarece pot exista cel mult $N - 1$ scânduri care lipsesc şi cel mult N care sunt deteriorate.

Scrierea datelor de ieşire constă în generarea unei modalități de traversare (care are lungimea cel mult N) şi a numărului modalităților de traversare. Deoarece determinarea scândurii precedente se realizează pe baza unor indici păstrați pentru fiecare scândură în parte, ordinul de complexitate al acestei operații este $O(1)$. Datorită faptului că vor fi cel mult N astfel de determinări, ordinul de complexitate al operației de determinare a modalității de traversare este O(N). Scrierea numărului posibilităților de traversare poate fi considerat a fi o operație elementară, deci are ordinul de complexitate $O(1)$. Aşadar, ordinul de complexitate al operației de scriere a datelor de ieşire este $O(N) + O(1) = O(N)$.

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

1.4.3. Suma

Pentru ca numărul termenilor să fie minim, trebuie ca suma termenilor care apar cu semn negativ să fie cât mai mică posibil.

Inițial vom presupune că nu va trebui să scădem nici o valoare şi ne propunem să găsim cel mai mic număr N pentru care:

$$S_N = \sum_{i=1}^{N} i \geq S .$$

Valoarea N se obține rezolvând ecuația de gradul II $S_N = S$ (vom avea $S_N \leq S + N$) şi rotunjind prin adaos rădăcina pozitivă obținută. Avem:

$$\frac{N \cdot (N+1)}{2} = S$$

$$N^2 + N - 2 \cdot S = 0$$

$$N = \frac{\sqrt{1 + 8 \cdot S} - 1}{2}.$$

Aşadar, valoarea N este:

$$N = \left\lceil \frac{\sqrt{1 + 8 \cdot S} - 1}{2} \right\rceil.$$

Datorită faptului că N este cea mai mică valoare pentru care suma atinge sau depăşeşte valoarea S, soluţia problemei nu poate fi dată de nici un număr $N' < N$. În cele ce urmează vom demonstra că soluţia problemei este dată de N, $N + 1$ sau $N + 2$.

Se observă că numerele S_N, S_{N+1} şi S_{N+2} nu pot avea toate aceeaşi paritate, deoarece $N + 1$ şi $N + 2$ au parităţi diferite. Vom nota prin D_N diferenţa dintre valoarea S_N şi valoarea care trebuie obţinută (S). Deoarece se scade aceeaşi valoare din S_N, S_{N+1} şi S_{N+2}, este evident că D_N, D_{N+1} şi D_{N+2} nu pot avea toate aceeaşi paritate.

Soluţia problemei este dată de indicele celui mai mic număr par dintre aceste trei numere; fie acest indice N'. Suma totală a elementelor cărora trebuie să le fie modificat semnul este $D_{N'} / 2$. Prin modificarea semnului unui element, valoarea expresiei scade cu dublul acestui element, deci avem $S_{N'} - D_{N'} / 2 \cdot 2 = S$.

Fie $x = D_{N'} / 2$ (suma totală a elementelor care trebuie scăzute). Deoarece $N' \le N + 2$, obţinem $S_{N'} \le S_N + N + 1 + N + 2$. Folosind relaţia $S_N \le S + N$, se obţine relaţia $S_{N'} \le S + 3 \cdot N + 3$. Cu alte cuvinte, avem:

$$x \le \frac{3 \cdot N + 3}{2} \le \frac{3 \cdot N' + 3}{2}.$$

Aşadar, valoarea x poate fi obţinută alegând doar două numere cuprinse între 1 şi N'. În cazul în care $x > N'$ vom scădea N' şi $N' - x$, iar în caz contrar vom scădea doar x.

Analiza complexităţii

Valoarea N poate fi obţinută folosind o simplă formulă matematică, deci ordinul de complexitate al acestei operaţii este $O(1)$.

Valorile S_N, S_{N+1}, S_{N+2}, D_N, D_{N+1}, D_{N+2} sunt calculate tot cu ajutorul unor formule matematice simple, aşadar ordinul de complexitate al operaţiei de determinare a acestora este tot $O(1)$.

Identificarea valorii N' se face pe baza verificării parităţii a cel mult trei numere, deci şi această operaţie are ordinul de complexitate $O(1)$.

Determinarea valorii x şi a celor cel mult două numere cărora li se va modifica semnul este realizată tot pe baza unor calcule simple al căror ordin de complexitate este $O(1)$.

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

1.4.4. Becuri

Este evident că pentru a obține prima linie a matricei, conform cerințelor, nu putem decât fie să comutăm toate coloanele pe care trebuie să se afle becuri aprinse şi să nu comutăm prima linie, fie să comutăm prima linie şi să comutăm toate coloanele pe care trebuie să se afle becuri stinse. Analog, pentru prima coloană putem fie să comutăm toate liniile pe care trebuie să se afle becuri aprinse şi să nu comutăm prima coloană, fie să comutăm prima coloană şi să comutăm toate liniile pe care trebuie să se afle becuri stinse. Astfel avem patru posibilități de a obține configurația cerută:

- comutăm liniile cărora le corespund becuri aprinse pe prima coloană şi coloanele cărora le corespund becuri aprinse pe prima linie;
- comutăm liniile cărora le corespund becuri aprinse pe prima coloană şi coloanele cărora le corespund becuri stinse pe prima linie;
- comutăm liniile cărora le corespund becuri stinse pe prima coloană şi coloanele cărora le corespund becuri aprinse pe prima linie;
- comutăm liniile cărora le corespund becuri stinse pe prima coloană şi coloanele cărora le corespund becuri stinse pe prima linie.

Vom verifica fiecare dintre aceste patru variante şi apoi o vom alege pe cea care implică cele mai puține comutări de linii şi coloane.

Dacă nici una dintre cele patru variante nu duce la obținerea configurației cerute, putem trage concluzia că aceasta nu poate fi obținută prin comutarea unor linii şi a unor coloane.

Analiza complexității

Citirea datelor de intrare implică parcurgerea unei matrice pătratice, deci ordinul de complexitate al acestei operații este $O(N^2)$.

Verificarea fiecăreia dintre cele patru posibilități de a obține soluția implică parcurgerea primei linii şi a primei coloane; ordinul de complexitate al unei astfel de parcurgere este $O(N) + O(N) = O(N)$. Pentru fiecare element este posibil ca linia/coloana corespunzătoare să fie comutată. Operația de comutare implică traversarea liniei sau coloanei, aşadar are ordinul de complexitate $O(N)$. Ca urmare, ordinul de complexitate al fiecăreia dintre cele patru posibilități este $O(N) \cdot O(N) = O(N^2)$. Întreaga operație de determinare a soluției are ordinul de complexitate $4 \cdot O(N^2) = O(N^2)$.

Scrierea datelor de ieşire implică traversarea şirurilor care indică dacă o linie sau coloană a fost comutată; ordinul de complexitate al operației este $O(N)$.

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

1.4.5. Discuri

Pentru identificarea discurilor dispensabile va trebui să determinăm atingerile dintre discurile care influențează lățimea figurii. De exemplu, dacă avem şase discuri, cu raze de 1000, 1, 2, 3, 1000, respectiv 500, atunci atingerile care influențează lățimea figurii sunt între primul şi al cincilea disc, respectiv între al cincilea şi al şaselea.

Pentru fiecare disc i vom determina, pe rând, coordonatele orizontale pe care le-ar avea discul dacă ar atinge unul dintre discurile anterioare. În final, vom alege discul (sau axa Oy) pentru care coordonata orizontală a centrului noului disc este maximă şi putem afirma că discul i ajunge în această poziție. Dacă discul i va atinge un disc anterior j, atunci discurile cu numerele de ordine cuprinse între $j + 1$ şi $i - 1$ sunt dispensabile.

După ce vom lua în considerare toate cele N discuri, vom putea determina numerele de ordine ale tuturor discurilor dispensabile.

În final vom verifica dacă există discuri introduse la sfârşit care sunt dispensabile. Pentru aceasta vom determina lățimea figurii şi ultimul disc care o influențează. Toate discurile introduse după acest disc sunt dispensabile.

Pentru determinarea coordonatei orizontale x_i a centrului unui disc i care atinge un disc j avem nevoie de coordonata orizontală x_j a centrului discului j, precum şi de razele r_i şi r_j ale celor două discuri.

Dacă aceste trei valori sunt cunoscute, se poate folosi următoarea formulă pentru a determina coordonata orizontală a centrului discului j:

$$x_j = x_i + \sqrt{(r_i + r_j)^2 - (r_i - r_j)^2} \ .$$

Analiza complexității

Pentru fiecare disc i care este introdus, se determină posibila coordonată x_i datorată atingerii cu toate cele $i - 1$ discuri inserate anterior. Pentru cele N discuri se vor determina, în total, $N \cdot (N + 1) / 2$ coordonate, deci ordinul de complexitate al acestei operații este $O(N^2)$.

La fiecare pas, pot fi marcate ca dispensabile cel mult toate discurile inserate anterior, aşadar ordinul de complexitate al acestei operații este tot $O(N^2)$.

Determinarea lățimii figurii şi a cercurilor dispensabile de la sfârşitul secvenței necesită o singură parcurgere a şirului care păstrează coordonatele centrelor discurilor, ceea ce implică ordinul de complexitate $O(N)$.

Afişarea cercurilor dispensabile, precum şi citirea razelor cercurilor sunt operații care se efectuează în timp liniar, necesitând o simplă parcurgere a unor şiruri.

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

1.4.6. Cod

Din condițiile precizate în enunț rezultă că, pe baza unei mulțimi de litere distincte, se poate construi un singur cuvânt care respectă condițiile date, şi anume cel care conține literele ordonate lexicografic. Aşadar, oricărei mulțimi de cel mult zece litere distincte îi corespunde un cuvânt care respectă condițiile din enunț. Se poate afirma că un cuvânt este o submulțime a mulțimii literelor; de aici rezultă că numărul cuvintelor formate din k litere este C_{26}^{k}. Mai mult, numărul cuvintelor formate din k dintre ultimele n litere ale alfabetului este C_{n}^{k}.

Numărul de ordine al cuvântului dat este mai mare decât cel al codurilor formate din mai multe cifre. Din aceste motive, pentru un cuvânt format din k litere, vom avea un cod mai mare decât $\sum_{i=1}^{k} C_{26}^{i}$.

În continuare, pentru prima literă a cuvântului, va trebui să găsim numărul cuvintelor care încep cu o literă mai mică (din punct de vedere lexicografic). În cazul în care cuvântul are k litere, vor exista C_{25}^{k-1} cuvinte valide care încep cu 'a', C_{24}^{k-1} cuvinte valide care încep cu 'b' etc. În general, vor exista C_{26-i}^{k-1} cuvinte valide care încep cu a i-a literă a alfabetului. Dacă prima literă a cuvântului este cea de-a n-a literă a alfabetului, vom avea $\sum_{i=1}^{n} C_{26-i}^{k-1}$ cuvinte valide care încep cu o literă mai mică.

În acest moment ştim numărul de ordine minim al unui cuvânt care începe cu prima literă a cuvântului dat. Pentru a doua literă vom proceda într-o manieră asemănătoare. Singura diferență este dată de faptul că a doua literă trebuie să fie strict mai mare decât prima. Aşadar, dacă prima literă este cea de-a n-a a alfabetului, iar a doua este cea de-a m-a, atunci vom avea $\sum_{i=n+1}^{m-1} C_{26-i}^{k-2}$ cuvinte care au pe prima poziție aceeaşi literă, iar pe cea de-a doua poziție o literă mai mică.

Procedeul va continua pentru fiecare literă în parte. În cazul în care litera curentă este cea de-a p-a a cuvântului, este a m-a literă a alfabetului, iar litera anterioară este a n-a literă a alfabetului, numărul de cuvinte care au pe primele $p - 1$ poziții aceleaşi litere ca şi cuvântul dat, iar pe cea de-a p-a poziție o literă mai mică, este dat de formula $\sum_{i=n+1}^{m-1} C_{26-i}^{k-p}$.

Adunând toate valorile obținute pe parcurs vom obține numărul cuvintelor care se află înaintea cuvântului dat. Adunând 1 la această valoare, vom obține numărul de ordine al cuvântului.

Analiza complexității

Pentru a analiza complexitatea acestui algoritm va trebui să precizăm faptul că numărul literelor din alfabet este constant, deci nu poate interveni în exprimarea ordinului de complexitate. Acesta va fi stabilit doar în funcție de lungimea *k* a cuvântului.

Inițial se calculează suma $\sum_{i=1}^{k} C_{26}^{i}$, operație realizabilă în timp liniar, având în vedere observația anterioară. Aşadar, primul pas al algoritmului are ordinul de complexitate $O(k)$.

Pentru fiecare literă a cuvântului se calculează suma $\sum_{i=n+1}^{m-1} C_{26-i}^{k-2}$, unde variabilele au semnificația prezentată anterior. Numărul de litere este implicat în determinarea valorii combinării. Aşadar, calculul combinării se realizează în timp liniar. Numărul de combinări calculate nu depinde de lungimea cuvântului, deci ordinul de complexitate al calculării acestei sume este $O(k)$. În total vor fi calculate *k* astfel de sume, deci ordinul de complexitate al celui de-al doilea pas al algoritmului este $O(k) \cdot O(k) = O(k^2)$.

Citirea datelor de intrare şi scrierea celor de ieşire se realizează cu ajutorul unor operații elementare, deci putem considera că au ordinul de complexitate $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(k) + O(k^2) + O(1) = O(k^2)$, având în vedere că numărul literelor din alfabetul englez este constant.

1.4.7. Hotel

Pentru început, vom determina numărul de etaje la care există angajați şi vom atribui fiecărui etaj câte o culoare. Pe măsură ce citim datele referitoare la angajați, vom verifica dacă etajul la care lucrează angajatul are asociată o culoare şi, dacă este cazul, îi vom atribui o culoare şi vom creşte numărul etajelor distincte. Dacă ajungem în situația în care numărul culorilor disponibile este mai mic decât numărul etajelor distincte, atunci problema nu are soluție. Atribuind câte o culoare fiecărui etaj, practic, am rezolvat cea de-a doua parte a problemei.

Pentru prima parte vom considera că avem la dispoziție *n* culori şi există *k* etaje distincte la care lucrează angajați. Va trebui să determinăm numărul de posibilități de a atribui celor *k* etaje câte o culoare, astfel încât fiecare etaj să aibă propria sa culoare.

Practic, având la dispoziție o mulțime formată din *n* elemente va trebui să determinăm câte posibilități de alegere a *k* dintre aceste elemente există. Este foarte uşor de observat că, de fapt, propoziția anterioară reprezintă definiția noțiunii matematice de aranjamente. Aşadar, soluția primei părți a problemei este dată de formula:

$$A_n^k = \frac{n!}{(n-k)!} = (n-k+1)\cdot(n-k+2)\cdot\cdots\cdot(n-1)\cdot n.$$

Datorită faptului că se obțin numere foarte mari, nu se pot folosi tipurile de date puse la dispoziție de limbajele de programare pentru operații aritmetice. Se observă că, folosind formula anterioară, avem nevoie doar de înmulțiri dintre un număr mare şi un număr cel mult egal cu 200. Ca urmare, va trebui să implementăm operația de înmulțire a unui număr mare cu un scalar. Pentru a mări viteza de execuție a programului şi a utiliza mai eficient memoria, se poate folosi baza 10000 pentru efectuarea calculelor.

Analiza complexității

Citirea datelor de intrare se realizează în timp liniar, deci ordinul de complexitate al acestei operații este $O(n)$.

Verificarea faptului că unui etaj îi corespunde o culoare şi eventuala atribuire a unei culori se realizează în timp constant. Deoarece această operație se realizează pentru etajul corespunzător fiecărui angajat, stabilirea corespondenței dintre etaje şi culori se realizează într-un timp cu ordinul de complexitate $O(n) \cdot O(1) = O(n)$.

Calcularea numărului de modalități de alegere a culorilor implică folosirea numerelor mari. Se observă că numărul de cifre (în baza 10000) al rezultatului este mai mic decât 100, aşadar putem considera că o înmulțire se realizează în timp liniar. Numărul înmulțirilor efectuate este k, deci ordinul de complexitate al operației de determinare a rezultatului este $O(k)$.

Afişarea numărului de modalități este realizată în timp constant, iar operația de afişare a culorilor corespunzătoare fiecărui angajat are ordinul de complexitate $O(n)$.

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

1.4.8. Lac

Pentru rezolvarea acestei probleme vom folosi o variantă puțin modificată a algoritmului lui *Lee*. Pentru fiecare poziție a matricei care reprezintă o zonă mlăştinoasă vom determina numărul minim de pontoane care trebuie să fie amplasate pentru a se ajunge în poziția respectivă.

Pentru aceasta vom considera că pentru a ajunge în imediata vecinătate a primei linii a matricei numărul de pontoane necesare este 0. Pentru o anumită poziție, numărul de pontoane necesare ajungerii în punctul respectiv este dat de cel mai mic număr corespunzător uneia dintre pozițiile învecinate la care se adaugă, eventual, un ponton dacă poziția nu corespunde unei porțiuni de uscat.

După o astfel de parcurgere a matricei, vom avea pentru fiecare poziție a matricei o anumită valoare, dar nu suntem siguri că aceasta este cea minimă. Vom parcurge succesiv matricea încercând să îmbunătățim valorile obținute. O valoare va fi modificată dacă se poate ajunge în poziția curentă dintr-o poziție învecinată şi se obține un număr mai mic de pontoane pentru poziția curentă. În momentul în care nu va mai exista nici o parcurgere care să aducă îmbunătățiri, am obținut rezultatul final.

Practic cele două tipuri de parcurgeri pot fi "asimilate" în una singură dacă, inițial, se marchează toate pozițiile matricei cu o valoare suficient de mare. De fiecare dată când are loc o îmbunătățire, vom păstra direcția din care s-a ajuns în poziția curentă pentru a putea reconstitui drumul.

În final, numărul minim de pontoane va fi dat de cea mai mică valoare de pe ultima linie a matricei. Pentru reconstituirea drumului, se va porni în sens invers, de pe poziția de pe ultima linie în poziția din care s-a ajuns în ea şi aşa mai departe, până se ajunge pe prima linie. Dacă se ajunge într-o poziție care nu corespunde unei zone de uscat, atunci va trebui amplasat un ponton în acea poziție.

Analiza complexității

Citirea datelor de intrare implică o traversare a matricei, ordinul de complexitate al acestei operații fiind $O(m \cdot n)$.

O parcurgere a matricei în vederea realizării unei îmbunătățiri are acelaşi ordin de complexitate. Datorită faptului că, după prima parcurgere, suntem siguri că se poate ajunge pe ultima linie folosind m pontoane (completarea integrală a coloanei), nu se vor realiza în nici o situație mai mult de m parcurgeri de îmbunătățire. Din aceste motive, ordinul de complexitate al operației de determinare a numărului de pontoane necesare pentru ajungerea în fiecare poziție este $O(m) \cdot O(m \cdot n) = O(m^2 \cdot n)$.

Determinarea valorii minime de pe ultima linie se realizează în timp liniar, ordinul de complexitate fiind $O(n)$.

Reconstituirea drumului ar putea, teoretic, să necesite traversarea întregii matrice. Astfel, în cel mai defavorabil caz, ordinul de complexitate al acestei operații este $O(m \cdot n)$.

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

1.4.9. Logic

Pentru început vom observa că, pentru ca trasarea să fie posibilă, fiecare zonă trebuie să fie delimitată de un număr par de segmente. Aceasta se datorează faptului că linia trebuie să intre şi să iasă din fiecare zonă de un anumit număr de ori.

Aşadar, pentru fiecare zonă va trebui să numărăm segmentele (aşa cum sunt definite ele în enunț) care o delimitează şi să verificăm dacă numărul obținut este par sau impar. În momentul în care găsim o zonă delimitată de un număr par de segmente, vom şti că trasarea nu este posibilă. Dacă toate zonele sunt delimitate de un număr par de segmente, atunci trasarea este posibilă.

Numărarea segmentelor care delimitează o zonă se realizează pe baza unui algoritm de umplere. În momentul în care ajungem la un punct de la marginea zonei, se pune problema creşterii numărului de segmente. Există mai multe situații în care numărul de segmente va creşte:

- se ajunge în partea de sus a zonei, iar punctul aflat imediat deasupra şi cel din dreapta acestuia fac parte din zone diferite;
- se ajunge în partea de jos a zonei, iar punctul aflat imediat dedesubt şi cel din dreapta acestuia fac parte din zone diferite;
- se ajunge în partea din stânga a zonei, iar punctul aflat imediat la stânga şi cel de deasupra sa fac parte din zone diferite;
- se ajunge în partea din dreapta a zonei, iar punctul aflat imediat la dreapta şi cel de deasupra sa fac parte din zone diferite;
- se ajunge într-unul din colțurile zonei.

Dacă ultima condiție este îndeplinită simultan cu una dintre primele patru, numărul segmentelor va creşte doar cu 1. Creşterile datorate primelor patru condiții sunt cumulative.

Analiza complexității

Vom studia acum complexitatea algoritmului pentru un desen. Citirea datelor de intrare implică o traversare a matricei, deci această operație se realizează într-un timp de ordinul $O(n^2)$.

La fiecare pas al algoritmului de umplere se numără segmentele care trebuie adăugate pentru poziția curentă. Se pot adăuga cel mult patru segmente, aşadar ordinul de complexitate al operației este $O(1)$. Algoritmul de umplere implică parcurgerea integrală a matricei, motiv pentru care vor fi vizitate toate cele n^2 poziții. Aşadar, algoritmul de verificare a posibilității de trasare a liniei are ordinul de complexitate $O(n^2) \cdot O(1) = O(n^2)$.

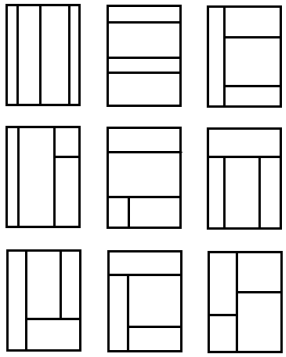
Datele de ieşire constau din scrierea unui singur mesaj, deci această operație se efectuează în timp liniar.

În concluzie, algoritmul de rezolvare a acestei probleme, pentru un desen, are ordinul de complexitate $O(n^2) + O(n^2) + O(1) = O(n^2)$.

Pentru a stabili ordinul de complexitate al algoritmului de rezolvare a întregii probleme, vom considera că un set de date cuprinde t desene. Aşadar, ordinul de complexitate este $O(t) \cdot \mathbf{O}(n^2) = O(n^2 \cdot t)$.

1.4.10. Foto

Se observă foarte uşor că există doar nouă posibilități de lipire a patru fotografii pe o pagină, indiferent de dimensiunile acestora. Toate celelalte posibilități sunt echivalente cu una dintre cele nouă, ele obținându-se prin translatări sau oglindiri. Cele nouă aranjamente posibile pe care le vom lua în considerare sunt prezentate în figura următoare.



Modalități de amplasare

Va trebui să alegem pozițiile celor patru fotografii pentru fiecare dintre cele nouă configurații. Teoretic, pentru fiecare dintre cele nouă configurații, există 4! = 24 posibilități. Practic, se observă că există şi în acest caz configurații echivalente. De exemplu, pentru prima şi a doua configurație, nu contează ordinea în care sunt aşezate fotografiile, pentru a treia configurație nu contează ordinea în care sunt dispuse cele trei poze de pe a doua coloană etc. Practic, pentru fiecare configurație vom avea 1, 2, 4, 6 sau 12 posibilități.

Cu excepția ultimei configurații, în toate celelalte există o fotografie care ocupă o întreagă latură a paginii. Cunoscând raportul dintre lungimea şi lățimea sa se determină laturile spațiului liber rămas pe foaie. În continuare, una dintre laturile spațiului este ocupat integral de o fotografie. Procedura continuă până determinăm dimensiunile tuturor fotografiilor. În final se verifică dacă foaia este acoperită integral. Pentru ultima configurație vom alege, pe rând, dimensiunile posibile ale fotografiei din colțul din stânga-sus şi dimensiunile celorlalte fotografii vor fi determinate în acelaşi mod ca şi în situațiile anterioare.

Analiza complexității

Citirea datelor se realizează în timp constant, deoarece numărul fotografiilor este întotdeauna 4. Studierea primelor opt configurații se realizează, de asemenea, în timp constant, deoarece numărul posibilităților de amplasare a fotografiilor respectând configurațiile considerate este constant.

Pentru cea de-a noua configurație putem avea cel mult $min(X, Y)$ dimensiuni posibile ale primei fotografii, unde X şi Y sunt dimensiunile paginii. Dacă notăm cu n acest minim, ordinul de complexitate al acestei operații va fi $O(n)$.

Datele de ieşire constau în exact opt numere, deci scrierea rezultatelor se realizează în timp constant.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(1) + 8 \cdot O(1) + O(n) + O(1) = O(n)$.

1.4.11. Balanță

Se observă că suma momentelor greutăților este cuprinsă între -6000 şi 6000 (dacă avem 20 de greutăți cu valoarea 20 şi acestea sunt amplasate pe cel mai îndepărtat cârlig față de centrul balanței, atunci modulul sumei momentelor forțelor este $15 \cdot 20 \cdot 20 = 6000$). Ca urmare, putem păstra un şir a ale cărui valori a_i vor conține numărul posibilităților ca suma momentelor greutăților să fie i. Indicii şirului vor varia între -6000 şi 6000. Pentru a îmbunătăți viteza de execuție a programului, indicii vor varia între $-300 \cdot g$ şi $300 \cdot g$, unde g este numărul greutăților. Pot fi realizate îmbunătățiri suplimentare dacă se determină distanțele maxime față de mijlocul balanței ale celor mai îndepărtate cârlige de pe cele două talere şi suma totală a greutăților. Dacă distanțele sunt d_1 şi d_2, iar suma este s, atunci indicii vor varia între $-d_1 \cdot s$ şi $d_2 \cdot s$.

Inițial, pe balanță nu este agățată nici o greutate, aşadar suma momentelor greutăților este 0. Ca urmare, inițial valorile a_i vor fi 0 pentru orice indice nenul şi $a_0 = 1$ (există o posibilitate ca inițial suma momentelor greutăților să fie 0 şi nu există nici o posibilitate ca ea să fie diferită de 0).

În continuare, vom încerca să amplasăm greutățile pe cârlige. Fiecare greutate poate fi amplasată pe oricare dintre cârlige. Să presupunem că la un moment dat există a_i posibilități de a obține suma i. Dacă vom amplasa o greutate de valoare g pe un cârlig aflat la distanța d față de centrul balanței, suma momentelor greutăților va creşte sau va scădea cu $g \cdot d$ (în funcție de brațul pe care se află cârligul). Ca urmare, după amplasarea noii greutăți există a_i posibilități de a obține suma $i + g \cdot d$. Considerăm că şirul b va conține valori care reprezintă numărul posibilităților de a obține sume ale momentelor forțelor după amplasarea greutății curente. Înainte de a testa posibilitățile de plasare a greutății, şirul b va conține doar zerouri. Pentru fiecare pereche (i, d), valoarea $b_i + g \cdot d$ va creşte cu a_i. După considerarea tuturor perechilor, vom putea trece la o nouă greutate. Valorile din şirul b vor fi salvate în şirul a, iar şirul b va fi reinițializat cu 0. Rezultatul final va fi dat de valoarea a_0 obținută după considerarea tuturor greutăților disponibile.

Analiza complexității

Pentru studiul complexității vom nota numărul greutăților cu g, iar cel al cârligelor cu c.

Citirea datelor de intrare corespunzătoare cârligelor şi greutăților se realizează în timp liniar, deci ordinul de complexitate al acestor operații este $O(g)$, respectiv $O(c)$.

Singura mărime care nu este considerată constantă şi de care depinde numărul de posibilități de a obține sumele este numărul greutăților. Aşadar, vom considera că ordinul de complexitate al traversării şirului a este $O(g)$. Numărul de traversări este dat tot de numărul greutăților disponibile, deci vom avea $O(g)$ traversări. În timpul traversării vom considera toate cârligele pentru fiecare element al şirului. Ca urmare, ordinul de complexitate al operațiilor efectuate asupra unui element într-o parcurgere este $O(c)$. Rezultă că ordinul de complexitate al unei parcurgeri este $O(g) \cdot O(c) = O(g \cdot c)$,

în timp ce ordinul de complexitate al întregii operații care duce la obținerea rezultatului este $O(g) \cdot O(g \cdot c) = O(g^2 \cdot c)$.

Afişarea numărului de posibilități de a echilibra balanța se realizează în timp constant.

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

1.4.12. Aliniere

Pentru fiecare soldat vom determina cel mai lung subşir strict crescător (din punct de vedere al înălțimii) de soldați care se termină cu el, respectiv cel mai lung subşir strict descrescător de soldați care urmează după el. După această operație, vom determina soldatul pentru care suma lungimilor celor două şiruri este maximă.

Chiar dacă s-ar părea că în acest mod am găsit soluția problemei, mai există o posibilitate de a mări numărul soldaților care rămân în şir. Să considerăm soldatul cel mai înalt în şirul rămas (cel căruia îi corespunde suma maximă). Acesta poate privi fie spre stânga, fie spre dreapta şirului. Din aceste motive, la stânga sau la dreapta sa poate să se afle un soldat de aceeaşi înălțime; unul dintre cei doi va privi spre dreapta, iar celălalt spre stânga. Totuşi, nu putem alege orice soldat cu aceeaşi înălțime, ci doar unul pentru care lungimea şirului strict crescător (dacă se află spre stânga) sau a celui strict descrescător (dacă se află spre dreapta) este aceeaşi cu lungimea corespunzătoare şirului strict crescător, respectiv strict descrescător, corespunzătoare celui mai înalt soldat dintre cei rămaşi în şir.

După identificarea celor doi soldați de înălțimi egale (sau demonstrarea faptului că nu există o pereche de acest gen care să respecte condițiile date) se marchează toți soldații din cele două subşiruri. Ceilalți soldați vor trebui să părăsească formația.

Analiza complexității

Citirea datelor de intrare se realizează în timp liniar, deci ordinul de complexitate al acestei operații este $O(n)$.

Chiar dacă există algoritmi eficienți (care rulează în timp liniar-logaritmic) de determinare a celui mai lung subşir ordonat, timpul de execuție admis ne permite folosirea unui algoritm simplu, cu ordinul de complexitate $O(n^2)$. Acesta va fi aplicat de două ori, după care se va căuta valoarea maximă a sumei lungimilor şirurilor corespunzătoare unui soldat; aşadar identificarea soldatului care poate privi în ambele direcții este o operație cu ordinul de complexitate $O(n^2) + O(n^2) + O(n) = O(n^2)$.

Urmează eventuala identificare a unui alt soldat de aceeaşi înălțime care respectă condițiile referitoare la lungimile subşirurilor. Pentru aceasta se parcurge şirul soldaților de la soldatul identificat anterior spre extremități; operația necesită un timp liniar.

Determinarea celor două subşiruri se realizează în timp liniar dacă, în momentul construirii celor două subşiruri, se păstrează predecesorul, respectiv succesorul fiecărui soldat. În timpul parcurgerii subşirurilor sunt marcați soldații care rămân în formație.

Pentru scrierea datelor de ieşire se parcurge şirul marcajelor şi sunt identificați soldații care părăsesc formația. Ordinul de complexitate al acestei operații este $O(n)$.

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

1.4.13. Arbore

Vom spune că un nod este *rezolvat* dacă şi numai dacă el face parte din exact un ciclu. Similar, vom spune că un subarbore este rezolvat dacă şi numai dacă toate nodurile acestuia sunt rezolvate.

Vom numi *fir* un subarbore care este lanț (fiecare nod, cu excepția frunzei, are un singur fiu). Un fir este ilustrat în figura 1.

Fig. 1: Un fir

Se observă că un fir poate fi rezolvat foarte simplu prin unirea rădăcinii sale cu frunza sa. Evident, firul va putea fi rezolvat numai dacă el conține cel puțin două noduri. Procedeul este ilustrat în figura 2.

Fig. 2: Un fir rezolvat

Pentru a rezolva această problemă vom alege, pentru început, un nod care va fi rădăcina arborelui. Vom spune că un subarbore este potențial rezolvabil dacă toți fiii rădăcinii subarborelui (în număr de cel puțin doi) sunt rădăcini ale unor fire.

În continuare vom arăta modul în care poate fi rezolvat un nod potențial rezolvabil. Vom alege cele mai scurte două fire (acestea pot conține unul, două sau mai multe

noduri) şi vom uni printr-o muchie frunzele acestor fire. Astfel vom obține un ciclu care va conține nodurile din componența celor două fire, precum şi nodul potențial rezolvabil. Eventualele fire rămase (dacă sunt formate din cel puțin trei noduri) vor fi rezolvate aşa cum s-a arătat în figura anterioară. Se observă că, în cazul în care există mai mult de două fire care conțin unul sau două noduri, subarborele nu poate fi rezolvat şi problema nu are soluție. Rezolvarea unui subarbore potențial rezolvabil este ilustrată în figura 3.

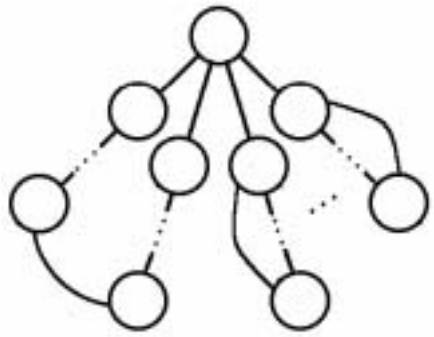


Fig. 3: Rezolvarea unui subarbore potențial rezolvabil

După rezolvarea unui subarbore, acesta este eliminat din arbore şi se va alege un alt nod potențial rezolvabil. Procedeul continuă până la rezolvarea tuturor nodurilor sau până la detectarea unui subarbore care nu poate fi rezolvat.

În cele ce urmează vom prezenta modul în care poate fi implementat algoritmul descris. Pentru început vom alege o modalitate de reprezentare a arborelui. Vom construi o listă de fii şi vom păstra, pentru fiecare nod, indicele primului şi al ultimului fiu în această listă. Prin convenție, vom considera că pentru un nod care nu are nici un fiu (frunză) indicele ultimului fiu va fi mai mic decât indicele primului fiu; de asemenea, pentru fiecare nod vom păstra nodul părinte. Datorită faptului că arborele este dat prin şirul muchiilor, va trebui să îl construim. Pentru aceasta vom realiza o parcurgere în lățime a arborelui, ceea ce permite construirea listei fiilor pe măsura traversării nodurilor arborelui.

În continuare vom alege rădăcina arborelui; aceasta poate fi orice nod care are gradul mai mare decât 1. Vom continua cu identificarea fiilor; evident, toate frunzele sunt fire de lungime 1. Pentru fiecare frunză vom "urca în arbore" atâta timp cât nodul curent are un singur fiu (părintele frunzei va fi rădăcina unui fir de lungime 2, părintele părintelui va fi rădăcina unui fir de lungime 3 şi aşa mai departe). Folosind acest procedeu vom determina şi lungimile firelor. Prin convenție, vom considera că dacă un nod nu face parte dintr-un fir, atunci lungimea corespunzătoare va fi -1.

Acum, la fiecare pas vom căuta un nod care conține cel puțin doi fii nerezolvați care sunt rădăcini ale unor fire. Vom determina numărul firelor de lungime 1 sau 2 şi,

în cazul în care există cel mult două astfel de fire, vom rezolva subarborele aşa cum se arată în figura 3. Dacă există mai mult de două astfel de fire vom scrie în fişierul de ieşire valoarea -1 şi vom opri execuția programului. După rezolvarea subarborelui, acesta este eliminat din arbore prin marcarea tuturor nodurilor sale ca fiind rezolvate. Acum părintele rădăcinii subarborelui ar putea să facă parte dintr-un fir. În cazul în care părintele a rămas cu un singur fiu care este rădăcină a unui fir, atunci părintele devine rădăcină a unui fir de lungime cu 1 mai mare. Vom "urca" din nou în arbore până la întâlnirea unui nod care are mai mulți fii nerezolvați şi vom păstra, pentru fiecare nod, lungimea firului cu rădăcina în nodul respectiv. O a doua posibilitate este ca părintele rădăcinii subarborelui eliminat să devină frunză. Se va "urca" în arbore în mod asemănător, dar noua frunză va fi rădăcina unui fir de lungime 1.

Există posibilitatea ca, la un moment dat, să rămânem cu un arbore format dintr-un singur fir. Dacă acesta are lungimea mai mare decât 2, atunci el va fi rezolvat aşa cum se arată în figura 2. În caz contrar, problema nu are soluție.

De fiecare dată când rezolvăm un subarbore, vom păstra o listă cu muchiile adăugate. În final, vom scrie în fişierul de ieşire numărul muchiilor adăugate, precum şi extremitățile muchiilor din listă.

Analiza complexității

Citirea datelor de intrare se realizează în timp liniar, deoarece implică citirea extremităților celor $N - 1$ muchii ale arborelui.

Construirea arborelui prin parcurgerea în lățime implică, pentru fiecare nod parcurs, studierea tuturor muchiilor. Deoarece avem N noduri şi $N - 1$ muchii, ordinul de complexitate al operației este $O(N^2)$. Pe parcursul construirii arborelui se creează şirul părinților, cel al fiilor, cel al gradelor, precum şi cele care păstrează indicii primului şi ultimului fiu pentru fiecare nod.

Pentru păstrarea informațiilor inițiale despre fire, vom parcurge, în cel mai defavorabil caz, toate nodurile arborelui, deci operația se realizează în timp liniar. Trebuie observat faptul că acest ordin de complexitate include şi timpul consumat pentru actualizarea necesară după eliminarea unui nod din arbore.

Alegerea unui nod potențial rezolvabil implică traversarea listei fiilor pentru a verifica dacă sunt rădăcini ale unor fire, deci ordinul său de complexitate este $O(N)$. La fiecare pas vom elimina cel puțin trei noduri, deci vom realiza cel mult $N / 3$ astfel de căutări. Ca urmare, ordinul de complexitate al tuturor alegerilor este $O(N^2)$.

Pentru rezolvarea arborelui este necesară parcurgerea tuturor nodurilor acestuia. Un nod rezolvat nu va fi parcurs încă o dată. Aşadar, ordinul de complexitate al acestei operații este $O(N)$.

Ordinul de complexitate al operațiilor de eliminare a nodurilor este tot $O(N)$ pentru că fiecare nod este eliminat o singură dată.

Scrierea datelor în fişierul de ieşire se realizează tot în timp liniar pentru că implică scrierea extremităților a cel mult $N / 3$ muchii.

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

1.4.14. Decodificare

Inițial vom determina numărul de poziții în care permutarea identică şi permutarea care trebuie determinată coincid.

În continuare vom încerca să determinăm pozițiile în permutare ale tuturor numerelor cuprinse între 1 şi N. Pentru fiecare număr i, dacă poziția sa nu a fost determinată la un pas anterior, va trebui să determinăm poziția acestuia în permutarea căutată. Pentru aceasta, în permutarea identică, vom interschimba, pe rând, numărul i cu fiecare dintre celelalte elemente. Astfel, vom obține permutări de forma (1, 2, 3, ..., i - 1, j, i + 1, ..., j - 1, i, j + 1, ..., N). Numărul pozițiilor, în care această permutare şi permutarea care trebuie considerată coincid, poate creşte dacă şi numai dacă numărul i ajunge pe poziția sa sau numărul care trebuie să se afle pe poziția i ajunge în această poziție. Aşadar, pentru doar două dintre aceste permutări numărul pozițiilor considerate poate să crească. Există trei cazuri:

- Numărul pozițiilor care coincid nu creşte în nici un caz. În această situație putem trage concluzia că numărul i se află deja pe poziția sa, deci va fi al i-lea element în permutarea care trebuie determinată.
- Numărul pozițiilor care coincid creşte o singură dată. Fie j numărul cu care a fost interschimbat i în permutarea care a dus la creşterea numărului de poziții care coincid. Este uşor de observat că, în permutarea care trebuie determinată, numărul i se va afla pe poziția j, iar numărul j se va afla pe poziția i.
- Numărul pozițiilor care coincid creşte de două ori. Fie j şi k pozițiile pe care a ajuns numărul i în cele două permutări. Va trebui să determinăm pe care dintre aceste două poziții se află i în permutarea care trebuie determinată. Ştim cu siguranță că unul dintre numerele j şi k se află pe poziția i în permutarea care trebuie determinată. Va trebui să studiem două cazuri:
  - i trece pe poziția j, j pe poziția k şi k pe poziția i;
  - i trece pe poziția k, j pe poziția i şi k pe poziția j.

În unul dintre aceste două cazuri numărul pozițiilor care coincid va fi cu cel puțin 2 mai mare decât numărul pozițiilor care coincid în permutarea identică. Aşadar, vom verifica una dintre permutări; dacă numărul pozițiilor care coincid creşte cu cel puțin 2, atunci vom şti care este poziția numărului i în permutarea care trebuie determinată şi care este numărul care se află pe poziția i în această permutare.

După determinarea pozițiilor tuturor numerelor, vom afişa permutarea determinată.

Analiza complexității

Pentru început, trebuie precizat faptul că apelul funcției `Check` implică traversarea unui vector cu N elemente care conține permutarea care trebuie determinată. Aşadar ordinul de complexitate al unui astfel de apel este $O(N)$.

Preluarea dimensiunii şirului se realizează în timp constant, deci are ordinul de complexitate $O(1)$.

Identificarea poziției unui număr implică realizarea a cel mult $N - 1$ interschimbări şi, eventual, a unei permutări care va diferi de permutarea identică în trei poziții. Pentru fiecare dintre aceste permutări se va realiza un apel al funcției `Check`, deci operația de determinare a poziției unui număr în permutarea căutată are ordinul de complexitate $O(N) \cdot O(N) = O(N^2)$. În total, vom efectua cel mult N determinări, deci ordinul de complexitate al operației de determinare a permutării căutate este $O(N) \cdot O(N^2) = O(N^3)$.

Scrierea datelor în fişierul de ieşire implică traversarea permutării determinate, deci ordinul de complexitate al operației este $O(N)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(N^3) + O(N) = O(N^3)$.

1.4.15. Seti

Pentru a determina numărul de apariții ale cuvintelor din dicționar în mesajul dat, vom ordona alfabetic toate şirurile (cuvintele) de 16 caractere consecutive din mesaj. Apoi, pentru fiecare cuvânt din dicționar vom determina, prin metoda căutării binare, prima şi ultima apariție a cuvântului în şirul sortat. Evident, dacă un cuvânt din dicționar conține k litere, atunci, în momentul în care determinăm aceste două poziții, vom lua în considerare doar primele k litere ale şirurilor.

O primă observație ar fi că, datorită faptului că nu avem suficientă memorie la dispoziție, şirul sortat trebuie să conțină indici şi nu cuvinte propriu-zise. Astfel, al i-lea element al şirului sortat va conține poziția la care începe al i-lea cuvânt în şirul sortat.

Datorită limitei de timp foarte stricte, algoritmii clasici de sortare (*quicksort*, *heapsort*) nu sunt suficient de performanți. Pentru această problemă este recomandabilă folosirea algoritmului *radixsort*.

O altă dificultate care apare este datorată tot insuficienței memoriei. Şirul sortat de indici ar trebui să conțină numere reprezentate pe patru octeți (teoretic ar fi suficienți trei) deoarece textul poate conține peste 100000 de cuvinte. Pentru a evita această problemă, vom împărți textul în patru blocuri şi vom aplica algoritmul descris pentru fiecare dintre ele. Fiecare bloc va conține cel mult $2^{15} = 32768$ litere. Datorită faptului că numărul de litere este cel mult $2047 \cdot 64 = 131008 < 2^{17} = 131072$, sunt suficiente patru blocuri. Vom avea şi patru şiruri sortate, dar fiecare element al acestor şiruri va putea fi reprezentat pe doi octeți.

Datorită împărțirii textelor în blocuri apar probleme pentru cuvintele care fac parte din două blocuri. Pentru a le evita, vom face în aşa fel încât ultimele 15 litere dintr-un bloc să fie adăugate ca primele 15 litere ale blocului următor. Dimensiunea totală a textului va creşte cu până la 45 de caractere, dar va ajunge doar la cel mult $131053 < 2^{17}$, deci nu apar blocuri suplimentare. Se observă că la începutul celui de-al doilea bloc sunt copiate 15 litere din primul, deci ultimele 15 litere ale acestui al doilea grup trebuie să fie mutate în al treilea. Alte 15 litere ajung să fie ultimele în al doilea bloc, deci trebuie copiate în al treilea. Aşadar, la începutul celui de-al treilea bloc vor fi ultimele 30 de litere ale celui de-al doilea bloc. Folosind acelaşi raționament, ajungem la concluzia că la începutul celui de-al patrulea bloc vor fi ultimele 45 de litere ale celui de-al treilea bloc.

Aşadar, pentru rezolvarea problemei, mai întâi vom împărți textul inițial în grupuri de 32768 de litere. Apoi vom realiza copierea grupurilor de 15, 30 şi 45 de litere dintr-un bloc în altul şi vom construi cele patru şiruri sortate, folosind algoritmul *radixsort*. În continuare, pentru fiecare cuvânt din dicționar, vom determina prima şi ultima apariție a sa în fiecare dintre şirurile sortate (dacă nu există nici o apariție vom spune, prin convenție, că poziția ultimei apariții este mai mică decât poziția primei apariții). Numărul de cuvinte dintre cele două poziții va fi adăugat la numărul de apariții în text ale cuvântului din dicționar. Pe măsură ce sunt determinate aparițiile cuvintelor din dicționar, se vor scrie în fişierul de ieşire numerele corespunzătoare.

Analiza complexității

Citirea textului care trebuie analizat se realizează linie cu linie, deci ordinul de complexitate al operației este $O(N)$. Deoarece dicționarul conține M cuvinte, ordinul de complexitate al operației de citire a acestor cuvinte este $O(M)$.

Construirea blocurilor se realizează pe parcursul citirii, ordinul de complexitate al operației fiind $O(N)$. Aceeaşi complexitate ($O(N)$) o are şi operația de transformare a blocurilor, astfel încât ultimele 15 litere dintr-un grup să fie identice cu primele 15 din următorul.

Algoritmul *radixsort* este liniar, deci ordinul de complexitate al operației de sortare este $O(N)$.

Pentru fiecare cuvânt din dicționar vom efectua cel mult opt căutări binare, ordinul de complexitate al unei căutări fiind $O(\log N)$. Pentru cele 8 căutări vom avea ordinul de complexitate $8 \cdot O(\log N) = O(\log N)$. Deoarece pentru fiecare dintre cele M cuvinte din dicționar se vor realiza căutări binare, ordinul de complexitate al operației de determinare a numărului de apariții pentru toate cuvintele din dicționar este $O(M) \cdot O(\log N) = O(M \cdot \log N)$.

Generarea datelor de ieşire implică scrierea a M numere, deci ordinul de complexitate al operației este $O(M)$.

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

1.4.16. Suma divizorilor

Numărul A poate fi scris ca produs de factori primi sub forma:

$$A = d_1^{p_1} \cdot d_2^{p_2} \cdot \ldots \cdot d_n^{p_n},$$

unde d_1, ..., d_n sunt divizorii primi ai lui A, iar p_1, ..., p_n sunt puterile la care apar aceştia în descompunerea lui A în factori primi.

Ca urmare, numărul A^B poate fi scris sub forma:

$$A^B = d_1^{p_1 \cdot B} \cdot d_2^{p_2 \cdot B} \cdot \ldots \cdot d_n^{p_n \cdot B}.$$

Aşadar, divizorii numărului A^B au forma $\prod_{i=1}^{n} d_i^{q_i}$, unde q_i variază între 0 şi $p_i \cdot B$.

Suma acestor divizori poate fi scrisă sub forma:

$$\sum_{\substack{q_i=0,\ldots,p_i \cdot B \\ i=1,\ldots,n}} \prod_{i=1}^{n} d_i^{q_i}.$$

Această sumă de produse poate fi scrisă sub forma unui produs de sume astfel:

$$\prod_{i=1}^{n} \sum_{j=0}^{p_i \cdot B} d_i^j.$$

Aşadar, suma divizorilor unui număr de forma A^B este dată de un produs de sume. Aceste sume au forma $1 + q + q^2 + q^3 + \ldots + q^m$, deci sunt sume ale unei progresii geometrice de raţie q. Este cunoscut faptul că valoarea unei astfel de sume, pentru $q > 1$ este $\frac{q^{m+1}-1}{q-1}$, iar pentru $q = 1$ este $m + 1$.

Datorită faptului că se cere determinarea restului împărţirii la 9901 a sumei divizorilor, toate operaţiile efectuate vor fi *modulo* 9901.

Vor trebui folosite relaţiile:

```
(a + b) mod n = ((a mod n) + (b mod n)) mod n şi
(a · b) mod n = ((a mod n) · (b mod n)) mod n.
```

De asemenea, datorită faptului că numărul 9901 este prim, pentru orice număr nenul a va exista un număr a^{-1} (numit *invers*) astfel încât ($a \cdot a^{-1} \bmod 9901 = 1$). Aşadar, pentru a efectua o împărţire cu un anumit număr vom realiza, de fapt, o înmulţire cu inversul acestuia.

Pentru a rezolva problema vom începe cu descompunerea numărului A în factori primi şi identificarea puterilor la care apar aceştia în descompunere. Vom înmulţi aceste puteri cu B şi apoi vom calcula sumele progresiilor corespunzătoare.

Pentru calculul sumei unei progresii avem nevoie de efectuarea unei ridicări la putere (evident, *modulo* 9901). O modalitate rapidă de calcul se bazează pe următoarea formulă recursivă:

$$\begin{cases} 1 & \text{pentru } n = 0 \\ a^{[n/2]} \cdot a^{[n/2]} & \text{pentru } n \neq 0 \text{ par} \\ a^{[n/2]} \cdot a^{[n/2]} \cdot a & \text{pentru } n \neq 0 \text{ impar} \end{cases}$$

O modalitate de determinare a inversului unui număr a (*modulo* 9901) este considerarea numerelor de forma $9901 \cdot k + 1$ şi alegerea primului număr b de această formă al cărui rest la împărţirea cu a este 1. Inversul va fi dat de valoarea b / a. Datorită faptului că 9901 este prim va exista întotdeauna un astfel de număr şi se poate demonstra că valoarea k corespunzătoare este mai mică decât 9901.

O altă observaţie necesară este faptul că scăderea valorii 1 (necesară pentru calcularea sumei progresiilor) este echivalentă cu adunarea valorii 9900 (*modulo* 9901).

Pe parcursul determinării sumei progresiilor, acestea vor fi înmulţite, *modulo* 9901. În final, rezultatul va fi scris în fişierul de ieşire.

Analiza complexităţii

Datele de intrare constau doar în două numere, deci operaţia de citire a acestora se realizează în timp constant.

Pentru determinarea divizorilor primi şi a puterilor la care apar aceştia în descompunerea numărului A vom lua în considerare numărul 2 şi numerele impare cuprinse între 3 şi cel mult $\sqrt{A}$, aşadar ordinul de complexitate al acestei operaţii este $O(\sqrt{A})$.

Numărul sumelor calculate este dat de numărul divizorilor primi ai numărului A. Folosind aproximarea lui *Stirling*, se poate deduce că există aproximativ $\ln A$ numere prime cuprinse între 2 şi A. Aşadar, vom calcula $O(\log A)$ sume. Calculul unei sume implică o ridicare la putere. Folosind algoritmul recursiv prezentat anterior, o ridicare la puterea n are ordinul de complexitate $O(\log n)$. Puterea la care poate apărea un factor prim în descompunerea numărului A este cel mult $\log_2 A$; în descompunerea lui A^B puterea poate fi cel mult $B \cdot \log_2 A$. Ca urmare, o ridicare la putere va avea ordinul de complexitate $O(\log (B \cdot \log A))$. Celelalte operaţii necesare pentru calculul sumelor se realizează în timp constant (considerăm că determinarea inversului are ordinul de complexitate $O(1)$ deoarece operaţia necesită cel mult 9901 paşi; există şi posibilitatea construirii şi utilizării unui tablou de constante a, unde a_i să conţină inversul numărului i modulo 9901), deci ordinul de complexitate al determinării sumei tuturor divizorilor este $O(\log A) \cdot O(\log (B \cdot \log A)) = O(\log A \cdot \log (B \cdot \log A))$.

În fişierul de ieşire trebuie scris un singur număr, deci ordinul de complexitate al operaţiei de scriere a datelor de ieşire este $O(1)$.

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

1.4.17. Sistem

Problema poate fi enunțată în termeni ai teoriei grafurilor astfel: *Să se determine numărul grafurilor neorientate distincte în care fiecare nod are gradul 2.*

Este uşor de determinat numărul grafurilor conexe cu proprietatea cerută; practic, orice permutare (p_1, p_2, ..., p_N) a mulțimii poate duce la obținerea unei soluții. Dacă unim printr-o muchie vârfurile p_i şi p_{i+1} (i = 1, …, N – 1) şi vârfurile p_1 şi p_N,, obținem un graf (hamiltonian) în care toate vârfurile au gradul 2. Luând în considerare toate cele N! permutări obținem N! grafuri care respectă proprietatea cerută. Totuşi, nu toate aceste grafuri sunt distincte; dacă alegem o anumită permutare, atunci toate celelalte permutări circulare ale acesteia duc la obținerea aceluiaşi graf. De exemplu, permutările (1, 2, ..., N) şi (N, 1, ..., N – 1) corespund aceluiaşi graf. Există N permutări circulare ale unei permutări date, deci numărul total al grafurilor distincte se reduce de N ori, devenind (N – 1)!. În plus, dacă parcurgem în ordine inversă o anumită permutare, obținem acelaşi graf, deoarece grafurile sunt neorientate. De exemplu, permutările (1, 2, ..., N) şi (1, N, ..., 2) duc la obținerea aceluiaşi graf. Fiecare permutare poate fi parcursă în sens invers într-un singur mod, deci numărul total al grafurilor distincte se reduce de 2 ori. În concluzie, numărul grafurilor conexe cu proprietatea cerută este (N – 1)! / 2. Această valoare este valabilă pentru grafuri care conțin cel puțin trei noduri; nu există grafuri cu proprietatea cerută care să conțină unul sau două noduri.

Urmează să determinăm numărul total al grafurilor (conexe şi neconexe) care au proprietatea cerută. Vom nota cu Nr_k numărul grafurilor care respectă proprietatea cerută şi conțin k noduri. Vom avea $Nr_1 = Nr_2 = 0$ deoarece nu există nici un astfel de graf care să conțină unul sau două noduri. Prin convenție vom considera $Nr_0 = 1$ (se poate considera că graful vid respectă proprietatea cerută, deoarece am putea spune că "toate cele 0 noduri au gradul 2", deşi această afirmație este, practic, lipsită de sens; totuşi această convenție se va dovedi utilă pentru simplificarea calculelor).

Vom considera, pe rând, că nodul 1 face parte din componente conexe cu 3, 4, ..., N noduri. Vom determina numărul de grafuri în care acest nod face parte din componente conexe cu k elemente (k este cuprins între 3 şi N) şi apoi vom aduna aceste valori.

Dacă nodul 1 face parte dintr-o componentă conexă cu k elemente, atunci există C_{N-1}^{k-1} posibilități de a alege celelalte k – 1 noduri care fac parte din aceeaşi componentă conexă. Folosind rezultatul demonstrat anterior, deducem că numărul componentelor conexe cu k elemente care respectă proprietatea cerută este (k – 1)! / 2. Indiferent care sunt cele k noduri care formează componenta conexă, celelalte $N – k$ noduri pot forma Nr_{N-k} grafuri cu proprietatea cerută. În concluzie, dacă nodul 1 face parte dintr-o componentă conexă cu k elemente, avem $Nr_{N-k} \cdot C_{N-1}^{k-1} \cdot \frac{(k-1)!}{2}$ grafuri distincte cu proprietatea cerută.

Se observă utilitatea convenției $Nr_0 = 1$ deoarece, pentru componenta conexă formată din toate cele N noduri ale grafului, obținem:

$Nr_{N-N} \cdot C_{N-1}^{N-1} \cdot \frac{(N-1)!}{2} = 1 \cdot 1 \cdot \frac{(N-1)!}{2} = \frac{(N-1)!}{2}$,adică exact numărul grafurilor conexe cu proprietatea cerută. Cazul $k = N$ ar fi putut fi tratat ca fiind unul particular, dar folosirea convenției duce la simplificarea formulelor de calcul.

Numărul total al grafurilor este obținut prin însumarea acestor valori pentru $k = 3$, …, N, deci vom avea $Nr_N = \sum_{k=3}^{N} Nr_{N-k} \cdot C_{N-1}^{k-1} \cdot \frac{(k-1)!}{2}$ grafuri cu proprietatea cerută.

Chiar şi pentru valori mici ale lui N, acest număr nu va putea fi reprezentat folosind tipuri standard de date, deci vor trebui simulate operații aritmetice cu numere mari. Folosind un artificiu de calcul, nu va trebui să implementăm decât adunarea a două numere mari şi înmulțirea dintre un număr mare şi un număr mai mic decât N.

Vom avea nevoie de adunarea numerelor mari pentru calcularea sumei care duce la obținerea valorilor Nr_k. Valorile însumate sunt obținute prin înmulțirea unui număr mare (Nr_{N-k} cu mai multe numere mai mici decât 100). Aceste numere sunt obținute folosind următorul artificiu:

$$C_{N-1}^{k-1} \cdot \frac{(k-1)!}{2} = \frac{(N-1)!}{(k-1)! \cdot (N-k)!} \cdot \frac{(k-1)!}{2} = \frac{(N-1)!}{2 \cdot (N-k)!} =$$
$$= \frac{(N-k+1) \cdot (N-k+2) \cdot \ldots \cdot (N-1)}{2}.$$

Aşadar, şirul numerelor cu care se vor realiza înmulțiri este $N - k + 1$, $N - k + 2$, $N - 1$. Deoarece k este cel puțin 3, acest şir are cel puțin două elemente; elementele şirului sunt numere consecutive, deci unul dintre primele două este par. Acest număr par va fi împărțit cu 2 şi se va obține un şir de elemente care vor fi înmulțite, pe rând, cu un număr mare.

Pentru determinarea valorii Nr_N este necesară calcularea tuturor valorilor Nr_k ($k < N$): acestea vor fi păstrate într-un şir de numere mari, iar în final valoarea Nr_N va fi scrisă în fişierul de ieşire.

Analiza complexității

Fiecare operație aritmetică se efectuează în timp constant, deoarece numărul cifrelor unui număr (dacă se foloseşte baza 10000) este mai mic decât 100.

Citirea datelor de intrare este realizată în timp constant pentru că se citeşte un singur număr.

Pentru calcularea unei valori Nr_K avem nevoie de însumarea a $k - 3$ termeni. Valorile Nr_i necesare sunt cunoscute de la paşii anteriori; pentru determinarea termenilor care trebuie însumați avem nevoie de $i - 1$ înmulțiri. Aşadar, în total vom avea $(k - 4) + (k - 5) + ... + 1 = (k - 4) \cdot (k - 3)/2$ înmulțiri. În concluzie, ordinul de complexitate al operației de determinare a numărului Nr_k este $O(k^2)$. Vom determina N astfel de numere, ordinul de complexitate al întregii operații fiind $O(N^3)$, deoarece avem $k = O(N)$.

Datele de ieşire constau într-un singur număr mare, deci operația de scriere a acestora se realizează în timp constant.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(1) + O(N^3) + O(1) = O(N^3)$.

1.4.18. Comitat

O posibilitate de rezolvare a acestei probleme este folosirea metodei programării dinamice. Vom încerca să determinăm un tablou tridimensional A, unde A_{ijk} va indica lungimea minimă a unui traseu care porneşte din *Mordor* (turnul 0), ultimele două turnuri de pe acest traseu sunt j şi k şi traseul conține în total i turnuri.

Se observă că lungimile traseelor care conțin i turnuri pot fi determinate cu ajutorul lungimilor traseelor care conține $i - 1$ turnuri. Astfel, vom lua în considerare toate traseele pe care se află $i - 1$ turnuri, şi ultimul turn al traseului este j şi îl vom alege pe cel mai scurt dintre acestea. La lungimile acestor trasee vom adăuga distanța dintre turnurile j şi k. Datorită faptului că un călăreț se poate deplasa doar paralel cu axele de coordonate, distanța dintre două turnuri i şi j va fi dată de $|x_i - x_j| + |y_i - y_j|$. Fie p, penultimul turn de pe traseul care se termină în turnul j; evident, unghiul determinat de turnurile p, i şi j nu trebuie să anuleze convexitatea traseului. Pentru a putea reconstitui traseul vom păstra, pentru fiecare triplet (i, j, k) numărul de ordine p al turnului care se află pe traseu înaintea turnurilor j şi k.

În final, vom obține toate elementele tabloului tridimensional A. Evident, traseul trebuie să fie un poligon închis, deci ultimul turn trebuie să fie 0 (*Mordor*). Aşadar, vom studia valorile A_{ij0}; vom determina cea mai mare valoare i pentru care există o valoare j, astfel încât valoarea A_{ij0} este cel mult egală cu lungimea maximă a traseului.

După identificarea valorilor i şi j, vom putea determina turnurile de pe traseu cu ajutorul predecesorilor păstrați pentru fiecare triplet (i, j, k).

Pentru a mări viteza de execuție a programului vom determina, pentru fiecare triplet (i, j, k), dacă cele trei turnuri corespunzătoare se pot afla pe poziții consecutive în traseu (se respectă condiția de convexitate) şi vom folosi aceste informații pe parcurs în loc să verificăm convexitatea de fiecare dată.

Inițial putem determina toate traseele pe care se află două turnuri (*Mordor* şi oricare dintre celelalte). În continuare vom aplica algoritmul pentru a determina lungimile traseului pe care se află 3 turnuri, 4 turnuri etc.

Analiza complexității

Datele de intrare constau în numărul turnurilor, coordonatele acestora şi lungimea maximă a traseului, deci ordinul de complexitate al operației de citire este $O(n)$.

Va trebui să verificăm pentru fiecare triplet (i, j, k) dacă se respectă condiția de convexitate, operație care are ordinul de complexitate $O(n^3)$, deoarece există $(n + 1)^3$ astfel de triplete (i, j şi k variază între 0 şi n).

Pentru a determina o valoare A_{ijk}, vor trebui luați în considerare toți predecesorii posibili, deci ordinul de complexitate al acestei operații este $O(n)$. Deoarece tabloul este tridimensional sunt realizate $O(n^3)$ astfel de operații, deci ordinul de complexitate al operației de determinare a tabloului A (şi a tabloului care conține predecesorii) este $O(n) \cdot O(n^3) = O(n^4)$.

Identificarea valorii A_{ij0} care indică lungimea traseului care conține cele mai multe turnuri se realizează în timp pătratic deoarece, în cel mai defavorabil caz, sunt luate în considerare toate perechile (i, j).

Refacerea traseului (şi scrierea turnurilor de pe traseu în fişierul de ieşire) pe baza tabloului predecesorilor se realizează în timp liniar, deoarece vor exista cel mult n turnuri pe acest traseu.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(n) + O(n^3) + O(n^4) + O(n^2) + O(n) = O(n^4)$.