

**PROIECT ALGORITMI FUNDAMENTALI**

METODA GREEDY

Studenți:

Angheluș Crina-Cosmina

Comnoiu Annelore

Ivan Rareș Cosmin

Macovei Răzvan Ionuț

Mocanu Maria-Alexandra

Morariu Adrian-Ștefan

Morariu Alexandru Mihai

Oltean Bianca

Oltean Miruna

CUPRINS

[**INTRODUCERE** 3](#_Toc60850942)

[**STRUCTURA UNUI ALGORITM GREEDY** 3](#_Toc60850943)

[**AVANTAJE:** 3](#_Toc60850944)

[**DEZAVANTAJE:** 4](#_Toc60850945)

[**STRUCTURA GENERALĂ A PROBLEMELOR CARE SE REZOLVĂ CU METODA GREEDY** 4](#_Toc60850946)

[**PROBLEME REZOLVATE** 5](#_Toc60850947)

[**BIBLIOGRAFIE** 72](#_Toc60850948)

# **INTRODUCERE**

Metoda Greedy (lacom) este o tehnică folosită în informatică, în general, în rezolvarea problemelor de optimizare (maximizare, minimizare), pentru a determina cel mai bun candidat dintr-o anumită categorie.

# **STRUCTURA UNUI ALGORITM GREEDY**

* Se împarte problema în subprobleme independente.
* Pentru fiecare subproblemă se alege un optim local.
* Algoritmul face întotdeauna alegerea care arată cel mai bine în acel moment.
* În final, mulțimea de optime locale ar trebui să ducă la un optim global.

Nu se ajunge întotdeauna la optimul global!

# **AVANTAJE:**

* Algoritmi simpli și intuitivi.
* Oferă cea mai bună soluție la nivel local.
* Complexitatea în timp este, în general, polinomială.
* Se poate folosi în multe domenii: determinarea celor mai scurte drumuri în grafuri (Dijkstra), determinarea arborelui minimal de acoperire (Prim, Kruskal), codificare arborilor Huffmann, planificarea activităților, problema spectacolelor și problema fracționară a rucsacului.

# **DEZAVANTAJE:**

* Succesiunea de optime locale nu duce întotdeauna la un optim global.
* Nu avem întotdeauna soluție.
* Demonstrația corectitudinii soluției găsite poate fi greoaie.

# **STRUCTURA GENERALĂ A PROBLEMELOR CARE SE REZOLVĂ CU METODA GREEDY**

* Se dă o mulţime A
* Se cere o submulţime S⊆A care trebuie:

-să îndeplinească anumite condiţii interne (să fie acceptabilă)

-să fie optimală (să realizeze un maxim sau un minim)

# **PROBLEME REZOLVATE**

**PROBLEMA 1:**

* *ENUNŢUL PROBLEMEI:*

Prietenul tău a decis să meargă mai departe în viață și să-și uite ultima iubită. El a decis să scape de toate scrisorile de dragoste pe care le primise de la prietena sa, arzându-le, dar poate arde scrisorile numai atunci când este singur acasă. El are un total de N scrisori de dragoste pe care le primise de la prietena sa și este singur acasă timp de S ore. Fiecare scrisoare de dragoste, în funcție de lungimea și hârtia din care este alcătuită, durează b minute pentru a arde. Pentru a uita de ultima lui iubită, amicul tău are nevoie să ardă cât mai multe scrisori, ajută-l și informează-l care este numărul maxim de scrisori pe care le poate arde în S ore, dacă are nevoie de timp extra sau dacă scrisorile ard în mai puțin de S ore. De asemenea, el vrea să știe ce scrisori au fost arse.

* *DATE DE INTRARE:*

Prima linie a fișierului conține două numere întregi separate prin spațiu N și S

(1≤ **N** ≤ **10^5**, **1** ≤ **S** ≤ **10^4)** , unde N este numărul de scrisori pe care prietenul tău le are în total și S este timpul, în ore pentru care ar trebui să ardă focul. Urmează N linii care conțin fiecare două numere întregi care reprezintă timpul de ardere și tipul hârtiei, produsul lor fiind un număr întreg b (**1** ≤ **b** ≤ **10^6**), care semnifică timpul de care are nevoie fiecare scrisoare pentru a arde.

* *DATE DE IEŞIRE:*

Dacă scrisorile de dragoste ard pentru S ore exacte, atunci se afișează numărul de scrisori de dragoste care sunt arse, dacă scrisorile ard mai puțin de S ore, se afișează numărul total de minute pentru care scrisorile ard, ceea ce se întâmplă în cazul în care el rămâne fără scrisori. Cu toate acestea, dacă scrisorile ard mai mult de S ore, atunci se afișează numărul maxim de scrisori arse și timpul minim în minute pentru care focul arde după S ore. În plus, să se afișeze scrisorile care au fost arse.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <fstream>

#include <algorithm>

struct Scrisoare {

int timp\_ardere, index;

};

void Citire(int& N, int& S, Scrisoare CutieScrisori[500], int& total)

{

std::ifstream f("Scrisori.in");

f >> N >> S;

std::cout << "Prietenul tau are " << N << " scrisori pe care doreste sa le arda, avand " << S << " ore la dispozitie." << std::endl << std::endl;

for (int i = 0; i < N; i++)

{

int lungime\_hartie, tip\_hartie;

f >> lungime\_hartie >> tip\_hartie;

CutieScrisori[i].timp\_ardere = lungime\_hartie \* tip\_hartie;

CutieScrisori[i].index = i+1;

total += CutieScrisori[i].timp\_ardere;

}

f.close();

}

void AfisareSolutie(int N, int S, int timp, int nrScrisoriArse, int total)

{

if (nrScrisoriArse == N)

{

if (timp == 0)

{

std::cout << "Toate cele " << N << " scrisori au fost arse in cele " << S / 60 << " ore." << std::endl << std::endl;

}

else

{

std::cout << "Toate cele " << N << " scrisori au fost arse in mai putin de " << S / 60 << " ore, ramanand " << timp << " minute." << std::endl << std::endl;

}

}

else

{

std::cout << "In cele " << S / 60 << " ore au fost arse " << nrScrisoriArse << " scrisori." << std::endl << std::endl;

std::cout << "Pentru a arde toate scrisorile prietenul tau are nevoie ca focul sa arda pentru inca " << total - S << " minute." << std::endl;

}

}

bool comparator(Scrisoare x, Scrisoare y)

{

return x.timp\_ardere < y.timp\_ardere;

}

void Rezolvare(int N, int S, Scrisoare CutieScrisori[500], int total)

{

std::sort(CutieScrisori, CutieScrisori + N,comparator);

S \*= 60;

int nrScrisoriArse = 0, timp = S;

std::cout << "Scrisorile arse sunt: ";

for (int i = 0; i < N && timp != 0; i++)

{

if (CutieScrisori[i].timp\_ardere <= timp)

{

nrScrisoriArse++;

timp -= CutieScrisori[i].timp\_ardere;

std::cout << CutieScrisori[i].index << " ";

}

}

std::cout <<" "<< std::endl;

AfisareSolutie(N, S, timp, nrScrisoriArse, total);

}

int main()

{

int N, S, total = 0;

Scrisoare CutieScrisori[500];

Citire(N, S, CutieScrisori, total);

Rezolvare(N, S, CutieScrisori, total);

}

**CORECTITUDINEA PROBLEMEI:**

Presupunem ca setul de scrisori este ordonat crescător după timpul de ardere necesar pentru fiecare scrisoare.

Proprietatea de alegere "greedy":

Fie O = (o1, o2, . . . , om) o soluție optimă și X = (, , . . . ,) soluția dată de algoritm. Există mai multe situații:

* Dacă k > m atunci O nu este soluția optimă. 2
* Dacă k = m atunci X este optimă.
* Dacă k < m, atunci putem înlocui în O pe cu (scrisoarea care arde cel mai repede) fără a altera restricția problemei și păstrând același număr (maxim) de spectacole selectate. Obținem soluția optimă O = (, , . . . , ).

Proprietatea de substructura optimă. Considerăm soluția optimă O' = (x1, o2… om) determinată anterior. Presupunem că (o2, o3... om) **nu** este soluție optimă a subproblemei selecției din {scrisoare\_2,scrisoare\_3,...,scrisoare\_n}. Rezultă că există O'' = (o''2... o''k'') o altă soluție cu k'' > m. Acest lucru ar duce la o soluție ( x1, o''2... o''k'') mai bună decât O' = (x1, o2… om). Contradicție.

**COMPLEXITATEA PROBLEMEI:**

*1. Subprogramul Citire() => Complexitatea acestui subprogram este dată de structura repetitivă “for” => Acest ciclu se referă la variabila ֦ i ” care pornește de la valoarea ֦ 0 ” și se oprește la valoarea ֦ n-1 ”, la fiecare pas crescând cu o unitate, ceea ce înseamnă că acesta se execută de ֦ n ” ori. => Complexitatea este O(n).*

*n*

1. *Subprogramul AfișareSoluție() => În acest subprogram nu există structuri repetitive, motiv pentru care complexitatea este constantă = > Complexitatea este O(1).*
2. *Subprogramul Rezolvare() => Complexitatea acestui subprogram este dată de structura repetitivă “for” => Acest ciclu se referă la variabila ֦ i ” care pornește de la valoarea ֦ 0 ” și se oprește la valoarea ֦ n-1 ”, la fiecare pas crescând cu o unitate, ceea ce înseamnă că acesta se execută de ֦ n ” ori. => Complexitatea este O(n).*

*n*

*Pentru a determina complexitatea totală a implementării alegem complexitatea maximă dintre cele determinate anterior =>* ***Complexitatea este O(n).***

**PROBLEMA2:**

* *ENUNŢUL PROBLEMEI*:

Tatăl Soniei este un cunoscut om de afaceri, remarcat prin firma de IT pe care o administrează singur. Acesta a obținut foarte multe proiecte la începutul acestui an, motiv pentru care dorește să angajeze un număr de k persoane pentru a își mări echipa și pentru a se asigura că proiectele vor fi finalizate la timp. Anul 2020 a fost unul greu și pentru firma acestuia așa că el vrea să angajeze persoane care să muncească mult, iar suma totală pe care trebuie să o plătească acestora în fiecare lună să fie minimă.

Din lipsă de timp, bărbatul o roagă pe Sonia să îl ajute în alegerea pe care o are de făcut, menționând faptul că dorește eficiență maximă, suma totală minimă și că nu poate oferi unui angajat mai mult de o anumită sumă. Munca pe care o are de făcut Sonia nu este una ușoară, însă tatăl ei are încredere că va reuși. Ajut-o pe Sonia să ducă la bun sfârșit sarcina pe care i-a dat-o tatăl ei.

* *DATE DE INTARE:*

Prima linie conține două numere întregi n și k (1≤k≤n≤105) - numărul de persoane care participă la interviu și respectiv numărul de posturi. Fiecare dintre următoarele n linii conține două numere și care reprezintă munca oferită de persoana i și salariul pe care aceasta îl solicită.

* *DATE DE IEŞIRE:*

Să se afișeze persoanele care constituie soluția optimă având în vedere cerințele care trebuie îndeplinite și suma pe care angajatorul trebuie să o plătească lunar noilor angajați. În cazul în care nu au fost ocupate toate posturile să se semnaleze acest fapt printr-un mesaj corespunzător.

**REZOLVAREA PROBLEMEI:**

Interviu.cpp

#include <iostream>

#include <algorithm>

#include "PersoaneInterviu.h"

bool comparePersoaneInterviu(PersoaneInterviu pers1, PersoaneInterviu pers2)

{

return (pers1.eficienta() > pers2.eficienta());

}

int main()

{

PersoaneInterviu sir[900];

int nr\_persoane, nr\_angajari, salariul\_maxim;

std::cout << "Introduceti numarul de persoane care participa la interviu: ";

std::cin >> nr\_persoane;

std::cout << "Introduceti maximul de persoane acceptate: ";

std::cin >> nr\_angajari;

std::cout << "Introduceti salariul maxim pe care il poate oferi angajatorul unui angajat: ";

std::cin >> salariul\_maxim;

std::cout << "Introduceti persoanele: " << std::endl;

for (int index = 0; index < nr\_persoane; index++)

{

sir[index].citire();

}

std::sort(sir, sir + nr\_persoane, comparePersoaneInterviu);

int suma\_totala=0;

std::cout << "Solutia optima este: " << std::endl;

for (int index = 0; index < nr\_persoane && nr\_angajari != 0; index++)

{

if (sir[index].\_salariul\_dorit <= salariul\_maxim)

{

sir[index].afisare();

nr\_angajari--;

suma\_totala += sir[index].\_salariul\_dorit;

}

}

std::cout << std::endl;

if (nr\_angajari != 0)

{

std::cout << "Ups! Nu au fost ocupate toate posturile oferite de angajator deoarece salariul maxim pe care il poate oferi ";

std::cout<<"acesta nu corespunde cu salariile dorite de participantii la interviu.";

}

std::cout << std::endl;

std::cout <<"Angajatorul trebuie sa plateasca noilor angajati suma de "<< suma\_totala<<" euro.";

}

PersoaneInterviu.cpp

#include <iostream>

#include "PersoaneInterviu.h"

void PersoaneInterviu::citire()

{

std::cin >> \_munca >> \_salariul\_dorit;

}

void PersoaneInterviu::afisare()

{

std::cout << "(" << \_munca << "," << \_salariul\_dorit << ")" << std::endl;

}

double PersoaneInterviu::eficienta()

{

return (double)\_munca/ \_salariul\_dorit;

}

PersoaneInterviu.h

#pragma once

#include <iostream>

struct PersoaneInterviu {

int \_munca, \_salariul\_dorit;

void citire();

void afisare();

double eficienta();};

**PROBLEMA3:**

* *ENUNŢUL PROBLEMEI:*

Piratul Roby dorește să găsească locația unei comori. Acesta trebuie să parcurgă un traseu pe care se află diferite premii și restricții. Scopul său este să ajungă la poziția comorii colectând un numar cât mai mare de premii de pe drum.

Traseul parcurs de Roby este reprezentat printr-o matrice de dimensiune m\*n. Pe fiecare celulă a matricii se află câte un premiu. El pornește de la poziția (0,0) și comoara se află pe poziția (x,y). Acesta nu are cum să se întoarcă înapoi deci singurele mutări pe care le poate face sunt în “jos” și în “dreapta”.

Ajutați-l pe Roby să ajungă la comoară adunând un număr cât mai mare de premii de pe drum.

* *DATE DE INTRARE:*

**“m”-**nr de linii ale matricei

**“n”-**nr de coloane ale matricei

**“a”-**o matrice de m\*n celule care contine valoarea premiilor

**(x,y) -**coordonatele pe care se află comoara (linia și coloanal)

* *DATE DE IEŞIRE:*

Coordonatele fiecarei cellule pe care Roby o parcurge si nr de premii adunate pana la comoara.

**Exemplu:**

(0,0)

Pozitie de start: **(0,0)**

(1,0)

Pozitie comoara: **(x,y)=(3,3)**

(0,1)

0 1 2 3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 7 | 2 | 1 |
| 3 | 5 | 6 | 2 |
| 14 | 8 | 3 | 3 |
| 40 | 2 | 1 | 5 |

0

(0,2)

(1,1)

1

(1,2)

(2,1)

2

3

(3,1)

(2,2)

Valoarea totală a premiilor adunate: 32

(2,3)

(3,2)

(3,3)

**CORECTITUDINEA PROBLEMEI:**

Pornim de la poziția (0,0) și la fiecare pas local alegem mutarea în căsuța din dreapta sau din jos în care se află un premiu mai mare cu speranța ca vom obține cea mai bună soluție globală.

Pentru orice traseu am alege să parcurgem vom avea maxim „m-1” mutări în jos și maxim „n-1” mutări la dreapta, așadar numărul maxim de mutări este „m+n-2”.

**COMPLEXITATEA PROBLEMEI:**

-în caz favorabil (atunci când comoara se află la o mutare distanță) : O(1)

-în caz nefavorabil: O(m+n)

-în caz mediu: \*1 + \*2 + \*3 + … + \*(n+m) = O(n+m)

Cu toate acestea algoritmul Greedy nu ne oferă în cazul acesta cea mai bună soluție optima globală, comoara maximă pe exemplul prezentat fiind 66 în timp ce Greedy gasește maximul 32.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

void Read(int& lines, int& columns, int array[][100], int& finalline, int& finalcolumn)

{

std::cout << "Lines=";

std::cin >> lines;

std::cout << "Columns=";

std::cin >> columns;

std::cout << "Final line=";

std::cin >> finalline;

std::cout << "Final column=";

std::cin >> finalcolumn;

std::cout << "Array:" << "\n";

for (int i = 0; i < lines; i++)

for (int j = 0; j < columns; j++)

std::cin >> array[i][j];

}

void Greedy(int& lines, int& columns, int array[][100], int& finalline, int& finalcolumn)

{

int total = array[0][0], i = 0, j = 0;

std::cout << i << " " << j << "\n";

while (i < finalline || j < finalcolumn)

{

if (array[i + 1][j] > array[i][j + 1] && i != finalline || j == finalcolumn)

i++;

else if (array[i][j + 1] >= array[i + 1][j] && j != finalcolumn||i == finalline)

j++;

total += array[i][j];

std::cout << i << " " << j << "\n";

}

std::cout << "The total value of the prizes the pirate ghatered is " << total;

}

int main()

{

int lines, columns, array[100][100], finalline, finalcolumn;

Read(lines, columns, array, finalline, finalcolumn);

Greedy(lines, columns, array, finalline, finalcolumn);

}

**PROBLEMA4:**

* *ENUNŢUL PROBLEMEI:*

Lucy are un program foarte încărcat pentru ziua următoare în care multe activități se desfășoară în același timp. Ea hotărăște să își aleagă cât mai multe activități care se desfășoară în momente diferite.

Ajutați-o pe Lucy să își organizeze activitățiile astfel încât să poată face cât mai multe într-o zi.

* *DATE DE INTRARE:*

**“n”-**numarul total de intervale de timp

**-**un vector cu **“n”** activități la care știm **“s”** si **“f”** ( timpul de START și de FINISH )

* *RESTRICŢII:*

Si < fi () i

* *DATE DE IEŞIRE:*

O serie maximă de activități care nu se desfășoara în același timp.

**Exemplu:**

n=9

Activitățiile:

1

Vectorul după sortare în funcție de timpul de sfârșit

1 5

0 2

0 1

2 4

3 8

1 3

4 6

7 9

4 7

V[0]

V[1]

V[2]

V[3]

V[4]

V[5]

V[6]

V[7]

V[8]

0 1

0 2

1 3

2 4

1 5

4 6

4 7

3 8

7 9

Vect[0]

Vect[1]

3

2

Vect[2]

4

Vect[3]

5

Vect[4]

6

Vect[5]

7

Vect[6]

8

Vect[7]

9

Vect[8]

FINISH: f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8]

* Maximul număr de activități care nu se suprapun este 4. Aceste activități pot fi:

{1, 3, 6, 9} sau {1, 4, 6, 9} sau {2, 4, 6, 9}

* **STRATEGIA OPTIMĂ DE REZOLVARE:**
* **Primul pas**: Ordonăm activitățiile în ordine crescătoare în funcție de timpul de FINISH.
* **Al doilea pas**: Parcurgem vectorul și la fiecare pas verificăm daca timpul de start al activității curente este mai mare decât timpul de sfârșit al ultimei activități salvate:

vect[ i ].start >= vect[ i - 1 ].finish

Dacă da, atunci numărul de activități posibile crește și salvăm poziția i.

* **Ultimul pas:** afișăm numărul maxim posibil și activitățiile care nu se suprapun.
* **COMPLEXITATEA:**

În orice caz, numărul de pași este același deoarece vom ordona activitățiile ( O(n \* log n) ) și apoi vom parcurge vectorul ( O(n) ). Așadar complexitatea depinde de tipul de ordonare și poate fi:

-în caz favorabil: n\* log n + n = O(n\*log n)

-în caz defavorabil: n2 + n = O(n2)

* **CORECTITUDINEA:**

După ordonarea în ordine crescătoare a activităților în funcție de timpul de sfârșit vom avea:

Proprietatea de alegere "greedy":

Fie O = (, , . . . , ) o soluție optimă și X = (, , . . . ,) soluția dată de algoritm. Există mai multe situații:

* Dacă k > m atunci O nu este soluția optimă. 2
* Dacă k = m atunci X este optimă.
* Dacă k < m, atunci putem înlocui în O pe cu (activitatea care se termină cel mai repede) fără a altera restricția problemei și păstrând același număr (maxim) de activități selectate. Obținem soluția optimă O = (, , . . . , ).

Proprietatea de substructură optimă. Considerăm soluția optimă O= ( , . . . , ).). Presupunem că (, . . . , ) nu este soluție optimă pentru subproblema selecției subsetul considerat. Rezultă că există O = (, . . . ,) o alta soluție cu k > m. Acest lucru ar conduce la o soluție (, , . . . ,) mai bună decât O= (, , . . . , ).). Contradicție.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <vector>

#include <algorithm>

struct Activity

{

int start;

int finish;

};

bool comp(Activity nr1, Activity nr2)

{

return nr1.finish < nr2.finish;

}

void Read(std::vector<Activity>& array)

{

int number;

std::cout << "Number of activities=";

std::cin >> number;

std::cout << "Activities:" << "\n";

array.resize(number);

for (int i = 0; i < array.size(); i++)

{

std::cout << i + 1 << " ";

std::cin >> array[i].start >> array[i].finish;

}

}

void Number(std::vector<Activity>& array)

{

int count = 1;

std::vector<int> maxi;

maxi.resize(array.size());

maxi[0] = 1;

for (int i = 1; i < array.size(); i++)

{

if (array[i].start >= array[maxi[count - 1] - 1].finish)

{

maxi[count] = i + 1;

count++;

}

}

std::cout << "Maximum number of activities is " << count << ".\n";

std::cout << "The activities are:";

for (int i = 0; i < count; i++)

{

std::cout << "\n";

std::cout << array[maxi[i] - 1].start <<" "<< array[maxi[i] - 1].finish;

}

}

int main()

{

std::vector<Activity> array;

Read(array);

sort(array.begin(), array.end(), comp);

Number(array);

}

**PROBLEMA5*:***

* *ENUNŢUL PROBLEMEI:*

Martinica are o galeata plin cu sirop de zmeura. Ursuletul doreste sa umple cat mai multe sticlute cu sirop. Acesta stie cati litri de sirop are in galeata. Martinica stie ca are N sticle . Sticlele sale sunt de dimensiuni diferite. Ajutat-l pe Martinica sa umple un numar maxim de sticle.

* *DATE DE INTRARE:*

Prima linie contine doua numere intregi, N(nrSticle) si X(capacitateGaleata), numarul de sticle si capacitatea unei galeti.

A doua linie contine N numere intregi separate de spatiu, capacitati de sticle (capacitateSticla).

* *DATE DE IEŞIRE:*

Afisati numarul maxim de sticle pe care le puteti umple.

File.In.txt

8 10 -numarul de sticle, capacitatea galetii

1 5 1 3 1 4 8 2 -capacitatile sticlelor

File.Out.txt

Martinica are o galeata de 10 l plina cu sirop de zmeura.

Ursuletul vrea sa umple cu sirop de zmeura cat mai multe din cele 8 sticle.

Sticlele au capacitatile urmatoare : 1 5 1 3 1 4 8 2

Am reusit sa umplem 5 sticle.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

using namespace std;

void CitireInfo(int& nrSticle, int& capacitateGaleata, vector<int>& capacitateSticla)

{

ifstream Citire("File.In.txt");

Citire >> nrSticle >> capacitateGaleata;

for (int i = 0; i < nrSticle; i++)

{

int elemCurent;

Citire >> elemCurent;

capacitateSticla.push\_back(elemCurent);

}

Citire.close();

}

int Rezolvare\_Tehnica\_Greedy(int nrSticle, int capacitateGaleata, vector<int> capacitateSticla)

{

int nrSticleUmplute, nrCurentSticle;

nrSticleUmplute = 0;

nrCurentSticle = 0;

sort(capacitateSticla.begin(), capacitateSticla.end());

for (int i = 0; i < nrSticle; i++)

{

nrCurentSticle += capacitateSticla[i];

if (nrCurentSticle > capacitateGaleata)

break;

nrSticleUmplute++;

}

return nrSticleUmplute;

}

void Rezolvare\_Afisare( int nrSticle, int capacitateGaleata, vector<int> capacitateSticla)

{

ofstream Afisare("File.Out.txt");

Afisare << "Martinica are o galeata de " << capacitateGaleata << " l plina cu sirop de zmeura.\n";

Afisare << "Ursuletul vrea sa umple cu sirop de zmeura cat mai multe din cele " << nrSticle << " sticle. \n ";

Afisare << "Sticlele au capacitatile urmatoare : ";

for (int i = 0; i < nrSticle; i++)

Afisare << capacitateSticla[i] << " ";

Afisare <<"\n\n";

Afisare << "Am reusit sa umplem " << Rezolvare\_Tehnica\_Greedy(nrSticle, capacitateGaleata, capacitateSticla) << " sticle.\n\n ";

}

int main()

{

int nrSticle, capacitateGaleata;

vector<int> capacitateSticla;

CitireInfo(nrSticle, capacitateGaleata, capacitateSticla);

Rezolvare\_Afisare(nrSticle, capacitateGaleata, capacitateSticla);

return 0;

}

**COMPLEXITATEA:**

1.Subprogramul CitireInfo()

2\*+3 => Complexitatea este θ(n)

2.Subprogramul Rezolvare\_Tehnica\_Greedy()

n\*log(n)+\* => Complexitatea este O(n\*log(n)), unde n\*log(n) este complexitatea pentru sort() din “algorithm”

3.Subprogramul Rezolvare\_Afisare()

7++ (n\*log(n)+\*) => Complexitatea este O(n\*log(n))

4.Functia main()

2\*+3+7++ (n\*log(n)+\*) => Complexitatea este O(n\*log(n))

**PROBLEMA6:**

* *ENUNŢUL PROBLEMEI:*

Vulturul Mamit este extrem de impresionat de strategiile noii sale cunoștințe Arjit, dar este puțin sceptic cu privire la el. Se gândește că dacă Arjit lucrează pentru domnul Babu și încearcă să-l omoare! Sau, mintea lui joacă un joc cu el cu toate aceste teorii ale conspirației? Dacă ar fi știut.

Pierdut în gânduri, vulturul Mamit s-a dus la Pilobar să bea un suc fără să-și dea seama că drogurile erau deja amestecate în băutura lui de către domnul Babu și trupa sa de ticăloși. Vulturul Mamit ajunge să fie atât de îndârjit încât nici măcar nu poate recunoaște că Arjit a fost cel care a amestecat droguri în băutura sa.

Acum, Arjit, văzând toată această scenă în fața lui, râde de vulturul Mamit ca un nebun, dezvăluind că este domnul Babu și că a planificat asta tot timpul. Între timp, vulturul Mamit vede un numar N de Arjiti stând în fața lui. Are dificultăți în a-l recunoaște pe cel real printre imaginile false. Deci, el atribuie mental fiecărui Arjit un coeficient “Minciuna” - și într-un mod mai simplu, el urmează acest algoritm special:

* El vede o secvență de N Arjiti stând într-o linie.
* El atribuie fiecărui Arjit un coeficient “Minciuna”.
* El selectează un index i după bunul său plac.
* El compară i Arjit cu i + 1 Arjit.
* Apoi, își dă seama că Arjit a cărui valoare a coeficientului “Minciuna”este mai mică - NU este real.
* După care, el înlocuiește cei doi Arjit, i și i + 1, cu un singur Arjit, care ar fi cel cu valoarea mai mare a coeficientului “Minciuna”.
* Vulturul Mamit pierde energie egală cu maximul celor doi Arjiti comparati.
* După (n-1) astfel de încercări ale vulturului Mamit, va rămâne un singur Arjit.

Acum, trebuie să-l ajuți pe vulturul Mamit să afle calea optimă de ieșire, astfel încât să poată reduce numărul de Arjits pentru a-și economisi energia. Pe scurt, trebuie să ajungă la un punct în care să poată vedea adevăratul Arjit din fața lui.

* *DATE DE INTRARE:*

Prima linie ar conține un număr întregr N, denotând numărul de Arjiti văzuți de vulturul Mamit. După care urmează N numere, fiecare denotând valoarea coeficientului “Minciuna” pentru fiecare Arjit.

* *DATE DE IEŞIRE:*

Trebuie să tipăriți costul de energie pentru al găsi pe Arjit cel real.

File.In.txt

3 -numarul de Arjiti pe care ii vede

3 1 2 -coeficientul atribuit fiecarui Arjit

File.Out.txt

Costul de energie pentru a-l gasi pe Artjit in grupul: 3 1 2 este: 5.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <vector>

#include <fstream>

using namespace std;

void CitireVectorArjit(ifstream& Citire,vector<int>& sirArjit, int& numarDeArjit)

{

Citire >> numarDeArjit;

for (int i = 0; i < numarDeArjit; i++)

{

int temporar;

Citire >> temporar;

sirArjit.push\_back(temporar);

}

}

int CalculareCostGreedy(vector<int> sirArjit, int numarDeArjit)

{

int cost\_energie = 0;

for (int indexArjit = 0; indexArjit < numarDeArjit - 1; indexArjit++)

{

cost\_energie += max(sirArjit[indexArjit], sirArjit[indexArjit + 1]);

}

return cost\_energie;

}

int main()

{

ifstream Citire;

Citire.open("file.in.txt");

ofstream Afisare("FileOut.txt");

int numarDeArjit;

vector <int> sirArjit;

CitireVectorArjit(Citire,sirArjit, numarDeArjit);

Afisare<< "Costul de energie pentru a-l gasi pe Artjit in grupul: ";

for (int i = 0; i < numarDeArjit; i++)

Afisare << sirArjit[i]<<" ";

Afisare << "este: " << CalculareCostGreedy(sirArjit, numarDeArjit) << ".\n";

Citire.close();

Afisare.close();

return 0;

}

**COMPLEXITATEA:**

1.Subprogramul CitireVectorArjit() => 1+2\*=> Complexitatea este θ(n)

2.Subprogramul CalculareCostGreedy()

=> => Complexitatea este θ(n)

Functia main()

=> F(n)= 1+2\*+1++ = 4\*+2 => Complexitatea este θ(n)

**PROBLEMA7:**

* *ENUNŢUL PROBLEMEI:*

Andrei pregateste o petrecere pentru M prieteni (numerotați de la 1 la M). A cumpărat N cosuri de bomboane (numerotate de la 1 la N), astfle incat fiecare cos i conține ai bomboane.

Fiecare prieten ar trebui sa primeasca cel mult o bomboana. Fiecare prieten I doreste sa obtina o bomboana fie de la unul dintre primele cosuri Li, fie de la unul dintre ultimele cosuri Ri (adica dintr-un cos b, astfle incat b≤ Li sau b>N- Ri).

Andrei doreste cat mai multi dintre prietenii lui sa obtina bomboane. Daca distribuie bomboanele in mod optim, care este numarul de prieteni care vor primi bomboane?

**Exemple:**

1. numarul de cosuri cu bomboane = 3

numarul de copii = 3

numar de bomboane in fiecare cos = {1, 1, 1}

Li, Ri = {(0,2), (1, 1), (0,2)}

1. numarul de cosuri cu bomboane = 3

numarul de copii = 3

numar de bomboane in fiecare cos = {1, 1, 1}

Li, Ri = {(1,1), (1, 1), (1,1)}

**Raspunsuri:**

1. 3 (Primul prieten primeste o bomboana din cosul 2, al doilea prieten primeste o bomboana din cosul 2, al treilea prieten primeste o bomboana din cosul 3)
2. 2 (Primul prieten primeste o bomboana din cosul 1, al doilea prieten primeste o bomboana din cosul 3, al treilea prieten nu poate primi bomboane)

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <algorithm>

#include<vector>

#define For(i,j,k) for(int i=(int)(j);i<=(int)(k);i++)

using namespace std;

void rezolvare()

{

int n, m, x;

pair<int, int>intervalCosuri[1000];

cout << "Introduceti numarul de cosuri cu bomboane: ";

cin >> n;

cout << "Introduceti numarul de copii: ";

cin >> m;

vector<int> numarBomboane(n), copiiRamasi(m);

For(i, 0, n - 1)

{

cin >> numarBomboane[i];

}

For(i, 0, m - 1)

{

cin >> intervalCosuri[i].first >> intervalCosuri[i].second;

}

For(i, 0, m - 1)

{

intervalCosuri[i].first = intervalCosuri[i].first - 1;

intervalCosuri[i].second = n - intervalCosuri[i].second;

}

sort(intervalCosuri, intervalCosuri + m);

int cosCurent = 0, raspuns = 0, numarCopiiRamasi = 0;

For(i, 0, m - 1)

{

for (; cosCurent <= intervalCosuri[i].first && !numarBomboane[cosCurent]; ++cosCurent);

if (cosCurent <= intervalCosuri[i].first)

{

--numarBomboane[cosCurent]; ++raspuns;

}

else

{

copiiRamasi[numarCopiiRamasi++] = intervalCosuri[i].second;

}

}

sort(copiiRamasi.begin(), copiiRamasi.end());

cosCurent = n - 1;

for (int i = numarCopiiRamasi - 1; i >= 0; i--)

{

for (; cosCurent >= copiiRamasi[i] && !numarBomboane[cosCurent]; --cosCurent);

if (cosCurent >= copiiRamasi[i])

{

--numarBomboane[cosCurent];

++raspuns;

}

}

cout << "Numarul de copii care primeste bomboane este: " << raspuns << endl;

}

int main()

{

rezolvare();

return 0;

}

**COMPLEXITATEA:**

O(m\*n) datorita faptului ca exita un loop (cu o lungime maxima n) intr-un alt loop (cu o lungime m).

**PROBLEMA8:**

* *ENUNŢUL PROBLEMEI:*

David lucreaza ca bucatar intr-un restaurant. In fiecare dimineata, el trebuie sa conduca de acasa la restaurant pe un drum drept cu lungimea X. Drumul are exact (doua benzi numerotate 1 si 2), dar exista N obstacole (numerotate de la 1 la N) pe el. Pentru fiecare i, al i-lea obstacol blocheaza banda Li la distanta Xi si nu blocheaza cealalta banda.

David nu poate trece printr-un obstacol. Pentru a evita obstacolul i, poate schimba banda la distanta xi-1, iar in aceasta pozitie nu trebuie sa existe un alt obstacol. Cu toate acestea, de fiecare data cand schimba banda, nu o poate schimba din nou pana cand conduce pentru cel putin o distanta D spre restaurant. David poate incepe pe orice banda isi doreste.

Uneori, este imposibil sa ajunga la restaurant. Care este distanta maxima pe care David o poate parcurge inainte de a ajunge la un obstacol? Daca ajunge la restaurant atunci raspunsul este X.

**Exemple:**

1. Distanta pana la restaurant = 10, distanta pentru a schimba banda = 20,

numar de obstacole = 2

distanta obstacolelor = {4, 7}, pozitia obstacolelor = {1, 2}

1. Distanta pana la restaurant = 15, distanta pentru a schimba banda = 20,

numar de obstacole = 4

distanta obstacolelor = {4, 6, 9, 13}, pozitia obstacolelor = {1, 2, 2, 1}

1. Distanta pana la restaurant = 10, distanta pentru a schimba banda = 1,

numar de obstacole = 5

distanta obstacolelor = {1, 3, 5, 7, 9}, pozitia obstacolelor = {1, 2, 1, 2, 1}

1. Distanta pana la restaurant = 10, distanta pentru a schimba banda = 2,

numar de obstacole = 2

distanta obstacolelor = {4, 5}, pozitia obstacolelor = {1, 2}

**Raspunsuri:**

1. 10 (Se incepe pe banda 2 si schimba banda inainte de al doilea obstacol pentru a ajunge la restaurant)
2. 13 (Se incepe pe banda 2, se schimba banda inainte de al doilea obstacol, dar nu poate sa evite ultimul obstacol)
3. 10 (Se incepe pe banda 2 si schimba banda inainte de fiecare obstacol pentru a ajunge la restaurant)
4. 5 (Se incepe pe banda 2, dar nu poate evita al doilea obstacol deoarece nu este suficient spatiu intre cele 2 obstacole)

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <vector>

using namespace std;

void rezolvare()

{

int numarObstacole, distantaRestaurant, distantaSchimbareBanda;

cout << "Distanta pana la restaurant: ";

cin >> distantaRestaurant;

cout << "Distanta pentru a putea schimba banda: ";

cin >> distantaSchimbareBanda;

cout << "Numarul de obstacole: ";

cin >> numarObstacole;

vector<int> distantaObstacol(numarObstacole);

for (int i = 0; i < numarObstacole; i++)

{

cin >> distantaObstacol[i];

}

vector<int> bandaObstacol(numarObstacole);

for (int i = 0; i < numarObstacole; i++)

{

cin >> bandaObstacol[i];

}

int distantaParcursaFinal = 0, bandaCurenta, distantaParcursaCurenta = 0, distantaParcursaMinima = 0;

if (bandaObstacol[0] == 1)

{

bandaCurenta = 2;

}

else

{

bandaCurenta = 1;

}

bool accidentat = false;

for (int i = 0; i < numarObstacole; i++)

{

if (bandaCurenta == bandaObstacol[i])

{

if (distantaObstacol[i] <= distantaParcursaMinima)

{

distantaParcursaFinal = distantaObstacol[i];

accidentat = true;

break;

}

distantaParcursaMinima = distantaObstacol[i];

distantaParcursaCurenta = distantaParcursaMinima;

distantaParcursaMinima += distantaSchimbareBanda;

if (bandaCurenta == 1)

{

bandaCurenta = 2;

}

else

{

bandaCurenta = 1;

}

}

else {

if (distantaParcursaMinima < distantaObstacol[i])

{

distantaParcursaMinima = distantaObstacol[i] + 1;

}

distantaParcursaCurenta = distantaObstacol[i];

}

}

if (!accidentat)

{

distantaParcursaFinal = distantaRestaurant;

}

cout << "Distanta parcura: " << distantaParcursaFinal << endl;

}

int main()

{

rezolvare();

return 0;

}

**COMPLEXITATEA:**

O(n) deoarece apare un loop de lungime numarObstacole.

**PROBLEMA9:**

* *ENUNŢUL PROBLEMEI:*

Iulia are o cofetărie in oraș și a primit o listă de n comenzi pe care trebuie să le realizeze pentru mai multe evenimente. Fiecare comandă necesită o unitate de timp pentru a putea fi făcută.

De asemenea, Iulia a observat ca are o cantitate limitată de ingrediente, astfel că nu vă putea onora toate comenzile.

Cunoscând numărul de comenzi, profitul pe care îl obține dacă livrează o anumită comandă, cât și termenul limită până la care trebuie livrate fiecare, ajutați-o pe Iulia să stabilească profitul maxim pe care îl poate obține.

* *DATE DE INTRARE:*

Pe prima linie a fișierului de intrare “comenzi.in” se găsește un număr natural n, reprezentând numărul de comenzi din listă.  
 Pe următoarele n linii din fișierul de intrare vor fi datele despre fiecare comanda, pe linia i găsindu-se 2 numere naturale p și t care reprezintă profitul, respectiv termenul limită de livrare pentru comanda i.

* *RESTRICȚII ȘI PRECIZĂRI:*
* 2 ≤ n ≤ 10000
* 2 ≤ p ≤ 1000
* 1 ≤ t ≤ 10000
* *DATE DE IEŞIRE:*

În fișierul “comenzi.out” se va afișa un singur număr natural, reprezentând profitul maxim pe care îl poate obține Iulia.

**Exemplu:**

comenzi.in comenzi.out

4 25

10 3

8 1

2 1

7 5

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <algorithm>

#include <fstream>

using namespace std;

ifstream f("comenzi.in");

ofstream g("comenzi.out");

struct comanda {

int profit;

int termenLimita;

};

bool compar(comanda comanda1, comanda comanda2)

{

return comanda1.profit > comanda2.profit;

}

int profit\_maxim(comanda vector[], long long n)

{

sort(vector, vector + n, compar);

bool interval[10001];

long long profitMaxim = 0;

for (int j = 0; j < n; j++)

interval[j] = false;

for (int i = 0; i < n; i++)

{

for (int j = vector[i].termenLimita -1; j >= 0; j--)

{

if (interval[j] == false)

{

interval[j] = true;

profitMaxim = profitMaxim + vector[i].profit;

break;

}

}

}

return profitMaxim;

}

int main()

{

long long n;

comanda vectorComenzi[10001];

f >> n;

for (int i = 0; i < n; i++)

{

f >> vectorComenzi[i].profit >> vectorComenzi[i].termenLimita;

}

g << profit\_maxim(vectorComenzi, n);

}

**COMPLEXITATEA:**

Complexitatea acestei probleme este O(n2) datorită celor două

FOR-uri imbricate din funcția *profit\_maxim*, unul care parcurge vectorul de comenzi și cel de-al doilea care parcurge intervalele de timp.

Funcțiile care influențează complexitatea:

int profit\_maxim(comanda vector[], long long n)

{

sort(vector, vector + n, compar); **=> O(n log n )**

for (int j = 0; j < n; j++)

interval[j] = false; **=> O(n)**

for (int i = 0; i < n; i++)

{

for (int j = vector[i].termenLimita -1; j >= 0; j--)

{ => **O(n2)**

...

}

}

}

int main()

{

for (int i = 0; i < n; i++)

f >> vectorComenzi[i].profit >> vectorComenzi[i].termenLimita; **=> O(n)**

}

Astfel comparând toate complexitățile găsite, o alegem pe cea mai mare, adică O(n2).

**CORECTITUDINEA:**

Presupunem lista de comenzi sortată descrescător în funcție de profit ( comanda\_1.profit > comanda\_2.profit > … > comanda\_n.profit).

1. Proprietatea de alegere Greedy

Fie O = (o1, o2... om) o soluție optimă și X = (x1, x2... xk) soluția dată de algoritm. ,

1. Dacă k > m: Soluția O nu este optimă
2. Dacă k = m: Soluția X este optimă
3. Dacă k < m: Atunci putem înlocui în O pe o1 cu x1 (comanda cu profitul cel mai mare) fără a altera restricția problemei (comenzile selectate respectă cerințele) și păstrând același număr (maxim) de comenzi selectate. Obținem soluția O' = (x1, o2… om).
4. Proprietatea de substructură optimă

Considerăm soluția optimă O' = (x1, o2… om) determinată anterior. Presupunem că (o2, o3... om) **nu** este soluție optimă a subproblemei selecției din {comanda\_2, comanda\_3... comanda\_n}. Rezultă că există O'' = (o''2... o''k'') o altă soluție cu k'' > m. Acest lucru ar duce la o soluție ( x1, o''2... o''k'') mai bună decât O' = (x1, o2… om). Contradicție.

**PROBLEMA10:**

* *ENUNŢUL PROBLEMEI:*

Alice și Bob s-au hotărât să se joace “Piatra-foarfece-hârtie”. Jocul are mai multe runde, fiecare fiind independentă de celelalte. În fiecare rundă, fiecare dintre cei doi jucători arată unul dintre următoarele obiecte, in același timp: piatră, foarfece sau hârtie. Dacă ambii jucători arată același obiect, se consideră remiză, altfel se aplică una dintre următoarele reguli:

* Dacă un jucător arată piatră, iar celălalt foarfece, jucătorul care a arătat piatră a câștigat runda;
* Dacă un jucător arată foarfece, iar celălalt hârtie, jucătorul care a arătat foarfece a câștigat runda;
* Dacă un jucător arată hârtie, iar celălalt piatră, jucătorul care a arătat hârtie a câștigat runda;

Alice și Bob au stabilit să joace un număr n de runde. Alice a hotărât să arate piatră de a1 ori, foarfece de a2 ori și hârtie de a3 ori. Bob a hotărât să arate piatră de b1 ori, foarfece de b2 ori și hârtie de b3 ori. Nu se cunoaște ordinea în care ei vor arăta aceste obiecte. De asemenea, este știut faptul că a1+ a2+a3=n și b1+ b2+b3=n.

Cunoscând numărul total de runde jucate și de câte ori un jucător va alege un anumit obiect, determinați numărul minim și numărul maxim de runde pe care le poate câștiga Alice.

* *DATE DE INTRARE:*

Pe prima linie a fișierului “jocuri.in” se află numărul total de runde jucate.

Pe a doua linie se află trei numere naturale: a1 a2 și a3, reprezentând de câte ori Alice va arăta piatră, foarfece și respectiv hârtie.

Pe a treia linie se află trei numere naturale: b1 b2 și b3, reprezentând de câte ori Bob va arăta piatră, foarfece și respectiv hârtie.

* *RESTRICȚII ȘI PRECIZĂRI:*
* 1 ≤ n ≤ 109
* 0 ≤ a ≤ n
* 0 ≤ b ≤ n
* *DATE DE IEŞIRE:*

Fișierul “jocuri.out” va conține pe aceeași linie, separate printr-un spațiu, două numere reprezentând numărul minim de runde pe care le poate câștiga Alice și respectiv numărul maxim.

**Exemplu:**

jocuri.in jocuri.out

686 22 334

479 178 29

11 145 530

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include<fstream>

#include<algorithm>

using namespace std;

ifstream f("jocuri.in");

ofstream g("jocuri.out");

struct joc {

long long piatra;

long long foarfece;

long long hartie;

};

void citire\_date\_intrare(long long& n, joc Jucator1[], joc Jucator2[])

{

f >> n;

f >> Jucator1->piatra;

f >> Jucator1->foarfece;

f >> Jucator1->hartie;

f >> Jucator2->piatra;

f >> Jucator2->foarfece;

f >> Jucator2->hartie;

}

int nr\_minim\_victorii(joc Jucator1[], joc Jucator2[])

{

long long nrVictoriiMinime = 0;

long long nrVictoriiImplicit = 0;

long long victoriePiatra = 0, victorieFoarfeca = 0, victorieHartie = 0;

victoriePiatra = (Jucator1->piatra - Jucator2->piatra) - Jucator2->hartie;

victorieFoarfeca = (Jucator1->foarfece - Jucator2->foarfece) - Jucator2->piatra;

victorieHartie = (Jucator1->hartie - Jucator2->hartie) - Jucator2->foarfece;

nrVictoriiMinime = max({ victoriePiatra, victorieFoarfeca, victorieHartie, nrVictoriiImplicit });

return nrVictoriiMinime;

}

int nr\_maxim\_victorii(joc Jucator1[], joc Jucator2[])

{

long long nrVictoriiMaxime = 0;

long long minim1 = 0, minim2 = 0, minim3 = 0;

minim1 = min(Jucator1->piatra, Jucator2->foarfece);

minim2 = min(Jucator1->foarfece, Jucator2->hartie);

minim3 = min(Jucator1->hartie, Jucator2->piatra);

nrVictoriiMaxime = minim1 + minim2 + minim3;

return nrVictoriiMaxime;

}

int main()

{

long long n;

joc Jucator1[1001];

joc Jucator2[1001];

citire\_date\_intrare(n, Jucator1, Jucator2);

g << nr\_minim\_victorii(Jucator1, Jucator2) << " ";

g << nr\_maxim\_victorii(Jucator1, Jucator2);

}

**COMPLEXITATEA:**

Rezolvarea acestei probleme are la bază efectuarea unor calcule (adunări și scăderi), acestea având o complexitate O(1). În plus, se determină atât minimul cât și maximul dintre anumite valori folosind funcțiile *min* și *max* din STL. Acestea au o complexitate liniară O(n).

Astfel complexitatea finală a algoritmului va fi O(n).

**PROBLEMA11:**

* *ENUNŢUL PROBLEMEI:*

X este un puternic magician. El are n grămezi de bomboane, grămada cu numărul i conținând a\_i bomboane. X poate folosi o vrajă "copy-paste" după cum urmează:

1. El alege 2 grămezi (i,j) astfel încât 1 ≤ i,j ≤ n si i ≠ j.

2. Toate bombonele din grămada i sunt copiate in grămada j.

Formal, are loc operația : =.

X poate folosi această vrajă de câte ori vrea, dar din păcate, dacă o anumită grămadă conține strict mai mult de k bomboane, el iși pierde puterea magică.

Care este numărul maxim de câte ori X poate să folosească vraja fără să își piardă puterea magică?

* *DATE DE INTRARE:*

Prima linie conține un număr natural T (1 ≤ T ≤ 500), reprezentând numărul de teste care se vor realiza.

Fiecare test conține 2 linii:

- pe prima linie se află 2 numere naturale n si k

(2 ≤ n ≤ 1000, 2 ≤ k ≤ .);

- pe a doua linie sunt n numere naturale: , , ...,

(1 ≤ ≤ k).

Se știe că suma lui n din toate testele nu depășește 1000 iar suma lui k din toate testele nu depășește .

* *DATE DE IEŞIRE:*

Pentru fiecare test se afișează un rezultat, reprezentând de câte ori X poate să își folosească vraja fără a își pierde puterea magică.

* *STRATEGIA OPTIMĂ DE REZOLVARE:*

Pentru fiecare test prima dată salvăm într-un vector **vec** cei **n** termeni citiți. Simultan determinăm termenul cel mai mic din șir pe care îl salvăm în variabila **minim.** Pentru a obține numărul maxim de utilizări a vrăjii vom aduna (de câte ori este necesar) termenul **minim** la fiecare a[i] din vector (i=), atâta timp cât se îndeplinește condiția: a[i] **k**. La final se scad din numărul obținut toate utilizările vrăjii asupra termenului minim, deoarece acesta nu se poate aduna cu el însuși.

Exemplu:

Input:

T = 3

n = 2, k = 2

1, 1

n = 3, k = 5

1, 2, = 3

n = 3, k = 7

3, 2, = 2

Output:

1

5

4

Demonstrație pentru primul test:

**minim** = 1

După folosirea vrăjii pentru prima data avem ori = = 2 si = 1 ori și = = 2 și este imposibil să se mai folosească vraja din nou (în fiecare caz existând un termen la care dacă se adună **minim** a doua oară acesta va depăși numărul k).

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

ifstream f("CopyPaste.in");

int T;

f>> T;

while(T)

{

int n, k, minim = 10000, suma = 0;

f >> n >> k;

int vec[1000];

for(int i = 0; i < n; i++)

{

f >> vec[i];

if (vec[i] < minim) minim = vec[i];

}

for(int i = 0; i < n; i++)

{

suma = suma + (k - vec[i]) / minim;

}

suma = suma - (k - minim) / minim;

cout << suma << endl;

T--;

}

return 0;

}

**COMPLEXITATEA:**

**Caz favorabil:**

T=1

Citirea→ O(n)

Calculul sumei→ O(n)

→ Complexitatea: O(n)

**Caz mediu:**

T ≥ 2

Citirea→ O(n)

Calculul sumei→ O(n)

Tot algoritmul se repetă de T ori așadar complexitatea obținută este O(Tn).

**PROBLEMA12:**

* *ENUNŢUL PROBLEMEI:*

Astăzi este ziua lui Sage așa că ea trebuie să meargă să cumpere sfere de gheață. Toate cele n sfere sunt așezate într-o linie și sunt numerotate de la 1 la n (de la stânga la dreapta). Fiecare sferă are ca preț un număr natural (prețurile sunt distincte 2 câte 2).

O sferă de gheață este considerată ieftină dacă prețul ei este strict mai mic decât al celor 2 sfere vecine (cea mai aproape din stânga și cea mai aproape din dreapta). Prima și ultima sferă din linie nu sunt considerate ieftine. Sage va alege numai sferele considerate ieftine și le va cumpăra numai pe acelea.

Poți vizita magazinul înaintea lui Sage și să rearanjezi sferele de gheață așa cum dorești.

Află numărul maxim de sfere de gheață pe care Sage le poate cumpăra și arată cum ar trebui să fie sferele rearanjate.

* *DATE DE INTRARE:*

Prima linie conține un număr natural n (1 ≤ n ≤ ), reprezentând numărul total al sferelor de gheață din magazin.

A doua linie conține n numere naturale distincte ,, ..., (1 ≤≤ 109) reprezentând prețurile sferelor de gheață.

* *DATE DE IEŞIRE:*

Prima linie va conține numărul maxim de sfere pe care Sage le poate cumpăra.

Pe a doua linie vor fi afișate prețurile sferelor în ordinea optimă. Dacă sunt multiple răspunsuri corecte, poate fi afișat oricare dintre acestea.

* *STRATEGIA OPTIMĂ DE REZOLVARE:*

Prima dată citim cei n termeni și îi înserăm într-un vector pe care îl sortăm crescător utilizând funcția **sort**. Oricare ar fi n (par sau impar), pot exista maxim **(n-1)/2** sfere de gheață considerate ieftine. Pentru a determina o ordine optimă, în funcție de paritatea lui **i** (i=) se vor afișa termenii din vector astfel: dacă **i** este impar se prelucrează în ordine descrescătoare a doua jumătate a vectorului, adică jumătatea cu numere mai mari (**vector[n - i / 2])** iar dacă **i** este par se prelucrează în ordine crescătoare prima jumătate a vectorului și anume numerele mai mici (**vector[i / 2]).** Astfel se obține un nou șir în care termenii de pe pozițiile pare vor reprezenta sferele de gheață ieftine (dacă n este par termenul de pe poziția a[n] nu se va lua în considerare, acesta neavând vecin în partea dreaptă).

Exemplu:

Input:

n = 5

1, = 2, = 3, = 4, = 5

Un output corect:

2

5 1 4 2 3

Demonstrație:

După sortare: **vector[]={1,2,3,4,5}**

Numărul de sfere ieftine: **= 2**

O ordine optimă va fi: **5 1 4 2 3**

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <algorithm>

#include <fstream>

using namespace std;

int main()

{

ifstream f("fisier.in");

int n;

f >> n;

int vector[100005];

for (int i = 1; i <= n; i++)

{

f >> vector[i];

}

sort(vector + 1, vector + n + 1);

cout << (n - 1) / 2 << endl;

for (int i = 1; i <= n; i++)

{

if (i % 2 == 1)

cout << vector[n - i / 2] << " ";

else

cout << vector[i / 2] << " ";

}

return 0;

}

**COMPLEXITATEA:**

Datorită structurilor repetitive **for** dar și a funcției **sort** complexitatea este: O(n\*log n).

**PROBLEMA13:**

* *ENUNŢUL PROBLEMEI:*

Stefan a terminat recent un curs de facut prajituri si cu abilitatile proaspat insusite si-a deschis o cofetarie. Face prajituri foarte bune asa ca numarul de clienti depaseste cateodata recordul!

Lui Stefan i se da numarul de clienti din n zile consecutive. Numarul de clienti din ziua i este b[i]. O zi depaseste recordul daca satisface urmatoarele 2 conditii:

1-numarul de clienti din ziua respectiva este strict mai mare decat numarul de clienti din toate zilele precedene

2-numarul de clienti din acea zi este mai mare decat numarul de clienti din urmatoarea zi

Atentie! Si prima zi poate sa fie o zi record!

Esti prieten cu Stefan din copilarie, te rog ajuta-l sa gaseasca numarul zilelor record!

* *DATE DE INTRARE:*

Prima linie se afla un numar natural n (numarul de zile).

//A doua linie contine n numere naturale. Al i-lea numar natural este b[i]. (Eu voi completa vectorul cu functia "srand")

* *DATE DE IEŞIRE:*

Linia de output contine un y, unde y este numarul de zile record.

Constrangeri

0 ≤ b[i] ≤ 2 × 10^5

1 ≤ n ≤ 1000

Exemplu Input

8

1 2 0 7 2 0 2 0

6

4 8 15 16 23 42

9

3 1 4 1 5 9 2 6 5

6

9 9 9 9 9 9

Exemplu Output

2

1

3

0

**Explicatie**

In cazul din exemplul #1 a doua si a patra zi respecta conditiile, deci rezultatul e: 2

In cazul din exemplul #2 doar ultima zi respecta conditiile, deci rezultatul e: 1

In cazul din exemplul #3 prima, a treia si a sasea zi sunt zile record, deci rezultatul e: 3

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <vector>

#include <stdlib.h>

#include <time.h>

#include <climits>

void citireNumarClienti(int, std::vector<int>&);

void afisareNumarZileRecord(std::vector<int>&);

void main()

{

int numar\_zile;

std::cout << "Numarul de zile: ";

std::cin >> numar\_zile;

std::vector<int> numar\_de\_clienti;

citireNumarClienti(numar\_zile, numar\_de\_clienti);

afisareNumarZileRecord(numar\_de\_clienti);

}

void citireNumarClienti(int numar\_zile, std::vector<int>& numar\_de\_clienti\_in\_ziua)

{

srand(time(NULL));

for (int index\_zi = 0; index\_zi < numar\_zile; index\_zi++)

{

numar\_de\_clienti\_in\_ziua.push\_back(rand() % 200);

std::cout << numar\_de\_clienti\_in\_ziua[index\_zi] << " ";

}

std::cout << std::endl;

}

void afisareNumarZileRecord(std::vector<int>& sir\_numar\_clienti\_pe\_zile)

{

int maxim = INT\_MIN, raspuns = 0;

for (int index\_zi = 0; index\_zi < sir\_numar\_clienti\_pe\_zile.size() - 1; index\_zi++)

{

if (sir\_numar\_clienti\_pe\_zile[index\_zi] > maxim)

{

maxim = sir\_numar\_clienti\_pe\_zile[index\_zi];

if (sir\_numar\_clienti\_pe\_zile[index\_zi] > sir\_numar\_clienti\_pe\_zile[index\_zi + 1])

raspuns++;

}

}

if (sir\_numar\_clienti\_pe\_zile[sir\_numar\_clienti\_pe\_zile.size() - 1] > maxim)

raspuns++;

std::cout << raspuns << std::endl;

}

**PROBLEMA14:**

* *ENUNŢUL PROBLEMEI:*

Într-un acvariu sunt n piranha cu dimensiunile a1, a2, ..., an. Peștii sunt numerotați de la stânga la dreapta în ordinea poziției din acvariu.

Studențîi de la Universitatea Transilvania din Brașov vor să vadă dacă există un piranha dominant în acvariu. Un peste este numit dominant dacă acesta poate mânca toți ceilalți pești din acvariu. Ceilalți pești nu se vor opune când acesta ii va mânca.

Deoarece acvariul este destul de îngust și lung, un peste poate mânca doar peștii adiacenți acestuia într-o mișcare. Peștii pot face oricâte micari vor. Mai precis:

-peștele i poate mânca peștele i-1 dacă peștele i-1 există și a[i-1]<a[i]

-peștele i poate mânca peștele i+1 dacă peștele i+1 există și a[i+1]<a[i]

Când peștele i mănâncă un piranha, dimensiunea lui crește cu 1.

Treaba ta este să vezi dacă există un astfel de piranha dominant în acvariu.

Dacă găsești un astfel de peste nu trebuie să găsești alți pești dominanți.

De exemplu, dacă a=[3, 3, 5\*, 4, 5], atunci cel de-al treilea peste reprezenta peștele dominant.

EXPLICAȚIE

Peștele de pe poziția a 3-a îl poate mânca pe cel de pe poziția a 2-a și a devine [3, 6\*, 4, 5]

Peștele aflat acum pe poziția a 2-a îl mănâncă pe cel aflat pe poziția a 3-a și a devine [3, 7\*, 5]

Acum peștele îl mănâncă primul și a devine [8\*, 5]

Peștele îl mănâncă pe cel de-al doilea și a=[9]

* *DATE DE INTRARE:*

Prima linie se da numărul peștilor din acvariu. (Numărul trebuie să fie ≥ 2)

* *DATE DE IEŞIRE:*

Pentru fiecare test, se va afișa răspunsul: -1 dacă nu există piranha dominanți în acvariu sau indexul oricărui peste dominant dacă există. Dacă există mai multe răspunsuri corecte se poate alege oricare dintre acestea.

Exemplu

input

5

5 3 4 4 5

3

1 1 1

5

4 4 3 4 4

5

5 5 4 3 2

3

1 1 2

5

5 4 3 5 5

output

1

-1

2

3

3

1

Primul exemplu este deja explicat.

(In al doilea test nu exista piranha dominanti.)

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <vector>

#include <stdlib.h>

#include <time.h>

#include <climits>

void rezolvare();

void citireMarimiPesti(int, std::vector<int>&);

void aflarePiranhaDominant(std::vector<int>&);

int aflareMaximSir(std::vector<int>&);

bool verificareEgaliateTermeniSir(std::vector<int>&);

void main()

{

rezolvare();

}

void rezolvare()

{

int numar\_pesti;

std::vector<int>pesti;

std::cout << "Numar de pesti: ";

std::cin >> numar\_pesti;

if (numar\_pesti < 2)

{

std::cout << "Numarul pestilor trebuie sa fie >= 2";

std::cout << std::endl;

}

else

{

citireMarimiPesti(numar\_pesti, pesti);

aflarePiranhaDominant(pesti);

}

}

void citireMarimiPesti(int numar\_de\_pesti, std::vector<int>& pesti)

{

srand(time(NULL));

for (int index\_peste = 0; index\_peste < numar\_de\_pesti; index\_peste++)

{

pesti.push\_back(rand() % 20 + 1);

std::cout << pesti[index\_peste] << " ";

}

std::cout << std::endl;

}

void aflarePiranhaDominant(std::vector<int>& sir\_de\_pesti)

{

int index\_peste\_dominant = -1;

int marime\_maxima = aflareMaximSir(sir\_de\_pesti);

if (verificareEgaliateTermeniSir(sir\_de\_pesti) == true)

{

std::cout << -1 << " (NU EXISTA PESTE DOMINANT) " << std::endl;

}

else

{

if (sir\_de\_pesti[0] == marime\_maxima && sir\_de\_pesti[1] < marime\_maxima)

{

std::cout << "Pestele dominant se afla pe pozitia " << 1 << std::endl;

}

else

if (sir\_de\_pesti[sir\_de\_pesti.size() - 1] == marime\_maxima && sir\_de\_pesti[sir\_de\_pesti.size() - 2] < marime\_maxima)

{

std::cout << "Pestele dominant se afla pe pozitia " << sir\_de\_pesti.size() << std::endl;

}

else

{

for (int index\_peste = 1; index\_peste < sir\_de\_pesti.size() - 1; index\_peste++)

{

if ((sir\_de\_pesti[index\_peste] == marime\_maxima && sir\_de\_pesti[index\_peste - 1] < marime\_maxima) || (sir\_de\_pesti[index\_peste] == marime\_maxima && sir\_de\_pesti[index\_peste + 1] < marime\_maxima))

{

index\_peste\_dominant = index\_peste + 1;

break;

}

}

std::cout << "Pestele dominant se afla pe pozitia: " << index\_peste\_dominant << std::endl;

}

}

}

bool verificareEgaliateTermeniSir(std::vector<int>& sir\_dat)

{

int primul\_termen = sir\_dat[0];

for (int index = 1; index < sir\_dat.size(); index++)

if (primul\_termen != sir\_dat[index])

return false;

return true;

}

int aflareMaximSir(std::vector<int>& sirDat)

{

int maxim = INT\_MIN;

for (int index = 0; index < sirDat.size(); index++)

if (sirDat[index] > maxim)

maxim = sirDat[index];

return maxim;

}

**COMPLEXITATEA:**

**Cazul favorabil**:

Fie numărul de pești n.

Dacă peștele dominant se află pe prima sau pe ultima poziție se execută:

+ + 1 = **2n + 1** operații, deci complexitatea este **O(n)**

**Caz defavorabil(/mediu)**:

Fie numărul de pești n.

Dacă peștele dominant nu este nici pe prima nici pe ultima poziție se execută:

+ + + 1 = **3n** **+ 1** operații, deci complexitatea este **O(n)**

**CORECTITUDINEA:**

Fie “a” vectorul în care reținem mărimile peștilor (, , ... ). Verificăm dacă mărimile tuturor peștilor sunt diferite, în caz contrar, nu există niciun pește dominant, deci algoritmul returnează valoarea -1. Pentru rezolvarea problemei vom căuta mărimea maximă a unui pește (notată cu "mărime\_maxima"). După aceea, parcurgem vectorul de mărimi din nou. Când întâlnim un piranha cu mărimea egală cu cea maximă verificăm dacă în stânga sau în dreapta acestuia găsim un pește mai mic (a[i] > a[i-1] sau a[i] > a[i+1]); dacă da, peștele de pe poziția i poate reprezenta un piranha dominant, dacă nu, trecem la poziția următoare. Corectitudinea problemei constă în faptul că oricare pește de mărime maximă poate reprezenta un pește dominant dacă poate mânca minim un alt pește, astfel crescând cu 1 (orice termen maxim din șir, mărit cu 1 va fi cu siguranță mai mare decât restul termenilor).

**PROBLEMA15:**

* *ENUNŢUL PROBLEMEI:*

Alex conduce un cinematograf. Datorită numărului tot mai mare de platforme pentru vizionarea de filme online, afacerea sa nu funcționează bine. Ca prieten, Alex l-a rugat pe Mihai să-l ajute să-și maximizeze profiturile. Deoarece Mihai este o persoană ocupată, are nevoie de ajutorul tău pentru a-și susține prietenul, Alex. Cinematograful lui Alex are patru spectacole: La ora 12, ora 15, ora 18 respectiv 21 PM. Are patru filme pe care și-ar dori să le redea - să le numim A, B, C și D. Fiecare dintre aceste filme trebuie redate exact o dată și toate cele patru trebuie redate la diferite ore. Pentru fiecare ora, prețul unui bilet trebuie să fie unul dintre următoarele: 25 lei, 50 lei, 75 lei respectiv 100 lei. Prețurile biletelor pentru diferite ore de film trebuie să fie, de asemenea, diferite. Prin intermediul aplicației sale, Alex primește diverse cereri de la clienții săi. Fiecare solicitare are formularul „Vreau să vizionez acest film la aceasta ora” .Să presupunem că numărul de persoane care vin să vizioneze un film la o anumita ora este același cu numărul de solicitări pentru acel film la acea ora. Nu este necesar pentru a satisface cererile tuturor. Alex vrea doar să câștige suma maximă de bani. Nu există nicio restricție asupra capacității teatrului. Cu toate acestea, pentru fiecare film care nu este vizionat de nimeni, Alex ar suferi o pierdere de 100 lei (dedusă din profit). Se dau N cereri pe care Alex le-a primit în cursul unei zile. Găsiți suma maximă de bani pe care o poate câștiga în acea zi, alegând când să redea filme și cu ce prețuri.

* *DATE DE INTRARE:*

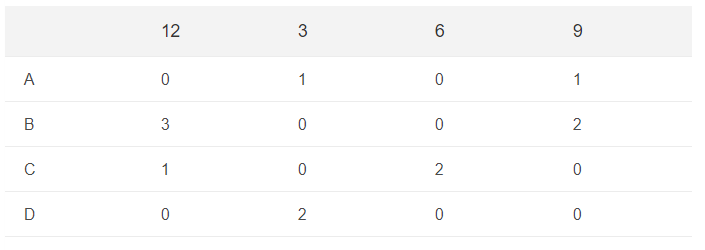
Prima linie a intrării conține un singur număr întreg **T** care indică numărul de cazuri de testare. Urmează descrierea cazurilor de test **T**. Prima linie a fiecărui caz de testare conține un singur număr întreg **n**. Urmează **n** linii. Fiecare dintre aceste linii conține un caracter **m**, urmat de un spațiu și un număr întreg t, descriind o cerere de a vedea filmul **m** la ora de prezentare **t**.

* *DATE DE IEŞIRE:*

Pentru fiecare caz de testare, tipăriți o singură linie care conține un număr întreg - profitul maxim pe care îl poate câștiga Alex (posibil negativ). În cele din urmă, tipăriți o linie care conține un număr întreg - profitul total pentru toate cazurile de testare, adică peste **T** zile.

**Exemplu:**

**Pentru datele de intrare 12(numarul de cereri de vizionari)** **A** 3 **B** 12 **C** 6 **A** 9 **B** 12 **C** 12 **D** 3 **B** 9 **D** 3 **B** 12 **B** 9 **C** 6 (Litera respectiva reprezinta filmul dorit iar cifra reprezentand ora) se va afisa 575. Deoarece 3⋅100+2⋅75+2⋅50+1⋅25=300+150+100+25=575



**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <string>

using namespace std;

void citireSiConstruireMatrice(int& n, int& t, int matrice[4][4])

{

char m;

cin >> n;

while (n)

{

cin >> m;

cin >> t;

if (m == 'A')

{

if (t == 3)

{

matrice[0][0]++;

}

if (t == 6)

{

matrice[0][1]++;

}

if (t == 9)

{

matrice[0][2]++;

}

if (t == 12)

{

matrice[0][3]++;

}

}

if (m == 'B')

{

if (t == 3)

{

matrice[1][0]++;

}

if (t == 6)

{

matrice[1][1]++;

}

if (t == 9)

{

matrice[1][2]++;

}

if (t == 12)

{

matrice[1][3]++;

}

}

if (m == 'C')

{

if (t == 3)

{

matrice[2][0]++;

}

if (t == 6)

{

matrice[2][1]++;

}

if (t == 9)

{

matrice[2][2]++;

}

if (t == 12)

{

matrice[2][3]++;

}

}

if (m == 'D')

{

if (t == 3)

{

matrice[3][0]++;

}

if (t == 6)

{

matrice[3][1]++;

}

if (t == 9)

{

matrice[3][2]++;

}

if (t == 12)

{

matrice[3][3]++;

}

}

n--;

}

}

void stergereLinieMatrice(int linieStearsa,int x,int matrice[4][4])

{

for (int i = linieStearsa; i < x; i++)

{

for (int j = 0; j < x; j++)

{

matrice[i][j] = matrice[i + 1][j];

}

}

}

void stergereColoanaMatrice(int coloanaStearsa, int x, int matrice[4][4])

{

for (int i = 0; i < x; i++)

{

for (int j = coloanaStearsa; j < x; j++)

{

matrice[i][j] = matrice[i][j+1];

}

}

}

int main()

{

int n, t, sumaFinala = 0;

int matrice[4][4] = { 0 };

citireSiConstruireMatrice(n, t, matrice);

int maximLocal;

int sumaTotala = 0;

int pretBilet = 100;

for (int x = 4; x > 0; x--)

{

maximLocal = INT\_MIN;

int pozI, pozJ;

for (int i = 0; i < x; i++)

{

for (int j = 0; j < x; j++)

{

if (matrice[i][j] > maximLocal)

{

maximLocal = matrice[i][j];

pozI = i;

pozJ = j;

}

}

}

sumaTotala = sumaTotala + (pretBilet \* maximLocal);

stergereLinieMatrice(pozI, x, matrice);

stergereColoanaMatrice(pozJ, x, matrice);

pretBilet = pretBilet - 25;

if (maximLocal == 0)

sumaTotala = sumaTotala - 100;

}

cout << sumaTotala << " ";

return 0;

}

**COMPLEXITATEA:**

Functia **citireSiConstruireMatrice** are complexitatea O(n).

Functia stergereLinieMatrice are complexitatea O(n^2)

Functia stergereColoanaMatrice are complexitatea O(n^2)

Functia main are complexitatea O(n^3)

Astfel complexitatea cea mai mare la care se poate ajunge este O(n^3).

**PROBLEMA16:**

* *ENUNŢUL PROBLEMEI:*

Un bloc este format din **m** etaje (numerotate de la 1 la **m**) și există un lift care este folosit pentru a se face deplasarea între diferite etaje. Ascensorul este conectat la un computer care înregistrează mișcarea într-o secvență **a**. Ori de câte ori ascensorul se mută la un etaj diferit, computerul adaugă noul număr de etaj la secvența **a**. În prezent, secvența **a** are **n** elemente.

Din păcate, computerul este infectat cu un virus care a înlocuit unele elemente ale lui **a** cu -1. Andrei, matematicianul blocului vrea acum să știe care ar putea fi numărul minim de ori în care liftul și-a schimbat direcția. Adică, de câte ori ascensorul, urcand, incepe sa coboare si invers

Ajutați-l pe Andrei răspunzând întrebării sale sau stabiliți că secvența **a** este invalidă.

* *DATE DE INTRARE:*

Prima linie conține un singur număr întreg **T** care indică numărul de cazuri de testare. Urmează descrierea cazurilor de test **T**.

Prima linie a fiecărui caz de testare conține două numere întregi **n** și **m**.

A doua linie conține **n** numere întregi separate prin spațiu **a1, a2, ..., an**

* *DATE DE IEŞIRE:*

Pentru fiecare caz de testare, tipăriți o singură linie care conține un număr întreg - numărul minim de ori când ascensorul și-a schimbat direcția sau −1 dacă secvența **a** dată este nevalidă.

**Exemplu:**

4 5

2 3 4 3

Se afiseaza 1 pentru ca liftul isi schimba sensul o singura data

4 5

2 -1 4 3

Se afiseaza 1 pentru ca liftul isi schimba sensul o singura data, inlocuind -1 cu 3.

Daca -1 s-ar fi inlocuit cu altceva, rezultatul nu ar mai fi minim.

5 4

-1 -1 3 -1 -1

Se afiseaza 1 pentru ca liftul isi schimba sensul o singura data, secventa poate fi inlocuita cu 1 2 3 4 3, astfel liftul isi schimba pozitia o singura data

5 5

-1 -1 3 -1 -1

Se afiseaza 0 pentru ca inlocuind secventa cu 1 2 3 4 5, liftul doar urca.

#include <iostream>

using namespace std;

bool suntToateNecunoscute(int n, int vector[])

{

for (int i = 0; i < n; i++)

{

if (vector[i] != -1)

return false;

}

return true;

}

int elementePozitive(int n, int vector[])

{

int k = 0;

for (int i = 0; i < n; i++)

{

if (vector[i] != -1)

k++;

}

return k;

}

void modificareVirus(int n, int m, int vector[])

{

int predecesor = 0, succesor = 0;

for (int i = 0; i < n; i++)

{

if (vector[i] != -1)

{

predecesor == i;

}

if (vector[i] < 0)

{

int j = i;

while (vector[j] == -1)

{

j++;

}

succesor = j;

}

if (vector[i] == -1)

{

if (predecesor == 0 && vector[predecesor] == -1)

{

for (int k = succesor; k < n; k++)

{

if (elementePozitive(n, vector) != 1)

{

if (vector[k] > vector[succesor] && vector[k] != -1)

{

vector[predecesor] = 1;

break;

}

if (vector[k] < vector[succesor] && vector[k] != -1)

{

vector[predecesor] = m;

break;

}

}

else

{

if (succesor - predecesor != 1 && m - vector[succesor] <= 1)

{

vector[predecesor] = 1;

}

else

{

vector[predecesor] = m;

}

break;

}

}

}

if (vector[predecesor] >= vector[succesor])

{

if (succesor == predecesor)

{

if (vector[succesor] > vector[succesor - 1])

{

for (int j = succesor; j < n; j++)

{

if (vector[j - 1] == m)

{

succesor = j - 1;

break;

}

vector[j] = vector[j - 1] + 1;

}

}

else

{

for (int j = succesor; j < n; j++)

{

if (vector[j - 1] == 1)

{

succesor = j - 1;

break;

}

vector[j] = vector[j - 1] - 1;

}

}

}

for (int j = predecesor; j < succesor; j++)

{

if (vector[j] == -1)

{

vector[j] = vector[j - 1] - 1;

}

}

predecesor = succesor;

i = predecesor;

}

else

{

if (succesor == predecesor)

{

if (vector[succesor] > vector[succesor - 1])

{

for (int j = succesor; j < n; j++)

{

if (vector[j - 1] == m)

{

succesor = j - 1;

break;

}

vector[j] = vector[j - 1] + 1;

}

}

else

{

for (int j = succesor; j < n; j++)

{

if (vector[j - 1] == 1)

{

succesor = j - 1;

break;

}

vector[j] = vector[j - 1] - 1;

}

}

}

for (int j = predecesor; j < succesor; j++)

{

if (vector[j] == -1)

{

vector[j] = vector[j - 1] + 1;

}

}

predecesor = succesor;

i = predecesor;

}

}

}

}

int main()

{

int n, m, deplasare[100];

cin >> n >> m;

for (int i = 0; i < n; i++)

{

cin >> deplasare[i];

}

if (suntToateNecunoscute(n, deplasare) == true)

{

cout << "-1";

}

int raspuns = 0;

modificareVirus(n, m, deplasare);

int diferenta = deplasare[0] - deplasare[1];

for (int i = 1; i < n - 1;)

{

if (deplasare[i] - deplasare[i + 1] != diferenta)

{

raspuns++;

diferenta = deplasare[i] - deplasare[i + 1];

}

i++;

}

cout<<raspuns;

return 0;

}

**PROBLEMA17:**

* *ENUNŢUL PROBLEMEI:*

Limak este un mic urs polar. El joacă un joc video și are nevoie de ajutorul tău. Există un rând cu N celule, fiecare liber sau ocupat de un soldat, notat cu „0” și respectiv „1”. Scopul jocului este de a muta toți soldații spre dreapta (fiecare soldat trebuie să ocupe cea mai din dreapta celulă liberă). Singura comandă posibilă este să alegi un soldat și să-i spui să se deplaseze spre dreapta cât mai mult posibil. Alegerea unui soldat durează o secundă, iar un soldat se mișcă cu viteza unei celule pe secundă. Soldatul se oprește imediat dacă se află în ultima celulă a rândului sau următoarea celulă este deja ocupată. Limak nu are voie să aleagă un soldat care nu se poate mișca deloc (soldatul ales trebuie să se mute cel puțin o celulă spre dreapta). Limak se bucură foarte mult de acest joc și vrea să se joace cât mai mult posibil. În special, el nu începe o comandă nouă până când soldatul ales anterior nu a finalizat comanda. Îi poți spune, câte secunde poate juca cel mai mult?

* *DATE DE INTRARE:*

Se citește un șir **S** care reprezintă rândul cu **N** celule. Fiecare caracter este fie „0”, fie „1”, reprezentând o celulă liberă, respectiv o celulă ocupată cu un soldat.

* *DATE DE IEŞIRE:*

Afisați numărul maxim de secunde posibil în care Limak va juca jocul.

* *EXEMPLU*

      Pentru S= 10100, se va afișa 8, pentru S=000000111, se va afișa 0, iar pentru S=001110100011010, se va afișa 48.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

#include <cstring>

using namespace std;

void numarareSecunde(char S[100], int N) {

int i, k, sec, nrGrupeZerouri = 0, totalNrSec;

totalNrSec = 0;

sec = 0;

for (i = 0; i < N; i++)

{

if (S[i] == '1' && i <= N - 1)

{

nrGrupeZerouri = 0;

for (k = i + 1; k < N; k++)

if (S[k] == '0')

{

sec++;

if (S[k + 1] == '1' && k + 1 <= N - 1)

nrGrupeZerouri++;

if (k == N - 1)

nrGrupeZerouri++;

}

totalNrSec += nrGrupeZerouri;

}

cout << "Numarul maxim de secunde pentru sirul S este: " << totalNrSec +sec << endl;

}

void citire() {

char S[100];

int N;

cout << "Dati sirul S: "; cin.getline(S, 100);

N = strlen(S);

numarareSecunde(S, N);

}

int main()

{

citire();

system("pause");

return 0;

}

**CORECTITUDINE ȘI COMPLEXITATE:**

Aplicăm în mod repetat următoarea strategie:

Se va alege soldatul cel mai din stânga care poate fi mutat și îl mutăm în dreapta sa cât mai mult posibil.

**Corectitudinea strategiei**

Implementarea directă a acestei strategii va avea o complexitate de O (n^2).

  Pentru fiecare mișcare se vor lua în considerare doi pași, unul se referă la alegerea soldatului, iar altul reprezintă mutarea soldatului în celulele din dreapta să.

      Numărul total de celule mutate de fiecare soldat va rămâne constant indiferent de modul în care soldații sunt selectati. Aceasta înseamnă că singura parte importantă pentru maximizarea timpului necesar jocului este de a alege soldații de cât mai multe ori posibil.

      Numărul maxim de ori în care un soldat poate fi ales să se miște, va fi egal cu numărul total de grupuri consecutive de zerouri care se află după soldat (adică se află în partea dreapta a soldatului).

      Astfel, această strategie conduce la o complexitate O (n^2).

**PROBLEMA18:**

* *ENUNŢUL PROBLEMEI:*

  Se dau 3 stive ce conțin numere reale pozitive. Se cere să se găsească suma maximă comună pentru cele 3 stive, aceasta obținându-se prin eliminarea elementelor din vârful fiecărei stive.  Stivele sunt reprezentate ca structuri unidimensionale de stocare, iar primul index al vectorului reprezintă elementul din vârful stivei.

* *Input:*

Stiva1[ ] = { 1, 1, 1, 2, 3}

Stiva2[ ] = {2, 3, 4}

Stiva3[ ] = {1, 4, 5, 2}

* *Output: 5*

***Soluție****:*  La fiecare pas se va elimina din stiva cu sumă maximă elementul din vârf, până când toate stivele se vor goli, sau până când toate cele 3 sume vor deveni egale.

**REZOLVAREA PROBLEMEI:**

#include <iostream>

using namespace std;

// Returnează suma maximă posibilă a tuturor celor 3 stive prin eliminarea //elementelor din vârf

int maxSum(int stiva1[], int stiva2[], int stiva3[], int dim1, int dim2, int dim3)

{

int sum1 = 0, sum2 = 0, sum3 = 0;

// Calculeaza suma inițială a stivei 1

for (int i = 0; i < dim1; i++)

sum1 += stiva1[i];

// Calculeaza suma inițială a stivei 2

for (int i = 0; i < dim2; i++)

sum2 += stiva2[i];

// Calculeaza suma inițială a stivei 3

for (int i = 0; i < dim3; i++)

sum3 += stiva3[i];

int varf1 = 0, varf2 = 0, varf3 = 0;

while (1) {

// Daca vreo stivă este goală

if (varf1 == dim1 || varf2 == dim2 || varf3 == dim3)

return 0;

// Daca sumele stivelor sunt egale

if (sum1 == sum2 && sum2 == sum3)

return sum1;

// Căutarea stivei cu sumă maximă si eliminarea elementului din vârf

if (sum1 >= sum2 && sum1 >= sum3)

sum1 -= stiva1[varf1++];

else if (sum2 >= sum1 && sum2 >= sum3)

sum2 -= stiva2[varf2++];

else if (sum3 >= sum2 && sum3 >= sum1)

sum3 -= stiva3[varf3++];

}

}

int main()

{

int stiva1[] = { 3, 2, 1, 1, 1 };

int stiva2[] = { 4, 3, 2 };

int stiva3[] = { 1, 1, 4, 1 };

int dim1 = sizeof(stiva1) / sizeof(stiva1[0]);

int dim2 = sizeof(stiva2) / sizeof(stiva2[0]);

int dim3 = sizeof(stiva3) / sizeof(stiva3[0]);

cout << maxSum(stiva1, stiva2, stiva3, dim1, dim2, dim3) << endl;

return 0;

}

**COMPLEXITATEA:**

Consider ca operație fundamentală atribuirea: sum1 = sum1 + stiva1[i]. Astfel, numărul de operații va fi egal cu dim1+dim2+dim3. La acestea se mai adaugă câteva comparații și atribuiri (nesemnificative).

  În concluzie, din punct de vedere al timpului de executare, complexitatea este O (n1+n2+n3) (unde n1,n2,n3 reprezintă dimesiunile celor 3 stive), algoritmul fiind liniar.

# **BIBLIOGRAFIE**

1. https://brilliant.org/wiki/greedy-algorithm/

2.https://www.tutorialspoint.com/data\_structures\_algorithms/greedy\_algorithms.htm

3.www.geeksforgeeks.org/greedy-algorithms/

4.https://www.codechef.com/problems/COEX01

5. https://codeforces.com/problemset/problem/1433/C

https://codeforces.com/problemset?tags=greedyV