**PROBLEME PREZENTARE GREEDY**

**PROBLEMA 1:**

* *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).

**PROBLEMA 2:**

* *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).***