

## TEHNICA DE PROGRAMARE "GREEDY"

**Tehnica Greedy se folosește pentru rezolvarea unor probleme de optim!**

### Exemplu:

Fie  $A$  o mulțime formată din  $n$  numere întregi. Să se determine o submulțime  $S \subseteq A$  cu proprietatea că suma elementelor sale este maximă.

$$A = \{5, 10, -7, 21, -3, 0, 18\} \Rightarrow S_1 = \{5, 10, 21, 18\} \text{ sau } S_2 = \{5, 10, 21, 0, 18\}$$

$$\sum_{x \in S_1} x = \sum_{x \in S_2} x$$

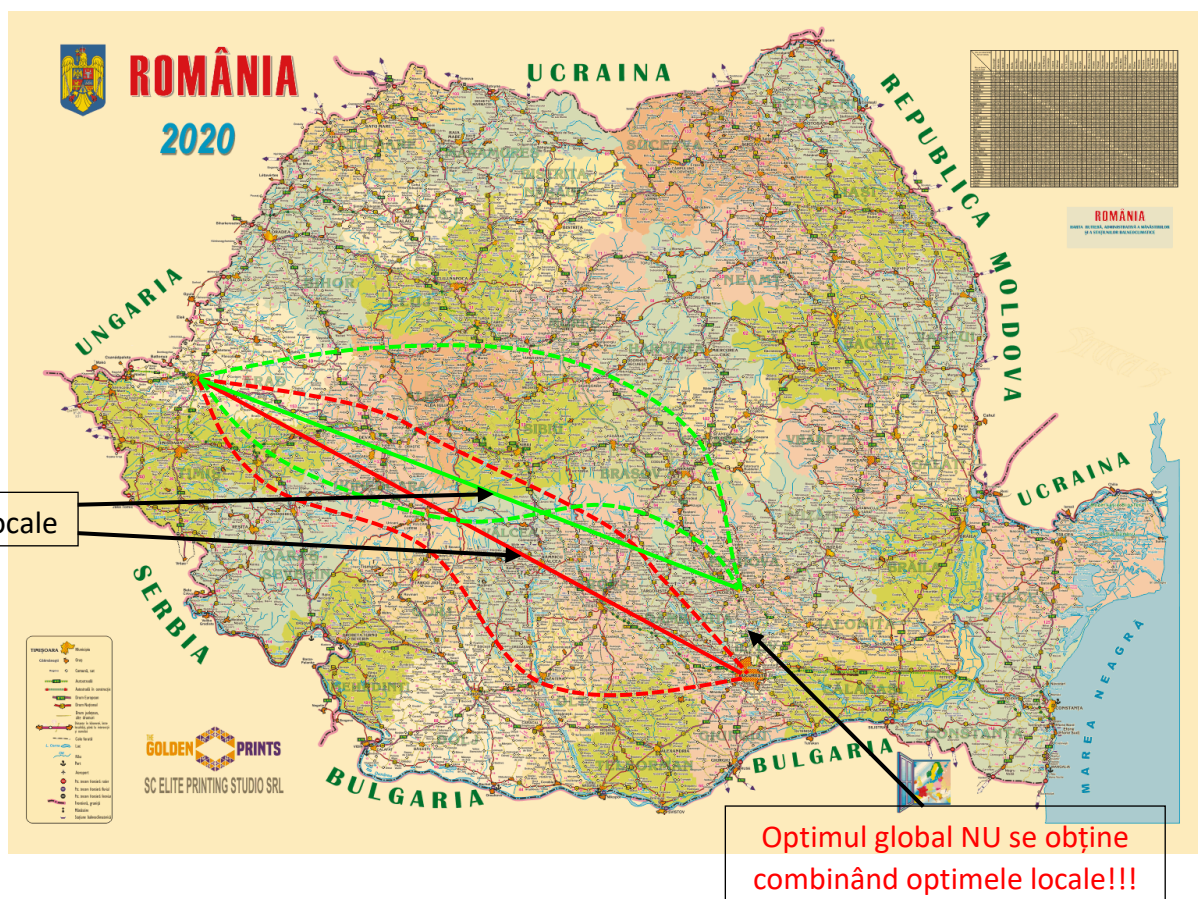
**Forță brută:** generăm toate submulțimile  $S$  ale mulțimii  $A$ , calculăm suma elementelor din  $S$  și reținem submulțimea  $S$  pentru care suma este maximă  $\Rightarrow$  complexitatea minimă este  $\mathcal{O}(2^n)$  !

**Tehnica Greedy:** parcurgem mulțimea  $A$  element cu element și introducem în submulțimea  $S$  elementele pozitive  $\Rightarrow$  complexitatea este  $\mathcal{O}(n)$ :

$$A = \{5, 10, -7, 21, -3, 0, 18\} \Rightarrow S = \{5, 10, 21, 18\}$$

$$A = \{-5, -10, -7, -21, -3, -18\} \Rightarrow S = \{-3\}$$

**Tehnica Greedy se utilizează în cazul în care combinând optime locale obținem un optim global și putem demonstra MATEMATIC acest lucru!**



## 1. Minimizarea timpului mediu de așteptare

La un ghișeu, stau la coadă  $n$  persoane  $p_1, p_2, \dots, p_n$  și pentru fiecare persoană  $p_i$  se cunoaște timpul său de servire  $t_i$ . Să se determine o modalitate de reazărare a celor  $n$  persoane la coadă, astfel încât timpul mediu de așteptare să fie minim.

De exemplu, să considerăm faptul că la ghișeu stau la coadă  $n = 6$  persoane, având timpii de servire  $t_1 = 7$ ,  $t_2 = 6$ ,  $t_3 = 3$ ,  $t_4 = 10$ ,  $t_5 = 6$  și  $t_6 = 3$ . Evident, pentru ca o persoană să fie servită, aceasta trebuie să aștepte ca toate persoanele aflate înaintea sa la coadă să fie servite, deci timpii de așteptare ai celor 6 persoane vor fi următorii:

Persoana	Timpul de servire ( $t_i$ )	Timp de așteptare ( $a_i$ )
$p_1$	7	7
$p_2$	6	$7 + 6 = 13$
$p_3$	3	$13 + 3 = 16$
$p_4$	10	$16 + 10 = 26$
$p_5$	6	$26 + 6 = 32$
$p_6$	3	$32 + 3 = 35$
Timpul mediu de așteptare (M):		$\frac{7 + 13 + 16 + 26 + 32 + 35}{6} = \frac{129}{6} = 21.5$

Deoarece timpul de servire al unei persoane influențează timpii de așteptare ai tuturor persoanelor aflate după ea la coadă, se poate intui foarte ușor faptul că minimizarea timpului mediu de așteptare se obține rearanjând persoanele la coadă în ordinea crescătoare a timpilor de servire:

Persoana	Timpul de servire ( $t_i$ )	Timp de așteptare ( $a_i$ )
$p_3$	3	3
$p_6$	3	$3 + 3 = 6$
$p_2$	6	$6 + 6 = 12$
$p_5$	6	$12 + 6 = 18$
$p_1$	7	$18 + 7 = 25$
$p_4$	10	$25 + 10 = 35$
<b>Timpul mediu de așteptare (<math>M</math>):</b>		$\frac{3 + 6 + 12 + 18 + 25 + 35}{6} = \frac{99}{6} = 16.5$

Practic, minimizarea timpului mediu de așteptare este echivalentă cu minimizarea timpului de așteptare al fiecărei persoane aflate la coadă, iar minimizarea timpului de așteptare al unei persoane se obține minimizând timpii de servire ai persoanelor aflate înaintea sa la coadă!

```
#include <fstream>
#include <iostream>
#include <algorithm>

using namespace std;

struct Persoana
{
    //ID = identificator al unei persoane = nr de ordine initial
    int ID;

    //ts = timpul de servire (individual)
    float ts;
};

//functie comparator pentru sortarea persoanelor
//in ordinea crescatoare a timpilor de servire
//trebuie sa intoarca "true" cand persoanele p1 si p2 sunt in
//ordinea dorita si "false" in caz contrar
bool cmpPersoane(Persoana p1, Persoana p2)
{
    return p1.ts < p2.ts;
}
```

```

int main()
{
    //n = numarul de persoane
    int n;
    //p = tablou alocat dinamic cu n elemente de tip Persoana
    Persoana *p;

    //citirea datelor de intrare din fisierul text "persoane.in"
    //pe prima linie se gaseste numarul n de persoane
    //pe a doua linie se gasesc timpii de servire,
    //despartiti intre ei prin cate un spatiu
    ifstream fin("persoane.in");

    fin >> n;

    //alocam dinamic tabloul p
    p = new Persoana[n];

    for(int i = 0; i < n; i++)
    {
        fin >> p[i].ts;
        p[i].ID = i+1;
    }

    fin.close();

    //afisam informatiile initiale despre persoane in fisierul text
    "persoane.out"
    ofstream fout("persoane.out");
    fout << "Timpii initiali:" << endl;
    fout << "Persoana\t\tTimp de servire\t\tTimp de asteptare" <<
endl;

    //ta = timpul de asteptare al persoanei curente
    float ta = 0;

    //tm = timpul mediu de asteptare al tuturor persoanelor
    float tm = 0;

    for(int i = 0; i < n; i++)
    {
        ta = ta + p[i].ts;
        tm = tm + ta;
        fout << "\t" << p[i].ID << "\t\t\t\t" << p[i].ts <<
"\t\t\t\t" << ta << endl;
    }
}

```

```

    fout << "Timpul mediu de asteptare initial: " << tm / n << endl
<< endl;

    //sortam tabloul p folosind comparatorul cmpPersoane
    //primul parametru = adresa de inceput a secventei pe care o vom
sorta
    //al doilea parametru = adresa elementului care urmeaza dupa
ultimul element din
    //secventa pe care o vom sorta
    //al treilea parametru = numele functiei comparator
    sort(p, p+n, cmpPersoane);

    fout << "Timpii optimi:" << endl;
    fout << "Persoana\t\tTimp de servire\t\tTimp de asteptare" <<
endl;

    //ta = timpul de asteptare al persoanei curente
    ta = 0;

    //tm = timpul mediu de asteptare al tuturor persoanelor
    tm = 0;

    for(int i = 0; i < n; i++)
    {
        ta = ta + p[i].ts;
        tm = tm + ta;
        fout << "\t" << p[i].ID << "\t\t\t\t" << p[i].ts <<
"\t\t\t\t" << ta << endl;
    }

    fout << "Timpul mediu de asteptare optim: " << tm / n << endl <<
endl;

    fout.close();

    //stergem din memorie tabloul p
    delete []p;

    return 0;
}

```

**persoane.in**

```

6
7 6 3 10 6 3

```

### persoane.out

Timpii initiali:

Persoana	Timp de servire	Timp de asteptare
1	7	7
2	6	13
3	3	16
4	10	26
5	6	32
6	3	35

Timpul mediu de asteptare initial: 21.5

Timpii optimi:

Persoana	Timp de servire	Timp de asteptare
3	3	3
6	3	6
2	6	12
5	6	18
1	7	25
4	10	35

Timpul mediu de asteptare optim: 16.5

### Forma generală a problemei:

Se consideră  $n$  procese (activități)  $p_1, p_2, \dots, p_n$  pentru care se cunosc timpii individuali de executare  $t_i$ , iar orice proces se poate executa doar accesând o resursă comună prin excludere reciprocă. Să se determine o modalitate de planificare a celor  $n$  procese, astfel încât timpul mediu de așteptare să fie minim.

## 2. Planificarea optimă a unor spectacole într-o singură sală

Considerăm  $n$  spectacole  $S_1, S_2, \dots, S_n$  pentru care cunoaștem intervalele lor de desfășurare  $[s_1, f_1), [s_2, f_2), \dots, [s_n, f_n)$ , toate dintr-o singură zi. Având la dispoziție o singură sală, în care putem să planificăm un singur spectacol la un moment dat, să se determine **numărul maxim de spectacole care pot fi planificate fără suprapuneri**. Un spectacol  $S_j$  poate fi programat după spectacolul  $S_i$  dacă  $s_j \geq f_i$ .

De exemplu, să considerăm  $n = 7$  spectacole având următoarele intervale de desfășurare:

$S_1: [10^{00}, 11^{20})$

$S_2: [09^{30}, 12^{10})$

$S_3: [08^{20}, 09^{50})$

$S_4: [11^{30}, 14^{00})$

$S_5: [12^{10}, 13^{10})$

$S_6: [14^{00}, 16^{00})$

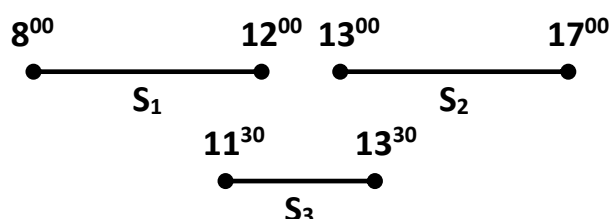
$S_7: [15^{00}, 15^{30})$

În acest caz, numărul maxim de spectacole care pot fi planificate este 4, iar o posibilă soluție este  $S_3, S_1, S_5$  și  $S_7$ .

Deoarece dorim să găsim o rezolvare de tip Greedy a acestei probleme, vom încerca planificarea spectacolelor folosind unul dintre următoarele criterii:

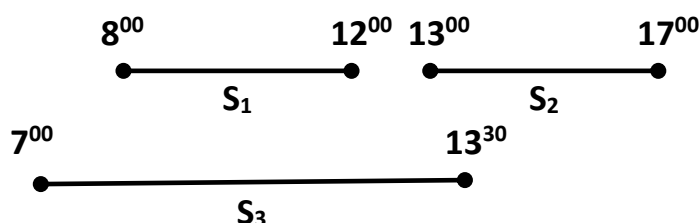
- în ordinea crescătoare a duratelor;
- în ordinea crescătoare a orelor de început;
- în ordinea crescătoare a orelor de terminare.

În cazul utilizării criteriului a), se observă ușor faptul că nu vom obține întotdeauna o soluție optimă. De exemplu, să considerăm următoarele 3 spectacole:



Aplicând criteriul a), vom planifica prima dată spectacolul  $S_3$  (deoarece durează cel mai puțin), iar apoi nu vom mai putea planifica nici spectacolul  $S_1$  și nici spectacolul  $S_2$ , deoarece ambele se suprapun cu spectacolul  $S_3$ , deci vom obține o planificare formată doar din  $S_3$ . Evident, planificarea optimă, cu număr maxim de spectacole, este  $S_1$  și  $S_2$ .

De asemenea, în cazul utilizării criteriului b), se observă ușor faptul că nu vom obține întotdeauna o soluție optimă. De exemplu, să considerăm următoarele 3 spectacole:



Aplicând criteriul b), vom planifica prima dată spectacolul  $S_3$  (deoarece începe primul), iar apoi nu vom mai putea planifica nici spectacolul  $S_1$  și nici spectacolul  $S_2$ , deoarece ambele se suprapun cu el, deci vom obține o planificare formată doar din  $S_3$ . Evident, planificarea optimă este  $S_1$  și  $S_2$ .

În cazul utilizării criteriului c), se observă faptul că vom obține soluțiile optime în ambele exemple prezentate mai sus:

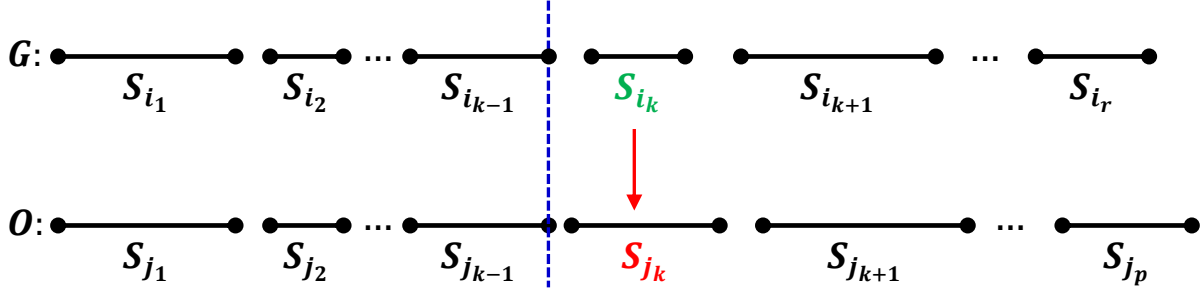
- în primul exemplu, vom planifica mai întâi spectacolul  $S_1$  (deoarece se termină primul), apoi nu vom putea planifica spectacolul  $S_3$  (deoarece se suprapune cu  $S_1$ ), dar vom putea planifica spectacolul  $S_2$ , deci vom obține planificarea optimă formată din  $S_1$  și  $S_2$ ;
- în al doilea exemplu, vom proceda la fel și vom obține planificarea optimă formată din  $S_1$  și  $S_2$ .

Practic, criteriul c) este o combinație a criteriilor a) și b), deoarece un spectacol care durează puțin și începe devreme se va termina devreme!

Pentru a demonstra optimalitatea criteriului c) de selecție, vom utiliza o demonstrație de tipul *exchange argument*: vom considera o soluție optimă furnizată de

un algoritm oarecare (nu contează metoda utilizată!), diferită de soluția furnizată de algoritmul de tip Greedy, și vom demonstra faptul că aceasta poate fi transformată, element cu element, în soluția furnizată de algoritmul de tip Greedy. Astfel, vom demonstra faptul că și soluția furnizată de algoritmul de tip Greedy este tot optimă!

Fie  $G$  soluția furnizată de algoritmul de tip Greedy și o soluție optimă  $O$ , diferită de  $G$ , obținută folosind orice alt algoritm:



Deoarece soluția optimă  $O$  este diferită de soluția Greedy  $G$ , rezultă că există un cel mai mic indice  $k$  pentru care  $S_{i_k} \neq S_{j_k}$ . Practic, este posibil ca ambii algoritmi pot să selecteze, până la pasul  $k - 1$ , aceleași spectacole în aceeași ordine, adică  $S_{i_1} = S_{j_1}, \dots, S_{i_{k-1}} = S_{j_{k-1}}$ . Spectacolul  $S_{j_k}$  din soluția optimă  $O$  poate fi înlocuit cu spectacolul  $S_{i_k}$  din soluția Greedy  $G$  fără a produce o suprapunere, deoarece:

- spectacolul  $S_{i_k}$  începe după spectacolul  $S_{j_{k-1}}$ , deoarece spectacolul  $S_{i_k}$  a fost programat după spectacolul  $S_{i_{k-1}}$  care este identic cu spectacolul  $S_{j_{k-1}}$ , deci  $s_{i_k} \geq f_{i_{k-1}} = f_{j_{k-1}}$ ;
- spectacolul  $S_{j_k}$  se termină după spectacolul  $S_{i_k}$ , adică  $f_{j_k} \geq f_{i_k}$ , deoarece, în caz contrar ( $f_{j_k} < f_{i_k}$ ) algoritmul Greedy ar fi selectat spectacolul  $S_{j_k}$  în locul spectacolului  $S_{i_k}$ ;
- spectacolul  $S_{i_k}$  se termină înaintea spectacolului  $S_{j_{k+1}}$ , adică  $f_{i_k} \leq s_{j_{k+1}}$ , deoarece am demonstrat anterior faptul că  $f_{i_k} \leq f_{j_k}$  și  $f_{j_k} \leq s_{j_{k+1}}$  (deoarece spectacolul  $S_{j_{k+1}}$  a fost programat după spectacolul  $S_{j_k}$ ).

Astfel, am demonstrat faptul că  $f_{j_{k-1}} \leq s_{i_k} < f_{j_k} \leq s_{j_{k+1}}$ , ceea ce ne permite să înlocuim spectacolul  $S_{j_k}$  din soluția optimă  $O$  cu spectacolul  $S_{i_k}$  din soluția Greedy  $G$  fără a produce o suprapunere. Repetând raționamentul anterior, putem transforma primele  $r$  elemente din soluția optimă  $O$  în soluția  $G$  furnizată de algoritmul Greedy.

Pentru a încheia demonstrația, trebuie să mai demonstrăm faptul că ambele soluții conțin același număr de spectacole, respectiv  $r = p$ . Presupunem prin absurd faptul că  $r \neq p$ . Deoarece soluția  $O$  este optimă, rezultă faptul că  $p > r$  (altfel, dacă  $p < r$ , ar însemna că soluția optimă  $O$  conține mai puține spectacole decât soluția Greedy  $G$ , ceea ce i-ar contrazice optimalitatea), deci există cel puțin un spectacol  $S_{j_{r+1}}$  în soluția optimă  $O$  care nu a fost selectat în soluția Greedy  $G$ . Acest lucru este imposibil, deoarece am demonstrat anterior faptul că orice spectacol  $S_{j_k}$  din soluția optimă se termină după spectacolul  $S_{i_k}$  aflat pe aceeași poziție în soluția Greedy (adică  $f_{j_k} \geq f_{i_k}$ ), deci am obține relația  $f_{i_r} \leq f_{j_r} \leq s_{j_{r+1}}$ , ceea ce ar însemna că spectacolul  $S_{j_{r+1}}$  ar fi trebuit să fie selectat și în soluția Greedy  $G$ ! În concluzie, presupunerea că  $r \neq p$  este falsă, deci  $r = p$ .



Astfel, am demonstrat faptul că putem transforma soluția optimă  $O$  în soluția  $G$  furnizată de algoritmul Greedy, deci și soluția furnizată de algoritmul Greedy este optimă!

În concluzie, algoritmul Greedy pentru rezolvarea problemei programării spectacolelor este următorul:

- sortăm spectacolele în ordinea crescătoare a orelor de terminare;
- planificăm primul spectacol (problema are întotdeauna soluție!);
- pentru fiecare spectacol rămas, verificăm dacă începe după ultimul spectacol programat și, în caz afirmativ, îl planificăm și pe el.

Citirea datelor de intrare are complexitatea  $\mathcal{O}(n)$ , sortarea are complexitatea  $\mathcal{O}(n \log_2 n)$ , programarea primului spectacol are complexitatea  $\mathcal{O}(1)$ , testarea spectacolelor rămase are complexitatea  $\mathcal{O}(n - 1)$ , iar afișarea planificării optime are cel mult complexitatea  $\mathcal{O}(n)$ , deci complexitatea algoritmului este  $\mathcal{O}(n \log_2 n)$ .

În continuare, vom prezenta implementarea algoritmului în limbajul C++ (pentru a compara ușor ore între ele, am transformat o oră dată sub forma  $hh:mm$  în minute față de miezul nopții, adică  $hh \cdot 60 + mm$ ):

```
#include<iostream>
#include<iomanip>
#include<fstream>
#include<algorithm>

using namespace std;

//structura Spectacol permite memorarea informațiilor despre un
//spectacol: numărul de ordine inițial (ID), ora de început (ts)
//și ora de terminare (tf) - ambele ore fiind memorate în minute
//față de miezul nopții
struct Spectacol
{
    int ID, ts, tf;
};

//funcție comparator utilizată pentru a sorta un tablou cu
elemente
//de tip Spectacol în ordinea crescătoare a orelor de terminare
bool cmpSpectacole(Spectacol p, Spectacol q)
{
    return p.tf <= q.tf;
}

int main()
{
    //n = numărul de spectacole date, k = numărul maxim de
    //spectacole programate, iar hs, ms, hf, mf = variabile
```

```

//utilizate pentru a citi un interval orar de forma hh:mm-
hh:mm
int i, k, n, hs, ms, hf, mf;
//variabila aux este folosita pentru a citi in gol
//caracterele ':' si '-' dintr-un interval orar
char aux;

//S = spectacolele date, P = spectacolele planificate
Spectacol *S, *P;

//datele de intrare se citesc din fişierul "spectacole.txt"
//care conţine pe prima linie numărul de spectacole n, iar pe
//fiecare din următoarele n linii câte un interval de
//desfăşurare al unui spectacol (de forma hh:mm-hh:mm)
ifstream fin("spectacole.txt");

fin >> n;
S = new Spectacol[n];
P = new Spectacol[n];

for(i = 0; i < n; i++)
{
    // 10 : 20 - 11 : 30
    fin>>hs>>aux>>ms>>aux>>hf>>aux>>mf;

    //ID-ul unui spectacol este numărul său de ordine iniţial
    S[i].ID = i + 1;

    //ora de început şi ora de terminare se transformă în
minute
    //faţă de miezul nopţii
    S[i].ts = hs * 60 + ms;
    S[i].tf = hf * 60 + mf;
}

fin.close();

//sortăm spectacolele în ordinea crescătoare a orelor de
//terminare
sort(S, S + n, cmpSpectacole);

//planificăm primul spectacol, deci numărul k al
spectacolelor
//planificate devine 1
P[0] = S[0];
k = 1;

//pentru fiecare spectacol rămas, verificăm dacă începe după

```

```

        //ultimul spectacol programat și, în caz afirmativ, îl
        adăugăm
        //și pe el în planificarea optimă
        for(i = 1; i < n; i++)
            if(S[i].ts >= P[k-1].tf)
                P[k++] = S[i];

        //afișăm soluția optimă determinată
        cout << "O planificare cu numar maxim de " << k
              << " spectacole:\n" << endl;
        for(i = 0; i < k; i++)
            cout << "S" << P[i].ID << " -> " << setfill('0') <<
setw(2)
              << P[i].ts/60 << ":" << setfill('0') << setw(2)
              << P[i].ts%60 << "-" << setfill('0') << setw(2)
              << setfill('0') << setw(2) << P[i].tf/60 << ":"
              << setfill('0') << setw(2) << P[i].tf%60 << endl;

        delete []S;
        delete []P;

        return 0;
}

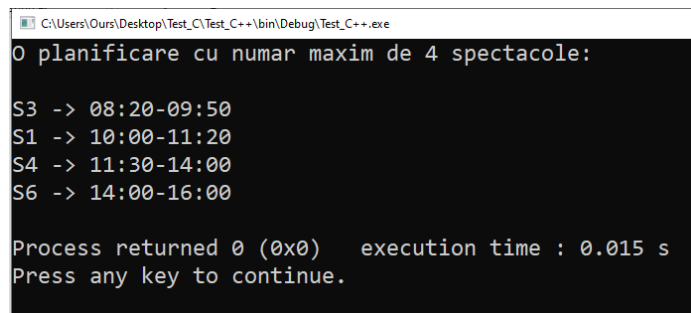
```

#### spectacole.txt

```

7
10:00-11:20
09:30-12:10
08:20-09:50
11:30-14:00
12:10-13:10
14:00-16:00
15:00-15:30

```



```

C:\Users\Ours\Desktop\Test_C\Test_C++\bin\Debug\Test_C++.exe
0 planificare cu numar maxim de 4 spectacole:

S3 -> 08:20-09:50
S1 -> 10:00-11:20
S4 -> 11:30-14:00
S6 -> 14:00-16:00

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.

```

Încheiem prezentarea acestei probleme precizând faptul că este tot o problemă de planificare, forma sa generală fiind următoarea: "*Se consideră  $n$  activități pentru care se cunosc intervalele orare de desfășurare și care partajează o resursă comună. Știind faptul că activitățile trebuie efectuate sub excludere reciprocă (respectiv, la un moment dat resursa comună poate fi alocată unei singure activități), să se determine o modalitate de planificare a unui număr maxim de activități care nu se suprapun.*".