CURS 8
TEHNICA DE PROGRAMARE "GREEDY"

1. 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:

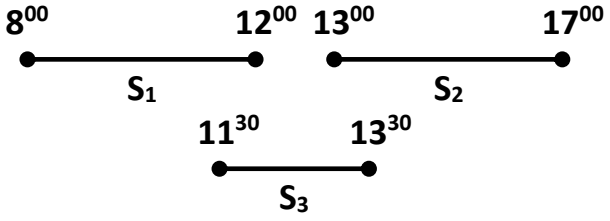
S_1: $[10^{00}, 11^{20})$
S_2: $[09^{30}, 12^{10})$
S_3: $[08^{20}, 09^{50})$
S_4: $[11^{30}, 14^{00})$
S_5: $[12^{10}, 13^{10})$
S_6: $[14^{00}, 16^{00})$
S_7: $[15^{00}, 15^{30})$

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:

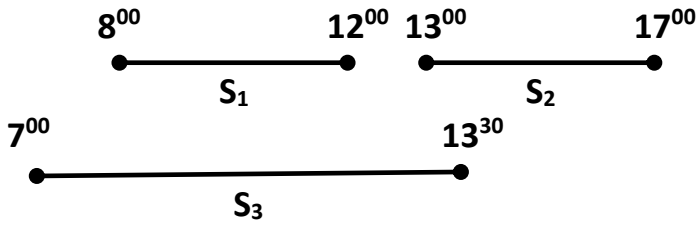
a) în ordinea crescătoare a duratelor;
b) în ordinea crescătoare a orelor de început;
c) în ordinea crescătoare a orelor de terminare.

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.

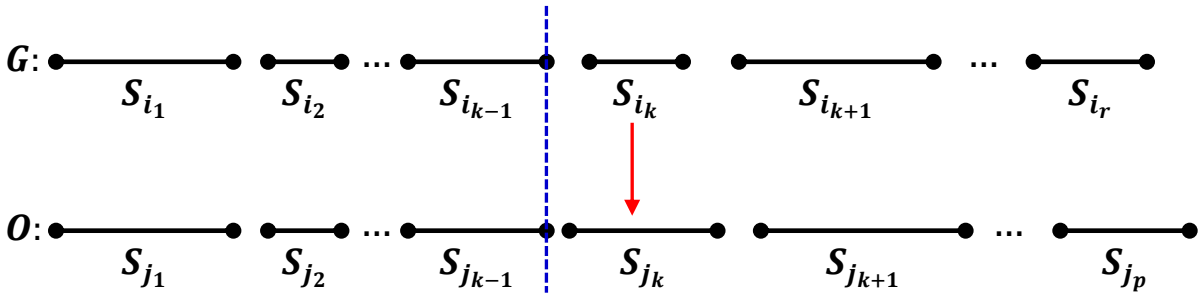
În cazul utilizării criteriului c), se observă faptul că vom obține soluțiile optime în ambele exemple prezentate mai sus:

- 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 optima 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*60+mm`):

```
#include<stdio.h>
#include<stdlib.h>

//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
typedef struct
{
    int ID, ts, tf;
}Spectacol;

//funcție comparator utilizată pentru a sorta un tablou cu elemente
//de tip Spectacol în ordinea crescătoare a orelor de terminare
int cmpSpectacole(const void *p, const void *q)
{
    return ((Spectacol*)p)->tf - ((Spectacol*)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;

    //S = spectacolele date, P = spectacolele planificate
    Spectacol S[100], P[100];

    //datele de intrare se citesc din fișierul "spectacole.txt"
    //care conține pe prima linie numărul de spectacole n, iar pe
    //fiecare din următoarele n linii câte un interval de
    //desfășurare al unui spectacol (de forma hh:mm-hh:mm)
    FILE *fin = fopen("spectacole.txt", "r");

    fscanf(fin, "%d", &n);
    for(i = 0; i < n; i++)
    {
        fscanf(fin, "%d:%d-%d:%d", &hs, &ms, &hf, &mf);

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

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

    fclose(fin);

    //sortăm spectacolele în ordinea crescătoare a orelor de
    //terminare
    qsort(S, n, sizeof(Spectacol), 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
    printf("O planificare cu numar maxim de %d spectacole:\n", k);
    for(i = 0; i < k; i++)
        printf("S%d -> %02d:%02d-%02d:%02d\n", P[i].ID,
                  P[i].ts/60, P[i].ts%60, P[i].tf/60, P[i].tf%60);

    return 0;
}
```

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.*".

2. Problema rucsacului (varianta continuă/fracționară)

Considerăm un rucsac având capacitatea maximă G și n obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaștem greutățile lor $g_1, g_2, \dots, g_n$ și câștigurile $c_1, c_2, \dots, c_n$ obținute prin încărcarea lor completă în rucsac. Știind faptul că orice obiect poate fi încărcat și fracționat (doar o parte din el), să se determine o modalitate de încărcare a rucsacului astfel încât câștigul total obținut să fie maxim. Dacă un obiect este încărcat fracționat, atunci vom obține un câștig direct proporțional cu fracțiunea încărcată din el (de exemplu, dacă vom încărca doar o treime dintr-un obiect, atunci vom obține un câștig egal cu o treime din câștigul integral asociat obiectului respectiv).

În afara variantei continue/fracționare a problemei rucsacului, mai există și varianta discretă a sa, în care un obiecte poate fi încărcat doar complet. Varianta respectivă nu se poate rezolva corect utilizând metoda Greedy, ci există alte metode de rezolvare, pe care le vom prezenta în cursul dedicat metodei programării dinamice.

Se observă foarte ușor faptul că varianta fracționară a problemei rucsacului are întotdeauna soluție (evident, dacă $G > 0$ și $n \geq 1$), chiar și în cazul în care cel mai mic obiect are o greutate strict mai mare decât capacitatea G a rucsacului (deoarece putem să încărcăm și fracțiuni dintr-un obiect), în timp ce varianta discretă nu ar avea soluție în acest caz.

Deoarece dorim să găsim o rezolvare de tip Greedy pentru varianta fracționară a problemei rucsacului, vom încerca să încărcăm obiectele în rucsac folosind unul dintre următoarele criterii:

a) în ordinea descrescătoare a câștigurilor integrale (cele mai valoroase obiecte ar fi primele încărcate);

b) în ordinea crescătoare a greutăţilor (cele mai mici obiecte ar fi primele încărcate, deci am încărca un număr mare de obiecte în rucsac);
c) în ordinea descrescătoare a greutăţilor.

Analizând cele 3 criterii propuse mai sus, putem găsi uşor contraexemple care să dovedească faptul că nu vom obţine o soluţii optime. De exemplu, criteriul c) ar putea fi corect doar presupunând faptul că, întotdeauna, un obiect cu greutate mare are asociat şi un câştig mare, ceea ce, evident, nu este adevărat! În cazul criteriului a), considerând $G = 10$ kg şi 3 obiecte având câştigurile $(100, 90, 80)$ RON şi greutăţile $(10, 5, 5)$ kg, vom încărca în rucsac primul obiect (deoarece are cel mai mare câştig integral) şi nu vom mai putea încărca niciun alt obiect, deci câştigul obţinut va fi de 100 RON. Totuşi, câştigul maxim de 170 RON se obţine încărcând în rucsac ultimele două obiecte! În mod asemănător (de exemplu, modificând câştigurilor obiectelor anterior menţionate în $(100, 9, 8)$ RON) se poate găsi un contraexemplu care să arate faptul că nici criteriul b) nu permite obţinerea unei soluţii optime în orice caz.

Se poate observa faptul că primele două criterii nu conduc întotdeauna la soluţia optimă deoarece ele iau în considerare fie doar câştigurile obiectelor, fie doar greutăţile lor, deci criteriul corect de selecţie trebuie să le ia în considerare pe ambele. Intuitiv, pentru a obţine un câştig maxim, trebuie să încărcăm mai întâi în rucsac obiectele care sunt cele mai "eficiente", adică au un câştig mare şi o greutate mică. Această "eficienţă" se poate cuantifica prin intermediul *câştigului unitar* al unui obiect, adică prin raportul $u_i = c_i/g_i$.

Algoritmul Greedy pentru rezolvarea variantei fracţionare a problemei rucsacului este următorul:

- sortăm obiectele în ordinea descrescătoare a câştigurilor unitare;
- pentru fiecare obiect testăm dacă încape integral în spaţiul liber din rucsac, iar în caz afirmativ îl încărcăm complet în rucsac, altfel calculăm fracţiunea din el pe care trebuie să o încărcăm astfel încât să umplem complet rucsacul (după încărcarea oricărui obiect, actualizăm spaţiul liber din rucsac şi câştigul total);
- algoritmul se termină fie când am încărcat toate obiectele în rucsac (în cazul în care $g_1 + g_2 + \cdots + g_n \leq G$), fie când nu mai există spaţiu liber în rucsac.

De exemplu, să considerăm un rucsac în care putem să încărcăm maxim $G = 53$ kg şi $n = 7$ obiecte, având greutăţile $g = (10, 5, 18, 20, 8, 40, 20)$ kg şi câştigurile integrale $c = (30, 40, 36, 10, 16, 30, 20)$ RON. Câştigurile unitare ale celor 7 obiecte sunt $u = \left(\frac{30}{10}, \frac{40}{5}, \frac{36}{18}, \frac{10}{20}, \frac{16}{8}, \frac{30}{40}, \frac{20}{20}\right) = (3, 8, 2, 0.5, 2, 0.75, 1)$ RON/kg, deci sortând descrescător obiectele în funcţie de câştigul unitar vom obţine următoarea ordine a lor: $O_2, O_1, O_3, O_5, O_7, O_6, O_4$. Prin aplicarea algoritmului Greedy prezentat anterior asupra acestor date de intrare, vom obţine următoarele rezultate:

Obiectul curent	Fracţiunea încărcată din obiectul curent	Spaţiul liber în rucsac	Câştigul total
—	—	53	0
O_2: $c_2 = 40, g_2 = 5 \leq 53$	1	$53 - 5 = 48$	$0 + 40 = 40$

Obiectul curent	Fracţiunea încărcată din obiectul curent	Spaţiul liber în rucsac	Câştigul total
O_1: $c_1 = 30, g_1 = 10 \leq 48$	1	48 – 10 = 38	40 + 30 = 70
O_3: $c_3 = 36, g_3 = 18 \leq 38$	1	38 – 18 = 20	70 + 36 = 106
O_5: $c_5 = 16, g_5 = 8 \leq 20$	1	20 – 8 = 12	106 + 16 = 122
O_7: $c_7 = 20, g_7 = 20 > 12$	12/20 = 0.6	0	122 + 0.6*20 = 134

În concluzie, pentru a obţine un câştig maxim de 134 RON, trebuie să încărcăm integral în rucsac obiectele O_2, O_1, O_3, O_5 şi o fracţiune de $0.6 = \frac{3}{5}$ din obiectul O_7.

Înainte de a demonstra corectitudinea algoritmului prezentat, vom face următoarele observaţii:

- vom considera obiectele $O_1, O_2, \dots, O_n$ ca fiind sortate descrescător în funcţie de câştigurile lor unitare, respectiv $\frac{c_1}{g_1} \geq \frac{c_2}{g_2} \geq \cdots \geq \frac{c_n}{g_n}$;
- o soluţie a problemei va fi reprezentată sub forma unui tuplu $X = (x_1, x_2, \dots, x_n)$, unde $x_i \in [0,1]$ reprezintă fracţiunea selectată din obiectul O_i;
- o soluţie furnizată de algoritmul Greedy va fi un tuplu de forma $X = (1, \dots, 1, x_j, 0, \dots, 0)$ cu n elemente, unde $x_j \in [0,1)$;
- în toate formulele vom considera implicit indicii ca fiind cuprinşi între 1 şi n;
- câştigul asociat unei soluţii a problemei de forma $X = (x_1, x_2, \dots, x_n)$ îl vom nota cu $C(X) = \sum c_i x_i$;
- dacă $g_1 + g_2 + \cdots + g_n \leq G$, atunci soluţia vom obţine soluţia banală $X = (1, \dots, 1)$, care este evident optimă, deci vom considera faptul că $g_1 + g_2 + \cdots + g_n > G$.

Fie $X = (1, \dots, 1, x_j, 0, \dots, 0)$, unde $x_j \in [0,1)$, soluţia furnizată de algoritmul Greedy prezentat, deci rucsacul va fi umplut complet (i.e., $\sum g_i x_i = G$). Presupunem că soluţia X nu este optimă, deci există o altă soluţie optimă $Y = (y_1, \dots, y_{k-1}, y_k, y_{k+1}, \dots, y_n)$ diferită de soluţia X, posibil obţinută folosind un alt algoritm. Deoarece Y este o soluţie optimă, obţinem imediat următoarele două relaţii: $\sum g_i y_i = G$ şi câştigul $C(Y) = \sum c_i y_i$ este maxim.

Deoarece $X \neq Y$, rezultă că există un cel mai mic indice k pentru care $x_k \neq y_k$, având următoarele proprietăţi:

- $k \leq j$ (deoarece, în caz contrar, am obţine $\sum g_i y_i > G$);
- $y_k < x_k$ (pentru $k < j$ este evident deoarece $x_k = 1$, iar dacă $y_j > x_j$ am obţine $\sum g_i y_i > G$).

Considerăm acum soluţia $Y' = (y_1, \dots, y_{k-1}, x_k, \alpha y_{k+1}, \dots, \alpha y_n)$, unde α este o constantă reală subunitară aleasă astfel încât $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n = G$. Practic, soluţia Y' a fost construită din soluţia Y, astfel:

- am păstrat primele $k-1$ componente din soluția Y;
- am înlocuit componenta y_k cu componenta x_k;
- deoarece $x_k > y_k$, am micșorat restul componentelor $y_{k+1}, \dots, y_n$ din soluția Y, înmulțindu-le cu o constantă subunitară α aleasă astfel încât rucsacul să rămână încărcat complet: $g_1y_1 + \cdots + g_{k-1}y_{k-1} + g_kx_k + g_{k+1}\alpha y_{k+1} + \cdots + g_n\alpha y_n = G$.

Deoarece Y este soluție a problemei, înseamnă că $g_1y_1 + \cdots + g_{k-1}y_{k-1} + g_ky_k + g_{k+1}y_{k+1} + \cdots + g_ny_n = G$. Dar și $g_1y_1 + \cdots + g_{k-1}y_{k-1} + g_kx_k + g_{k+1}\alpha y_{k+1} + \cdots + g_n\alpha y_n = G$, deci $g_1y_1 + \cdots + g_{k-1}y_{k-1} + g_ky_k + g_{k+1}y_{k+1} + \cdots + g_ny_n = g_1y_1 + \cdots + g_{k-1}y_{k-1} + g_kx_k + g_{k+1}\alpha y_{k+1} + \cdots + g_n\alpha y_n$, de unde obținem, după reducerea termenilor egali, relația $g_ky_k + g_{k+1}y_{k+1} + \cdots + g_ny_n = g_kx_k + g_{k+1}\alpha y_{k+1} + \cdots + g_n\alpha y_n$, pe care o putem rescrie astfel:

$$g_k(x_k - y_k) = (1-\alpha)(g_{k+1}y_{k+1} + \cdots + g_ny_n) \tag{1}$$

Comparăm acum câștigurile asociate soluțiilor Y și Y', calculând diferența dintre ele:

$$\begin{aligned} C(Y') - C(Y) &= c_1y_1 + \cdots + c_{k-1}y_{k-1} + c_kx_k + c_{k+1}\alpha y_{k+1} + \cdots + c_n\alpha y_n \\ &\quad - (c_1y_1 + \cdots + c_{k-1}y_{k-1} + c_ky_k + c_{k+1}y_{k+1} + \cdots + c_ny_n) = \\ &= c_k(x_k - y_k) + (\alpha - 1)(c_{k+1}y_{k+1} + \cdots + c_ny_n) = \\ &= \frac{c_k}{g_k}\left[g_k(x_k - y_k) + (\alpha - 1)\left(\frac{g_k}{c_k}c_{k+1}y_{k+1} + \cdots + \frac{g_k}{c_k}c_ny_n\right)\right] \end{aligned}$$

Rescriind ultima relație, obținem:

$$C(Y') - C(Y) = \frac{c_k}{g_k}\left[g_k(x_k - y_k) + (\alpha - 1)\left(\frac{g_kc_{k+1}}{c_k}y_{k+1} + \cdots + \frac{g_kc_n}{c_k}y_n\right)\right] \tag{2}$$

Dar $\frac{g_k}{c_k} \le \frac{g_i}{c_i}$ pentru orice $i > k$ (deoarece obiectele sunt sortate descrescător în funcție de câștigurile lor unitare, deci $\frac{c_k}{g_k} \ge \frac{c_i}{g_i}$ pentru orice $i > k$), de unde rezultă că $\frac{g_kc_i}{c_k} \le g_i$ pentru orice $i > k$, deci obținem relațiile:

$$\frac{g_kc_{k+1}}{c_k} \le g_{k+1}, \dots, \frac{g_kc_n}{c_k} \le g_n \tag{3}$$

Aplicând relațiile (3) în relația (2), obținem:

$$C(Y') - C(Y) \ge \frac{c_k}{g_k}[g_k(x_k - y_k) + (\alpha - 1)(g_{k+1}y_{k+1} + \cdots + g_ny_n)] \tag{4}$$

Din relația (1) obținem că $g_k(x_k - y_k) + (\alpha - 1)(g_{k+1}y_{k+1} + \cdots + g_ny_n) = 0$, deci relația (4) devine $C(Y') - C(Y) \ge 0$, de unde rezultă faptul că $C(Y') \ge C(Y)$. Există acum două posibilități:

a) $C(Y') > C(Y)$, ceea ce contrazice optimalitatea soluției Y, așadar presupunerea că ar exista o soluție optimă Y diferită de soluția X furnizată de algoritmul Greedy este falsă, ceea ce înseamnă că $X = Y$, deci și soluția furnizată de algoritmul Greedy este optimă;
b) $C(Y') = C(Y)$, ceea ce înseamnă că putem să reluăm procedeul prezentat anterior înlocuind Y cu Y' până când, după un număr finit de pași, vom obține o contradicție de tipul celei de la punctul a).

În concluzie, după un număr finit de pași, vom transforma soluția optimă Y în soluția X furnizată de algoritmul Greedy, ceea ce înseamnă că și soluția furnizată de algoritmul Greedy este, de asemenea, optimă.

În continuare, vom prezenta implementarea în limbajul C a algoritmului Greedy pentru rezolvarea variantei fracționare a problemei rucsacului:

```
#include<stdio.h>
#include<stdlib.h>

//structura Obiect permite memorarea informațiilor despre un obiect
//ID-ul unui obiect este numărul său de ordine inițial
typedef struct
{
    float castig, greutate;
    int ID;
}Obiect;

//funcție comparator utilizată pentru a sorta un tablou cu elemente
//de tip Obiect în ordinea descrescătoare a câștigurilor unitare
int cmpObiecte(const void *a, const void *b)
{
    Obiect pa = *(Obiect*)a;
    Obiect pb = *(Obiect*)b;

    if(pa.castig/pa.greutate < pb.castig/pb.greutate)
        return 1;

    if(pa.castig/pa.greutate > pb.castig/pb.greutate)
        return -1;

    return 0;
}

int main()
{
    //n = numărul de obiecte
    int i, n;
    //G = capacitatea maximă a rucsacului
    //total = câștigul total (maxim)
    //p = fracțiunea care se va încărca din ultimul obiect care
    //poate fi selectat pentru a umple complet rucsacul
    float G, total, p;
    //ob = obiectele date
    Obiect ob[1000];
```

```
    //datele de intrare se vor citi din fișierul text "obiecte.txt"
    //care conține pe prima linie numărul n de obiecte, pe
    //următoarele n linii sunt informațiile despre cele n obiecte
    //sub forma "greutate câștig", iar pe ultima linie se găsește
    //capacitatea maximă G a rucsacului
    FILE* fin = fopen("obiecte.txt", "r");
    fscanf(fin, "%d", &n);
    for(i = 0; i < n; i++)
    {
        fscanf(fin, "%f %f", &ob[i].greutate, &ob[i].castig);
        ob[i].ID = i+1;
    }
    fscanf(fin, "%f", &G);
    fclose(fin);

    //sortam obiectele descrescător în funcție de câștigul unitar
    qsort(ob, n, sizeof(Obiect), cmpObiecte);

    //inițializăm câștigul total (maxim) cu 0
    total = 0;
    for (i = 0; i < n; i++)
        //dacă obiectul curent ob[i] încape complet în rucsac,
        //îl încărcăm, după care actualizăm câștigul total și
        //spațiul liber din rucsac
        if (ob[i].greutate <= G)
        {
            printf("Obiectul %d -> 100%%\n", ob[i].ID);
            total = total + ob[i].castig;
            G = G - ob[i].greutate;
        }
        //dacă obiectul curent ob[i] nu încape complet în rucsac,
        //calculăm fracțiunea din el pe care trebuie să o încărcăm
        //pentru a umple complet rucsacul, actualizăm câștigul
        //total și algoritmul se termină

        else
        {
            p = G / ob[i].greutate;
            printf("Obiectul %d -> %.2f%%\n", ob[i].ID, p*100);
            total = total + p*ob[i].castig;
            break;
        }

    //afișăm câștigul total (maxim) pe care l-am obținut
    printf("Castig total: %.2f\n", total);

    return 0;
}
```

Citirea datelor de intrare are complexitatea $\mathcal{O}(n)$, sortarea are complexitatea $\mathcal{O}(n \log_2 n)$, selectarea și încărcarea obiectelor în rucsac are cel mult complexitatea $\mathcal{O}(n)$, iar afișarea câștigului maxim obținut $\mathcal{O}(1)$, deci complexitatea algoritmului este $\mathcal{O}(n \log_2 n)$.

3. Planificarea proiectelor cu profit maxim

Se consideră n proiecte $P_1, P_2, \dots, P_n$ pe care poate să le execute o echipă de programatori într-o anumită perioadă de timp (de exemplu, o lună), iar pentru fiecare proiect se cunoaşte termenul său limită (exprimat prin numărul de ordine al unei zi din perioada respectivă), precum şi profitul pe care îl va obţine echipa dacă proiectul este finalizat la timp (altfel, echipa nu va obţine niciun profit pentru proiectul respectiv). Echipa are la dispoziţie exact o zi pentru executarea oricărui proiect. Să se determine o modalitate de planificare a proiectelor astfel încât profitul obţinut de echipă să fie maxim.

Exemplu:
Vom considera faptul că datele de intrare se citesc din fişierul text *proiecte.in*, care conţine pe prima linie numărul n de proiecte, iar fiecare dintre următoarele n linii conţine termenul limită şi profitul unui proiect. De exemplu, a doua linie din fişierul de intrare conţine informaţiile despre proiectul P_1, respectiv termenul său limită egal cu 2 (ceea ce înseamnă că proiectul trebuie executat fie în prima zi, fie în a doua) şi profitul egal cu 800 RON. Datele de ieşire se vor scrie în fişierul text *proiecte.out*, în forma indicată mai jos.

proiecte.in	**proiecte.out**
8 2 800 5 700.75 1 150 2 900 1 850 3 1000 3 950.50 2 900	Ziua 1: Proiectul 4 - 900.50 RON Ziua 2: Proiectul 7 - 950.00 RON Ziua 3: Proiectul 6 - 1000.00 RON Ziua 5: Proiectul 2 - 700.75 RON Profitul maxim al echipei: 3551.25 RON

În primul rând, observăm faptul că numărul maxim de zile în care putem să planificăm proiecte este egal cu maximul termenelor limită (în exemplul dat, acesta este egal cu 5).

O prima idee de rezolvare ar fi aceea de a planifica în fiecare zi proiectul neplanificat care are profitul maxim. Aplicând acest algoritm pentru datele de intrare de mai sus, vom planifica în ziua 1 proiectul P_6 (deoarece are profitul maxim de 1000 RON), în ziua 2 vom planifica proiectul P_7 (deoarece are profitul maxim de 950 RON dintre proiectele care mai pot fi planificate în ziua respectivă), iar în ziua 3 vom planifica proiectul P_2 (singurul care mai pot fi planificat în ziua respectivă), deci vom obţine un profit de 1000+950+700.75 = 2650.75 RON, mai mic decât profitul maxim de 3551.25 RON din exemplu.

O altă idee de rezolvare ar fi aceea de a planifica în fiecare zi proiectul cu profit maxim care are termenul limită în ziua respectivă. Aplicând acest algoritm pentru datele de intrare de mai sus, vom planifica în ziua 1 proiectul P_5 (deoarece are profitul maxim de 850 RON dintre proiectele care au termenul limită egal cu ziua 1, respectiv P_3 şi P_5), în ziua 2 vom planifica proiectul P_4 (deoarece are profitul maxim de 900.50 RON dintre proiectele care au termenul limită egal cu 2, respectiv P_1, P_4 şi P_8), în ziua 3 vom planifica proiectul P_6, în ziua 4 nu vom planifica niciun proiect, iar în ziua 5 vom planifica proiectul P_2, deci vom obţine un profit de 850+900.50+1000+700.75 = 3451.25 RON, mai mic decât profitul maxim de 3551.25 RON din exemplu.

Algoritmul optim de tip Greedy pentru rezolvarea acestei probleme se obține observând faptul că în fiecare dintre cele două variante prezentate anterior am programat prea repede proiectele cu profit maxim dintr-o anumită zi (de exemplu, în prima variantă, proiectul P_6 a fost planificat în ziua 1, deși el ar fi putut fi programat și în ziua 2 sau în ziua 3). Astfel, algoritmul Greedy optim constă în parcurgerea proiectelor în ordinea descrescătoare a profiturilor, iar fiecare proiect vom încerca să-l planificăm cât mai târziu, adică în ziua liberă cea mai apropiată de termenul limită al proiectului. Pentru exemplul de mai sus, vom considera proiectele în ordinea $P_6, P_7, P_4, P_8, P_5, P_1, P_2, P_3$. Proiectul P_6 poate fi planificat în ziua 3 (chiar termenul său limită), proiectul P_7 nu mai poate fi planificat tot în ziua 3, dar poate fi planificat în ziua 2, proiectul P_4 nu poate fi planificat în ziua 2, dar poate fi planificat în ziua 1, proiectele P_8, P_5 și P_1 nu mai pot fi planificate, proiectul P_5 poate fi planificat chiar în ziua 5, iar proiectul P_3 nu mai poate fi planificat. În concluzie, vom planifica proiectele P_6, P_7, P_4 și P_2, obținând profitul maxim de 3551.25 RON.

În continuare, vom prezenta direct implementarea în limbajul C a acestui algoritm de tip Greedy, argumentarea corectitudinii algoritmului fiind foarte simplă:

```
#include <stdio.h>
#include <stdlib.h>

//structura Proiect permite stocarea informațiilor despre un
//proiect, ID-ul fiind numărul său de ordine inițial
typedef struct
{
   int ID, termen;
   float profit;
}Proiect;




//funcție comparator utilizată pentru a sorta un tablou cu elemente
//de tip Proiect în ordinea descrescătoare a profiturilor
int cmpProiecte(const void *a, const void *b)
{
    Proiect pa = *(Proiect*)a;
    Proiect pb = *(Proiect*)b;

    if(pa.profit < pb.profit)
        return 1;

    if(pa.profit > pb.profit)
        return -1;

    return 0;
}

int main()
{
   //n = numărul de proiecte, zmax = maximul termenelor limită
   int n, zmax, i, j;
   //pmax = profitul maxim obținut de echipă
```

```
float pmax;
//p = tablou care conține informațiile despre cele n  proiecte
//z = tablou care conține planificarea optimă (cu câștig maxim)
Proiect *p, *z;

FILE *f;

//citirea datelor din fișierul de intrare
f = fopen("proiecte.in", "r");

fscanf(f,"%d", &n);

//alocăm dinamic tabloul p, în care vom memora informațiile
//despre cele n proiecte
p = (Proiect*)malloc(n * sizeof(Proiect));

//în timpul citirii informațiilor despre proiecte, calculăm
//maximul termenelor limită în variabila zmax
zmax = 0;
for(i = 0; i < n; i++)
{
      p[i].ID = i + 1;
      fscanf(f, "%d %f", &p[i].termen, &p[i].profit);
      if(p[i].termen > zmax)
            zmax = p[i].termen;
}

fclose(f);

//în tabloul z vom planifica proiectele pe zile, iar zilele
//posibile sunt 1,2,...,zmax, deci îl alocăm cu zmax+1 elemente
//am utilizat funcția calloc pentru a inițializa toate câmpurile
//fiecărui element de tip structură cu valori nule
z = (Proiect*)calloc(zmax+1, sizeof(Proiect));

//sortăm proiectele descrescător după profituri
qsort(p, n, sizeof(Proiect), cmpProiecte);

//inițializăm profitul maxim pmax care poate obținut de echipă
pmax = 0;
//încercăm să planificăm fiecare proiect cât mai târziu posibil,
//dar fără a-i depăși termenul limită, deci căutăm prima zi
//liberă începând cu termenul său limită
for(i = 0; i < n; i++)
      for(j = p[i].termen; j >= 1; j--)
            //dacă z[j].ID este 0, înseamnă că nu am planificat
            //niciun proiect în ziua respectivă (ID-urile
            //proiectelor sunt cuprinse între 1 și n), deci ziua
            //respectivă este liberă
            if(z[j].ID == 0)
            {
                  z[j] = p[i];
                  pmax = pmax + p[i].profit;
                  //este esențial să ne oprim după ce planificăm
                  //un proiect într-o anumită zi j, altfel el va
                  //fi planificat și în toate zilele 1,2,...,j-1
```

```
                    //care sunt libere!!!
                    break;
                }

    //scriem planificarea optimă în fișierul text de ieșire
    f = fopen("proiecte.out", "w");

    for(i = 1; i <= zmax; i++)
          if(z[i].ID != 0)
                fprintf(f, "Ziua %d: Proiectul %d - %.2f RON\n",
                                              i, z[i].ID, z[i].profit);
    fprintf(f, "\nProfitul maxim al echipei: %.2f RON", pmax);
    fclose(f);

    //eliberăm cele două tablouri alocate dinamic
    free(p);
    free(z);

    return 0;
}
```

Complexitatea implementării prezentate mai sus este $\mathcal{O}(n^2)$ și nu este minimă, deși algoritmul Greedy este optim! Pentru a obține o implementare care să aibă complexitatea optimă $\mathcal{O}(n \log_2 n)$, trebuie să utilizăm o structură de date numită *Union-Find*: https://www.geeksforgeeks.org/job-sequencing-using-disjoint-set-union/.