TEHNICA DE PROGRAMARE "GREEDY"

1. Prezentare generală

Tehnica de programare Greedy este utilizată, de obicei, pentru rezolvarea problemelor de optimizare, adică a acelor probleme în care se cere determinarea unei submulțimi a unei mulțimi date pentru care se minimizează sau se maximizează valoarea unei funcții obiectiv. Formal, o problemă de optimizare poate fi enunțată astfel: "*Fie A o mulțime nevidă și $f: \mathcal{P}(A) \rightarrow \mathbb{R}$ o funcție obiectiv asociată mulțimii A, unde prin $\mathcal{P}(A)$ am notat mulțimea tuturor submulțimilor mulțimii A. Să se determine o submulțime $S \subseteq A$ astfel încât valoarea funcției f să fie minimă/maximă pe S (i.e., pentru orice altă submulțime $T \subseteq A, T \neq S$, valoarea funcției obiectiv f va fi cel puțin /cel mult egală cu valoarea funcției obiectiv f pe submulțimea S).*"

O problemă foarte simplă de optimizare este următoarea: "*Fie A o mulțime nevidă de numere întregi. Să se determine o submulțime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*". Se observă faptul că funcția obiectiv nu este dată în formă matematică și nu se precizează explicit faptul că suma elementelor submulțimii S trebuie să fie maximă *în raport cu suma oricărei alte submulțimi*, acest lucru subînțelegându-se. Formal, problema poate fi enunțată astfel: "*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ și $f: \mathcal{P}(A) \rightarrow \mathbb{R},\ f(S) = \sum_{x \in S} x$. Să se determine o submulțime $S \subseteq A$ astfel încât valoarea funcției f să fie maximă pe S, i.e. $\forall T \subseteq A, T \neq S \Rightarrow f(T) \leq f(S)$ sau, echivalent, $\forall T \subseteq A, T \neq S \Rightarrow \sum_{x \in T} x \leq \sum_{x \in S} x$.*"

Evident, orice problemă de acest tip poate fi rezolvată prin *metoda forței-brute*, astfel: se generează, pe rând, toate submulțimile S ale mulțimii A și pentru fiecare dintre ele se calculează $f(S)$, iar dacă valoarea obținută este mai mică/mai mare decât minimul/maximul obținut până în acel moment, atunci se actualizează minimul/maximul și se reține submulțimea S. Deși aceasta rezolvare este corectă, ea are o complexitate exponențială, respectiv $\mathcal{O}(2^{|A|})$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulțimea S, la fiecare pas, cel mai bun element disponibil din mulțimea A din punct de vedere al optimizării funcției obiectiv. Practic, metoda Greedy încearcă să găsească optimul global al funcției obiectiv combinând optimele sale locale. Totuși, prin combinarea unor optime locale nu se obține întotdeauna un optim global! De exemplu, să considerăm cel mai scurt drum posibil dintre București și Arad (un optim local), precum și cel mai scurt drum posibil dintre Arad și Ploiești (alt optim local). Combinând cele două optime locale nu vom obține un optim global, deoarece, evident, cel mai scurt drum de la București la Ploiești nu trece prin Arad! Din acest motiv, aplicare tehnicii de programare Greedy pentru rezolvarea unei probleme trebuie să fie însoțită de o demonstrație a corectitudinii (optimalității) criteriului de selecție pe care trebuie să-l îndeplinească un element al mulțimii A pentru a fi adăugat în soluția S.

De exemplu, să considerăm *problema plății unei sume folosind un număr minim de monede*. O rezolvare de tip Greedy a acestei probleme ar putea consta în utilizarea, la fiecare pas, a unui număr maxim de monede cu cea mai mare valoare admisibilă. Astfel, pentru monede cu valorile de 8$, 7$ și 5 $, o sumă de 23$ va fi plătită în următorul mod: 23$ = 2*8$ + 7$ = 2*8$ + 1*7$, deci se vor utiliza 3 monede, ceea ce reprezintă o soluție optimă. Dacă vom considera monede cu valorile de 8$, 7$ și 1 $, o sumă de 14$ va fi plătită în următorul mod: 14$ = 1*8$ + 6$ = 1*8$ + 6*1$, deci se vor utiliza 7 monede, ceea ce nu reprezintă o soluție optimă (i.e., 2*7$). Mai mult, pentru monede cu 8$, 7$ și 5 $, o sumă de 14$ nu va putea fi plătită deloc: 14$ = 1*8$ + 6$ = 1*8$ + 1*5$ + 1$, deoarece restul rămas, de 1$, nu mai poate fi plătit (evident, soluția optimă este 2*7$). În concluzie, acest algoritm de tip Greedy, numit *algoritmul casierului*, nu furnizează întotdeauna o soluție optimă pentru plata unei sume folosind un număr minim de monede. Totuși, pentru anumite valori ale monedelor, el poate furniza o soluție optimă pentru orice sumă dată (de exemplu, pentru monedele din Statele Unite ale Americii: https://personal.utdallas.edu/~sxb027100/cs6363/coin.pdf)

Revenind la problema determinării unei submulțimi S cu sumă maximă, observăm faptul că aceasta trebuie să conțină toate elementele pozitive din mulțimea A, deci criteriul de selecție este ca elementul curent din A să fie pozitiv (demonstrația optimalității este banală). Dacă mulțimea A nu conține niciun număr pozitiv, care va fi soluția problemei?

În anumite probleme, criteriul de selecție poate fi aplicat mai eficient dacă se realizează o prelucrare inițială a elementelor mulțimii A – de obicei, o sortare a lor. De exemplu, să considerăm următoarea problemă: "*Fie A o mulțime nevidă formată din n numere întregi. Să se determine o submulțime $S \subseteq A$ având exact k elemente $(k \leq n)$ cu proprietatea că suma elementelor sale este maximă.*". Evident, submulțimea S trebuie să conțină cele mai mari k elemente ale mulțimii A, iar acestea pot fi selectate în două moduri:

- de k ori se selectează maximul din mulțimea A și se elimină (sau doar se marchează – important este ca, la fiecare pas, să nu mai luăm în considerare maximul determinat anterior), deci această soluție va avea complexitatea $\mathcal{O}(kn)$, care oscilează între $\mathcal{O}(n)$ pentru valori ale lui k mult mai mici decât n și $\mathcal{O}(n^2)$ pentru valori ale lui k apropiate de n;
- sortăm crescător elementele mulțimii A și apoi selectăm ultimele k elemente, deci această soluție va avea complexitatea $\mathcal{O}(k + n \log_2 n) \approx \mathcal{O}(n \log_2 n)$, care nu depinde de valoarea k.

În plus, a doua varianta de implementare are avantajul unei implementări mai simple decât prima.

În concluzie, pentru o mulțime A cu n elemente, putem considera următoarea formă generală a unui algoritm de tip Greedy:

```
prelucrarea inițială a elementelor mulțimii A
S = []
for x in A:
   if elementul x verifică criteriul de selecție:
         S.append(x)
afișarea elementelor mulțimii S
```

Se observă faptul că, de obicei, un algoritm de acest tip are o complexitate relativ mică, de tipul $\mathcal{O}(n\log_2 n)$, dacă prin sortarea (prelucrarea) elementelor mulțimii A cu complexitatea $\mathcal{O}(n\log_2 n)$ se poate ulterior testa criteriul de selecție în $\mathcal{O}(1)$. Dacă nu se realizează prelucrarea inițială a elementelor mulțimii A, atunci algoritmul (care trebuie puțin adaptat) va avea complexități de tipul $\mathcal{O}(n)$ sau $\mathcal{O}(n^2)$, induse de complexitatea verificării criteriului de selecție. Evident, acestea nu sunt toate complexitățile posibile pentru un algoritm de tip Greedy, ci doar sunt cele mai des întâlnite!

2. 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 reașezare 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 $t_1 = 7,\ t_2 = 6,\ t_3 = 5,\ t_4 = 10,\ t_5 = 6$ și $t_6 = 4$. 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
Timpul mediu de așteptare (M):		$\frac{3+6+12+18+25+35}{6} = \frac{99}{6} = 16.5$

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

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumerota persoanele $p_1, p_2, \ldots, p_i, \ldots, p_j, \ldots, p_n$ în ordinea crescătoare a timpilor de servire, astfel încât vom avea $t_1 \leq t_2 \leq \cdots \leq t_i \leq \cdots \leq t_j \leq \cdots \leq t_n$. De asemenea, vom presupune faptul că timpii individuali de servire $t_1, t_2, \ldots, t_n$ nu sunt toți egali între ei (în acest caz, problema ar fi trivială), deci există $i < j$ astfel încât $t_i < t_j$. În continuare, presupunem faptul că această modalitate P_1 de aranjare a persoanelor la coadă (o permutare, de fapt) nu este optimă, deci există o altă modalitate optimă P_2 de aranjare $p_1, p_2, \ldots, p_j, \ldots, p_i, \ldots p_n$ diferită de cea inițială, în care $t_j > t_i$ (practic, am interschimbat persoanele p_i și p_j din varianta inițială, adică persoana p_j se află acum pe poziția i în coadă, iar persoana p_i se află acum pe poziția j, unde $i < j$).

În cazul primei modalități de aranjare P_1, timpul mediu de servire M_1 este egal cu:

$$M_1 = \frac{t_1 + (t_1 + t_2) + \cdots + (t_1 + \cdots + t_i) + \cdots + \left(t_1 + \cdots + t_j\right) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_i + \cdots + (n-j+1)t_j + \cdots + 2t_{n-1} + t_n}{n}$$

În cazul celei de-a doua modalități de aranjare P_2, timpul mediu de servire M_2 este egal cu:

$$M_2 = \frac{t_1 + (t_1 + t_2) + \cdots + \left(t_1 + \cdots + t_j\right) + \cdots + (t_1 + \cdots + t_i) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_j + \cdots + (n-j+1)t_i + \cdots + 2t_{n-1} + t_n}{n}$$

Comparăm acum M_1 cu M_2, calculând diferența dintre ele:

$$M_1 - M_2 = \frac{(n-i+1)t_i + (n-j+1)t_j - (n-i+1)t_j - (n-j+1)t_i}{n} =$$
$$= \frac{t_i(n-i+1-n+j-1) + t_j(n-j+1-n+i-1)}{n} =$$
$$= \frac{t_i(-i+j) + t_j(-j+i)}{n} = \frac{-t_i(i-j) + t_j(i-j)}{n} = \frac{(t_j - t_i)(i-j)}{n}$$

Deoarece $i < j$ și $t_j > t_i$, obținem faptul că $M_1 - M_2 = \frac{(t_j - t_i)(i-j)}{n} < 0$ (evident, $n \geq 1$), ceea ce implică $M_1 < M_2$. Acest fapt contrazice optimalitatea modalității de aranjare P_2, deci presupunerea că modalitatea de aranjare P_1 (în ordinea crescătoare a timpilor de servire) nu ar fi optimă este falsă!

Atenție, soluția acestei probleme constă într-o rearanjare a persoanelor $p_1, p_2, \dots, p_n$, deci în implementarea acestui algoritm nu este suficient să sortăm crescător timpii de servire, ci trebuie să memorăm perechi de forma (p_i, t_i), folosind, de exemplu, un tuplu, iar apoi să le sortăm crescător după componenta t_i.

```python
# functie folosita pentru sortarea crescătoare a persoanelor
# în raport de timpii de servire (cheia)
def cheieTimpServire(t):
    return t[1]

# funcția afișează, într-un format tabelar, timpii de servire
# și timpii de așteptare ai persoanelor
# ts = o listă cu timpii individuali de serivre
def afisareTimpi(ts):
    print("Persoana\tTimp de servire\tTimp de asteptare")
    # timpul de așteptare al persoanei curente
    tcrt = 0
    # timpul total de așteptare
    ttotal = 0
    for t in ts:
        tcrt = tcrt + t[1]
        ttotal = ttotal + tcrt
        print(str(t[0]).center(len("Persoana")),
              str(t[1]).center(len("Timp de serivire")),
              str(tcrt).center(len("Timp de așteptare")), sep="\t")
    print("Timpul mediu de așteptare:", round(ttotal/len(ts), 2))


# timpii de servire ai persoanelor se citesc de la tastatură
aux = [int(x) for x in input("Timpii de servire: ").split()]
# asociem fiecărui timp de servire numărul de ordine al persoanei
tis = [(i+1, aux[i]) for i in range(len(aux))]
```

```
print("Varianta inițială:")
afisareTimpi(tis)

# sortăm persoanele în ordinea crescătoare a timpilor de servire
tis.sort(key=cheieTimpServire)

print("\nVarianta optimă:")
afisareTimpi(tis)
```

Evident, complexitatea algoritmului este dată de complexitatea operației de sortare utilizate, deci complexitatea sa, optimă, este $\mathcal{O}(n \log_2 n)$.

Încheiem prezentarea acestei probleme precizând faptul că este o problemă de planificare, forma sa generală fiind următoarea: "*Se consideră n activități cu duratele $t_1, t_2, \dots, t_n$ 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 activităților astfel încât timpul mediu de așteptare să fie minim.*".

3. 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:

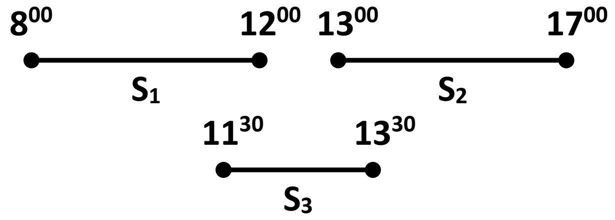
S_1: $[10^{00}, 11^{20})$
S_2: $[09^{30}, 12^{10})$
S_3: $[08^{20}, 09^{50})$
S_4: $[11^{30}, 14^{00})$
S_5: $[12^{10}, 13^{10})$
S_6: $[14^{00}, 16^{00})$
S_7: $[15^{00}, 15^{30})$

Se observă faptul că numărul maxim de spectacole care pot fi planificate este 4, iar o posibilă soluție este S_3, S_1, S_5 și S_7. Atenție, soluția nu este unică (de exemplu, o altă soluție optimă este S_3, S_1, S_5 și S_6)!

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

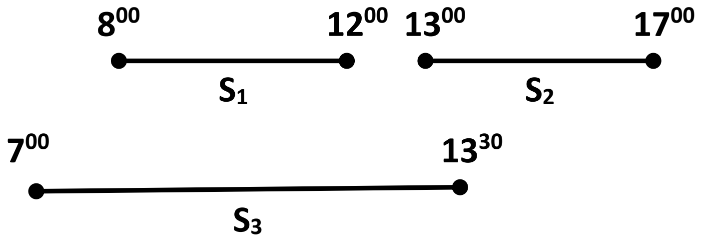
a) în ordinea crescătoare a duratelor;
b) în ordinea crescătoare a orelor de început;
c) în ordinea crescătoare a orelor de terminare.

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.

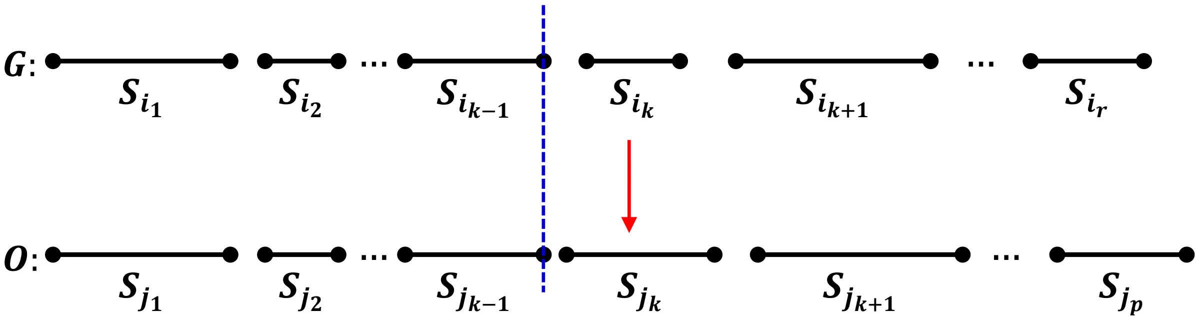
În cazul utilizării criteriului c), se observă faptul că vom obține soluțiile optime în ambele exemple prezentate mai sus:

- 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 $S_{i_1} = S_{j_1}, \ldots, S_{i_{k-1}} = S_{j_{k-1}}$. 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 Python:

```python
# functie folosita pentru sortarea crescătoare a spectacolelor
# în raport de ora de sfârșit (cheia)
def cheieOraSfârșit(sp):
    return sp[2]

# citim datele de intrare din fișierul text "spectacole.txt"
fin = open("spectacole.txt")
# lsp = lista spectacolelor, fiecare spectacol fiind memorat
# sub forma unui tuplu (ID, ora de început, ora de sfârșit)
lsp = []
crt = 1
for linie in fin:
    aux = linie.split("-")
    # aux[0] = ora de început a spectacolului curent
    # aux[1] = ora de sfârșit a spectacolului curent
    lsp.append((crt, aux[0].strip(), aux[1].strip()))
    crt = crt + 1
fin.close()

# sortăm spectacolele în ordinea crescătoare a timpilor de sfârșit
lsp.sort(key=cheieOraSfârșit)
```

```
# posp = o listă care conține o programare optima a spectacolelor,
# inițializată cu primul spectacol
posp = [lsp[0]]
# parcurgem restul spectacolelor
for sp in lsp[1:]:
    # dacă spectacolul curent începe după ultimul spectacol
    # programat, atunci îl programăm și pe el
    if sp[1] >= posp[len(posp)-1][2]:
        posp.append(sp)

# scriem datele de ieșire în fișierul text "programare.txt"
fout = open("programare.txt", "w")
fout.write("Numarul maxim de spectacole: "+str(len(posp))+"\n")
fout.write("\nSpectacolele programate:\n")
for sp in posp:
    fout.write(sp[1]+"-"+sp[2]+" Spectacol "+str(sp[0])+"\n")
fout.close()
```

Pentru exemplul de mai sus, fișierele text de intrare și de ieșire sunt următoarele:

spectacole.txt	**programare.txt**
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	Numarul maxim de spectacole: 4 Spectacolele programate: 08:20-09:50 Spectacol 3 10:00-11:20 Spectacol 1 12:10-13:10 Spectacol 5 15:00-15:30 Spectacol 7

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.*".

4. Problema rucsacului (varianta continuă/fracționară)

Considerăm un rucsac având capacitatea maximă G și n obiecte $O_1, O_2, \ldots, O_n$ pentru care cunoaștem greutățile lor $g_1, g_2, \ldots, g_n$ și câștigurile $c_1, c_2, \ldots, c_n$ 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 greutăților (cele mai mici obiecte ar fi primele încărcate, deci am încărca un număr mare de obiecte în rucsac);
c) în ordinea descrescătoare a greutăților.

Analizând cele 3 criterii propuse mai sus, putem găsi ușor contraexemple care să dovedească faptul că nu vom obține o soluții optime. De exemplu, criteriul c) ar putea fi corect doar presupunând faptul că, întotdeauna, un obiect cu greutate mare are asociat și un câștig mare, ceea ce, evident, nu este adevărat! În cazul criteriului a), considerând $G = 10$ kg și 3 obiecte având câștigurile $(100, 90, 80)$ RON și greutățile $(10, 5, 5)$ kg, vom încărca în rucsac primul obiect (deoarece are cel mai mare câștig integral) și nu vom mai putea încărca niciun alt obiect, deci câștigul obținut va fi de 100 RON. Totuși, câștigul maxim de 170 RON se obține încărcând în rucsac ultimele două obiecte! În mod asemănător (de exemplu, modificând câștigurilor obiectelor anterior menționate în $(100, 9, 8)$ RON) se poate găsi un contraexemplu care să arate faptul că nici criteriul b) nu permite obținerea unei soluții optime în orice caz.

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

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

- sortăm obiectele în ordinea descrescătoare a câștigurilor unitare;
- pentru fiecare obiect testăm dacă încape integral în spațiul liber din rucsac, iar în caz afirmativ îl încărcăm complet în rucsac, altfel calculăm fracțiunea din el pe care trebuie să o încărcăm astfel încât să umplem complet rucsacul (după încărcarea oricărui obiect, actualizăm spațiul liber din rucsac și câștigul 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 spațiu liber în rucsac.

De exemplu, să considerăm un rucsac în care putem să încărcăm maxim $G = 53$ kg și $n = 7$ obiecte, având greutățile $g = (10, 5, 18, 20, 8, 40, 20)$ kg și câștigurile integrale $c = (30, 40, 36, 10, 16, 30, 20)$ RON. Câștigurile 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 funcție de câștigul unitar vom obține 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 obține următoarele rezultate:

Obiectul curent	Fracțiunea încărcată din obiectul curent	Spațiul liber în rucsac	Câștigul total
—	—	53	0
O_2: $c_2 = 40, g_2 = 5 \leq 53$	1	53 – 5 = 48	0 + 40 = 40
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 obține un câștig maxim de 134 RON, trebuie să încărcăm integral în rucsac obiectele O_2, O_1, O_3, O_5 și o fracțiune de $0.6 = \frac{3}{5}$ din obiectul O_7.

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

- vom considera obiectele $O_1, O_2, \ldots, O_n$ ca fiind sortate descrescător în funcție de câștigurile lor unitare, respectiv $\frac{c_1}{g_1} \geq \frac{c_2}{g_2} \geq \cdots \geq \frac{c_n}{g_n}$;
- o soluție a problemei va fi reprezentată sub forma unui tuplu $X = (x_1, x_2, \ldots, x_n)$, unde $x_i \in [0,1]$ reprezintă fracțiunea selectată din obiectul O_i;
- o soluție furnizată de algoritmul Greedy va fi un tuplu de forma $X = (1, \ldots, 1, x_j, 0, \ldots, 0)$ cu n elemente, unde $x_j \in [0,1)$;
- în toate formulele vom considera implicit indicii ca fiind cuprinși între 1 și n;

- câștigul asociat unei soluții 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 soluția vom obține soluția 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 $x_j \in [0,1)$, soluția furnizată de algoritmul Greedy prezentat, deci rucsacul va fi umplut complet (i.e., $\sum g_i x_i = G$). Presupunem că soluția X nu este optimă, deci există o altă soluție optimă $Y = (y_1, \dots, y_{k-1}, y_k, y_{k+1}, \dots, y_n)$ diferită de soluția X, posibil obținută folosind un alt algoritm. Deoarece Y este o soluție optimă, obținem imediat următoarele două relații: $\sum g_i y_i = G$ și câștigul $C(Y) = \sum c_i y_i$ este maxim.

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

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

Considerăm acum soluția $Y' = (y_1, \dots, y_{k-1}, x_k, \alpha y_{k+1}, \dots, \alpha y_n)$, 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, soluția Y' a fost construită din soluția 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: $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n = G$.

Deoarece Y este soluție a problemei, înseamnă că $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k y_k + g_{k+1} y_{k+1} + \cdots + g_n y_n = G$. Dar și $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n = G$, deci $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k y_k + g_{k+1} y_{k+1} + \cdots + g_n y_n = g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n$, de unde obținem, după reducerea termenilor egali, relația $g_k y_k + g_{k+1} y_{k+1} + \cdots + g_n y_n = g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n$, pe care o putem rescrie astfel:

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

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

$$\begin{aligned}
C(Y') - C(Y) &= c_1 y_1 + \cdots + c_{k-1} y_{k-1} + c_k x_k + c_{k+1} \alpha y_{k+1} + \cdots + c_n \alpha y_n \\
&\quad - (c_1 y_1 + \cdots + c_{k-1} y_{k-1} + c_k y_k + c_{k+1} y_{k+1} + \cdots + c_n y_n) = \\
&= c_k (x_k - y_k) + (\alpha - 1)(c_{k+1} y_{k+1} + \cdots + c_n y_n) = \\
&= \frac{c_k}{g_k} \left[g_k (x_k - y_k) + (\alpha - 1) \left(\frac{g_k}{c_k} c_{k+1} y_{k+1} + \cdots + \frac{g_k}{c_k} c_n y_n \right) \right]
\end{aligned}$$

Rescriind ultima relație, obținem:

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

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

$$\frac{g_k c_{k+1}}{c_k} \leq g_{k+1}, \dots, \frac{g_k c_n}{c_k} \leq g_n \tag{3}$$

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

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

Din relația (1) obținem că $g_k(x_k - y_k) + (\alpha - 1)(g_{k+1}y_{k+1} + \cdots + g_n y_n) = 0$, deci relația (4) devine $C(Y') - C(Y) \geq 0$, de unde rezultă faptul că $C(Y') \geq C(Y)$. 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 Python a algoritmului Greedy pentru rezolvarea variantei fracționare a problemei rucsacului:

```python
# functie folosita pentru sortarea descrescătoare a obiectelor
# în raport de câștigul unitar (cheia)
def cheieCâștigUnitar(ob):
    return ob[2] / ob[1]
# citim datele de intrare din fișierul text "rucsac.in"
fin = open("rucsac.in")
# de pe prima linie citim capacitatea G a rucsacului
G = float(fin.readline())
# fiecare dintre următoarele linii conține
# greutatea și câștigul unui obiect
obiecte = []
crt = 1
```

```
for linie in fin:
    aux = linie.split()
    # un obiect este un tuplu (ID, greutate, câștig)
    obiecte.append((crt, float(aux[0]), float(aux[1])))
    crt += 1
fin.close()

# sortăm obiectele descrescător în funcție de câștigul unitar
obiecte.sort(key=cheieCâștigUnitar, reverse=True)
# n reprezintă numărul de obiecte
n = len(obiecte)
# solutie este o listă care va conține fracțiunile încărcate
# din fiecare obiect
soluție = [0] * n
# inițial, spațiul liber din rucsac este chiar G
spațiu_liber_rucsac = G
# considerăm, pe rând, fiecare obiect
for i in range(n):
    # dacă obiectul curent încape complet în spațiul liber
    # din rucsac, atunci îl încărcăm complet
    if obiecte[i][1] <= spațiu_liber_rucsac:
        spațiu_liber_rucsac -= obiecte[i][1]
        soluție[i] = 1
    else:
        # dacă obiectul curent nu încape complet în spațiul liber
        # din rucsac, atunci calculăm fracțiunea din el necesară
        # pentru a încărca complet rucsacul și algoritmul se termină
        soluție[i] = spațiu_liber_rucsac / obiecte[i][1]
        break

# calculăm câștigul maxim
câștig = sum([soluție[i] * obiecte[i][2] for i in range(n)])

# scriem datele de ieșire în fișierul text "rucsac.out"
fout = open("rucsac.out", "w")
fout.write("Castig maxim: " + str(câștig) + "\n")
fout.write("\nObiectele incarcate:\n")
i = 0
while i < n and soluție[i] != 0:
    # trunchiem procentul încărcat din obiectul curent
    # la două zecimale
    procent = format(soluție[i]*100, '.2f')
    fout.write("Obiect "+str(obiecte[i][0])+": "+procent+"%\n")
    i = i + 1
fout.close()
```

Pentru exemplul de mai sus, fișierele text de intrare și de ieșire sunt următoarele:

rucsac.in	**rucsac.out**
53 10 30 5 40 18 36 20 10 8 16 40 30 20 20	Castig maxim: 134.0 Obiectele incarcate: Obiect 2: 100.00% Obiect 1: 100.00% Obiect 3: 100.00% Obiect 5: 100.00% Obiect 7: 60.00%

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 $\mathcal{O}(n \log_2 n)$.