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:

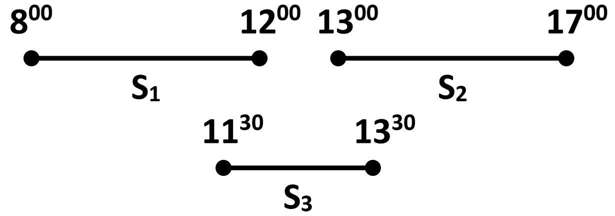
S_1: $[10^{00}, 11^{20})$
S_2: $[09^{30}, 12^{10})$
S_3: $[08^{20}, 09^{50})$
S_4: $[11^{30}, 14^{00})$
S_5: $[12^{10}, 13^{10})$
S_6: $[14^{00}, 16^{00})$
S_7: $[15^{00}, 15^{30})$

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:

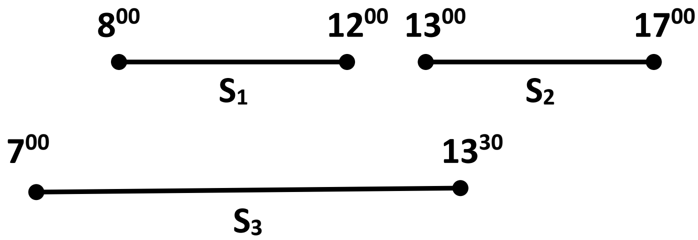
a) în ordinea crescătoare a duratelor;
b) în ordinea crescătoare a orelor de început;
c) în ordinea crescătoare a orelor de terminare.

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.

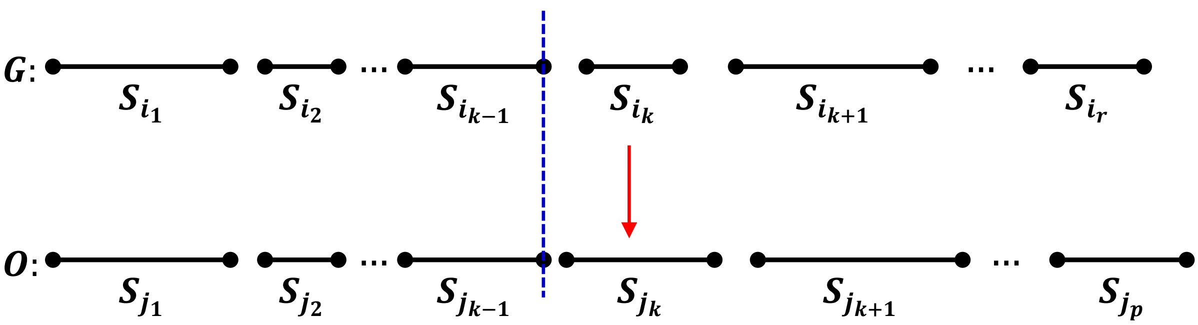
În cazul utilizării criteriului c), se observă faptul că vom obține soluțiile optime în ambele exemple prezentate mai sus:

- 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.*".

2. Planificarea unor spectacole folosind un număr minim de săli

Considerăm n spectacole $S_1, S_2, \ldots, S_n$ pentru care cunoaștem intervalele lor de desfășurare $[s_1, f_1), [s_2, f_2), \ldots, [s_n, f_n)$, toate dintr-o singură zi. Să se determine numărul minim de săli necesare astfel încât toate spectacolele să fie programate astfel încât să nu existe suprapuneri în nicio sală. Un spectacol S_j poate fi programat după spectacolul S_i dacă $s_j \geq f_i$.

De exemplu, cele 7 spectacolele de mai jos pot fi planificate folosind minim 3 săli:

spectacole.txt	programare.txt

10:00-11:20
09:30-12:10
08:20-09:50
11:30-14:00
12:10-13:10
11:15-13:15
15:00-15:30
	Numar minim de sali: 3
	
	Sala 1:
	      11:15-13:15 Spectacol 6
	
	Sala 2:
	      08:20-09:50 Spectacol 3
	      10:00-11:20 Spectacol 1
	      11:30-14:00 Spectacol 4
	
	Sala 3:
	      09:30-12:10 Spectacol 2
	      12:10-13:10 Spectacol 5
	      15:00-15:30 Spectacol 7

Deoarece dorim să găsim o rezolvare de tip Greedy a acestei probleme, vom încerca să planificăm spectacolele folosind următoarea strategie: vom încerca, pe rând, să planificăm fiecare spectacol într-una dintre sălile deja utilizate (după ultimul spectacol planificat în sala respectivă), iar dacă acest lucru nu este posibil, vom programa spectacolul respectiv într-o sală nouă (i.e., o sală neutilizată până atunci).

Spectacolele pot fi parcurse în mai multe moduri: în ordinea crescătoare a orelor de terminare, în ordinea crescătoare a duratelor sau în ordinea crescătoare a orelor de început. În continuare, vom analiza planificările pe care le vom obține pentru spectacolele din exemplul dat, utilizând una dintre modalități de parcurgere precizate anterior:

a) *în ordinea crescătoare a orelor de terminare*: vom planifica primul spectacol (S3) în Sala 1, al doilea spectacol (S1) se poate planifica tot în sala 1, al treilea spectacol (S2) nu se poate planifica tot în Sala 1, deci va fi planificat într-o sală nouă (Sala 2) ș.a.m.d.

Spectacol	Interval de desfășurare	Sala
S3	08:20 – 09:50	1
S1	10:00 – 11:20	1
S2	09:30 – 12:10	2
S5	12:10 – 13:10	1
S6	11:15 – 13:15	3
S4	11:30 – 14:00	4
S7	15:00 – 15:30	1

Evident, planificarea obținută nu este optimă, deoarece folosește 4 săli în loc de 3!

b) *în ordinea crescătoare a duratelor*: vom planifica primul spectacol (S7) în Sala 1, al doilea spectacol (S5) nu se poate planifica tot în Sala 1 (deoarece nu începe după ultimul spectacol planificat în Sala 1!), deci îl vom planifica într-o sală nouă (Sala 2), al treilea spectacol nu se poate programa niciuna dintre sălile 1 și 2, deci va fi programat într-o sală nouă (Sala 3) ș.a.m.d.

Spectacol	Interval de desfășurare	Durată	Sala
S7	15:00 – 15:30	0:30	1
S5	12:10 – 13:10	1:00	2
S1	10:00 – 11:20	1:20	3
S3	08:20 – 09:50	1:30	4
S6	11:15 – 13:15	2:00	4
S4	11:30 – 14:00	2:30	3
S2	09:30 – 12:10	2:40	5

Evident, nici această planificarea nu este optimă, deoarece folosește 5 săli în loc de 3!

c) *în ordinea crescătoare a orelor de început*: vom planifica primul spectacol (S3) în Sala 1, al doilea spectacol (S2) nu se poate planifica tot în Sala 1, deci îl vom planifica într-o sală nouă (Sala 2), al treilea spectacol (S1) se poate programa tot în Sala 1 ș.a.m.d.

Spectacol	Interval de desfășurare	Sala
S3	08:20 – 09:50	1
S2	09:30 – 12:10	2
S1	10:00 – 11:20	1
S6	11:15 – 13:15	3
S4	11:30 – 14:00	1
S5	12:10 – 13:10	2
S7	15:00 – 15:30	1

Evident, această planificarea este optimă, deoarece folosește tot 3 săli, chiar dacă spectacolele sunt distribuite în săli altfel decât în exemplul dat!

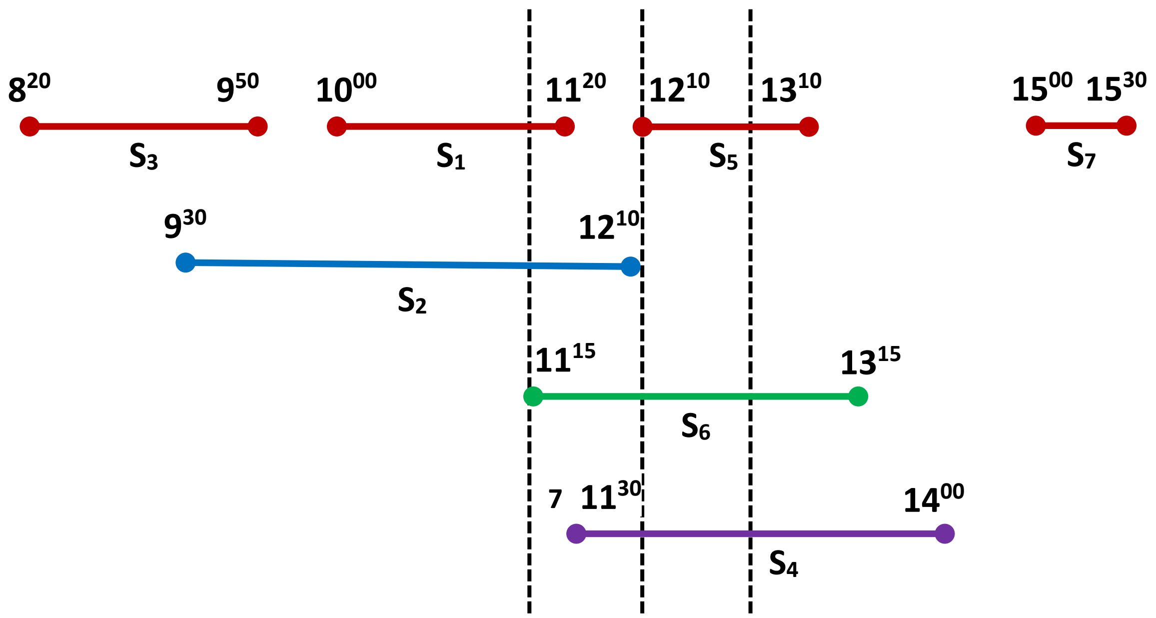
Astfel, analizând exemplele de mai sus, un algoritm Greedy posibil corect, pare a fi următorul:

- sortăm spectacolele în ordinea crescătoare a orelor de început;
- planificăm primul spectacol în prima sală;
- fiecare spectacol rămas îl planificăm fie în prima sală deja utilizată în care acest lucru este posibil (i.e., ora de început a spectacolului curent este mai mare sau egală decât ora de terminare a ultimului spectacol programat în sala respectivă), fie utilizăm o nouă sală pentru el.

În continuare, vom demonstra corectitudinea algoritmului Greedy propus mai sus, folosind următoarele observații:

a) *Algoritmul Greedy nu planifică niciodată două spectacole care se suprapun în aceeași sală*. Argumentați!

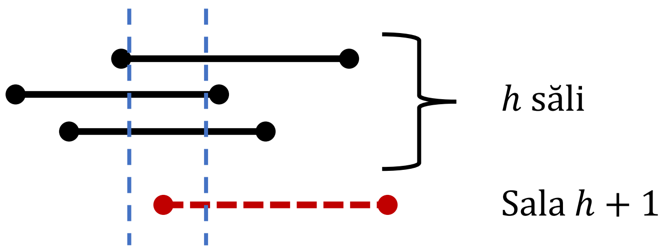
b) Definim *adâncimea h a unui șir de intervale de desfășurare ale unor spectacole* ca fiind numărul maxim de spectacole care se suprapun în orice moment posibil. De exemplu, considerând spectacolele din exemplele de mai sus, putem observa faptul că adâncimea șirului intervalelor lor de desfășurare este $h = 3$:



Se observă foarte ușor faptul că *orice planificare corectă a unor spectacole va utiliza un număr de săli cel puțin egal cu adâncimea șirului intervalelor lor de desfășurare*!

c) Pentru un șir de intervale de desfășurare ale unor spectacole având adâncimea h, planificarea realizată de algoritmul Greedy va folosi cel mult h săli.

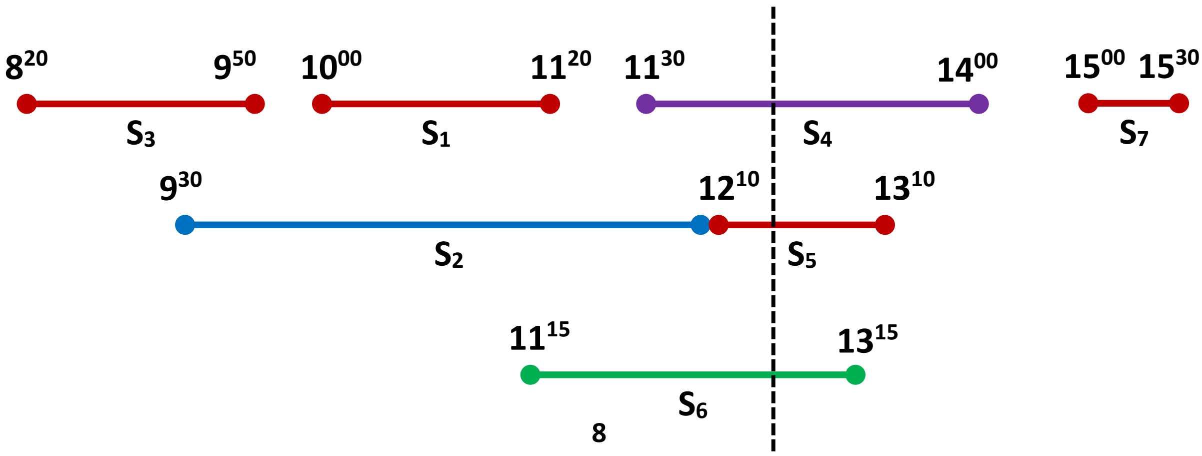
Demonstrație: Presupunem faptul că algoritmul Greedy ar folosi cel puțin $h + 1$ săli pentru a planifica un șir de spectacole având adâncimea intervalelor lor de desfășurare egală cu h. Acest lucru s-ar putea întâmpla doar dacă algoritmul Greedy ar fi utilizat deja h săli pentru a planifica spectacolele anterioare spectacolului curent, iar spectacolul curent nu ar putea fi planificat în niciuna dintre ele:



Acest lucru ar însemna faptul că spectacolul curent începe strict înaintea minimului dintre orele de terminare ale spectacolelor deja planificate, deoarece, în caz contrar, spectacolul curent ar fi fost programat într-una dintre sălile deja utilizate. Totodată, știm faptul că spectacolul curent începe după maximul dintre orele de început ale spectacolelor deja planificate, deoarece acestea sunt ordonate crescător după ora de început. Așadar, există un interval de timp (delimitat în figura de mai sus de cele două linii albastre întrerupte) în care spectacolul curent se suprapune cu câte un spectacol din fiecare dintre cele h săli deja utilizate, deci adâncimea șirului de intervale de desfășurare ale spectacolelor respective ar fi $h + 1$. Acest lucru reprezintă o contradicție cu faptul că adâncimea șirului intervalelor de desfășurare ale spectacolelor date este h, deci presupunerea făcută este falsă și, în consecință, algoritmul Greedy va utiliza cel mult h săli pentru a planifica spectacolele respective.

Din observațiile b) și c) rezultă faptul că algoritmul Greedy este optim, deoarece va utiliza exact h săli pentru a planifica spectacole având adâncimea șirului intervalelor de desfășurare egală cu h.

Pentru spectacolele din exemplul utilizat, având adâncimea șirului intervalelor de desfășurare $h = 3$, o planificare optimă (i.e., care folosește 3 săli) este următoarea:



În limbajul Python se poate implementa ușor acest algoritm, păstrând sălile într-o listă, iar fiecare sală va fi tot o listă conținând spectacolele planificate în ea. Totuși, această variantă de implementare ar avea complexitatea maximă $\mathcal{O}(n^2)$, deoarece ora de început a fiecăruia dintre cele n spectacole date ar trebui să fie comparată cu ora de terminare a ultimului spectacol planificat în fiecare dintre cele maxim n săli.

O implementare cu complexitatea optimă $\mathcal{O}(n\log_2 n)$ necesită utilizarea unei structuri de date numită *coadă cu priorități* (*priority queue*). Într-o astfel de structură de date, fiecărei valori îi este asociat un număr întreg reprezentând prioritatea sa, iar operațiile specifice unei cozi se realizează în funcție de prioritățile elementelor, ci nu în funcție de ordinea în care ele au fost inserate. Astfel, operația de inserare a unui nou element și operația de extragere a elementului cu prioritate minimă/maximă (depinde de tipul cozii cu priorități, respectiv *min-priority queue* sau *max-priority queue*) vor avea, de obicei, complexitatea $\mathcal{O}(\log_2 n)$.

În limbajul Python, clasa `PriorityQueue` implementează o coadă cu priorități în care un element trebuie să fie un tuplu de forma $(prioritate, valoare)$. Principalele metode ale acestei clase sunt:

- `get()`: furnizează elementul din coadă care are prioritatea minimă sau, dacă există mai multe elemente cu prioritate minimă, pe primul inserat;
- `put(element)`: inserează în coadă un element de forma indicată mai sus;
- `empty()`: returnează `True` în cazul în care coada nu mai conține niciun element sau `False` în caz contrar;
- `qsize()`: returnează numărul de elemente din coadă.

Așa cum deja am menționat, metodele `get` și `put` au complexitatea $\mathcal{O}(\log_2 n)$, iar metodele `empty` și `qsize` au complexitatea $\mathcal{O}(1)$.

În implementarea algoritmului Greedy prezentat, vom folosi o coadă cu priorități pentru a memora sălile utilizate, prioritatea unei săli fiind dată de ora de terminare a ultimului spectacol planificat în ea, iar spectacolele planificate într-o sală vor fi păstrate folosind o listă. Astfel, vom extrage sala cu timpul minim de terminare al ultimului spectacol planificat în ea și vom verifica dacă spectacolul curent poate fi planificat în sala respectivă sau nu (dacă spectacolul curent nu poate fi planificat în această sală, atunci el nu poate fi planificat în nicio altă sală!). În caz afirmativ, vom planifica spectacolul curent în sala respectivă și îi vom actualiza prioritatea la ora de terminare a spectacolului adăugat, altfel vom insera în coadă o sală nouă în care vom planifica spectacolul curent și prioritatea sălii va fi egală cu ora de terminare a spectacolului respectiv.

Considerând faptul că fișierele de intrare și ieșire sunt de forma prezentată în primul exemplu, o implementare în limbajul Python a algoritmului Greedy este următoarea:

```
import queue

# functie folosita pentru sortarea crescătoare a spectacolelor
# în raport de ora de început (cheia)
def cheieOraÎnceput(sp):
    return sp[1]
```

```python
# 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 crescător după orelor de început
lsp.sort(key=cheieOraÎnceput)

# sălile vor fi stocate într-o coadă cu priorități în care
# prioritatea unei săli este dată de ora de terminare a
# ultimului spectacol planificat în sala respectivă, iar
# spectacolele planificate în ea vor fi păstrate într-o listă
sali = queue.PriorityQueue()

# planificăm primul spectacol în prima sală
sali.put((lsp[0][2], list((lsp[0],))))

# parcurgem restul spectacolelor
for k in range(1, len(lsp)):
    # extragem sala cu ora minimă de terminare a ultimului
    # spectacol planificat în ea
    min_timp_final = sali.get()

    # dacă spectacolul curent lsp[k] poate fi planificat în
    # sala extrasă, atunci îl adăugăm în lista spectacolelor
    # planificate în ea și reintroducem sala în coada cu
    # priorități, dar cu prioritatea actualizată la ora de
    # terminare a spectacolului adăugat
    if lsp[k][1] >= min_timp_final[0]:
        min_timp_final[1].append(lsp[k])
        sali.put((lsp[k][2], min_timp_final[1]))
    # dacă spectacolul curent lsp[k] nu poate fi planificat în
    # sala extrasă, atunci reintroducem sala extrasă în coada
    # cu priorități fără a-i modifica prioritatea și adăugăm
    # o sală nouă în care planificăm spectacolul curent
    else:
        sali.put(min_timp_final)
        sali.put((lsp[k][2], list((lsp[k],))))

# scriem datele de ieșire în fișierul text "programare.txt"
fout = open("programare.txt", "w")
fout.write("Numar minim de sali: " + str(sali.qsize()) + "\n")
```

```
scrt = 1
while not sali.empty():
    sala = sali.get()
    fout.write("\nSala " + str(scrt) + ":\n")
    for sp in sala[1]:
        fout.write("\t"+sp[1]+"-"+sp[2]+" Spectacol "+
                   str(sp[0]) + "\n")
    scrt += 1

fout.close()
```

Observați faptul că în fișierul de ieșire sălile vor fi scrise în ordinea priorităților, ci nu în ordinea în care au fost inserate!

3. 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 $X = (x_1, x_2, \ldots, x_n)$ 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 $X = (1, \ldots, 1)$, care este evident optimă, deci vom considera faptul că $g_1 + g_2 + \cdots + g_n > G$.

Fie $X = (1, \ldots, 1, x_j, 0, \ldots, 0)$, 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 $Y = (y_1, \ldots, y_{k-1}, y_k, y_{k+1}, \ldots, y_n)$ 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 $Y' = (y_1, \ldots, y_{k-1}, x_k, \alpha y_{k+1}, \ldots, \alpha y_n)$, 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 $y_{k+1}, \ldots, y_n$ 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:

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

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