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 submulţimi a unei mulţimi date pentru care se minimizează sau se maximizează valoarea unei funcţii obiectiv. Formal, o problemă de optimizare poate fi enunţată astfel: "*Fie A o mulţime nevidă şi $f: \mathcal{P}(A) \to \mathbb{R}$ o funcţie obiectiv asociată mulţimii A, unde prin $\mathcal{P}(A)$ am notat mulţimea tuturor submulţimilor mulţimii A. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei f să fie minimă/maximă pe S (i.e., pentru orice altă submulţime $T \subseteq A, T \neq S$, valoarea funcţiei obiectiv f va fi cel puţin /cel mult egală cu valoarea funcţiei obiectiv f pe submulţimea S).*"

O problemă foarte simplă de optimizare este următoarea: "*Fie A o mulţime nevidă de numere întregi. Să se determine o submulţime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*". Se observă faptul că funcţia obiectiv nu este dată în formă matematică şi nu se precizează explicit faptul că suma elementelor submulţimii S trebuie să fie maximă *în raport cu suma oricărei alte submulţimi*, acest lucru subînţelegându-se. Formal, problema poate fi enunţată astfel: "*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ şi $f: \mathcal{P}(A) \to \mathbb{R}$, $f(S) = \sum_{x \in S} x$. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei 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 forţei-brute*, astfel: se generează, pe rând, toate submulţimile S ale mulţimii A şi pentru fiecare dintre ele se calculează $f(S)$, iar dacă valoarea obţinută este mai mică/mai mare decât minimul/maximul obţinut până în acel moment, atunci se actualizează minimul/maximul şi se reţine submulţimea S. Deşi aceasta rezolvare este corectă, ea are o complexitate exponenţială, respectiv $\mathcal{O}(2^{|A|})$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulţimea S, la fiecare pas, cel mai bun element disponibil din mulţimea A din punct de vedere al optimizării funcţiei obiectiv. Practic, metoda Greedy încearcă să găsească optimul global al funcţiei obiectiv combinând optimele sale locale. Totuşi, prin combinarea unor optime locale nu se obţine întotdeauna un optim global! De exemplu, să considerăm cel mai scurt drum posibil dintre Bucureşti şi Arad (un optim local), precum şi cel mai scurt drum posibil dintre Arad şi Ploieşti (alt optim local). Combinând cele două optime locale nu vom obţine un optim global, deoarece, evident, cel mai scurt drum de la Bucureşti la Ploieşti nu trece prin Arad! Din acest motiv, aplicare tehnicii de programare Greedy pentru rezolvarea unei probleme trebuie să fie însoţită de o demonstraţie a corectitudinii (optimalităţii) criteriului de selecţie pe care trebuie să-l îndeplinească un element al mulţimii A pentru a fi adăugat în soluţia 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$, 7$ şi 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$ şi 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$, 7$ şi 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. Totuşi, 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 submulţimi S cu sumă maximă, observăm faptul că aceasta trebuie să conţină toate elementele pozitive din mulţimea A, deci criteriul de selecţie este ca elementul curent din A să fie pozitiv (demonstraţia optimalităţii este banală). Dacă mulţimea A nu conţine niciun număr pozitiv, care va fi soluția problemei?

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

- de k ori se selectează maximul din mulţimea A şi 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 şi $\mathcal{O}(n^2)$ pentru valori ale lui k apropiate de n;
- sortăm crescător elementele mulţimii A şi 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 aşteptare se obţine rearanjând persoanele la coadă în ordinea crescătoare a timpilor de servire:

Persoana	**Timpul de servire** (t_i)	**Timp de aşteptare** (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 aşteptare (*M*):		$\frac{3+6+12+18+25+35}{6} = \frac{99}{6} = 16.5$

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

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumerota persoanele $p_1, p_2, \dots, p_i, \dots, p_j, \dots, p_n$ în ordinea crescătoare a timpilor de servire, astfel încât vom avea $t_1 \le t_2 \le \cdots \le t_i \le \cdots \le t_j \le \cdots \le t_n$. De asemenea, vom presupune faptul că timpii individuali de servire $t_1, t_2, \dots, t_n$ nu sunt toţi 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 $p_1, p_2, \dots, p_j, \dots, p_i, \dots p_n$ diferită de cea iniţială, în care $t_j > t_i$ (practic, am interschimbat persoanele p_i şi p_j din varianta iniţială, adică persoana p_j se află acum pe poziţia i în coadă, iar persoana p_i se află acum pe poziţia j, unde $i < j$).

În cazul primei modalităţi 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 modalităţi 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.

```
# 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:

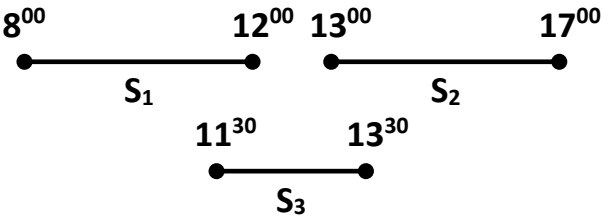
$S_1: [10^{00}, 11^{20})$
$S_2: [09^{30}, 12^{10})$
$S_3: [08^{20}, 09^{50})$
$S_4: [11^{30}, 14^{00})$
$S_5: [12^{10}, 13^{10})$
$S_6: [14^{00}, 16^{00})$
$S_7: [15^{00}, 15^{30})$

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:

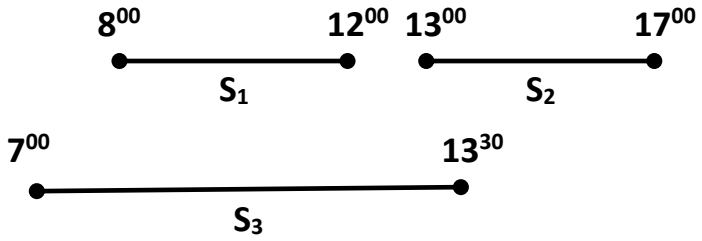
a) în ordinea crescătoare a duratelor;
b) în ordinea crescătoare a orelor de început;
c) în ordinea crescătoare a orelor de terminare.

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



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

De asemenea, în cazul utilizării criteriului b), se observă uşor faptul că nu vom obţine întotdeauna o soluţie 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 şi nici spectacolul S_2, deoarece ambele se suprapun cu el, deci vom obţine o planificare formată doar din S_3. Evident, planificarea optimă este S_1 şi S_2.

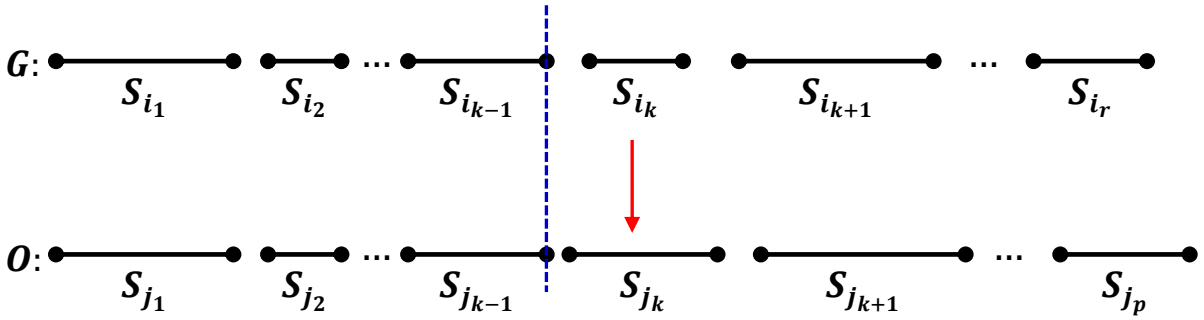
În cazul utilizării criteriului c), se observă faptul că vom obţine soluţiile optime în ambele exemple prezentate mai sus:

- 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 obţine planificarea optimă formată din S_1 şi S_2;
- în al doilea exemplu, vom proceda la fel şi vom obţine planificarea optimă formată din S_1 şi S_2.

Practic, criteriul c) este o combinaţie a criteriilor a) şi b), deoarece un spectacol care durează puţin şi începe devreme se va termina devreme!

Pentru a demonstra optimalitatea criteriului c) de selecție, vom utiliza o demonstrație de tipul *exchange argument*: vom considera o soluție optimă furnizată de un algoritm oarecare (nu contează metoda utilizată!), diferită de soluția furnizată de algoritmul de tip Greedy, și vom demonstra faptul că aceasta poate fi transformată, element cu element, în soluția furnizată de algoritmul de tip Greedy. Astfel, vom demonstra faptul că și soluția furnizată de algoritmul de tip Greedy este tot optimă!

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



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

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

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

Pentru a încheia demonstrația, trebuie să mai demonstrăm faptul că ambele soluții conțin același număr de spectacole, respectiv $r = p$. Presupunem prin absurd faptul că $r \neq p$. Deoarece soluția O este optimă, rezultă faptul că $p > r$ (altfel, dacă $p < r$, ar însemna că soluția optimă O conține mai puține spectacole decât soluția Greedy G, ceea ce i-ar contrazice optimalitatea), deci există cel puțin un spectacol $S_{j_{r+1}}$ în soluția optimă

O care nu a fost selectat în soluția Greedy G. Acest lucru este imposibil, deoarece am demonstrat anterior faptul că orice spectacol S_{j_k} din soluția optimă se termină după spectacolul S_{i_k} aflat pe aceeași poziție în soluția Greedy (adică $f_{j_k} \geq f_{i_k}$), deci am obține relația $f_{i_r} \leq f_{j_r} \leq s_{j_{r+1}}$, ceea ce ar însemna că spectacolul $S_{j_{r+1}}$ ar fi trebuit să fie selectat și în soluția Greedy G! În concluzie, presupunerea că $r \neq p$ este falsă, deci $r = p$.

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

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

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

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

În continuare, vom prezenta implementarea algoritmului în limbajul 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, fişierele text de intrare şi de ieşire 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 activităţi pentru care se cunosc intervalele orare de desfăşurare şi care partajează o resursă comună. Ştiind faptul că activităţile trebuie efectuate sub excludere reciprocă* (*respectiv, la un moment dat resursa comună poate fi alocată unei singure activităţi*)*, să se determine o modalitate de planificare a unui număr maxim de activităţi care nu se suprapun.*".

4. Planificarea unor spectacole folosind un număr minim de săli

Considerăm n spectacole $S_1, S_2, \dots, S_n$ pentru care cunoaştem intervalele lor de desfăşurare $[s_1, f_1), [s_2, f_2), \dots, [s_n, f_n)$, 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 şi 2, deci va fi programat într-o sală nouă (Sala 3) ş.a.m.d.

Spectacol	Interval de desfăşurare	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 foloseşte 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 ş.a.m.d.

Spectacol	Interval de desfăşurare	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 foloseşte tot 3 săli, chiar dacă spectacolele sunt distribuite în săli altfel decât în exemplul dat!

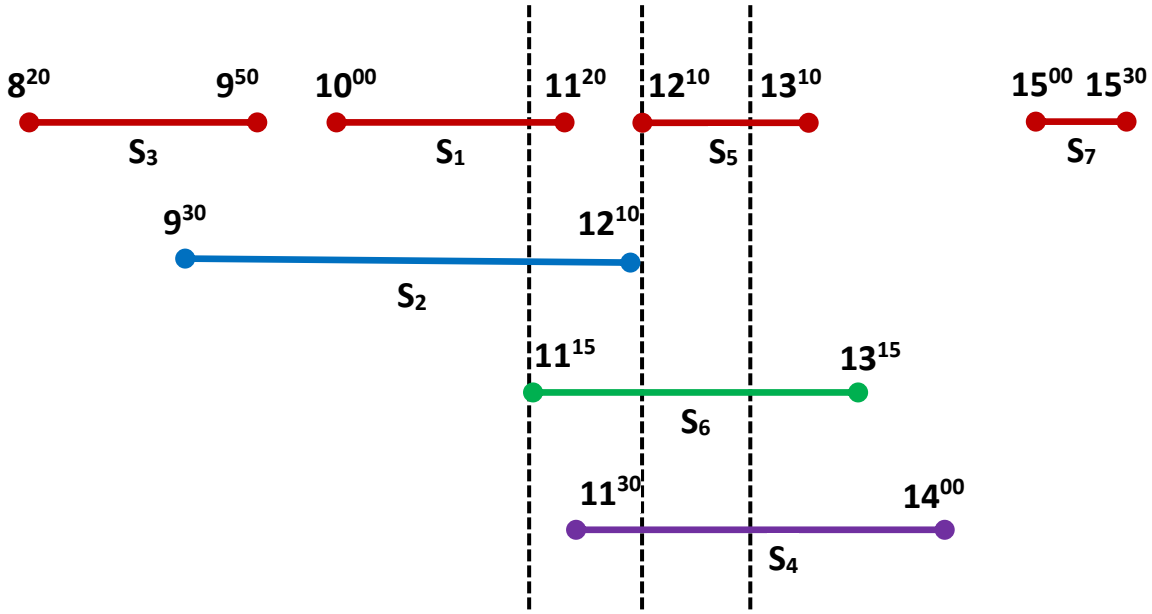
Astfel, analizând exemplele de mai sus, un algoritm Greedy posibil corect, pare a fi următorul:

- 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!

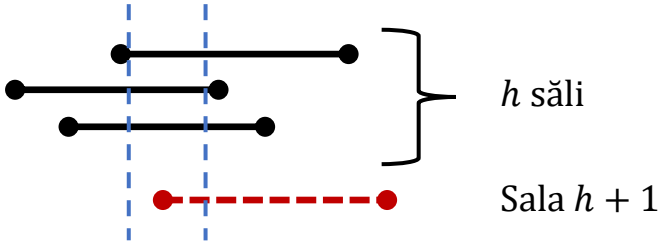
b) Definim *adâncimea h a unui șir de intervale de desfășurare ale unor spectacole* ca fiind numărul maxim de spectacole care se suprapun în orice moment posibil. De exemplu, considerând spectacolele din exemplele de mai sus, putem observa faptul că adâncimea șirului intervalelor lor de desfășurare este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



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.

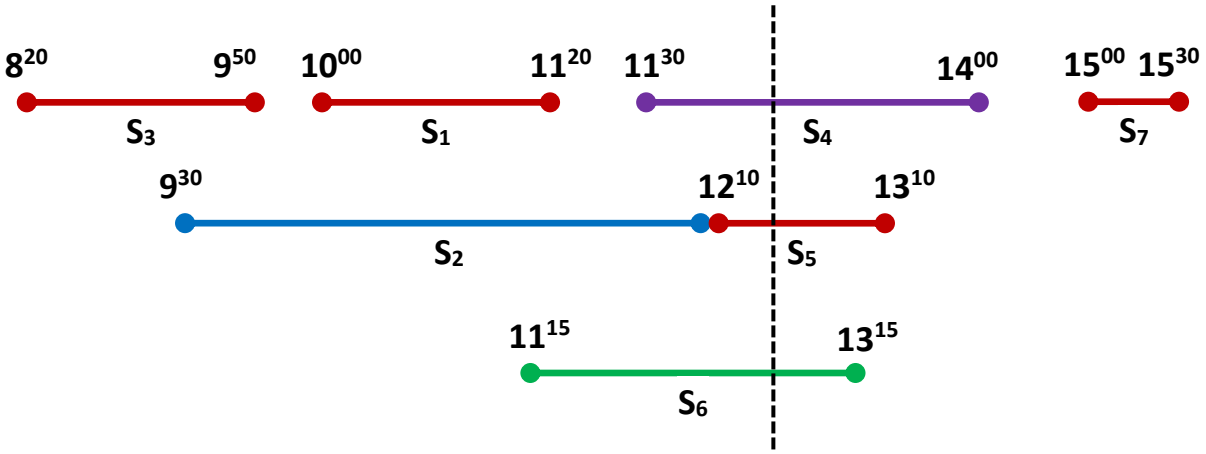
Demonstrație: Presupunem faptul că algoritmul Greedy ar folosi cel puțin $h + 1$ săli pentru a planifica un șir de spectacole având adâncimea intervalelor lor de desfășurare egală cu h. Acest lucru s-ar putea întâmpla doar dacă algoritmul Greedy ar fi utilizat deja h săli pentru a planifica spectacolele 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, ştim faptul că spectacolul curent începe după maximul dintre orele de început ale spectacolelor deja planificate, deoarece acestea sunt ordonate crescător după ora de început. Aşadar, 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 şirului de intervale de desfăşurare ale spectacolelor respective ar fi $h + 1$. Acest lucru reprezintă o contradicţie cu faptul că adâncimea şirului intervalelor de desfăşurare ale spectacolelor date este h, deci presupunerea făcută este falsă şi, în consecinţă, algoritmul Greedy va utiliza cel mult h săli pentru a planifica spectacolele respective.

Din observaţiile b) şi c) rezultă faptul că algoritmul Greedy este optim, deoarece va utiliza exact h săli pentru a planifica spectacole având adâncimea şirului intervalelor de desfăşurare egală cu h.

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



În limbajul Python se poate implementa uşor acest algoritm, păstrând sălile într-o listă, iar fiecare sală va fi tot o listă conţinând spectacolele planificate în ea. Totuşi, 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 priorităţi* (*priority queue*). Într-o astfel de structură de date, fiecărei valori îi este asociat un număr întreg reprezentând prioritatea sa, iar operaţiile specifice unei cozi se realizează în funcţie de priorităţile elementelor, ci nu în funcţie de ordinea în care ele au fost inserate. Astfel, operaţia de inserare a unui nou element şi operaţia de extragere a elementului cu prioritate minimă/maximă (depinde de tipul cozii cu priorităţi, 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 priorităţi 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 conţine niciun element sau `False` în caz contrar;
- `qsize()`: returnează numărul de elemente din coadă.

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

În implementarea algoritmului Greedy prezentat, vom folosi o coadă cu priorităţi 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 şi 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 şi 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 şi prioritatea sălii va fi egală cu ora de terminare a spectacolului respectiv.

Considerând faptul că fişierele de intrare şi ieşire 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]


# 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!

5. Problema rucsacului (varianta continuă/fracţionară)

Considerăm un rucsac având capacitatea maximă G şi n obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaştem greutăţile lor $g_1, g_2, \dots, g_n$ şi câştigurile $c_1, c_2, \dots, c_n$ obţinute prin încărcarea lor completă în rucsac. Ştiind faptul că orice obiect poate fi încărcat şi fracţionat (doar o parte din el), să se determine o modalitate de încărcare a rucsacului astfel încât câştigul total obţinut să fie maxim. Dacă un obiect este încărcat fracţionat, atunci vom obţine un câştig direct proporţional cu fracţiunea încărcată din el (de exemplu, dacă vom încărca doar o treime dintr-un obiect, atunci vom obţine un câştig egal cu o treime din câştigul integral asociat obiectului respectiv).

În afara variantei continue/fracţionare a problemei rucsacului, mai există şi 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 uşor faptul că varianta fracţionară a problemei rucsacului are întotdeauna soluţie (evident, dacă $G > 0$ şi $n \geq 1$), chiar şi în cazul în care cel mai mic obiect are o greutate strict mai mare decât capacitatea G a rucsacului (deoarece putem să încărcăm şi fracţiuni dintr-un obiect), în timp ce varianta discretă nu ar avea soluţie în acest caz.

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

a) în ordinea descrescătoare a câştigurilor 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
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 $O_1, O_2, \ldots, O_n$ 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 soluție a problemei va fi reprezentată sub forma unui tuplu $X = (x_1, x_2, \dots, x_n)$, 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 $X = (1, \dots, 1, x_j, 0, \dots, 0)$ cu n elemente, unde $x_j \in [0,1)$;
- în toate formulele vom considera implicit indicii ca fiind 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:

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

Comparăm acum câştigurile asociate soluţiilor Y şi Y', calculând diferenţa dintre ele:

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

Rescriind ultima relaţie, obţinem:

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

Dar $\frac{g_k}{c_k} \leq \frac{g_i}{c_i}$ pentru orice $i > k$ (deoarece obiectele sunt sortate descrescător în funcţie de câştigurile lor unitare, deci $\frac{c_k}{g_k} \geq \frac{c_i}{g_i}$ pentru orice $i > k$), de unde rezultă că $\frac{g_kc_i}{c_k} \leq g_i$ pentru orice $i > k$, deci obţinem relaţiile:

$$\frac{g_kc_{k+1}}{c_k} \leq g_{k+1}, \ldots, \frac{g_kc_n}{c_k} \leq g_n \quad (3)$$

Aplicând relaţiile (3) în relaţia (2), obţinem:

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

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

a) $C(Y') > C(Y)$, ceea ce contrazice optimalitatea soluţiei Y, aşadar presupunerea că ar exista o soluţie optimă Y diferită de soluţia X furnizată de algoritmul Greedy este falsă, ceea ce înseamnă că $X = Y$, deci şi soluţia 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 paşi, vom obţine o contradicţie de tipul celei de la punctul a).

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

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

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

TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

1. Prezentare generală

Tehnica de programare *Divide et Impera* constă în descompunerea repetată a unei probleme în două sau mai multe subprobleme de acelaşi tip până când se obţin probleme direct rezolvabile (etapa *Divide*), după care, în sens invers, soluţia fiecărei probleme se obţine combinând soluţiile subproblemelor în care a fost descompusă (etapa *Impera*).

Se poate observa cu uşurinţă faptul că tehnica *Divide et Impera* are în mod nativ un caracter recursiv, însă există şi cazuri în care ea este implementată iterativ.

Evident, pentru ca o problemă să poată fi rezolvată folosind tehnica Divide et Impera, ea trebuie să îndeplinească următoarele două condiţii:

1. condiţia *Divide*: problema poate fi descompusă în două (sau mai multe) subprobleme de acelaşi tip;
2. condiţia *Impera*: soluţia unei probleme se poate obţine combinând soluţiile subproblemelor în care ea a fost descompusă.

De obicei, subproblemele în care se descompune o problemă au dimensiunile datelor de intrare aproximativ egale sau, altfel spus, aproximativ egale cu jumătate din dimensiunea datelor de intrare ale problemei respective.

De exemplu, folosind tehnica Divide et Impera, putem calcula suma elementelor unei liste t formată din n numere întregi, astfel:

1. împărţim lista t, în mod repetat, în două jumătăţi până când obţinem liste cu un singur element (caz în care suma se poate calcula direct, fiind chiar elementul respectiv);
2. în sens invers, calculăm suma elementelor dintr-o listă adunând sumele elementelor celor două sub-liste în care ea a fost descompusă.

Se observă imediat faptul că problema verifică ambele condiţii menţionate mai sus!

Pentru a putea manipula uşor cele două sub-liste care se obţin în momentul împărţirii listei t în două jumătăţi, vom considera faptul că lista curentă este secvenţa cuprinsă între doi indici st şi dr, unde $st \leq dr$. Astfel, indicele mij al mijlocului listei curente este aproximativ egal cu $[(st + dr)/2]$, iar cele două sub-liste în care va fi descompus lista curentă sunt secvenţele cuprinse între indicii st şi mij, respectiv $mij + 1$ şi dr.

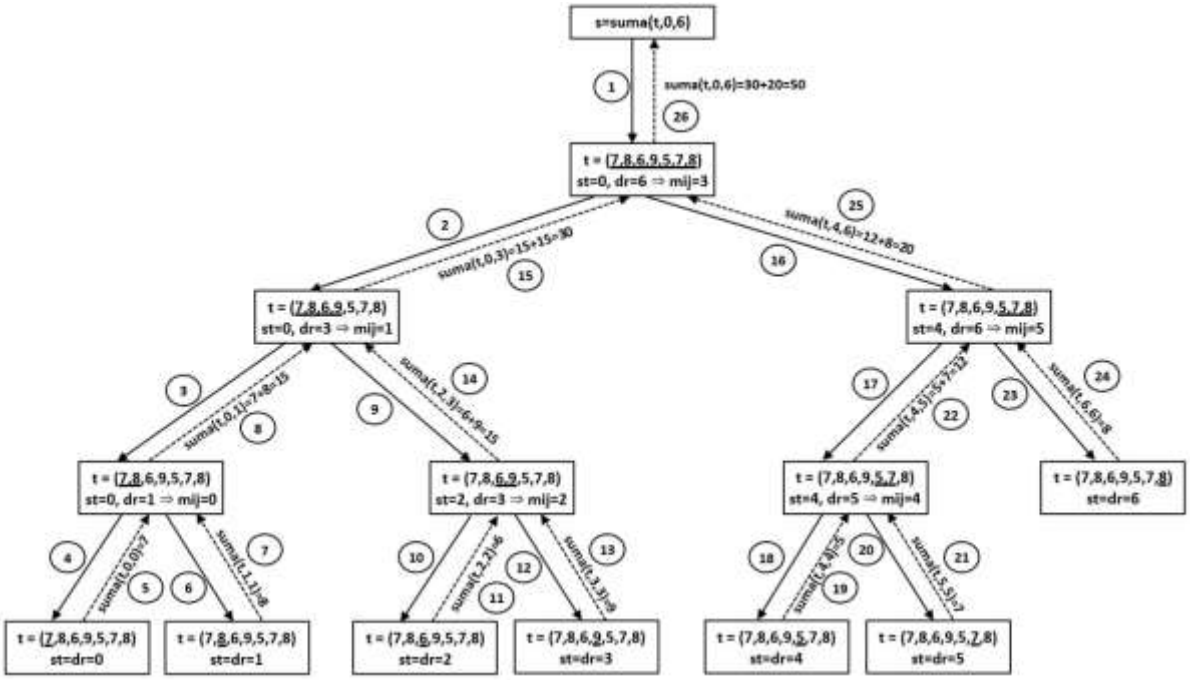
Considerând $suma(t, st, dr)$ o funcţie care calculează suma secvenţei $t[st], t[st + 1], \dots, t[dr]$, putem să o definim în manieră Divide et Impera astfel:

$$suma(t, st, dr) = \begin{cases} t[st], & \text{dacă } st = dr \quad (1) \\ suma(t, st, mij) + suma(t, mij + 1, dr), & \text{dacă } st < dr \quad (2) \end{cases}$$

unde $mij = [(st + dr)/2]$.

Pentru o listă t cu n elemente, suma s a tuturor elementelor sale se va obţine în urma apelului $s = suma(t, 0, n - 1)$.

De exemplu, considerând lista $t = (7, 8, 6, 9, 5, 7, 8)$ cu $n = 7$ elemente, pentru a calcula suma elementelor sale, se vor efectua următoarele apeluri recursive:



În figura de mai sus, săgeţile "pline" reprezintă apelurile recursive, efectuate folosind relaţia (2) de mai sus, în timp ce săgeţile "întrerupte" reprezintă revenirile din apelurile recursive, care au loc în momentul în care se ajunge la o subproblemă direct rezolvabilă folosind relaţia (1) de mai sus. Ordinea în care se execută apelurile şi revenirile este indicată prin numerele scrise în cercuri. Numerele subliniate din lista *t* reprezintă secvenţa curentă (i.e., care se prelucrează în apelul respectiv).

2. Forma generală a unui algoritm de tip Divide et Impera

Plecând chiar de la exemplul se mai sus, se poate deduce foarte uşor forma generală a unui algoritm de tip Divide et Impera aplicat asupra unei liste *t*:

```
# functie care furnizeaza solutia unei probleme combinand solutiile
# subproblemelor in care ea a fost descompusa
def combinare(sol_st, sol_dr):
    pass

def divimp(t, st, dr):
   # daca subproblema curentă este direct rezolvabilă
   if dr-st <= k:          # k este, de obicei, 0 sau 1
      return solutie_problema_directa

   # etapa Divide
   mij = (st + dr) // 2
   sol_st = divimp(t, st, mij)
   sol_dr = divimp(t, mij+1, dr)

   # etapa Impera
   return combinare(sol_st, sol_dr)
```

Vizavi de algoritmul general prezentat mai sus trebuie făcute câteva precizări:

- există algoritmi Divide et Impera în care nu sunt utilizate liste (de exemplu, calculul lui a^n - https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/), dar aceştia sunt mult mai rari decât cei în care sunt utilizate liste;
- de obicei, o subproblemă este direct rezolvabilă dacă secvenţa curentă din lista `t` este vidă (i.e., `st>dr`) sau are un singur element (i.e., `st==dr`);
- variabila `mij` va conţine indicele mijlocului secvenţei `t[st], t[st+1], …, t[dr]`;
- variabilele `sol_st` şi `sol_dr` vor conţine soluţiile celor două subprobleme în care se descompune problema curentă, iar `combinare(sol_st, sol_dr)` este o funcţie care determină soluţia problemei curente combinând soluţiile subproblemelor în care aceasta a fost descompusă (în unele cazuri, nu este necesară o funcţie, ci se poate folosi o simplă expresie);
- dacă funcţia `divimp` nu furnizează nicio valoare, atunci vor lipsi variabilele `sol_st` şi `sol_dr`, precum şi cele două instrucţiuni `return`.

Aplicând algoritmul general pentru a calcula suma elementelor dintr-o listă de numere întregi, vom obţine următoarea implementare în limbajul Python:

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă,
    # respectiv lista curenta are un singur element
    if dr == st:
        return t[st]

    # etapa Divide
    mij = (st + dr) // 2
    sol_st = suma(t, st, mij)
    sol_dr = suma(t, mij + 1, dr)

    # etapa Impera
    return sol_st + sol_dr
```

Aşa cum am menţionat în observaţiile anterioare, nu a mai fost necesară implementarea unei funcţii `combinare`, ci a fost suficientă utilizarea unei simple expresii.

3. Determinarea complexităţii unui algoritm de tip Divide et Impera

Deoarece algoritmii de tip Divide et Impera sunt implementaţi, de obicei, folosind funcţii recursive, determinarea complexităţii computaţionale a unui astfel de algoritm este mai complicată decât în cazul algoritmilor iterativi.

Primul pas în determinarea complexităţii unei algoritm Divide et Impera îl constituie determinarea unei relaţii de recurenţă care să exprime complexitatea $T(n)$ a rezolvării unei probleme având dimensiunea datelor de intrare egală cu n în raport de timpul necesar rezolvării subproblemelor în care aceasta este descompusă şi de complexitatea operaţiei de combinare a soluţiilor lor pentru a obţine soluţia problemei iniţiale. Presupunând faptul că orice problemă se descompune în a subprobleme, fiecare având dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{b}$, iar împărţirea problemei curente în subprobleme şi combinarea soluţiilor subproblemelor pentru a obţine soluţia sa se

realizează folosind un algoritm cu complexitatea $f(n)$, se obține foarte ușor forma generală a relației de recurență căutate:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3}$$

unde $a \geq 1$, $b > 1$ și $f(n)$ este o funcție asimptotic pozitivă (i.e., există $n_0 \in \mathbb{N}$ astfel încât pentru orice $n \geq n_0$ avem $f(n) \geq 0$). De asemenea, vom presupune faptul că $T(1) \in \mathcal{O}(1)$.

Reluăm algoritmul Divide et Impera prezentat mai sus pentru calculul sumei elementelor unei liste, cu scopul de a-i determina complexitatea:

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă,
    # respectiv lista curenta are un singur element
    if dr == st:
        return t[st]

    # etapa Divide
    mij = (st + dr) // 2
    sol_st = suma(t, st, mij)
    sol_dr = suma(t, mij + 1, dr)

    # etapa Impera
    return sol_st + sol_dr
```

Analizând fiecare etapa a algoritmului, obținem următoarea relație de recurență pentru $T(n)$:

$$T(n) = \begin{cases} 1 + T\left(\left[\frac{n}{2}\right]\right) + T\left(\left[\frac{n}{2}\right]\right) + 1, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases} = \begin{cases} 2T\left(\left[\frac{n}{2}\right]\right) + 2, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

În concluzie, o problemă având dimensiunea datelor de intrare egală cu n se descompune în $a = 2$ subprobleme, fiecare având dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{2}$ (deci $b = 2$), iar împărțirea problemei curente în subprobleme și combinarea soluțiilor subproblemelor pentru a obține soluția sa se realizează folosind un algoritm cu complexitatea $\mathcal{O}(1)$, deci relația de recurență este următoarea:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \tag{4}$$

Al doilea pas în determinarea complexității unei algoritm Divide et Impera îl constituie rezolvarea relației de recurență de mai sus, utilizând diverse metode matematice, pentru a determina expresia analitică a lui $T(n)$. În continuare, vom prezenta două dintre cele mai utilizate metode, simplificate, respectiv *iterarea directă a relației de recurență* și *teorema master*.

În cazul primei metode, bazată pe *iterarea directă a relației de recurență*, vom presupune faptul că n este o putere a lui b, după care vom itera relația de recurență până

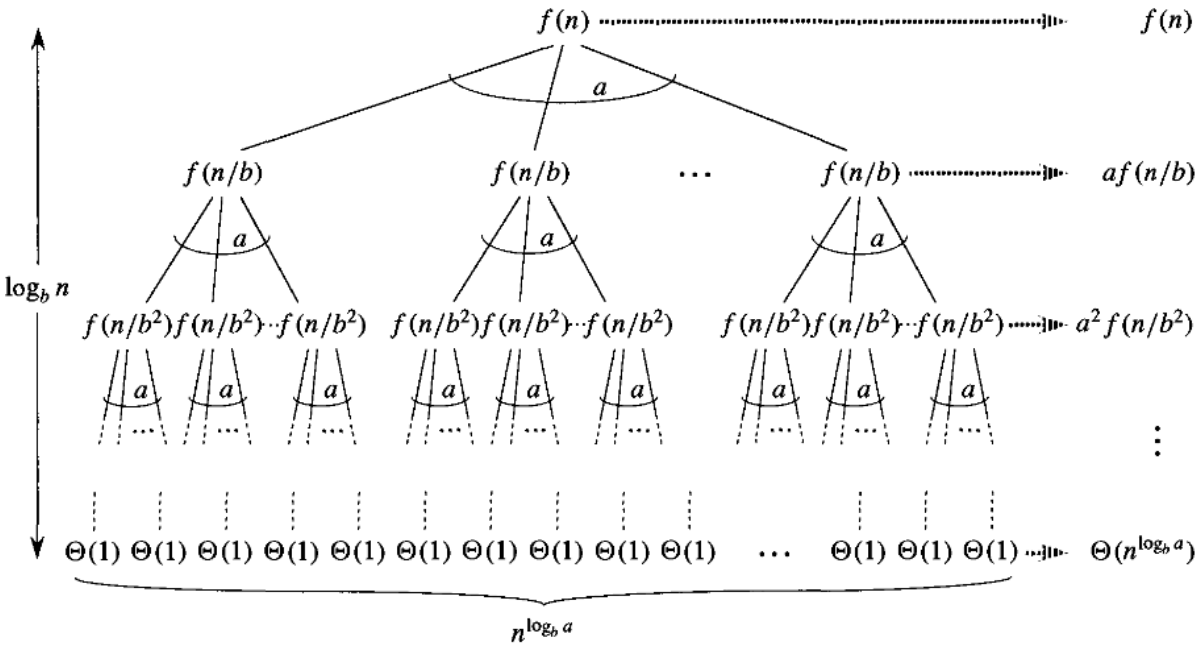
când vom ajunge la $T(1)$ sau $T(0)$, care sunt ambele egale cu 1 fiind complexităţile unor probleme direct rezolvabile.

De exemplu, pentru a rezolva relaţia de recurenţă (4), vom presupune că $n = 2^k$ şi apoi o vom itera, astfel:

$$\begin{aligned} T(n) = T(2^k) &= 2T(2^{k-1}) + 1 = 2[2T(2^{k-2}) + 1] + 1 = 2^2T(2^{k-2}) + 2 + 1 = \\ &= 2^2[2T(2^{k-3}) + 1] + 2 + 1 = 2^3T(2^{k-3}) + 2^2 + 2 + 1 = \cdots = \\ &= 2^kT(2^0) + 2^{k-1} + \cdots + 2 + 1 = 2^k + 2^{k-1} + \cdots + 2 + 1 = 2^{k+1} - 1 = \\ &= 2 \cdot 2^k - 1 = 2n - 1 \end{aligned}$$

În concluzie, am obţinut faptul că $T(n) = 2n - 1$, deci complexitatea algoritmului Divide et Impera pentru calculul sumei elementelor unei liste formate din n numere întregi este $\mathcal{O}(2n - 1) \approx \mathcal{O}(n)$.

A doua metodă constă în utilizarea *teoremei master* pentru a afla direct soluţia analitică a unei relaţii de recurenţă de tipul (3): $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Arborele de recursie asociat unei relaţii de recurenţă de acest tip este următorul (sursa imaginii: https://en.wikipedia.org/wiki/Introduction_to_Algorithms):



Se observă faptul că arborele este unul perfect (i.e., orice nod interior are exact 2 fii şi frunzele se află toate pe acelaşi nivel) cu înălţimea $h = \log_b n$, deci pe fiecare nivel, mai puţin pe ultimul, se găsesc $a^i f\left(\frac{n}{b^i}\right)$ noduri interne, iar pe ultimul nivel se găsesc $a^h = a^{\log_b n} = n^{\log_b a}$ frunze. Evident, complexitatea totală $T(n)$ se obţine însumând complexităţile asociate tuturor nodurilor:

$$T(n) = \underbrace{\sum_{i=0}^{\log_b n - 1} \left[a^i \cdot f\left(\frac{n}{b^i}\right)\right]}_{\substack{\text{timpul necesar pentru} \\ \text{divizarea problemei} \\ \text{şi reconstituirea soluţiei}}} + \underbrace{\mathcal{O}\left(n^{\log_b a}\right)}_{\substack{\text{timpul necesar pentru} \\ \text{rezolvarea subproblemelor} \\ \text{directe}}}$$

Pentru anumite forme particulare ale funcției f, se poate calcula suma din expresia lui $T(n)$ și, implicit, se poate determina forma sa analitică.

Teorema master: Fie o relație de recurență de forma (3) și presupunem faptul că $f \in \mathcal{O}(n^p)$. Atunci:

a) dacă $p < \log_b a$, atunci $T(n) \in \mathcal{O}(n^{\log_b a})$;
b) dacă $p = \log_b a$, atunci $T(n) \in \mathcal{O}(n^p \log_2 n)$;
c) dacă $p > \log_b a$ și $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare, atunci $T(n) \in \mathcal{O}(f(n))$.

Exemple:

1. Pentru a rezolva relația de recurență (4), observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(1) = \mathcal{O}(n^0) \Rightarrow p = 0 < \log_b a = \log_2 2 = 1 \xrightarrow{\text{Cazul } a)} T(n) \in \mathcal{O}(n)$.

2. Pentru a rezolva relația de recurență $T(n) = 2T\left(\frac{n}{2}\right) + n$, observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(n) \Rightarrow p = 1 = \log_b a = \log_2 2 = 1 \xrightarrow{\text{Cazul } b)} T(n) \in \mathcal{O}(n \log_2 n)$.

3. Pentru a rezolva relația de recurență $T(n) = 2T\left(\frac{n}{2}\right) + n^2$, observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(n^2) \Rightarrow p = 2 > \log_b a = \log_2 2 = 1$. În acest caz, trebuie să verificăm și faptul că $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare $\Leftrightarrow \exists c < 1$ astfel încât $2\frac{n^2}{4} \leq cn^2 \Leftrightarrow \exists c < 1$ astfel încât $\frac{n^2}{2} \leq cn^2 \Leftrightarrow \exists c < 1$ astfel încât $n^2 \leq 2cn^2 \Leftrightarrow \exists c < 1$ astfel încât $1 \leq 2c$, ceea ce este adevărat, de exemplu pentru $c = \frac{1}{2}$ sau, în general, pentru orice $c \in \left[\frac{1}{2}, 1\right)$. Astfel, obținem că $T(n) \in \mathcal{O}(n^2)$.

Varianta prezentată mai sus a teoremei master este una simplificată, dar care acoperă majoritatea cazurilor întâlnite în practică. O variantă extinsă a acestei teoreme este prezentată aici: https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms). Totuși, nici varianta extinsă nu acoperă toate cazurile posibile. Mai mult, teorema de master nu poate fi utilizată dacă subproblemele nu au dimensiuni aproximativ egale, fiind necesară utilizarea teoremei Akra-Bazzi (https://en.wikipedia.org/wiki/Akra-Bazzi_method).

Observație importantă: În general, algoritmii de tip Divide et Impera au complexități mici, de tipul $\mathcal{O}(\log_2 n), \mathcal{O}(n)$ sau $\mathcal{O}(n \log_2 n)$, care se obțin datorită faptului că o problemă este împărțită în două subprobleme de același tip cu dimensiunea datelor de intrare înjumătățită față de problema inițială și, mai mult, subproblemele nu se suprapun! Dacă aceste condiții nu sunt îndeplinite simultan, atunci complexitatea algoritmul poate să devină foarte mare, de ordin exponențial! De exemplu, o implementare recursivă, de tip Divide et Impera, care să calculeze termenul de rang n al șirului lui Fibonacci ($F_n = F_{n-1} + F_{n-2}$ și $F_0 = 0, F_1 = 1$) nu respectă condițiile precizate anterior (dimensiunile subproblemelor nu sunt aproximativ jumătate din dimensiunea unei probleme și subproblemele se suprapun, respectiv mulți termeni vor fi calculați de mai multe ori), ceea ce va conduce la o complexitate exponențială! Astfel, relația de recurență pentru

complexitatea algoritmului este $T(n) = 1 + T(n-1) + T(n-2)$. Cum $T(n-1) > T(n-2)$, obţinem că $2T(n-2) < T(n) < 2T(n-1)$. Iterând dubla inegalitate, obţinem $2^{\frac{n}{2}} < T(n) < 2^n$, ceea ce dovedeşte faptul că implementarea recursivă are o complexitate exponenţială. Totuşi, există mai multe metode iterative cu complexitate liniară pentru rezolvarea acestei probleme: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/.

În continuare, vom prezenta câteva probleme clasice care se pot rezolva utilizând tehnica de programare Divide et Impera.

4. Problema căutării binare

Fie t o listă formată din n numere întregi sortate crescător şi x un număr întreg. Să se verifice dacă valoarea x apare în lista t.

Evident, problema ar putea fi rezolvată printr-o simplă parcurgere a listei t (*căutare liniară*), obţinând un algoritm având complexitatea $\mathcal{O}(n)$, dar nu am utiliza deloc faptul că elementele listei sunt în ordine crescătoare. Pentru a efectua o căutare binară într-o secvenţă $t[st], t[st+1], \dots, t[dr]$ a listei t în care $st \leq dr$ vom folosi această ipoteză, comparând valoarea căutată x cu valoarea $t[mij]$ aflată în mijlocul secvenţei. Astfel, vom obţine următoarele 3 cazuri:

a) $x < t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[st], \dots, t[mij-1]$;
b) $x > t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[mij+1], \dots, t[dr]$;
c) $x = t[mij] \Longrightarrow$ am găsit valoarea x, deci operaţia de căutare se încheie cu succes.

Dacă la un moment dat $st > dr$, înseamnă că nu mai există nicio secvenţă $t[st],, \dots, t[dr]$ în care să aibă sens să căutăm valoarea x, deci operaţia de căutare eşuează.

O implementare a căutării binare în limbajul Python, sub forma unei funcţii care furnizează o poziţie pe care apare valoarea x în lista t sau valoarea -1 dacă x nu apare deloc în listă, este următoarea:

```
def cautare_binara(t, x, st, dr):
    if st > dr:
        return -1

    mij = (st+dr) // 2
    if x == t[mij]:
        return mij
    elif x < t[mij]:
        return cautare_binara(t, x, st, mij-1)
    else:
        return cautare_binara(t, x, mij+1, dr)
```

Se observă faptul că acest algoritm de tip Divide et Impera constă doar din etapa Divide, nemaifiind combinate soluţiilor subproblemelor (etapa Impera). Practic, la fiecare pas, problema curentă se restrânge la una dintre cele două subprobleme, ci nu se rezolvă ambele subprobleme! Astfel, ţinând cont de faptul că etapa Divide are complexitatea

$\mathcal{O}(1)$, relaţia de recurenţă asociată complexităţii algoritmului de căutare binară este următoarea:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Folosind atât iterarea directă a relaţiei de recurenţă, cât şi teorema master, demonstraţi faptul că $T(n) \in \mathcal{O}(\log_2 n)$!

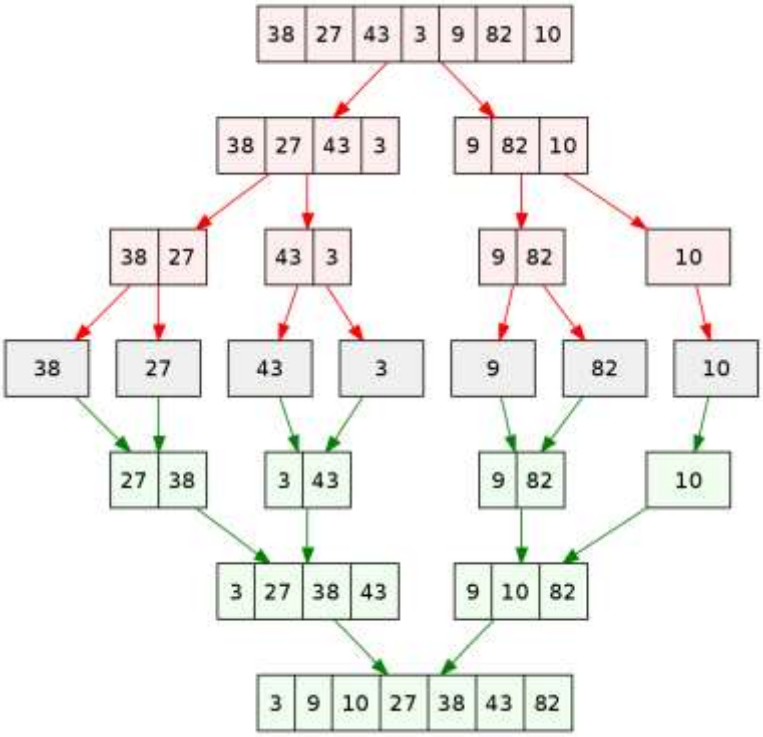
Observaţie importantă: Complexitatea $\mathcal{O}(\log_2 n)$ reprezintă strict complexitatea algoritmului de căutare binară, deci complexitatea unui program, chiar foarte simplu, în care se va utiliza acest algoritm va fi mai mare! De exemplu, un simplu program de test pentru funcţia de mai sus necesită citirea celor n elemente ale listei t şi afişarea valorii furnizate de funcţie, deci complexitatea sa va fi $\mathcal{O}(n) + \mathcal{O}(\log_2 n) + \mathcal{O}(1) \approx \mathcal{O}(n)$!

5. Sortarea prin interclasare (Mergesort)

Sortarea prin interclasare utilizează tehnica de programare Divide et Impera pentru a sorta crescător o listă t, astfel:

- se împarte secvenţa curentă $t[st], \dots, t[dr]$, în mod repetat, în două secvenţe $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$ până când se ajunge la secvenţe implicit sortate, adică secvenţe de lungime 1;
- în sens invers, se sortează secvenţa $t[st], \dots, t[dr]$ interclasând cele două secvenţe în care a fost descompusă, respectiv $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$, şi care au fost deja sortate la un pas anterior.

O reprezentară grafică a modului în care rulează această metodă de sortare se poate observa în următoarea imagine (sursa: https://en.wikipedia.org/wiki/Merge_algorithm):



Începem prezentarea detaliată a acestei metodei de sortare reamintind faptul că interclasarea este un algoritm care permite obținerea unei liste sortate crescător din două liste care sunt, de asemenea, sortate crescător. Considerând dimensiunile listelor care vor fi interclasate ca fiind egale cu m și n, complexitatea algoritmului de interclasare este $\mathcal{O}(m+n)$.

În cazul sortării prin interclasare, se vor interclasa secvențele $t[st], \dots, t[mij]$ și $t[mij+1], \dots, t[dr]$ într-o listă aux de lungime $dr-st+1$, iar la sfârșit elementele acesteia se vor copia în secvența $t[st], \dots, t[dr]$:

```
def interclasare(t, st, mij, dr):
    i = st
    j = mij+1
    aux = []
    while i <= mij and j <= dr:
        if t[i] <= t[j]:
            aux.append(t[i])
            i += 1
        else:
            aux.append(t[j])
            j += 1

    aux.extend(t[i:mij+1])
    aux.extend(t[j:dr+1])

    t[st:dr+1] = aux[:]
```

Se observă ușor faptul că funcția `interclasare` are o complexitate egală cu $\mathcal{O}(dr-st+1) \leq \mathcal{O}(n)$.

Sortarea listei t se va efectua, folosind tehnica Divide et Impera, în cadrul următoarei funcții:

```
def mergesort(t, st, dr):
    if st < dr:
        mij = (dr+st) // 2
        mergesort(t, st, mij)
        mergesort(t, mij+1, dr)
        interclasare(t, st, mij, dr)
```

Observați faptul că, în acest caz, funcția `interclasare` are rolul funcției `combinare` din algoritmul generic Divide et Impera!

Complexitatea funcției `mergesort` și, implicit, complexitatea sortării prin interclasare, se obține rezolvând următoarea relație de recurență:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

În exemplul 2 de la teorema master am demonstrat faptul că $T(n) \in \mathcal{O}(n \log_2 n)$, deci sortarea prin interclasare are complexitatea $\mathcal{O}(n \log_2 n)$.

6. Selecția celui de-al k-lea minim (Quickselect)

Considerăm o listă L de lungime n și dorim să determinăm al k-lea minim al său (evident, $1 \leq k \leq n$), respectiv valoarea care s-ar afla pe poziția $k-1$ după sortarea crescătoare a sa. De exemplu, pentru $A = [10, 7, 25, 4, 3, 4, 9, 12, 7]$ și $k = 5$ vom obține valoarea 7, deoarece lista A sortată crescător este $[3, 4, 4, 7, 7, 9, 10, 12, 25]$. Evident, problema poate fi rezolvată sortând lista și apoi accesând elementul aflat pe poziția $k-1$, complexitatea acestei soluții fiind $\mathcal{O}(n \log_2 n)$.

În continuare, vom prezenta un algoritm de tip Divide et Impera pentru rezolvarea acestei probleme:

- alegem aleatoriu un element din listă ca pivot;
- partiționăm lista A în 3 liste în raport de pivotul respectiv:
  - o listă L formată din elementele strict mai mici decât pivotul;
  - o listă E formată din elementele egale cu pivotul;
  - o listă G formată din elementele strict mai mari decât pivotul;
- comparăm valoarea k cu lungimile celor 3 liste de partiție și fie furnizăm valoarea căutată (dacă a k-a valoare se găsește în lista E), fie reluăm procedeul pentru una dintre listele L sau G.

De exemplu, considerând $A = [10, 7, 25, 4, 3, 4, 9, 12, 7]$, $k = 5$ și pivotul aleatoriu 9, vom obține următoarele mulțimi de partiție: $L = [7, 4, 3, 4, 7]$, $E = [9]$ și $G = [10, 25, 12]$. Deoarece $k = 5 \leq \text{len}(L)$, vom relua procedeul pentru lista L și aceeași valoare $k = 5$. Dacă am fi considerat $k = 8$, atunci am fi reluat procedeul pentru lista G și $k = 8 - len(L) - len(E) = 8 - 5 - 1 = 2$!

Implementarea în limbajul Python a acestui algoritm de tip Divide et Impera este următoarea:

```
import random

def quickselect(A, k, f_pivot=random.choice):
    pivot = f_pivot(A)

    L = [x for x in A if x < pivot]
    E = [x for x in A if x == pivot]
    G = [x for x in A if x > pivot]

    if k < len(L):
        return quickselect(L, k, f_pivot)
    elif k < len(L) + len(E):
        return E[0]
    else:
        return quickselect(G, k - len(L) - len(E), f_pivot)
```

Deoarece pivotul poate fi selectat și în alte moduri (ci nu numai aleatoriu), am folosit mecanismul de call-back în cadrul funcției `quickselect` astfel încât ea să poată utiliza o anumită funcție pentru selectarea pivotului, primită prin intermediul parametrului `f_pivot`. Implicit, acest parametru este inițializat cu funcția `choice` din pachetul `random`, care furnizează în mod aleatoriu o valoare dintr-o colecție dată ca parametru. Atenție, funcția trebuie apelată prin `quickselect(A, k-1)`!

Deoarece pivotul este ales în mod aleatoriu, estimarea complexității algoritmului `quickselect` prezentat anterior este destul de complicată. Totuși, se poate observa ușor faptul că un pivot "bun" ar trebui să genereze două liste de partiție L și G cu dimensiuni aproximativ egale cu jumătate din numărul de elemente ale listei inițiale A. În acest caz, complexitatea algoritmului `quickselect` ar fi dată de următoarea relație de recurență:

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\substack{\text{se alege una dintre} \\ \text{listele } L \text{ sau } G}} + \underbrace{n}_{\substack{\text{crearea listelor} \\ L,E \text{ și } G}}$$

Pentru a rezolva această relație de recurență folosind teorema master, observăm faptul că $a = 1$ și $b = 2$, iar $f(n) = n \in \mathcal{O}(n^1) \Rightarrow p = 1 > \log_b a = \log_2 1 = 0$. Deoarece suntem în cazul c) al teoremei master, trebuie să verificăm și faptul că $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare $\Leftrightarrow \exists c < 1$ astfel încât $\frac{n}{2} \leq cn \Leftrightarrow \exists c < 1$ astfel încât $n \leq 2cn \Leftrightarrow \exists c < 1$ astfel încât $1 \leq 2c$, ceea ce este adevărat pentru orice valoare $c \in \left[\frac{1}{2}, 1\right)$. Astfel, obținem că $T(n) \in \mathcal{O}\big(f(n)\big) = \mathcal{O}(n)$. Pe de altă parte, selectarea repetată ca pivot a valorii minime/maxime din listă va conduce la complexitatea $\mathcal{O}(n^2)$!

Practic, un pivot "bun" ar trebui să fie *mediana listei* respective, adică valoarea aflată la mijlocul listei sortate crescător, sau, cel puțin, o valoare apropiată de aceasta. În continuare, vom prezenta algoritmul *mediana medianelor*, tot de tip Divide et Impera, care permite determinarea unui pivot "bun" pentru o listă. Algoritmul a fost creat în anul 1973 de către informaticienii M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest și R. Tarjan, din acest motiv fiind cunoscut și sub numele de *algoritmul BFPRT*. Pașii algoritmului, pentru o listă A formată din n elemente, sunt următorii:

- se împarte lista A în $\left[\frac{n}{5}\right]$ liste de lungime 5 (dacă ultima listă nu are exact 5 elemente, atunci ea poate fi eliminată);
- pentru fiecare dintre cele $\left[\frac{n}{5}\right]$ liste de lungime 5 se determină direct mediana sa (e.g., se sortează lista și se returnează elementul aflat în mijlocul său);
- se reia procedeul pentru lista formată din medianele celor $\left[\frac{n}{5}\right]$ liste de lungime 5 până când se obține mediana medianelor (i.e., lista medianelor are cel mult 5 elemente).

Exemplu: Considerăm lista $A = [3, 14, 10, 2, 15, 10, 5, 51, 15, 20, 40, 4, 18, 13, 8, 40, 21, 61, 19, 50, 12, 35, 8, 7, 22, ~~100, 17~~]$ cu $n = 27$ elemente. După sortarea fiecărei liste de lungime 5 (am eliminat lista [100,17], deoarece este formată doar din două elemente) și determinarea medianelor lor, reluăm procedeul pentru lista medianelor, respectiv [10, 15, 13, 40, 12], și obținem pivotul 13:

	2	5	4	19	7
	3	10	8	21	8
Mediane	10	15	13	40	12
	14	20	18	50	22
	15	51	40	61	35

Implementarea acestui algoritm în limbajul Python este foarte simplă:

```
def BFPRT(A):
    if len(A) <= 5:
        return sorted(A)[len(A) // 2]

    grupuri = [sorted(A[i:i + 5]) for i in range(0, len(A), 5)]
    mediane = [grup[len(grup) // 2] for grup in grupuri]

    return BFPRT(mediane)
```

Rearanjând listele de lungime 5 în ordinea crescătoare a medianelor lor, obținem:

Mediane

2	7	4	5	19
3	8	8	10	21
10	12	13	15	40
14	22	18	20	50
15	35	40	51	61

Se observă faptul că elementele marcate cu galben sunt mai mici decât pivotul 13, iar elementele marcate cu verde sunt mai mari decât pivotul. De asemenea, se observă faptul că aproximativ jumătate dintre medianele grupurilor sunt mai mici decât pivotul și aproximativ jumătate sunt mai mari decât el. Astfel, rezultă faptul că fiecare dintre cele două grupuri va avea cel puțin $\frac{3n}{10}$ elemente, deoarece în fiecare grup intră aproximativ jumătate din numărul de $\left[\frac{n}{5}\right]$ liste de lungime 5, deci $\left[\frac{n}{10}\right]$ liste, iar în fiecare listă sunt 3 elemente. În același timp, fiecare dintre cele două grupuri nu poate avea mai mult de $n - \frac{3n}{10} = \frac{7n}{10}$ elemente, deci oricare dintre cele două grupuri va avea un număr de elemente cuprins între $\frac{3n}{10}$ și $\frac{7n}{10}$.

Practic, elementele marcate cu galben sunt cele din lista L definită în algoritmul `quickselect`, iar cele marcate cu verde sunt cele din lista G, deci utilizând algoritmul `BFPRT` pentru selectarea pivotului, prin apelul `quickselect(A, k-1, BFPRT)`, vom obține următoarea complexitate a sa:

$$T(n) \leq \underbrace{T\left(\frac{n}{5}\right)}_{\substack{\text{determinarea pivotului} \\ \text{folosind mediana medianelor}}} + \underbrace{T\left(\frac{7n}{10}\right)}_{\substack{\text{selectarea uneia dintre} \\ \text{listele } L \text{ sau } G}} + \underbrace{n}_{\substack{\text{crearea listelor} \\ L,E \text{ și } G}}$$

Pentru rezolvarea acestei relații de recurență nu putem utiliza teorema master, deoarece dimensiunile subproblemelor nu sunt egale, dar se poate intui faptul că $T(n) \leq cn$ și apoi demonstra acest lucru folosind inducția matematică (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec01.pdf), deci $T(n) \in \mathcal{O}(n)$. Cu toate acestea, deoarece valoarea constantei c pentru care se obține o complexitate liniară este mare, respectiv $c = 140$, în practică se preferă utilizarea variantei cu pivot ales aleatoriu deoarece are o complexitate medie mai bună!

TEHNICA PROGRAMĂRII DINAMICE

1. Prezentare generală

Programarea dinamică este o tehnică de programare utilizată, de obicei, tot pentru rezolvarea problemelor de optimizare. Programarea dinamică a fost inventată de către matematicianul american Richard Bellman în anii '50 în scopul optimizării unor probleme de planificare, deci cuvântul *programare* din denumirea acestei tehnici are, de fapt, semnificaţia de *planificare*, ci nu semnificaţia din informatică! Practic, tehnica programării dinamice poate fi utilizată pentru planificarea optimă a unor activităţi, iar decizia de a planifica sau nu o anumită activitate se va lua *dinamic*, ţinând cont de activităţile planificate până în momentul respectiv. Astfel, se observă faptul că tehnica programării dinamice diferă de tehnica Greedy (care este utilizată tot pentru rezolvarea problemelor de optim), în care decizia de a planifica sau nu o anumită activitate se ia într-un mod static, fără a ţine cont de activităţile planificate anterior, ci doar verificând dacă activitatea curentă îndeplineşte un anumit criteriu predefinit. Totuşi, cele două tehnici de programare se aseamănă prin faptul că ambele determină o singură soluţie a problemei, chiar dacă există mai multe.

În general, tehnica programării dinamice se poate utiliza pentru rezolvarea problemelor de optim care îndeplinesc următoarele două condiţii:

1. *condiţia de substructură optimală*: problema dată se poate descompune în subprobleme de acelaşi tip, iar soluţia sa optimă (optimul global) se obţine combinând soluţiile optime ale subproblemelor în care a fost descompusă (optime locale);
2. *condiţia de superpozabilitate*: subproblemele în care se descompune problema dată se suprapun.

Se poate observa faptul că prima condiţie este o combinaţie dintre o condiţie specifică tehnicii de programare *Divide et Impera* (problema dată se descompune în subprobleme de acelaşi tip) şi o condiţie specifică tehnicii *Greedy* (optimul global se obţine din optimele locale). Totuşi, o rezolvare a acestui tip de problemă folosind tehnica *Divide et Impera* ar fi ineficientă, deoarece subproblemele se suprapun (a doua condiţie), deci aceeaşi subproblemă ar fi rezolvată de mai multe ori, ceea ce ar conduce la un timp de executare foarte mare (de obicei, chiar exponenţial)!

Pentru a evita rezolvarea repetată a unei subprobleme, se va utiliza *tehnica memoizării*: fiecare subproblemă va fi rezolvată o singură dată, iar soluţia sa va fi păstrată într-o structură de date corespunzătoare, de obicei, unidimensională sau bidimensională.

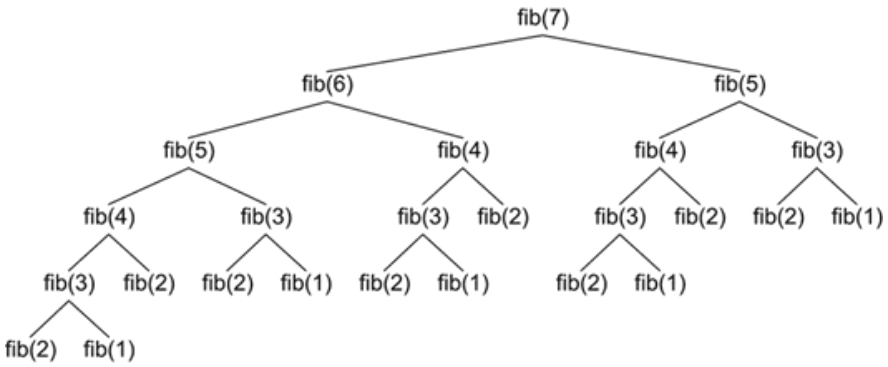
De exemplu, să considerăm şirul lui Fibonacci, definit recurent astfel:

$$f_n = \begin{cases} 0, & \text{dacă } n = 1 \\ 1, & \text{dacă } n = 2 \\ f_{n-2} + f_{n-1}, & \text{dacă } n \geq 3 \end{cases}$$

O implementare directă a relației de recurență de mai sus pentru a calcula termenul f_n se poate realiza utilizând o funcție recursivă:

```
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Pentru a calcula termenul f_7 vom apela funcția prin `fib(7)` și vom obține următorul arbore de apeluri recursive (sursa imaginii: https://medium.com/@shmuel.lotman/the-2-00-am-javascript-blog-about-memoization-41347e8fa603):



Se observă faptul că anumiți termeni ai șirului se calculează în mod repetat (de exemplu, termenul $f_3 =$ `fib(3)` se va calcula de 5 ori), ceea ce va conduce la o complexitate exponențială (am demonstrat acest fapt în capitolul dedicat tehnicii Divide et Impera).

Pentru a evita calcularea repetată a unor termeni ai șirului, vom folosi tehnica memoizării: vom utiliza o listă `f` pentru a memora termenii șirului, iar fiecare termen nou `f[i]` va fi calculat ca sumă a celor doi termeni precedenți, respectiv `f[i-2]` și `f[i-1]`:

```
def fib(n):
    f = [-1, 0, 1]
    for i in range(3, n+1):
        f.append(f[i-2] + f[i-1])
    return f[n]
```

Se observă faptul că lista `f` este completată într-o maniera ascendentă (*bottom-up*), respectiv se începe cu subproblemele direct rezolvabile (cazurile $n = 0$ și $n = 1$) și apoi se calculează restul termenilor șirului, până la valoarea dorită `f[n]`. Practic, putem afirma faptul că se efectuează doar etapa Impera din rezolvarea de tip Divide et Impera!

În concluzie, rezolvarea unei probleme utilizând tehnica programării dinamice necesită parcurgerea următoarelor etape:

1) se identifică subproblemele problemei date și se determină o relație de recurență care să furnizeze soluția optimă a problemei în funcție de soluțiile optime ale subproblemelor sale (se utilizează substructura optimală a problemei);
2) se identifică o structură de date capabilă să rețină soluțiile subproblemelor;

3) se rezolvă iterativ relaţia de recurenţă, folosind tehnica memoizării într-o manieră ascendentă (respectiv se rezolvă subproblemele în ordinea crescătoare a dimensiunilor lor), obţinându-se astfel valoarea optimă căutată;
4) se construieşte o soluţie care furnizează valoarea optimă, utilizând soluţiile subproblemelor calculate anterior (această etapă este opţională).

2. Suma maximă într-un triunghi de numere

Considerăm un triunghi format din n şiruri din numere întregi, astfel: prima linie conţine un număr, a doua linie conţine două numere, ..., a n-a linie (ultima) conţine n numere (un astfel de triunghi este, de fapt, jumătatea inferioară a unei matrice pătratice de dimensiune n). De exemplu, un triunghi t de numere cu dimensiunea $n = 5$ este următorul:

$$t = \begin{pmatrix} 10 & & & & \\ -2 & 15 & & & \\ 13 & -8 & -10 & & \\ -17 & 1 & 21 & 16 & \\ 7 & 3 & -11 & 14 & 1 \end{pmatrix}$$

Să se determine cea mai mare sumă formată din elemente aflate pe un traseu care începe pe prima linie şi se termină pe ultima, iar succesorul fiecărui element de pe traseu (mai puţin în cazul ultimului) este situat pe linia următoare, fie sub el, fie în dreapta sa. Practic, succesorul unui element `t[i][j]` este fie `t[i+1][j]`, fie `t[i+1][j+1]`.

Pentru triunghiul `t` de mai sus, suma maximă care se poate obţine este 52, pe traseul `t[0][0]->t[1][1]->t[2][1]->t[3][2]->t[4][3]`, marcat cu roşu în triunghi.

Fiind o problemă de optim, într-o primă variantă de rezolvare am putea încerca aplicarea unui algoritm de tip Greedy, respectiv succesorul fiecărui element `t[i][j]` de pe traseu să fie ales maximul dintre `t[i+1][j]`şi `t[i+1][j+1]`. Deşi traseul marcat cu roşu în triunghiul de mai sus are această proprietate, se poate observa uşor faptul că algoritmul Greedy ar eşua dacă modificăm valoarea 7 din colţul stânga-jos în 700! În acest caz, algoritmul Greedy ar selecta tot elementele aflate pe traseul marcat cu roşu, dar suma maximă s-ar obţine, de fapt, pe traseul format din elementele aflate pe prima coloană a triunghiului. Mai mult, traseul selectat de algoritmul Greedy ar rămâne cel marcat cu roşu, indiferent cum am modifica elementele marcate cu albastru!

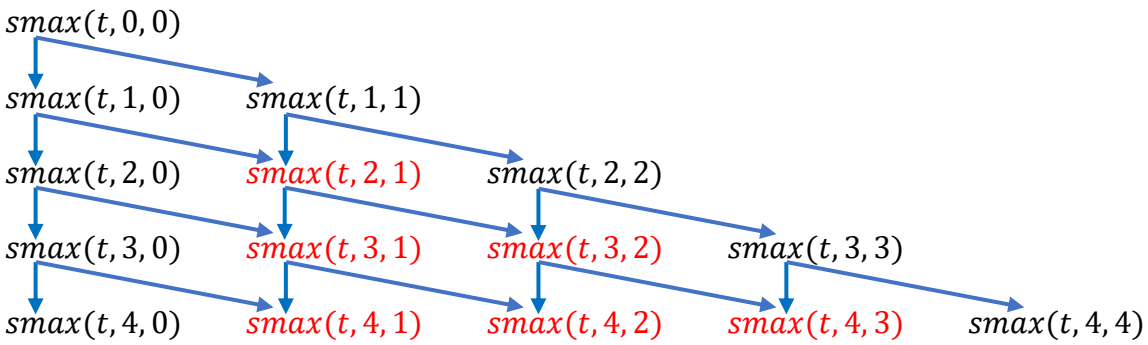
O altă variantă de rezolvare ar putea utiliza tehnica Backtracking pentru generarea tuturor traseelor care respectă cerinţele problemei şi selectarea celui care are suma elementelor maximă. Această variantă de rezolvare este corectă, dar ineficientă, deoarece are complexitatea exponenţială $\mathcal{O}(2^{n-1})$. Pentru a demonstra acest lucru, vom calcula numărul total de trasee care respectă cerinţele problemei. În acest scop, vom codifica deplasarea din elementul curent `t[i][j]` în elementul `t[i+1][j]` cu 0 şi deplasarea din elementul curent `t[i][j]` în elementul `t[i+1][j+1]` cu 1, iar oricărui traseu corect îi vom asocia un şir binar de lungime $n - 1$. De exemplu, traseului marcat cu roşu în triunghiul de mai sus i se asociază şirul binar 1011, traseului format din elementele de pe prima coloană i se asociază şirul binar 0000 (cel mai mic în sens lexicografic), iar traseului

format din elementele de pe diagonala i se asociază şirul binar 1111 (cel mai mare în sens lexicografic). Deoarece această asociere este bijectivă (unui traseu îi corespunde un singur şir binar, iar unui şir binar îi corespunde un singur traseu), rezultă că numărul traseelor corecte este egal cu numărul şirurilor binare de lungime $n-1$, deci cu 2^{n-1}.

O altă variantă de rezolvare ar putea utiliza tehnica Divide et Impera, observând faptul că problema considerată îndeplineşte condiţiile cerute pentru utilizarea acestei tehnici de programare: suma maximă pe un traseu care începe cu elementul `t[i][j]` este egală cu el plus maximul sumelor care se obţin pe cele două trasee care încep cu `t[i+1][j]`şi `t[i+1][j+1]`, iar problema direct rezolvabilă o constituie calcularea sumei maxime care se poate obţine pe un traseu care începe cu un element aflat pe ultima linie a triunghiului, deoarece, evident, aceasta este egală chiar cu elementul respectiv. Considerând $smax(t,i,j)$ o funcţie care calculează suma maximă pe un traseu care începe cu elementul `t[i][j]`, rezultă că putem să o definim în manieră Divide et Impera astfel:

$$smax(t,i,j)=\begin{cases} t[i][j], & \text{dacă } i=n-1 \\ t[i][j]+max\,(smax(t,i+1,j),smax(t,i+1,j+1)), & \text{dacă } i<n-1 \end{cases}$$

unde $max(x,y)$ este o funcţie care calculează maximul dintre numerele întregi x şi y, iar suma maximă cerută se obţine în urma apelului $smax(t,0,0)$. Analizând graful apelurilor recursive, se poate observa faptul că sumele maxime corespunzătoare anumitor trasee se calculează de câte două ori:



De exemplu, suma maximă corespunzătoare traseului care începe cu `t[2][1]`, adică $smax(t,2,1)$ se calculează atât în cadrul apelului $smax(t,1,0)$, cât şi în cazul apelului $smax(t,1,1)$. Mai mult, acest lucru se întâmplă pentru toate traseele care nu încep cu element aflat pe prima coloană sau pe diagonală (marcate cu roşu în graful de mai sus)! Astfel, se poate observa faptul că utilizarea metodei Divide et Impera este corectă, dar ineficientă, deoarece subproblemele se suprapun. Practic, complexitatea acestei variante este tot $\mathcal{O}(2^{n-1})$, deoarece, la fel ca şi în cazul variantei Backtracking, se parcurg, până la urmă, toate traseele din triunghi!

Totuşi, formula recurentă prin care este definită funcţia $smax(t,i,j)$ exprimă *condiţia de substructură optimală* a problemei date, iar faptul că subproblemele se suprapun pe cea de *superpozabilitate*, deci putem să rezolvăm această problemă utilizând tehnica programării dinamice. Practic, pentru rezolvarea relaţiei de recurenţă, vom aplica tehnica memoizării, utilizând un triunghi de numere *smax* pentru a reţine soluţiile subproblemelor, respectiv elementul `smax[i][j]` va păstra suma maximă pe un traseu care începe cu elementul `t[i][j]`:

$$smax[i][j] = \begin{cases} t[i][j], & \text{dacă } i = n-1 \\ t[i][j] + \max\{smax[i+1][j], smax[i+1][j+1]\}, & \text{dacă } 0 \le i < n-1 \end{cases}$$

pentru fiecare $j \in \{0,1,\dots,i\}$.

Valorile elementelor matricei *smax* se vor calcula într-o manieră ascendentă (*bottom-up*), respectiv se va copia ultima linie a triunghiului *t* în matricea *smax*, după care se vor completa restul liniilor, de jos în sus și de la stânga la dreapta:

$$t = \begin{pmatrix} 10 & & & & \\ -2 & 15 & & & \\ 13 & -8 & -10 & & \\ -17 & 1 & 21 & 16 & \\ 7 & 3 & -11 & 14 & 1 \end{pmatrix} \quad smax = \begin{pmatrix} \boxed{52} & & & & \\ 25 & 42 & & & \\ 17 & 27 & 25 & & \\ -10 & 4 & 35 & 30 & \\ 7 & 3 & -11 & 14 & 1 \end{pmatrix}$$

De exemplu, elementul $smax[2][1] = 27$ a fost calculat astfel: $smax[2][1] = t[2][1] + \max\{smax[3][1], smax[3][2]\} = -8 + \max\{4, 35\} = 27$.

Soluția problemei (suma maximă) este dată de $smax[0][0] = 52$, iar reconstituirea traseului pe care a fost obținută suma maximă se poate realiza într-o manieră Greedy, utilizând elementele matricei *smax*, respectiv plecăm din elementul aflat pe prima linie a matricei *smax* și apoi ne deplasăm pe cel mai mare dintre cei doi succesori posibili ai elementului curent:

$$smax = \begin{pmatrix} 52 & & & & \\ 25 & 42 & & & \\ 17 & 27 & 25 & & \\ -10 & 4 & 35 & 30 & \\ 7 & 3 & -11 & 14 & 1 \end{pmatrix}$$

Reconstituirea traseului se poate realiza într-o manieră Greedy deoarece matricea *smax* este o matrice de optime locale care au condus la un optim global!

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că triunghiul de numere se citește din fișierul text `triunghi.txt`:

```
# citim triunghiul de numere din fișierul text "triunghi.txt"
f = open("triunghi.txt")
t = []
for linie in f:
    aux = [int(x) for x in linie.split()]
    t.append(aux)
f.close()
```

```
# creăm un triunghi smax de aceeași dimensiune cu triunghiul t
n = len(t)
smax = [[0]*(i+1) for i in range(n)]

# copiem ultima linie a triunghiului t în triunghiul smax
for i in range(n):
    smax[n-1][i] = t[n-1][i]

# calculăm restul elementelor triunghiului smax
for i in range(n-2, -1, -1):
    for j in range(i+1):
        smax[i][j] = t[i][j] + max(smax[i+1][j], smax[i+1][j+1])

# afișăm suma maximă care se poate obține
print("Suma maxima:", smax[0][0])

# afișăm un traseu pe care se poate obține suma maximă
print("\nTraseul cu suma maxima:")
j = 0
for i in range(n-1):
    print("{}({}, {}) -> ".format(t[i][j], i, j), end="")
    if smax[i+1][j+1] > smax[i+1][j]:
        j += 1
print("{}({}, {})".format(t[n-1][j], n-1, j))
```

Evident, complexitatea algoritmului prezentat este egală cu $\mathcal{O}(n^2)$.

O altă variantă de rezolvare a acestei probleme se poate obține modificând semnificația unui element $smax[i][j]$, respectiv acesta va păstra suma maximă pe un traseu care se termină cu elementul `t[i][j]`. În acest caz, pentru a scrie relațiile de recurență care să exprime *condiția de substructură optimală* a problemei date, vom considera elementele triunghiului din care se poate ajunge în elementul `t[i][j]` respectând restricțiile problemei (predecesorii săi), respectiv elementele `t[i-1][j-1]` și `t[i-1][j]`. Se observă faptul că în elementele `t[0][j]` (cele aflate pe prima coloană a triunghiului) se poate ajunge doar din elementele `t[i-1][j]` aflate imediat deasupra sa, iar în elementele `t[i][i]` (cele aflate pe diagonala triunghiului) se poate ajunge doar din elementele `t[i-1][j-1]` aflate tot pe diagonală în direcția stânga-sus față de ele! Astfel, obținem următoarea relație de recurență:

$$smax[i][j] = \begin{cases} t[0][0], & \text{dacă } i = 0 \text{ și } j = 0 \\ t[i][j] + smax[i-1][j], & \text{dacă } 1 \le i < n-1 \text{ și } j = 0 \\ t[i][j] + smax[i-1][j-1], & \text{dacă } 1 \le i \le n-1 \text{ și } j = i \\ t[i][j] + \max\{smax[i-1][j], smax[i-1][j-1]\}, & \text{în orice alt caz} \end{cases}$$

În acest caz, valorile elementelor matricei $smax$ se vor calcula astfel: se va copia primul element al triunghiului t în matricea $smax$, după care se vor completa restul liniilor, de sus în jos și de la stânga la dreapta:

$$t = \begin{pmatrix} 10 & & & & \\ -2 & 15 & & & \\ 13 & -8 & -10 & & \\ -17 & 1 & 21 & 16 & \\ 7 & 3 & -11 & 14 & 1 \end{pmatrix} \quad \left| \quad smax = \begin{pmatrix} 10 & & & & \\ 8 & 25 & & & \\ 21 & 17 & 15 & & \\ 4 & 22 & 38 & 31 & \\ 11 & 25 & 27 & \boxed{52} & 32 \end{pmatrix} \right.$$

De exemplu, elementul $smax[3][1] = 22$ a fost calculat astfel: $smax[3][1] = t[3][1] + \max\{smax[2][1], smax[2][2]\} = 1 + \max\{21, 17\} = 22$.

Soluția problemei (suma maximă) este dată de maximul de pe ultima linie a matricei *smax*, respectiv $smax[4][3] = 52$, iar reconstituirea traseului pe care a fost obținută suma maximă se poate realiza într-o manieră Greedy, utilizând elementele matricei *smax*, astfel: plecăm din elementul maxim de pe ultima linie a matricei *smax* și apoi ne deplasăm pe cel mai mare dintre cei doi predecesori posibili ai elementului curent:

$$smax = \begin{pmatrix} 10 & & & & \\ 8 & 25 & & & \\ 21 & 17 & 15 & & \\ 4 & 22 & 38 & 31 & \\ 11 & 25 & 27 & 52 & 32 \end{pmatrix}$$

În acest caz, traseul se va reconstitui în sens invers, deci pentru afișarea sa începând cu elementul din vârful triunghiului trebuie să utilizăm o structură de date auxiliară în care să salvăm soluția și apoi să o afișăm invers.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că triunghiul de numere se citește din fișierul text `triunghi.txt`:

```
# citim triunghiul de numere din fișierul text "triunghi.txt"
f = open("triunghi.txt")
t = []
for linie in f:
    aux = [int(x) for x in linie.split()]
    t.append(aux)
f.close()

# creăm un triunghi smax de aceeași dimensiune cu triunghiul t
n = len(t)
smax = [[0]*(i+1) for i in range(n)]

# copiem elementul din vârful triunghiului t
# în vârful triunghiului smax
smax[0][0] = t[0][0]
```

```
# calculăm restul elementelor triunghiului smax
for i in range(1, n):
    for j in range(i+1):
        if j == 0:
            smax[i][j] = t[i][j] + smax[i-1][j]
        elif j == i:
            smax[i][j] = t[i][j] + smax[i-1][j-1]
        else:
            smax[i][j] = t[i][j] + max(smax[i-1][j], smax[i-1][j-1])

# determinăm poziția maximului de pe ultima linie din smax
pmax = 0
for j in range(1, n):
    if smax[n-1][j] > smax[n-1][pmax]:
        pmax = j

# construim în lista sol un traseu pe care se obține suma maximă,
# respectiv sol[i] va reține coloana pe care se află elementul
# selectat de pe linia i
j = pmax
sol = []
for i in range(n-1, 0, -1):
    sol.append(j)
    if j == 0:
        continue
    if i == j:
        j -= 1
    elif smax[i-1][j] < smax[i-1][j-1]:
        j -= 1

# adăugăm în lista sol și coloana 0, corespunzătoare
# elementului din vârful triunghiului
sol.append(0)

# deoarece traseul este construit de jos în sus,
# inversăm ordinea elementelor din lista sol
sol.reverse()

# afișăm suma maximă care se poate obține
print("Suma maxima:", smax[n-1][pmax])

# afișăm un traseu pe care se poate obține suma maximă
print("\nTraseul cu suma maxima:")
for i in range(n-1):
    print("{}({}, {}) -> ".format(t[i][sol[i]], i, sol[i]), end="")
print("{}({}, {})".format(t[n-1][sol[n-1]], n-1, sol[n-1]))
```

Complexitatea acestui algoritm este egală tot cu $\mathcal{O}(n^2)$.

Încheiem prezentarea problemei sumei maxime într-un triunghi de numere precizând faptul că ea a fost unul dintre subiectele date la Olimpiada Internațională de Informatică (ediția a VI-a) desfășurată în 1994 în Suedia: https://ioinformatics.org/page/ioi-1994/20 (problema *The Triangle*).

În literatura de specialitate, se consideră faptul că există 3 variante ale metodei programării dinamice, în funcţie de modul în care sunt calculate soluțiile subproblemelor:

- *varianta înainte*, dacă soluția subproblemei i depinde de soluțiile subproblemelor $i+1, i+2, \ldots, n-1$ (prima variantă de rezolvare a problemei sumei maxime într-un triunghi de numere);
- *varianta înapoi*, dacă soluția subproblemei i depinde de soluțiile subproblemelor $0,1, \ldots, i-1$ (a doua variantă de rezolvare a problemei sumei maxime într-un triunghi de numere);
- *varianta mixtă,* dacă se combină cele două variante anterioare.

TEHNICA PROGRAMĂRII DINAMICE

1. Subşir crescător maximal

Considerăm un şir t format din n numere întregi $t = (t_0, t_1, \dots, t_{n-1})$. Să se determine un subşir crescător de lungime maximă al şirului dat t.

Reamintim faptul că un subşir al unui şir este format din elemente ale şirului iniţial ai căror indici sunt în ordine strict crescătoare (i.e., un subşir de lungime m al şirului t este $s = (t_{i_0}, t_{i_1}, \dots, t_{i_m})$ cu $0 \le i_0 < i_1 < \dots < i_m \le n-1$) sau, echivalent, este format din elemente ale şirului iniţial între care se păstrează ordinea relativă iniţială.

De exemplu, în şirul $t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$ un subşir crescător maximal este $(3, 6, 8, 8, 10)$. Soluţia nu este unică, un alt subşir crescător maximal fiind $(3, 4, 5, 8, 10)$. Se observă faptul că problema are întotdeauna soluţie, chiar şi în cazul în care şirul dat este strict descrescător (orice element al şirului este un subşir crescător maximal de lungime 1)!

Algoritmii de tip Greedy nu rezolvă corect această problemă în orice caz. De exemplu, pentru fiecare element din şirul dat, am putea încerca să construim un subşir crescător maximal selectând, de fiecare dată, cel mai apropiat element mai mare sau egal decât ultimul element din subşirul curent şi să reţinem subşirul crescător de lungime maximă astfel obţinut. Aplicând acest algoritm pentru şirul $t = (7, 3, 9, 4, 5)$ vom obţine subşirurile $(7, 9)$, $(3, 9)$, (9), $(4, 5)$ şi (5), deci soluţia furnizată de algoritmul Greedy ar fi unul dintre cele 3 subşiruri crescătoare de lungime 2. Evident, soluţia ar fi incorectă, deoarece soluţia optimă este subşirul $(3, 4, 5)$, de lungime 3!

Un algoritm de tip Backtracking ar trebui să genereze toate submulţimile strict crescătoare de indici (combinări) cu $1, 2, \dots, n$ elemente, pentru fiecare submulţime de indici să testeze dacă subşirul asociat este crescător şi, în caz afirmativ, să reţină subşirul de lungime maximă. Algoritmul este corect, dar ineficient, deoarece numărul submulţimilor generate şi testate ar fi egal cu $C_n^1 + C_n^2 + \dots + C_n^n = 2^n - 1$, deci algoritmul are avea o complexitate exponenţială!

În continuare, vom prezenta un algoritm pentru rezolvarea acestei probleme folosind tehnica programării dinamice (un algoritm de tip Divide et Impera s-ar baza pe aceeaşi idee, însă fără a utiliza tehnica memoizării).

Pentru a determina un subşir crescător maximal, vom calcula, într-o manieră ascendentă, lungimea maximă a unui subşir crescător care se termină cu $t[0]$, apoi lungimea maximă a unui subşir crescător care se termină cu $t[1]$, ..., respectiv lungimea maximă a unui subşir crescător care se termină cu $t[n-1]$, iar valorile obţinute (optimele locale) le vom păstra într-o listă $lmax$ cu n elemente, respectiv $lmax[i]$ va memora lungimea maximă a unui subşir crescător care se termină cu $t[i]$.

Pentru a calcula lungimea maximă a unui subşir crescător care se termină cu elementul $t[i]$, vom lua în considerare, pe rând, toate subşirurile care se termină cu elementele $t[0], t[1], \dots, t[i-1]$, deoarece cunoaştem deja lungimile maxime $lmax[0], lmax[1], \dots, lmax[i-1]$ ale subşirurilor crescătoare care se termină cu ele, şi vom încerca să alipim elementul $t[i]$ la fiecare dintre ele. Dacă acest lucru este posibil, respectiv dacă $t[i] \ge t[j]$ pentru un indice $j \in \{0, 1, \dots, i-1\}$, vom compara lungimea subşirului care s-ar obţine, egală cu $1 + lmax[j]$, cu lungimea maximă $lmax[i]$ găsită până în acel moment şi, dacă noua lungime este mai mare, vom actualiza valoarea lui $lmax[i]$.

Se observă uşor faptul că problema dată îndeplineşte atât *condiţia de substructură optimală* (calcularea valorii $lmax[i]$ depinde de valorile $lmax[0], lmax[1], \dots, lmax[i-1]$), cât şi *condiţia de superpozabilitate* (valoarea $lmax[i]$ va fi utilizată în calculul valorilor $lmax[i+1], \dots, lmax[n-1]$), iar relaţia de recurenţă care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

$$lmax[i] = \begin{cases} 1, & \text{pentru } i = 0 \\ 1 + \max\limits_{0 \le j < i} \{lmax[j] | t[i] \ge t[j]\}, & \text{pentru } 1 \le i \le n-1 \end{cases}$$

Pentru a reconstitui mai simplu un subşir crescător maximal al şirului iniţial, vom utiliza o listă auxiliară $pred$, în care un element $pred[i]$ va conţine valoarea -1 dacă elementul $t[i]$ nu a putut fi alipit la niciunul dintre subşirurile crescătoare maximale care se termină cu $t[0], t[1], \dots, t[i-1]$ sau va conţine indicele $j \in \{0,1, \dots, i-1\}$ al subşirului crescător maximal $t[j]$ la care a fost alipit elementul $t[i]$, adică indicele j pentru care s-a obţinut $\max\limits_{0 \le j < i} \{lmax[j] | t[i] \ge t[j]\}$.

Considerând şirul t din exemplul iniţial, vom obţine următoarele valori:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
t	9	7	3	6	2	8	5	4	5	8	7	10	4
lmax	1	1	1	2	1	3	2	2	3	4	4	5	3
pred	-1	-1	-1	2	-1	3	2	2	6	5	8	9	7

Valorile din listele $lmax$ şi $pred$ au fost calculate astfel:

- $lmax[0] = 1$ şi $pred[0] = -1$, deoarece este evident faptul că lungimea maximă a unui subşir care se termină cu $t[0]$este egală cu 1;
- $lmax[1] = 1$ şi $pred[1] = -1$, deoarece elementul $t[1] = 7$ nu poate fi alipit la un subşir crescător maximal care se termină cu $t[0] = 9 > 7$;
- $lmax[2] = 1$ şi $pred[2] = -1$, deoarece elementul $t[2] = 3$ nu poate fi alipit nici la un subşir crescător maximal care se termină cu $t[0] = 9 > 3$ şi nici la un subşir crescător maximal care se termină cu $t[1] = 7 > 3$;
- $lmax[3] = 2$ şi $pred[3] = 2$, deoarece elementul $t[3] = 6$ poate fi alipit la un subşir crescător maximal care se termină cu $t[2] = 3 \le 6$, deci $lmax[3] = 1 + lmax[2] = 2$;
- $lmax[4] = 1$ şi $pred[4] = -1$, deoarece elementul $t[4] = 2$ nu poate fi alipit nici la un subşir crescător maximal care se termină cu $t[0], t[1], t[2]$ sau $t[3]$;
- $lmax[5] = 3$ şi $pred[5] = 3$, deoarece elementul $t[5] = 8$ poate fi alipit la oricare dintre subşirurile crescătoare maximale care se termină cu $t[1], t[2], t[3]$ sau $t[4]$, dar lungimea maximă se obţine când îl alipim la subşirul crescător maximal care se termină cu $t[3]$, deoarece $lmax[3]$ este cea mai mare dintre valorile $lmax[1]$, $lmax[2], lmax[3]$ şi $lmax[4]$;
- .....

Lungimea maximă a unui subşir crescător maximal al şirului iniţial este dată de valoarea maximă din lista $lmax$, iar pentru a reconstitui un subşir crescător maximal vom

utiliza informațiile din lista $pred$ și faptul că un subșir crescător maximal se termină cu elementul $t[pmax]$, unde $pmax$ este poziția pe care se află valoarea maximă din $lmax$.

În cazul exemplului de mai sus, valoarea maximă din lista $lmax$ este $lmax[11] = 5$, deci $pmax = 11$, ceea ce înseamnă că un subșir crescător maximal se termină cu $t[11]$. Pentru afișarea unui subșir crescător maximal vom proceda astfel:

- inițializăm un indice i cu $pmax$, deci $i = 11$;
- $i = 11 \neq -1$, deci afișăm elementul $t[i] = t[11] = 10$ și indicele i devine egal cu $pred[11] = 9$;
- $i = 9 \neq -1$, deci afișăm elementul $t[i] = t[9] = 8$ și indicele i devine egal cu $pred[9] = 5$;
- $i = 5 \neq -1$, deci afișăm elementul $t[i] = t[5] = 8$ și indicele i devine egal cu $pred[5] = 3$;
- $i = 3 \neq -1$, deci afișăm elementul $t[i] = t[3] = 6$ și indicele i devine egal cu $pred[3] = 2$;
- $i = 2 \neq -1$, deci afișăm elementul $t[i] = t[2] = 3$ și indicele i devine egal cu $pred[2] = -1$;
- $i = -1$, deci am terminat de afișat un subșir crescător maximal (dar inversat!) și ne oprim.

Pentru a afișa subșirul crescător maximal reconstituit neinversat, îl vom stoca într-o listă și apoi o vom afișa invers.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că elementele șirului t se citesc din fișierul text `sir.txt`:

```
f = open("sir.txt")
t = [int(x) for x in f.readline().split()]
f.close()

n = len(t)
lmax = [1] * n
pred = [-1] * n
lmax[0] = 1
for i in range(1, n):
    for j in range(0, i):
        if t[j] <= t[i] and lmax[i] < 1 + lmax[j]:
            pred[i] = j
            lmax[i] = 1 + lmax[j]
pmax = 0
for i in range(1, n):
    if lmax[i] > lmax[pmax]:
        pmax = i
print("Lungimea maxima a unui subsir crescator:", lmax[pmax])
print("Un subsir crescator maximal:")
i = pmax
sol = []
while i != -1:
    sol.append(t[i])
    i = pred[i]

print(*sol[::-1])
```

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice (deoarece pentru a calcula soluția optimă $lmax[i]$ a subproblemei i am utilizat soluțiile optime $lmax[0], lmax[1], \dots, lmax[i-1]$ ale subproblemelor $0,1, \dots, i-1$), iar complexitatea sa este egală cu $\mathcal{O}(n^2)$.

În continuare, vom prezenta un algoritm care utilizează varianta înainte a tehnicii programării dinamice, respectiv vom calcula soluția optimă $lmax[i]$ a subproblemei i utilizând soluțiile optime $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$ ale subproblemelor $i+1, i+2, \dots, n-1$. În acest scop, vom calcula, într-o manieră ascendentă, lungimea maximă a unui subșir crescător care începe cu $t[n-1]$, apoi lungimea maximă a unui subșir crescător care începe cu $t[n-2]$, ..., respectiv lungimea maximă a unui subșir crescător care începe cu $t[n-1]$, iar valorile obținute (optimele locale) le vom păstra în lista $lmax$ cu n elemente, respectiv $lmax[i]$ va memora lungimea maximă a unui subșir crescător care începe cu $t[i]$.

Pentru a calcula lungimea maximă a unui subșir crescător care începe cu elementul $t[i]$, vom lua în considerare, pe rând, toate subșirurile care încep cu elementele $t[i+1], t[i+2], \dots, t[n-1]$, deoarece cunoaștem deja lungimile maxime $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$ ale subșirurilor crescătoare care încep cu ele, și vom încerca să adăugăm elementul $t[i]$ înaintea fiecăruia dintre ele. Dacă acest lucru este posibil, respectiv dacă $t[i] \leq t[j]$ pentru un indice $j \in \{i+1, i+2, \dots, n-1\}$, vom compara lungimea subșirului care s-ar obține, egală cu $1 + lmax[j]$, cu lungimea maximă $lmax[i]$ găsită până în acel moment și, dacă noua lungime este mai mare, vom actualiza valoarea lui $lmax[i]$.

Se observă ușor faptul că problema dată îndeplinește atât *condiția de substructură optimală* (calcularea valorii $lmax[i]$ depinde de valorile $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$), cât și *condiția de superpozabilitate* (valoarea $lmax[i]$ va fi utilizată în calculul valorilor $lmax[0], \dots, lmax[i-1]$), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

$$lmax[i] = \begin{cases} 1, & \text{pentru } i = n-1 \\ 1 + \max\limits_{i+1 \leq j < n} \{lmax[j] | t[i] \leq t[j]\}, & \text{pentru } 0 \leq i < n-1 \end{cases}$$

Pentru a reconstitui mai simplu un subșir crescător maximal al șirului inițial, vom utiliza o listă auxiliară $succ$, în care un element $succ[i]$ va conține valoarea -1 dacă elementul $t[i]$ nu a putut fi adăugat înaintea niciunuia dintre subșirurile crescătoare maximale care încep cu $t[i+1], t[i+2], \dots, t[n-1]$ sau va conține indicele $j \in \{i+1, i+2, \dots, n-1\}$ al subșirului crescător maximal $t[j]$ înaintea căruia a fost adăugat elementul $t[i]$, adică indicele j pentru care s-a obținut $\max\limits_{i+1 \leq j < n} \{lmax[j] | t[i] \leq t[j]\}$.

Considerând șirul t din exemplul inițial, vom obține următoarele valori:

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
t	**9**	**7**	**3**	**6**	**2**	**8**	**5**	**4**	**5**	**8**	**7**	**10**	**4**
lmax	**2**	**4**	**5**	**4**	**5**	**3**	**4**	**4**	**3**	**2**	**2**	**1**	**1**
succ	**11**	**5**	**3**	**5**	**6**	**9**	**8**	**8**	**9**	**11**	**11**	**-1**	**-1**

Valorile din listele $lmax$ şi $succ$ au fost calculate astfel:

- $lmax[12] = 1$ şi $succ[12] = -1$, deoarece este evident faptul că lungimea maximă a unui subşir care începe cu $t[12]$este egală cu 1;
- $lmax[11] = 1$ şi $succ[11] = -1$, deoarece elementul $t[11] = 10$ nu poate fi adăugat înaintea unui subşir crescător maximal care începe cu $t[12] = 4 < 10$;
- $lmax[10] = 2$ şi $succ[10] = 11$, deoarece elementul $t[10] = 7$ poate fi adăugat înaintea unui subşir crescător maximal care începe cu $t[11] = 10 \geq 7$, deci $lmax[10] = 1 + lmax[11] = 2$;
- $lmax[9] = 2$ şi $succ[9] = 11$, deoarece elementul $t[9] = 8$ poate fi adăugat înaintea unui subşir crescător maximal care începe cu $t[11] = 10 \geq 8$, deci $lmax[9] = 1 + lmax[11] = 2$;
- $lmax[8] = 3$ şi $succ[8] = 9$, deoarece elementul $t[8] = 5$ poate fi adăugat înaintea oricăruia dintre subşirurile crescătoare maximale care încep cu $t[9], t[10]$ sau $t[11]$, dar lungimea maximă se obţine când îl alipim la subşirul crescător maximal care începe cu $t[9]$, deoarece $lmax[9]$ este cea mai mare dintre valorile $lmax[9]$, $lmax[10]$ şi $lmax[11]$;
- .....

Lungimea maximă a unui subşir crescător maximal al şirului iniţial este dată de valoarea maximă din lista $lmax$, iar pentru a reconstitui un subşir crescător maximal vom utiliza informaţiile din lista $succ$ şi faptul că un subşir crescător maximal începe cu elementul $t[pmax]$, unde $pmax$ este poziţia pe care se află valoarea maximă din lista $lmax$.

În cazul exemplului de mai sus, valoarea maximă din lista $lmax$ este $lmax[2] = 5$, deci $pmax = 2$, ceea ce înseamnă că un subşir crescător maximal începe cu $t[2]$. Pentru afişarea unui subşir crescător maximal vom proceda astfel:

- iniţializăm un indice i cu $pmax$, deci $i = 2$;
- $i = 2 \neq -1$, deci afişăm elementul $t[i] = t[2] = 3$ şi indicele i devine egal cu $succ[2] = 3$;
- $i = 3 \neq -1$, deci afişăm elementul $t[i] = t[3] = 6$ şi indicele i devine egal cu $succ[3] = 5$;
- $i = 5 \neq -1$, deci afişăm elementul $t[i] = t[5] = 8$ şi indicele i devine egal cu $succ[5] = 9$;
- $i = 9 \neq -1$, deci afişăm elementul $t[i] = t[9] = 8$ şi indicele i devine egal cu $succ[9] = 11$;
- $i = 11 \neq -1$, deci afişăm elementul $t[i] = t[11] = 10$ şi indicele i devine egal cu $succ[11] = -1$;
- $i = -1$, deci am terminat de afişat un subşir crescător maximal şi ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că elementele şirului t se citesc din fişierul text `sir.txt`:

```
f = open("sir.txt")
t = [int(x) for x in f.readline().split()]
f.close()

n = len(t)
lmax = [1] * n
```

```
succ = [-1] * n

lmax[n-1] = 1
for i in range(n-2, -1, -1):
    for j in range(i+1, n):
        if t[i] <= t[j] and lmax[i] < 1 + lmax[j]:
            succ[i] = j
            lmax[i] = 1 + lmax[j]

pmax = 0
for i in range(1, n):
    if lmax[i] > lmax[pmax]:
        pmax = i

print("Lungimea maxima a unui subsir crescator:", lmax[pmax])
print("Un subsir crescator maximal: ")
i = pmax
while i != -1:
    print(t[i], end=" ")
    i = succ[i]
```

Algoritmul prezentat are complexitatea egală tot cu $\mathcal{O}(n^2)$, aceasta nefiind însă minimă. O rezolvare cu complexitatea $\mathcal{O}(n \log_2 n)$ se poate obține utilizând căutarea binară: https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/.

2. Plata unei sume folosind un număr minim de monede cu valori date

Considerând faptul că avem la dispoziție n monede cu valorile $v_1, v_2, \dots, v_n$ pe care putem să le folosim pentru a plăti o sumă P, trebuie să determinăm o modalitate de plată a sumei date folosind un număr minim de monede (vom presupune faptul că avem la dispoziție un număr suficient de monede din fiecare tip).

De exemplu, dacă avem la dispoziție $n = 3$ tipuri de monede cu valorile $v = (2\$, 3\$, 5\$)$, atunci putem să plătim suma $P = 12\$$ în 5 moduri: $4 \times 3\$, 1 \times 2\$ + 2 \times 5\$, 2 \times 2\$ + 1 \times 3\$ + 1 \times 5\$, 3 \times 2\$ + 2 \times 3\$$ și $6 \times 2\$$. Evident, numărul minim de monede pe care putem să-l folosim este 3, corespunzător variantei $1 \times 2\$ + 2 \times 5\$$.

Pentru a genera toate modalitățile de plată a unei sume folosind monede cu valori date se poate utiliza tehnica Backtracking, algoritmul fiind deja prezentat în capitolul dedicat tehnicii de programare respective. Modificând algoritmul respectiv, putem determina și o modalitate de plată a unei sume folosind un număr minim de monede (pentru fiecare modalitate de plată vom calcula numărul de monede utilizate și vom reține modalitatea cu număr minim de monede), dar algoritmul va avea o complexitate exponențială, deci va fi ineficient!

Fiind o problemă de optim, putem încerca și o rezolvare de tip Greedy, respectiv să utilizăm pentru plata sumei, la fiecare pas, un număr maxim de monede cu cea mai valoare dintre cele neutilizate deja pentru plata sumei. Pentru exemplul de mai sus, vom considera monedele în ordinea descrescătoare a valorilor lor, respectiv $v = (5\$, 3\$, 2\$)$, și vom plăti suma $P = 12\$$, astfel:

- utilizăm 2 monede cu valoarea de 5$, deci suma de plată rămasă devine $P = 2$\$;
- nu putem utiliza nicio monedă cu valoarea de 3$;
- utilizăm o monedă cu valoarea de 2$, deci suma de plată rămasă devine $P = 0$\$ și algoritmul se termină cu succes;
- numărul de monede utilizate, respectiv 3 monede, este minim.

Totuși, această rezolvare de tip Greedy nu va furniza rezultatul optim în orice caz. De exemplu, dacă monedele au valorile $v = (5\$, 4\$, 1\$)$ și suma de plată este $P = 8$\$, folosind algoritmul Greedy vom găsi următoarea soluție:

- utilizăm o monedă cu valoarea de 5$, deci suma de plată rămasă devine $P = 3$\$;
- nu putem utiliza nicio monedă cu valoarea de 4$;
- utilizăm 3 monede cu valoarea de 1$, deci suma de plată rămasă devine $P = 0$\$ și algoritmul se termină cu succes;
- numărul de monede utilizate, respectiv 4 monede, nu este minim (numărul minim de monede se obține când se utilizează două monede cu valoarea de 4$).

Se observă faptul că existența monedei cu valoarea de 1$ permite algoritmului Greedy să găsească întotdeauna o soluție, chiar dacă aceasta nu este optimă. Totuși, sunt cazuri în care algoritmul Greedy nu va găsi nicio soluție, deși problema are cel puțin una. De exemplu, dacă monedele au valorile $v = (5\$, 4\$, 2\$)$ și suma de plată este tot $P = 8$\$, vom proceda astfel:

- utilizăm o monedă cu valoarea de 5$, deci suma de plată rămasă devine $P = 3$\$;
- nu putem utiliza nicio monedă cu valoarea de 4$;
- utilizăm o monedă cu valoarea de 2$, deci suma de plată rămasă devine $P = 1$\$;
- deoarece nu mai există alte tipuri de monede, algoritmul se termină fără să găsească o soluție, optimă sau nu!

Deoarece această problema are o importanță practică deosebită, în anumite țări sunt utilizate așa-numitele *sisteme canonice de valori pentru monede*, care permit algoritmului Greedy prezentat mai sus (numit și *algoritmul casierului*) să furnizeze o soluție optimă pentru orice sumă de plată (https://www.cs.princeton.edu/courses/archive/spring07/cos423/lectures/greed-dp.pdf).

Pentru a rezolva problema folosind metoda programării dinamice, observăm faptul că numărul minim de monede necesare pentru a plăti o sumă P folosind o monedă cu valoarea x (evident, $1 \leq x \leq P$) se obține adăugând 1 la numărul minim de monede necesar pentru a plăti suma $P - x$ utilizând toate tipurile de monede disponibile. De exemplu, numărul minim de monede necesare pentru a plăti suma $P = 12$\$ folosind o monedă cu valoarea $x = 5$\$ se obține adăugând 1 la numărul minim de monede necesare pentru a plăti suma $P - x = 7$\$ utilizând toate tipurile de monede disponibile. Deoarece suma $P - x$ poate să aibă orice valoare cuprinsă între 0 și $P - 1$, rezultă că pentru a putea calcula numărul minim de monede necesare pentru a plăti suma P folosind o monedă cu valoarea x trebuie să cunoaștem numărul minim de monede necesare pentru a plăti orice sumă cuprinsă între 0 și $P - 1$ folosind toate tipurile de monede disponibile cu valorile $v_1, v_2, \dots, v_n$. Generalizând această observație pentru toate tipurile de monede date, observăm faptul că numărul minim de monede necesare pentru a plăti o sumă P folosind toate tipurile de monede se obține adăugând 1 la minimul dintre: numărul minim de monede necesare pentru a plăti suma $P - v_1$, numărul minim de monede necesare pentru a plăti suma $P - v_2$,..., numărul minim de monede necesare pentru a plăti suma $P - v_n$ (evident, se vor lua în considerare doar cazurile în care moneda cu valoare v_i poate fi utilizată pentru plata sumei P, adică $v_i \leq P$).

Considerând o listă $nrmin$ cu $P+1$ elemente de tip întreg în care elementul $nrmin[i]$ va reține numărul minim de monede necesare pentru a plăti suma i, cuprinsă între 0 și P, folosind toate tipurile de monede disponibile, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

$$nrmin[i] = \begin{cases} 0, & \text{pentru } i = 0 \\ 1 + \min\limits_{0 \le j < n} \{nrmin[i - v[j]] \mid v[j] \le i\}, & \text{pentru } 1 \le i \le P \end{cases}$$

Inițial, toate elementele listei $nrmin$ trebuie să aibă valoarea "$+\infty$", adică o valoare strict mai mare decât orice valoare posibilă pentru elementele sale. Deoarece numărul maxim de monede pe care îl putem folosi pentru a plăti suma maximă P este chiar P (valoarea minimă a unei monede este 1$!), vom inițializa toate elementele listei $nrmin$ cu $P+1$.

Soluția problemei, adică numărul minim de monede necesare pentru a plăti suma P, este dată de valoarea $nrmin[P]$, dacă ea este diferită de valoarea de inițializare $P+1$, altfel, dacă rămâne egală cu P+1, înseamnă că suma P nu poate fi plătită folosind monede cu valorile date. Pentru a reconstitui mai ușor o modalitate optimă de plată a sumei P vom folosi o listă $pred$, tot cu $P+1$ elemente de tip întreg, în care un element $pred[i]$ va conține valoarea -1 dacă nu există nicio modalitate de plată a sumei i folosind monedele date sau va conține valoarea monedei $v[j]$ utilizată pentru a plăti suma i cu un număr minim de monede, adică valoarea $v[j]$ pentru care s-a obținut $\min\limits_{0 \le j < n} \{nrmin[i - v[j]] \mid v[j] \le i\}$.

Considerând exemplul dat, cu $P = 12\$$ și $v = (2\$, 5\$, 3\$)$ (valorile monedelor nu trebuie să fie sortate!), vom obține următoarele valori pentru elementele listelor $nrmin$ și $pred$ (am notat cu $+\infty$ valoarea $P + 1 = 13$):

i	0	1	2	3	4	5	6	7	8	9	10	11	12
nrmin	0	+∞	1	1	2	1	2	2	2	3	2	3	3
pred	-1	-1	2	3	2	5	3	2	5	2	5	5	2

Valorile evidențiate din listele $nrmin$ și $pred$ au fost calculate astfel:

- $nrmin[1] = +\infty$ și $pred[1] = -1$, deoarece niciuna dintre monedele cu valorile 2$, 5$ și 3$ nu poate fi utilizată pentru a plăti suma $i = 1\$$;
- $nrmin[4] = 2$ și $pred[4] = 2$, deoarece doar monedele cu valorile 2$ și 3$ pot fi utilizate pentru a plăti suma $i = 4\$$ și $nrmin[4] = 1 + \min\{nrmin[4-2], nrmin[4-3]\} = 1 + \min\{nrmin[2], nrmin[1]\} = 1 + \min\{1, +\infty\} = 2$, deci minimul a fost obținut pentru moneda cu valoarea 2$;
- $nrmin[7] = 2$ și $pred[7] = 2$, deoarece toate monedele pot fi utilizate pentru a plăti suma $i = 7\$$ și $nrmin[7] = 1 + \min\{nrmin[7-2], nrmin[7-5], nrmin[7-3]\} = 1 + \min\{nrmin[5], nrmin[2], nrmin[4]\} = 1 + \min\{1,1,2\} = 2$, deci minimul a fost obținut pentru moneda cu valoarea 2$;
- $nrmin[12] = 3$ și $pred[12] = 2$, deoarece toate monedele pot fi utilizate pentru a plăti suma $i = 12\$$ și $nrmin[12] = 1 + \min\{nrmin[12-2], nrmin[12-5], nrmin[12-3]\} = 1 + \min\{nrmin[10], nrmin[7], nrmin[9]\} = 1 + \min\{2,2,3\} = 3$, deci minimul a fost obținut pentru moneda cu valoarea 2$.

Numărul minim de monede necesare pentru a plăti suma $P = 12\$$ folosind monede cu valorile $v = (2\$, 5\$, 3\$)$ este $nrmin[12] = 3$, iar pentru a reconstitui o modalitate optimă de plată vom utiliza informațiile din lista $pred$, astfel:

- inițializăm un indice i cu P, deci $i = 12$ (variabila i reprezintă suma curentă de plată);
- $pred[i] = pred[12] = 2 \neq -1$, deci pentru a plăti suma $i = 12\$$ folosind un număr minim de monede a fost utilizată o monedă cu valoarea de 2$, pe care o afișăm, și apoi indicele i devine egal cu $i - pred[i] = 10$ (suma de plată rămasă);
- $pred[i] = pred[10] = 5 \neq -1$, deci pentru a plăti suma $i = 10\$$ folosind un număr minim de monede a fost utilizată o monedă cu valoarea de 5$, pe care o afișăm, și apoi indicele i devine egal cu $i - pred[i] = 5$ (suma de plată rămasă);
- $pred[i] = pred[5] = 5 \neq -1$, deci pentru a plăti suma $i = 5\$$ folosind un număr minim de monede a fost utilizată o monedă cu valoarea de 5$, pe care o afișăm, și apoi indicele i devine egal cu $i - pred[i] = 0$ (suma de plată rămasă);
- $pred[i] = pred[0] = -1$, deci am terminat de afișat o modalitate de plată a sumei folosind un număr minim de monede și ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că datele de intrare se citesc din fișierul text `monede.txt`, care conține pe prima linie valorile monedelor, iar pe a doua linie se află suma de plată P:

```
# citim datele de intrare din fișierul text "monede.txt"
# pe prima linie avem valorile monedelor (elementele listei v)
f = open("monede.txt")
aux = f.readline()
v = [int(x) for x in aux.split()]
# a doua linie conține suma de plată P
aux = f.readline()
P = int(aux)

# inițializăm listele nrmin și pred
nrmin = [P+1] * (P+1)
nrmin[0] = 0
pred = [-1] * (P+1)
# calculăm valorile nrmin[1],..., nrmin[P]
# folosind relația de recurență prezentată
for suma in range(1, P+1):
    for moneda in v:
        if moneda <= suma and 1 + nrmin[suma-moneda] < nrmin[suma]:
            nrmin[suma] = 1 + nrmin[suma-moneda]
            pred[suma] = moneda

# afișăm datele de ieșire
if nrmin[P] == P+1:
    print("Suma", P, "nu poate fi platita!")
else:
    print("Numărul minim de monede necesare pentru a plăti suma", P,
"este", nrmin[P])
    print("O modalitate de plată:")
    suma = P
```

```
    while pred[suma] != -1:
        print(pred[suma], end=" ")
        suma = suma - pred[suma]
```

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice, iar complexitatea sa este egală cu $\mathcal{O}(nP)$. O astfel de complexitate se numeşte *complexitate pseudo-polinomială*, deoarece P nu reprezintă o dimensiune a datelor de intrare, ci o valoare a unei date de intrare! Pentru a exprima complexitatea acestui algoritm doar în raport de dimensiunile datelor de intrare vom folosi faptul ca un număr întreg strict pozitiv x poate fi reprezentat în formă binară folosind minim $1 + [\log_2 x]$ biţi, deci complexitatea acestui algoritm este, de fapt, $\mathcal{O}\left(n2^{1+[\log_2 P]}\right) \approx \mathcal{O}\left(n2^{[\log_2 P]}\right)$, ceea ce înseamnă că are *o complexitate liniară în raport cu numărul n de monede* şi *o complexitate exponenţială în raport cu lungimea reprezentării binare a sumei P de plată*!

3. Problema rucsacului (varianta discretă)

Considerăm un rucsac având capacitatea maximă G şi n obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaştem greutăţile lor $g_1, g_2, \dots, g_n$ şi câştigurile $c_1, c_2, \dots, c_n$ obţinute prin încărcarea lor în rucsac. Ştiind faptul că toate greutăţile şi toate câştigurile sunt numere naturale nenule, iar orice obiect poate fi încărcat doar complet în rucsac (nu poate fi "tăiat"), să se determine o modalitate de încărcare a rucsacului astfel încât câştigul total obţinut să fie maxim.

De exemplu, considerând $G = 10$ kg şi $n = 5$ obiecte O_1, O_2, O_3, O_4, O_5 având câştigurile $c = (80, 50, 400, 60, 70)$ RON şi greutăţile $g = (5, 2, 20, 3, 4)$ kg, putem obţine un câştig maxim egal cu 190 RON, încărcând obiectele O_1, O_2 şi O_4.

În capitolul dedicat tehnicii de programare Greedy am văzut faptul că varianta fracţionară a acestei probleme poate fi rezolvată corect utilizând tehnica respectivă. În cazul variantei discrete, tehnica Greedy nu va mai furniza o soluţie corectă întotdeauna. Astfel, câştigurile unitare ale obiectelor din exemplul de mai sus vor fi $u = (16, 25, 20, 20, 17.5)$ RON/kg. Astfel, algoritmul Greedy ar selecta obiectele O_2, O_4 şi O_5, deoarece obiectele nu pot fi "tăiate" în varianta discretă a problemei rucsacului, şi ar obţine un câştig total egal cu 180 RON, evident mai mic decât cel maxim de 190 RON!

Se observă foarte uşor faptul că varianta discretă a problemei rucsacului nu are întotdeauna soluţie, respectiv în cazul în care greutatea celui mai mic obiect este strict mai mare decât capacitatea G a rucsacului, în timp ce varianta fracţionară ar avea soluţie în acest caz (ar "tăia" din obiectul cu cel mai mare câştig unitar o bucată cu greutatea G).

Pentru a rezolva problema folosind metoda programării dinamice, vom proceda într-un mod asemănător cu cel utilizat pentru a rezolva problema plăţii unei sume folosind un număr minim de monede, astfel:

- considerăm faptul că am analizat, pe rând, obiectele $O_1, O_2, \dots, O_{n-1}$ şi am calculat câştigul maxim pe care îl putem obţine folosindu-le (nu neapărat pe toate!) în limita întregii capacităţi G a rucsacului, deci mai trebuie să calculăm doar câştigul maxim pe care îl putem obţine folosind şi ultimul obiect O_n;
- dacă obiectul O_n nu încape în rucsac (deci $g_n > G$), înseamnă că nu-l putem folosi deloc, deci câştigul maxim rămâne cel pe care l-am obţinut deja utilizând obiectele $O_1, O_2, \dots, O_{n-1}$;
- dacă obiectul O_n încape în rucsac (deci $g_n \leq G$), înseamnă că trebuie să decidem dacă este rentabil să-l încărcăm sau nu, comparând câştigul maxim deja obţinut

folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita întregii capacități G a rucsacului cu câștigul care s-ar obține prin încărcarea obiectului O_n, respectiv cu suma dintre c_n și câștigul maxim care se poate obține folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita capacității $G - g_n$ rămase în rucsac. Deoarece $1 \leq g_n \leq G$, rezultă că trebuie să cunoaștem câștigurile maxime care se pot obține folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacități cuprinse între 0 și G-1, la care se adaugă câștigul maxim care se poate obține folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita întregii capacități G a rucsacului (pentru cazul anterior), deci, de fapt, trebuie să cunoaștem câștigurile maxime care se pot obține folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacități cuprinse între 0 și G!

- pentru a calcula câștigurile maxime care se pot obține folosind primele $n-1$ obiecte $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacități cuprinse între 0 și G vom repeta raționamentul anterior pentru obiectul O_{n-1} și obiectele $O_1, O_2, \dots, O_{n-2}$, apoi pentru obiectul O_{n-2} și obiectele $O_1, O_2, \dots, O_{n-3}$ și așa mai departe, până când vom calcula câștigurile maxime care se pot obține folosind doar primul obiect O_1 în limita oricărei capacități cuprinse între 0 și G.

În concluzie, pentru a rezolva problema utilizând tehnica programării dinamice, trebuie să cunoaștem toate câștigurile maxime care se pot obține folosind primele i obiecte ($i \in \{0,1,\dots,n\}$), în limita oricărei capacități j cuprinse între 0 și G, deci, aplicând tehnica memoizării, vom considera un tablou bidimensional $cmax$ cu $n+1$ linii și $G+1$ coloane în care un element $cmax[i][j]$ va memora câștigul maxim care se poate obține folosind primele i obiecte în limita a j kilograme.

Astfel, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

$$cmax[i][j] = \begin{cases} 0, & \text{dacă i} = 0 \text{ sau } j = 0 \\ cmax[i-1][j], & \text{dacă } g_i > j \\ max\{cmax[i-1][j], c[i] + cmax[i-1][j - g[i]]\}, & \text{dacă } g_i \leq j \end{cases}$$

pentru fiecare $i \in \{0,1,\dots,n\}$ și fiecare $j \in \{0,1,\dots,G\}$. În plus față de modalitatea de calcul a elementului $cmax[i][j]$ descrisă mai sus, am adăugat cazurile particulare $cmax[0][j] = cmax[i][0] = 0$ (evident, câștigul maxim $cmax[0][j]$ care se poate obține folosind 0 obiecte în limita oricărei capacități j este 0 și câștigul maxim $cmax[i][0]$ care se poate obține folosind primele i obiecte în limita unei capacități nule este tot 0). De asemenea, am considerat tablourile c și g ca fiind indexate de la 1, pentru a păstra indicii din relațiile prezentate mai sus!

Considerând exemplul dat, vom obține următoarele valori pentru elementele matricei $cmax$:

	c_i	g_i	i/j	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0	0
O_1	80	5	1	0	0	0	0	0	80	80	80	80	80	80
O_2	50	2	2	0	0	50	50	50	80	80	130	130	130	130
O_3	400	20	3	0	0	50	50	50	80	80	130	130	130	130
O_4	60	3	4	0	0	50	60	60	110	110	130	140	140	190
O_5	70	4	5	0	0	50	60	70	110	120	130	140	180	190

Elementele evidențiate în matricea $cmax$ au fost calculate astfel:

- $cmax[1][1] = cmax[1][2] = cmax[1][3] = cmax[1][4] = 0$, deoarece obiectul O_1 are greutatea $g_1 = 5$, deci poate fi încărcat doar în cazul în care capacitatea j a rucsacului este cel puțin egală cu 5 (de exemplu, folosind relația de recurență, obținem $cmax[1][2] = cmax[0][2] = 0$), caz în care am obținut $cmax[1][5] = \cdots = cmax[1][10] = 80$ (de exemplu, folosind relația de recurență, obținem $cmax[1][9] = \max\{cmax[0][9], c[1] + cmax[0][9-5]\} = \max\{0, 80+0\} = 80$);
- $cmax[2][7] = \max\{cmax[1][7], c[2] + cmax[1][7-2]\} = \max\{80, 50+80\} = 130$, deoarece în limita a $j = 7$ kg încap ambele obiecte O_1 și O_2;
- linia 3 este egală cu linia 2, deoarece $g_3 = 20$ kg $> G = 10$ kg, deci obiectul O_3 nu se poate încărca în niciun caz în rucsac;
- $cmax[4][10] = \max\{cmax[3][10], c[4] + cmax[3][10-3]\} = \max\{130, 60+130\} = 190$, deoarece în limita a $j = 10$ kg se poate adăuga obiectul O_4 la obiectele O_1 și O_2 care au fost încărcate pentru a obține câștigul maxim folosind primele $i = 3$ obiecte în limita a $j = 7$ kg;
- $cmax[5][9] = \max\{cmax[4][9], c[5] + cmax[4][9-4]\} = \max\{140, 70+110\} = 180$, deoarece în limita a $j = 9$ kg este mai rentabil să încărcăm obiectul O_5 alături de obiectele O_2 și O_4 (pentru care s-a obținut câștigul maxim de 110 RON folosind primele $i = 4$ obiecte în limita a $j = 5$ kg) decât să nu-l încărcăm, caz în care am păstra câștigul maxim de 140 RON obținut prin încărcarea obiectelor O_1 și O_4 dintre primele $i = 4$ obiecte în limita a $j = 9$ kg.

Câștigul maxim care se poate obține folosind toate cele n obiecte este dat de valoarea elementului $cmax[n][G]$, iar pentru a reconstitui o modalitate optimă de încărcare a rucsacului vom utiliza informațiile din matricea $cmax$, astfel:

- considerăm doi indici $i = n$ și $j = G$;
- dacă $cmax[i][j] = cmax[i-1][j]$, înseamnă fie că obiectul O_i nu încape în rucsac, fie încape, dar nu ar fi fost rentabil să-l încărcăm. Indiferent de motiv, obiectul O_i nu a fost încărcat în rucsac (nu face parte din soluția optimă), deci trecem la următorul obiect O_{i-1}, decrementând valoarea indicelui i;

- dacă $cmax[i][j] \neq cmax[i-1][j]$, înseamnă că a fost rentabil să încărcăm obiectul O_i în limita a j kg, deci îl afișăm și trecem la reconstituirea soluției optime pentru restul de $j - g[i]$ kg folosind obiectele $O_1, O_2, \dots, O_{i-1}$, scăzând din indicele j valoarea $g[i]$ și decrementând indicele i.

Se observă faptul că obiectele se vor afișa în ordinea descrescătoare a indicilor lor (în "sens invers"), deci trebuie utilizată o structură de date auxiliară sau o funcție recursivă pentru a le afișa în ordinea crescătoare a indicilor lor!

În cazul exemplului de mai sus, avem $cmax[5][10] = 190$, deci profitul maxim care se poate obține este de 190 RON, iar pentru reconstituirea unei modalități optime de încărcare a rucsacului vom urma traseul marcat cu roșu în matricea $cmax$, obiectele care se vor încărca în rucsac corespunzând liniilor pe care se află elementele încadrate cu un dreptunghi, respectiv obiectele O_1, O_2 și O_4:

	c_i	g_i	i/j	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0	0
O_1	80	5	1	0	0	0	0	0	80	80	80	80	80	80
O_2	50	2	2	0	0	50	50	50	80	80	130	130	130	130
O_3	400	20	3	0	0	50	50	50	80	80	130	130	130	130
O_4	60	3	4	0	0	50	60	60	110	110	130	140	140	190
O_5	70	4	5	0	0	50	60	70	110	120	130	140	180	190

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că datele de intrare se citesc din fișierul text `obiecte.txt`, care conține pe prima linie capacitatea G a rucsacului, iar pe fiecare dintre următoarele n linii se află greutatea și câștigul câte unui obiect. Datele de ieșire, respectiv o modalitate optimă de încărcare a rucsacului, se vor scrie în fișierul text `rucsac.txt`.

```
# fișierul de intrare conține pe prima linie
# capacitatea G a rucsacului
f = open("obiecte.txt")
G = int(f.readline())

# greutățile obiectelor se vor memora într-o listă g, iar
# câștigurile lor într-o listă c
# ambele liste trebuie să fie indexate de la 1, deci adăugăm
# în fiecare câte o valoare egală cu 0
g = [0]
c = [0]
# pe fiecare dintre liniile rămase, fișierul text conține
# greutatea și câștigul unui obiect
for linie in f:
    aux = linie.split()
```

```
    g.append(int(aux[0]))
    c.append(int(aux[1]))

f.close()

# n = numărul de obiecte
n = len(g) - 1

# inițializăm toate elementele matricei cmax cu 0
cmax = [[0 for x in range(G+1)] for x in range(n+1)]

# calculăm elementele matricei cmax folosind relația de recurență
for i in range(1, n+1):
for i in range(1, n+1):
    for j in range(1, G+1):
        cmax[i][j] = cmax[i-1][j]
        if g[i] <= j and c[i]+cmax[i-1][j-g[i]] > cmax[i-1][j]:
            cmax[i][j] = c[i]+cmax[i-1][j-g[i]]

# scriem în fișierul text rucsac.txt o modalitate optimă
# de încărcare a rucsacului
f = open("rucsac.txt", "w", encoding="UTF-8")

f.write("Câștigul maxim: " + str(cmax[n][G]) + "\n")
f.write("Obiectele încărcate: ")
i, j = n, G
while i != 0:
    if cmax[i][j] != cmax[i-1][j]:
        f.write(str(i) + " ")
        j = j - g[i]
    i = i - 1

f.close()
```

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice, iar complexitatea sa este una de tip pseudo-polinomial, fiind egală cu $\mathcal{O}(nG) \approx \mathcal{O}\left(n2^{[\log_2 G]}\right)$.

4. Planificarea proiectelor cu bonus maxim

Considerăm n proiecte $P_1, P_2, \ldots, P_n$ 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 un interval de timp în care acesta trebuie executat (exprimat prin numerele de ordine a două zile din perioada respectivă), precum şi bonusul pe care îl va obține echipa dacă proiectul este finalizat la timp (altfel, echipa nu va obține niciun bonus pentru proiectul respectiv). Să se determine o modalitate de planificare a unor proiecte care nu se suprapun astfel încât bonusul obținut de echipă să fie maxim. Vom considera faptul că un proiect care începe într-o anumită zi nu se suprapune cu un proiect care se termină în aceeaşi zi!

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 intervalul de timp în care proiectul trebuie executat şi bonusul acordat. De exemplu, a doua linie din fişierul de intrare conţine informaţiile despre proiectul P_1, respectiv intervalul [7, 13] în care acesta trebuie efectuat pentru ca echipa să obţină bonusul de 850 RON. Datele de ieşire se vor scrie în fişierul text `proiecte.out`, în forma indicată mai jos:

proiecte.in	**proiecte.out**
P1 7 13 850	P4: 02-06 -> 650 RON
P2 4 12 800	P1: 07-13 -> 850 RON
P3 1 3 250	P5: 13-18 -> 1000 RON
P4 2 6 650	P7: 25-27 -> 300 RON
P5 13 18 1000	
P6 4 16 900	Bonusul echipei: 2800 RON
P7 25 27 300	
P8 15 22 900	

Deşi problema este asemănătoare cu *problema planificării unor proiecte cu profit maxim*, prezentată în capitolul dedicat tehnicii de programare Greedy, în care intervalele de executare ale proiectelor sunt restrânse la o singură zi, o strategie de tip Greedy nu va furniza întotdeauna o soluţie corectă. De exemplu, dacă am planifica proiectele în ordinea descrescătoare a bonusurilor, atunci un proiect $P_1([1,10], 1000$ RON) cu bonus mare şi durată mare ar fi programat înaintea a două proiecte $P_2([1,5], 900$ RON) şi $P_3([6,9], 800$ RON) cu bonusuri şi durate mai mici, dar având suma bonusurilor mai mare decât bonusul primului proiect (900+800 = 1700 > 1000). Într-un mod asemănător se pot găsi contraexemple şi pentru alte criterii de selecţie bazate pe ziua de început, pe ziua de sfârşit, pe durată sau pe raportul dintre bonusul şi durata unui proiect!

Pentru a rezolva problema folosind metoda programării dinamice, vom proceda în următorul mod:

- considerăm proiectele $P_1, P_2, \dots, P_n$ ca fiind sortate în ordine crescătoare după ziua de sfârşit (vom vedea imediat de ce);
- considerăm faptul că am calculat bonusurile maxime $bmax_1, bmax_2, \dots, bmax_{i-1}$ pe care echipa le poate obţine planificând o parte dintre primele i proiecte $P_1, P_2, \dots, P_{i-1}$ (sau chiar pe toate!), iar acum trebuie să calculăm bonusul maxim $bmax_i$ pe care echipa îl poate obţine luând în considerare şi proiectul P_i;
- înainte de a calcula $bmax_i$, vom determina cel mai mare indice $j \in \{1,2, \dots, i-1\}$ al unui proiect P_j după care poate fi planificat proiectul P_i (i.e., ziua de început a proiectului P_i este mai mare sau egală decât ziua în care se termină proiectul P_j) şi vom nota acest indice j cu ult_i (dacă nu există nici un proiect P_j după care să poată fi planificat proiectul P_i, atunci vom considera $ult_i = 0$);
- calculăm $bmax_i$ ca fiind maximul dintre bonusul pe care îl echipa poate obţine dacă nu planifică proiectul P_i, adică $bmax_{i-1}$, şi bonusul pe care îl echipa poate obţine dacă planifică proiectul P_i după proiectul P_{ult_i}, adică $bonus_i + bmax_{ult_i}$,

unde prin $bonus_i$ am notat bonusul pe care îl primește echipa dacă finalizează proiectul P_i la timp.

Se observă faptul că ult_i se poate calcula mai ușor dacă proiectele sunt sortate crescător după ziua de terminare, deoarece ult_i va fi primul indice $j \in \{i-1, i-2, \dots, 1\}$ pentru care ziua de început a proiectului P_i este mai mare sau egală decât ziua în care se termină proiectul P_j. De asemenea, se observă faptul că valorile ult_i trebuie păstrate într-o listă, deoarece sunt necesare pentru reconstituirea soluției.

Folosind observațiile și notațiile anterioare, precum și tehnica memoizării, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

$$bmax[i] = \begin{cases} 0, & \text{dacă } i = 0 \\ \max\{bmax[i-1], bonus[i] + bmax[ult[i]]\}, & \text{dacă } i \geq 1 \end{cases}$$

Bonusul maxim pe care îl poate obține echipa este dat de valoarea elementului $bmax[n]$, iar pentru a reconstitui o modalitate optimă de planificare a proiectelor vom utiliza informațiile din matricea $bmax$, astfel:

- considerăm un indice $i = n$;
- dacă $bmax[i] \neq bmax[i-1]$, înseamnă că proiectul P_i a fost utilizat în planificarea optimă, deci îl afișăm și trecem la reconstituirea soluției optime care se termină cu proiectul $P_{ult[i]}$ după care a fost planificat proiectul P_i, respectiv indicele i ia valoarea $ult[i]$;
- dacă $bmax[i] = bmax[i-1]$, înseamnă că proiectul P_i nu a fost utilizat în planificarea optimă, deci trecem la următorul proiect P_{i-1}, decrementând valoarea indicelui i.

Se observă faptul că proiectele se vor afișa invers, deci trebuie utilizată o structură de date auxiliară pentru a le afișa în ordinea intervalelor în care trebuie executate!

Pentru exemplul dat, vom obține următoarele valori pentru elementele listelor ult și $bmax$ (informațiile despre proiectele $P_1, P_2, \dots, P_n$ ale echipei vor fi memorate într-o listă lp cu elemente de tip tuplu și sortare crescător în funcție de ziua de sfârșit):

i	0	1	2	3	4	5	6	7	8
lp	–	P_3 1 3	P_4 2 6	P_2 4 12	P_1 7 13	P_6 4 16	P_5 13 18	P_8 15 22	P_7 25 27
		250	650	800	850	900	1000	900	300
ult	–	0	0	1	2	1	4	4	7
bmax	0	250	650	1050	1500	1500	2500	2500	2800
		0 250	250 650	650 800+250	1050 850+650	1500 900+250	1500 1000+1500	2500 900+850	2500 300+2500

Valorile din lista $bmax$ sunt cele scrise cu roșu și au fost calculate ca fiind maximul dintre cele două valori scrise cu albastru, determinate folosind relația de recurență. De exemplu, $bmax[4] = \max\{bmax[3],\ bonus[4] + bmax[ult[4]]\} = \max\{1050,\ 850 + bmax[2]\} = \max\{1050,\ 850 + 650\} = 1500$.

Pentru exemplul considerat, bonusul maxim pe care îl poate obține echipa este $bmax[8] = 2800$ RON, iar pentru a reconstitui o planificare optimă vom utiliza informațiile din listele $bmax$ și ult, astfel:

- inițializăm un indice $i = n = 8$;

- $bmax[i] = bmax[8] = 2800 \neq bmax[i-1] = bmax[7] = 2500$, deci proiectul $lp[8] = P_7$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[8] = 7$;
- $bmax[i] = bmax[7] = 2500 = bmax[i-1] = bmax[6] = 2500$, deci proiectul $lp[7] = P_8$ nu a fost programat și indicele i devine $i = i - 1 = 6$;
- $bmax[i] = bmax[6] = 2500 \neq bmax[i-1] = bmax[5] = 1500$, deci proiectul $lp[6] = P_5$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[6] = 4$;
- $bmax[i] = bmax[4] = 1500 \neq bmax[i-1] = bmax[3] = 1050$, deci proiectul $lp[4] = P_1$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[4] = 2$;
- $bmax[i] = bmax[2] = 650 \neq bmax[i-1] = bmax[1] = 250$, deci proiectul $lp[2] = P_4$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[2] = 0$;
- $i = 0$, deci am terminat de afișat o modalitate optimă de planificare a proiectelor și ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că datele de intrare se citesc din fișierul text `proiecte.txt`, care conține pe fiecare linie informațiile despre câte un proiect, în ordinea denumire, ziua inițială, ziua finală și bonusul:

```
# funcție folosită pentru sortarea crescătoare a proiectelor
# în raport de data de sfârșit (cheia)
def cheieDataSfarsitProiect(t):
    return t[2]

f = open("proiecte.in")

# lp este lista proiectelor în care am adăugat un prim proiect
# "inexistent" pentru a avea proiectele indexate de la 1
lp = [("", 0, 0, 0)]
for linie in f:
    # un proiect = un tuplu (ID, data început, data sfârșit, profit)
    aux = linie.split()
    lp.append((aux[0], int(aux[1]), int(aux[2]), int(aux[3])))

f.close()

# n = numărul proiectelor
n = len(lp) - 1

# sortăm proiectele crescător dupa data de sfârșit
lp.sort(key=cheieDataSfarsitProiect)

# calculăm elementele listelor pmax și ult
pmax = [0] * (n + 1)
ult = [0] * (n + 1)
```

```
for i in range(1, n+1):
    for j in range(i-1, 0, -1):
        if lp[j][2] <= lp[i][1]:
            ult[i] = j
            break

    if lp[i][3] + pmax[ult[i]] > pmax[i-1]:
        pmax[i] = lp[i][3] + pmax[ult[i]]
    else:
        pmax[i] = pmax[i-1]

# reconstituim o soluție
i = n
sol = []
while i >= 1:
    if pmax[i] != pmax[i-1]:
        sol.append(lp[i])
        i = ult[i]
    else:
        i -= 1

# inversăm soluția obținută
sol.reverse()

# scriem soluția în fișierul text proiecte.out
fout = open("proiecte.out", "w")

for ps in sol:
    fout.write("{}: {:02d}-{:02d} -> {} RON\n".format(ps[0], ps[1],
                ps[2], ps[3]))

fout.write("\nBonusul echipei: " + str(pmax[n]) + " RON")

fout.close()
```

Complexitatea algoritmului prezentat este $\mathcal{O}(n^2)$ și poate fi scăzută la $\mathcal{O}(n \log_2 n)$ dacă utilizăm o căutare binară modificată pentru a calcula valoarea $ult[i]$: https://www.geeksforgeeks.org/weighted-job-scheduling-log-n-time/.

TEHNICA DE PROGRAMARE "BACKTRACKING"

1. Prezentare generală

Tehnica de programare Backtracking este utilizată, de obicei, pentru determinarea tuturor soluțiilor unei probleme într-un mod progresiv, astfel încât să se evite generarea întregului spațiu al soluțiilor problemei. Practic, soluțiile se construiesc componentă cu componentă și se testează la fiecare pas validitatea lor (se verifică dacă sunt *soluții parțiale*), iar în cazul în care se constată faptul că nu se poate obține o soluție a problemei plecând de la soluția parțială curentă, aceasta este abandonată. Astfel, se evită o *rezolvare de tip forță-brută* a problemei, adică generarea și testarea tuturor soluțiilor posibile pentru a determina soluțiile problemei. Totuși, complexitatea algoritmilor de tip Backtracking rămâne una ridicată, deoarece se va genera și testa un procent semnificativ din întregul spațiu al soluțiilor.

De exemplu, să considerăm problema generării tuturor permutărilor mulțimii $A = \{1,2,\dots,n\}$ pentru $n = 6$.

O rezolvare de tip forță-brută presupune generarea tuturor posibilelor soluții, adică a tuplurilor de forma $p = (p_1, p_2, p_3, p_4, p_5, p_6)$ cu $p_1, \dots, p_6 \in \{1,2,\dots,6\}$, și selectarea celor care sunt permutări, adică au toate valorile diferite între ele ($p_1 \neq p_2 \neq \dots \neq p_6$). Se observă foarte ușor faptul că se vor genera și testa $6^6 = 46656$ tupluri, din care doar $6! = 720$ vor fi permutări, deci eficiența acestei metode este foarte mică - în jurul unui procent de 1.5%! Eficiența scăzută a acestei metode este indusă de faptul că se generează multe tupluri inutile, care nu sunt sigur permutări. De exemplu, se vor genera toate tuplurile de forma $p = (1,1, p_3, p_4, p_5, p_6)$, adică $6^4 = 1296$ de tupluri inutile deoarece $p_1 = p_2$, deci, evident, aceste tupluri nu pot fi permutări!

O rezolvare de tip Backtracking presupune generare progresivă a soluțiilor, evitând generarea unor tupluri inutile, astfel (vom ține cont de faptul că $p_1, \dots, p_6 \in \{1,2,\dots,6\}$):

- $p = (1)$ – este o soluție parțială (componentele sale sunt diferite între ele, deci poate fi o permutare), dar nu este o soluție a problemei (nu are 6 componente), astfel că trebuie să adăugăm cel puțin încă o componentă;
- $p = (1,1)$ – nu este o soluție parțială (componentele sale sunt egale, deci nu vom obține o permutare indiferent de ce valori vom adăuga în continuare), astfel că nu are sens să adăugăm încă o componentă, ci vom genera următorul tuplu tot cu două componente;
- $p = (1,2)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left.\begin{matrix} p = (1,2,1) \\ p = (1,2,2) \end{matrix}\right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3)$ – este o soluție parțială, dar nu este o soluție a problemei;
- $\left.\begin{matrix} p = (1,2,3,1) \\ p = (1,2,3,2) \\ p = (1,2,3,3) \end{matrix}\right\}$ – nu sunt soluții parțiale;
- $p = (1,2,3,4)$ – este o soluție parțială, dar nu este o soluție a problemei;

- $\left.\begin{matrix} p=(1,2,3,4,1) \\ p=(1,2,3,4,2) \\ p=(1,2,3,4,3) \\ p=(1,2,3,4,4) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p=(1,2,3,4,5)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p=(1,2,3,4,5,1) \\ p=(1,2,3,4,5,2) \\ p=(1,2,3,4,5,3) \\ p=(1,2,3,4,5,4) \\ p=(1,2,3,4,5,5) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p=(1,2,3,4,5,6)$ – este o soluţie parţială care este şi soluţie a problemei, deci o memorăm sau o prelucrăm (de exemplu, o afişăm), după care vom încerca generarea următorul tuplu. Deoarece ultima componentă are valoarea 6 (ultima posibilă), înseamnă că am epuizat toate valorile posibile pentru aceasta, deci o vom elimina şi vom genera următorul tuplu format doar din 5 componente;
- $p=(1,2,3,4,6)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p=(1,2,3,4,6,1) \\ p=(1,2,3,4,6,2) \\ p=(1,2,3,4,6,3) \\ p=(1,2,3,4,6,4) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p=(1,2,3,4,6,5)$ – este o soluţie parţială care este şi soluţie a problemei, deci o memorăm sau o prelucrăm, după care vom genera următorul tuplu;
- $p=(1,2,3,4,6,6)$ – nu este o soluţie parţială, dar ultima componentă are valoarea maxim posibilă 6, deci o vom elimina şi vom genera următorul tuplu format doar din 5 componente;
- $p=(1,2,3,4,6)$ – ultima componentă are valoarea maxim posibilă 6, deci o vom elimina şi vom genera următorul tuplu format doar din 4 componente;
- $p=(1,2,3,5)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- .....
- $p=(6,5,4,3,2,1)$ – este o soluţie parţială care este şi ultima soluţie a problemei, deci o memorăm sau o prelucrăm, după care vom încerca generarea următorului tuplu. Se observă cu uşurinţă faptul că, pe rând, nu vom mai găsi niciun tuplu convenabil, deci algoritmul se va termina.

Observaţie importantă: Determinarea exactă a complexităţii unui algoritm de tip Backtracking nu este simplă. Totuşi, plecând de la observaţia că un algoritm nu poate avea o complexitate mai mică decât citirea datelor de intrare şi/sau afişarea datelor de ieşire, de obicei, putem aproxima complexitatea unui astfel de algoritm prin numărul soluţiilor pe care el le afişează. De exemplu, algoritmul pentru generarea permutărilor de ordin n are complexitatea minimă egală cu $O(n!)$, deoarece vor fi afişate $n!$ permutări. O astfel de complexitate este foarte mare, depăşind-o pe cea exponenţială!

2. Forma generală a unui algoritm de tip Backtracking

Vom începe prin a preciza faptul că majoritatea problemele de generare pot fi formalizate astfel: "*Fie mulțimile nevide* $A_1, A_2, \dots, A_n$ *și un predicat* $P: A_1 \times \dots \times A_n \to \{0,1\}$. *Să se genereze toate tuplurile de forma* $S = (s_1, s_2, \dots, s_n)$ *pentru care* $s_1 \in A_1, \dots, s_n \in A_n$ *și* $P(s_1, s_2, \dots, s_n) = 1$.". Practic, predicatul P cuantifică o proprietate pe care trebuie să o îndeplinească componentele tuplului S (o soluție a problemei) sub forma unei funcții de tip boolean (0 = *fals* și 1 = *adevărat*).

De exemplu, problema generării tuturor permutărilor poate fi formalizată în acest mod considerând $A_1 = \dots = A_n = \{1,2, \dots, n\}$ și $P(s_1, s_2, \dots, s_n) = (s_1 \neq s_2)\ AND\ (s_2 \neq s_3)\ AND \dots AND\ (s_{n-1} \neq s_n)$.

În continuare, vom prezenta câteva notații, definiții și observații:

- pentru orice componentă s_k vom nota cu min_k cea mai mică valoare posibilă a sa, iar cu max_k pe cea mai mare;
- într-un tuplu $(s_1, s_2, \dots, s_k)$, componenta s_k se numește *componentă curentă* (i.e., asupra sa se acționează în momentul respectiv);
- *condițiile de continuare* reprezintă condițiile pe care trebuie să le îndeplinească tuplul curent $(s_1, s_2, \dots, s_k)$ astfel încât să aibă sens extinderea sa cu o nouă componentă s_{k+1} sau, altfel spus, există valori pe care le putem adăuga la el astfel încât să obținem o soluție $S = (s_1, \dots, s_n)$ a problemei;
- tuplul curent $(s_1, \dots, s_k)$ este o *soluție parțială* dacă îndeplinește condițiile de continuare;
- *condițiile de continuare* se deduc din predicatul P și sunt neapărat necesare, fără a fi întotdeauna și suficiente;
- orice *soluție* a problemei este implicit și soluție parțială, dar trebuie să mai îndeplinească și alte condiții suplimentare.

Folosind observațiile anterioare, forma generală a unui algoritm de tip Backtracking, implementat folosind o funcție recursivă este următoarea:

```
# k reprezintă indicele componentei curente s[k]
# dintr-o listă s indexată de la 1
def bkt(k):
    global s
    # parcurgem toate valorile posibile v pentru s[k]
    for v in range(mink, maxk+1):
        # atribuim componentei curente s[k] valoarea v
        s[k] = v

        # dacă s[1],…,s[k] este soluție parțială
        if s[1],…,s[k] este soluție parțială:
            # dacă s[1],…,s[k] este o soluție
            if s[1],…,s[k] este soluție:
                # prelucrăm soluția curentă s[1],…,s[k]
            else:
               # s[1],…,s[k] este soluție parțială, dar nu este
               # soluție, deci adăugăm o nouă componentă s[k + 1]
               bkt(k+1)
```

Referitor la algoritmul general de Backtracking prezentat mai sus trebuie făcute câteva observații:

- bucățile de cod scrise cu roșu trebuie particularizate pentru fiecare problemă;
- am considerat tabloul `s` indexat de la 1, ci nu de la 0, pentru a permite o scriere naturală a unor condiții în care se utilizează indicii tabloului;
- $\texttt{min}_k$ și $\texttt{max}_k$ se deduc din semnificația componentei `s[k]` a unei soluții;
- pentru a testa că `s[1],…,s[k]` este soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar pe cele suplimentare lor;
- dacă `s[1],…,s[k]` nu este soluție parțială, atunci nu vom adăuga o nouă componentă `s[k+1]` prin apelul recursiv `bkt(k+1)`, deci instrucțiunea `for` va continua și componentei curente `s[k]` i se va atribui următoarea valoare posibilă `v`, dacă aceasta există, iar în cazul în care aceasta nu există, instrucțiunea `for` corespunzătoare componentei curente `s[k]` se va termina și, implicit, apelul funcției `bkt` corespunzător, deci se va reveni la componenta anterioară `s[k-1]`.

De exemplu, pentru a genera toate permutările de ordin n, observațiile de mai sus se particularizează, astfel:

- `s[k]` reprezintă un element al permutării, deci $\texttt{min}_k$ `= 1` și $\texttt{max}_k$ `= n`;
- `s[1],…,s[k-1],s[k]` este soluție parțială dacă valoarea componentei curente `s[k]` nu a mai fost utilizată anterior, adică `s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`. Se observă faptul că această condiție este dedusă din predicatul P (care impune ca toate cele n valori `s[1],…,s[n]` dintr-o permutare de ordin n să fie distincte) și este neapărat necesară (dacă `s[1],…,s[k]` nu sunt distincte, atunci, indiferent de ce valori am atribui celorlalte `n-k` componente `s[k+1],…,s[n]` nu vom obține o permutare de ordin `n`), fără a fi și suficientă (dacă valorile `s[1],…,s[k]` sunt distincte nu înseamnă că ele formează o permutare de ordin `n`, ci trebuie impusă condiția suplimentară `k=n`);
- pentru a testa că `s[1],…,s[k]` este soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare (`s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`), ci doar condiția suplimentară `k==n`.

Aplicând observațiile anterioare în forma generală a algoritmului de Backtracking, obținem următoarea implementare în limbajul Python a algoritmului de generare a tuturor permutărilor mulțimii $\{1,2,\dots,n\}$:

```
def bkt(k):
    global s, n

    for v in range(1, n+1):
        s[k] = v
        if s[k] not in s[:k]:
            if k == n:
                print(*s[1:], sep=",")
            else:
                bkt(k+1)
```

```
n = int(input("n = "))
# o soluție s va avea n elemente
s = [0]*(n+1)
print("Toate permutările de lungime " + str(n) + ":")
bkt(1)
```

Aşa cum am menţionat deja, complexitatea minimă a acestui algoritm este $\mathcal{O}(n!)$.

3. Probleme de generări combinatoriale

Pe lângă generarea permutărilor unei mulţimi, algoritmii de tip Backtracking mai pot fi utilizaţi şi pentru rezolvarea altor probleme de generări combinatoriale, cum ar fi generarea aranjamentelor sau a combinărilor unei mulţimi. În continuare, vom exemplifica algoritmii pe care îi vom prezenta pentru mulţimea $A = \{1,2, \dots, n\}$, deoarece elementele oricărei alte mulţimi cu n elemente pot fi accesate considerând elementele mulţimii A ca fiind indicii elementelor sale.

3.1. Generarea aranjamentelor

Aranjamentele cu m elemente ale unei mulţimi cu n elemente ($m \leq n$) reprezintă toate tuplurile care se pot forma utilizând m elemente distincte dintre cele n ale mulţimii. Numărul lor se notează cu A_n^m şi este dat de formula $\frac{n!}{(n-m)!}$. De exemplu, numărul aranjamentelor cu $m = 3$ elemente ale unei mulţimi cu $n = 5$ elemente este $A_5^3 = 60$, o parte a lor fiind: (1,2,3), (1,3,2), … , (3,2,1), … , (1,3,5), … , (5,3,1), … , (3,4,5), … , (5,4,3).

Se observă foarte uşor faptul că singura diferenţă faţă de generarea permutărilor o constituie lungimea unei soluţii, care în acest caz este m în loc de n. De fapt, pentru $m = n$, aranjamentele unei mulţimi sunt chiar permutările sale!

La fel ca şi în cazul generării permutărilor, putem aproxima complexitatea minimă a acestui algoritm prin numărul soluţiilor pe care le va afişa, deci prin $\mathcal{O}(A_n^m)$.

3.2. Generarea combinărilor

Combinările cu m elemente ale unei mulţimi cu n elemente ($m \leq n$) reprezintă toate submulţimile cu m elemente ale unei mulţimi cu n elemente. Numărul lor se notează cu C_n^m şi este dat de formula $\frac{n!}{m!(n-m)!}$. De exemplu, numărul tuturor submulţimilor cu $m = 3$ elemente ale unei mulţimi cu $n = 5$ elemente este $C_5^3 = 10$, toate aceste submulţimi fiind: $\{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\}, \{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}$ şi $\{3,4,5\}$.

Spre deosebire de tupluri, în care contează ordinea elementelor (de exemplu, tuplurile (1,2,3) şi (1,3,2) sunt considerate diferite), în cazul submulţimilor aceasta nu contează (de exemplu, submulţimile $\{1,2,3\}$ şi $\{3,1,2\}$ sunt considerate egale). Din acest motiv, trebuie să găsim o posibilitate de a evita prelucrarea unei soluţii care are aceleaşi elemente ca o altă soluţie generată anterior, dar în altă ordine. În acest sens, o variantă simplă, dar ineficientă, o reprezintă prelucrarea doar a soluţiilor care au elementele în ordine strict crescătoare, dar astfel vom încălca chiar principiul de bază al metodei Backtracking, acela de a evita generarea şi testarea unor tupluri inutile cât mai devreme posibil. O altă variantă o reprezintă generarea doar a soluţiilor cu elemente în ordine

strict crescătoare, această restricție putând fi impusă soluției curente în mai multe etape ale unui algoritm de tip Backtracking:

- *când testăm condițiile de continuare, verificând faptul că* `s[k]> s[k-1]` – această variantă este mai eficientă decât prima, dar, totuși se vor genera și testa multe tupluri inutile. De exemplu, pentru a extinde soluția parțială (1, 3), se vor genera și testa inutil tuplurile (1,3,1), (1,3,2) și (1,3,3), deși este evident faptul că la tuplul (1, 3) are sens să adăugăm doar o valoare cel puțin egală cu 4;
- *inițializând componenta curentă cu prima valoare strict mai mare decât componenta anterioară* ($\texttt{min}_k$ `= s[k-1]+1`) – este cea mai eficientă variantă posibilă, deoarece nu se generează și testează tupluri inutile și, mai mult, orice tuplu este soluție parțială (elementele sale sunt generate direct în ordine strict crescătoare, deci sunt distincte), ceea ce înseamnă că putem renunța la testarea condițiilor de continuare!

Astfel, vom obține următorul program eficient de tip Backtracking pentru generarea combinărilor:

```
def bkt(k):
    global n, m, sol, cnt

    for v in range(sol[k-1]+1, n+1):
        sol[k] = v
        if k == m:
            cnt += 1
            print(str(cnt).rjust(3) + ". ", end="")
            print(*sol[1:], sep=",")
        else:
            bkt(k+1)


n = int(input("n = "))
m = int(input("m = "))
# contor pentru soluțiile generate
cnt = 0
# o soluție va avea lungimea m
sol = [0] * (m+1)
print("Toate submulțimile cu ", m, "elemente ale unei mulțimi cu ",
      n, "elemente")
bkt(1)
```

Pentru `k=1`, variabila `v` din ciclul `for` va fi inițializată cu valoarea `s[0]+1`, respectiv chiar cu valoarea corectă `1`, deoarece `s[0]` are valoarea `0`.

La fel ca și în cazul generării permutărilor, putem aproxima complexitatea minimă a acestui algoritm prin numărul soluțiilor pe care le va afișa, deci prin $\mathcal{O}(C_n^m)$.

Problemele de generare prezentate pot fi utilizate și pentru a genera permutările/ aranjamentele/combinările unei colecții oarecare, considerând elementele mulțimii $\{1, 2, \ldots, n\}$ ca fiind pozițiile elementelor colecției respective!

De exemplu, pentru a genera toate anagramele distincte ale unui cuvânt, vom genera toate permutările de lungime egală cu lungimea cuvântului, pentru fiecare permutare vom reconstitui cuvântul asociat și îl vom salva într-o mulțime (i.e., colecție de tip `set`):

```
def bkt(k):
    global s, n, cuv, cuv_dist

    for v in range(1, n+1):
        s[k] = v
        if s[k] not in s[1:k]:
            if k == n:
                aux = "".join([cuv[s[i]-1] for i in range(1, n+1)])
                cuv_dist.add(aux)
            else:
               bkt(k+1)

cuv = input("Cuvantul: ")
n = len(cuv)
cuv_dist = set()
s = [0] * (n+1)
bkt(1)
print("Anagramele distincte ale cuvântului " + cuv + ": ")
print(*cuv_dist, sep="\n")
```

4. Descompunerea unui număr natural ca sumă de numere naturale nenule

Această problemă apare destul de des în practică, în diverse forme: împărțirea unui produs dintr-un depozit între mai multe magazine de desfacere, distribuirea unui sume de bani (un buget) între mai multe firme sau persoane fizice, partiționarea unui teren între mai mulți cumpărători etc.

De exemplu, numărul natural `n=4` poate fi descompus ca sumă de numere naturale nenule, astfel: `1+1+1+1`, `1+1+2`, `1+2+1`, `1+3`, `2+1+1`, `2+2`, `3+1` și `4`. Restricția ca termenii sumei să fie numere naturale nenule este esențială, altfel problema ar avea o infinitate de soluții!

În cazul acestei probleme, observațiile de la forma generală a unui algoritm de tip Backtracking pot fi particularizate astfel :

- `s[k]` reprezintă un termen al sumei, deci $\texttt{min}_k=1$ și $\texttt{max}_k=\texttt{n-k+1}$ (în momentul în care componenta curentă este `s[k]`, celelalte `k-1` componente anterioare `s[1],…,s[k-1]` au, fiecare, cel puțin valoarea `1`, deci `s[k]` nu poate să depășească valoarea `n-(k-1)=n-k+1` deoarece atunci suma `s[1]+…+s[k]` ar fi strict mai mare decât `n`);
- soluțiile problemei nu mai au toate lungimi egale, ci ele variază de la `1` la `n`;
- `s[1],…,s[k]` este soluție parțială dacă `s[1]+…+s[k]<=n`. Se observă faptul că această condiție este dedusă din predicatul *P* (care impune `s[1]+…+s[k]=n`) și este neapărat necesară (dacă `s[1]+…+s[k]>n` atunci, indiferent de ce numere naturale nenule `s[k+1],s[k+2],…` am mai adăuga (inclusiv niciunul!), nu vom mai putea obține `s[1]+…+s[k]==n`), fără însă a fi și suficientă (dacă `s[1]+…+s[k]<=n` nu înseamnă obligatoriu că `s[1]+…+s[k]==n`);
- `s[1],…,s[k]` este soluție dacă `s[1]+…+s[k]==n`.

Aplicând observaţiile anterioare în forma generală a algoritmului de Backtracking, obţinem următoarea implementare a acestui algoritm în limbajul Python:

```
def bkt(k):
    global sol, n

    for v in range(1, n-k+2):
        sol[k] = v
        scrt = sum(sol[:k+1])
        if scrt <= n:
            if scrt == n:
                print(*sol[1: k+1], sep="+")
            else:
                bkt(k+1)


n = int(input("n = "))
sol = [0]*(n+1)
bkt(1)
```

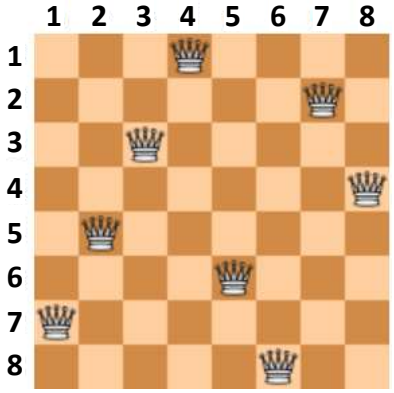
În practică, această problema apare în multe variante, câteva dintre ele fiind următoarele (în toate exemplele am considerat n=4):

- *descompuneri distincte* (care nu conţin aceiaşi termeni, dar în altă ordine): 1+1+1+1, 1+1+2, 1+3, 2+2 şi 4;
- *descompuneri cu termeni distincţi* (care nu conţin termeni egali): 1+3, 3+1 şi 4;
- *descompuneri distincte cu termeni distincţi*: 1+3 şi 4;
- *descompuneri ale căror lungimi verifică anumite condiţii* (de exemplu, descompuneri de lungime egală cu 3: 1+1+2, 1+2+1 şi 2+1+1);
- *descompuneri ale căror termeni verifică anumite condiţii* (de exemplu, descompuneri cu toţi termenii numere pare: 2+2 şi 4);
- *descompuneri care verifică simultan mai multe dintre condiţiile de mai sus*.

Complexitatea acestui algoritm poate fi estimată doar folosind cunoştinţe avansate de teoria numerelor pentru a aproxima numărul soluţiilor pe care le va afişa (https://en.wikipedia.org/wiki/Partition_function_(number_theory)). Astfel, se poate demonstra faptul că acest algoritm are o complexitate de tip exponenţial (de exemplu, numărul $n = 1000$ are aproximativ $24061467864032622473692149727991 \approx 2.4 \times 10^{31}$ descompuneri distincte!).

5. Problema celor n regine

Fiind dată o tablă de şah de dimensiune $n \times n$, problema cere să se determine toate modurile în care pot fi plasate n regine pe tablă astfel încât oricare două să nu se atace între ele. Două regine se atacă pe tabla de şah dacă se află pe aceeaşi linie, coloană sau diagonală. De exemplu, pentru $n = 8$, o posibilă soluţie dintre cele 92 existente, este următoarea (sursa: https://en.wikipedia.org/wiki/Eight_queens_puzzle):



Problema a fost formulată de către creatorul de probleme şahistice Max Bezzel în 1848 pentru $n = 8$. În 1850 Franz Nauck a publicat primele soluţii ale problemei şi a generalizat-o pentru orice număr natural $n \geq 4$ (pentru $n \leq 3$ problema nu are soluţii), ulterior ea fiind analizată de mai mulţi matematicieni (e.g., C. F. Gauss) şi informaticieni (e.g., E.W. Dijkstra) celebri. Problema poate fi rezolvată prin mai multe metode, astfel:

- considerând reginele numerotate de la 1 la n^2, generăm, pe rând, toate cele $C_{n^2}^n$ submulţimi formate n regine şi apoi le testăm (de exemplu, pentru $n = 8$ vom genera şi testa $C_{64}^8 = 4426165368$ submulţimi). Evident, această metodă de tip forţă-brută este foarte ineficientă, deoarece vom genera şi testa inutil foarte multe submulţimi care sigur nu pot fi soluţii (de exemplu, toate submulţimile care cuprind cel puţin două regine pe aceeaşi linie, coloană sau diagonală);
- observând faptul că pe o linie se poate poziţiona exact o regină, vom genera, pe rând, toate cele n^n tupluri conţinând coloanele pe care se află reginele de pe fiecare linie şi le vom testa (de exemplu, pentru $n = 8$ vom genera şi testa $8^8 = 16777216$ tupluri). Deşi această metodă, tot de tip forţă-brută, este de aproximativ 260 de ori mai rapidă decât precedenta, tot va genera şi testa inutil multe tupluri care nu pot fi soluţii (de exemplu, toate tuplurile care conţin cel puţin două valori egale, deoarece acest lucru înseamnă faptul că mai mult de două regine se află pe aceeaşi coloană, deci se atacă între ele);
- observând faptul că pe o linie şi o coloană se poate poziţiona exact o regină, vom genera, pe rând, utilizând metoda Backtracking, toate cele $n!$ permutări cu n elemente, testând la fiecare pas şi condiţia ca reginele să nu se atace pe diagonală (de exemplu, pentru $n = 8$ vom genera şi testa $8! = 40320$ permutări). Evident, această metodă este mult mai eficientă decât primele două, fiind de aproximativ 420 de ori mai rapidă decât a doua metodă şi de peste 110000 de ori decât prima!

În continuare, vom detalia puţin cea de-a treia variantă de rezolvare prezentată mai sus, bazată pe metoda Backtracking. Revenind la observaţiile generale de la metoda Backtracking, acestea se vor particulariza, astfel:

- `s[k]` reprezintă coloană pe care este poziţionată regina de pe linia `k`. De exemplu, soluţiei prezentate în figura de mai sus îi corespunde tuplul $s = (4,7,3,8,2,5,1,6)$;
- deoarece pe o linie `k` regina poate fi poziţionată pe orice coloană `s[k]` cuprinsă între `1` şi `n`, obţinem $\texttt{min}_k=1$ şi $\texttt{max}_k=\texttt{n}$;
- `s[1],…,s[k-1],s[k]` este soluţie parţială dacă regina curentă R_k`(k,s[k])`, adică regina aflată pe linia `k` şi coloana `s[k]`, nu se atacă pe coloană sau diagonală cu nicio regină anterior poziţionată pe o linie `i` şi o coloană `s[i]`, pentru orice `i`$\in$`{1,…,k-1}`.

Condiţia referitoare la coloană se deduce imediat, respectiv trebuie ca `s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`. Condiţia referitoare la diagonală se poate deduce, de exemplu, astfel: regina $R_k(k, s[k])$ se atacă pe diagonală cu o altă regină $R_i(i, s[i])$ dacă şi numai dacă dreapta R_kR_i este paralelă cu una dintre cele două diagonale ale tablei de şah. Două drepte sunt paralele dacă şi numai dacă au pantele egale, iar cele două diagonale au pantele egale cu $\operatorname{tg} 45° = 1$ şi $\operatorname{tg} 135° = -1$, deci panta dreptei R_kR_i trebuie să fie diferită de ± 1, deci $m_{R_kR_i} = \frac{s[k]-s[i]}{k-i} \neq \pm 1$. Aplicând funcţia modul, obţinem $\left|\frac{s[k]-s[i]}{k-i}\right| \neq 1$ sau, echivalent, $|s[k] - s[i]| \neq |k - i|$ pentru orice $i \in \{1,2, \dots, k-1\}$;

- pentru a testa că `s[1],…,s[k]` este soluţie vom ţine cont de faptul că `s[1],…,s[k]` este soluţie parţială, deci nu vom retesta condiţiile de continuare, ci doar condiţia suplimentară `k==n`.

Practic, se observă faptul că problema celor n regine se reduce la generarea permutărilor de ordin n care verifică şi condiţia referitoare la diagonale, deci putem utiliza direct algoritmul de generare a permutărilor în care modificăm doar condiţia de continuare, astfel:

```
if s[k] not in s[:k] and \
  True not in [abs(k - i) == abs(s[k] - s[i]) for i in range(1, k)]:
```

Complexitatea algoritmului de tip Backtracking de mai sus poate fi aproximată prin $\mathcal{O}(n!)$, fiind o variantă puţin modificată a algoritmului de generare a permutărilor de ordin n.

6. Problema plăţii unei sume folosind monede cu valori date

Considerând faptul că avem la dispoziţie n monede cu valorile $v_1, v_2, \dots, v_n$ pe care putem să le folosim pentru a plăti o sumă P, trebuie să determinăm toate modalităţile în care putem realiza acest lucru (vom presupune faptul că avem la dispoziţie un număr suficient de monede de fiecare tip).

Exemplu: Dacă avem la dispoziţie $n = 3$ tipuri de monede cu valorile $v = (2\$, 3\$, 5\$)$, atunci putem să plătim suma $P = 12\$$ în următoarele 5 moduri: $4 \times 3\$$, $1 \times 2\$ + 2 \times 5\$$, $2 \times 2\$ + 1 \times 3\$ + 1 \times 5\$$, $3 \times 2\$ + 2 \times 3\$$ şi $6 \times 2\$$.

Pentru a rezolva această problemă vom particulariza algoritmul generic de Bactracking, astfel:

- `s[k]` reprezintă numărul de monede cu valoarea `v[k]` utilizate pentru plata sumei `P`, deci obţinem $\texttt{min}_k$=`0` şi $\texttt{max}_k$=`P/v[k]`;
- `s[1],…,s[k-1],s[k]` este soluţie parţială dacă suma curentă este cel mult egală cu suma de plată `P`, adică `s[1]*v[1]+…+s[k]*v[k]<=P`;
- `s[1],…,s[k]` este soluţie dacă suma curentă este egală cu suma de plată `P`, adică `s[1]*v[1]+…+s[k]*v[k]==P` (se observă faptul că soluţiile au lungimi variabile, cuprinse între `1` şi `n`);

- deoarece problema nu are întotdeauna soluție (de exemplu, dacă toate monedele date au valori pare și suma de plată este impară), vom adăuga o variabila `nrs` care să contorizeze numărul soluțiilor găsite, iar după terminarea algoritmului vom verifica dacă problema a avut cel puțin o soluție sau nu.

Observație: Deoarece $\texttt{min}_k\texttt{=0}$, înseamnă că tabloul s va conține, pe rând, valorile $(0), (0,0), \ldots, (\underbrace{0,0,\ldots,0}_{\text{de } n \text{ ori}})$, pentru că, la fiecare pas, va fi verificată condiția de continuare de mai sus (`0*v[1]+...+0*v[k]=0<=P`). Din acest motiv, trebuie să limităm extinderea tabloului la mai mult de n elemente pentru a evita erorile accesare a memoriei, deci vom efectua apelul recursiv `bkt(k+1)` doar în cazul în care `k<n`!

În continuare prezentăm implementarea algoritmului în limbajul Python:

```
def bkt(k):
    global s, P, v, n

    # s[k] = numarul de monede cu valoarea v[k] utilizate
    for m in range(0, P // v[k] + 1):
        s[k] = m
        scrt = sum([s[i] * v[i] for i in range(k+1)])
        if scrt <= P:
            if scrt == P and k == n:
                for i in range(1, n+1):
                    if s[i] != 0:
                        print(s[i], "x", v[i], "$ + ", end="")
                print()
            else:
                if k < len(v[1:]):
                    bkt(k+1)


P = int(input("Suma de plată: "))
aux = [int(x) for x in input("Valorile monedelor: ").split()]
v = [0]
v.extend(aux)
n = len(v[1:])
s = [0]*(len(v))
print("Toate modalitățile de plată:")
bkt(1)
```

Observație: Deoarece $\texttt{min}_k\texttt{=0}$, înseamnă că tabloul s va conține, pe rând, valorile $(0), (0,0), \ldots, (\underbrace{0,0,\ldots,0}_{\text{de } n \text{ ori}})$, pentru că, la fiecare pas, va fi verificată condiția de continuare de mai sus (`0*v[1]+...+0*v[k]=0<=P`). Din acest motiv, trebuie să limităm extinderea tabloului la mai mult de n elemente pentru a evita erorile accesare a memoriei, deci vom efectua apelul recursiv `bkt(k+1)` doar în cazul în care `k<n`!

Complexitatea acestui algoritmului poate fi aproximată prin numărul maxim de tupluri care pot fi generate şi testate, respectiv $\frac{P}{v_1} \cdot \frac{P}{v_2} \cdot \ldots \cdot \frac{P}{v_n}$. În cazul unor date de intrare "reale", putem presupune faptul ca valorile monedelor $v_1, v_2, \ldots, v_n$ sunt cel mult egale cu suma P (o monedă cu o valoare strict mai mare decât P este inutilă, deci ar putea fi eliminată din datele de intrare), astfel încât fiecare raport $\frac{P}{v_k}$ va fi mai mare sau egal decât 1. De fapt, în realitate, valorile monedelor $v_1, v_2, \ldots, v_n$ sunt mult mai mici decât suma de plată P, deci fiecare raport $\frac{P}{v_k}$ va fi, în general, mai mare sau egal decât 2, deci complexitatea algoritmului poate fi aproximată prin $\mathcal{O}(2^n)$.

Observaţie: Metoda Backtracking poate fi modificată astfel încât să fie utilizată şi pentru rezolvarea altor tipuri de probleme, în afara celor de generare a tuturor soluţiilor, astfel:

- *pentru probleme de numărare*: se generează toate soluţiile posibile şi se înlocuieşte secţiunea pentru afişarea unei soluţii cu o simplă incrementare a unui contor, iar după terminarea algoritmului se afişează valoarea contorului. Atenţie, de multe ori, problemele de numărare se pot rezolva mult mai eficient, fie utilizând o formulă matematică (de exemplu, numărul permutărilor p de ordin n fără puncte fixe, i.e. $p[k] \neq k, \forall k \in \{1, \ldots, n\}$, poate fi calculat folosind o formulă: https://en.wikipedia.org/wiki/Derangement), fie utilizând alte tehnici de programare (de exemplu, numărul modalităţilor de plată a unei sume folosind monede cu valori date se poate calcula cu un algoritm care utilizează metoda programării dinamice şi are complexitatea $\mathcal{O}(n^2)$: https://www.geeksforgeeks.org/coin-change-dp-7/).

- *pentru probleme de decizie*: într-o problemă de decizie ne interesează doar faptul că o problemă are soluţie sau nu (de exemplu, problema plăţii unei sume folosind monede cu valori date poate fi transformată într-o problemă de decizie, astfel: "*Să se verifice dacă o sumă de bani P poate fi plătită utilizând monede cu valorile* $v_1, v_2, \ldots, v_n$."), deci fie vom opri forţat algoritmul Backtracking în momentul în care găsim prima soluţie, fie acesta se va termina normal în cazul în care problema nu are soluţie. Şi în acest caz, de obicei, există algoritmi mai eficienţi, care utilizează alte tehnici de programare (de exemplu, pentru a verifica dacă o sumă de bani poate fi plătită folosind anumite monede se poate utiliza algoritmul menţionat anterior, având complexitatea $\mathcal{O}(n^2)$).

- *pentru probleme de optimizare*: într-o problemă de optimizare trebuie să găsim, de obicei, o singură soluţie care, în plus, minimizează sau maximizează o anumită expresie matematică (de exemplu, se poate cere determinarea unei modalităţi de plată a unei sume folosind un număr minim de monede cu valori date). În acest caz, vom genera toate soluţiile problemei şi vom reţine, într-o structură de date auxiliară, o soluţie optimă. În cazul acestor probleme există, de obicei, algoritmi mai eficienţi, care utilizează alte tehnici de programare, cum ar fi metoda Greedy sau metoda programării dinamice. De exemplu, pentru a determinarea o modalitate de plată a unei sume folosind un număr minim de monede, există un algoritm cu complexitatea $\mathcal{O}(n \cdot P)$ bazat pe metoda programării dinamice: https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/.

În încheiere, precizăm faptul că, în soluţiile problemelor pe care le-am prezentat, am dorit să accentuăm aspecte generale prin care algoritmul generic de Backtracking poate

fi particularizat pentru a rezolva diverse tipuri de probleme, neinsistând asupra unor modalităţi particulare de optimizare a lor, cum ar fi găsirea unui interval de valori cât mai mic pentru o componentă a soluției (i.e., diferența max_k-min_k să fie minimă), utilizarea unor structuri de date auxiliare pentru a marca valorile deja utilizate (de exemplu, în algoritmul de generare a permutărilor se poate utiliza un vector de marcaje pentru a verifica direct dacă o anumită valoare a fost deja utilizată) sau actualizarea dinamică a unor valori necesare în verificarea condițiilor de continuare (de exemplu, în algoritmul pentru descompunerea unui număr natural ca sumă de numere naturale nenule se poate actualiza dinamic suma curentă, în momentele în care se adaugă la o soluție parţială o nouă componentă, se modifică valoarea componentei curente sau se renunţă la componenta curentă). Din punctul nostru de vedere, aceste optimizări complică destul de mult codul sursă şi nu sunt foarte utile, deoarece, oricum, algoritmii de tip Backtracking au un timp de executare acceptabil doar pentru dimensiuni mici ale datelor de intrare.