

## SEMINAR 7

## Metoda programării dinamice. Metoda Backtracking

1. Se consideră  $n$  perechi  $(x, y)$  cu proprietatea că  $x < y$ . Să se determine lungimea maximă  $k$  a unui lanț de perechi de forma  $(x_1, y_1), \dots, (x_i, y_i), (x_{i+1}, y_{i+1}), \dots, (x_k, y_k)$  cu proprietatea că  $y_i < x_{i+1}$  pentru orice  $i \in \{1, 2, \dots, n-1\}$ .

**Exemplu:**

Pentru  $n = 6$  și perechile  $(12, 15), (6, 8), (5, 7), (20, 30), (9, 11), (13, 18)$ , lungimea maximă  $k$  a unui lanț cu proprietatea cerută este egală cu 4, iar un posibil astfel de lanț este  $(5, 7), (9, 11), (12, 15), (20, 30)$ . Atenție, perechile pot fi selectate în orice ordine!

**Rezolvare:**

Determinarea lungimii maxime a unui lanț de perechi având proprietatea cerută se poate realiza folosind metoda programării dinamice, respectiv o variantă modificată a algoritmului pentru determinarea subșirului crescător maximal, dar aplicată asupra șirului de perechi după sortarea acestora în ordinea crescătoare a valorilor componentelor secunde (i.e., valoarea  $y_i$ ):

```
# datele de intrare se citesc din fisierul text perechi.txt
# care conține pe fiecare linie câte o pereche de numere
f = open("perechi.txt")
# lp = lista în care vor fi memorate perechile
lp = []
for linie in f:
    aux = linie.split()
    lp.append((int(aux[0]), int(aux[1])))
f.close()

# n = numărul de perechi
n = len(lp)

# sortăm perechile în ordinea crescătoare a componentelor secunde
lp.sort(key=lambda k: k[1])

# modificăm algoritmul pentru determinarea
# subșirului crescător maximal
pred = [-1 for i in range(n)]
lmax = [1 for i in range(n)]

for i in range(n):
    for j in range(i):
        if lp[j][1] < lp[i][0] and lmax[j] + 1 > lmax[i]:
            pred[i] = j
            lmax[i] = lmax[j] + 1
```

```

# determinăm poziția pmax a maximului din lmax
pozmax = lmax.index(max(lmax))

print("Lungimea maximă a unui lanț de perechi:", lmax[pozmax])

# construim un lanț maximal în lista lantmax
lantmax = []
pozcrt = pozmax
while pozcrt != -1:
    lantmax.append(lp[pozcrt])
    pozcrt = pred[pozcrt]

# lanțul a fost reconstituit în ordine inversă,
# deci afișăm lista lantmax inversată
print("Un lanț maximal de perechi:")
print(*lantmax[::-1], sep=", ")

```

Complexitatea algoritmului prezentat este  $O(n^2)$ . Se observă foarte ușor faptul că problema este, de fapt, o reformulare a problemei programării într-o singură sală a unui număr maxim de spectacole care să nu se suprapună (i.e., o pereche  $(x_i, y_i)$  poate fi considerată intervalul de desfășurare a unui spectacol), deci poate fi rezolvată folosind algoritmul Greedy deja prezentat și care are complexitatea  $O(n \log_2 n)$ !

## 2. Partiționarea unei multiset în două submultiseturi cu sume cât mai apropiate

Considerăm un multiset  $A = \{a_1, a_2, \dots, a_n\}$  format din  $n$  numere naturale nenule ( $n \geq 2$ ). Să se determine o modalitate de partiționare a multisetului  $A$  în două submultiseturi  $X$  și  $Y$  (i.e.,  $X, Y \subseteq A, X \cup Y = A$  și  $X \cap Y = \emptyset$ ) astfel încât valoarea absolută a diferenței dintre suma elementelor submultisetului  $X$  și suma elementelor submultisetului  $Y$  să fie minimă. Un *multiset* este o extindere a conceptului de mulțime, respectiv elementele unui multiset nu trebuie să mai fie distincte, ci se pot repeta.

### Exemplu:

Multisetul  $A = \{4, 6, 4, 6, 15\}$  poate fi partiționat în submultiseturile  $X = \{4, 6, 6\}$  și  $Y = \{4, 15\}$  astfel încât să se obțină valoarea minimă posibilă pentru diferența dintre sumele elementelor a două submultiseturi de partiție, respectiv 3. Atenție, cele două submultiseturi sunt disjuncte din punct de vedere al elementelor utilizate din multisetul  $A$ , ci nu din punct de vedere al valorilor acestor elemente, o observație asemănătoare fiind valabilă și pentru reuniunea lor! Astfel, putem considera  $X = \{a_1, a_2, a_4\}$  și  $Y = \{a_3, a_5\}$ .

Această problemă este o variantă a unei probleme NP-complete denumită *problema partiției* ([https://en.wikipedia.org/wiki/Partition\\_problem](https://en.wikipedia.org/wiki/Partition_problem)), în care se cere să se verifice dacă un multiset poate fi partiționat în două submultiseturi având sumele elementelor egale, iar problema partiției este o variantă a altei probleme NP-complete, denumită *problema submulțimii cu sumă dată* ([https://en.wikipedia.org/wiki/Subset\\_sum\\_problem](https://en.wikipedia.org/wiki/Subset_sum_problem)), în care se cere să se verifice dacă un multiset are o submulțime având suma elementelor egală cu o valoare dată, iar aceasta este, de fapt, un caz particular al variantei discrete a problemei rucsacului!

Dacă notăm cu  $S$  suma elementelor multisetului  $A$ , cu  $S_X$  suma elementelor submultisetului  $X$  și cu  $S_Y$  suma elementelor submultisetului  $Y$ , cele două submultiseturi  $X$  și  $Y$  trebuie determinate astfel încât valoarea expresiei  $|S_X - S_Y|$  să fie minimă. Cum  $S_X + S_Y = S$ , rezultă că  $|S_X - S_Y| = |S_X - (S - S_X)| = |2S_X - S| = |S - 2S_X|$ . Deoarece valoarea minimă a unei expresii de forma  $|E(x)|$  este 0 și se obține când  $E(x) = 0$ , rezultă că suma  $S_X$  trebuie să fie cât mai apropiată de valoarea  $\left\lfloor \frac{S}{2} \right\rfloor$ , respectiv suma  $S_X$  trebuie să fie cea mai mare sumă mai mică sau egală decât  $\left\lfloor \frac{S}{2} \right\rfloor$  care se poate obține folosind elementele multisetului  $A$ . În acest caz, valoarea absolută a diferenței dintre suma elementelor submultisetului  $X$  și suma elementelor submultisetului  $Y$  va fi minimă și egală cu  $S - 2S_X$ . Evident, submultisetul  $Y$  va fi egal cu  $A \setminus X$ , unde  $X$  este submultisetul determinat astfel încât  $S_X$  este cea mai mare sumă mai mică sau egală decât  $\left\lfloor \frac{S}{2} \right\rfloor$  care se poate obține folosind elementele multisetului  $A$ .

Un algoritm pentru rezolvarea acestei probleme, bazat pe metoda programării dinamice, este asemănător cu cel utilizat pentru rezolvarea variantei discrete a problemei rucsacului, respectiv vom utiliza o matrice *sume* cu elemente de tip boolean (0/False sau 1/True), având  $n + 1$  linii și  $\left\lfloor \frac{S}{2} \right\rfloor + 1$  coloane, în care un element  $sume[i][j]$  va avea valoarea 1 dacă suma  $j$  se poate obține utilizând primele  $i$  elemente  $a_1, a_2, \dots, a_i$  ale multisetului  $A$  sau 0 în caz contrar. Valorile elementelor matricei *sume* se calculează plecând de la următoarele observații:

- $sume[i][0] = 1$  pentru orice  $i \in \{0, 1, \dots, n\}$  deoarece suma 0 se poate obține întotdeauna din primele  $i$  elemente ale multisetului  $A$ , respectiv neselectând niciunul dintre ele (se consideră faptul că suma elementelor multisetului vid este 0);
- $sume[0][j] = 0$  pentru orice  $j \in \{1, \dots, \left\lfloor \frac{S}{2} \right\rfloor\}$  deoarece folosind 0 elemente din multisetul  $A$  nu se poate obține nicio sumă  $j$  nenulă (atenție, elementul  $sume[0][0]$  va rămâne egal cu 1!);
- folosind primele  $i$  elemente  $a[1], a[2], \dots, a[i]$  ale multisetului  $A$  se poate obține suma  $j$  în următoarele două cazuri:
  - nu se utilizează elementul  $a[i]$ , ceea ce presupune ca suma  $j$  să poată fi obținută folosind doar primele  $i - 1$  elemente  $a[1], a[2], \dots, a[i - 1]$  ale multisetului  $A$ , deci elementul  $sume[i - 1][j]$  trebuie să fie egal cu 1;
  - se utilizează elementul  $a[i]$ , ceea ce presupune ca  $a[i] \leq j$  și suma  $j - a[i]$  să poată fi obținută folosind doar primele  $i - 1$  elemente  $a[1], a[2], \dots, a[i - 1]$  ale multisetului  $A$ , deci elementul  $sume[i - 1][j - a[i]]$  trebuie să fie egal cu 1.

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

$$sume[i][j] = \begin{cases} 1, & \text{dacă } 0 \leq i \leq n \text{ și } j = 0 \\ 0, & \text{dacă } i = 0 \text{ și } 1 \leq j \leq \left\lfloor \frac{S}{2} \right\rfloor \\ (sume[i - 1][j] == 1) \text{ OR } \\ ((a[i] \leq j) \text{ AND } (sume[i - 1][j - a[i]] == 1)), & \text{în orice alt caz} \end{cases}$$

Considerând exemplul dat, respectiv multisetul  $A = \{4, 6, 4, 6, 15\}$  având  $n = 5$  elemente cu suma  $S = 35$ , vom obține următoarele valori pentru elementele matricei *sume* (atenție,  $i$  nu reprezintă indicele unui element din multisetul  $A$ , ci are semnificația descrisă mai sus – *primele  $i$  elemente din multisetul  $A$* ):

$a_{i-1}$	$i/j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	2	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0
4	3	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0
6	4	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
15	5	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	1	1	0

Elementele matricei *sume* au fost calculate astfel:

- pe linia 1, doar  $sume[1][0] = 1$  și  $sume[1][4] = 1$ , iar restul elementelor sunt egale cu 0, deoarece folosind doar primul element  $a[1] = 4$  din multisetul  $A$  se pot obține sumele 0 (dacă nu îl selectăm pe  $a[1]$ ) și 4 (putem să-l selectăm pentru că  $a[1] = 4 \leq j = 4$  și  $sume[1-1][4-4] = sume[0][0] = 1 \Rightarrow sume[1][4] = 1$ );
- $sume[3][8] = 1$ , pentru că  $a[3] = 4 \leq j = 8$  și  $sume[3-1][8-4] = 1$ ;
- deși elementul  $a[4] = 6$  nu poate fi utilizat pentru a obține suma  $j = 8$  (deoarece  $sume[4-1][8-6] = sume[3][2] = 0$ , deci folosind primele 3 elemente ale multisetului  $A$  nu putem obține suma  $8 - 6 = 2$ ), totuși,  $sume[4][8] = 1$  deoarece suma  $j = 8$  se poate obține din primele  $i = 3$  elemente ale multisetului  $A$  ( $sume[3][8] = 1$ ).

Cea mai mare sumă mai mică sau egală decât  $\left\lfloor \frac{S}{2} \right\rfloor$  care se poate obține folosind elementele multisetului  $A$  este dată de valoarea celui mai mare indice  $j$  pentru care  $sume[n][j] = 1$ , în cazul exemplului nostru acesta fiind  $j = 16$ , deci diferența minimă cerută este egală cu  $S - 2 \cdot j = 35 - 32 = 3$ .

Pentru a reconstitui cele două submultiseturi de partiție  $X$  și  $Y$  vom utiliza informațiile din matricea *sume*, astfel:

- considerăm indicele  $i = n$  și indicele  $j$  determinat anterior;
- dacă  $sume[i][j] \neq sume[i-1][j]$ , înseamnă că elementul  $a_i$  a fost utilizat pentru a obține suma  $j$  din primele  $i$  elemente ale multisetului  $A$ , deci îl vom adăuga în submultisetul  $X$ , vom decrementa indicele  $i$ , iar indicele  $j$  va deveni  $j - a[i]$ ;
- dacă  $sume[i][j] = sume[i-1][j]$ , înseamnă că elementul  $a_i$  nu a fost utilizat pentru a obține suma  $j$  din primele  $i$  elemente ale multisetului  $A$ , deci îl vom adăuga în submultisetul  $Y$  și vom decrementa indicele  $i$ .

În cazul exemplului de mai sus, avem  $j = 16$ , iar pentru reconstituirea celor două submultiseturi de partiție  $X$  și  $Y$  vom urma traseul marcat cu roșu și albastru în matricea

*sume*, elementele care sunt încadrate cu un pătrat albastru corespunzând elementelor mulțimii  $X = \{6, 6, 4\}$ , iar cele cu roșu elementelor mulțimii  $Y = \{15, 4\}$ .

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python:

```
# lista A, indexată de la 1, va conține elementele multisetului A
# (am adăugat în lista A un prim element "inexistent" egal cu 0)
A = [0]
A.extend([int(x) for x in input("Multisetul A: ").split()])

# n = numărul de elemente din multisetul A
n = len(A) - 1
# S = suma elementelor multisetului A
S = sum(A)

# inițializăm elementele matricei sume
# aflate pe prima linie și prima coloană
sume = [[False for j in range(S//2 + 1)] for i in range(n + 1)]
for i in range(n+1):
    sume[i][0] = True

# calculăm restul elementelor matricei sume
# folosind relația de recurență prezentată
for i in range(1, n+1):
    for j in range(1, S//2+1):
        sume[i][j] = sume[i-1][j]
        if A[i] <= j:
            sume[i][j] = sume[i][j] or sume[i-1][j - A[i]]

# reconstituim o soluție a problemei
for j in range(S//2, -1, -1):
    if sume[n][j] == True:
        i = n
        A1 = []
        A2 = []
        while i > 0:
            if sume[i][j] != sume[i-1][j]:
                A1.append(A[i])
                j = j - A[i]
            else:
                A2.append(A[i])
            i = i-1
        print("A1 = {" + ", ".join([str(x) for x in A1]) + "}")
        print("A2 = {" + ", ".join([str(x) for x in A2]) + "}")
        print("Diferența minimă:", abs(sum(A1) - sum(A2)))
        break
```

Complexitatea algoritmului prezentat este una de tip pseudo-polinomial, fiind egală cu  $O(n \binom{S}{2}) \approx O(nS) \approx O(n2^{\lceil \log_2 S \rceil})$ .

### 3. Planificarea proiectelor cu bonus maxim

Considerăm  $n$  proiecte  $P_1, P_2, \dots, P_n$  pe care poate să le execute o echipă de programatori într-o anumită perioadă de timp (de exemplu, o lună), iar pentru fiecare proiect se cunoaște un interval de timp în care acesta trebuie executat (exprimat prin numerele de ordine a două zile din perioada respectivă), precum și 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 aceeași zi!

#### Exemplu:

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	

Deși 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 soluție corectă. De exemplu, dacă am planifica proiectele în ordinea descrescătoare a bonusurilor, atunci un proiect  $P_1$  ([1,10], 1000 RON) cu bonus mare și durată mare ar fi programat înaintea a două proiecte  $P_2$  ([1,5], 900 RON) și  $P_3$  ([6,9], 800 RON) cu bonusuri și 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 și pentru alte criterii de selecție bazate pe ziua de început, pe ziua de sfârșit, pe durată sau pe raportul dintre bonusul și 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 sfârșit (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 obține 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 obține luând în considerare și 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  $P_j$ ) și 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 obține dacă nu planifică proiectul  $P_i$ , adică  $bmax_{i-1}$ , și bonusul pe care îl echipa poate obține 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-un tablou, 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 <sub>3</sub>		P <sub>4</sub>		P <sub>2</sub>		P <sub>1</sub>	
		1	3	2	6	4	12	7	13
		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	650	1050	1500	1500	2500	2500
		250	650	800+250	850+650	900+250	1000+1500	900+1500	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.in`, care conține pe fiecare linie informațiile despre un proiect, în ordinea denumire, ziua inițială, ziua finală și bonusul, iar datele de ieșire se vor scrie în fișierul text `proiecte.out`, în forma din exemplul dat:

```
# 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:
    # 1 proiect = 1 tuplu (denumire, data început, data sfârșit, bonus)
    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 după data de sfârșit
lp.sort(key=cheieDataSfarsitProiect)
```



```

# calculăm elementele listelor pmax și ult
bmax = [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] + bmax[ult[i]] > bmax[i-1]:
        bmax[i] = lp[i][3] + bmax[ult[i]]
    else:
        bmax[i] = bmax[i-1]

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

sol.reverse()

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(bmax[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/>.

4. Modificați algoritmul de tip Backtracking pentru descompunerea unui număr natural ca sumă de numere naturale nenule astfel încât să afișeze doar:

- a) *descompunerile distincte*, respectiv descompunerile care nu au aceiași termeni în altă ordine (de exemplu, pentru  $n = 4$  aceste descompuneri sunt  $1+1+1+1$ ,  $1+1+2$ ,  $1+3$ ,  $2+2$  și  $4$ );

În acest caz, vom păstra elementele soluției în ordine crescătoare, inițializând elementul curent  $s[k]$  cu valoarea elementului anterior  $s[k-1]$  pentru  $k \geq 2$ , respectiv cu 1 pentru  $k = 1$ :

```
for v in range(1 if k==1 else sol[k-1], n-k+2):
    .....
```

- b) *descompunerile cu termeni distincți*, respectiv descompunerile care nu au termeni egali (de exemplu, pentru  $n = 4$  aceste descompuneri sunt 1+3, 3+1 și 4);

În acest caz, vom verifica în condițiile de continuare, în plus, faptul că elementul curent  $s[k]$  nu a mai fost folosit deja, respectiv  $s[k]$  nu este egal cu niciuna dintre valorile  $s[1], s[2], \dots, s[k-1]$ :

```
if sqrt <= n and sol[k] not in sol[1: k]:
    .....
```

- c) *descompuneri distincte cu termeni distincți*, respectiv descompunerile care nu au aceiași termeni în altă ordine și nici nu conțin termeni egali (de exemplu, pentru  $n = 4$ , aceste descompuneri sunt 1+3 și 4);

În acest caz, vom păstra elementele soluției în ordine strict crescătoare, inițializând elementul curent  $s[k]$  cu valoarea  $s[k-1] + 1$  pentru  $k \geq 2$ , respectiv cu 1 pentru  $k = 1$ :

```
for v in range(1 if k == 1 else sol[k-1]+1, n-k+2):
    .....
```

- d) *soluțiile ale căror lungimi au o anumită proprietate*, respectiv lungimile lor sunt mai mici, egale sau mai mari decât un număr natural  $p$  (de exemplu, pentru  $n = 4$ , soluțiile având lungimea  $p = 3$  sunt 1+1+2, 1+2+1 și 2+1+1).

În acest caz, vom verifica în condițiile de soluție, în plus, faptul că lungimea  $k$  a soluției are proprietatea cerută:

```
if sqrt == n and k == p:
    .....
```

5. Să se afișeze toate numerele naturale formate din cifre distincte și având suma cifrelor egală cu o valoare  $c$  dată. De exemplu, pentru  $c = 3$ , trebuie să fie afișate numerele: 102, 12, 120, 201, 21, 210, 3 și 30 (nu neapărat în această ordine).

### Rezolvare:

Analizând enunțul problemei, observăm faptul că orice număr care este soluție a problemei are cel mult 10 cifre, deoarece acestea trebuie să fie distincte, și problema are soluție doar în cazul în care  $c \in \{0, 1, \dots, 45\}$ , deoarece cea mai mare sumă care se poate obține din cifre distincte este egală cu  $1 + 2 + \dots + 9 = 45$ .

Pentru a rezolva problema vom utiliza metoda Backtracking, astfel:

- $s[k]$  reprezintă cifra aflată pe poziția  $k$  într-un număr (considerăm cifrele unui număr ca fiind numerotate de la stânga spre dreapta), deci obținem  $\min_k=1$  pentru  $k=1$  (prima cifră a unui număr nu poate fi 0) sau  $\min_k=0$  pentru  $k \geq 2$ , respectiv  $\max_k=9$ ;
- $s[1], \dots, s[k-1], s[k]$  este soluție parțială dacă cifra curentă  $s[k]$  nu a mai fost utilizată anterior, adică  $s[k] \neq s[i]$  pentru orice  $i \in \{1, \dots, k-1\}$ , și  $s[1] + \dots + s[k] \leq c$ ;
- pentru a testa dacă  $s[1], \dots, s[k]$  este soluție vom ține cont de faptul că cifrele  $s[1], \dots, s[k]$  sunt distincte (din condițiile de continuare), deci vom verifica doar condiția suplimentară  $s[1] + \dots + s[k] == c$ .

Implementând algoritmul Backtracking corespunzător observațiilor de mai sus, nu vom obține toate soluțiile, ci doar o parte dintre ele! De exemplu, pentru  $c = 3$ , **nu** vom obține numerele scrise îngroșat: 102, 12, **120**, 201, 21, **210**, 3 și **30**. Explicația acestui fapt necesită o înțelegere aprofundată a metodei Backtracking: numerele scrise îngroșat sunt soluții care se obțin din soluțiile care nu conțin cifra 0 (de exemplu, numărul **120** se obține din numărul 12, care nu conține cifra 0, prin adăugarea unui 0 la sfârșitul său)! În forma sa generală, algoritmul Backtracking **nu** va furniza niciodată numerele scrise îngroșat, deoarece după găsirea unei soluții a problemei, algoritmul **nu** va încerca niciodată să adauge încă un element (o cifră, în acest caz) la ea! Din acest motiv, o soluție completă care nu modifică foarte mult algoritmul general Backtracking se poate obține astfel: în momentul afișării unei soluții, verificăm dacă ea conține deja o cifră egală cu 0, iar în caz negativ o afișăm încă o dată și-i adăugăm un 0 la sfârșit.

Programul scris în limbajul Python care implementează rezolvarea completă a acestei probleme este următorul:

```
import sys

def bkt(k):
    global c
    for v in range(1 if k == 1 else 0, 10):
        st[k] = v
        scrt = sum(st[1:k+1])
        if st[k] not in st[1:k] and scrt <= c:
            if scrt == c:
                print(*st[1:k+1], sep="")
                if 0 not in st[1:k+1]:
                    print(*st[1:k + 1], 0, sep="")
            else:
                bkt(k+1)

c = int(input("c = "))
if c < 0 or c > 45:
    print("Problema nu are soluție!")
    sys.exit()
```

```
st = [0 for i in range(11)]
print("Toate numerele cerute:")
bkt(1)
```

Complexitatea acestui algoritm poate dedusă foarte greu în raport de valoarea  $c$ , dar se poate observa ușor faptul că numărul maxim de soluții se obține pentru  $c = 45$  și este egal cu  $9! + 9 \cdot 9! = 10! = 3628800$ .

6. Se consideră  $n$  spectacole pentru care se cunosc intervalele de desfășurare. Să se găsească toate planificările cu număr maxim de spectacole care se pot efectua într-o singură sală astfel încât, în cadrul fiecărei planificări, spectacolele să nu se suprapună.

### Exemplu:

spectacole.in	spectacole.out
10:00-11:20 Scufita Rosie	08:20-09:50 Vrajitorul din Oz
09:30-12:10 Punguta cu doi bani	10:00-11:20 Scufita Rosie
08:20-09:50 Vrajitorul din Oz	12:10-13:10 Micul Print
11:30-14:00 Capra cu trei iezi	15:00-15:30 Frumoasa si Bestia
12:10-13:10 Micul Print	
14:00-16:00 Povestea porcului	08:20-09:50 Vrajitorul din Oz
15:00-15:30 Frumoasa si Bestia	10:00-11:20 Scufita Rosie
	12:10-13:10 Micul Print
	14:00-16:00 Povestea porcului
	08:20-09:50 Vrajitorul din Oz
	10:00-11:20 Scufita Rosie
	11:30-14:00 Capra cu trei iezi
	15:00-15:30 Frumoasa si Bestia
	08:20-09:50 Vrajitorul din Oz
	10:00-11:20 Scufita Rosie
	11:30-14:00 Capra cu trei iezi
	14:00-16:00 Povestea porcului

### Rezolvare:

Problema poate fi rezolvată folosind doar metoda Backtracking, respectiv generând toate planificările posibile ale celor  $n$  spectacole date și păstrându-le pe cele care sunt corecte și au lungimea maximă. Deoarece ordinea în care sunt planificate spectacolele contează, trebuie utilizată o variantă modificată a generării tuturor aranjamentelor de lungime  $m$  ale unei mulțimi cu  $n$  elemente. Din cauza faptului că numărul maxim de spectacole care se pot planifica fără suprapuneri poate varia între 1 (dacă toate cele  $n$  spectacole se suprapun) și  $n$  (dacă niciun spectacol nu se suprapune cu toate celelalte  $n - 1$ ), va trebui să considerăm și valoarea lui  $m$  ca fiind cuprinsă între 1 și  $n$ , deci numărul total de planificări generate va fi aproximativ  $A_n^1 + A_n^2 + \dots + A_n^n \gg A_n^n = n!$ , deci complexitatea acestui algoritm va fi mult mai mare decât  $\mathcal{O}(n!)$ .

O variantă mai eficientă de rezolvare a acestei probleme o constituie utilizarea metodei Greedy pentru a afla numărul maxim de spectacole  $nms$  care se pot programa fără

suprapuneri, cu o complexitate  $\mathcal{O}(n \log_2 n)$  și generarea doar a aranjamentelor cu  $nms$  elemente ale unei mulțimi cu  $n$  elemente:

```
# funcție care determina numărul maxim de spectacole care pot fi
# programate fără suprapuneri folosind metoda Greedy
def numarMaximSpectacole(lsp):
    # sortăm spectacolele în ordinea crescătoare a orelor de sfârșit
    lsp.sort(key=lambda s: s[2])

    # ora de sfârșit a ultimului spectacol programat
    ult = "00:00"
    # cnt = numărul maxim de spectacole care se pot programa corect
    cnt = 0
    for sp in lsp:
        if sp[1] >= ult:
            cnt += 1
            ult = sp[2]

    return cnt

# generarea tuturor programărilor cu număr maxim de spectacole
# folosind metoda backtracking, respectiv generarea aranjamentelor
# cu nms elemente ale celor n spectacole
def bkt(k):
    # fout = fișierul text în care vom scrie soluțiile
    # lsp = lista cu spectacolele
    # n = numărul de spectacole din lista lsp
    global s, nms, fout, lsp, n

    # s[k] = spectacolul aflat pe poziția k în planificarea curentă
    for v in range(n):
        s[k] = v
        if v not in s[:k] \
            and (k == 0 or lsp[s[k]][1] >= lsp[s[k-1]][2]):
            if k == nms-1:
                for p in s:
                    fout.write(lsp[p][1] + "-" + lsp[p][2] + " " +
                               lsp[p][0] + "\n")
                fout.write("\n")
            else:
                bkt(k+1)

fin = open("spectacole.in")

# lsp = lista spectacolelor
lsp = []
```

```

for linie in fin:
    aux = linie.split()
    # ora de inceput si ora de sfarsit pentru spectacolul curent
    tsp = aux[0].split("-")
    lsp.append(" ".join(aux[1:]), tsp[0], tsp[1])

fin.close()

# n = numărul de spectacole
n = len(lsp)

fout = open("spectacole.out", "w")

# nms = numărul maxim de spectacole care se pot programa corect
# = lungimea soluțiilor care vor fi generate cu backtracking
nms = numarMaximSpectacole(lsp)

s = [0] * nms
bkt(0)

fout.close()

```

### Probleme propuse

1. Fie  $A$  un multiset format din  $n$  numere naturale nenule și  $S$  un număr natural nenul. Folosind metoda Backtracking, să se afișeze toate submultiseturile lui  $A$  care au suma elementelor egală cu  $S$ .
2. Fie  $A$  un multiset format din  $n$  numere naturale nenule și  $S$  un număr natural nenul. Folosind metoda programării dinamice, să se afișeze un submultiset a lui  $A$  care are suma elementelor egală cu  $S$ .
3. Folosind doar metoda backtracking, generați toate subșirurile crescătoare maximale ale un șir format din  $n$  numere întregi.
4. Optimizați algoritmul de la problema anterioară, utilizând metoda programării dinamice pentru a determina lungimea maximă a unui subșir crescător al șirului dat.