# TEHNICA PROGRAMĂRII DINAMICE

## 1. Subșir crescător maximal

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

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

De exemplu, în șirul $t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$ un subșir crescător maximal este $(3, 6, 8, 8, 10)$. Soluția nu este unică, un alt subșir crescător maximal fiind $(3, 4, 5, 8, 10)$. Se observă faptul că problema are întotdeauna soluție, chiar și în cazul în care șirul dat este strict descrescător (orice element al șirului este un subșir crescător maximal de lungime 1)!

Algoritmii de tip Greedy nu rezolvă corect această problemă în orice caz. De exemplu, pentru fiecare element din șirul dat, am putea încerca să construim un subșir crescător maximal selectând, de fiecare dată, cel mai apropiat element mai mare sau egal decât ultimul element din subșirul curent și să reținem subșirul crescător de lungime maximă astfel obținut. Aplicând acest algoritm pentru șirul $t = (7, 3, 9, 4, 5)$ vom obține subșirurile $(7, 9), (3, 9), (9), (4, 5)$ și $(5)$, deci soluția furnizată de algoritmul Greedy ar fi unul dintre cele 3 subșiruri crescătoare de lungime 2. Evident, soluția ar fi incorectă, deoarece soluția optimă este subșirul (3, 4, 5), de lungime 3!

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

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 aceeași idee, însă fără a utiliza tehnica memoizării).

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

Pentru a calcula lungimea maximă a unui subșir crescător care se termină cu elementul $t[i]$, vom lua în considerare, pe rând, toate subșirurile care se termină cu elementele $t[0], t[1], \dots, t[i-1]$, deoarece cunoaștem deja lungimile maxime $lmax[0], lmax[1], \dots, lmax[i-1]$ ale subșirurilor crescătoare care se termină cu ele, și vom încerca să alipim elementul $t[i]$ la fiecare dintre ele. Dacă acest lucru este posibil, respectiv dacă $t[i] \geq t[j]$ pentru un indice $j \in \{0,1, \dots, i-1\}$, 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 $lmax[0], lmax[1], \dots, lmax[i-1]$), cât și *condiția de superpozabilitate* (valoarea $lmax[i]$ va fi utilizată în calculul valorilor $lmax[i+1], \dots, lmax[n-1]$), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

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

Pentru a reconstitui mai simplu un subșir crescător maximal al șirului inițial, vom utiliza o listă auxiliară $pred$, în care un element $pred[i]$ va conține valoarea $-1$ dacă elementul $t[i]$ nu a putut fi alipit la niciunul dintre subșirurile crescătoare maximale care se termină cu $t[0], t[1], \dots, t[i-1]$ sau va conține indicele $j \in \{0,1,\dots,i-1\}$ al subșirului crescător maximal $t[j]$ la care a fost alipit elementul $t[i]$, adică indicele $j$ pentru care s-a obținut $\max\limits_{0 \le j < i}\{lmax[j] \mid t[i] \ge t[j]\}$.

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 | 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 $lmax$ și $pred$ au fost calculate astfel:

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

Lungimea maximă a unui subșir crescător maximal al șirului inițial este dată de valoarea maximă din lista $lmax$, iar pentru a reconstitui un subșir 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 $lmax[0], lmax[1], \ldots, lmax[i-1]$ ale subproblemelor $0,1, \ldots, i-1$), 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 $lmax[i+1], lmax[i+2], \ldots, lmax[n-1]$ ale subproblemelor $i+1, i+2, \ldots, n-1$. 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 $t[i+1], t[i+2], \ldots, t[n-1]$, deoarece cunoaștem deja lungimile maxime $lmax[i+1], lmax[i+2], \ldots, lmax[n-1]$ 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 $j \in \{i+1, i+2, \ldots, n-1\}$, 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 $lmax[i+1], lmax[i+2], \ldots, lmax[n-1]$), cât și *condiția de superpozabilitate* (valoarea $lmax[i]$ va fi utilizată în calculul valorilor $lmax[0], \ldots, lmax[i-1]$), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

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

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 $t[i+1], t[i+2], \ldots, t[n-1]$ sau va conține indicele $j \in \{i+1, i+2, \ldots, n-1\}$ al subșirului crescător maximal $t[j]$ înaintea căruia a fost adăugat elementul $t[i]$, adică indicele $j$ pentru care s-a obținut $\max\limits_{i+1 \leq j < n} \{lmax[j] \mid t[i] \leq t[j]\}$.

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 $lmax$ și $succ$ au fost calculate astfel:

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

Lungimea maximă a unui subșir crescător maximal al șirului inițial este dată de valoarea maximă din lista $lmax$, iar pentru a reconstitui un subșir crescător maximal vom utiliza informațiile din lista $succ$ și faptul că un subșir crescător maximal începe cu elementul $t[pmax]$, unde $pmax$ este poziția 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 subșir crescător maximal începe cu $t[2]$. Pentru afișarea unui subșir crescător maximal vom proceda astfel:

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

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
```

```
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 numește *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 $1 + [\log_2 x]$ biți, 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* și *o complexitate exponențială în raport cu lungimea reprezentării binare a sumei P de plată*!

## 3. Problema rucsacului (varianta discretă)

Considerăm un rucsac având capacitatea maximă $G$ și $n$ obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaștem greutățile lor $g_1, g_2, \dots, g_n$ și câștigurile $c_1, c_2, \dots, c_n$ obținute prin încărcarea lor în rucsac. Știind faptul că toate greutățile și toate câștigurile 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 câștigul total obținut să fie maxim.

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

În capitolul dedicat tehnicii de programare Greedy am văzut faptul că varianta fracționară a acestei probleme poate fi rezolvată corect utilizând tehnica respectivă. În cazul variantei discrete, tehnica Greedy nu va mai furniza o soluție corectă întotdeauna. Astfel, câștigurile 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 $O_2, O_4$ și $O_5$, deoarece obiectele nu pot fi "tăiate" în varianta discretă a problemei rucsacului, și ar obține un câștig total egal cu 180 RON, evident mai mic decât cel maxim de 190 RON!

Se observă foarte ușor faptul că varianta discretă a problemei rucsacului nu are întotdeauna soluție, respectiv în cazul în care greutatea celui mai mic obiect este strict mai mare decât capacitatea $G$ a rucsacului, în timp ce varianta fracționară ar avea soluție în acest caz (ar "tăia" din obiectul cu cel mai mare câștig 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 plății unei sume folosind un număr minim de monede, astfel:

- considerăm faptul că am analizat, pe rând, obiectele $O_1, O_2, \dots, O_{n-1}$ și am calculat câștigul maxim pe care îl putem obține folosindu-le (nu neapărat pe toate!) în limita întregii capacități $G$ a rucsacului, deci mai trebuie să calculăm doar câștigul maxim pe care îl putem obține folosind și ultimul obiect $O_n$;
- dacă obiectul $O_n$ nu încape în rucsac (deci $g_n > G$), înseamnă că nu-l putem folosi deloc, deci câștigul maxim rămâne cel pe care l-am obținut 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 câștigul maxim deja obținut

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:

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

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 $O_1, O_2, \ldots, O_{i-1}$, 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 | $\boxed{80}$ | 80 | 80 | 80 | 80 | 80 |
| $O_2$ | 50 | 2 | 2 | 0 | 0 | 50 | 50 | 50 | 80 | 80 | $\boxed{130}$ | 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 | $\boxed{190}$ |
| $O_5$ | 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 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:**
Vom considera faptul că datele de intrare se citesc din fișierul text `proiecte.in`, care conține pe prima linie numărul $n$ de proiecte, iar fiecare dintre următoarele $n$ linii conține intervalul de timp în care proiectul trebuie executat și bonusul acordat. De exemplu, a doua linie din fișierul de intrare conține informațiile despre proiectul $P_1$, respectiv intervalul [7, 13] în care acesta trebuie efectuat pentru ca echipa să obțină bonusul de 850 RON. Datele de ieșire se vor scrie în fișierul 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` | |

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 $P_1([1,10], 1000\text{ RON})$ cu bonus mare și durată mare ar fi programat înaintea a două proiecte $P_2([1,5], 900\text{ RON})$ și $P_3([6,9], 800\text{ RON})$ 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-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 | – | $P_3$ | $P_4$ | $P_2$ | $P_1$ | $P_6$ | $P_5$ | $P_8$ | $P_7$ |
| | | 1 3 | 2 6 | 4 12 | 7 13 | 4 16 | 13 18 | 15 22 | 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 | 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.txt`, care conține pe fiecare linie informațiile despre câte un proiect, în ordinea denumire, ziua inițială, ziua finală și bonusul:

```python
# 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/.