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!

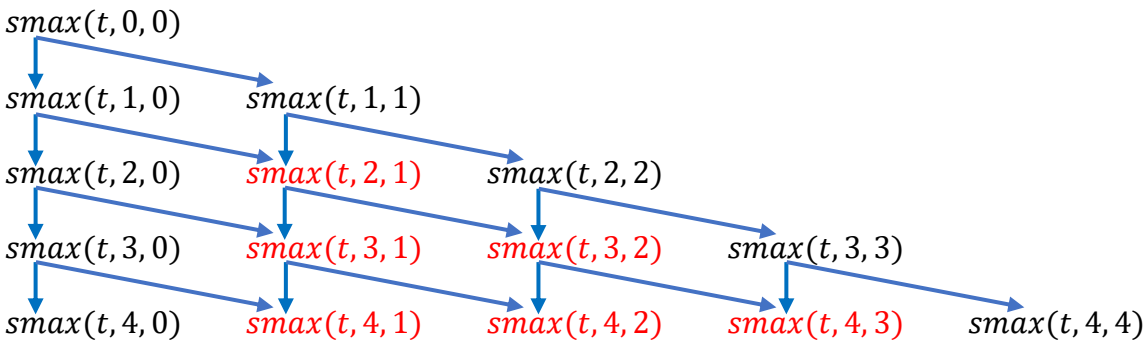
O altă variantă de rezolvare ar putea utiliza tehnica Backtracking pentru generarea tuturor traseelor care respectă cerinţele problemei şi selectarea celui care are suma elementelor maximă. Această variantă de rezolvare este corectă, dar ineficientă, deoarece are complexitatea exponenţială $\mathcal{O}(2^{n-1})$. 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:

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

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 soluţiile 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.

3. Subşir crescător maximal

Considerăm un şir t format din n numere întregi $t = (t_0, t_1, \ldots, t_{n-1})$. 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 $s = (t_{i_0}, t_{i_1}, \ldots, t_{i_m})$ cu $0 \leq i_0 < i_1 < \cdots < i_m \leq n-1$) 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 $1, 2, \ldots, n$ 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 $C_n^1 + C_n^2 + \cdots + C_n^n = 2^n - 1$, 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 $t[i] \geq t[j]$ 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:

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

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 $\max\limits_{0 \leq j < i} \{lmax[j] | t[i] \geq t[j]\}$.

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 $t[1] = 7 > 3$;

- $lmax[3] = 2$ și $pred[3] = 2$, deoarece elementul $t[3] = 6$ poate fi alipit la un subșir crescător maximal care se termină cu $t[2] = 3 \leq 6$, 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 $lmax[0], lmax[1], \dots, lmax[i-1]$ 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 subşirurile crescătoare maximale care încep cu $t[i+1], t[i+2], \dots, t[n-1]$ sau va conţine indicele $j \in \{i+1, i+2, \dots, n-1\}$ al subşirului crescător maximal $t[j]$ înaintea căruia a fost adăugat elementul $t[i]$, adică indicele j pentru care s-a obţinut $\max_{i+1 \le j < n} \{lmax[j] | t[i] \le t[j]\}$.

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	**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 $t[11] = 10 \ge 7$, 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 $t[11] = 10 \ge 8$, 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 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/.

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.

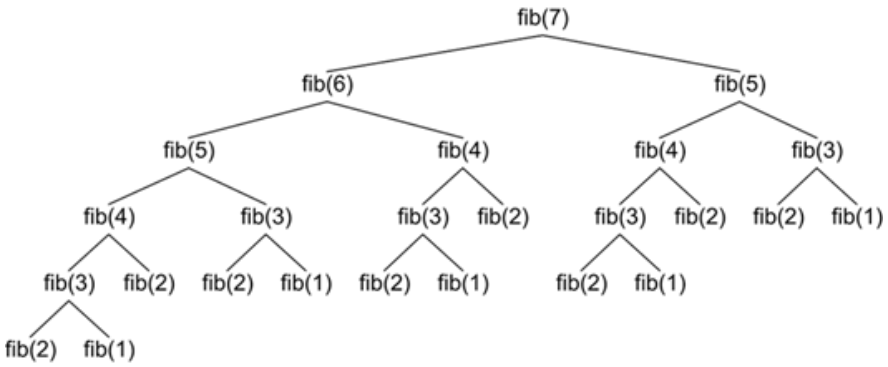
De exemplu, să considerăm şirul lui Fibonacci, definit recurent astfel:

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

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!

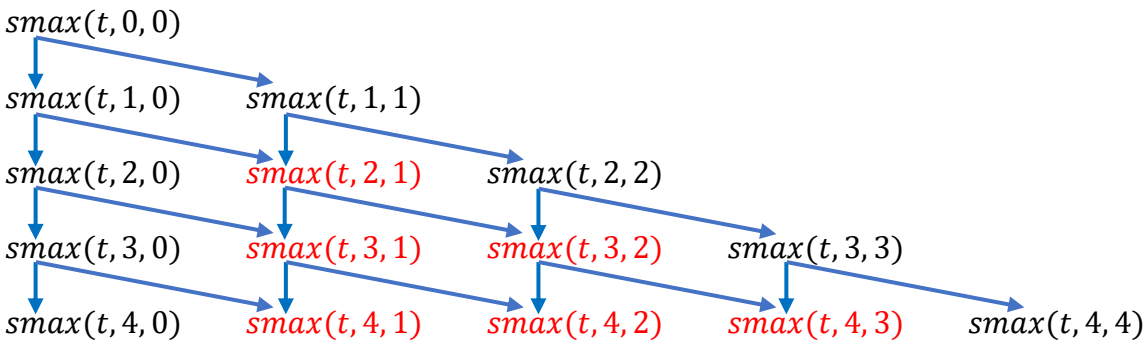
O altă variantă de rezolvare ar putea utiliza tehnica Backtracking pentru generarea tuturor traseelor care respectă cerinţele problemei şi selectarea celui care are suma elementelor maximă. Această variantă de rezolvare este corectă, dar ineficientă, deoarece are complexitatea exponenţială $\mathcal{O}(2^{n-1})$. 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:

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

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 soluţiile 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.

3. Subşir crescător maximal

Considerăm un şir t format din n numere întregi $t = (t_0, t_1, \ldots, t_{n-1})$. 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 $s = (t_{i_0}, t_{i_1}, \ldots, t_{i_m})$ cu $0 \leq i_0 < i_1 < \cdots < i_m \leq n-1$) 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 $1, 2, \ldots, n$ 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 $C_n^1 + C_n^2 + \cdots + C_n^n = 2^n - 1$, 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 $t[i] \geq t[j]$ 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:

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

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 $\max\limits_{0 \leq j < i} \{lmax[j] | t[i] \geq t[j]\}$.

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 $t[2] = 3 \leq 6$, 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 informaţiile din lista $pred$ şi faptul că un subşir crescător maximal se termină cu elementul $t[pmax]$, unde $pmax$ este poziţia 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 subşir crescător maximal se termină cu $t[11]$. Pentru afişarea unui subşir crescător maximal vom proceda astfel:

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

Pentru a afişa subşirul crescător maximal reconstituit neinversat, îl vom stoca într-o listă şi apoi o vom afişa invers.

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
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 $\max_{i+1 \le j < n} \{lmax[j] | t[i] \le t[j]\}$.

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 $t[11] = 10 \ge 7$, 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 $t[11] = 10 \ge 8$, 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/.

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

Considerând faptul că avem la dispoziție n monede cu valorile $v_1, v_2, \ldots, v_n$ 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: $4 \times 3\$$, $1 \times 2\$ + 2 \times 5\$$,

$2 \times 2\$ + 1 \times 3\$ + 1 \times 5\$, 3 \times 2\$ + 2 \times 3\$$ şi $6 \times 2\$$. 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 modalităţile 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 şi o modalitate de plată a unei sume folosind un număr minim de monede (pentru fiecare modalitate de plată vom calcula numărul de monede utilizate şi vom reţine modalitatea cu număr minim de monede), dar algoritmul va avea o complexitate exponenţială, deci va fi ineficient!

Fiind o problemă de optim, putem încerca şi 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 $v = (5\$, 3\$, 2\$)$, şi 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 $P = 0\$$ şi algoritmul se termină cu succes;
- numărul de monede utilizate, respectiv 3 monede, este minim.

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

- 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 $P = 0\$$ şi algoritmul se termină cu succes;
- numărul de monede utilizate, respectiv 4 monede, nu este minim (numărul minim de monede se obţine când se utilizează două monede cu valoarea de 4$).

Se observă faptul că existenţa monedei cu valoarea de 1$ permite algoritmului Greedy să găsească întotdeauna o soluţie, chiar dacă aceasta nu este optimă. Totuşi, sunt cazuri în care algoritmul Greedy nu va găsi nicio soluţie, deşi problema are cel puţin una. De exemplu, dacă monedele au valorile $v = (5\$, 4\$, 2\$)$ şi 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 soluţie, optimă sau nu!

Deoarece această problema are o importanţă practică deosebită, în anumite ţări sunt utilizate aşa-numitele *sisteme canonice de valori pentru monede*, care permit algoritmului Greedy prezentat mai sus (numit şi *algoritmul casierului*) să furnizeze o soluţie 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 obţine 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 obţine 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 şi $P - 1$, rezultă că pentru a putea calcula numărul minim de monede necesare pentru a plăti suma P folosind o monedă cu valoarea x trebuie să cunoaştem numărul minim de monede necesare pentru a plăti orice sumă cuprinsă între 0 şi $P - 1$ folosind toate tipurile de monede disponibile cu valorile $v_1, v_2, \dots, v_n$. Generalizând această observaţie 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 obţine 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 reţine numărul minim de monede necesare pentru a plăti suma i, cuprinsă între 0 şi P, folosind toate tipurile de monede disponibile, relaţia de recurenţă care caracterizează substructura optimală a problemei este următoarea:

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

Iniţial, 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 iniţializa toate elementele listei $nrmin$ cu $P + 1$.

Soluţia 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 iniţializare $P + 1$, altfel, dacă rămâne egală cu P+1, înseamnă că suma P nu poate fi plătită folosind monede cu valorile date. Pentru a reconstitui mai uşor 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 conţine valoarea -1 dacă nu există nicio modalitate de plată a sumei i folosind monedele date sau va conţine valoarea monedei $v[j]$ utilizată pentru a plăti suma i cu un număr minim de monede, adică valoarea $v[j]$ pentru care s-a obţinut $\min\limits_{0 \leq j < n} \{nrmin[i - v[j]] \,|\, v[j] \leq i\}$.

Considerând exemplul dat, cu $P = 12\$$ şi $v = (2\$, 5\$, 3\$)$ (valorile monedelor nu trebuie să fie sortate!), vom obţine următoarele valori pentru elementele listelor $nrmin$ şi $pred$ (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*!

5. 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 capacităţi G a rucsacului cu câştigul care s-ar obţine prin încărcarea obiectului O_n, respectiv cu suma dintre c_n şi câştigul maxim care se poate obţine folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita capacităţii $G - g_n$ rămase în rucsac. Deoarece $1 \leq g_n \leq G$, rezultă că trebuie să cunoaştem câştigurile maxime care se pot obţine folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacităţi cuprinse între 0 şi G-1, la care se adaugă câştigul maxim care se poate obţine folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita întregii capacităţi G a rucsacului (pentru cazul anterior), deci, de fapt, trebuie să cunoaştem câştigurile maxime care se pot obţine folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacităţi cuprinse între 0 şi G!
- pentru a calcula câştigurile maxime care se pot obţine folosind primele $n-1$ obiecte $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacităţi cuprinse între 0 şi G vom repeta raţionamentul anterior pentru obiectul O_{n-1} şi obiectele $O_1, O_2, \dots, O_{n-2}$, apoi pentru obiectul O_{n-2} şi obiectele $O_1, O_2, \dots, O_{n-3}$ şi aşa mai departe, până când vom calcula câştigurile maxime care se pot obţine folosind doar primul obiect O_1 în limita oricărei capacităţi cuprinse între 0 şi G.

În concluzie, pentru a rezolva problema utilizând tehnica programării dinamice, trebuie să cunoaştem toate câştigurile maxime care se pot obţine folosind primele i obiecte ($i \in \{0,1,\dots,n\}$), în limita oricărei capacităţi j cuprinse între 0 şi G, deci, aplicând tehnica memoizării, vom considera un tablou bidimensional $cmax$ cu $n+1$ linii şi $G+1$ coloane în care un element $cmax[i][j]$ va memora câştigul maxim care se poate obţine folosind primele i obiecte în limita a j kilograme. Astfel, relaţia de recurenţă 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 $i \in \{0,1,\dots,n\}$ şi fiecare $j \in \{0,1,\dots,G\}$. În plus faţă de modalitatea de calcul a elementului $cmax[i][j]$ descrisă mai sus, am adăugat cazurile particulare $cmax[0][j] =$

$cmax[i][0] = 0$ (evident, câştigul maxim $cmax[0][j]$ care se poate obţine folosind 0 obiecte în limita oricărei capacităţi j este 0 şi câştigul maxim $cmax[i][0]$ care se poate obţine folosind primele i obiecte în limita unei capacităţi nule este tot 0). De asemenea, am considerat tablourile c şi g ca fiind indexate de la 1, pentru a păs

Considerând exemplul dat, vom obţine 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 evidenţiate 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 puţin egală cu 5 (de exemplu, folosind relaţia de recurenţă, obţinem $cmax[1][2] = cmax[0][2] = 0$), caz în care am obţinut $cmax[1][5] = \cdots = cmax[1][10] = 80$ (de exemplu, folosind relaţia de recurenţă, obţinem $cmax[1][9] = \max\{cmax[0][9], c[1] + cmax[0][9-5]\} = \max\{0, 80 + 0\} = 80$);
- $cmax[2][7] = \max\{cmax[1][7], c[2] + cmax[1][7-2]\} = \max\{80, 50 + 80\} = 130$, deoarece în limita a $j = 7$ kg încap ambele obiecte O_1 şi 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 şi O_2 care au fost încărcate pentru a obţine câştigul 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 şi O_4 (pentru care s-a obţinut câştigul 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 câştigul maxim de 140 RON obţinut prin încărcarea obiectelor O_1 şi O_4 dintre primele $i = 4$ obiecte în limita a $j = 9$ kg.

Câştigul maxim care se poate obţine 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 informaţiile din matricea $cmax$, astfel:

- considerăm doi indici $i = n$ şi $j = G$;

- 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 valoarea "inexistentă" 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 $\mathcal{O}(nG) \approx \mathcal{O}\left(n2^{[\log_2 G]}\right)$.

6. 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 obţine echipa dacă proiectul este finalizat la timp (altfel, echipa nu va obţine niciun bonus

pentru proiectul respectiv). Să se determine o modalitate de planificare a unor proiecte care nu se suprapun astfel încât bonusul obţinut 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 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	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 soluţiilor unei probleme într-un mod progresiv, astfel încât să se evite generarea întregului spaţiu al soluţiilor problemei. Practic, soluţiile se construiesc componentă cu componentă şi se testează la fiecare pas validitatea lor (se verifică dacă sunt *soluţii parţiale*), iar în cazul în care se constată faptul că nu se poate obţine o soluţie a problemei plecând de la soluţia parţială curentă, aceasta este abandonată. Astfel, se evită o *rezolvare de tip forţă-brută* a problemei, adică generarea şi testarea tuturor soluţiilor posibile pentru a determina soluţiile problemei. Totuşi, complexitatea algoritmilor de tip Backtracking rămâne una ridicată, deoarece se va genera şi testa un procent semnificativ din întregul spaţiu al soluţiilor.

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

O rezolvare de tip forţă-brută presupune generarea tuturor posibilelor soluţii, adică a tuplurilor de forma $p = (p_1, p_2, p_3, p_4, p_5, p_6)$ cu $p_1, \dots, p_6 \in \{1,2,\dots,6\}$, şi selectarea celor care sunt permutări, adică au toate valorile diferite între ele ($p_1 \neq p_2 \neq \cdots \neq p_6$). Se observă foarte uşor faptul că se vor genera şi testa $6^6 = 46656$ tupluri, din care doar $6! = 720$ vor fi permutări, deci eficienţa acestei metode este foarte mică - în jurul unui procent de 1.5%! Eficienţa 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 soluţiilor, evitând generarea unor tupluri inutile, astfel (vom ţine cont de faptul că $p_1, \dots, p_6 \in \{1,2,\dots,6\}$):

- $p = (1)$ – este o soluţie parţială (componentele sale sunt diferite între ele, deci poate fi o permutare), dar nu este o soluţie a problemei (nu are 6 componente), astfel că trebuie să adăugăm cel puţin încă o componentă;
- $p = (1,1)$ – nu este o soluţie parţială (componentele sale sunt egale, deci nu vom obţine 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 soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p = (1,2,1) \\ p = (1,2,2) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p = (1,2,3)$ – este o soluţie parţială, dar nu este o soluţie 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 soluţii parţiale;
- $p = (1,2,3,4)$ – este o soluţie parţială, dar nu este o soluţie 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 $\mathcal{O}(n!)$, 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 observaţii:

- bucăţile de cod scrise cu roşu 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 condiţii în care se utilizează indicii tabloului;
- $\texttt{min}_k$ şi $\texttt{max}_k$ se deduc din semnificaţia componentei `s[k]` a unei soluţii;
- 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 pe cele suplimentare lor;
- dacă `s[1],…,s[k]` nu este soluţie parţială, atunci nu vom adăuga o nouă componentă `s[k+1]` prin apelul recursiv `bkt(k+1)`, deci instrucţiunea `for` va continua şi componentei curente `s[k]` i se va atribui următoarea valoare posibilă `v`, dacă aceasta există, iar în cazul în care aceasta nu există, instrucţiunea `for` corespunzătoare componentei curente `s[k]` se va termina şi, implicit, apelul funcţiei `bkt` corespunzător, deci se va reveni la componenta anterioară `s[k-1]`.

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

- `s[k]` reprezintă un element al permutării, deci $\texttt{min}_k$ `= 1` şi $\texttt{max}_k$ `= n`;
- `s[1],…,s[k-1],s[k]` este soluţie parţială 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 condiţie 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) şi 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 obţine o permutare de ordin `n`), fără a fi şi suficientă (dacă valorile `s[1],…,s[k]` sunt distincte nu înseamnă că ele formează o permutare de ordin `n`, ci trebuie impusă condiţia suplimentară `k=n`);
- 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 (`s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`), ci doar condiţia suplimentară `k==n`.

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

```
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)
```

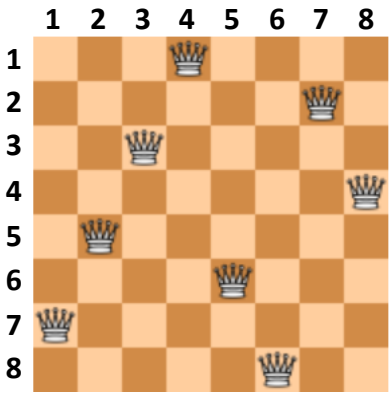
În practică, această problema apare în multe variante, câteva dintre ele fiind următoarele (în toate exemplele am considerat n=4):

- *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}`.

Condiția referitoare la coloană se deduce imediat, respectiv trebuie ca `s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`. Condiția 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 și numai dacă dreapta R_kR_i este paralelă cu una dintre cele două diagonale ale tablei de șah. Două drepte sunt paralele dacă și numai dacă au pantele egale, iar cele două diagonale au pantele egale cu $\operatorname{tg} 45° = 1$ și $\operatorname{tg} 135° = -1$, 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 funcția modul, obținem $\left|\frac{s[k]-s[i]}{k-i}\right| \neq 1$ 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 soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar condiția suplimentară `k==n`.

Practic, se observă faptul că problema celor n regine se reduce la generarea permutărilor de ordin n care verifică și condiția referitoare la diagonale, deci putem utiliza direct algoritmul de generare a permutărilor în care modificăm doar condiția 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 puțin modificată a algoritmului de generare a permutărilor de ordin n.

6. Problema plății unei sume folosind 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 toate modalitățile în care putem realiza acest lucru (vom presupune faptul că avem la dispoziție un număr suficient de monede de fiecare tip).

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

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 obținem $\texttt{min}_k$=`0` și $\texttt{max}_k$=`P/v[k]`;
- `s[1],…,s[k-1],s[k]` este soluție parțială 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 soluție dacă suma curentă este egală cu suma de plată `P`, adică `s[1]*v[1]+…+s[k]*v[k]==P` (se observă faptul că soluțiile au lungimi variabile, cuprinse între `1` și `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 $\texttt{max}_k\texttt{-min}_k$ 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.