

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, semnificația de *planificare*, ci nu semnificația din informatică! Practic, tehnica programării dinamice poate fi utilizată pentru planificarea optimă a unor activități, iar decizia de a planifica sau nu o anumită activitate se va lua *dinamic*, ținând cont de activitățile 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 ține cont de activitățile planificate anterior, ci doar verificând dacă activitatea curentă îndeplinește un anumit criteriu predefinit. Totuși, cele două tehnici de programare se aseamănă prin faptul că ambele determină o singură soluție a problemei, chiar dacă există mai multe.

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

1. *condiția de substructură optimă*: problema dată se poate descompune în subprobleme de același tip, iar soluția sa optimă (optimumul global) se obține combinând soluțiile optime ale subproblemelor în care a fost descompusă (optime locale);
2. *condiția de superpozabilitate*: subproblemele în care se descompune problema dată se suprapun.

Se poate observa faptul că prima condiție este o combinație dintre o condiție specifică tehnicii de programare *Divide et Impera* (problema dată se descompune în subprobleme de același tip) și o condiție specifică tehnicii *Greedy* (optimumul global se obține din optimele locale). Totuși, o rezolvare a acestui tip de problemă folosind tehnica *Divide et Impera* ar fi ineficientă, deoarece subproblemele se suprapun (a doua condiție), deci aceeași subproblemă ar fi rezolvată de mai multe ori, ceea ce ar conduce la un timp de executare foarte mare (de obicei, chiar exponențial)!

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

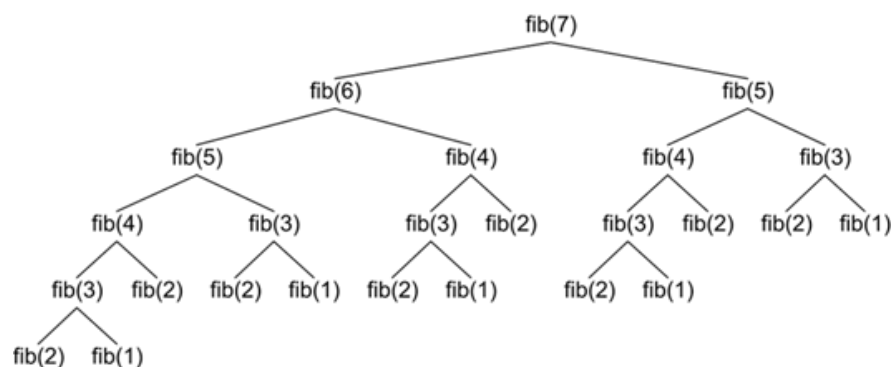
De exemplu, să considerăm șirul lui Fibonacci, definit recurent astfel:

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

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 $f_3 = \text{fib}(3)$ 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 manieră 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 relația de recurență, folosind tehnica memoizării într-o manieră ascendentă (respectiv se rezolvă subproblemele în ordinea crescătoare a dimensiunilor lor), obținându-se astfel valoarea optimă căutată;
- 4) se construiește o soluție care furnizează valoarea optimă, utilizând soluțiile subproblemelor calculate anterior (această etapă este opțională).

2. Suma maximă într-un triunghi de numere

Considerăm un triunghi format din n șiruri din numere întregi, astfel: prima linie conține un număr, a doua linie conține două numere, ..., a n -a linie (ultima) conține n numere (un astfel de triunghi este, de fapt, jumătatea inferioară a unei matrice pătratică 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 și se termină pe ultima, iar succesorul fiecărui element de pe traseu (mai puțin 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 obține este 52, pe traseul $t[0][0] \rightarrow t[1][1] \rightarrow t[2][1] \rightarrow t[3][2] \rightarrow t[4][3]$, marcat cu **roșu** 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 $t[i+1][j]$ și $t[i+1][j+1]$. Deși traseul marcat cu **roșu** în triunghiul de mai sus are această proprietate, se poate observa ușor faptul că algoritmul Greedy ar eșua dacă modificăm valoarea 7 din colțul stânga-jos în 700! În acest caz, algoritmul Greedy ar selecta tot elementele aflate pe traseul marcat cu **roșu**, dar suma maximă s-ar obține, 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 **roșu**, indiferent cum am modifica elementele marcate cu **albastru**!

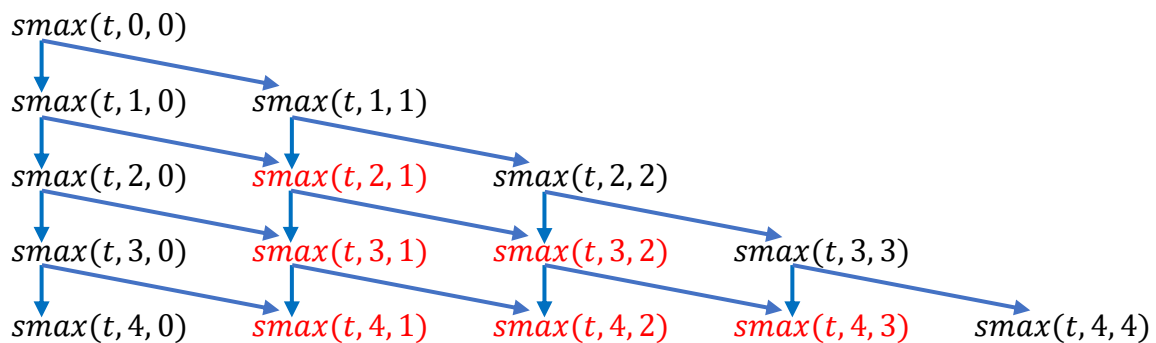
O altă variantă de rezolvare ar putea utiliza tehnica Backtracking pentru generarea tuturor traseelor care respectă cerințele problemei și selectarea celui care are suma elementelor maximă. Această variantă de rezolvare este corectă, dar ineficientă, deoarece are complexitatea exponențială $O(2^{n-1})$.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

format din elementele de pe diagonala i se asociază șirul binar 1111 (cel mai mare în sens lexicografic). Deoarece această asociere este bijectivă (unui traseu îi corespunde un singur șir binar, iar unui șir binar îi corespunde un singur traseu), rezultă că numărul traseelor corecte este egal cu numărul șirurilor 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 îndeplinește condițiile cerute pentru utilizarea acestei tehnici de programare: suma maximă pe un traseu care începe cu elementul $t[i][j]$ este egală cu el plus maximumul sumelor care se obțin pe cele două trasee care încep cu $t[i+1][j]$ și $t[i+1][j+1]$, iar problema direct rezolvabilă o constituie calcularea sumei maxime care se poate obține 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 funcție 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 funcție care calculează maximumul dintre numerele întregi x și y , iar suma maximă cerută se obține 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 și 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 **roșu** 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 și în cazul variantei Backtracking, se parcurg, până la urmă, toate traseele din triunghi!

Totuși, formula recurentă prin care este definită funcția $smax(t, i, j)$ exprimă *condiția de substructură optimă* 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 relației de recurență, vom aplica tehnica memoizării, utilizând un triunghi de numere $smax$ pentru a reține soluțiile subproblemelor, respectiv elementul $smax[i][j]$ va păstra suma maximă pe un traseu care începe cu elementul $t[i][j]$:

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

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:

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

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 $O(n^2)$.

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 optimă* 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:

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

```

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

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

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

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

3. Subșir crescător maximal

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- $lmax[0] = 1$ și $pred[0] = -1$, deoarece este evident faptul că lungimea maximă a unui subșir care se termină cu $t[0]$ este egală cu 1;
- $lmax[1] = 1$ și $pred[1] = -1$, deoarece elementul $t[1] = 7$ nu poate fi alipit la un subșir crescător maximal care se termină cu $t[0] = 9 > 7$;
- $lmax[2] = 1$ și $pred[2] = -1$, deoarece elementul $t[2] = 3$ nu poate fi alipit nici la un subșir crescător maximal care se termină cu $t[0] = 9 > 3$ și nici la un subșir crescător maximal care se termină cu $t[1] = 7 > 3$;

- $lmax[3] = 2$ și $pred[3] = 2$, deoarece elementul $t[3] = 6$ poate fi alipit la un subșir crescător maximal care se termină cu $t[2] = 3 \leq 6$, 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 maxime 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 neinvertat, 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 $O(n^2)$.

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 optimă* (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 optimă a problemei, în formă memoizată, este următoarea:

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

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

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

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

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

- $lmax[12] = 1$ și $succ[12] = -1$, deoarece este evident faptul că lungimea maximă a unui subșir care începe cu $t[12]$ este egală cu 1;
- $lmax[11] = 1$ și $succ[11] = -1$, deoarece elementul $t[11] = 10$ nu poate fi adăugat înaintea unui subșir crescător maximal care începe cu $t[12] = 4 < 10$;
- $lmax[10] = 2$ și $succ[10] = 11$, deoarece elementul $t[10] = 7$ poate fi adăugat înaintea unui subșir crescător maximal care începe cu $t[11] = 10 \geq 7$, deci $lmax[10] = 1 + lmax[11] = 2$;
- $lmax[9] = 2$ și $succ[9] = 11$, deoarece elementul $t[9] = 8$ poate fi adăugat înaintea unui subșir crescător maximal care începe cu $t[11] = 10 \geq 8$, deci $lmax[9] = 1 + lmax[11] = 2$;
- $lmax[8] = 3$ și $succ[8] = 9$, deoarece elementul $t[8] = 5$ poate fi adăugat înaintea oricăruia dintre subșirurile crescătoare maxime 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 subsir 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/>.