TEHNICA PROGRAMĂRII DINAMICE

1. Prezentare generală

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

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

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

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

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

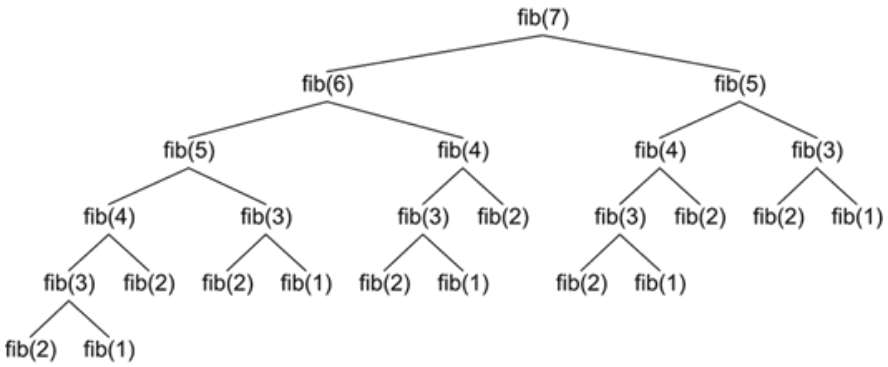
De exemplu, să considerăm şirul lui Fibonacci, definit recurent astfel:

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

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

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

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



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

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

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

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

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

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

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

2. Suma maximă într-un triunghi de numere

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

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

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

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

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

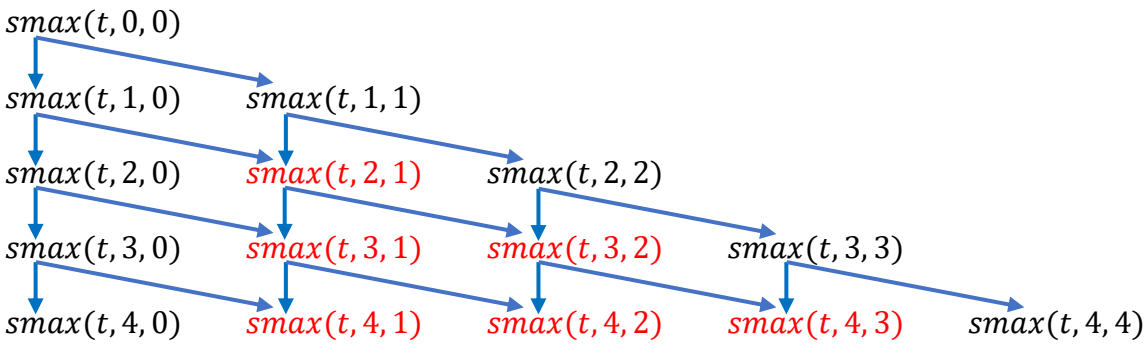
O altă variantă de rezolvare ar putea utiliza tehnica Backtracking pentru generarea tuturor traseelor care respectă cerinţele problemei şi selectarea celui care are suma elementelor maximă. Această variantă de rezolvare este corectă, dar ineficientă, deoarece are complexitatea exponenţială $\mathcal{O}(2^{n-1})$.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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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