

CURS 11

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 optimală*: 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, un tablou unidimensional sau bidimensional).

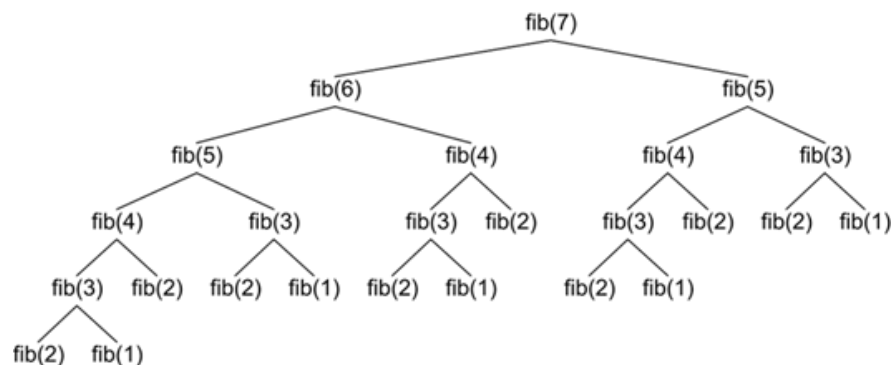
De exemplu, să considerăm șirul lui Fibonacci, definit recurent astfel:

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

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

```
unsigned long long int fib(unsigned int 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, astfel: vom utiliza un tablou unidimensional `f` pentru a memora termenii din șir, iar fiecare termen nou `f[i]` va fi calculat utilizând cei doi termeni anterior calculați, respectiv `f[i-2]` și `f[i-1]` (am presupus faptul că $1 \leq n \leq 100$):

```
unsigned long long int fib(unsigned int n)
{
    unsigned long long int f[101];
    unsigned int i;

    f[1] = 0;
    f[2] = 1;
    for(i = 3; i <= n; i++)
        f[i] = f[i-2] + f[i-1];
    return f[n];
}
```

Se observă faptul că tabloul `f` este completat 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**!

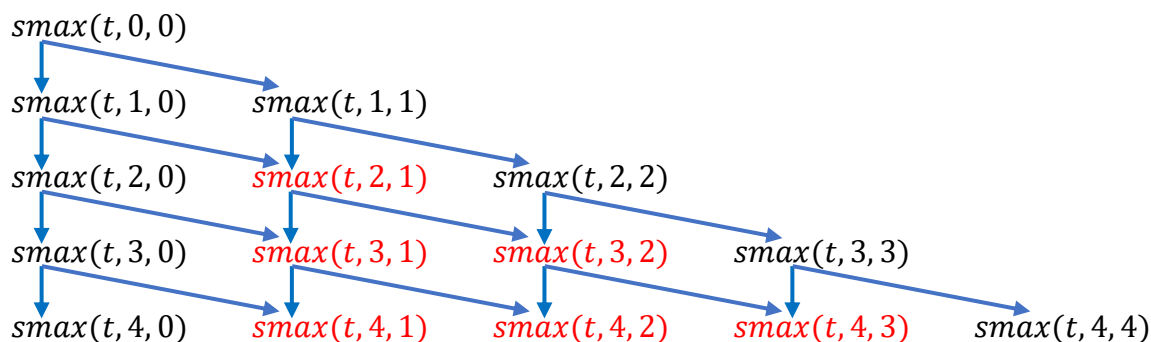
O altă variantă de rezolvare ar putea utiliza tehnica Backtracking pentru generarea tuturor traseelor care respectă cerințele problemei și selectarea celui care are suma elementelor maximă. Această variantă de rezolvare este corectă, dar ineficientă, deoarece are complexitatea exponențială $O(2^{n-1})$.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



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 $smax$, astfel: 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 C, considerând faptul că datele de intrare se citesc din fișierul text `triunghi.txt`, care conține pe prima linie dimensiunea n a unui triunghi de numere, iar următoarele n linii se află triunghiul de numere:

```

#include<stdio.h>
#include<stdlib.h>

int maxim(int a, int b) { return a > b ? a : b; }

int main()
{
    int n, i, j, t[100][100], smax[100][100];

    FILE* f = fopen("triunghi.txt", "r");

    fscanf(f, "%d", &n);
    for(i = 0; i < n; i++)
        for(j = 0; j <= i; j++)
            fscanf(f, "%d", &t[i][j]);
    fclose(f);

    for(i = 0; i < n; i++)
        smax[n-1][i] = t[n-1][i];

    for(i = n-2; i >= 0; i--)
        for(j = 0; j <= i; j++)
            smax[i][j] = t[i][j] + maxim(smax[i+1][j], smax[i+1][j+1]);

    printf("Suma maxima: %d\n", smax[0][0]);

    printf("\nTraseul cu suma maxima:\n");

    j = 0;
    for(i = 0; i < n-1; i++)
    {
        printf("t[%d][%d] -> ", i, j);
        if(smax[i+1][j+1] > smax[i+1][j]) j++;
    }

    printf("t[%d][%d]\n", i, j);

    return 0;
}

```

Complexitatea algoritmului prezentat este egală cu $\mathcal{O}(n^2)$, iar prin alocarea dinamică a tablourilor t și $smax$ sub forma unor tablouri bidimensionale triunghiulare de numere întregi s-ar putea optimiza spațiul de memorie utilizat.

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:

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

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 fie o structură de date auxiliară în care să salvăm soluția și apoi să o afișăm invers (variantă pe care o vom utiliza în implementarea de mai jos), fie o funcție recursivă!

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fișierul text `triunghi.txt`, care conține pe prima linie dimensiunea *n* a unui triunghi de numere, iar următoarele *n* linii se află triunghiul de numere:

```

#include<stdio.h>

int maxim(int a, int b) { return a > b ? a : b; }

int main()
{
    int n, i, j, k, t[100][100], smax[100][100], sol[100], pmax;

    FILE* f = fopen("triunghi.txt", "r");

    fscanf(f, "%d", &n);
    for(i = 0; i < n ; i++)
        for(j = 0; j <= i; j++)
            fscanf(f, "%d", &t[i][j]);
    fclose(f);

    smax[0][0] = t[0][0];
    for(i = 1; i < n; i++)
        for(j = 0; j < i+1; j++)
            if(j == 0)
                smax[i][j] = t[i][j] + smax[i-1][j];
            else
                if(j == i)
                    smax[i][j] = t[i][j] + smax[i-1][j-1];
                else
                    smax[i][j] = t[i][j] + maxim(smax[i-1][j],
                    smax[i-1][j-1]);

    pmax = 0;
    for(j = 1; j < n; j++)
        if(smax[n-1][j] > smax[n-1][pmax]) pmax = j;

    printf("Suma maxima: %d\n", smax[n-1][pmax]);

    j = pmax;
    k = 0;
    for(i = n-1; i > 0; i--)
    {
        sol[k++] = j;

        if(j == 0) continue;

        if(i == j) j--;
        else
            if(smax[i-1][j] < smax[i-1][j-1])
                j--;
    }
    sol[n-1] = 0;

    printf("\nTraseul cu suma maxima:\n");
    for(i = n-1; i > 0; i--)
        printf("t[%d][%d] -> ", n-1-i, sol[i]);
    printf("t[%d][%d]", n-1-i, sol[i]);

    return 0;
}

```


Complexitatea acestui algoritm este egală tot cu $\mathcal{O}(n^2)$, iar prin alocarea dinamică a tablourilor t și s_{\max} sub forma unor tablouri bidimensionale triunghiulare de numere întregi s-ar putea optimiza spațiul de memorie utilizat.

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:

- *variantea 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);
- *variantea 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);
- *variantea mixtă*, dacă se combină cele două variante anterioare.