CURS 5
TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

1. Prezentare generală

Tehnica de programare Divide et Impera constă în descompunerea repetată a unei probleme în două sau mai multe subprobleme de acelaşi tip până când se obţin probleme direct rezolvabile (etapa *Divide*), după care, în sens invers, soluţia fiecărei probleme se obţine combinând soluţiile subproblemelor în care a fost descompusă (etapa *Impera*).

Se poate observa cu uşurinţă faptul că tehnica Divide et Impera are în mod nativ un caracter recursiv, însă există şi cazuri în care ea este implementată iterativ.

Evident, pentru ca o problemă să poată fi rezolvată folosind tehnica Divide et Impera, ea trebuie să îndeplinească următoarele două condiţii:

1. condiţia *Divide*: problema poate fi descompusă în două (sau mai multe) subprobleme de acelaşi tip;
2. condiţia *Impera*: soluţia unei probleme se poate obţine combinând soluţiile subproblemelor în care ea a fost descompusă.

De obicei, subproblemele în care se descompune o problemă au dimensiunile datelor de intrare aproximativ egale sau, altfel spus, aproximativ egale cu jumătate din dimensiunea datelor de intrare ale problemei respective.

De exemplu, folosind tehnica Divide et Impera, putem calcula suma elementelor unui tablou unidimensional t format din n numere întregi, astfel:

1. împărţim tabloul t în două jumătăţi până când obţinem tablouri cu un singur element (caz în care suma se poate calcula direct, fiind chiar elementul respectiv);
2. în sens invers, calculăm suma elementelor dintr-un tablou adunând sumele elementelor celor două sub-tablouri în care el a fost descompus.

Se observă imediat faptul că problema verifică ambele condiţii menţionate mai sus!

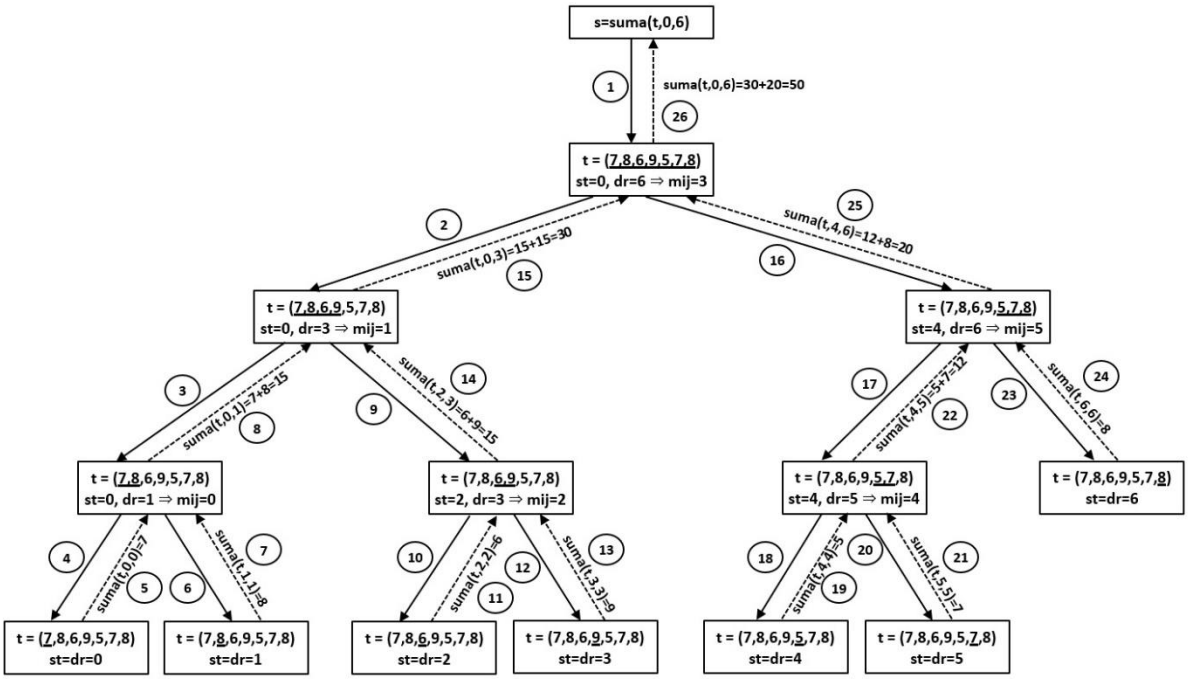
Pentru a putea manipula uşor cele două sub-tablouri care se obţin în momentul împărţirii tabloului t în două jumătăţi, vom considera faptul că tabloul curent este secvenţa cuprinsă între doi indici st şi dr, unde $st \leq dr$. Astfel, indicele mij al mijlocului tabloului curent este aproximativ egal cu $[(st + dr)/2]$, iar cele două sub-tablouri în care va fi descompus tabloul curent sunt secvenţele cuprinse între indicii st şi mij, respectiv $mij + 1$ şi dr. Considerând $suma(t, st, dr)$ o funcţie care calculează suma secvenţei $t[st], t[st + 1], \dots, t[dr]$, putem să o definim în manieră Divide et Impera astfel:

$$suma(t, st, dr) = \begin{cases} t[st], & \text{dacă } st = dr \quad (1) \\ suma(t, st, mij) + suma(t, mij + 1, dr), & \text{dacă } st < dr \quad (2) \end{cases}$$

unde $mij = [(st + dr)/2]$.

Considerând tabloul t cu n elemente ca fiind indexat de la 0, suma s a tuturor elementelor sale se va obţine în urma apelului $s = suma(t, 0, n - 1)$.

De exemplu, pentru tabloul $t = (7, 8, 6, 9, 5, 7, 8)$ având $n = 7$ elemente, pentru a calcula suma elementelor sale, se vor efectua următoarele apeluri recursive:



În figura de mai sus, săgeţile "pline" reprezintă apelurile recursive, efectuate folosind relaţia (2) de mai sus, în timp ce săgeţile "întrerupte" reprezintă revenirile din apelurile recursive, care au loc în momentul în care se ajunge la o subproblemă direct rezolvabilă folosind relaţia (1) de mai sus. Ordinea în care se execută apelurile şi revenirile este indicată prin numerele scrise în cercuri. Numerele subliniate din tabloul t reprezintă secvenţa curentă (care se prelucrează în apelul respectiv).

2. Forma generală a unui algoritm de tip Divide et Impera

Plecând chiar de la exemplul se mai sus, se poate deduce foarte uşor forma generală a unui algoritm de tip Divide et Impera aplicat asupra unui tablou unidimensional t şi implementat folosind o funcţie recursivă:

```
tip_de_date divimp(t, st, dr)
{
   if (sub-problema curentă este direct rezolvabilă)
         return valoare;
   else
   {
         int mij = (st + dr) / 2;
         sol_st = divimp(t, st, mij);
         sol_dr = divimp(t, mij+1, dr);
         return soluție(sol_st, sol_dr);
   }
}
```

Vizavi de algoritmul general prezentat mai sus trebuie făcute câteva precizări:

- există algoritmi Divide et Impera în care nu sunt utilizate tablouri (de exemplu, calculul lui a^n - https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/), dar aceştia sunt mult mai rari decât cei în care sunt utilizate tablouri;
- de obicei, o subproblemă este direct rezolvabilă dacă secvenţa curentă din tabloul `t` este vidă `st>dr` sau are un singur element `st=dr`;
- variabila `mij` va conţine indicele mijlocului secvenţei `t[st]`, `t[st+1]`, ..., `t[dr]`, operatorul "`/`" calculând câtul împărţirii;
- variabilele `sol_st` şi `sol_dr` vor conţine soluţiile celor două subproblemele în care se descompune problema curentă, iar `soluţie(sol_st, sol_dr)` este o funcţie care determină soluţia problemei curente combinând soluţiile subproblemelor în care aceasta a fost descompusă (în unele cazuri, nu este necesară o funcţie, ci se poate folosi o simplă expresie);
- dacă funcţia `divimp` nu furnizează nicio valoare (este de tip `void`), atunci vor lipsi variabilele `sol_st` şi `sol_dr`, precum şi cele două instrucţiuni `return`.

Aplicând algoritmul general pentru a calcula suma elementelor dintr-un tablou unidimensional, vom obţine următoarea implementare în limbajul C:

```
int suma_divimp(int t[], int st, int dr)
{
    int mij, sol_st, sol_dr;

    if(st == dr)
        return t[st];
    else
    {
        mij = (st + dr)/2;
        sol_st = suma_divimp(t, st, mij);
        sol_dr = suma_divimp(t, mij + 1, dr);
        return sol_st + sol_dr;
    }
}
```

Aşa cum am menţionat în observaţiile anterioare, nu a mai fost necesară implementarea unei funcţii `soluţie`, ci a fost suficientă utilizarea unei expresii simple.

3. Determinarea complexităţii unui algoritm de tip Divide et Impera

Deoarece algoritmii de tip Divide et Impera sunt implementaţi, de obicei, folosind funcţii recursive, determinarea complexităţii computaţionale a unui astfel de algoritm este mai complicată decât în cazul algoritmilor iterativi.

Primul pas în determinarea complexităţii unei algoritm Divide et Impera îl constituie determinarea unei relaţii de recurenţă care să exprime complexitatea $T(n)$ a rezolvării unei probleme având dimensiunea datelor de intrare egală cu n în raport de timpul necesar rezolvării subproblemelor în care aceasta este descompusă şi de complexitatea operaţiei de combinare a soluţiilor lor pentru a obţine soluţia problemei iniţiale. Presupunând faptul că orice problemă se descompune în a subprobleme, fiecare având

dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{b}$, iar împărţirea problemei curente în subprobleme şi combinarea soluţiilor subproblemelor pentru a obţine soluţia sa se realizează folosind un algoritm cu complexitatea $f(n)$, se obţine foarte uşor forma generală a relaţiei de recurenţă căutate:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3}$$

unde $a \geq 1$, $b > 1$ şi $f(n)$ este o funcţie asimptotic pozitivă (i.e., există $n_0 \in \mathbb{N}$ astfel încât pentru orice $n \geq n_0$ avem $f(n) \geq 0$) şi presupunem faptul că $T(1) \in \mathcal{O}(1)$.

Reluăm algoritmul Divide et Impera care calculează suma elementelor dintr-un tablou unidimensional cu n elemente (i.e., $dr - st + 1 \leq n$) pentru a evidenţia complexităţile componentelor sale:

```
int suma_divimp(int t[], int st, int dr)
{
    int mij, sol_st, sol_dr;

    if(st == dr)
        return t[st];          } T(1)=1
    else                  O(1)
    {
        mij = (st + dr)/2;                ≈ T(n/2)

        sol_st = suma_divimp(t, st, mij);
        sol_dr = suma_divimp(t, mij + 1, dr);

        return sol_st + sol_dr;          ≈ T(n/2)
    }
}
                  O(1)
```

Putem observa faptul că o problemă având dimensiunea datelor de intrare egală cu n se descompune în $a = 2$ subprobleme, fiecare având dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{2}$ (deci $b = 2$), iar împărţirea problemei curente în subprobleme şi combinarea soluţiilor subproblemelor pentru a obţine soluţia sa se realizează folosind un algoritm cu complexitatea $\mathcal{O}(1)$, dată de cele 4 instrucţiuni subordonate instrucţiunii `else`, deci relaţia de recurenţă este următoarea:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \underbrace{\mathcal{O}(1) + \mathcal{O}(1)}_{f(n)} = 2T\left(\frac{n}{2}\right) + \mathcal{O}(2) = 2T\left(\frac{n}{2}\right) + 1 \tag{4}$$

De asemenea, se observă faptul că $T(1) \in \mathcal{O}(1)$.

Al doilea pas în determinarea complexităţii unei algoritm Divide et Impera îl constituie rezolvarea relaţiei de recurenţă de mai sus, utilizând diverse metode matematice, pentru a determina expresia analitică a lui $T(n)$. În continuare, vom prezenta două dintre cele mai utilizate metode, simplificate, respectiv *iterarea directă a relaţiei de recurenţă* şi *teorema master*.

În cazul primei metode, bazată pe *iterarea directă a relației de recurență*, vom presupune faptul că n este o putere a lui b, după care vom itera relația de recurență până când vom ajunge la $T(1)$ sau $T(0)$, care sunt ambele egale cu 1 fiind complexitățile unor probleme direct rezolvabile.

De exemplu, pentru a rezolva relația de recurență (4), vom presupune că $n = 2^k$ și apoi o vom itera, astfel:

$$\begin{aligned} T(n) = T(2^k) &= 2T(2^{k-1}) + 1 = 2[2T(2^{k-2}) + 1] + 1 = 2^2T(2^{k-2}) + 2 + 1 = \\ &= 2^2[2T(2^{k-3}) + 1] + 2 + 1 = 2^3T(2^{k-3}) + 2^2 + 2 + 1 = \cdots = \\ &= 2^kT(2^0) + 2^{k-1} + \cdots + 2 + 1 = 2^k + 2^{k-1} + \cdots + 2 + 1 = 2^{k+1} - 1 = \\ &= 2 \cdot 2^k - 1 = 2n - 1 \end{aligned}$$

În concluzie, am obținut faptul că $T(n) = 2n - 1$, deci complexitatea algoritmului Divide et Impera pentru calculul sumei elementelor dintr-un tablou unidimensional cu n elemente este $\mathcal{O}(2n - 1) \approx \mathcal{O}(n)$.

A doua metodă constă în utilizarea *teoremei master* pentru a afla direct soluția analitică a unei relații de recurență de tipul (3).

Teorema master: Fie o relație de recurență de forma (3) și presupunem faptul că $f \in \mathcal{O}(n^p)$. Atunci:

a) dacă $p < \log_b a$, atunci $T(n) \in \mathcal{O}(n^{\log_b a})$;
b) dacă $p = \log_b a$, atunci $T(n) \in \mathcal{O}(n^p \log_2 n)$;
c) dacă $p > \log_b a$ și $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare, atunci $T(n) \in \mathcal{O}(f(n))$.

Exemple:

1. Pentru a rezolva relația de recurență (4), observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(1) = \mathcal{O}(n^0) \Rightarrow p = 0 < \log_b a = \log_2 2 = 1 \xrightarrow{\text{Cazul } a)} T(n) \in \mathcal{O}(n)$.

2. Pentru a rezolva relația de recurență $T(n) = 2T\left(\frac{n}{2}\right) + n$, observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(n) \Rightarrow p = 1 = \log_b a = \log_2 2 = 1 \xrightarrow{\text{Cazul } b)} T(n) \in \mathcal{O}(n \log_2 n)$.

3. Pentru a rezolva relația de recurență $T(n) = 2T\left(\frac{n}{2}\right) + n^2$, observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(n^2) \Rightarrow p = 2 > \log_b a = \log_2 2 = 1$. În acest caz, trebuie să verificăm și faptul că $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare $\Leftrightarrow \exists c < 1$ astfel încât $2\frac{n^2}{4} \leq cn^2 \Leftrightarrow \exists c < 1$ astfel încât $\frac{n^2}{2} \leq cn^2 \Leftrightarrow \exists c < 1$ astfel încât $n^2 \leq 2cn^2 \Leftrightarrow \exists c < 1$ astfel încât $1 \leq 2c$, ceea ce este adevărat, de exemplu pentru $c = \frac{1}{2}$ sau, în general, pentru orice $c \in \left[\frac{1}{2}, 1\right)$. Astfel, obținem că $T(n) \in \mathcal{O}(n^2)$.

Varianta prezentată mai sus a teoremei master este una simplificată, dar care acoperă majoritatea cazurilor întâlnite în practică. O variantă extinsă a acestei teoreme este

prezentată aici: https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms). Totuşi, nici varianta extinsă nu acoperă toate cazurile posibile. Mai mult, teorema de master nu poate fi utilizată dacă subproblemele nu au dimensiuni aproximativ egale, fiind necesară utilizarea teoremei Akra-Bazzi (https://en.wikipedia.org/wiki/Akra-Bazzi_method).

Observaţie importantă: În general, algoritmii de tip Divide et Impera au complexităţi mici, de tipul $\mathcal{O}(\log_2 n), \mathcal{O}(n)$ sau $\mathcal{O}(n \log_2 n)$, care se obţin datorită faptului că o problemă este împărţită în două subprobleme de acelaşi tip cu dimensiunea datelor de intrare înjumătăţită faţă de problema iniţială şi, mai mult, subproblemele nu se suprapun! Dacă aceste condiţii nu sunt îndeplinite simultan, atunci complexitatea algoritmul poate să devină foarte mare, de ordin exponenţial! De exemplu, o implementare recursivă, de tip Divide et Impera, care să calculeze termenul de rang n al şirului lui Fibonacci ($F_n = F_{n-1} + F_{n-2}$ şi $F_0 = 0, F_1 = 1$) nu respectă condiţiile precizate anterior (dimensiunile subproblemelor nu sunt aproximativ jumătate din dimensiunea unei probleme şi subproblemele se suprapun, respectiv mulţi termeni vor fi calculaţi de mai multe ori), ceea ce va conduce la o complexitate exponenţială! Astfel, relaţia de recurenţă pentru complexitatea algoritmului este $T(n) = 1 + T(n-1) + T(n-2)$. Cum $T(n-1) > T(n-2)$, obţinem că $2T(n-2) < T(n) < 2T(n-1)$. Iterând dubla inegalitate, obţinem $2^{\frac{n}{2}} < T(n) < 2^n$, ceea ce dovedeşte faptul că implementarea recursivă are o complexitate exponenţială. Totuşi, există mai multe metode iterative cu complexitate liniară pentru rezolvarea acestei probleme: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/.