#2 Metoda Divide et Impera:
Prezentare generală

Obiective:

1. Prezentarea principiului de rezolvare a problemelor folosind metoda Divide et Impera.
2. Exersarea metodei prin implementarea de probleme specifice:
   - Determinarea valorii maxime a unei mulţimi de numere întregi
   - Căutarea binară a unei valori într-o mulţime ordonată crescător
   - Căutarea unei sume într-o mulţime de numere întregi
3. Verificarea înţelegerii conceptului de rezolvare prin abordarea unor probleme de dificultate asemănătoare cu cele prezentate.

2.1 Metoda Divide et Impera

Metoda Divide et Impera reprezintă o tehnică de programare care poate fi aplicată acelor probleme pentru care există o modalitate de descompunere succesivă în două sau mai multe subprobleme de dimensiuni mai mici (şi implicit mai uşoare), subprobleme care trebuie rezolvate la rândul lor. Procedeul de împărţire succesivă (**etapa divide**) continuă până când se ajunge la subprobleme care admit o rezolvare directă şi imediată (spre exemplu, probleme de dimensiuni mici, `n=1,2`). Soluţia problemei iniţiale se obţine combinând soluţiile rezultate prin rezolvarea subproblemelor (**etapa impera**).

Istoric şi explicația denumirii

Expresia „Divide et Impera” este atribuită în general lui Iulius Cezar potrivit căruia adversarii trebuie împărţiţi şi divizaţi de luptele dintre ei. Puterea este împărţită între grupuri care sunt astfel mai slabe individual decât unite sub aceeaşi strategie.

Această strategie de a conduce a fost atribuită de-a lungul timpului şi altor suverani, de la Louis al XI-lea până la dinastia Habsburg[1]. Recent, reprezintă şi o strategie adaptată în economie în contextul unei pieţe competitive cu mai mulţi jucători.

În domeniul programării, ideea a fost introdusă de (Karatsuba şi Ofman) în anii 1960 sub forma unui algoritm de înmulţire a două numere de `n` digiţi cu o complexitate sub $O(n^2)$.

Etapele de rezolvare ale unei probleme folosind această tehnică sunt:

1. **Divide**: problema iniţială este descompusă într-un număr de subprobleme de aceeaşi natură însă de dimensiuni mai mici. Procedeul continuă de manieră recursivă până când se ajunge la probleme de dimensiuni elementare (de exemplu, `n=1,2`, etc.) care pot fi rezolvate direct.
2. **Impera**: problemele de dimensiuni elementare sunt rezolvate direct iar soluţiile lor sunt combinate pentru a obţine soluţiile subproblemelor de dimensiuni mai mari, inclusiv soluţia problemei iniţiale.

Etapele succesive de împărţire a problemei în subprobleme de aceeaşi natură însă de dimensiuni mai mici permit atacarea unor probleme relativ grele prin reducerea lor la cazuri elementare care pot fi rezolvate direct şi, folosind o tehnică de combinare a soluţiilor parţiale, putem obţine soluţia problemei iniţiale. Procedura generală descriind tehnica Divide et Impera este prezentată în continuare folosind limbajul pseudocod. Varianta prezentată foloseşte recursivitatea însă acest lucru nu este obligatoriu şi nici recomandat în practică.

[1] http://en.wikipedia.org/wiki/Divide_and_rule

Pseudocod

```
    procedura DivideEtImpera(problema P, int n)
1       *) problema P poate fi rezolvata direct?
2       *) afirmativ pentru valori mici ale lui n
3       dacă n <= c atunci
4           soluţie ← RezolvăDirect(P, n)
5           întoarce soluţie
6       altfel
7           *) împarte P în m subprobleme
8           (P,n) = (P1,n1)U(P2,n2)U ... U(Pm,nm)
9           *) rezolvă subproblemele
10          pentru i ← 1, m execută
11              soluţiei ← DivideEtImpera(Pi,ni)
12          ■
13          *) combină soluţiile subproblemelor
14          *) pentru a obţine soluţia problemei P
15          soluţie ← Combină(
16                          soluţie1,
17                          soluţie2,
18                          ...
19                          soluţiem
20                      )
21          întoarce soluţie
22      ■
23  sf.procedură
```

Procedurile `RezolvăDirect(...)` şi `Combină(...)` depind de natura problemei şi implementează rezolvarea cazurilor simple respectiv combinarea soluţiilor parţiale pentru a obţine soluţia la o problemă de o dimensiune mai mare. Câteva observaţii generale privind metoda sunt punctate în continuare:

1. Nu toate problemele pot fi rezolvate folosind tehnica Divide et Impera ci numai acelea care permit o descompunere în subprobleme **de aceeaşi natură**.
2. Prin modalitatea de descompunere şi rezolvare a subproblemelor, tehnica este adaptată pentru execuţie multi-procesor întrucât subproblemele distincte pot fi rezolvate distribuit şi independent, soluţiile lor fiind apoi combinate.
3. Un dezavantaj al metodei în forma prezentată este dat de folosirea mecanismului recursivităţii ce presupune apeluri succesive de funcţii, recomandându-se pe cât posibil evitarea acestora.

2.2 Determinarea valorii maxime dintr-o mulţime de numere întregi

Problemă exemplu

Determinarea valorii maxime

Fie o mulţime de numere întregi de dimensiune `n`. Să se găsească elementul de valoare maximă.

Datele vor fi citite dintr-un fişier de intrare ce conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulţimii separate printr-un spaţiu. Fişierul de ieşire va conţine o singură linie reprezentând elementul de valoare maximă.

Intrare	Ieşire
7 3 1 8 2 7 3 5	8

Pentru a determina valoarea maximă pornim de la problema iniţială de dimensiune `n`: Cum identificăm elementul maxim aflat între elementele cu indicii `0` şi `n-1` din şir? În acest moment putem aplica principiul diviziunii şi anume vom împărţi problema iniţială de dimensiune `n` în două subprobleme de aceeaşi natură dar de dimensiuni mai mici, respectiv jumătate din dimensiunea problemei iniţiale, `n/2`. Astfel, etapa divide va consta în împărţirea şirului iniţial în două subşiruri ale căror limite vor fi date de indicii `0`, `(n-1)/2` şi `(n-1)/2+1`, `n-1`. Dacă vom rezolva aceste două subprobleme vom avea drept soluţii valorile maxime din cele două subşiruri, max_1 şi max_2. Cunoscând aceste valori le putem combina (etapa impera) pentru a obţine soluţia la problema de dimensiune `n`: valoarea maximă a elementelor dintre indicii `0` şi `n-1` va fi cea mai mare valoare dintre max_1 şi max_2. Pentru a obţine max_1 şi max_2 procedăm la divizări succesive ale celor două şiruri (etapa divide) până când ajungem la subşiruri de dimensiuni mici `n=1` sau `n=2` pentru care soluţia este imediată.

Pseudocod

```
    int DeterminăMaxim(int[] a, int p, int q)
1       *) p şi q sunt limitele între care căutăm
2       *) valoarea maximă
3       dacă p == q atunci
4           întoarce a[p]
5       altfel
6           m ← (p + q) / 2
7           max1 ← DeterminăMaxim(a, p, m)
8           max2 ← DeterminăMaxim(a, m + 1, q)
9           întoarce max1 < max2 ? max2 : max1
10      ■
11  sf.procedură
```

Procedura `DeterminăMaxim` identifică elementul de valoare maximă în subşirul din vectorul `a` specificat de indicii `p≤q`, interval închis. Vom apela procedura de determinare a maximului cu limitele `0` și `n-1` astfel: `DeterminăMaxim(a,0,n-1)`.

Un exemplu care ilustrează grafic parcursul algoritmului pentru fişierul de intrare din enunţul problemei este prezentat în continuare. Aplicăm succesiv etapa Divide până când ajungem la subşiruri de dimensiuni mici (`n=1` sau `n=2`) pentru care elementul maxim poate fi determinat imediat efectuând cel mult o comparaţie (Figura 2.1).

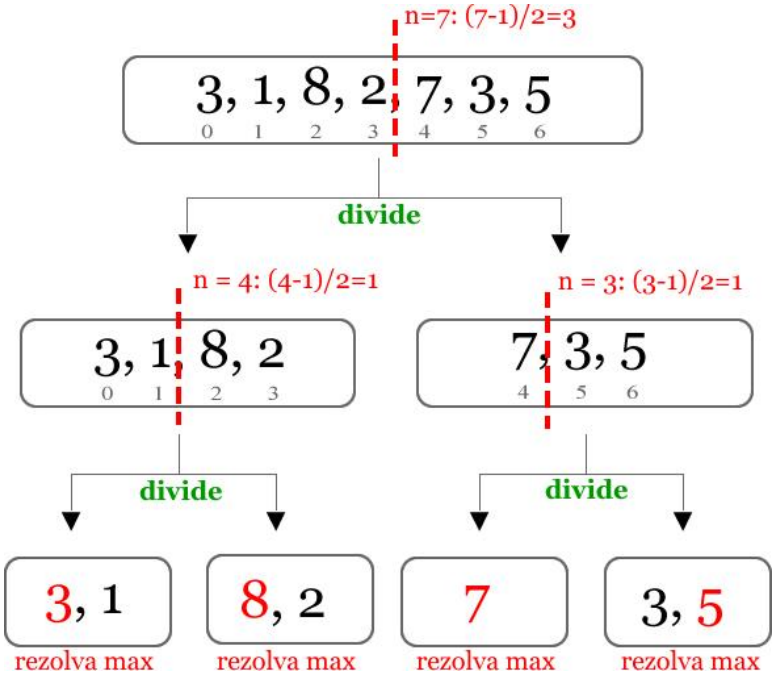

Figura 2.1. Aplicarea etapei Divide pentru problema determinării valorii maxime a unei mulțimi de numere.

În continuare vom aplica principiul impera care constă în combinarea soluțiilor (i.e., valorile maxime) ale submulțimilor obţinute în mod recursiv până ajungem la mulțimea iniţială a problemei de dimensiune `n` (Figura 2.2). Combinarea constă în efectuarea unei singure comparații dintre valorile maxime obținute pentru două subprobleme de dimensiune jumatate din dimensiunea problemei pentru care căutăm soluția. De exemplu, pentru a obține soluția subproblemei `{3,1,8,2}` de dimensiune `4` vom calcula maximul dintre valorile `3` și `8` reprezentând soluțiile subproblemelor `{3,1}` și `{8,2}` de dimensiune `2`.

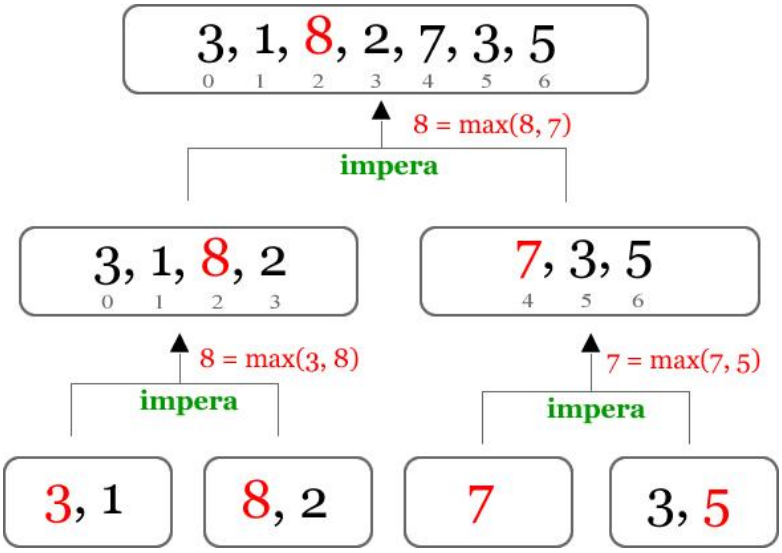

Figura 2.2. Aplicarea etapei Impera pentru problema determinării valorii maxime a unei mulțimi de numere.

Pentru analiza complexităţii vom vizualiza divizările succesive realizate de către algoritm. Plecând de la problema de dimensiune `n` vom obţine două subprobleme de dimensiune `n/2`, fiecare dintre acestea fiind iarăşi împărţite în câte două probleme de dimensiune `n/4`, etc. La pasul `j` vom avea 2^j probleme de dimensiune $n/2^j$.

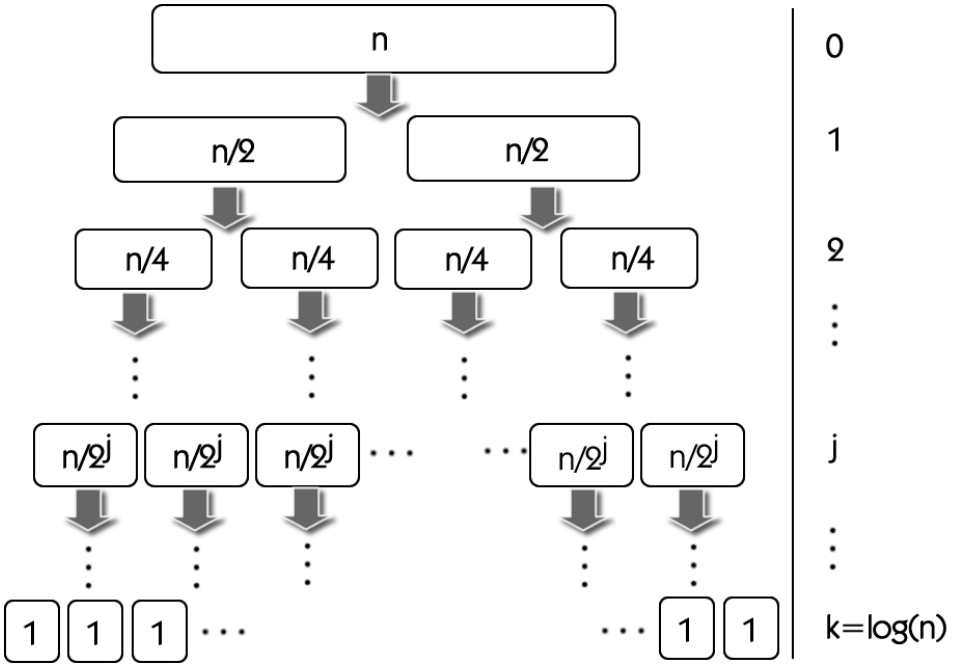

Figura 2.3. Diviziunile succesive pentru analiza complexității problemei determinării valorii maxime a unei mulțimi de numere.

Putem scrie următoarea relaţie de recurenţă pentru timpul de execuţie necesar algoritmului `DeterminăMaxim` pentru a rezolva o problemă de dimensiune n astfel:

$$T(n)=\begin{cases}\Theta(1) & n=1\\ 2\cdot T(n/2)+\Theta(1) & n>1\end{cases}$$

pentru care avem succesiv:

$$\begin{array}{l|l}
T(n)=2\cdot T(n/2)+\Theta(1) & \cdot 1\\
T(n/2)=2\cdot T(n/4)+\Theta(1) & \cdot 2\\
\vdots & \\
T(n/2^{j})=2\cdot T(n/2^{j+1})+\Theta(1) & \cdot 2^{j}\\
\vdots & \\
T(1)=\Theta(1) & \cdot 2^{\lfloor \log(n)\rfloor}
\end{array}$$

Adunând ecuaţiile obţinem expresia pentru T(n):

$$T(n)=2^{\lfloor \log(n)\rfloor}\cdot\Theta(1)+\left(\sum_{j=0}^{\lfloor \log(n)\rfloor}2^{j}\right)\cdot\Theta(1)$$

$$T(n)=n\cdot\Theta(1)+\frac{2^{\lfloor \log(n)\rfloor+1}-1}{2-1}\cdot\Theta(1)=(n+2n-1)\cdot\Theta(1)=\Theta(n)$$

2.3 Căutarea binară

Problemă exemplu

Căutarea binară

Fie o mulţime de numere întregi `a` de dimensiune `n` ale cărei elemente sunt ordonate crescător. Să se determine dacă un element `x` element se regăseşte în cadrul mulţimii.

Datele vor fi citite dintr-un fişier care conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulţimii separate printr-un spaţiu. A treia linie conţine elementul `x` ce va fi căutat. Fişierul de ieşire va conţine o singură linie reprezentând poziţia pe care s-a găsit elementul `x` în şir respectiv `-1` în caz contrar.

Intrare	Ieşire
7 1 2 3 3 5 7 8 7	5

Căutarea elementului `x` într-un vector de numere a poate fi realizată în timp liniar parcurgând fiecare element din vector şi comparându-l cu valoarea căutată. Cazul cel mai defavorabil pentru acest algoritm apare atunci când elementul `x` nu este găsit întrucât sunt necesare `n` comparaţii. Complexitatea acestui algoritm naiv de căutare este prin urmare liniară $\Theta(n)$.

Pseudocod

```
   int Caută(int[] a, int n, int x)
1      pentru i ← 0, n-1 execută
2      |   *) compară x cu fiecare element din a
3      |   dacă a[i] == x atunci întoarce i
4      ■
5      întoarce -1
6  sf.procedură
```

Cunoscând faptul că valorile vectorului sunt deja sortate în ordine crescătoare putem aplica principiul metodei Divide et Impera pentru a obţine un algoritm mai rapid. Să efectuăm de exemplu prima comparaţie dintre elementul pe care îl căutăm `x` şi valoarea aflată la mijlocul şirului `a`, `a[(n-1)/2]`. Dacă `x` este mai mic atunci este evident că nu mai este necesar să îl comparăm pe `x` cu elementele aflate după poziţia `(n-1)/2` întrucât toate acestea vor fi mai mari. Luăm astfel decizia de a căuta valoarea `x` în prima jumătate a şirului `a` eliminând jumătate din comparaţiile pe care primul algoritm le-ar fi efectuat. Dacă `x` ar fi fost mai mare decât valoarea de la mijlocul şirului, `a[(n-1)/2]`, atunci am fi luat decizia căutării lui `x` în a doua jumătate a şirului `a`. Oricare jumătate am alege-o, repetăm acelaşi principiu: comparăm elementul `x` cu valoarea aflată la mijlocul subşirului. Reducem astfel problema (etapa divide) la o subproblemă de dimensiune jumătate din dimensiunea problemei iniţiale până când:

- Fie găsim condiţia de egalitate dintre `x` şi valoarea de la mijlocul subşirului curent pe care îl procesăm.
- Fie nu mai putem divide şirul în continuare întrucât am ajuns la o subproblemă de dimensiune `n=1`.

Algoritmul de căutare binară folosind această idee este prezentat în continuare. Procedura `CautăBinar(...)` primeşte ca argumente şirul `a`, valoarea de căutat `x` şi doi indici `p` şi `q` reprezentând limitele subşirului din `a` între care căutăm valoarea `x`. Procedura va fi apelată `CautăBinar(a,0,n-1,x)`.

Pseudocod

```
    *) p,q reprezintă indicii între care căutăm
    *) valoarea x în şirul a
    int CautăBinar(int[] a, int p, int q, int x)
1       *) condiţia de oprire (x nu a fost găsit)
2       dacă p > q atunci întoarce -1
3       *) calculăm mijlocul intervalului
4       m ← (p + q) / 2
5       dacă a[m] == x atunci întoarce m
6       altfel
7        dacă a[m] < x atunci
8           întoarce CautăBinar(a, m + 1, q, x)
9        altfel
10          întoarce CautăBinar(a, p, m - 1, x)
11       ■
12      ■
13  sf.procedură
```

O versiune iterativă a algoritmului de căutare binară este prezentată în continuare.

Pseudocod

```
    int CautăBinar-Iterativ(int[] a, int n, int x)
1       p ← 0
2       q ← n-1
3       cât timp p <= q execută
4           *) calculăm mijlocul intervalului
5           m ← (p + q) / 2
6           dacă a[m] == x atunci
7               întoarce m
8           ■
9           dacă a[m] < x atunci
10              p ← m + 1
11          altfel
12              q ← m - 1
13          ■
14      ■
15      întoarce -1 *) nu am găsit elementul x
16  sf.procedură
```

Pentru a analiza complexitatea procedurii de căutare binară vom porni prin a determina numărul maxim de diviziuni succesive pe care algoritmul le realizează. Problema iniţială de dimensiune `n` este redusă la o subproblemă de dimensiune `n/2` care, la rândul ei, este redusă la una de dimensiune `n/4`, etc.

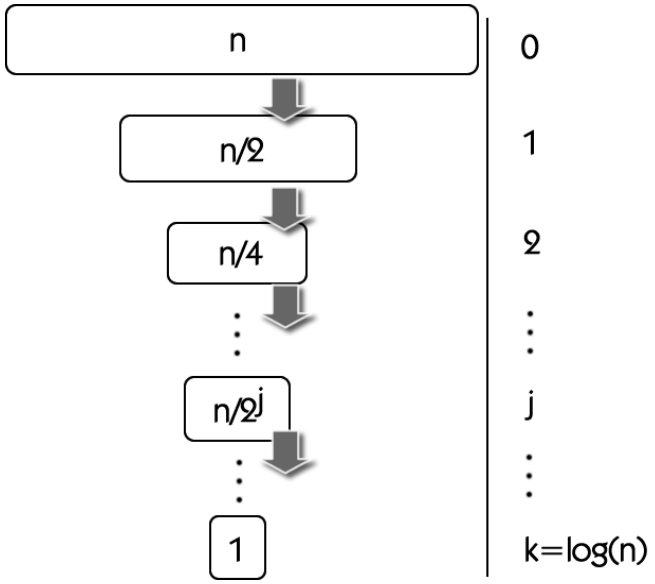

Figura 2.4. Diviziunile succesive ale dimensiunii problemei pentru analiza complexității căutării binare.

La fiecare pas dimensiunea subşirului în care căutăm valoarea `x` (sau, echivalent, dimensiunea subproblemei pe care o rezolvăm) se reduce la jumătate. Drept urmare, la pasul `j` vom căuta valoarea `x` într-un subşir de dimensiune $n/2^j$. Numărul maxim de paşi `k` până la terminarea algoritmului este obţinut atunci când valoarea căutată `x` nu se află în şir, respectiv când ajungem prin divizări succesive la un subşir de dimensiune `1`:

$$\left\lfloor \frac{n}{2^k} \right\rfloor = 1$$

Aplicând inegalitatea $x-1<\lfloor x \rfloor \leq x$ obţinem succesiv:

$$\frac{n}{2^k}-1<\left\lfloor \frac{n}{2^k} \right\rfloor \leq \frac{n}{2^k}$$

$$\frac{n}{2^k}-1<1\leq \frac{n}{2^k}$$

$\begin{cases} \frac{n}{2^k} < 2 \\ 1 \le \frac{n}{2^k} \end{cases}$ sau, echivalent, $\begin{cases} \log(n) - 1 < k \\ k \le \log(n) \end{cases}$

$\log(n) - 1 < k \le \log(n)$ şi, în final:

$k = \lfloor \log(n) \rfloor$

De unde putem deduce complexitatea algoritmului de căutare binară ca fiind logaritmică, `O(log(n))`. Observăm o reducere substanţială a complexităţii faţă de algoritmul de căutare liniară care ne furniza acelaşi rezultat într-un timp liniar `O(n)`.

Implementarea în limbajul C# a funcțiilor de căutare binară este prezentată în continuare, urmată de o analiză comparativă a timpului de execuție (Figura 2.5).

```
class CautareBinara
{
    /// <summary>
    /// Cauta binar elementul x in vectorul a.
    /// (implementare recursiva)
    /// Vectorul a trebuie sa fie in prealabil sortat crescator.
    /// </summary>
    /// <returns>Pozitia in sir pe care se afla elementul x
    ///  sau -1 in caz contrar</returns>
    public static int CautaBinar_Recursiv(
              int[] a, int left, int right, int x)
    {
        if (left > right) return -1; // x nu se afla in sir
        int mid = (left + right) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x)
             return CautaBinar_Recursiv(a, mid + 1, right, x);
        else return CautaBinar_Recursiv(a, left, mid - 1, x);
    }

    /// <summary>
    /// Cauta binar elementul x in vectorul a.
    /// (implementare iterativa)
    /// Vectorul a trebuie sa fie in prealabil sortat crescator.
    /// </summary>
    /// <param name="a"></param>
    /// <param name="x"></param>
    /// <returns>Pozitia in sir pe care se afla elementul x
    /// sau -1 in caz contrar</returns>
```

```
    public static int CautaBinar_Iterativ(int[] a, int x)
    {
        int p = 0;
        int q = a.Length - 1;
        while (p <= q)
        {
            int mid = (p + q) / 2;
            if (a[mid] == x) return mid;
            if (a[mid] < x)
                p = mid + 1;
            else q = mid - 1;
        }
        return -1; // x nu se afla in sir
    }
}
```

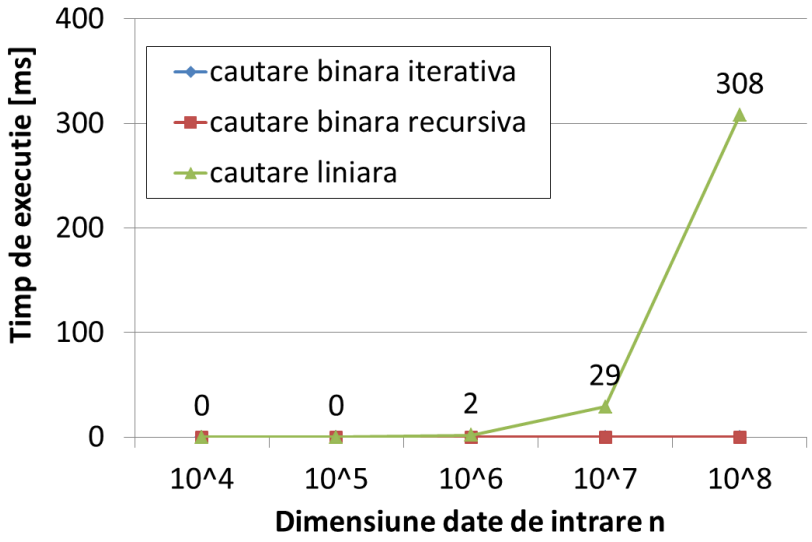

Figura 2.5. Timpul de execuție în milisecunde obținut pentru rularea căutării binare și căutării liniare pentru diverse dimensiuni ale datelor de intrare. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

2.4 Complexitatea algoritmilor Divide et Impera

Tehnica Divide et Impera poate fi descrisă ca o serie de apeluri recursive la subprobleme de aceeaşi natură de dimensiuni mai mici, ceea ce face ca și timpul de execuţie să poată fi descris printr-o relaţie de recurenţă (Cormen et al., 2000) (p. 46):

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ a \cdot T(n/b) + D(n) + I(n) & n > c \end{cases}$$

unde `D(n)` reprezintă timpul necesar aplicării etapei divide iar `I(n)` timpul necesar aplicării etapei impera. Soluționarea acestei recurențe poate fi realizată aplicând teorema Master (Cormen et al., 2000) (p. 54).

Teoremă

Teorema Master

Fie $a \geq 1$ şi $b > 1$ două constante, f(n) o funcţie şi T(n) o funcţie definită prin recurenţa:

$$T(n) = a \cdot T(n/b) + f(n)$$

Atunci T(n) poate fi delimitată asimptotic astfel:

1. Dacă $f(n) = O(n^{\log_b a - \varepsilon})$ pt. $\varepsilon > 0$ atunci $T(n) = \Theta(n^{\log_b a})$
2. Dacă $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pt. $\varepsilon > 0$ şi dacă $a \cdot f(n/b) \leq c \cdot f(n)$ pt. $c < 1$ atunci $T(n) = \Theta(f(n))$
3. Dacă $f(n) = \Theta(n^{\log_b a})$ atunci $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

Exemplu

Fie funcţia $T(n) = 8 \cdot T(n/2) + 3n$.

Avem a=8, b=2, f(n)=3n, $n^{\log_b a} = n^{\log_2 8} = n^3$

Cum exista un $\varepsilon > 0$ astfel încât $f(n) = O(n^{3-\varepsilon})$, obţinem aproximarea $T(n) = \Theta(n^3)$ **(cazul 1)**.

Fie funcţia $T(n) = 2 \cdot T(n/3) + n \log n$

Avem a=2, b=3, f(n)=nlog(n), $n^{\log_b a} = n^{\log_3 2} = n^{0.63}$

Cum exista un $\varepsilon > 0$ astfel încât $f(n) = \Omega(n^{0.63+\varepsilon})$ obţinem $T(n) = \Theta(n \log n)$ **(cazul 2)**.

Fie funcţia $T(n) = T(n/2) + 12$

Avem a=1, b=2, f(n)=12, $n^{\log_b a} = n^{\log_2 1} = 1$

Cum $f(n) = \Theta(1)$, obţinem aproximarea $T(n) = \Theta(\log n)$ **(cazul 3)**.

Fie funcţia $T(n) = 5 \cdot T(n/2) + 12 \cdot (n^2 + 2n)$

Avem a=5, b=2, f(n)= $12 \cdot (n^2 + 2n)$, $n^{\log_b a} = n^{\log_2 5} = n^{2.32}$

Cum exista un $\varepsilon > 0$ astfel încât $f(n) = O(n^{2.32-\varepsilon})$, obţinem aproximarea $T(n) = \Theta(n^{2.32})$ **(cazul 1)**.

Fie funcţia $T(n) = 2 \cdot T(n/2) + 2 \cdot n^2$

Avem a=2, b=2, f(n)=2n^2, $n^{\log_b a} = n^{\log_2 2} = n$

Cum exista un $\varepsilon > 0$ astfel încât $f(n) = \Omega(n^{1+\varepsilon})$, obţinem aproximarea $T(n) = \Theta(n^2)$ **(cazul 2)**.

Putem aplica teorema Master pentru analiza complexităţii algoritmilor de determinare a elementului de valoare maximă respectiv de căutare binară prezentați anterior, verificând astfel rezultatele la care am ajuns prin numărarea efectivă a operațiilor elementare.

Exemplu

Problema determinării elementului de valoare maximă

Timpul de execuţie este descris de recurența:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2 \cdot T(n/2) + \Theta(1) & n > 1 \end{cases}$$

în care atât D(n) cât şi I(n) se execută în Θ(1).

Avem a=2, b=2, $f(n) = \Theta(1)$, $n^{\log_b a} = n^{\log_2 2} = n$

Cum exista un $\varepsilon > 0$ astfel încât $f(n) = O(n^{1-\varepsilon})$ obţinem aproximarea $T(n) = \Theta(n)$ **(cazul 1)**.

Exemplu

Problema căutării binare

Timpul de execuţie este descris de recurența:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n/2) + \Theta(1) & n > 1 \end{cases}$$

în care atât D(n) cât şi I(n) se execută în Θ(1).

Avem a=1, b=2, $f(n) = \Theta(1)$, $n^{\log_b a} = n^{\log_2 1} = 1$

Cum $f(n) = \Theta(1)$ obţinem aproximarea $T(n) = \Theta(\log n)$ **(cazul 3)**.

2.5 Problema elementului sumă

Vom considera în continuare o problemă care poate fi rezolvată eficient prin reducerea la aplicarea repetată a unei probleme rezolvate folosind metoda Divide et Impera. Vom analiza atât algoritmul naiv cât şi o nouă variantă, mai puţin intuitivă, însă mult mai performantă din punct de vedere al complexităţii temporale.

Problemă

Elementul sumă

Fie o mulțime de numere întregi de dimensiune `n` şi fie un număr `x`. Să se determine dacă există două elemente din mulțime a căror sumă să fie `x`.

Fişierul de intrare va conţine pe prima linie numărul de elemente ale mulțimii iar pe a doua linie elementele mulțimii separate printr-un spaţiu. A treia linie conţine valoarea `x`. Fişierul de ieşire va avea o singură linie conţinând cele două elemente care adunate dau suma `x`, respectiv valoarea `-1` dacă nu există soluţie.

Intrare	Ieşire
6 7 1 5 3 2 8 11	3 8

O primă variantă de rezolvare este prezentată în continuare şi presupune testarea sumei tuturor perechilor de elemente din mulțime. Întrucât avem `n(n-1)/2` perechi de elemente de testat complexitatea algoritmului va fi pătratică, $O(n^2)$.

Pseudocod

```
    procedura Suma(int[] a, int n, int x)
1       pentru i ← 0, n-2 execută
2           pentru j ← i+1, n-1 execută
3               *) testăm perechea (i, j)
4               dacă a[i] + a[j] == x atunci
5                   scrie a[i], a[j]
6                   stop
7               ■
8           ■
9       ■
10      scrie -1 *) nu există soluție
11  sf.procedură
```

O variantă de rezolvare mai eficientă însă mai puţin intuitivă constă în sortarea tabloului urmată de căutarea binară a valorii `x-a[i]` pentru fiecare element `a[i]`. Dacă aceasta se află în tablou atunci avem soluţia (`a[i],x-a[i]`).

Pseudocod

```
   int Suma-2(int[] a, int n, int x)
1       *) sortează crescător şirul a
2       pentru i ← 0, n-1 execută
3           j ← CautăBinar(a, 0, n-1, x-a[i])
4           dacă j != -1 și j != i atunci
5               scrie a[i], a[j]
6               stop
7           ■
8       ■
9       scrie -1 *) nu există soluție
10  sf.procedură
```

Am arătat anterior pentru algoritmul de căutare binară o complexitate logaritmică, `O(log(n))`. Procedura `Suma-2(...)` aplică căutarea binară în mod repetat pentru fiecare element din a deci vom avea pentru liniile 2-8 o complexitate de `O(nlog(n))`. Alegând o metodă de sortare care ordonează crescător şirul a într-o complexitate liniar logaritmică[2], `O(nlog(n))`, obţinem pentru `Suma-2` complexitatea `O(nlog(n))`.

Implementarea în limbajul C# a celor două variante de rezolvare este prezentată în continuare, urmată de o analiză comparativă a timpului de execuție (Figura 2.6).

```
class ElementulSuma
{
    public struct Pair
    {
        public int Index1;
        public int Index2;
    };
    /// <summary>
    /// Cauta in vectorul a doua elemente care adunate
    /// dau valoarea x (implementare O(n^2)).
    /// </summary>
    public static Pair ElementulSuma_1(int[] a, int x)
    {
        for (int i = 0; i < a.Length - 1; i++)
            for (int j = i + 1; j < a.Length; j++)
                if (a[i] + a[j] == x)
                    return new Pair() { Index1 = i, Index2 = j };
        return new Pair() { Index1 = -1, Index2 = -1 };
    }
```

[2] Spre exemplu metoda sortării prin interclasare (care foloseşte tot principiul Divide et Impera).

```
    /// <summary>
    /// Cauta in vectorul a doua elemente care adunate
    /// dau valoarea x (implementare O(nlog(n)).
    /// </summary>
    /// <returns>
    /// Pozitiile indicilor (i, j) elementelor din a care
    /// adunate dau x sau (-1,-1) daca nu exista solutie
    </returns>
    public static Pair ElementulSuma_2(int[] a, int x)
    {
        Array.Sort(a);
        for (int i = 0; i < a.Length; i++)
        {
            int j = CautareBinara.CautaBinar_Iterativ(a, x - a[i]);
            if (j != -1 && j != i)
                return new Pair { Index1 = i, Index2 = j };
        }
        return new Pair() { Index1 = -1, Index2 = -1 };
    }
}
```

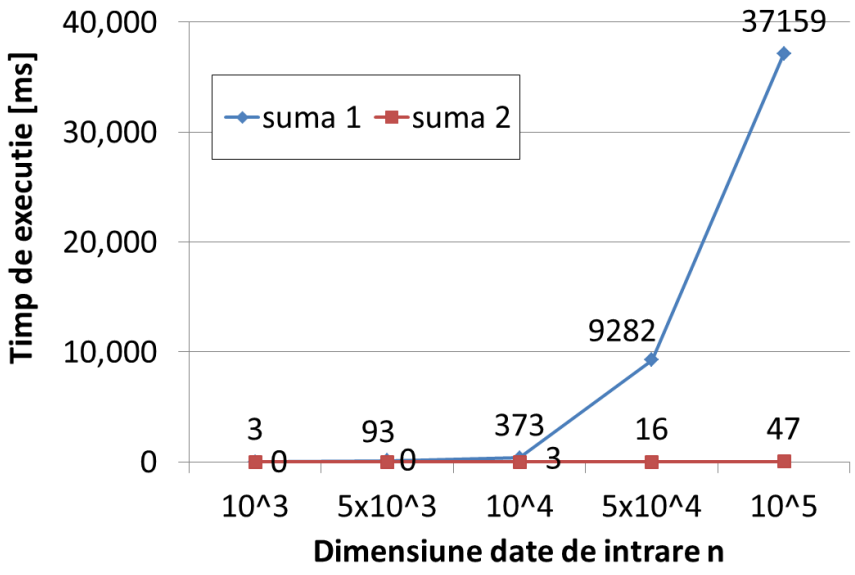

Figura 2.6. Timpul de execuție în milisecunde obținut pentru rularea celor două variante de rezolvare a problemei elementului sumă. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

2.6 Probleme propuse

Problema #1

Maxim, minim

Implementaţi problema căutării elementului maxim dintr-o mulțime de numere întregi.

Fișierul de intrare va avea pe prima linie numărul `n` de elemente ale mulțimii iar a doua linie va conține elementele mulțimii separate prin spațiu. Fișierul de ieșire va conține o singură linie cu elementul de valoarea maximă.

Intrare	Ieşire
4 5 25 -75 10	25

Modificaţi algoritmul astfel încât să obţineţi atât valoarea maximă cât şi cea minimă. Care este timpul de calcul necesar pentru algoritmul modificat? Care este complexitatea asociată?

Implementaţi varianta nerecursivă a algoritmului de determinare a elementului de valoare maximă.

Problema #2

Căutarea binară

Implementaţi algoritmul de căutare binară a unei valori într-o mulțime de numere întregi sortate crescător.

Fișierul de intrare va avea pe prima linie numărul `n` de elemente ale mulțimii iar a doua linie va conține elementele mulțimii separate prin spațiu. A treia linie va conține valoarea elementului căutat. Fișierul de ieșire va conține o singură linie cu poziția elementului în șir, respectiv valoarea `-1` dacă elementul nu se află în mulțime.

Intrare	Ieşire
8 1 4 5 8 12 15 18 25 18	6

Modificați algoritmul de căutare binară astfel încât să împărțiți problema în trei, respectiv în patru subprobleme de aceeași natură. Aplicați teorema Master pentru a determina complexitatea algoritmului modificat în cele două cazuri.

Implementaţi şi algoritmul de căutare liniară şi măsuraţi timpul de execuţie pentru valori mari ale lui `n` (de ex., 1,000,000, 10,000,000,

100,000,000). Elementele mulțimii vor fi generate automat folosind un generator de numere aleatoare.

Comparaţi timpul de execuţie pentru cei doi algoritmi de căutare şi verificaţi grafic dependenţa liniară respectiv logaritmică.

Cum puteți aplica algoritmul căutării binare pentru o mulțime de numere a cărei dimensiune este prea mare pentru a putea fi încărcată în memoria sistemului?

Rezolvaţi următoarele probleme aplicând tehnica Divide et Impera.
Determinaţi pentru fiecare problemă timpul de execuţie folosind teorema Master.

Problema #3

Cel mai mare divizor comun

Fie o mulțime de numere întregi de dimensiune `n`. Să se calculeze cel mai mare divizor comun al acestora.

Fişierul de intrare va conţine pe prima linie numărul de elemente iar pe a doua linie elementele mulțimii separate printr-un spaţiu. Fişierul de ieşire va avea pe prima linie cel mai mare divizor comun.

Intrare	Ieşire
3 125 75 1500	25

Problema #4

Ghiceşte numărul

Scrieţi un program care va ghici un număr ales de dumneavoastră între `1` şi `n`. Valoarea `n` va fi citită în prealabil de la tastatură. De fiecare dată când calculatorul va propune o soluţie, îi veţi răspunde cu una din următoarele variante:

- numărul este prea mare (introduceți textul “PREA MARE”).
- numărul este prea mic (textul “PREA MIC”).
- numărul a fost ghicit (textul “BRAVO”).

Valoarea `n` cât şi variantele de răspuns vor fi citite într-o manieră interactivă de la tastatură.

Realizați implementarea astfel încât calculatorul să ghicească numărul folosind un număr logaritmic de încercări în funcție de valoarea lui `n`.

Problema #5

Suma pătratelor

Fie o mulțimie de numere întregi de dimensiune `n`. Să se calculeze suma pătratelor elementelor şirului folosind abordarea specifică metodei Divide et Impera.

Fişierul de intrare va conţine pe prima linie numărul de elemente din şir iar pe a doua linie elementele şirului separate printr-un spaţiu. Fişierul de ieşire va avea pe prima linie suma pătratelor.

Intrare	Ieşire
4 7 1 5 3	84