TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

Condiții necesare pentru aplicare:

C1 (divide): Problema dată se poate împărți în două sau mai multe subprobleme de același tip și aproximativ aceeași dimensiune a datelor de intrare.

C2 (impera): Rezultatul unei probleme se obține combinând rezultatele subproblemelor în care a fost descompusă.

Exemplu:

$suma(t, st, dr) = t[st] + t[st+1] + \cdots + t[dr]$

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

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

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă
    if dr == st:
        return t[st]

    # etapa Divide
    mij = (st + dr) // 2
    sol_st = suma(t, st, mij)
    sol_dr = suma(t, mij + 1, dr)

    # etapa Impera
    return sol_st + sol_dr
```

Forma generală a unui algoritm de tip Divide et Impera

```
# functie care furnizeaza solutia unei probleme combinand solutiile
# subproblemelor in care ea a fost descompusa
def combinare(sol_st, sol_dr):
    pass

def divimp(t, st, dr):
   # daca subproblema curentă este direct rezolvabilă
   if dr - st <= k:          # k este, de obicei, 0 sau 1
      return solutie_problema_directa

      # etapa Divide
      mij = (st + dr) // 2
      sol_st = divimp(t, st, mij)
      sol_dr = divimp(t, mij+1, dr)

      # etapa Impera
      return combinare(sol_st, sol_dr)
```

$T(n)$ = complexitatea rezolvării unei probleme având dimensiunea datelor de intrare egală cu n

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă
    if dr == st:
        return t[st]

    # etapa Divide
    mij = (st + dr) // 2
    sol_st = suma(t, st, mij)
    sol_dr = suma(t, mij + 1, dr)

    # etapa Impera
    return sol_st + sol_dr
```

Complexitate suma:

$$T(n) = \begin{cases} 1 + T\left(\left[\frac{n}{2}\right]\right) + T\left(\left[\frac{n}{2}\right]\right) + 1, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

$$= \begin{cases} 2T\left(\left[\frac{n}{2}\right]\right) + 2, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

Determinarea complexității:

~~a) ghicirea formulei + inducție matematică~~

b) metoda substituțiilor repetate

c) teorema master

~~d) teorema Akra-Bazzi~~

b) pentru suma

$T(n)$ = complexitatea rezolvării unei probleme având dimensiunea datelor de intrare egală cu n

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{dacă } n = 1 \\ 2T\left(\frac{n}{2}\right) + 2, & \text{dacă } n \geq 2 \end{cases}$$

Presupunem faptul că $n = 2^k \Longrightarrow$

$T(n) = T(2^k) = 2T(2^{k-1}) + 2 = 2[2T(2^{k-2}) + 2] + 2 = 2^2T(2^{k-2}) + 2^2 + 2 = 2^2[2T(2^{k-3}) + 2] + 2^2 + 2 = 2^3T(2^{k-3}) + 2^3 + 2^2 + 2 = \cdots = 2^k \underbrace{T(2^0)}_{1} + 2^k + 2^{k-1} + \cdots + 2^2 + 2 = 2^k + 2^k + 2^{k-1} + \cdots + 2^2 + 2 = 2^k + 2 \cdot (2^k - 1) = 2^k(1 + 2) - 2 = 3 * 2^k - 2 = 3n - 2 \Rightarrow T(n) \in \mathcal{O}(n)$

Teorema master

$T(n)$ = complexitatea rezolvării unei probleme având dimensiunea datelor de intrare egală cu n

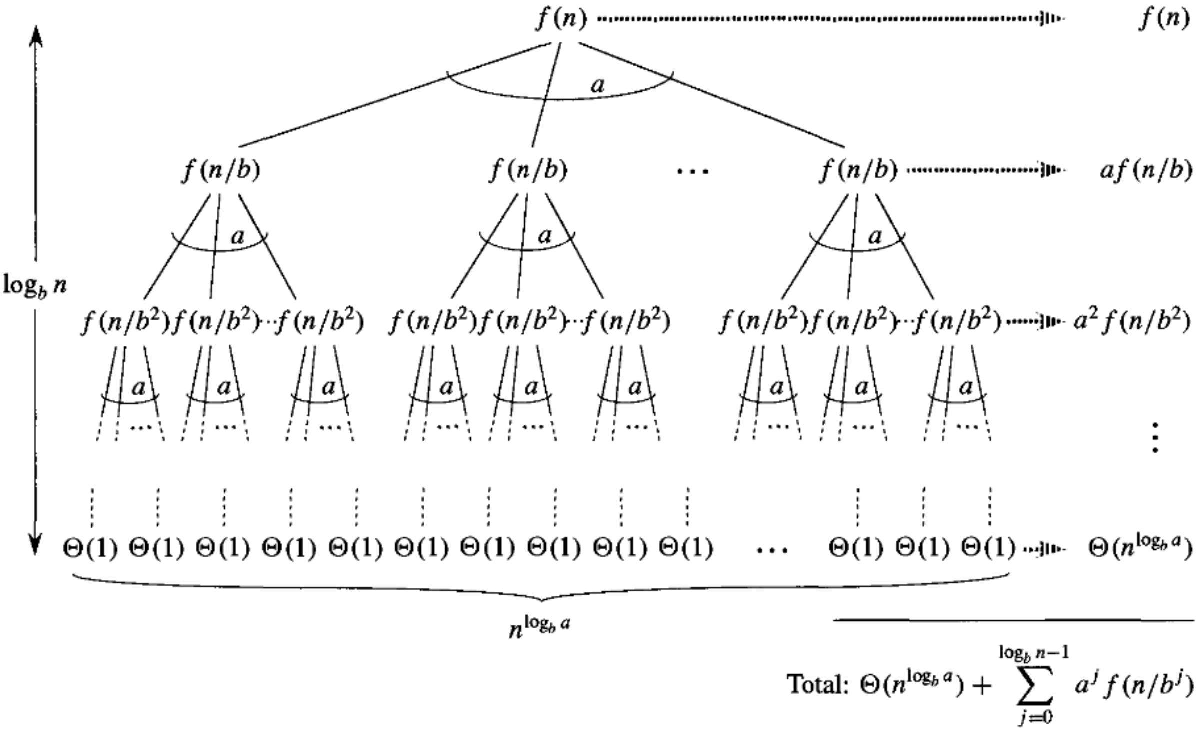
Considerăm că:

- problema se descompune/reduce în $a \geq 1$ subprobleme
- fiecare subproblemă are dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{b}$, unde $b \geq 2$
- împărțirea problemei curente în subprobleme și combinarea soluțiilor subproblemelor pentru a obține soluția sa se realizează folosind un algoritm cu complexitatea $f(n)$, unde $f(n)$ este o funcție asimptotic pozitivă (i.e., există $n_0 \in \mathbb{N}$ astfel încât pentru orice $n \geq n_0$ avem $f(n) \geq 0$)

Relația de recurență pentru caracterizarea complexității:

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

Arborele de recursie = arbore complet/perfect



- Înălțimea arborelui este $h = \log_b n$
- Numărul de frunze este $numar\_fii^{\,inaltime\_arbore} = a^{\log_b n} = n^{\log_b a}$

$$T(n) = \underbrace{\sum_{i=0}^{\log_b n-1} \left[a^i \cdot f\left(\frac{n}{b^i}\right)\right]}_{\substack{\text{timpul necesar pentru} \\ \text{divizarea problemei} \\ \text{și reconstituirea soluției}}} + \underbrace{n^{\log_b a} \cdot O(1)}_{\substack{\text{timpul necesar pentru} \\ \text{rezolvarea subproblemelor} \\ \text{directe}}}$$

Teorema master

Fie o relație de recurență de forma $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ 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 $\exists k < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq kf(n)$ pentru orice n suficient de mare, atunci $T(n) \in \mathcal{O}(f(n))$.

Exemple:

1) **Suma elementelor:**

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{dacă } n = 1 \\ 2T\left(\frac{n}{2}\right) + 2, & \text{dacă } n \geq 2 \end{cases}$$

$f(n) = 2 = 2 \cdot n^0 = n^0$

$\left.\begin{matrix} a = 2 \\ b = 2 \end{matrix}\right\} \Rightarrow \left.\begin{matrix} \log_b a = 1 \\ p = 0 \end{matrix}\right\} \Rightarrow p < \log_b a \Rightarrow$ caz a) teorema master $\Rightarrow$

$T(n) \in \mathcal{O}\left(n^{\log_b a}\right) = \mathcal{O}(n^1) = \mathcal{O}(n)$

2) **Căutarea binară:**

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{dacă } n = 0 \\ 1 \cdot T\left(\frac{n}{2}\right) + 1, & \text{dacă } n \geq 1 \end{cases}$$

$f(n) = 1 = n^0$

$\left.\begin{matrix} a = 1 \\ b = 2 \end{matrix}\right\} \Rightarrow \left.\begin{matrix} \log_b a = 0 \\ p = 0 \end{matrix}\right\} \Rightarrow p = \log_b a \Rightarrow$ caz b) teorema master $\Rightarrow$

$T(n) \in \mathcal{O}(n^p \log_2 n) = \mathcal{O}(n^0 \log_2 n) = \mathcal{O}(\log_2 n)$

3)

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{dacă } n = 0 \\ 2T\left(\frac{n}{2}\right) + n^2, & \text{dacă } n \geq 1 \end{cases}$$

$f(n) = n^2$

$\left.\begin{matrix} a = 2 \\ b = 2 \end{matrix}\right\} \Rightarrow \left.\begin{matrix} \log_b a = 1 \\ p = 2 \end{matrix}\right\} \Rightarrow p > \log_b a \Rightarrow \text{caz c) teorema master}$

$\exists k < 1 \text{ a.î. } a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n) \Rightarrow 2 \cdot \left(\frac{n}{2}\right)^2 \leq k \cdot n^2 \Rightarrow \frac{n^2}{2} \leq k \cdot n^2 \Rightarrow k \geq \frac{1}{2}$

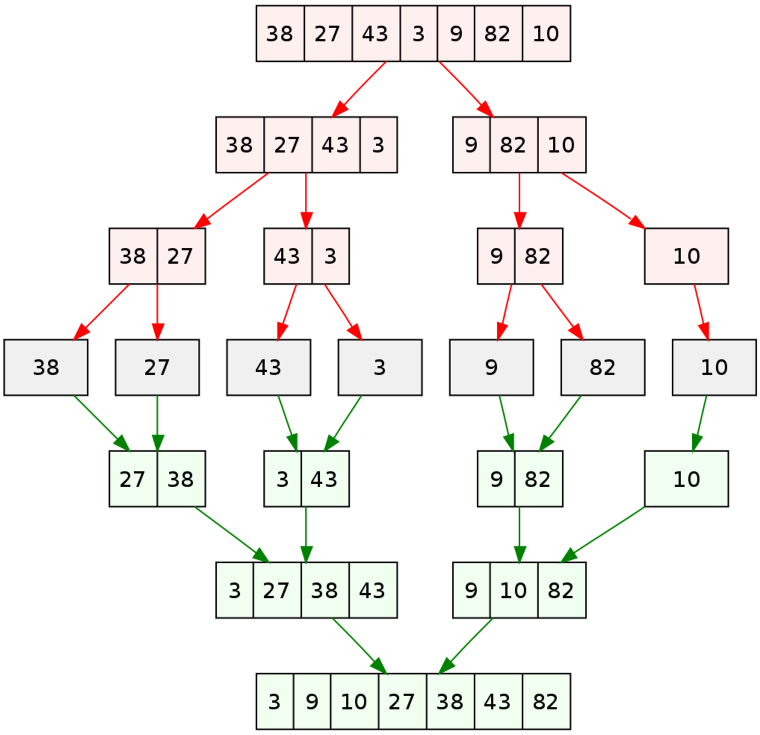
$T(n) \in \mathcal{O}\big(f(n)\big) = \mathcal{O}(n^2)$

Sortarea prin interclasare (Mergesort) – 1945 John von Neumann

Sortarea prin interclasare utilizează tehnica de programare Divide et Impera pentru a sorta crescător un tablou unidimensional de numere, astfel:

- se împarte secvența curentă $t[st], \ldots, t[dr]$, în mod repetat, în două secvențe $t[st], \ldots, t[mij]$ și $t[mij+1], \ldots, t[dr]$ până când se ajunge la secvențe implicit sortate, adică secvențe de lungime 1;
- în sens invers, se sortează secvența $t[st], \ldots, t[dr]$ interclasând cele două secvențe în care a fost descompusă, respectiv $t[st], \ldots, t[mij]$ și $t[mij+1], \ldots, t[dr]$, și care au fost deja sortate la un pas anterior.

O reprezentară grafică a modului în care rulează această metodă de sortare se poate observa în următoarea imagine (sursa: https://en.wikipedia.org/wiki/Merge_algorithm):



Sursa -> în varianta finală a cursului!!!

Complexitate:

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{dacă } n = 1 \\ 2T\left(\frac{n}{2}\right) + n, & \text{dacă } n \geq 2 \end{cases}$$

$$f(n) = n = n^1$$
$$\left.\begin{matrix} a = 2 \\ b = 2 \end{matrix}\right\} \Rightarrow \left.\begin{matrix} \log_b a = 1 \\ p = 1 \end{matrix}\right\} \Rightarrow p = \log_b a \Rightarrow \text{caz b) teorema master } \Rightarrow$$

$$T(n) \in \mathcal{O}(n^p \log_2 n) = \mathcal{O}(n^1 \log_2 n) = \mathcal{O}(n \log_2 n)$$

Selecția celui de-al k-lea minim (quickselect)

```
A =  [10, 7, 25, 4, 3, 4, 9, 12, 7]
sorted(A) = [3, 4, 4, 7, 7, 9, …]
k = 5 => minim = 7
```

Aleg un pivot aleatoriu => `pivot = 9`

```
#L = less
L = [7, 4, 3, 4, 7]

#E = equals
E = [9]

#G = greater
G = [10, 25, 12]
```

```
def quickselect(A, k, f_pivot=random.choice):
    pivot = f_pivot(A)

    L = [x for x in A if x < pivot]
    E = [x for x in A if x == pivot]
    G = [x for x in A if x > pivot]

    if k < len(L):
        return quickselect(L, k, f_pivot)
    elif k < len(L) + len(E):
        return E[0]
    else:
        return quickselect(G, k - len(L) - len(E), f_pivot)
```

Apel: `quickselect(A, k-1)`

Complexitate (pentru un pivot "bun"):

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{dacă } n = 1 \\ 1 \cdot T\left(\frac{n}{2}\right) + n, & \text{dacă } n \geq 2 \end{cases}$$

$f(n) = n$

$$\left.\begin{array}{l} a = 1 \\ b = 2 \end{array}\right\} \Rightarrow \left.\begin{array}{r} \log_b a = 0 \\ p = 1 \end{array}\right\} \Rightarrow p > \log_b a \Rightarrow \text{caz c) teorema master}$$

$$\exists k < 1 \text{ a.î. } a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n) \Rightarrow 1 \cdot \frac{n}{2} \leq k \cdot n \Rightarrow n \leq 2 \cdot k \cdot n \Rightarrow k \geq \frac{1}{2}$$

$T(n) \in \mathcal{O}(f(n)) = \mathcal{O}(n)$

Algoritmul BFPRT (Blum-Floyd-Pratt-Rivest-Tarjan) – 1973

Mediana unei liste = elementul aflat în mijlocul listei sortată crescător

Exemplu: A = [2, 7, 19, 23, 32, 45] => mediana = 23

Pivotul din BFPRT = mediana medianelor

`A = [L, pivot, G], unde |A| = n`

Secvențele `L` și `G` vor avea un număr de elemente cuprins între $\frac{3n}{10}$ și $\frac{7n}{10}$.

Exemplu:

A = [3, 14, 10, 2, 15, 10, 5, 51, 15, 20, 40, 4, 18, 13, 8,
40, 21, 61, 19, 50, 12, 35, 8, 7, 22, ~~100, 17~~]

	2	7	4	5	19
	3	8	8	10	21
Mediane	10	12	pivot 13	15	40
	14	22	18	20	50
	15	35	40	51	61

Numărul elementelor < pivot: $\frac{1}{2} \cdot \frac{n}{5} \cdot 3 = \frac{3n}{10}$

Numărul elementelor > pivot: $\frac{1}{2} \cdot \frac{n}{5} \cdot 3 = \frac{3n}{10}$

```python
def BFPRT(A):
    if len(A) <= 5:
        return sorted(A)[len(A) // 2]

    grupuri = [sorted(A[i:i + 5]) for i in range(0, len(A), 5)]

    mediane = [grup[len(grup) // 2] for grup in grupuri]

    return BFPRT(mediane)
```

Complexitate:

$$T(n) = \underbrace{T\left(\frac{n}{5}\right)} + \underbrace{T\left(\frac{7n}{10}\right)} + O(n)$$

$$T(n) \leq 10 \cdot c \cdot n \;\Rightarrow T(n) \in O(n)$$