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 unei liste t formată din n numere întregi, astfel:

1. împărţim lista t, în mod repetat, în două jumătăţi până când obţinem liste cu un singur element (caz în care suma se poate calcula direct, fiind chiar elementul respectiv);
2. în sens invers, calculăm suma elementelor dintr-o listă adunând sumele elementelor celor două sub-liste în care ea a fost descompusă.

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

Pentru a putea manipula uşor cele două sub-liste care se obţin în momentul împărţirii listei t în două jumătăţi, vom considera faptul că lista curentă este secvenţa cuprinsă între doi indici st şi dr, unde $st \leq dr$. Astfel, indicele mij al mijlocului listei curente este aproximativ egal cu $[(st + dr)/2]$, iar cele două sub-liste în care va fi descompus lista curentă sunt secvenţele cuprinse între indicii st şi mij, respectiv $mij + 1$ şi dr.

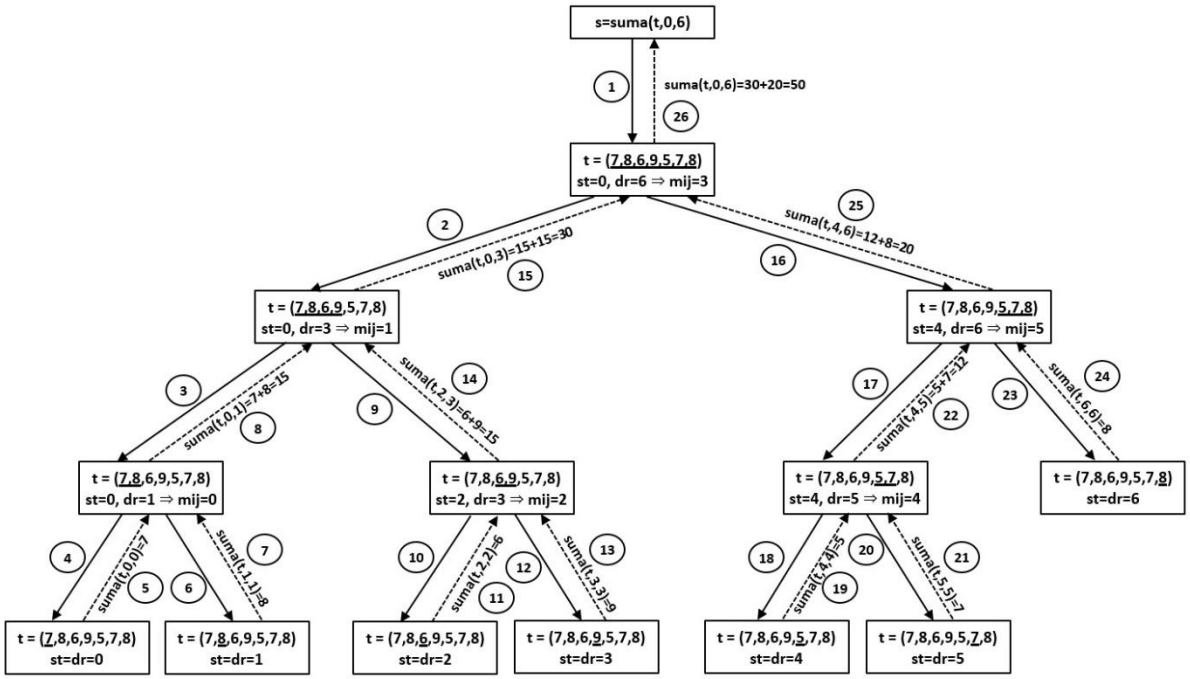
Considerând $suma(t, st, dr)$ o funcţie care calculează suma secvenţei $t[st], t[st + 1], \dots, t[dr]$, putem să o definim în manieră Divide et Impera astfel:

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

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

Pentru o listă t cu n elemente, suma s a tuturor elementelor sale se va obţine în urma apelului $s = suma(t, 0, n - 1)$.

De exemplu, considerând lista $t = (7, 8, 6, 9, 5, 7, 8)$ cu $n = 7$ 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 lista *t* reprezintă secvenţa curentă (i.e., 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 unei liste *t*:

```
# 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)
```

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

- există algoritmi Divide et Impera în care nu sunt utilizate liste (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 liste;
- de obicei, o subproblemă este direct rezolvabilă dacă secvenţa curentă din lista `t` este vidă (i.e., `st>dr`) sau are un singur element (i.e., `st==dr`);
- variabila `mij` va conţine indicele mijlocului secvenţei `t[st], t[st+1], …, t[dr]`;
- variabilele `sol_st` şi `sol_dr` vor conţine soluţiile celor două subprobleme în care se descompune problema curentă, iar `combinare(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, atunci vor lipsi variabilele `sol_st` şi `sol_dr`, precum şi cele două instrucţiuni `return`.

Aplicând algoritmul general pentru a calcula suma elementelor dintr-o listă de numere întregi, vom obţine următoarea implementare în limbajul Python:

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă,
    # respectiv lista curenta are un singur element
    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
```

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

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 obține foarte ușor forma generală a relației de recurență căutate:

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

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

Reluăm algoritmul Divide et Impera prezentat mai sus pentru calculul sumei elementelor unei liste, cu scopul de a-i determina complexitatea:

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă,
    # respectiv lista curenta are un singur element
    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
```

Analizând fiecare etapa a algoritmului, obținem următoarea relație de recurență pentru $T(n)$:

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

În concluzie, 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 împărțirea problemei curente în subprobleme și combinarea soluțiilor subproblemelor pentru a obține soluția sa se realizează folosind un algoritm cu complexitatea $\mathcal{O}(1)$, deci relația de recurență este următoarea:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \tag{4}$$

Al doilea pas în determinarea complexității unei algoritm Divide et Impera îl constituie rezolvarea relației de recurență 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 relației de recurență* și *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

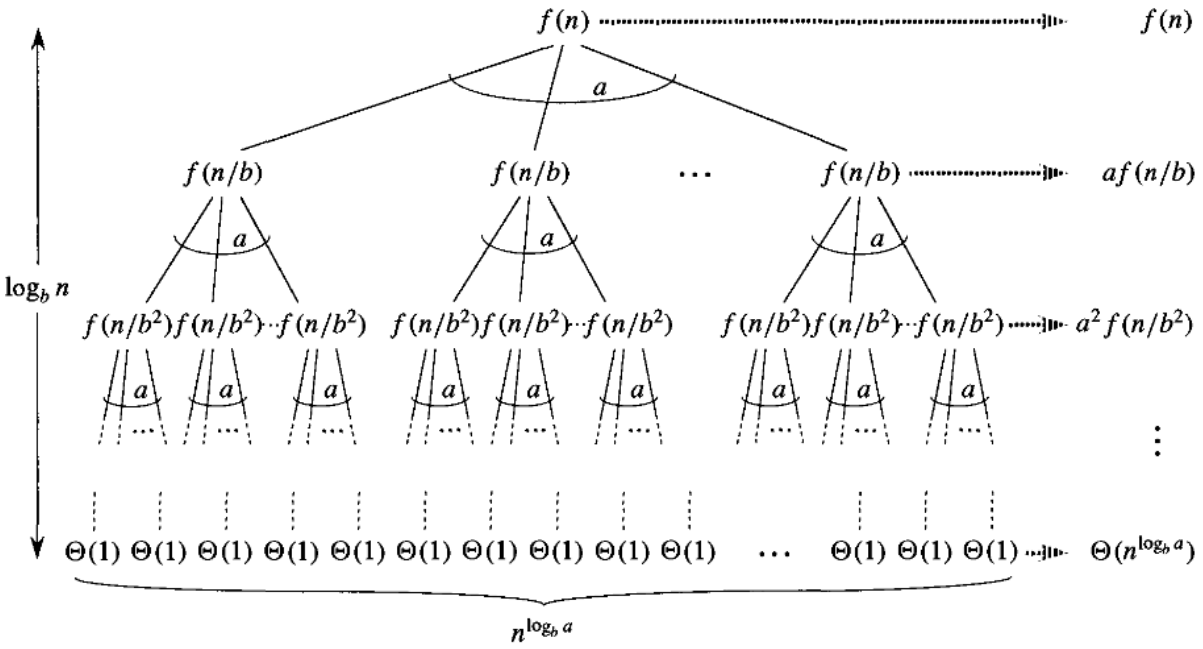
când vom ajunge la $T(1)$ sau $T(0)$, care sunt ambele egale cu 1 fiind complexităţile unor probleme direct rezolvabile.

De exemplu, pentru a rezolva relaţia de recurenţă (4), vom presupune că $n = 2^k$ şi 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 obţinut faptul că $T(n) = 2n - 1$, deci complexitatea algoritmului Divide et Impera pentru calculul sumei elementelor unei liste formate din n numere întregi este $\mathcal{O}(2n - 1) \approx \mathcal{O}(n)$.

A doua metodă constă în utilizarea *teoremei master* pentru a afla direct soluţia analitică a unei relaţii de recurenţă de tipul (3): $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Arborele de recursie asociat unei relaţii de recurenţă de acest tip este următorul (sursa imaginii: https://en.wikipedia.org/wiki/Introduction_to_Algorithms):



Se observă faptul că arborele este unul perfect (i.e., orice nod interior are exact a fii şi frunzele se află toate pe acelaşi nivel) cu înălţimea $h = \log_b n$, deci pe fiecare nivel, mai puţin pe ultimul, se găsesc $a^i f\left(\frac{n}{b^i}\right)$ noduri interne, iar pe ultimul nivel se găsesc $a^h = a^{\log_b n} = n^{\log_b a}$ frunze. Evident, complexitatea totală $T(n)$ se obţine însumând complexităţile asociate tuturor nodurilor:

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

Pentru anumite forme particulare ale funcției f, se poate calcula suma din expresia lui $T(n)$ și, implicit, se poate determina forma sa analitică.

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). Totuși, 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).

Observație importantă: În general, algoritmii de tip Divide et Impera au complexități mici, de tipul $\mathcal{O}(\log_2 n), \mathcal{O}(n)$ sau $\mathcal{O}(n \log_2 n)$, care se obțin datorită faptului că o problemă este împărțită în două subprobleme de același tip cu dimensiunea datelor de intrare înjumătățită față de problema inițială și, mai mult, subproblemele nu se suprapun! Dacă aceste condiții nu sunt îndeplinite simultan, atunci complexitatea algoritmul poate să devină foarte mare, de ordin exponențial! De exemplu, o implementare recursivă, de tip Divide et Impera, care să calculeze termenul de rang n al șirului lui Fibonacci ($F_n = F_{n-1} + F_{n-2}$ și $F_0 = 0, F_1 = 1$) nu respectă condițiile precizate anterior (dimensiunile subproblemelor nu sunt aproximativ jumătate din dimensiunea unei probleme și subproblemele se suprapun, respectiv mulți termeni vor fi calculați de mai multe ori), ceea ce va conduce la o complexitate exponențială! Astfel, relația de recurență 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/.

În continuare, vom prezenta câteva probleme clasice care se pot rezolva utilizând tehnica de programare Divide et Impera.

4. Problema căutării binare

Fie t o listă formată din n numere întregi sortate crescător şi x un număr întreg. Să se verifice dacă valoarea x apare în lista t.

Evident, problema ar putea fi rezolvată printr-o simplă parcurgere a listei t (*căutare liniară*), obţinând un algoritm având complexitatea $\mathcal{O}(n)$, dar nu am utiliza deloc faptul că elementele listei sunt în ordine crescătoare. Pentru a efectua o căutare binară într-o secvenţă $t[st], t[st+1], \dots, t[dr]$ a listei t în care $st \leq dr$ vom folosi această ipoteză, comparând valoarea căutată x cu valoarea $t[mij]$ aflată în mijlocul secvenţei. Astfel, vom obţine următoarele 3 cazuri:

a) $x < t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[st], \dots, t[mij-1]$;
b) $x > t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[mij+1], \dots, t[dr]$;
c) $x = t[mij] \Longrightarrow$ am găsit valoarea x, deci operaţia de căutare se încheie cu succes.

Dacă la un moment dat $st > dr$, înseamnă că nu mai există nicio secvenţă $t[st],, \dots, t[dr]$ în care să aibă sens să căutăm valoarea x, deci operaţia de căutare eşuează.

O implementare a căutării binare în limbajul Python, sub forma unei funcţii care furnizează o poziţie pe care apare valoarea x în lista t sau valoarea -1 dacă x nu apare deloc în listă, este următoarea:

```
def cautare_binara(t, x, st, dr):
    if st > dr:
        return -1

    mij = (st+dr) // 2
    if x == t[mij]:
        return mij
    elif x < t[mij]:
        return cautare_binara(t, x, st, mij-1)
    else:
        return cautare_binara(t, x, mij+1, dr)
```

Se observă faptul că acest algoritm de tip Divide et Impera constă doar din etapa Divide, nemaifiind combinate soluţiilor subproblemelor (etapa Impera). Practic, la fiecare pas, problema curentă se restrânge la una dintre cele două subprobleme, ci nu se rezolvă ambele subprobleme! Astfel, ţinând cont de faptul că etapa Divide are complexitatea

$\mathcal{O}(1)$, relaţia de recurenţă asociată complexităţii algoritmului de căutare binară este următoarea:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Folosind atât iterarea directă a relaţiei de recurenţă, cât şi teorema master, demonstraţi faptul că $T(n) \in \mathcal{O}(\log_2 n)$!

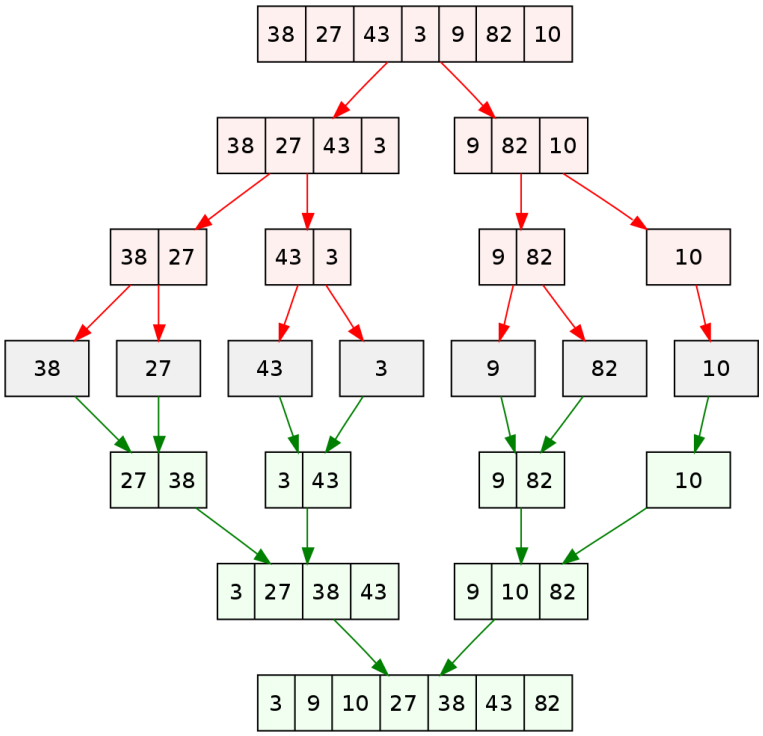
Observaţie importantă: Complexitatea $\mathcal{O}(\log_2 n)$ reprezintă strict complexitatea algoritmului de căutare binară, deci complexitatea unui program, chiar foarte simplu, în care se va utiliza acest algoritm va fi mai mare! De exemplu, un simplu program de test pentru funcţia de mai sus necesită citirea celor n elemente ale listei t şi afişarea valorii furnizate de funcţie, deci complexitatea sa va fi $\mathcal{O}(n) + \mathcal{O}(\log_2 n) + \mathcal{O}(1) \approx \mathcal{O}(n)$!

5. Sortarea prin interclasare (Mergesort)

Sortarea prin interclasare utilizează tehnica de programare Divide et Impera pentru a sorta crescător o listă t, astfel:

- se împarte secvenţa curentă $t[st], \dots, t[dr]$, în mod repetat, în două secvenţe $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$ până când se ajunge la secvenţe implicit sortate, adică secvenţe de lungime 1;
- în sens invers, se sortează secvenţa $t[st], \dots, t[dr]$ interclasând cele două secvenţe în care a fost descompusă, respectiv $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$, şi 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):



Începem prezentarea detaliată a acestei metodei de sortare reamintind faptul că interclasarea este un algoritm care permite obținerea unei liste sortate crescător din două liste care sunt, de asemenea, sortate crescător. Considerând dimensiunile listelor care vor fi interclasate ca fiind egale cu m și n, complexitatea algoritmului de interclasare este $\mathcal{O}(m+n)$.

În cazul sortării prin interclasare, se vor interclasa secvențele $t[st], \dots, t[mij]$ și $t[mij+1], \dots, t[dr]$ într-o listă aux de lungime $dr-st+1$, iar la sfârșit elementele acesteia se vor copia în secvența $t[st], \dots, t[dr]$:

```
def interclasare(t, st, mij, dr):
    i = st
    j = mij+1
    aux = []
    while i <= mij and j <= dr:
        if t[i] <= t[j]:
            aux.append(t[i])
            i += 1
        else:
            aux.append(t[j])
            j += 1

    aux.extend(t[i:mij+1])
    aux.extend(t[j:dr+1])

    t[st:dr+1] = aux[:]
```

Se observă ușor faptul că funcția `interclasare` are o complexitate egală cu $\mathcal{O}(dr-st+1) \leq \mathcal{O}(n)$.

Sortarea listei t se va efectua, folosind tehnica Divide et Impera, în cadrul următoarei funcții:

```
def mergesort(t, st, dr):
    if st < dr:
        mij = (dr+st) // 2
        mergesort(t, st, mij)
        mergesort(t, mij+1, dr)
        interclasare(t, st, mij, dr)
```

Observați faptul că, în acest caz, funcția `interclasare` are rolul funcției `combinare` din algoritmul generic Divide et Impera!

Complexitatea funcției `mergesort` și, implicit, complexitatea sortării prin interclasare, se obține rezolvând următoarea relație de recurență:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

În exemplul 2 de la teorema master am demonstrat faptul că $T(n) \in \mathcal{O}(n \log_2 n)$, deci sortarea prin interclasare are complexitatea $\mathcal{O}(n \log_2 n)$.

6. Selecția celui de-al k-lea minim (Quickselect)

Considerăm o listă L de lungime n și dorim să determinăm al k-lea minim al său (evident, $1 \leq k \leq n$), respectiv valoarea care s-ar afla pe poziția $k-1$ după sortarea crescătoare a sa. De exemplu, pentru $A = [10, 7, 25, 4, 3, 4, 9, 12, 7]$ și $k = 5$ vom obține valoarea 7, deoarece lista A sortată crescător este $[3, 4, 4, 7, 7, 9, 10, 12, 25]$. Evident, problema poate fi rezolvată sortând lista și apoi accesând elementul aflat pe poziția $k-1$, complexitatea acestei soluții fiind $\mathcal{O}(n \log_2 n)$.

În continuare, vom prezenta un algoritm de tip Divide et Impera pentru rezolvarea acestei probleme:

- alegem aleatoriu un element din listă ca pivot;
- partiționăm lista A în 3 liste în raport de pivotul respectiv:
  - o listă L formată din elementele strict mai mici decât pivotul;
  - o listă E formată din elementele egale cu pivotul;
  - o listă G formată din elementele strict mai mari decât pivotul;
- comparăm valoarea k cu lungimile celor 3 liste de partiție și fie furnizăm valoarea căutată (dacă a k-a valoare se găsește în lista E), fie reluăm procedeul pentru una dintre listele L sau G.

De exemplu, considerând $A = [10, 7, 25, 4, 3, 4, 9, 12, 7]$, $k = 5$ și pivotul aleatoriu 9, vom obține următoarele mulțimi de partiție: $L = [7, 4, 3, 4, 7]$, $E = [9]$ și $G = [10, 25, 12]$. Deoarece $k = 5 \leq \text{len}(L)$, vom relua procedeul pentru lista L și aceeași valoare $k = 5$. Dacă am fi considerat $k = 8$, atunci am fi reluat procedeul pentru lista G și $k = 8 - len(L) - len(E) = 8 - 5 - 1 = 2$!

Implementarea în limbajul Python a acestui algoritm de tip Divide et Impera este următoarea:

```
import random

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)
```

Deoarece pivotul poate fi selectat și în alte moduri (ci nu numai aleatoriu), am folosit mecanismul de call-back în cadrul funcției `quickselect` astfel încât ea să poată utiliza o anumită funcție pentru selectarea pivotului, primită prin intermediul parametrului `f_pivot`. Implicit, acest parametru este inițializat cu funcția `choice` din pachetul `random`, care furnizează în mod aleatoriu o valoare dintr-o colecție dată ca parametru. Atenție, funcția trebuie apelată prin `quickselect(A, k-1)`!

Deoarece pivotul este ales în mod aleatoriu, estimarea complexității algoritmului `quickselect` prezentat anterior este destul de complicată. Totuși, se poate observa ușor faptul că un pivot "bun" ar trebui să genereze două liste de partiție L și G cu dimensiuni aproximativ egale cu jumătate din numărul de elemente ale listei inițiale A. În acest caz, complexitatea algoritmului `quickselect` ar fi dată de următoarea relație de recurență:

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\substack{\text{se alege una dintre} \\ \text{listele } L \text{ sau } G}} + \underbrace{n}_{\substack{\text{crearea listelor} \\ L,E \text{ și } G}}$$

Pentru a rezolva această relație de recurență folosind teorema master, observăm faptul că $a = 1$ și $b = 2$, iar $f(n) = n \in \mathcal{O}(n^1) \Rightarrow p = 1 > \log_b a = \log_2 1 = 0$. Deoarece suntem în cazul c) al teoremei master, 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 $\frac{n}{2} \leq cn \Leftrightarrow \exists c < 1$ astfel încât $n \leq 2cn \Leftrightarrow \exists c < 1$ astfel încât $1 \leq 2c$, ceea ce este adevărat pentru orice valoare $c \in \left[\frac{1}{2}, 1\right)$. Astfel, obținem că $T(n) \in \mathcal{O}\big(f(n)\big) = \mathcal{O}(n)$. Pe de altă parte, selectarea repetată ca pivot a valorii minime/maxime din listă va conduce la complexitatea $\mathcal{O}(n^2)$!

Practic, un pivot "bun" ar trebui să fie *mediana listei* respective, adică valoarea aflată la mijlocul listei sortate crescător, sau, cel puțin, o valoare apropiată de aceasta. În continuare, vom prezenta algoritmul *mediana medianelor*, tot de tip Divide et Impera, care permite determinarea unui pivot "bun" pentru o listă. Algoritmul a fost creat în anul 1973 de către informaticienii M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest și R. Tarjan, din acest motiv fiind cunoscut și sub numele de *algoritmul BFPRT*. Pașii algoritmului, pentru o listă A formată din n elemente, sunt următorii:

- se împarte lista A în $\left\lceil\frac{n}{5}\right\rceil$ liste de lungime 5 (dacă ultima listă nu are exact 5 elemente, atunci ea poate fi eliminată);
- pentru fiecare dintre cele $\left\lceil\frac{n}{5}\right\rceil$ liste de lungime 5 se determină direct mediana sa (e.g., se sortează lista și se returnează elementul aflat în mijlocul său);
- se reia procedeul pentru lista formată din medianele celor $\left\lceil\frac{n}{5}\right\rceil$ liste de lungime 5 până când se obține mediana medianelor (i.e., lista medianelor are cel mult 5 elemente).

Exemplu: Considerăm lista $A = [3, 14, 10, 2, 15, 10, 5, 51, 15, 20, 40, 4, 18, 13, 8, 40, 21, 61, 19, 50, 12, 35, 8, 7, 22, \sout{100, 17}]$ cu $n = 27$ elemente. După sortarea fiecărei liste de lungime 5 (am eliminat lista [100,17], deoarece este formată doar din două elemente) și determinarea medianelor lor, reluăm procedeul pentru lista medianelor, respectiv $[10, 15, 13, 40, 12]$, și obținem pivotul 13:

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

Implementarea acestui algoritm în limbajul Python este foarte simplă:

```
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)
```

Rearanjând listele de lungime 5 în ordinea crescătoare a medianelor lor, obţinem:

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

Se observă faptul că elementele marcate cu galben sunt mai mici decât pivotul 13, iar elementele marcate cu verde sunt mai mari decât pivotul. De asemenea, se observă faptul că aproximativ jumătate dintre medianele grupurilor sunt mai mici decât pivotul şi aproximativ jumătate sunt mai mari decât el. Astfel, rezultă faptul că fiecare dintre cele două grupuri va avea cel puţin $\frac{3n}{10}$ elemente, deoarece în fiecare grup intră aproximativ jumătate din numărul de $\left\lceil \frac{n}{5} \right\rceil$ liste de lungime 5, deci $\left\lceil \frac{n}{10} \right\rceil$ liste, iar în fiecare listă sunt 3 elemente. În acelaşi timp, fiecare dintre cele două grupuri nu poate avea mai mult de $n - \frac{3n}{10} = \frac{7n}{10}$ elemente, deci oricare dintre cele două grupuri va avea un număr de elemente cuprins între $\frac{3n}{10}$ şi $\frac{7n}{10}$.

Practic, elementele marcate cu galben sunt cele din lista L definită în algoritmul `quickselect`, iar cele marcate cu verde sunt cele din lista G, deci utilizând algoritmul `BFPRT` pentru selectarea pivotului, prin apelul `quickselect(A, k-1, BFPRT)`, vom obţine următoarea complexitate a sa:

$$T(n) \leq \underbrace{T\left(\frac{n}{5}\right)}_{\substack{\text{determinarea pivotului} \\ \text{folosind mediana medianelor}}} + \underbrace{T\left(\frac{7n}{10}\right)}_{\substack{\text{selectarea uneia dintre} \\ \text{listele } L \text{ sau } G}} + \underbrace{n}_{\substack{\text{crearea listelor} \\ L,E \text{ şi } G}}$$

Pentru rezolvarea acestei relaţii de recurenţă nu putem utiliza teorema master, deoarece dimensiunile subproblemelor nu sunt egale, dar se poate intui faptul că $T(n) \leq cn$ şi apoi demonstra acest lucru folosind inducţia matematică (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec01.pdf), deci $T(n) \in \mathcal{O}(n)$. Cu toate acestea, deoarece valoarea constantei c pentru care se obţine o complexitate liniară este mare, respectiv $c = 140$, în practică se preferă utilizarea variantei cu pivot ales aleatoriu deoarece are o complexitate medie mai bună!