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.

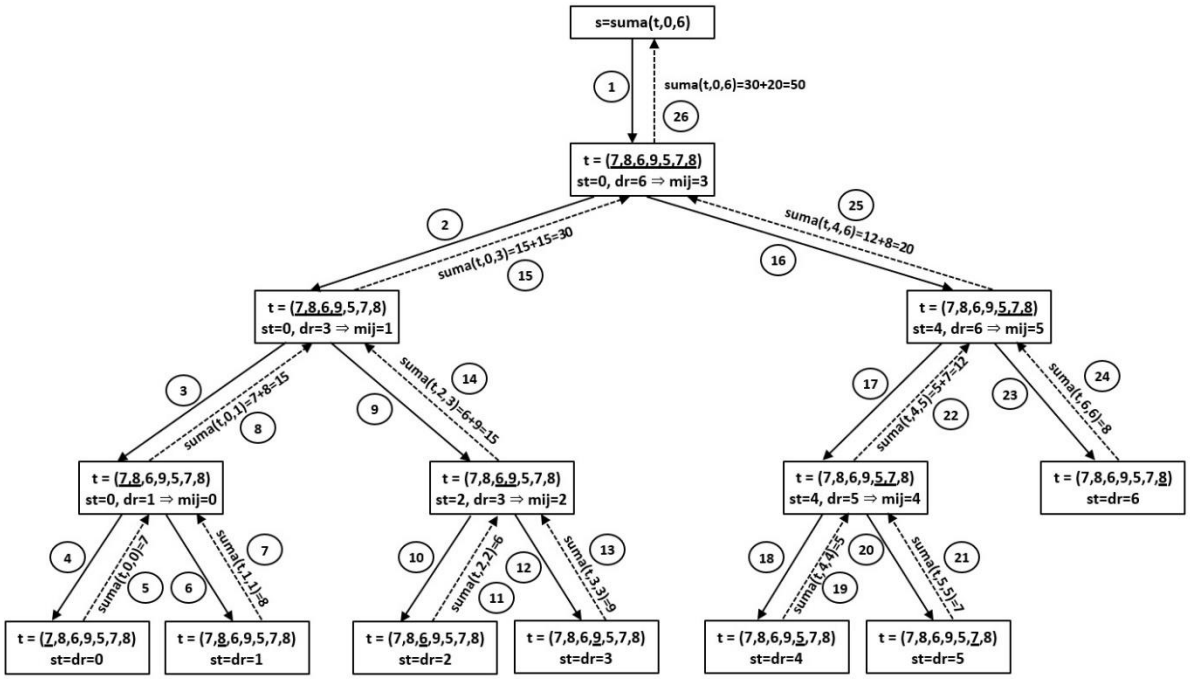
Considerând $suma(t, st, dr)$ o funcţie care calculează suma secvenţei $t[st], t[st + 1], \dots, t[dr]$, putem să o definim în manieră Divide et Impera astfel:

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

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:

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

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 una dintre cele mai utilizate metode, simplificate, respectiv *iterarea directă a relației de recurență*.

În cazul metodei bazate 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 relaţia de recurenţă (4), vom presupune că $n = 2^k$ şi apoi o vom itera, astfel:

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

În concluzie, am obţinut faptul că $T(n) = 3n - 2$, deci complexitatea algoritmului Divide et Impera pentru calculul sumei elementelor unei liste formate din n numere întregi este $\mathcal{O}(3n - 2) \approx \mathcal{O}(n)$.

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/.

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 $t[st], t[st+1], \ldots, t[dr]$ 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 $t[st], \ldots, t[mij-1]$;
b) $x > t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[mij+1], \ldots, t[dr]$;
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 $T(n) \in \mathcal{O}(\log_2 n)$!

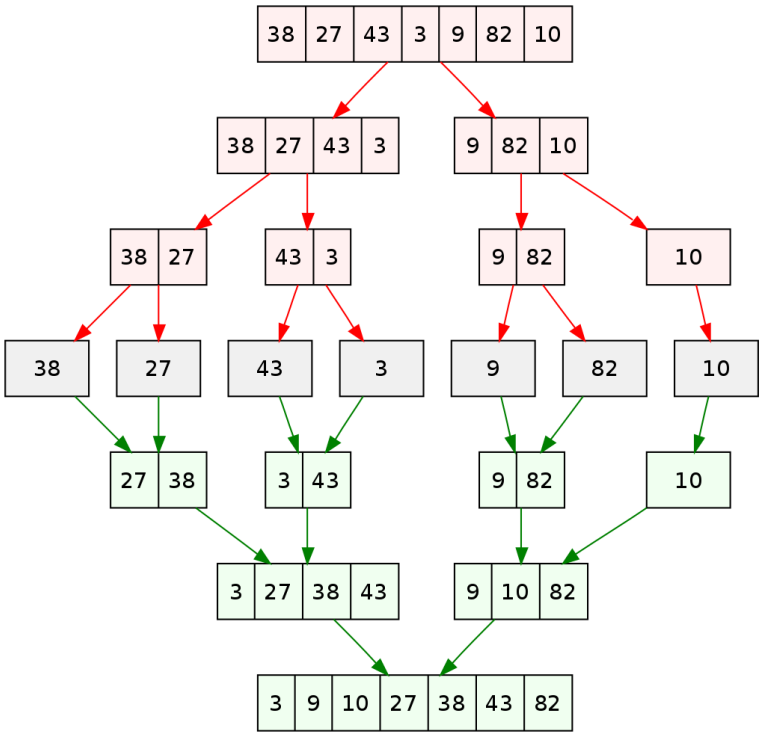
Observaţie importantă: Complexitatea $\mathcal{O}(\log_2 n)$ reprezintă strict complexitatea algoritmului de căutare binară, deci complexitatea unui program, chiar foarte simplu, în care se va utiliza acest algoritm va fi mai mare! De exemplu, un simplu program de test pentru funcţia de mai sus necesită 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 funcţii:

```
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)
```

Observaţi faptul că, în acest caz, funcţia `interclasare` are rolul funcţiei `combinare` din algoritmul generic Divide et Impera!

Complexitatea funcţiei `mergesort` şi, implicit, complexitatea sortării prin interclasare, se obţine rezolvând următoarea relaţie de recurenţă:

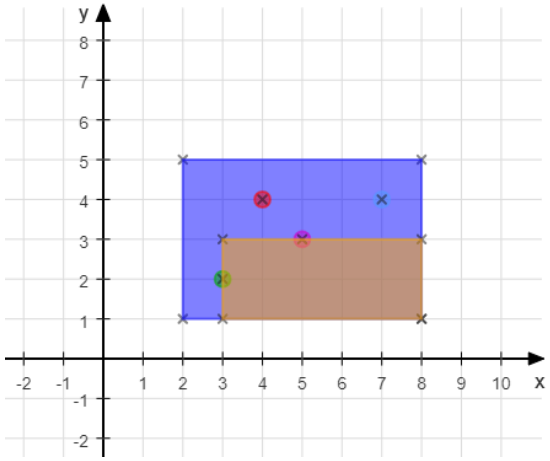
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Demonstraţi faptul că $T(n) \in \mathcal{O}(n \log_2 n)$, deci sortarea prin interclasare are complexitatea $\mathcal{O}(n \log_2 n)$!

6. Problema tăieturilor într-o placă

Într-o placă dreptunghiulară de lemn în care sunt date mai multe găuri putem efectua, folosind o maşină veche de debitat, doar tăieturi complete pe lungimea sau pe lăţimea plăcii şi paralele cu laturile sale. Să se determine o suprafaţă dreptunghiulară fără găuri cu arie maximă care se poate obţine efectuând doar tăieturi de tipul precizat.

Exemplu:



Placa de lemn este un dreptunghi având colţul stânga-jos de coordonate (2,1) şi colţul dreapta-sus de coordonate (8,5). În placă sunt date 4 găuri, la coordonatele (3,2), (4,4), (5,3) şi (7,4).

Dreptunghiul cu suprafaţa maximă (egală cu 10) şi care nu conţine nici un copac are coordonatele (3,1) pentru colţul stânga-jos şi (8,3) pentru colţul dreapta-sus şi se poate obţine, de exemplu, efectuând o tăietură orizontală completă de-a lungul dreptei $y = 3$ şi apoi a unei tăieturi verticale complete de-a lungul dreptei $x = 3$.

Algoritmul de tip Divide et Impera pentru rezolvarea acestei probleme este foarte simplu, respectiv verificăm dacă în dreptunghiul curent mai există cel puţin o gaură şi apoi tratăm cele două cazuri care pot să apară, astfel:

- dacă în dreptunghiul curent nu mai există nicio gaură, atunci comparăm aria sa cu aria maximă găsită până în acel moment şi, eventual, o actualizăm pe cea din urmă;

- dacă în dreptunghiul curent mai există cel puțin gaură, atunci efectuăm cele două tipuri de tăieturi și reluăm algoritmul pentru fiecare dintre cele 4 dreptunghiuri care se formează.

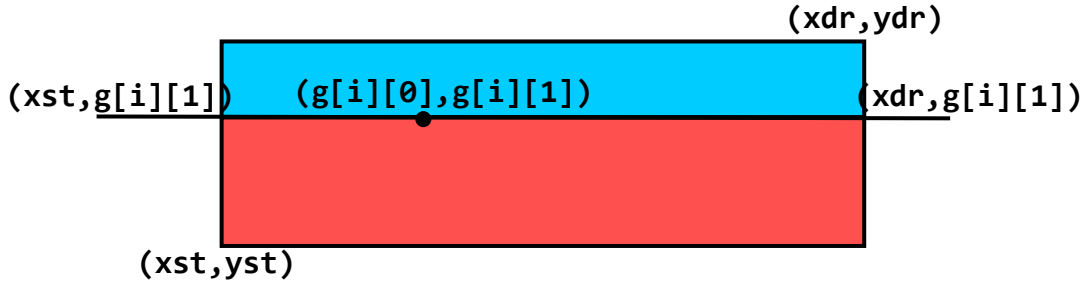
Vom implementa în limbajul Python algoritmul de mai sus, folosind o funcție cu antetul `dreptunghiArieMaxima(xst, yst, xdr, ydr)` ai cărei parametrii `xst` și `yst` sunt coordonatele colțului stânga-jos al dreptunghiului curent, iar `xdr` și `ydr` sunt cele ale colțului dreapta-sus. De asemenea, vom utiliza următoarele variabile globale:

- `coordonateGauri`: listă de tupluri care conțin abscisele și ordonatele găurilor;
- `arieMaxima`: aria maximă a unui dreptunghi fără găuri;
- `dMaxim`: dreptunghiul cu arie maximă.

Dacă în dreptunghiul curent se găsește o gaură cu coordonatele `g[i]`, atunci prin efectuarea celor două tipuri de tăieturi prin gaura respectivă se vor obține următoarele dreptunghiuri:

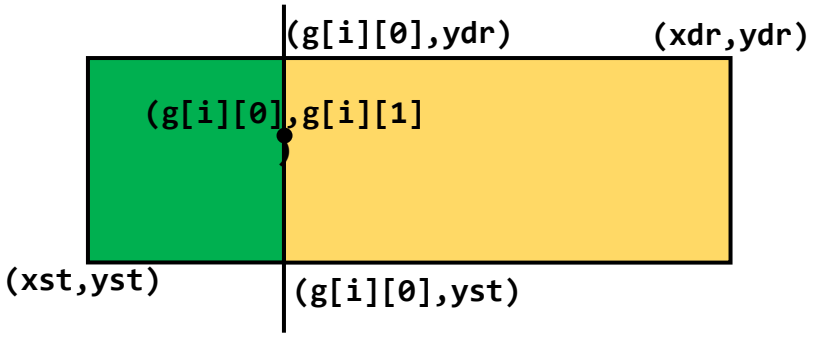
a) după o tăietură orizontală completă:

- *dreptunghiul superior* (albastru) având coordonatele colțului stânga-jos egale cu `xst` și `g[i][1]`, iar cele ale colțului dreapta-sus egale cu `xdr` și `ydr`;
- *dreptunghiul inferior* (roșu) având coordonatele colțului stânga-jos egale cu `xst` și `yst`, iar cele ale colțului dreapta-sus egale cu `xdr` și `g[i][1]`.



b) după o tăietură verticală completă:

- *dreptunghiul stâng* (verde) având coordonatele colțului stânga-jos egale cu `xst` și `yst`, iar cele ale colțului dreapta-sus egale cu `g[i][0]` și `ydr`;
- *dreptunghiul drept* (galben) având coordonatele colțului stânga-jos egale cu `g[i][0]` și `yst`, iar cele ale colțului dreapta-sus egale cu `xdr` și `ydr`.



În continuare, prezentăm codul Python complet pentru rezolvarea acestei probleme:

```
# funcție care citește datele de intrare din fișierul text placa.in
# prima linie din fișier conține coordonatele colțului stânga-jos al
# dreptunghiului inițial, a doua linie pe cele ale colțului
# dreapta-sus, iar pe următoarele linii sunt coordonatele găurilor
def citireDate():
    f = open("placa.in")

    aux = f.readline().split()
    xst, yst = int(aux[0]), int(aux[1])

    aux = f.readline().split()
    xdr, ydr = int(aux[0]), int(aux[1])

    coordonateGauri = []
    for linie in f:
        aux = linie.split()
        coordonateGauri.append((int(aux[0]), int(aux[1])))

    f.close()

    return xst, yst, xdr, ydr, coordonateGauri


# funcția recursivă care prelucrează dreptunghiul curent
def dreptunghiArieMaxima(xst, yst, xdr, ydr):
    global arieMaxima, coordonateGauri, dMaxim

    # considerăm, pe rând, fiecare gaură
    for g in coordonateGauri:
        # dacă gaura curentă se găsește în interiorul dreptunghiului
        # curent, atunci reapelăm funcția pentru cele 4
        # dreptunghiuri care se formează aplicând o tăietură
        # orizontală și una verticală prin gaura curentă
        if xst < g[0] < xdr and yst < g[1] < ydr:
            # dreptunghiurile obținute după o tăietură orizontală
            dreptunghiArieMaxima(xst, yst, xdr, g[1])
            dreptunghiArieMaxima(xst, g[1], xdr, ydr)
            # dreptunghiurile obținute după o tăietură verticală
            dreptunghiArieMaxima(xst, yst, g[0], ydr)
            dreptunghiArieMaxima(g[0], yst, xdr, ydr)
            break
    # dacă dreptunghiul curent nu conține nicio gaură, atunci
    # comparăm aria sa cu aria maximă a unui dreptunghi fără găuri
    # determinată până în momentul respectiv
    else:
        if (xdr-xst)*(ydr-yst) > arieMaxima:
            arieMaxima = (xdr-xst)*(ydr-yst)
            dMaxim = (xst, yst, xdr, ydr)
```

```
#citirea datelor de intrare din fișierul text placa.in
xst, yst, xdr, ydr, coordonateGauri = citireDate()
# inițializăm aria maximă a unui dreptunghi fără găuri
arieMaxima = 0
# inițializăm coordonatele drepunghiul cu arie maximă fără găuri
dMaxim = (0, 0, 0, 0)
# apelăm funcția pentru drepunghiul inițial
dreptunghiArieMaxima(xst, yst, xdr, ydr)

# scriem datele de ieșire în fișierul text placa.out
f = open("placa.out", "w")
f.write("Dreptunghiul:\n" + str(dMaxim[0]) + " " + str(dMaxim[1]) +
        "\n" + str(dMaxim[2]) + " " + str(dMaxim[3]))
f.write("\nAria maxima:\n" + str(arieMaxima))
f.close()
```

Deoarece dimensiunile subproblemelor nu sunt aproximativ egale, nu putem determina complexitatea funcției utilizând cele două metode prezentate. Totuși, folosind teorema Akra-Bazzi, se poate demonstra faptul că funcția are complexitatea $\mathcal{O}(l * L)$, unde l și L reprezintă lățimea și lungimea dreptunghiului inițial, respectiv $l = ydr\_init - yst\_init$ și $L = xdr\_init - xst\_init$.