

## SEMINAR 6

### Metoda Greedy. Metoda Divide et Impera

1. Se consideră  $n$  șiruri de numere sortate crescător. Știind faptul că interclasarea a două șiruri de lungimi  $p$  și  $q$  are complexitatea  $\mathcal{O}(p + q)$ , să se determine o modalitate de interclasare a celor  $n$  șiruri astfel încât complexitatea totală să fie minimă.

#### Exemplu:

Fie 4 șiruri sortate crescător având lungimile 30, 20, 10 și 25. Interclasarea primelor două șiruri necesită  $30+20=50$  de operații elementare și obținem un nou șir de lungime 50, deci mai trebuie să interclasăm 3 șiruri cu lungimile 50, 10 și 30. Interclasarea primelor două șiruri necesită  $50+10=60$  de operații elementare și numărul total de operații elementare devine  $50+60=110$ , după care obținem un nou șir de lungime 60, deci mai trebuie să interclasăm două șiruri cu lungimile 60 și 25. Interclasarea celor două șiruri necesită  $60+25=85$  de operații elementare, iar numărul total de operații elementare devine  $110+85=195$ , după obținem șirul final de lungime 85.

Numărul total minim de operații elementare se obține interclasând de fiecare dată cele mai mici două șiruri din puncte de vedere al lungimilor (i.e., primele două minime). Astfel, pentru exemplul de mai sus, prima dată interclasăm șirurile cu lungimile 10 și 20 (primele două minime) folosind  $10+20=30$  de operații elementare și obținem un nou șir de lungime 30, deci mai trebuie să interclasăm 3 șiruri cu lungimile 30, 25 și 30. Interclasăm acum șirurile cu lungimile 25 și 30 folosind  $25+30=55$  de operații elementare și numărul total de operații elementare devine  $30+55=85$ , după care obținem un nou șir de lungime 55, deci mai trebuie să interclasăm două șiruri cu lungimile 30 și 55. Interclasarea celor două șiruri necesită  $30+55=85$  de operații elementare, iar numărul total de operații elementare devine  $85+85=170$  și obținem șirul final de lungime 85.

Pentru a implementa algoritmul optim prezentat vom utiliza o structură de date numită *coadă cu priorități* (*priority queue*), deoarece aceasta permite realizarea operațiilor de inserare în coadă în funcție de prioritate și, respectiv, de extragere a valorii cu prioritate minimă cu complexitatea  $\mathcal{O}(\log_2 n)$ , unde  $n$  reprezintă numărul de elemente din coada cu priorități. Elementele unei cozi cu priorități sunt, de obicei, tupluri de forma (*prioritate, valoare*), dar pot și valori simple, caz în care valoarea este considerată și prioritate.

În limbajul Python, o structură de date de tip coadă cu priorități este implementată în clasa `PriorityQueue` din pachetul `queue`. Principalele metode ale acestei clase sunt:

- `PriorityQueue()` – creează o coadă cu priorități cu lungimea maximă nelimitată;
- `PriorityQueue(max)` – creează o coadă cu priorități cu lungimea maximă `max` ;
- `put(elem)` – inserează în coadă elementul `elem` în funcție de prioritatea sa;
- `get()` – extrage din coadă unul dintre elementele cu prioritate minimă;
- `qsize()` – furnizează numărul de elemente existente în coada cu priorități;
- `empty()` – furnizează `True` dacă respectiva coadă este vidă sau `False` în caz contrar;
- `full()` – furnizează `True` dacă respectiva coadă este plină sau `False` în caz contrar.

**Exemplu:**

```
import queue

# se creeaza o coada cu prioritați de lungime "infinită"
pq = queue.PriorityQueue()

# inseram elemente în coada cu prioritați
pq.put(7)
pq.put(10)
pq.put(3)
pq.put(6)
pq.put(7)
pq.put(10)

# afisam elementele cozii prin extragerea repetată a valorii minime,
# deci în ordine crescătoare: 3 6 7 7 10 10
while not pq.empty():
    print(pq.get(), end=" ")
```

Atenție, o coadă cu prioritați nu este o structură de date iterabilă, deoarece ea este implementată, de obicei, folosind *arbori binari de tip heap* ([https://en.wikipedia.org/wiki/Binary\\_heap](https://en.wikipedia.org/wiki/Binary_heap))! Un arbore binar de tip heap este un arbore binar complet cu proprietatea că valoarea din orice nod neterminal este mai mică decât valorile fiilor săi, deci valoarea din rădăcina sa va fi minimul tuturor valorilor din arbore, iar extragerea valorii minime dintr-un arbore binar de tip heap presupune ștergerea rădăcinii sale și refacerea structurii sale de heap.

Pentru a rezolva problema prezentată, folosind metoda Greedy, vom utiliza o coadă cu prioritați având elemente de forma (*lungime șir, șir*), deci prioritatea unui șir va fi dată de lungimea sa. La fiecare pas, vom extrage din coadă două șiruri având lungimile minime, le vom interclasa, după care vom introduce în coadă șirul obținut prin interclasare. Vom repeta acest procedeu până când coada va avea un singur element, respectiv șirul rezultat prin interclasarea tuturor șirurilor date.

```
import queue

# Funcție pentru interclasarea a două șiruri sortate crescător
def interclasare(a, b):
    i = j = 0

    rez = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            rez.append(a[i])
            i += 1
        else:
            rez.append(b[j])
            j += 1
```

```

    rez.extend(a[i:])
    rez.extend(b[j:])

    return rez

f = open("siruri.txt")

# fiecare linie din fișierul text de intrare siruri.txt
# conține câte un șir ordonat crescător
siruri = queue.PriorityQueue()

for linie in f:
    aux = [int(x) for x in linie.split()]
    siruri.put((len(aux), aux))

f.close()

# t = numărul total de operații elementare efectuate
t = 0
while siruri.qsize() != 1:
    # extragem primele două șiruri a și b cu lungimi minime
    a = siruri.get()
    b = siruri.get()
    # interclasăm șirurile a și b
    r = interclasare(a[1], b[1])
    # actualizăm numărul total de operații elementare
    t = t + len(r)
    # introducem în coada cu priorități șirul r rezultat
    # prin interclasarea șirurilor a și b
    siruri.put((len(r), r))

# afișăm rezultatele
print("Numar minim de operații elementare:", t)
r = siruri.get()[1]
print("Șirul obținut prin interclasarea tuturor șirurilor:", *r)

```

Estimarea complexității acestui algoritm este dificilă, deoarece complexitatea operației de interclasare a celor două șiruri cu lungimile minime depinde de lungimile celor două șiruri, deci nu poate calculată doar în funcție de dimensiunea datelor de intrare, respectiv numărul  $n$  de șiruri.

## 2. Planificarea unor proiecte cu profit maxim – complexitate $O(n \log_2 n)$

Se consideră  $n$  proiecte, pentru fiecare proiect cunoscându-se profitul, un termen limită (sub forma unei zi din lună) și faptul că implementarea sa durează exact o zi. Să se găsească o modalitate de planificare a unor proiecte astfel încât profitul total să fie maxim.

**Exemplu:**

proiecte.in		proiecte.out
BlackFace	2 800	Ziua 1: BestJob 900.0
Test2Test	5 700	Ziua 2: FileSeeker 950.0
Java4All	1 150	Ziua 3: JustDoIt 1000.0
BestJob	2 900	Ziua 5: Test2Test 700.0
NiceTry	1 850	
JustDoIt	3 1000	Profit maxim: 3550.0
FileSeeker	3 950	
OzWizard	2 900	

**Rezolvare:**

În primul rând, vom observa faptul că dacă termenul limită al unui proiect este strict mai mare decât numărul  $n$  de proiecte, atunci putem să-l înlocuim chiar cu  $n$ . Aplicând această observație, putem afirma că soluția prezentată în seminarul anterior, având complexitatea  $\mathcal{O}(n \cdot zi\_max)$ , poate fi ușor modificată într-una cu complexitatea  $\mathcal{O}(n^2)$ !

Rezolvarea optimă a acestei probleme (i.e., cu complexitatea  $\mathcal{O}(n \log_2 n)$ ) constă în următorii pași:

- fiecare termen limită strict mai mare decât numărul  $n$  de proiecte îl vom înlocui cu  $n$ ;
- sortăm proiectele descrescător după termenul limită;
- pentru fiecare zi  $zcrt$  de la  $n$  la 1 procedăm în următorul mod:
  - introducem într-o coadă cu priorități toate proiectele care au termenul limită  $zcrt$ , considerând prioritatea unui proiect ca fiind dată de profitul său;
  - extragem proiectul cu profit maxim și îl planificăm în ziua  $zcrt$ .

Pentru exemplul dat, vom obține o planificare optimă a proiectelor astfel:

Proiectele (sortate descrescător după termenul limită)	Ziua curentă (zcrt)	Coadă cu priorități	Planificarea optimă
Test2Test 5 700 JustDoIt 3 1000 FileSeeker 3 950 BlackFace 2 800 BestJob 2 900 OzWizard 2 900 Java4All 1 150 NiceTry 1 850	5	(700, Test2Test, 5)	Ziua 5: Test2Test
	4	–	Ziua 5: Test2Test Ziua 4: –
	3	(1000, JustDoIt, 3) (950, FileSeeker, 3)	Ziua 5: Test2Test Ziua 4: – Ziua 3: JustDoIt
	2	(950, FileSeeker, 3) (900, BestJob, 2) (900, OzWizard, 2) (800, BlackFace, 2)	Ziua 5: Test2Test Ziua 4: – Ziua 3: JustDoIt Ziua 2: FileSeeker
	1	(900, BestJob, 2) (900, OzWizard, 2) (850, NiceTry, 1) (800, BlackFace, 2) (150, Java4All, 1)	Ziua 5: Test2Test Ziua 4: – Ziua 3: JustDoIt Ziua 2: FileSeeker Ziua 1: BestJob

Deoarece clasa `PriorityQueue` din pachetul `queue` implementează o coadă cu priorități care permite extragerea minimului, vom considera prioritatea unui proiect ca fiind egală cu `-profit`.

În continuare, prezentăm implementarea în limbajul Python a algoritmului Greedy prezentat mai sus:

```
import queue

# funcție care furnizează cheia necesară sortării
# proiectelor descrescător după termenul limită
def cheieTermenLimitaProiect(p):
    return p[1]

fin = open("proiecte.in")
# citim toate liniile din fișier pentru a afla simplu
# numărul n de proiecte
toate_liniile = fin.readlines()
fin.close()

n = len(toate_liniile)
# lsp = lista cu toate proiectele
lsp = []
for linie in toate_liniile:
    aux = linie.split()
    # un proiect va fi un tuplu (-profit, termen limită, denumire)
    # pentru a-l putea insera direct într-o coadă de priorități,
    # iar cheia este -profitul deoarece clasa PriorityQueue
    # implementează o coadă care permite doar extragerea minimului
    lsp.append((-float(aux[2]), min(int(aux[1]), n), aux[0]))

fin.close()

# sortăm proiectele descrescător după termenul limită
lsp.sort(key=cheieTermenLimitaProiect, reverse=True)

# planificarea optimă o vom memora într-un dicționar
# cu intrări de forma zi:proiect
planificare = {k: None for k in range(1, n + 1)}

# coada cu priorități (prioritatea = -profit)
coada = queue.PriorityQueue()

k = 0
profitmax = 0
for zcrt in range(n, 0, -1):
    # încărcăm în coadă toate proiectele care au
    # termenul limită egal cu zcrt
```

```

while k <= n-1 and lsp[k][1] == zcrt:
    coada.put(lsp[k])
    k += 1

# extragem din coadă proiectul cu profit maxim și
# îl planificăm în ziua zcrt
if coada.qsize() > 0:
    planificare[zcrt] = coada.get()
    profitmax += abs(planificare[zcrt][0])

# scriem o planificare optimă în fișierul text proiecte.out
fout = open("proiecte.out", "w")

for z in planificare:
    if planificare[z] != None:
        fout.write("Ziua " + str(z) + ": " + planificare[z][2] + " "
                  + str(abs(planificare[z][0])) + "\n")
fout.write("\nProfit maxim: " + str(profitmax))

fout.close()

```

Așa cum deja am menționat, complexitatea acestui algoritm este  $\mathcal{O}(n \log_2 n)$ .

3. Se consideră o listă *lsb* formată din valori egale cu 0, urmate de valori egale cu 1 (este posibil ca în șir să nu existe nicio valoare egală cu 0 sau nicio valoare egală cu 1). Scrieți o funcție cu complexitate minimă care să furnizeze poziția primei valori egale cu 1 din lista *lsb* sau -1 dacă în listă nu există nicio valoare egală cu 1.

#### Exemple:

```

lsb = [0, 0, 0, 0, 1, 1, 1] => poziția = 4
lsb = [0, 0, 0] => poziția = -1
lsb = [1, 1, 1] => poziția = 0

```

#### Rezolvare:

Vom folosi o variantă modificată a algoritmului de căutare binară, deci un algoritm de tip Divide et Impera. Astfel, considerând faptul că lista conține și valori egale cu 0 și valori egale cu 1 (vom elimina înainte de apelarea funcției cazurile particulare în care lista conține doar valori egale cu 0 sau doar valori egale cu 1), vom avea următoarele cazuri:

- dacă valoarea curentă (i.e., valoarea aflată în mijlocul secvenței curente) este 1:
  - dacă valoarea aflată în stânga sa este 0, atunci returnăm poziția curentă;
  - dacă valoarea aflată în stânga sa este 1, atunci reluăm căutarea primei valori egale cu 1 în stânga poziției curente;
- dacă valoarea curentă este 0, atunci reluăm căutarea primei valori egale cu 1 în dreapta poziției curente.

```

def cautare_binara(lsb, st, dr):
    mij = (st + dr) // 2
    if lsb[mij] == 1:
        if lsb[mij-1] == 0:
            return mij
        else:
            return cautare_binara(lsb, st, mij-1)
    else:
        return cautare_binara(lsb, mij+1, dr)

def pozitie_1(lsb):
    # toate valorile din lista sunt egale cu 1
    if lsb[0] == 1:
        return 0
    # toate valorile din lista sunt egale cu 0
    if lsb[len(lsb) - 1] == 0:
        return -1

    # lista conține și valori egale cu 0 și valori egale cu 1
    return cautare_binara(lsb, 0, len(lsb) - 1)

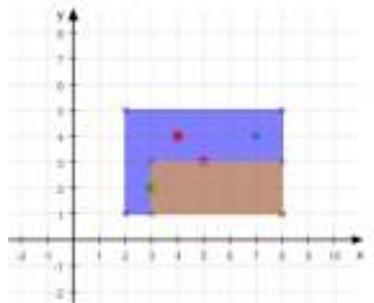
```

Evident, complexitatea acestei funcții este  $\mathcal{O}(\log_2 n)$ .

#### 4. Problema debitării

Se consideră o placă de tablă de formă dreptunghiulară având colțul stânga-jos în punctul  $(xst, yst)$  și colțul dreapta-sus în punctul  $(xdr, ydr)$ . Placa are pe suprafața sa  $n$  găuri cu coordonate numere întregi. Știind că sunt permise doar tăieturi orizontale sau verticale complete, se cere să se decupeze din placă o bucată de arie maximă fără găuri.

##### Exemplu:

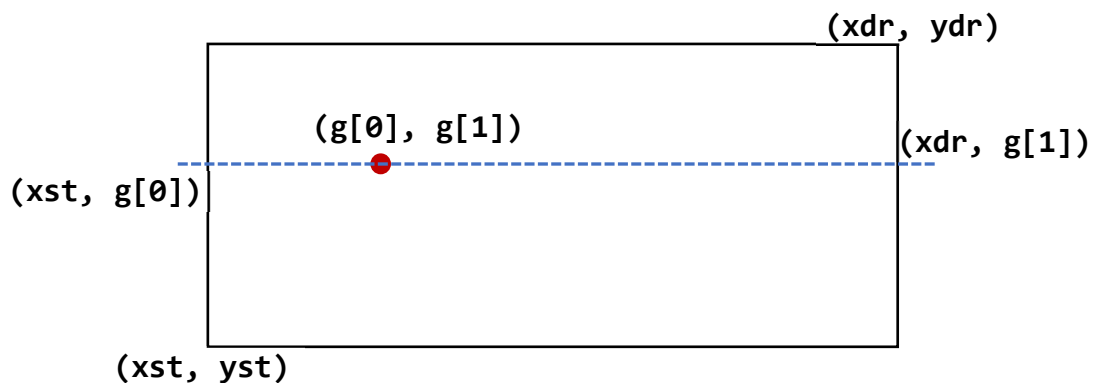
placa.in	placa.out	Explicație
2 1	Dreptunghiul:	<p>Placa de tablă este un dreptunghi având colțul stânga-jos de coordonate <math>(2,1)</math> și colțul dreapta-sus de coordonate <math>(8,5)</math>.</p> <p>În placă sunt date 4 găuri, având coordonatele <math>(3,2)</math>, <math>(4,4)</math>, <math>(5,3)</math> și <math>(7,4)</math>.</p> <p>Dreptunghiul cu aria maximă de 10 și care nu conține nicio gaură are coordonatele <math>(3,1)</math> pentru colțul stânga-jos și <math>(8,3)</math> pentru colțul dreapta-sus.</p> 
8 5	3 1	
3 2	8 3	
4 4	Aria maximă:	
5 3	10	
7 4		

##### Rezolvare:

Vom utiliza tehnica Divide et Impera pentru a rezolva această problemă, respectiv vom defini o funcție dreptunghiArieMaxima( $xst, yst, xdr, ydr$ ) care să prelucreze dreptunghiului curent, indicat prin cei 4 parametri ai săi, astfel:

- dacă dreptunghiul curent conține o gaură, atunci vom reapele funcția pentru fiecare dintre cele 4 dreptunghiuri care se formează după realizarea unei tăieturi complete pe orizontală sau pe verticală;
- dacă dreptunghiul curent nu conține nicio gaură, atunci vom compara aria sa cu aria maximă a unui dreptunghi fără găuri determinată până în momentul respectiv și, eventual, o vom actualiza, împreună cu coordonatele dreptunghiului de arie maximă.

Coordonatele găurilor le vom memora într-o listă de tuple, fiecare tuplu fiind de forma *(abscisa, ordonata)*. Dacă dreptunghiul curent are coordonatele  $(xst, yst, xdr, ydr)$  și gaura curentă  $g$  are coordonatele  $(g[0], g[1])$ , atunci în urma unei tăieturi orizontale complete se vor forma următoarele două dreptunghiuri:



Se observă faptul că dreptunghiul aflat sub tăietură are coordonatele  $(xst, yst, xdr, g[1])$ , iar cel aflat deasupra tăieturii are coordonatele  $(xst, g[0], xdr, ydr)$ . Analog, dreptunghiul aflat în stânga unei tăieturi verticale complete are coordonatele  $(xst, yst, g[0], ydr)$ , iar cel aflat în dreapta are coordonatele  $(g[0], yst, xdr, ydr)$ .

Implementarea algoritmului de tip Divide et Impera în limbajul Python este următoarea:

```
# 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 dreptunghiului cu arie maximă fără găuri
dMaxim = (0, 0, 0, 0)
# apelăm funcția pentru dreptunghiul 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 celor 4 subproblemele nu sunt egale, complexitatea acestui algoritm nu poate fi determinată folosind metodele prezentate la curs!

**Probleme propuse**

1. Rezolvați *problema programării spectacolelor într-o singură sală* utilizând o coadă cu priorități.
2. Se consideră o listă sortată crescător de numere întregi. Scrieți o funcție cu complexitate minimă care să furnizeze numărul de apariții ale unei valori în listă. De exemplu, în lista [1, 1, 2, 2, 2, 2, 4, 4, 4, 4, 5] valoarea 2 apare de 4 ori.

**Indicație de rezolvare:**

Scriem două funcții bazate pe căutarea binară, una pentru a determina prima poziție pe care apare valoarea căutată în lista dată și una pentru a determina ultima poziție pe care apare valoarea în listă. Apelăm prima funcție și, dacă valoarea căutată apare în listă, apelăm și a doua funcție, după care calculăm diferența dintre cele două poziții. Rezolvarea completă a acestei probleme, având complexitatea  $\mathcal{O}(\log_2 n)$ , poate fi găsită aici: <https://www.geeksforgeeks.org/count-number-of-occurrences-or-frequency-in-a-sorted-array/>.

3. Scrieți un algoritm de tip Divide et Impera pentru a rezolva *problema turnurilor din Hanoi* (<https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/>).