Programarea Algoritmilor
– SEMINAR NR. 6 –
(grupa 131)

1. Se consideră n șiruri de numere sortate crescător. Știind faptul că interclasarea a două șiruri de lungimi p și q are complexitatea 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 vom obține 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 vom obține 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, și vom obține șirul final de lungime 85.

   Indicație de rezolvare: Se folosește o coadă de priorități pentru a păstra șirurile sortate crescător după numărul de elemente (cheia este dată de lungimea șirului). La fiecare pas se extrag din coadă și se interclasează cele două șiruri de lungime minimă, iar apoi șirul rezultat este adăugat în coadă. Astfel, pentru exemplul de mai sus, vom interclasa mai întâi șirurile cu lungimile minime 10 și 20, ceea ce necesită 10+20=30 de operații elementare și vom obține un nou șir de lungime 30, deci mai trebuie să interclasăm 3 șiruri cu lungimile 30, 30 și 25. Interclasarea celor două șiruri de lungime minimă necesită 30+25=55 de operații elementare și numărul total de operații elementare devine 30+55=85, după care vom obține 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 și numărul total de operații elementare devine 85+85=170, după care vom obține șirul final de lungime 85.

   Algoritm de complexitate O(n*log(n)):

```
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 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 = 0
while siruri.qsize() != 1:
    a = siruri.get()
    b = siruri.get()
    t += a[0] + b[0]
    # print(t, a[1], b[1])
    r = interclasare(a[1], b[1])
    siruri.put((len(r), r))


print("Numar minim de comparari: ", t)

# Șirul obținut prin interclasarea tuturor șirurilor date
# se va scrie la sfârșitul fișierului de intrare, pe o linie nouă
f = open("siruri.txt", "a")

r = siruri.get()[1]

f.write("\n" + " ".join([str(x) for x in r]))

f.close()
```

2. **Planificarea unor proiecte cu profit maxim –** complexitate O(n*log(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	

Indicație de rezolvare: În primul rând, se observă faptul că dacă termenul limită al unui proiect este strict mai mare decât numărul n de proiecte, putem să-l înlocuim chiar cu n. După aceea, sortăm descrescător proiectele după termenul limită și pentru fiecare zi z de la n la 1 introducem într-o coadă cu priorități toate proiectele care au termenul limită z și extragem proiectul cu profit maxim și îl planificăm în ziua z, deci cheia folosită în coada cu priorități va fi profitul proiectului.

Algoritm de complexitate O(n*log(n)):

```
import queue

def cmp_data_proiect(p):
    return p[1]


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

n = len(toate_liniile)
lsp = []
for linie in toate_liniile:
    aux = linie.split()
    # vom memora un proiect printr-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]), int(aux[1]), aux[0]))

fin.close()

lsp.sort(key=cmp_data_proiect, reverse=True)

programare = {k: None for k in range(1, n + 1)}

coada = queue.PriorityQueue()

k = 0
profitmax = 0
for z in range(n, 0, -1):
    while k <= n-1 and lsp[k][1] == z:
        coada.put(lsp[k])
        k += 1

    if coada.qsize() > 0:
        programare[z] = coada.get()
        profitmax += abs(programare[z][0])

fout = open("proiecte_profit_maxim.txt", "w")

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

fout.close()
```

Observație: Pentru algoritmul de complexitate $O(n^2)$ vezi seminarul 5.

- Divide et Impera -

3. Se consideră un șir ls format din n 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 care să furnizeze, cu o complexitate minimă, poziția primului 1 din șirul ls sau valoarea -1 dacă nu există nicioi valoare egală cu 1 în șir.
   Exemple:
   ls = [0,0,0,0,1,1,1] => 4
   ls = [0,0] => -1
   ls = [1,1,1] => 0

   ***Indicație de rezolvare*:** Se va folosi o variantă modificată a algoritmului de căutare binară.

   Algoritm de complexitate O(log(n)):

```
def cbin(ls, st, dr):
    if st > dr:
        return -1

    mij = (st + dr) // 2
    if ls[mij] == 1:
        if mij == 0 or ls[mij-1] == 0:
            return mij
        else:
            return cbin(ls, st, mij-1)
    else:
        if mij == len(ls) - 1:
            return -1
        else:
            return cbin(ls, mij+1, dr)
```

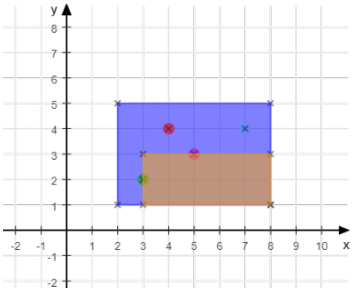
   TEMĂ: Având un șir format din n numere întregi sortate crescător, să se găsească numărul de apariții în șir ale unei valori v date.
   Exemplu: s = [1,1,2,2,2,2,3,4,4,4,4,5] , v = 2 => numărul de apariții = 4

   Indicație de rezolvare: Scriem două funcții bazate pe căutarea binară, una pentru a determina prima poziție pe care apare valoarea v în șirul dat și una pentru a determina ultima poziție pe care apare valoarea v în șir. Apelăm prima funcție și, dacă valoarea v apare în șirul dat, apelăm și a doua funcție, după care calculăm diferența dintre cele două poziții.

   Algoritm de complexitate O(log(n)): https://www.geeksforgeeks.org/count-number-of-occurrences-or-frequency-in-a-sorted-array/

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

Observație: Pentru rezolvarea completă vezi laborator 6.

Algoritm de complexitate O((xdr-xst)*(ydr-yst)):

```
def citireDate():
    f = open("placa.in")

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

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

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

    f.close()

    return xst, yst, xdr, ydr, coordonateGauri


def dreptunghiArieMaxima(xst, yst, xdr, ydr):
    global arieMaxima, coordonateGauri, dMaxim

    for g in coordonateGauri:
        if xst < g[0] < xdr and yst < g[1] < ydr:
            dreptunghiArieMaxima(xst, yst, g[0], ydr)
            dreptunghiArieMaxima(g[0], yst, xdr, ydr)
            dreptunghiArieMaxima(xst, yst, xdr, g[1])
            dreptunghiArieMaxima(xst, g[1], xdr, ydr)
            break
    else:
        if (xdr-xst)*(ydr-yst) > arieMaxima:
            arieMaxima = (xdr-xst)*(ydr-yst)
            dMaxim = (xst, yst, xdr, ydr)


xst, yst, xdr, ydr, coordonateGauri = citireDate()

arieMaxima = 0
dMaxim = (0, 0, 0, 0)
dreptunghiArieMaxima(xst, yst, xdr, ydr)
```

```
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()
```

- Backtracking -

5. Descompunerea unui număr natural ca sumă de numere naturale nenule.

Variante (exemple pentru n = 4)**:**

a) toate descompunerile posibile (1+1+1+1, 1+1+2, 1+2+1, 1+3, 2+1+1, 2+2, 3+1, 4)

```
def bkt(k):
    global sol, n

    for v in range(1, n-k+1):
        sol[k] = v
        scrt = sum(sol[:k+1])
        if scrt <= n:
            if scrt == n:
                print(sol[:k+1])
            else:
                bkt(k+1)


n = int(input("n = "))
sol = [0]*n
bkt(0)
```

b) descompuneri distincte, respectiv nu au aceiași termeni în altă ordine (1+1+1+1, 1+1+2, 1+3, 2+2, 4) => elementele soluției vor fi păstrate în ordine crescătoare => vom inițializa elementul curent sol[k] al soluției cu valoarea elementului anterior sol[k-1], astfel:

```
for v in range(1 if k == 0 else sol[k-1], n - k + 1):
```

c) descompuneri distincte cu termeni distincți (1+3, 4) => elementele soluției vor fi păstrate în ordine strict crescătoare:

```
for v in range(1 if k == 0 else sol[k - 1]+1, n - k + 1):
```

d) descompuneri cu termeni distincți (1+3, 3+1, 4) => verificăm $X[k] \notin X[0:k]$ (la fel ca în problema generării permutărilor):

```
if scrt <= n and sol[k] not in sol[:k]:
```

e) soluție cu lungime dată: $\leq p, == p, \geq p$ (pentru $p = 3$: 1+1+2, 1+2+1, 2+1+1)

```
def bkt(k):
    global sol, n, p

    for v in range(1, n-k+1):
        sol[k] = v
        scrt = sum(sol[:k+1])
        if scrt <= n:
            if scrt == n and k == p-1:
                print(sol[:k+1])
            else:
                if k < p-1:
                    bkt(k+1)
```

```
n = int(input("n = "))
p = int(input("p = "))
sol = [0]*p
bkt(0)
```

6. Se consideră n spectacole pentru care se cunosc intervalele de desfășurare. Să se găsească toate planificările cu număr maxim de spectacole care se pot efectua într-o singură sală astfel încât, în cadrul fiecărei planificări, spectacolele să nu se suprapună.

Exemplu:

spectacole.in	spectacole.out
10:00-11:20 Scufita Rosie 09:30-12:10 Punguta cu doi bani 08:20-09:50 Vrajitorul din Oz 11:30-14:00 Capra cu trei iezi 12:10-13:10 Micul Print 14:00-16:00 Povestea porcului 15:00-15:30 Frumoasa si Bestia	08:20-09:50 Vrajitorul din Oz 10:00-11:20 Scufita Rosie 12:10-13:10 Micul Print 15:00-15:30 Frumoasa si Bestia 08:20-09:50 Vrajitorul din Oz 10:00-11:20 Scufita Rosie 12:10-13:10 Micul Print 14:00-16:00 Povestea porcului 08:20-09:50 Vrajitorul din Oz 10:00-11:20 Scufita Rosie 11:30-14:00 Capra cu trei iezi 15:00-15:30 Frumoasa si Bestia 08:20-09:50 Vrajitorul din Oz 10:00-11:20 Scufita Rosie 11:30-14:00 Capra cu trei iezi 14:00-16:00 Povestea porcului

Indicație de rezolvare:

a) Folosind metoda Greedy, calculăm numărul maxim de spectacole (nms) care se pot planifica în sală fără suprapuneri.

b) Folosind metoda Backtracking, generăm aranjamente de n luate câte nms.

```
# comparator dupa ora de sfarsit a unui spectacol
def cmp_spectacole(sp):
    return sp[2]

# functie care determina numarul maxim de spectacole care pot fi
# programate intr-o sala folosind metoda Greedy
def numarMaximSpectacole(lsp):
    lsp.sort(key=cmp_spectacole)

    ult = "00:00"
    cnt = 0
    for sp in lsp:
        if sp[1] >= ult:
            cnt += 1
            ult = sp[2]

    return cnt
```

```
# generarea aranjamentelor de n luate cate nms
# folosind metoda backtracking
def bkt(k):
    global sol, nms, fout, lsp

    for v in range(len(lsp)):
        sol[k] = v
        if (v not in sol[:k]) and (k == 0 or lsp[sol[k]][1] >=
                                                lsp[sol[k-1]][2]):
            if k == nms-1:
                for p in sol:
                    fout.write(lsp[p][1] + "-" + lsp[p][2] + " " +
                                                lsp[p][0] + "\n")
                fout.write("\n")
            else:
                bkt(k+1)


fin = open("spectacole.txt")

# lsp = lista spectacolelor
lsp = []
for linie in fin:
    aux = linie.split()
    # ora de inceput si ora de sfarsit pentru spectacolul curent
    tsp = aux[0].split("-")
    lsp.append((" ".join(aux[1:]), tsp[0], tsp[1]))

fin.close()

fout = open("programari.txt", "w")

# nms = numarul maxim de spectacole = lungimea solutiilor care vor fi
generate cu backtracking
nms = numarMaximSpectacole(lsp)

sol = [0] * nms
bkt(0)

fout.close()
```