SEMINAR 5
Complexitatea algoritmilor. Metoda Greedy

1. *Problema candidatului majoritar*
Se consideră o listă v formată din n numere naturale nenule reprezentând voturile a n alegători. Să se afișeze, dacă există, câștigătorul alegerilor, adică un candidat care a obținut cel puțin $\left[\frac{n}{2}\right] + 1$ voturi (*candidatul majoritar*).

Exemplu:
- dacă $v = [1, 5, 5, 1, 1, 5]$, atunci nu există niciun câștigător al alegerilor
- dacă $v = [7, 3, 7, 4, 7, 7]$, atunci candidatul 7 a câștigat alegerile

Observație: Dacă există un candidat majoritar, atunci el este unic!

Rezolvare:
Vom prezenta mai multe variante de rezolvare a acestei probleme, cu diferite complexități computaționale. Fiecare variantă va fi implementată folosind o funcție care va avea ca parametru lista voturilor și va returna candidatul care a câștigat alegerile sau 0 dacă nu există niciun câștigător.

În prima variantă de rezolvare vom calcula direct numărul de voturi primite de fiecare candidat distinct:

```
def castigator(voturi):
    for v in set(voturi):
        if voturi.count(v) > len(voturi) // 2:
            return v
    return 0
```

Deoarece numărul candidaților distincți poate fi cel mult n, iar metoda `count` are complexitatea maximă $\mathcal{O}(n)$, această funcție va avea complexitatea maximă $\mathcal{O}(n^2)$.

În a doua variantă de rezolvare vom sorta crescător lista `voturi` și apoi vom verifica dacă ea conține o secvență de voturi egale având lungimea strict mai mare decât `n//2`:

```
def castigator(voturi):
    voturi.sort()

    candidat = voturi[0]
    nr_voturi = 1
    for vot in voturi[1:]:
        if vot == candidat:
            nr_voturi = nr_voturi + 1
            if nr_voturi > len(voturi) // 2:
                return candidat
```

```
        else:
            candidat = vot
            nr_voturi = 1
    return 0
```

Sortarea listei `voturi` se realizează cu complexitatea $\mathcal{O}(n \log_2 n)$, iar căutarea unei secvenţe cu lungimea strict mai mare decât `n//2` are complexitatea maximă $\mathcal{O}(n)$, deci această funcţie va avea complexitatea maximă $\mathcal{O}(n \log_2 n)$.

A treia variantă de rezolvare se obţine observând faptul că în lista voturilor sortată crescător singurul posibil câştigător este elementul din mijlocul listei, pentru că orice secvenţă formată din cel puţin `n//2+1` elemente egale trebuie să conţină şi elementul de pe poziţia `n//2`:

```
def castigator(voturi):
    voturi.sort()
    n = len(voturi)
    if voturi.count(voturi[n//2]) > n//2:
        return voturi[n//2]
    return 0
```

Sortarea listei `voturi` se realizează cu complexitatea $\mathcal{O}(n \log_2 n)$, iar metoda `count` are complexitatea maximă $\mathcal{O}(n)$, deci şi această funcţie va avea complexitatea maximă tot $\mathcal{O}(n \log_2 n)$.

În a patra variantă de rezolvare mai întâi vom construi un dicţionar `nr_voturi` format din perechi `candidat distinct : număr voturi`, după care vom verifica dacă există un câştigător:

```
def castigator(voturi):
    nr_voturi = {}
    for x in voturi:
        if x not in nr_voturi:
            nr_voturi[x] = 1
        else:
            nr_voturi[x] += 1
    for x in nr_voturi:
        if nr_voturi[x] > len(voturi) // 2:
            return x
    return 0
```

Complexitatea computaţională a acestei funcţii este $\mathcal{O}(n)$, deci este optimă, dar are dezavantajul de a utiliza memorie suplimentară (dicţionarul). Atenţie, dicţionarul `nr_voturi` ar putea fi creat şi prin `nr_voturi = {x: voturi.count(x) for x in set(voturi)}`, dar complexitatea acestei operaţii ar fi $\mathcal{O}(n^2)$, deci complexitatea funcţiei ar creşte la $\mathcal{O}(n^2)$!

Algoritmul optim pentru rezolvarea acestei probleme este *algoritmul Boyer-Moore*, elaborat în anul 1981 de către informaticienii americani Robert Boyer și J Strother Moore. În articolul publicat în 1981 (https://www.cs.utexas.edu/~boyer/mjrty.pdf), cei doi autori își descriu algoritmul într-un mod foarte sugestiv: "*Imaginați-vă o sală în care s-au adunat toți alegătorii care au participat la vot, iar fiecare alegător are o pancartă pe care este scris numele candidatului pe care l-a votat. Să presupunem că alegătorii se încaieră între ei, iar în momentul în care se întâlnesc față în față doi alegători care au votat candidați diferiți, aceștia se doboară reciproc. Evident, dacă există un candidat care a obținut mai multe voturi decât toate voturile celorlalți candidați cumulate (i.e., un candidat majoritar), atunci alegătorii săi vor câștiga lupta și, la sfârșitul luptei, toți alegătorii rămași în picioare sunt votanți ai candidatului majoritar. Totuși, chiar dacă nu există o majoritate clară pentru un anumit candidat, la finalul luptei pot rămâne în picioare alegători care au votat toți un același candidat, fără ca acesta să fie majoritar. Astfel, dacă la sfârșitul luptei mai există alegători rămași în picioare, președintele adunării trebuie neapărat să realizeze o numărare a tuturor voturilor acordate candidatului votat de alegătorii respectivi pentru a verifica dacă, într-adevăr, el este majoritar sau nu.*"

În implementarea acestui algoritm vom utiliza o variabilă `majoritar` care va reține candidatul cu cele mai mari șanse de a fi majoritar până în momentul respectiv și un contor `avantaj` care va reține numărul alegătorilor care au votat cu el și încă nu au fost doborâți de votanți ai altor candidați. Vom prelucra cele *n* voturi unul câte unul și, notând cu `v` votul curent, vom actualiza cele două variabile astfel:

- dacă `avantaj == 0`, atunci `majoritar = v` și `avantaj = 1` (posibilul candidat majoritar curent nu mai are niciun avantaj, deci el va fi înlocuit de candidatul căruia i-a fost acordat votul curent);
- dacă `avantaj > 0`, atunci verificăm dacă votul curent este pro sau contra posibilului candidat majoritar curent (i.e., `majoritar == v`) și actualizăm contorul `avantaj` în mod corespunzător.

Dacă la sfârșit, după ce am terminat de analizat toate voturile în modul prezentat mai sus, vom avea `avantaj > 0`, atunci vom realiza o numărare a tuturor voturilor acordate posibilului candidat majoritar rămas pentru a verifica dacă, într-adevăr, el este candidatul majoritar:

```
def castigator(voturi):
    avantaj = 0
    majoritar = None
    for v in voturi:
        if avantaj == 0:
            avantaj = 1
            majoritar = v
        elif v == majoritar:
            avantaj += 1
        else:
            avantaj -= 1
```

```
    if avantaj == 0:
        return 0

    if voturi.count(majoritar) > len(voturi) // 2:
        return majoritar

    return 0
```

Se observă faptul că algoritmul Boyer-Moore rezolvă această problemă în mod optim, deoarece are complexitatea computațională $\mathcal{O}(n)$ și nu folosește memorie suplimentară!

2. Se citește o listă de numere naturale sortată strict crescător și un număr natural S. Să se afișeze toate perechile distincte formate din valori distincte din lista dată cu proprietatea că suma lor este egală cu S.

Exemplu: Pentru lista $L = [2, 5, 7, 8, 10, 12, 15, 17, 25]$ și $S = 20$, trebuie afișate perechile $(5, 15)$ și $(8, 12)$.

Vom prezenta mai multe variante de rezolvare a acestei probleme, cu diferite complexități computaționale (vom nota cu `n` lungimea listei `L`). Fiecare variantă va fi implementată folosind o funcție cu parametrii lista `L` și suma `S` și care returnează o listă cu perechile cerute (evident, aceasta poate fi și vidă!).

În prima variantă de rezolvare vom căuta, pentru fiecare element `L[i]` al listei, valoarea sa complementară față de `S` (i.e., `S - L[i]`) în sublista `L[i+1:]`:

```
def perechi(L, S):
    rez = []
    for i in range(len(L)):
        if S-L[i] in L[i+1:]:
            rez.append((L[i], S-L[i]))
    return rez
```

Căutarea valorii `S-L[i]` în sublista `L[i+1:]` se realizează liniar, cu complexitatea maximă $\mathcal{O}(n)$, deci această funcție va avea complexitatea $\mathcal{O}(n^2)$. Totuși, complexitatea medie poate fi îmbunătățită observând faptul că putem opri căutarea valorii `S-L[i]` în sublista `L[i+1:]` în momentul în care `L[i] >= S//2`.

În a doua variantă de rezolvare vom îmbunătăți complexitatea funcției înlocuind căutarea liniară a valorii `S-L[i]` în sublista `L[i+1:]` cu o căutare binară:

```
def perechi(L, S):
    rez = []
    for i in range(len(L)):
        st = i+1
        dr = len(L)-1
        while st <= dr:
            mij = (st+dr)//2
```

```
            if S-L[i] == L[mij]:
                rez.append((L[i], S-L[i]))
                break
            elif S-L[i] < L[mij]:
                dr = mij-1
            else:
                st = mij+1
    return rez
```

Deoarece căutarea binară a valorii `S-L[i]` în sublista `L[i+1:]` are complexitatea maximă $\mathcal{O}(\log_2 n)$, această funcție va avea complexitatea $\mathcal{O}(n\log_2 n)$.

În a treia variantă de rezolvare vom îmbunătăți și mai mult complexitatea funcției, înlocuind căutarea binară a valorii `S-L[i]` în sublista `L[i+1:]` cu testarea apartenenței sale la mulțimea `M` formată din elementele listei `L`:

```
def perechi(L, S):
    M = set(L)
    rez = []
    for x in M:
        if x <= S//2 and x != S-x and S-x in M:
            rez.append((x, S - x))
    return rez
```

Condiția `x <= S//2` este necesară pentru a evita includerea în soluție a perechilor formate din aceleași valori, dar în ordine inversă (e.g., perechile `(5, 15)` și `(15, 5)`), iar condiția `x != S-x` este necesară pentru a evita includerea în soluție a perechilor formate din două valori egale (e.g., perechea `(10, 10)`). Deoarece crearea mulțimii `M` are complexitatea $\mathcal{O}(n)$, iar testarea apartenenței valorii `S-L[i]` în mulțimea `M` are, în general, complexitatea $\mathcal{O}(1)$, această funcție va avea complexitatea $\mathcal{O}(n)$, dar folosind memorie suplimentară.

Varianta optimă de rezolvare, având complexitatea $\mathcal{O}(n)$ și neutilizând memorie suplimentară, se bazează pe *metoda arderii lumânării la două capete* (*two pointers*). Astfel, vom parcurge lista `L` simultan din ambele capete, folosind 2 indici `st` și `dr` (inițial, `st = 0` și `dr = len(L)-1`), în următorul mod:

- dacă `L[st]+L[dr]<S`, atunci `st=st+1` (suma elementelor curente este prea mică și putem să o creștem doar trecând la următorul element din partea stânga a listei);
- dacă `L[st]+L[dr]>S`, atunci `dr=dr-1` (suma elementelor curente este prea mare și putem să o micșorăm doar trecând la următorul element din partea dreapta a listei);
- dacă `L[st]+L[dr]=S`, atunci adăugăm perechea `(L[st],L[dr])` la soluție, după care actualizăm ambii indici, respectiv `st=st+1` și `dr=dr-1`.

```
def perechi(L, S):
    st = 0
    dr = len(L) - 1
    rez = []
```

```
    while st < dr:
        if L[st] + L[dr] < S:
            st = st + 1
        elif L[st] + L[dr] > S:
            dr = dr - 1
        else:
            rez.append((L[st], L[dr]))
            st = st + 1
            dr = dr - 1
    return rez
```

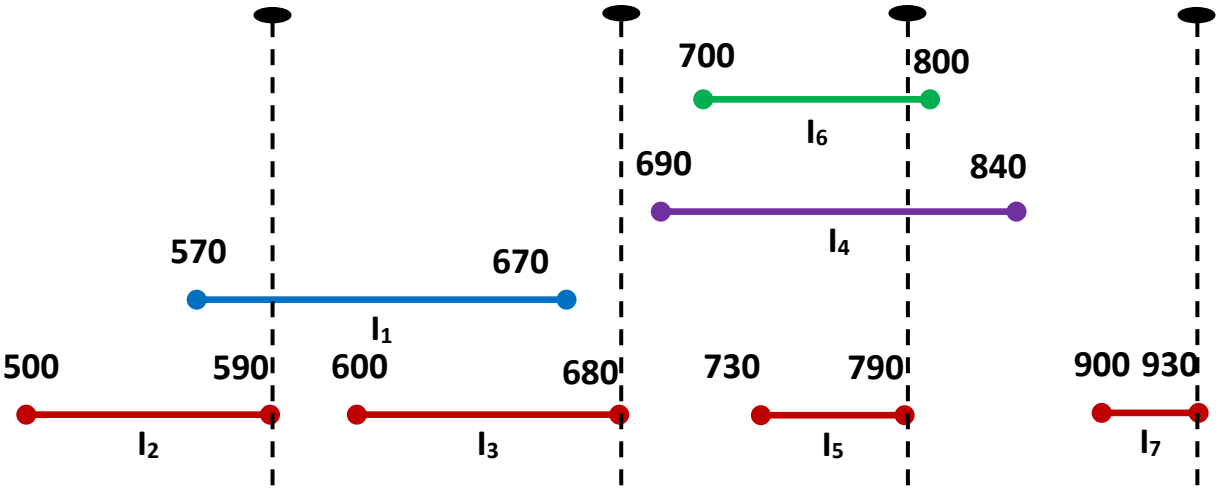
Argumentaţi faptul că această variantă de rezolvare are complexitatea $\mathcal{O}(n)$!

3. *Problema mulţimii de acoperire*
Fie n intervale închise $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$. Să se determine o mulţime M cu număr minim de elemente astfel încât $\forall k \in \overline{1,n}, \exists x \in M$ astfel încât $x \in I_k = [a_k, b_k]$. Mulţimea M se numeşte *mulţime de acoperire* a şirului de intervale respectiv.

Exemplu:

intervale.txt	acoperire.txt
570 670	590
500 590	680
600 680	790
690 840	930
730 790	
700 800	
900 930	

Rezolvare:
Problema mai este cunoscută şi sub numele de *problema cuielor*: "Se consideră n scânduri, fiecare fiind dată printr-un interval închis de pe axa reală. Să se determine numărul minim de cuie pe care trebuie să le batem astfel încât în fiecare scândură să fie bătut cel puţin un cui. Se consideră faptul că orice cui are o lungime suficient de mare pentru a trece prin oricâte scânduri este nevoie.".



Parcurgând scândurile de la stânga spre dreapta, în ordinea crescătoare a extremităţilor din dreapta, se observă faptul că primul cui trebuie să fie bătut în extremitatea dreaptă a primei scânduri (i.e., scândura cu extremitatea dreaptă minimă). Pentru fiecare dintre scândurile rămase verificăm dacă există deja un cui bătut în ea (i.e., extremitatea sa stângă este strict mai mică decât poziția ultimului cui bătut), iar în cazul în care nu există, batem cuiul în extremitatea sa dreaptă:

```
def cheieExtremitateDreapta(interval):
    return interval[1]


fin = open("intervale.txt")
intervale = []
for linie in fin:
    aux = linie.split()
    intervale.append((int(aux[0]), int(aux[1])))
fin.close()

intervale.sort(key=cheieExtremitateDreapta)

M = [intervale[0][1]]
for icrt in intervale[1:]:
    if icrt[0] > M[len(M)-1]:
        M.append(icrt[1])

fout = open("acoperire.txt", "w")
fout.write("\n".join([str(x) for x in M]))
fout.close()
```

Evident, algoritmul prezentat are complexitatea $\mathcal{O}(n\log_2 n)$. Observaţi faptul că problema se rezolvă într-un mod foarte asemănător cu problema programării unui număr maxim de spectacole folosind o singură sală. De ce? Dacă am considera cele n intervale ca fiind nişte spectacole/activităţi, ce semnificaţie ar avea mulţimea de acoperire a lor?

4. Fie n intervale închise $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$. Să se determine reuniunea intervalelor date, precum şi lungimea sa.

Exemplu:

intervale.txt	**reuniune.txt**
570 670	Reuniunea intervalelor:
500 590	[500, 680]
690 840	[690, 840]
600 680	[900, 930]
730 790	
700 800	Lungimea reuniunii: 360
900 930	

Rezolvare:
Pentru a calcula reuniunea celor n intervale, am putea să păstrăm într-o listă intervalele din care aceasta este formată la un moment dat (atenţie, reuniunea unor intervale nu este neapărat tot un interval!) şi să actualizăm intervalele respective pentru fiecare nou interval analizat dintre cele n date:

Interval	Reuniune
[100, 300]	[100, 300]
[700, 900]	[100, 300] [700, 900]
[400, 600]	[100, 300] [700, 900] [400, 600]
[920, 990]	[100, 300] [700, 900] [400, 600]
[500, 650]	[100, 300] [700, 900] [400, 650]
[200, 750]	[100, 750] [200, 900] => [100, 900] => [100, 900] [200, 750]

Putem observa faptul că această modalitate de rezolvare are complexitatea $\mathcal{O}(n^2)$, deoarece fiecare interval dintre cele n date trebuie reunit cu toate intervalele din reuniune cu care nu este disjunct, respectiv intervalul curent trebuie adăugat la reuniune dacă este disjunct cu toate intervalele din reuniune. După actualizarea în acest mod a intervalelor din reuniune s-ar putea ca ele să nu mai fie disjuncte (vezi ultima linie din tabelul de mai sus), deci mai trebuie efectuată o operaţie suplimentară de reactualizare a tuturor intervalelor din reuniune. Această problemă apare deoarece intervalul [200, 750] este analizat prea târziu şi, în acel moment, el se intersectează cu mai multe intervale din reuniunea curentă.

Pentru a eficientiza rezolvarea acestei probleme este suficient să parcurgem intervalele în ordinea crescătoare a capetelor din stânga, deoarece în acest mod vom evita problema apărută anterior:

Interval	Reuniune
[100, 300]	[100, 300]
[200, 750]	[100, 750]
[400, 600]	[200, 750]
[500, 650]	[200, 750]

Interval	Reuniune
[700, 900]	[200, 900]
[920, 990]	[200, 900] [920, 990]

Parcurgând astfel cele n intervale date, se poate observa faptul că toate intervalele care nu sunt disjuncte sunt analizate grupat, iar din momentul în care intervalul curent este disjunct cu reuniunea curentă nu se va mai întâlni niciun alt interval care să nu fie disjunct cu intervalele din reuniunea curentă, deci acestea nu mai trebuie reactualizate la sfârșit!

Lungimea unui interval $[a, b]$ este definită astfel:

$$l([a,b]) = \begin{cases} b-a, & \text{dacă } a \leq b \\ 0, & \text{dacă } a > b \end{cases}$$

Lungimea reuniunii a două intervale se calculează astfel:

$$l([a,b] \cup [c,d]) = l([a,b]) + l([c,d]) - l([a,b] \cap [c,d])$$

Explicitând formula de mai sus în raport de pozițiile relative ale intervalelor $[a, b]$ și $[c, d]$, obținem următoarea formulă echivalentă:

$$l([a,b] \cup [c,d]) = \begin{cases} b-a, & \text{dacă } [c,d] \subseteq [a,b] \\ b-a+d-c, & \text{dacă } [a,b] \cap [c,d] = \emptyset \\ b-a+d-b, & \text{dacă } [a,b] \cap [c,d] \neq \emptyset \end{cases}$$

Pentru a testa mai simplu faptul că intervalul curent este inclus în reuniunea curentă, vom sorta intervalele în ordinea crescătoare a capetelor din stânga și, în cazul în care acestea sunt egale, în ordinea descrescătoare a capetelor din dreapta.

Ținând cont de toate observațiile de mai sus, implementarea în limbajul Python a acestui algoritm, având complexitatea $\mathcal{O}(n\log_2 n)$, este următoarea:

```
# funcție care furnizează cheia necesară sortării intervalelor
# crescător în raport de capetele din stânga și, în cazul unora
# egale, descrescător după capetele din dreapta
def cheieIntervale(t):
    return t[0], -t[1]

# citim datele de intrare din fișierul text "intervale.txt"
# care conține pe fiecare linie extremitățile unui interval
f = open("intervale.txt")
intervale = []
for linie in f:
    aux = linie.split()
    intervale.append((int(aux[0]), int(aux[1])))
f.close()
```

```
# sortăm intervalele crescător în raport de capetele din stânga și,
# în cazul unora egale, descrescător după capetele din dreapta
intervale.sort(key=cheieIntervale)

# minr = capătul din stânga al reuniunii curente
minr = intervale[0][0]
# maxr = capătul din stânga al reuniunii curente
maxr = intervale[0][1]
# reuniune = listă care conține intervalele reuniunii curente
reuniune = []
# lungime = lungimea reuniunii curente
lungime = maxr - minr

# parcurgem, pe rând, intervalele date, începând cu al doilea
for i in range(1, len(intervale)):
    # intervalul curent este inclus în reuniunea curentă
    if intervale[i][1] <= maxr:
        continue
    # intervalul curent nu este disjunct cu reuniunea curentă
    elif intervale[i][0] < maxr:
        lungime += intervale[i][1] - maxr
        maxr = intervale[i][1]
    # intervalul curent este disjunct cu reuniunea curentă,
    # deci am terminat de analizat un grup de intervale care
    # nu sunt disjuncte și reinițializăm reuniunea curentă cu
    # intervalul curent
    else:
        lungime += intervale[i][1] - intervale[i][0]
        reuniune.append((minr, maxr))
        minr = intervale[i][0]
        maxr = intervale[i][1]

# adăugăm ultima reuniune curentă la reuniunea intervalelor
reuniune.append((minr, maxr))

# scriem datele de ieșire în fișierul text "reuniune.txt"
fout = open("reuniune.txt", "w")
fout.write("Reuniunea intervalelor:\n")
for icrt in reuniune:
    fout.write("[{}, {}]\n".format(icrt[0], icrt[1]))
fout.write("\nLungimea reuniunii: " + str(lungime))
fout.close()
```

Încheiem prezentarea acestei probleme menționând faptul că algoritmul de rezolvare nu este unul de tip Greedy, deoarece nu se determină un optim global folosind optime locale! Totuși, am ales să prezentăm această problemă în capitolul dedicat metodei Greedy deoarece ea se utilizează, de obicei, pentru a determina informații suplimentare într-o problemă de planificare (e.g., durata totală a unor activități, intervalele în care nu se desfășoară nicio activitate etc.).

5. *Planificarea unor proiecte cu profit maxim*
Se consideră n proiecte, pentru fiecare proiect cunoscându-se profitul, un termen limită (sub forma unei zi din lună) şi 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 Test2Test 5 700 Java4All 1 150 BestJob 2 900 NiceTry 1 850 JustDoIt 3 1000 FileSeeker 3 950 OzWizard 2 900	Ziua 1: BestJob 900.0 Ziua 2: FileSeeker 950.0 Ziua 3: JustDoIt 1000.0 Ziua 5: Test2Test 700.0 Profit maxim: 3550.0

Rezolvare:
În primul rând, observăm faptul că numărul maxim de zile în care putem să planificăm proiectele este egal cu maximul `zi_max` al termenelor limită (pentru exemplul dat, avem `zi_max = 5`).

O prima idee de rezolvare ar fi aceea de a planifica în fiecare zi proiectul neplanificat care are profitul maxim. Aplicând acest algoritm pentru datele de intrare de mai sus, vom planifica în ziua 1 proiectul `JustDoIt` (deoarece are profitul maxim de 1000 RON), în ziua 2 vom planifica proiectul `FileSeeker` (deoarece are profitul maxim de 950 RON dintre proiectele care mai pot fi planificate în ziua respectivă), iar în ziua 3 vom planifica proiectul `Test2Test` (singurul care mai pot fi planificat în ziua respectivă), deci vom obține un profit de 1000+950+700 = 2650 RON, mai mic decât profitul maxim de 3550 RON.

O altă idee de rezolvare ar fi aceea de a planifica în fiecare zi proiectul cu profit maxim care are termenul limită în ziua respectivă. Aplicând acest algoritm pentru datele de intrare de mai sus, vom planifica în ziua 1 proiectul `NiceTry` (deoarece are profitul maxim de 850 RON dintre proiectele care au termenul limită egal cu ziua 1, respectiv `Java4All` şi `NiceTry`), în ziua 2 vom planifica proiectul `BestJob` (deoarece are profitul maxim de 900 RON dintre proiectele care au termenul limită egal cu 2, respectiv `BlackFace`, `BestJob` şi `OzWizard`), în ziua 3 vom planifica proiectul `JustDoIt`, în ziua 4 nu vom planifica niciun proiect, iar în ziua 5 vom planifica proiectul `Test2Test`, deci vom obține un profit de 850+900+1000+700 = 3450 RON, mai mic decât profitul maxim de 3550 RON.

Algoritmul optim de tip Greedy pentru rezolvarea acestei probleme se obține observând faptul că în fiecare dintre cele două variante prezentate anterior am programat prea repede proiectele cu profit maxim dintr-o anumită zi (de exemplu, în prima variantă, proiectul `JustDoIt` a fost planificat în ziua 1, deşi el ar fi putut fi programat şi în ziua 2 sau în ziua 3). Astfel, algoritmul Greedy optim constă în parcurgerea proiectelor în

ordinea descrescătoare a profiturilor, iar fiecare proiect vom încerca să-l planificăm cât mai târziu, adică în ziua liberă cea mai apropiată de termenul limită al proiectului. Pentru exemplul de mai sus, vom considera proiectele în ordinea `JustDoIt`, `FileSeeker`, `BestJob`, `OzWizard`, `NiceTry`, `BlackFace`, `Test2Test` şi `Java4All`. Proiectul `JustDoIt` poate fi planificat în ziua 3 (chiar termenul său limită), proiectul `FileSeeker` nu mai poate fi planificat tot în ziua 3, dar poate fi planificat în ziua 2, proiectul `BestJob` nu poate fi planificat în ziua 2, dar poate fi planificat în ziua 1, proiectele `OzWizard`, `NiceTry` şi `BlackFace` nu mai pot fi planificate, proiectul `Test2Test` poate fi planificat chiar în ziua 5, iar proiectul `Java4All` nu mai poate fi planificat. În concluzie, vom planifica proiectele `JustDoIt`, `FileSeeker`, `BestJob` şi `Test2Test`, obţinând profitul maxim de 3550 RON. Argumentaţi singuri corectitudinea acestui algoritm!

În continuare, vom prezenta implementarea în limbajul Python a acestui algoritm de tip Greedy:

```
# funcție care furnizează cheia necesară sortării
# proiectelor descrescător după profit
def cheieProfit(p):
    return p[2]


# citim datele de intare din fișierul text "proiecte.in"
fin = open("proiecte.in")

# lp = lista proiectelor în care un proiect este memorat
# sub forma unui tuplu (denumire, termen limită, profit)
lp = []
for linie in fin:
    aux = linie.split()
    lp.append((aux[0], int(aux[1]), float(aux[2])))

fin.close()

# sortăm proiectele descrescător după profit
lp.sort(key=cheieProfit, reverse=True)

# calculăm numărul maxim de zile în care putem
# să planificăm proiectele
zi_max = max([p[1] for p in lp])

# planificarea optimă va fi construită folosind un dicționar
# cu intrări de forma zi: proiect
planificare = {k: None for k in range(1, zi_max+1)}

# profit = profitul total al echipei
profit = 0
```

```
# parcurgem lista proiectelor și încercăm să planificăm
# fiecare proiect într-o zi cât mai apropiată de termenul său limită
for proiect in lp:
    for z in range(proiect[1], 0, -1):
        if planificare[z] is None:
            planificare[z] = proiect
            profit += proiect[2]
            break

# scriem soluția 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][0] +
                   " " + str(planificare[z][2]) + "\n")
fout.write("\nProfit maxim: " + str(profit))

fout.close()
```

Complexitatea implementării prezentate mai sus este $\mathcal{O}(n^2)$ şi nu este minimă, deşi algoritmul Greedy este optim! Pentru a obţine o implementare care să aibă complexitatea optimă $\mathcal{O}(n \log_2 n)$, fie trebuie să utilizăm o structură de date numită *Union-Find* (https://www.geeksforgeeks.org/job-sequencing-using-disjoint-set-union/), fie o coadă cu priorităţi (vezi seminarul nr. 6).

Probleme propuse

1. O *matrice dublu sortată* este o matrice în care liniile şi coloanele sunt sortate strict crescător. De exemplu, o matrice M dublu sortată cu $m = 5$ linii şi $n = 4$ coloane este următoarea:

$$M = \begin{pmatrix} 7 & 10 & 14 & 21 \\ 10 & 15 & 18 & 22 \\ 14 & 23 & 32 & 41 \\ 41 & 43 & 51 & 71 \\ 66 & 70 & 75 & 90 \end{pmatrix}$$

a) Scrieţi o funcţie care să genereze o matrice dublu sortată de dimensiune $m \times n$ cu elemente aleatorii.
b) Scrieţi 3 funcţii, având complexităţile $\mathcal{O}(mn)$, $\mathcal{O}(m \log_2 n)$ şi $\mathcal{O}(m + n)$, care să verifice dacă un număr x se găseşte sau nu într-o matrice dublu sortată. Funcţia va furniza o poziţie pe care apare valoarea x în matrice, sub forma unui tuplu `(linie, coloană)`, sau valoarea `None` dacă x nu se găseşte în matrice.

2. Modificaţi rezolvarea problemei 5 astfel încât planificarea optimă să nu mai conţină zilele libere (e.g., ziua 4 din planificarea prezentată în exemplul dat).

3. În fiecare zi, n şoferi transmit către Compania Naţională de Administrare a Infrastructurii Rutiere informaţii referitoare la starea unei anumite autostrăzi, respectiv intervale închise de forma $[x, y]$ având semnificaţia "*asfaltul de pe autostradă este avariat între kilometrii x şi y*". Considerând faptul că toate informaţiile transmise de către şoferi sunt corecte şi cunoscând lungimea autostrăzii respective, scrieţi un program care, la sfârşitul zilei respective, să furnizeze angajaţilor Companiei Naţionale de Administrare a Infrastructurii Rutiere următoarele informaţii:

a) porţiunile de autostradă care sunt avariate, sub forma unei reuniuni de intervale;
b) porţiunile de autostradă care nu sunt avariate, sub forma unei reuniuni de intervale deschise;
c) gradul de uzură al autostrăzii, respectiv raportul dintre numărul total de kilometri avariaţi de autostradă şi lungimea autostrăzii.

Exemplu:

autostrada.in	**autostrada.out**	**Explicaţii**

`200`
`57 67`
`50 59`
`69 84`
`100 121`
`60 68`
`73 79`
`160 170`
`70 80`
`120 127`
	`[50, 68]`
	`[69, 84]`
	`[100, 127]`
	`[160, 170]`
	
	`(0, 50)`
	`(68, 69)`
	`(84, 100)`
	`(127, 160)`
	`(170, 200)`
	
	`35%`
		Lungimea autostrăzii este `200` km.
		
		Porţiunile avariate sunt `[50, 68]`, `[69, 84]`, `[100, 127]` şi `[160, 170]`.
		
		Porţiunile neavariate sunt `(0, 50)`, `(68, 69)`, `(84, 100)`, `(127, 160)` şi `(170, 200)`.
		
		Gradul de uzură al autostrăzii este `35%`, deoarece numărul total de kilometri avariaţi este `70`.

4. Se consideră o mulţime A formată din $n + 1$ numere reale şi un număr real x_0. Considerând faptul că mulţimea A conţine cel puţin un număr real nenul, scrieţi un program care să determine un polinom $P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ de grad n având toate elementele mulţimii A drept coeficienţii pentru care valoarea $P(x_0)$ este maximă. De exemplu, pentru $A = \{2, -1, 7, 3\}$ şi $x_0 = 2$, polinomul cerut este $P(x) = 7x^3 + 3x^2 + 2x - 1$.