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 şi 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, eficiența algoritmului 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 $\mathcal{O}(n\log_2 n)$. La fel ca în varianta precedentă, putem îmbunătăți eficiența algoritmului oprind căutarea valorii `S-L[i]` în sublista `L[i+1:]` dacă `L[i] >= S//2`.

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 L:
        if x >= S // 2:
            break
        if S-x in M:
            rez.append((x, S - x))
    return rez
```

Condiția `x >= S//2` este necesară atât pentru a evita includerea în soluție a perechilor formate din aceleași valori, dar în ordine inversă (e.g., perechile `(5, 15)` și `(15, 5)`), dar și 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-x` 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
```

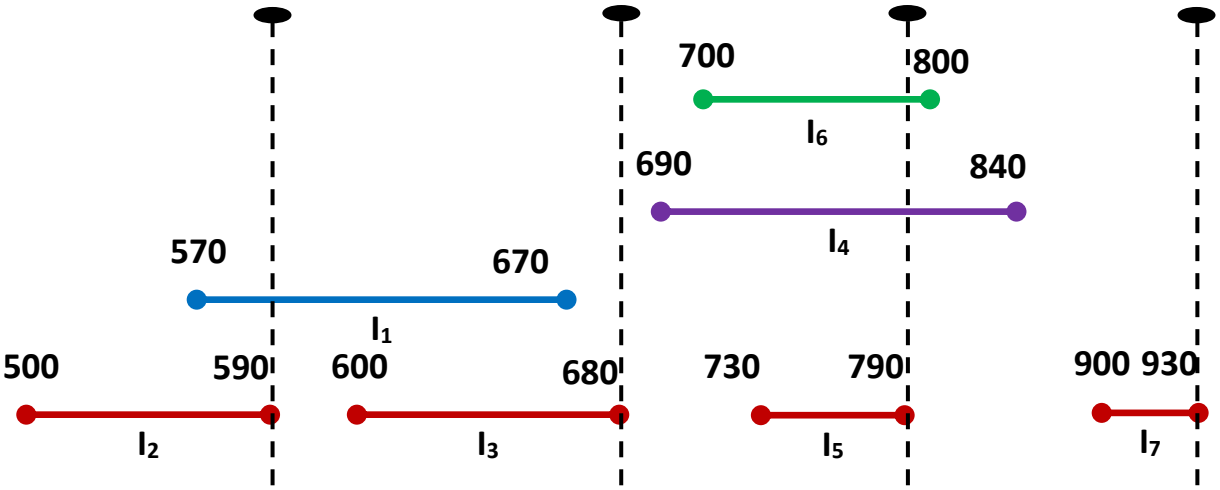
Argumentaţi faptul că această variantă de rezolvare are complexitatea $\mathcal{O}(n)$!

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 $\mathcal{O}(n \cdot zi\_max)$ şi nu este minimă, deşi algoritmul Greedy este optim! Pentru a obţine o implementare care să aibă complexitatea optimă $\mathcal{O}(n \log_2 n)$ trebuie să utilizăm o structură de date numită *Union-Find* (https://www.geeksforgeeks.org/job-sequencing-using-disjoint-set-union/). Un alt algoritm cu complexitatea $\mathcal{O}(n \log_2 n)$ care utilizează o coadă cu priorităţi va fi prezentat în seminarul următor.

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 $P(x) = 7x^3 + 3x^2 + 2x - 1$.

SEMINAR 6
Metoda Greedy. Metoda Divide et Impera

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

Exemplu:
Fie 4 şiruri sortate crescător având lungimile 30, 20, 10 şi 25. Interclasarea primelor două şiruri necesită 30+20=50 de operaţii elementare şi obţinem un nou şir de lungime 50, deci mai trebuie să interclasăm 3 şiruri cu lungimile 50, 10 şi 30. Interclasarea primelor două şiruri necesită 50+10=60 de operaţii elementare şi numărul total de operaţii elementare devine 50+60=110, după care obţinem un nou şir de lungime 60, deci mai trebuie să interclasăm două şiruri cu lungimile 60 şi 25. Interclasarea celor două şiruri necesită 60+25=85 de operaţii elementare, iar numărul total de operaţii elementare devine 110+85=195, după obţinem şirul final de lungime 85.

Numărul total minim de operaţii elementare se obţine interclasând de fiecare dată cele mai mici două şiruri din puncte de vedere al lungimilor (i.e., primele două minime). Astfel, pentru exemplul de mai sus, prima dată interclasăm şirurile cu lungimile 10 şi 20 (primele două minime) folosind 10+20=30 de operaţii elementare şi obţinem un nou şir de lungime 30, deci mai trebuie să interclasăm 3 şiruri cu lungimile 30, 25 şi 30. Interclasăm acum şirurile cu lungimile 25 şi 30 folosind 25+30=55 de operaţii elementare şi numărul total de operaţii elementare devine 30+55=85, după care obţinem un nou şir de lungime 55, deci mai trebuie să interclasăm două şiruri cu lungimile 30 şi 55. Interclasarea celor două şiruri necesită 30+55=85 de operaţii elementare, iar numărul total de operaţii elementare devine 85+85=170 şi obţinem şirul final de lungime 85.

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

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

- `PriorityQueue()` – creează o coadă cu priorităţi cu lungimea maximă nelimitată;
- `PriorityQueue(max)` – creează o coadă cu priorităţi cu lungimea maximă `max` ;
- `put(elem)` – inserează în coadă elementul `elem` în funcţie de prioritatea sa;
- `get()` – extrage din coadă unul dintre elementele cu prioritate minimă;
- `qsize()` – furnizează numărul de elemente existente în coada cu priorităţi;
- `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 priorități de lungime "infinită"
pq = queue.PriorityQueue()

# inseram elemente în coada cu priorități
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 priorități 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)! 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 priorități 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 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, 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)	Coada 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.

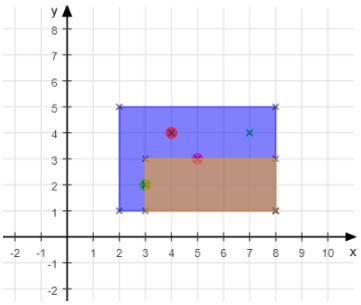
```python
def cautare_binara(lsb, st, dr):
    mij = (st + dr) // 2
    if lsb[mij] == 1:
        if lsb[mij-1] == 0:
            return mij
        else:
            return cautare_binara(lsb, st, mij-1)
    else:
        return cautare_binara(lsb, mij+1, dr)

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 funcţii este $\mathcal{O}(\log_2 n)$.

4. Problema debitării

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

Exemplu:

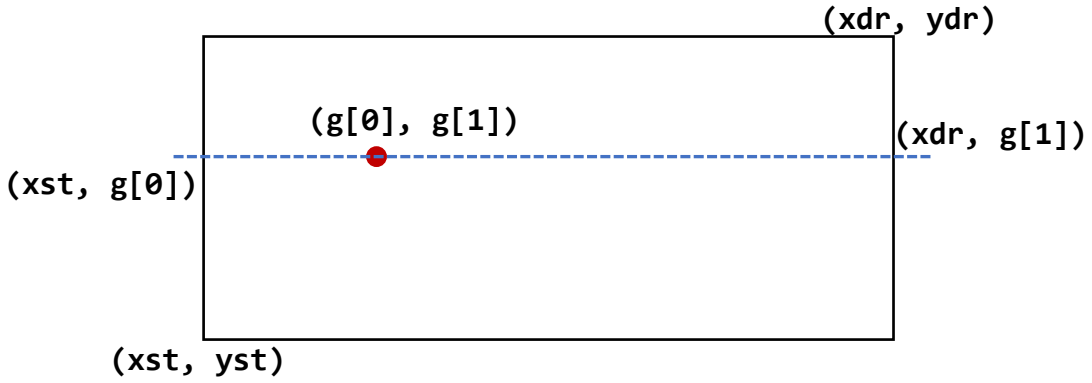
placa.in	placa.out	Explicație
2 1 8 5 3 2 4 4 5 3 7 4	Dreptunghiul: 3 1 8 3 Aria maximă: 10	Placa de tablă este un dreptunghi având colțul stânga-jos de coordonate (2,1) și colțul dreapta-sus de coordonate (8,5). În placă sunt date 4 găuri, având coordonatele (3,2), (4,4), (5,3) și (7,4). Dreptunghiul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

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 parametrii ai săi, astfel:

- dacă dreptunghiul curent conține o gaură, atunci vom reapela 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 tupluri, 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 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 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/).

SEMINAR 7
Metoda programării dinamice. Metoda Backtracking

1. Se consideră n perechi (x, y) cu proprietatea că $x < y$. Să se determine lungimea maximă k a unui lanț de perechi de forma $(x_1, y_1), \dots, (x_i, y_i), (x_{i+1}, y_{i+1}), \dots, (x_k, y_k)$ cu proprietatea că $y_i < x_{i+1}$ pentru orice $i \in \{1,2, \dots, n-1\}$.

Exemplu:
Pentru $n = 6$ și perechile $(12, 15), (6, 8), (5, 7), (20, 30), (9, 11), (13, 18)$, lungimea maximă k a unui lanț cu proprietatea cerută este egală cu 4, iar un posibil astfel de lanț este $(5, 7), (9, 11), (12, 15), (20, 30)$. Atenție, perechile pot fi selectate în orice ordine!

Rezolvare:
Determinarea lungimii maxime a unui lanț de perechi având proprietatea cerută se poate realiza folosind metoda programării dinamice, respectiv o variantă modificată a algoritmului pentru determinarea subșirului crescător maximal, dar aplicată asupra șirului de perechi după sortarea acestora în ordinea crescătoare a valorilor componentelor secunde (i.e., valoarea y_i):

```
# datele de intrare se citesc din fisierul text perechi.txt
# care conține pe fiecare linie câte o pereche de numere
f = open("perechi.txt")
# lp = lista în care vor fi memorate perechile
lp = []
for linie in f:
    aux = linie.split()
    lp.append((int(aux[0]), int(aux[1])))
f.close()

# n = numărul de perechi
n = len(lp)

# sortăm perechile în ordinea crescătoare a componentelor secunde
lp.sort(key=lambda k: k[1])

# modificăm algoritmul pentru determinarea
# subșirului crescător maximal
pred = [-1 for i in range(n)]
lmax = [1 for i in range(n)]

for i in range(n):
    for j in range(i):
        if lp[j][1] < lp[i][0] and lmax[j] + 1 > lmax[i]:
            pred[i] = j
            lmax[i] = lmax[j] + 1
```

```
# determinăm poziția pmax a maximului din lmax
pozmax = lmax.index(max(lmax))

print("Lungimea maximă a unui lanț de perechi:", lmax[pozmax])

# construim un lanț maximal în lista lantmax
lantmax = []
pozcrt = pozmax
while pozcrt != -1:
    lantmax.append(lp[pozcrt])
    pozcrt = pred[pozcrt]

# lanțul a fost reconstituit în ordine inversă,
# deci afișăm lista lantmax inversată
print("Un lanț maximal de perechi:")
print(*lantmax[::-1], sep=", ")
```

Complexitatea algoritmului prezentat este $\mathcal{O}(n^2)$. Se observă foarte uşor faptul că problema este, de fapt, o reformulare a problemei programării într-o singură sală a unui număr maxim de spectacole care să nu se suprapună (i.e., o pereche (x_i, y_i) poate fi considerată intervalul de desfăşurare a unui spectacol), deci poate fi rezolvată folosind algoritmul Greedy deja prezentat şi care are complexitatea $O(n \log_2 n)$!

2. Partiţionarea unei multiset în două submultiseturi cu sume cât mai apropiate
Considerăm un multiset $A = \{a_1, a_2, \dots, a_n\}$ format din n numere naturale nenule ($n \geq 2$). Să se determine o modalitate de partiţionare a multisetului A în două submultiseturi X şi Y (i.e., $X, Y \subseteq A, X \cup Y = A$ şi $X \cap Y = \emptyset$) astfel încât valoarea absolută a diferenţei dintre suma elementelor submultisetului X şi suma elementelor submultisetului Y să fie minimă. Un *multiset* este o extindere a conceptului de mulţime, respectiv elementele unui multiset nu trebuie să mai fie distincte, ci se pot repeta.

Exemplu:
Multisetul $A = \{4, 6, 4, 6, 15\}$ poate fi partiţionat în submultiseturile $X = \{4, 6, 6\}$ şi $Y = \{4, 15\}$ astfel încât să se obţină valoarea minimă posibilă pentru diferenţa dintre sumele elementelor a două submultiseturi de partiţie, respectiv 3. Atenţie, cele două submultiseturi sunt disjuncte din punct de vedere al elementelor utilizate din multisetul A, ci nu din punct de vedere al valorilor acestor elemente, o observaţie asemănătoare fiind valabilă şi pentru reuniunea lor! Astfel, putem considera $X = \{a_1, a_2, a_4\}$ şi $Y = \{a_3, a_5\}$.

Această problemă este o variantă a unei probleme NP-complete denumită *problema partiţiei* (https://en.wikipedia.org/wiki/Partition_problem), în care se cere să se verifice dacă un multiset poate fi partiţionat în două submultiseturi având sumele elementelor egale, iar problema partiţiei este o variantă a altei probleme NP-complete, denumită *problema submulţimii cu sumă dată* (https://en.wikipedia.org/wiki/Subset_sum_problem), în care se cere să se verifice dacă un multiset are o submulţime având suma elementelor egală cu o valoare dată, iar aceasta este, de fapt, un caz particular al variantei discrete a problemei rucsacului!

Dacă notăm cu S suma elementelor multisetului A, cu S_X suma elementelor submultisetului X şi cu S_Y suma elementelor submultisetului Y, cele două submultiseturi X şi Y trebuie determinate astfel încât valoarea expresiei $|S_X - S_Y|$ să fie minimă. Cum $S_X + S_Y = S$, rezultă că $|S_X - S_Y| = |S_X - (S - S_X)| = |2S_X - S| = |S - 2S_X|$. Deoarece valoarea minimă a unei expresii de forma $|E(x)|$ este 0 şi se obţine când $E(x) = 0$, rezultă că suma S_X trebuie să fie cât mai apropiată de valoarea $\left[\frac{S}{2}\right]$, respectiv suma S_X trebuie să fie cea mai mare sumă mai mică sau egală decât $\left[\frac{S}{2}\right]$ care se poate obţine folosind elementele multisetului A. În acest caz, valoarea absolută a diferenţei dintre suma elementelor submultisetului X şi suma elementelor submultisetului Y va fi minimă şi egală cu $S - 2S_X$. Evident, submultisetul Y va fi egal cu $A \backslash X$, unde X este submultisetul determinat astfel încât S_X este cea mai mare sumă mai mică sau egală decât $\left[\frac{S}{2}\right]$ care se poate obţine folosind elementele multisetului A.

Un algoritm pentru rezolvarea acestei probleme, bazat pe metoda programării dinamice, este asemănător cu cel utilizat pentru rezolvarea variantei discrete a problemei rucsacului, respectiv vom utiliza o matrice *sume* cu elemente de tip boolean (`0/False` sau `1/True`), având $n + 1$ linii şi $\left[\frac{S}{2}\right] + 1$ coloane, în care un element $sume[i][j]$ va avea valoarea 1 dacă suma j se poate obţine utilizând primele i elemente $a_1, a_2, \dots, a_i$ ale multisetului A sau 0 în caz contrar. Valorile elementelor matricei *sume* se calculează plecând de la următoarele observaţii:

- $sume[i][0] = 1$ pentru orice $i \in \{0, 1, \dots, n\}$ deoarece suma 0 se poate obţine întotdeauna din primele i elemente ale multisetului A, respectiv neselectând niciunul dintre ele (se consideră faptul că suma elementelor multisetului vid este 0);
- $sume[0][j] = 0$ pentru orice $j \in \left\{1, \dots, \left[\frac{S}{2}\right]\right\}$ deoarece folosind 0 elemente din multisetul A nu se poate obţine nicio sumă j nenulă (atenţie, elementul $sume[0][0]$ va rămâne egal cu 1!);
- folosind primele i elemente $a[1], a[2], \dots, a[i]$ ale multisetului A se poate obţine suma j în următoarele două cazuri:
  - nu se utilizează elementul $a[i]$, ceea ce presupune ca suma j să poată fi obţinută folosind doar primele $i - 1$ elemente $a[1], a[2], \dots, a[i-1]$ ale multisetului A, deci elementul $sume[i-1][j]$ trebuie să fie egal cu 1;
  - se utilizează elementul $a[i]$, ceea ce presupune ca $a[i] \leq j$ şi suma $j - a[i]$ să poată fi obţinută folosind doar primele $i - 1$ elemente $a[1], a[2], \dots, a[i-1]$ ale multisetului A, deci elementul $sume[i-1][j - a[i]]$ trebuie să fie egal cu 1.

Astfel, relaţia de recurenţă care caracterizează substructura optimală a problemei este următoarea:

$$sume[i][j] = \begin{cases} 1, & \text{dacă } 0 \leq i \leq n \text{ şi } j = 0 \\ 0, & \text{dacă } i = 0 \text{ şi } 1 \leq j \leq \left[\frac{S}{2}\right] \\ (sume[i-1][j] == 1) \text{ OR} \\ \Big((a[i] \leq j) \text{ AND } \big(sume[i-1][j - a[i]] == 1\big)\Big), & \text{în orice alt caz} \end{cases}$$

Considerând exemplul dat, respectiv multisetul $A = \{4, 6, 4, 6, 15\}$ având $n = 5$ elemente cu suma $S = 35$, vom obține următoarele valori pentru elementele matricei *sume* (atenție, i nu reprezintă indicele unui element din multisetul A, ci are semnificația descrisă mai sus – *primele i elemente din multisetul A*!):

a_{i-1}	i/j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	2	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0
4	3	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0
6	4	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
15	5	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	1	1	0

Elementele matricei *sume* au fost calculate astfel:

- pe linia 1, doar $sume[1][0] = 1$ și $sume[1][4] = 1$, iar restul elementelor sunt egale cu 0, deoarece folosind doar primul element $a[1] = 4$ din multisetul A se pot obține sumele 0 (dacă nu îl selectăm pe $a[1]$) și 4 (putem să-l selectăm pentru că $a[1] = 4 \leq j = 4$ și $sume[1-1][4-4] = sume[0][0] = 1 \Rightarrow sume[1][4] = 1$);
- $sume[3][8] = 1$, pentru că $a[3] = 4 \leq j = 8$ și $sume[3-1][8-4] = 1$;
- deși elementul $a[4] = 6$ nu poate fi utilizat pentru a obține suma $j = 8$ (deoarece $sume[4-1][8-6] = sume[3][2] = 0$, deci folosind primele 3 elemente ale multisetului A nu putem obține suma $8 - 6 = 2$), totuși, $sume[4][8] = 1$ deoarece suma $j = 8$ se poate obține din primele $i = 3$ elemente ale multisetului A ($sume[3][8] = 1$).

Cea mai mare sumă mai mică sau egală decât $\left[\frac{S}{2}\right]$ care se poate obține folosind elementele multisetului A este dată de valoarea celui mai mare indice j pentru care $sume[n][j] = 1$, în cazul exemplului nostru acesta fiind $j = 16$, deci diferența minimă cerută este egală cu $S - 2 \cdot j = 35 - 32 = 3$.

Pentru a reconstitui cele două submultiseturi de partiție X și Y vom utiliza informațiile din matricea *sume*, astfel:

- considerăm indicele $i = n$ și indicele j determinat anterior;
- dacă $sume[i][j] \neq sume[i-1][j]$, înseamnă că elementul a_i a fost utilizat pentru a obține suma j din primele i elemente ale multisetului A, deci îl vom adăuga în submultisetul X, vom decrementa indicele i, iar indicele j va deveni $j - a[i]$;
- dacă $sume[i][j] = sume[i-1][j]$, înseamnă că elementul a_i nu a fost utilizat pentru a obține suma j din primele i elemente ale multisetului A, deci îl vom adăuga în submultisetul Y și vom decrementa indicele i.

În cazul exemplului de mai sus, avem $j = 16$, iar pentru reconstituirea celor două submultiseturi de partiție X și Y vom urma traseul marcat cu roșu și albastru în matricea

sume, elementele care sunt încadrate cu un pătrat albastru corespunzând elementelor mulțimii $X = \{6, 6, 4\}$, iar cele cu roșu elementelor mulțimii $Y = \{15, 4\}$.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python:

```
# lista A, indexată  de la 1, va conține elementele multisetului A
# (am adăugat în lista A un prim element "inexistent" egal cu 0)
A = [0]
A.extend([int(x) for x in input("Multisetul A: ").split()])

# n = numărul de elemente din multisetul A
n = len(A) - 1
# S = suma elementelor multisetului A
S = sum(A)

# inițializăm elementele matricei sume
# aflate pe prima linie și prima coloană
sume = [[False for j in range(S//2 + 1)] for i in range(n + 1)]
for i in range(n+1):
    sume[i][0] = True

# calculăm restul elementelor matricei sume
# folosind relația de recurență prezentată
for i in range(1, n+1):
    for j in range(1, S//2+1):
        sume[i][j] = sume[i-1][j]
        if A[i] <= j:
            sume[i][j] = sume[i][j] or sume[i-1][j - A[i]]

# reconstituim o soluție a problemei
for j in range(S//2, -1, -1):
    if sume[n][j] == True:
        i = n
        A1 = []
        A2 = []
        while i > 0:
            if sume[i][j] != sume[i-1][j]:
                A1.append(A[i])
                j = j - A[i]
            else:
                A2.append(A[i])
            i = i-1
        print("A1 = {" + ",".join([str(x) for x in A1]) + "}")
        print("A2 = {" + ",".join([str(x) for x in A2]) + "}")
        print("Diferența minimă:", abs(sum(A1) - sum(A2)))
        break
```

Complexitatea algoritmului prezentat este una de tip pseudo-polinomial, fiind egală cu $\mathcal{O}(n\left[\frac{S}{2}\right]) \approx \mathcal{O}(nS) \approx \mathcal{O}\left(n2^{[\log_2 S]}\right)$.

3. Planificarea proiectelor cu bonus maxim

Considerăm n proiecte $P_1, P_2, \dots, P_n$ pe care poate să le execute o echipă de programatori într-o anumită perioadă de timp (de exemplu, o lună), iar pentru fiecare proiect se cunoaşte un interval de timp în care acesta trebuie executat (exprimat prin numerele de ordine a două zile din perioada respectivă), precum şi bonusul pe care îl va obţine echipa dacă proiectul este finalizat la timp (altfel, echipa nu va obţine niciun bonus pentru proiectul respectiv). Să se determine o modalitate de planificare a unor proiecte care nu se suprapun astfel încât bonusul obţinut de echipă să fie maxim. Vom considera faptul că un proiect care începe într-o anumită zi nu se suprapune cu un proiect care se termină în aceeaşi zi!

Exemplu:

proiecte.in	proiecte.out
P1 7 13 850	P4: 02-06 -> 650 RON
P2 4 12 800	P1: 07-13 -> 850 RON
P3 1 3 250	P5: 13-18 -> 1000 RON
P4 2 6 650	P7: 25-27 -> 300 RON
P5 13 18 1000	
P6 4 16 900	Bonusul echipei: 2800 RON
P7 25 27 300	
P8 15 22 900	

Deşi problema este asemănătoare cu *problema planificării unor proiecte cu profit maxim*, prezentată în capitolul dedicat tehnicii de programare Greedy, în care intervalele de executare ale proiectelor sunt restrânse la o singură zi, o strategie de tip Greedy nu va furniza întotdeauna o soluţie corectă. De exemplu, dacă am planifica proiectele în ordinea descrescătoare a bonusurilor, atunci un proiect $P_1([1,10], 1000$ RON) cu bonus mare şi durată mare ar fi programat înaintea a două proiecte $P_2([1,5], 900$ RON) şi $P_3([6,9], 800$ RON) cu bonusuri şi durate mai mici, dar având suma bonusurilor mai mare decât bonusul primului proiect (900+800 = 1700 > 1000). Într-un mod asemănător se pot găsi contraexemple şi pentru alte criterii de selecţie bazate pe ziua de început, pe ziua de sfârşit, pe durată sau pe raportul dintre bonusul şi durata unui proiect!

Pentru a rezolva problema folosind metoda programării dinamice, vom proceda în următorul mod:

- considerăm proiectele $P_1, P_2, \dots, P_n$ ca fiind sortate în ordine crescătoare după ziua de sfârşit (vom vedea imediat de ce);
- considerăm faptul că am calculat bonusurile maxime $bmax_1, bmax_2, \dots, bmax_{i-1}$ pe care echipa le poate obţine planificând o parte dintre primele i proiecte $P_1, P_2, \dots, P_{i-1}$ (sau chiar pe toate!), iar acum trebuie să calculăm bonusul maxim $bmax_i$ pe care echipa îl poate obţine luând în considerare şi proiectul P_i;
- înainte de a calcula $bmax_i$, vom determina cel mai mare indice $j \in \{1,2, \dots, i-1\}$ al unui proiect P_j după care poate fi planificat proiectul P_i (i.e., ziua de început a proiectului P_i este mai mare sau egală decât ziua în care se termină proiectul P_j) şi vom nota acest indice j cu ult_i (dacă nu există nici un proiect P_j după care să poată fi planificat proiectul P_i, atunci vom considera $ult_i = 0$);

- calculăm $bmax_i$ ca fiind maximul dintre bonusul pe care îl echipa poate obține dacă nu planifică proiectul P_i, adică $bmax_{i-1}$, și bonusul pe care îl echipa poate obține dacă planifică proiectul P_i după proiectul P_{ult_i}, adică $bonus_i + bmax_{ult_i}$, unde prin $bonus_i$ am notat bonusul pe care îl primește echipa dacă finalizează proiectul P_i la timp.

Se observă faptul că ult_i se poate calcula mai ușor dacă proiectele sunt sortate crescător după ziua de terminare, deoarece ult_i va fi primul indice $j \in \{i-1, i-2, \dots, 1\}$ pentru care ziua de început a proiectului P_i este mai mare sau egală decât ziua în care se termină proiectul P_j. De asemenea, se observă faptul că valorile ult_i trebuie păstrate într-un tablou, deoarece sunt necesare pentru reconstituirea soluției.

Folosind observațiile și notațiile anterioare, precum și tehnica memoizării, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

$$bmax[i] = \begin{cases} 0, & \text{dacă } i = 0 \\ \max\{bmax[i-1], bonus[i] + bmax[ult[i]]\}, & \text{dacă } i \geq 1 \end{cases}$$

Bonusul maxim pe care îl poate obține echipa este dat de valoarea elementului $bmax[n]$, iar pentru a reconstitui o modalitate optimă de planificare a proiectelor vom utiliza informațiile din matricea $bmax$, astfel:

- considerăm un indice $i = n$;
- dacă $bmax[i] \neq bmax[i-1]$, înseamnă că proiectul P_i a fost utilizat în planificarea optimă, deci îl afișăm și trecem la reconstituirea soluției optime care se termină cu proiectul $P_{ult[i]}$ după care a fost planificat proiectul P_i, respectiv indicele i ia valoarea $ult[i]$;
- dacă $bmax[i] = bmax[i-1]$, înseamnă că proiectul P_i nu a fost utilizat în planificarea optimă, deci trecem la următorul proiect P_{i-1}, decrementând valoarea indicelui i.

Se observă faptul că proiectele se vor afișa invers, deci trebuie utilizată o structură de date auxiliară pentru a le afișa în ordinea intervalelor în care trebuie executate!

Pentru exemplul dat, vom obține următoarele valori pentru elementele listelor ult și $bmax$ (informațiile despre proiectele $P_1, P_2, \dots, P_n$ ale echipei vor fi memorate într-o listă lp cu elemente de tip tuplu și sortare crescător în funcție de ziua de sfârșit):

i	0	1	2	3	4	5	6	7	8
lp	—	P_3 1 3	P_4 2 6	P_2 4 12	P_1 7 13	P_6 4 16	P_5 13 18	P_8 15 22	P_7 25 27
		250	650	800	850	900	1000	900	300
ult	—	0	0	1	2	1	4	4	7
bmax	0	250 0 250	650 250 650	1050 650 800+250	1500 1050 850+650	1500 1500 900+250	2500 1500 1000+1500	2500 2500 900+1500	2800 2500 300+2500

Valorile din lista $bmax$ sunt cele scrise cu roșu și au fost calculate ca fiind maximul dintre cele două valori scrise cu albastru, determinate folosind relația de recurență. De exemplu, $bmax[4] = \max\{bmax[3],\ bonus[4] + bmax[ult[4]]\} = \max\{1050,\ 850 + bmax[2]\} = \max\{1050,\ 850 + 650\} = 1500$.

Pentru exemplul considerat, bonusul maxim pe care îl poate obține echipa este $bmax[8] = 2800$ RON, iar pentru a reconstitui o planificare optimă vom utiliza informațiile din listele $bmax$ și ult, astfel:

- inițializăm un indice $i = n = 8$;
- $bmax[i] = bmax[8] = 2800 \neq bmax[i-1] = bmax[7] = 2500$, deci proiectul $lp[8] = P_7$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[8] = 7$;
- $bmax[i] = bmax[7] = 2500 = bmax[i-1] = bmax[6] = 2500$, deci proiectul $lp[7] = P_8$ nu a fost programat și indicele i devine $i = i - 1 = 6$;
- $bmax[i] = bmax[6] = 2500 \neq bmax[i-1] = bmax[5] = 1500$, deci proiectul $lp[6] = P_5$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[6] = 4$;
- $bmax[i] = bmax[4] = 1500 \neq bmax[i-1] = bmax[3] = 1050$, deci proiectul $lp[4] = P_1$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[4] = 2$;
- $bmax[i] = bmax[2] = 650 \neq bmax[i-1] = bmax[1] = 250$, deci proiectul $lp[2] = P_4$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[2] = 0$;
- $i = 0$, deci am terminat de afișat o modalitate optimă de planificare a proiectelor și ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul Python, considerând faptul că datele de intrare se citesc din fișierul text `proiecte.in`, care conține pe fiecare linie informațiile despre un proiect, în ordinea denumire, ziua inițială, ziua finală și bonusul, iar datele de ieșire se vor scrie în fișierul text `proiecte.out`, în forma din exemplul dat:

```
# funcție folosită pentru sortarea crescătoare a proiectelor
# în raport de data de sfârșit (cheia)
def cheieDataSfarsitProiect(t):
    return t[2]

f = open("proiecte.in")

# lp este lista proiectelor în care am adăugat un prim proiect
# "inexistent" pentru a avea proiectele indexate de la 1
lp = [("", 0, 0, 0)]
for linie in f:
    # 1 proiect = 1 tuplu (denumire, data început, data sfârșit, bonus)
    aux = linie.split()
    lp.append((aux[0], int(aux[1]), int(aux[2]), int(aux[3])))

f.close()

# n = numărul proiectelor
n = len(lp) - 1

# sortăm proiectele crescător după data de sfârșit
lp.sort(key=cheieDataSfarsitProiect)
```

```
# calculăm elementele listelor pmax și ult
bmax = [0] * (n + 1)
ult = [0] * (n + 1)

for i in range(1, n+1):
    for j in range(i-1, 0, -1):
        if lp[j][2] <= lp[i][1]:
            ult[i] = j
            break

    if lp[i][3] + bmax[ult[i]] > bmax[i-1]:
        bmax[i] = lp[i][3] + bmax[ult[i]]
    else:
        bmax[i] = bmax[i-1]

# reconstituim o soluție
i = n
sol = []
while i >= 1:
    if bmax[i] != bmax[i-1]:
        sol.append(lp[i])
        i = ult[i]
    else:
        i -= 1

sol.reverse()

fout = open("proiecte.out", "w")

for ps in sol:
    fout.write("{}: {:02d}-{:02d} -> {} RON\n".format(ps[0],
        ps[1], ps[2], ps[3]))

fout.write("\nBonusul echipei: " + str(bmax[n]) + " RON")

fout.close()
```

Complexitatea algoritmului prezentat este $\mathcal{O}(n^2)$ și poate fi scăzută la $\mathcal{O}(n \log_2 n)$ dacă utilizăm o căutare binară modificată pentru a calcula valoarea $ult[i]$: https://www.geeksforgeeks.org/weighted-job-scheduling-log-n-time/.

4. Modificați algoritmul de tip Backtracking pentru descompunerea unui număr natural ca sumă de numere naturale nenule astfel încât să afișeze doar:

a) *descompunerile distincte*, respectiv descompunerile care nu au aceiași termeni în altă ordine (de exemplu, pentru $n = 4$ aceste descompuneri sunt 1+1+1+1, 1+1+2, 1+3, 2+2 și 4);

În acest caz, vom păstra elementele soluției în ordine crescătoare, inițializând elementul curent $s[k]$ cu valoarea elementului anterior $s[k-1]$ pentru $k \geq 2$, respectiv cu 1 pentru $k = 1$:

```
for v in range(1 if k==1 else sol[k-1], n-k+2):
    .........
```

b) *descompunerile cu termeni distincți*, respectiv descompunerile care nu au termeni egali (de exemplu, pentru $n = 4$ aceste descompuneri sunt 1+3, 3+1 și 4);

În acest caz, vom verifica în condițiile de continuare, în plus, faptul că elementul curent $s[k]$ nu a mai fost folosit deja, respectiv $s[k]$ nu este egal cu niciuna dintre valorile $s[1], s[2], \dots, s[k-1]$:

```
if scrt <= n and sol[k] not in sol[1: k]:
    .........
```

c) *descompuneri distincte cu termeni distincți*, respectiv descompunerile care nu au aceiași termeni în altă ordine și nici nu conțin termeni egali (de exemplu, pentru $n = 4$, aceste descompuneri sunt 1+3 și 4);

În acest caz, vom păstra elementele soluției în ordine strict crescătoare, inițializând elementul curent $s[k]$ cu valoarea $s[k-1]+1$ pentru $k \geq 2$, respectiv cu 1 pentru $k = 1$:

```
for v in range(1 if k == 1 else sol[k-1]+1, n-k+2):
    .........
```

d) *soluțiile ale căror lungimi au o anumită proprietate*, respectiv lungimile lor sunt mai mici, egale sau mai mari decât un număr natural p (de exemplu, pentru $n = 4$, soluțiile având lungimea $p = 3$ sunt 1+1+2, 1+2+1 și 2+1+1).

În acest caz, vom verifica în condițiile de soluție, în plus, faptul că lungimea k a soluției are proprietatea cerută:

```
if scrt == n and k == p:
    .........
```

5. Să se afișeze toate numerele naturale formate din cifre distincte și având suma cifrelor egală cu o valoare c dată. De exemplu, pentru $c = 3$, trebuie să fie afișate numerele: 102, 12, 120, 201, 21, 210, 3 și 30 (nu neapărat în această ordine).

Rezolvare:
Analizând enunțul problemei, observăm faptul că orice număr care este soluție a problemei are cel mult 10 cifre, deoarece acestea trebuie să fie distincte, și problema are soluție doar în cazul în care $c \in \{0,1, \dots,45\}$, deoarece cea mai mare sumă care se poate obține din cifre distincte este egală cu $1 + 2 + \cdots + 9 = 45$.

Pentru a rezolva problema vom utiliza metoda Bactracking, astfel:

- $s[k]$ reprezintă cifra aflată pe poziția k într-un număr (considerăm cifrele unui număr ca fiind numerotate de la stânga spre dreapta), deci obținem $\texttt{min}_k=1$ pentru k=1 (prima cifră a unui număr nu poate fi 0) sau $\texttt{min}_k=0$ pentru k≥2, respectiv $\texttt{max}_k=9$;
- s[1],…,s[k-1],s[k] este soluție parțială dacă cifra curentă s[k] nu a mai fost utilizată anterior, adică s[k]≠s[i] pentru orice i∈{1,...,k-1}, și s[1]+…+s[k]≤c;
- pentru a testa dacă s[1],…,s[k] este soluție vom ține cont de faptul că cifrele s[1],…,s[k] sunt distincte (din condițiile de continuare), deci vom verifica doar condiția suplimentară s[1]+…+s[k]==c.

Implementând algoritmul Backtracking corespunzător observațiilor de mai sus, nu vom obține toate soluțiile, ci doar o parte dintre ele! De exemplu, pentru $c = 3$, **nu** vom obține numerele scrise îngroșat: 102, 12, **120**, 201, 21, **210**, 3 și **30**. Explicația acestui fapt necesită o înțelegere aprofundată a metodei Backtracking: numerele scrise îngroșat sunt soluții care se obțin din soluțiile care nu conțin cifra 0 (de exemplu, numărul **120** se obține din numărul 12, care nu conține cifra 0, prin adăugarea unui 0 la sfârșitul său)! În forma sa generală, algoritmul Backtracking **nu** va furniza niciodată numerele scrise îngroșat, deoarece după găsirea unei soluții a problemei, algoritmul **nu** va încerca niciodată să adauge încă un element (o cifră, în acest caz) la ea! Din acest motiv, o soluție completă care nu modifică foarte mult algoritmul general Backtracking se poate obține astfel: în momentul afișării unei soluții, verificăm dacă ea conține deja o cifră egală cu 0, iar în caz negativ o afișăm încă o dată și-i adăugăm un 0 la sfârșit.

Programul scris în limbajul Python care implementează rezolvarea completă a acestei probleme este următorul:

```
import sys

def bkt(k):
    global c
    for v in range(1 if k == 1 else 0, 10):
        st[k] = v
        scrt = sum(st[1:k+1])
        if st[k] not in st[1:k] and scrt <= c:
            if scrt == c:
                print(*st[1:k+1], sep="")
                if 0 not in st[1:k+1]:
                    print(*st[1:k + 1], 0, sep="")
            else:
                bkt(k+1)


c = int(input("c = "))
if c < 0 or c > 45:
    print("Problema nu are soluție!")
    sys.exit()
```

```
st = [0 for i in range(11)]
print("Toate numerele cerute:")
bkt(1)
```

Complexitatea acestui algoritm poate dedusă foarte greu în raport de valoarea c, dar se poate observa ușor faptul că numărul maxim de soluții se obține pentru $c = 45$ și este egal cu $9! + 9 \cdot 9! = 10! = 3628800$.

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

Rezolvare:
Problema poate fi rezolvată folosind doar metoda Backtracking, respectiv generând toate planificările posibile ale celor n spectacole date și păstrându-le pe cele care sunt corecte și au lungimea maximă. Deoarece ordinea în care sunt planificate spectacolele contează, trebuie utilizată o variantă modificată a generării tuturor aranjamentelor de lungime m ale unei mulțimi cu n elemente. Din cauza faptului că numărul maxim de spectacole care se pot planifica fără suprapuneri poate varia între 1 (dacă toate cele n spectacole se suprapun) și n (dacă niciun spectacol nu se suprapune cu toate celelalte $n - 1$), va trebui să considerăm și valoarea lui m ca fiind cuprinsă între 1 și n, deci numărul total de planificări generate va fi aproximativ $A_n^1 + A_n^2 + \cdots + A_n^n \gg A_n^n = n!$, deci complexitatea acestui algoritm va fi mult mai mare decât $\mathcal{O}(n!)$.

O variantă mai eficientă de rezolvare a acestei probleme o constituie utilizarea metodei Greedy pentru a afla numărul maxim de spectacole nms care se pot programa fără

suprapuneri, cu o complexitate $\mathcal{O}(n \log_2 n)$ și generarea doar a aranjamentelor cu nms elemente ale unei mulțimi cu n elemente:

```
# funcție care determina numărul maxim de spectacole care pot fi
# programate fără suprapuneri folosind metoda Greedy
def numarMaximSpectacole(lsp):
    # sortăm spectacolele în ordinea crescătoare a orelor de sfârșit
    lsp.sort(key=lambda s: s[2])

    # ora de sfârșit a ultimului spectacol programat
    ult = "00:00"
    # cnt = numărul maxim de spectacole care se pot programa corect
    cnt = 0
    for sp in lsp:
        if sp[1] >= ult:
            cnt += 1
            ult = sp[2]

    return cnt


# generarea tuturor programărilor cu număr maxim de spectacole
# folosind metoda backtracking, respectiv generarea aranjamentelor
# cu nms elemente ale celor n spectacole
def bkt(k):
    # fout = fișierul text în care vom scrie soluțiile
    # lsp = lista cu spectacolele
    # n = numărul de spectacole din lista lsp
    global s, nms, fout, lsp, n

    # s[k] = spectacolul aflat pe poziția k în planificarea curentă
    for v in range(n):
        s[k] = v
        if v not in s[:k] \
                and (k == 0 or lsp[s[k]][1] >= lsp[s[k-1]][2]):
            if k == nms-1:
                for p in s:
                    fout.write(lsp[p][1] + "-" + lsp[p][2] + " " +
                               lsp[p][0] + "\n")
                fout.write("\n")
            else:
                bkt(k+1)


fin = open("spectacole.in")

# 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()

# n = numărul de spectacole
n = len(lsp)

fout = open("spectacole.out", "w")

# nms = numărul maxim de spectacole care se pot programa corect
# = lungimea soluțiilor care vor fi generate cu backtracking
nms = numarMaximSpectacole(lsp)

s = [0] * nms
bkt(0)

fout.close()
```

Probleme propuse

1. Fie A un multiset format din n numere naturale nenule şi S un număr natural nenul. Folosind metoda Backtracking, să se afişeze toate submultiseturile lui A care au suma elementelor egală cu S.

2. Fie A un multiset format din n numere naturale nenule şi S un număr natural nenul. Folosind metoda programării dinamice, să se afişeze un submultiset a lui A care are suma elementelor egală cu S.

3. Folosind doar metoda backtracking, generaţi toate subşirurile crescătoare maximale ale un şir format din n numere întregi.

4. Optimizaţi algoritmul de la problema anterioară, utilizând metoda programării dinamice pentru a determina lungimea maximă a unui subşir crescător al şirului dat .