

SEMINAR 1

Operatori pe biți

1. Să se interschimbe valorile a două variabile de tip întreg folosind operatorul \wedge (XOR/"sau exclusiv" pe biți).

Rezolvare:

Operatorul \wedge are următoarele proprietăți:

- a) $x \wedge x = 0$
- b) $x \wedge 0 = x$
- c) $x \wedge y = y \wedge x$
- d) $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

Folosind proprietățile enunțate anterior, putem interschimba valorile a două variabile x și y folosind operatorul \wedge , astfel (corectitudinea algoritmului este demonstrată în comentarii):

$$\begin{aligned} x &= x \wedge y \\ y &= x \wedge y \quad \# \quad y = (x \wedge y) \wedge y = x \wedge (y \wedge y) = x \wedge 0 = x \\ x &= x \wedge y \quad \# \quad x = (x \wedge y) \wedge x = x \wedge (y \wedge x) = (x \wedge x) \wedge y = 0 \wedge y = y \end{aligned}$$

De asemenea, putem observa faptul că expresia $x \wedge y$ furnizează diferențele dintre valorile x și y din punct de vedere al biților aflați pe aceeași poziție. Argumentați corectitudinea algoritmului folosind această observație!

2. Să se verifice dacă un număr natural nenul n este de forma 2^k sau nu. În caz afirmativ să se afișeze exponentul k , altfel să se afișeze un mesaj corespunzător.

Rezolvare:

Un număr natural n este de forma 2^k dacă reprezentarea sa binară conține un singur bit nenul. De exemplu, reprezentarea binară a numărului $n = 512 = 2^9$ este $0b100000000$, în timp ce numărul $n = 600$ are reprezentarea binară $0b1001011000$. Practic, pozițiile biților nenuli din reprezentarea binară a unui număr natural n (numerotate începând cu 0 de la dreapta spre stânga) indică puterile lui 2 pe care trebuie să le însumăm pentru a-l obține pe n (în exemplul următor am considerat numărul $n = 600$):

Poziția bitului	9	8	7	6	5	4	3	2	1	0
Valoarea bitului	1	0	0	1	0	1	1	0	0	0
Puterea lui 2 corespunzătoare poziției	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Astfel, obținem faptul că $n = 600 = 2^9 + 2^6 + 2^4 + 2^3 = 512 + 64 + 16 + 8$.

O primă variantă de rezolvare a problemei constă în eliminarea din numărul n a bitului de pe poziția 0 (denumit și *bitul cel mai puțin semnificativ* sau *bitul de paritate*) cât timp acesta este nul, iar dacă la sfârșit valoarea lui n a devenit 1 atunci n este de forma 2^k :

```
n = int(input("n = "))

aux = n
k = 0
while aux & 1 == 0:
    aux = aux >> 1
    k = k + 1

if aux == 1:
    print("Numarul", n, "este egal cu 2**", k)
else:
    print("Numarul", n, "nu este o putere a lui 2")
```

Deoarece lungimea reprezentării binare a unui număr natural n este egală cu $1 + \lceil \log_2 n \rceil$, rezultă faptul că algoritmul anterior are complexitatea $\mathcal{O}(\log_2 n)$.

O rezolvare mai eficientă a acestei probleme se bazează pe următoarea idee: dacă dintr-un număr natural nenul n scădem 1, atunci în reprezentarea sa binară toți biții egali cu 0 de la sfârșit vor deveni 1 până se va întâlni primul bit egal cu 1, iar acesta va deveni 0. Astfel, valoarea expresiei $n \& (n-1)$ va fi egală cu 0 dacă și numai dacă reprezentarea binară a numărului natural nenul n conține un singur bit nenul (i.e., numărul n este o putere a lui 2), așa cum se poate observa din următoarele exemple:

$n = 512 = 0b1000000000$		$n = 600 = 0b1001011000$
$n-1 = 511 = 0b0111111111$		$n-1 = 599 = 0b1001010111$
$n \& (n-1) = 0b0000000000$		$n \& (n-1) = 0b1001010000$

În plus, dacă $n = 2^k$, atunci putem calcula direct exponentul $k = \log_2 n$. Un program Python în care sunt implementate aceste idei este următorul:

```
import math

n = int(input("n = "))

if n & (n-1) == 0:
    print("Numarul", n, "este egal cu 2**", int(math.log2(n)))
else:
    print("Numarul", n, "nu este o putere a lui 2")
```

Deoarece în majoritatea limbajelor de programare actuale funcțiile care calculează logaritmi sunt foarte rapide, putem considera faptul că algoritmul de mai sus are complexitatea $\mathcal{O}(1)$.

3. Să se determine în mod eficient numărul de biți nenuli din reprezentarea binară a unui număr natural nenul. De exemplu, reprezentarea binară a numărului 600 este $0b1001011000$ și conține 4 biți nenuli.

Rezolvare:

O variantă directă de rezolvare a problemei constă în numărarea biților nenuli din reprezentarea binară a numărului dat:

```
n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))
aux = n
k = 0
while aux != 0:
    if aux & 1 == 1:
        k = k + 1
    aux = aux >> 1
print("Reprezentarea binară a numărului", n, "conține", k, "biți nenuli")
```

Evident, algoritmul de mai sus are complexitatea $\mathcal{O}(\log_2 n)$.

Așa cum deja am menționat, orice număr natural n poate fi scris ca o sumă de puteri ale lui 2, iar pozițiile biților nenuli din reprezentarea sa binară reprezintă, de fapt, exponenții puterilor respective. Astfel, numărul biților nenuli din reprezentarea binară a unui număr natural n este egal cu numărul puterilor lui 2 utilizate pentru scrierea lui n în baza 2 (vezi rezolvarea problemei anterioare).

Pentru a calcula în mod eficient numărul k de biți nenuli din reprezentarea binară a numărului natural n vom folosi ideea din problema anterioară. Astfel, dacă $n \& (n-1)$ este o valoare nenulă, înseamnă că mai există puteri ale lui 2 pe care trebuie să le însumăm pentru a-l obține pe n , deci mai există biți nenuli în reprezentarea binară a lui n , ceea ce înseamnă că trebuie să reluăm procedeul până când n devine 0 (exemplul de mai jos este dat pentru $n = 600$):

```

n = 0b1001011000
n - 1 = 0b1001010111
n = n & (n-1) = 0b1001010000

n = 0b1001010000
n - 1 = 0b1001001111
n = n & (n-1) = 0b1001000000

n = 0b1001000000
n - 1 = 0b1000111111
n = n & (n-1) = 0b1000000000

n = 0b1000000000
n - 1 = 0b0111111111
n = n & (n-1) = 0b0000000000
```

Practic, prin fiecare instrucțiune de atribuire $n = n \& (n-1)$ eliminăm câte un bit nenul din reprezentarea binară a lui n , fără a mai lua în considerare și biții nuli!

Programul Python în care este implementată această idee este următorul:

```
n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))

aux = n
k = 0
while aux != 0:
    k = k + 1
    aux = aux & (aux - 1)

print("Reprezentarea binară a numărului", n, "conține", k, "biți nenuli")
```

Complexitatea acestui algoritmul este dată de numărul k de biți nenuli din reprezentarea binară a numărului natural n . Deoarece valoarea lui k nu se poate calcula direct pe baza valorii lui n , putem afirma doar faptul că algoritmul are complexitatea maximă $\mathcal{O}(\log_2 n)$. Totuși, pentru majoritatea valorilor numărului n , această variantă va fi mai eficientă decât prima variantă de rezolvare prezentată!

Observați faptul că problema anterioară este un caz particular al acestei probleme, respectiv cazul în care suma puterilor lui 2 conține un singur termen!

4. Să se găsească lungimea maximă a unei secvențe de biți egali cu 1 din reprezentarea binară a unui număr natural dat.

Rezolvare:

O variantă directă de rezolvare a acestei probleme constă în actualizarea lungimii $lcrt$ a secvenței curente de biți egali cu 1 și a lungimii maxime $lmax$ a unei secvențe de biți egali cu 1 în funcție de valoarea bitului de paritate, astfel:

```
n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))
aux = n
lcrt = lmax = 0
while aux != 0:
    if aux & 1 == 1:
        lcrt = lcrt + 1
    else:
        lcrt = 0
    if lcrt > lmax:
        lmax = lcrt
    aux = aux >> 1

print("Lungimea maximă a unei secvențe de biți nenului este", lmax)
```

Evident, algoritmul de mai sus are complexitatea $\mathcal{O}(\log_2 n)$.

Pentru a calcula în mod eficient lungimea maximă a unei secvențe de biți nenuli din reprezentarea binară a unui număr natural n vom folosi următoarea idee: prin deplasarea cu o poziție spre stânga a biților numărului n toate secvențele de biți egali cu 1 se vor deplasa cu o poziție spre stânga, deci primului bit din fiecare secvență de biți nenuli din $n \ll 1$ îi va corespunde unui bit egal cu 0 în reprezentarea binară a numărului n . În consecință, prin instrucțiunea de atribuire $n = n \& (n \ll 1)$ vom elimina primul bit al fiecărei secvențe de biți nenuli din reprezentarea binară a numărului n , deci lungimile tuturor secvențelor de biți nenuli se vor micșora cu 1, așa cum se poate observa în exemplul de mai jos, în care $n = 14130$:

```

n = 0b11011100110010
n << 1 = 0b10111001100100
n = n & (n << 1) = 0b10011000100000

n = 0b10011000100000
n << 1 = 0b00110001000000
n = n & (n << 1) = 0b00010000000000

n = 0b00010000000000
n << 1 = 0b00100000000000
n = n & (n << 1) = 0b00000000000000

```

Se observă cu ușurință faptul că numărul n va deveni egal cu 0 după un număr de iterații egal cu lungimea maximă a unei secvențe de biți nenuli din reprezentarea sa binară!

Programul Python în care este implementată această rezolvare este următorul:

```

n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))

aux = n
lmax = 0
while aux != 0:
    aux = aux & (aux << 1)
    lmax = lmax + 1

print("Lungimea maximă a unei secvențe de biți nenului este", lmax)

```

Complexitatea acestui algoritmul este dată de lungimea maximă $lmax$ a unei secvențe de biți nenuli din reprezentarea binară a numărului natural n . Deoarece valoarea lui $lmax$ nu se poate calcula direct pe baza valorii lui n , putem afirma doar faptul că algoritmul are complexitatea maximă $\mathcal{O}(\log_2 n)$. Totuși, pentru majoritatea valorilor numărului n , această variantă va fi mai eficientă decât prima variantă prezentată!

5. Se citește un șir format din cel puțin 3 numere naturale cu proprietatea că fiecare valoare distinctă apare de exact două ori în șir, mai puțin una care apare o singură dată. Să se afișeze valoarea care apare o singură dată în șir. De exemplu, în șirul 100, -7, 20, 20, 1, -7, 1, 3, 100 valoarea 3 apare o singură dată.

Rezolvare:

Prin aplicarea operatorului \wedge între perechi de numere egale se obține valoarea 0, deci aplicând acest operator între toate numerele date vom obține chiar numărul x care apare o singură dată în șir:

```
n = int(input("Numărul de valori: "))
x = 0
for i in range(n):
    v = int(input("Valoare: "))
    x = x ^ v
print("Numărul care apare o singură dată în șir:", x)
```

Complexitatea acestui algoritmul este, evident, $\mathcal{O}(n)$.

6. Să se calculeze numărul x obținut prin aplicarea operatorului XOR între toate elementele tuturor submulțimilor mulțimii $A = \{1, 2, \dots, n\} \subset \mathbb{N}$, mai puțin mulțimea vidă. De exemplu, dacă $n = 3$ obținem $x = (1) \wedge (2) \wedge (3) \wedge (1 \wedge 2) \wedge (1 \wedge 3) \wedge (2 \wedge 3) \wedge (1 \wedge 2 \wedge 3) = 0$ (am folosit parantezele doar pentru a pune în evidență elementele submulțimilor lui A).

Indicație de rezolvare:

O rezolvare a acestei probleme folosind forța brută (i.e., generăm toate submulțimile lui A și calculăm numărul x) va avea complexitatea exponențială $\mathcal{O}(2^n)$.

O variantă de rezolvare mult mai eficientă se obține observând faptul că, pentru $n \geq 2$, fiecare element al mulțimii A va apărea într-un număr par de submulțimi (orice element $k \geq 2$ al mulțimii A va fi adăugat tuturor submulțimilor care se pot forma cu numerele $1, 2, \dots, k-1$, deci elementul k va apărea în 2^{k-1} submulțimi), deci vom obține $x = 0$. Evident, pentru $n = 1$ numărul cerut este $x = 1$. Complexitatea acestei variante de rezolvare este $\mathcal{O}(1)$!

7. Se citesc $n - 1$ numere naturale distincte dintre primele n numere naturale nenule. Să se afișeze numărul lipsă x .

Exemple:

- Dacă se citesc numerele 5, 1, 4, 2, atunci numărul lipsă este $x = 3$.
- Dacă se citesc numerele 1, 2, 3, 4, atunci numărul lipsă este $x = 5$.
- Dacă se citesc numerele 2, 3, 4, 5, atunci numărul lipsă este $x = 1$.

Rezolvare:

Există mai multe metode de rezolvare pentru această problemă, având diverse complexități:

- a) *se caută căutarea fiecare număr de la 1 la n într-o listă cu cele $n - 1$ numere citite*
- complexitate computațională: $\mathcal{O}(n^2)$
 - spațiu de memorie utilizat: $\mathcal{O}(n)$

- b) se sortează cele $n - 1$ numere citite și se caută prima pereche de numere aflate pe poziții consecutive care nu sunt consecutive (atenție la cazurile particulare când lipsește numărul 1 sau numărul n)
- complexitate computațională: $\mathcal{O}(n \log_2 n)$ – dacă folosim Quicksort sau Mergesort
 - spațiu de memorie utilizat: $\mathcal{O}(n)$
- c) se citesc, pe rând, cele $n - 1$ numere și se marchează într-un vector de apariții/frecvențe, după care se caută primul număr nemarcat
- complexitate computațională: $\mathcal{O}(n)$
 - spațiu de memorie utilizat: $\mathcal{O}(n)$
- d) calculăm diferența dintre suma primelor n numere naturale și suma numerelor citite
- complexitate computațională: $\mathcal{O}(n)$
 - spațiu de memorie utilizat: $\mathcal{O}(1)$

Rezolvarea problemei folosind operatori pe biți constă în aplicarea operatorului \wedge între cele $n - 1$ numere citite și toate numerele naturale de la 1 la n (vezi problema 5). De exemplu, dacă se citesc numerele 5, 1, 4, 2 (deci $n = 5$), atunci numărul lipsă este $x = (5 \wedge 1 \wedge 4 \wedge 2) \wedge (1 \wedge 2 \wedge 3 \wedge 4 \wedge 5) = 3$.

Programul Python care implementează această rezolvare este următorul:

```
n = int(input("n = "))
x = 0
for k in range(1, n):
    v = int(input("v = "))
    x = x ^ v ^ k

x = x ^ n
print("Numărul lipsă:", x)
```

Complexitate computațională a acestui algoritm este $\mathcal{O}(n)$, iar spațiul de memorie utilizat este $\mathcal{O}(1)$.

8. Se citesc $n - 2$ numere naturale distincte dintre primele n numere naturale nenule. Să se afișeze numerele lipsă x și y .

Exemple:

- Dacă se citesc numerele 5, 1, 4, atunci numerele lipsă sunt $x = 2$ și $y = 3$.
- Dacă se citesc numerele 1, 3, 5, atunci numerele lipsă sunt $x = 2$ și $y = 4$.
- Dacă se citesc numerele 2, 3, 4, atunci numerele lipsă sunt $x = 1$ și $y = 5$.

Rezolvare:

Această problemă este asemănătoare cu problema anterioară, iar modalitățile de rezolvare ale problemei anterioare pot fi adaptate destul de ușor și pentru ea. Astfel, primele 3 modalități de rezolvare ale problemei anterioare, care utilizează toate un spațiu de memorie de ordinul $\mathcal{O}(n)$, trebuie modificate pentru a găsi două numere lipsă, ci nu

doar unul singur, iar complexitățile computaționale vor rămâne aceleași. A patra variantă de rezolvare, care utilizează un spațiu de memorie de ordinul $\mathcal{O}(1)$ și calculează numărul lipsă ca diferența dintre suma tuturor numerelor de la 1 la n și suma numerelor citite, trebuie modificată pentru a calcula nu numai suma celor două numere lipsă, ci și produsul lor. Din acest motiv, chiar și în limbajul Python, pot să apară erori din cauza valorilor foarte mari implicate în calcule!

În continuare, vom prezenta pașii unui algoritm eficient pentru rezolvarea acestei probleme și în care nu pot să apară erorile menționate anterior. Vom considera $n = 10$ și cele $n-2 = 8$ numere naturale citite memorate într-o listă $v = [3, 7, 1, 10, 5, 8, 4, 9]$, deci numerele lipsă sunt $x = 2$ și $y = 6$.

Pasul 1:

Calculăm valoarea $x \wedge y$ în variabila `x_xor_y`, aplicând operatorul \wedge între numerele din lista v și toate numerele naturale cuprinse între 1 și $n = 10$, deci vom obține $x_xor_y = x \wedge y = 2 \wedge 6 = 0b010 \wedge 0b110 = 0b100 = 4$.

Pasul 2:

Deoarece $x \neq y$, rezultă că $x_xor_y \neq 0$, deci în reprezentarea binară a valorii x_xor_y există cel puțin un bit nenul, aflat pe o poziție p . În exemplul nostru, $x_xor_y = 0b100$, deci poziția respectivă este $p = 2$ (biții dintr-o reprezentare binară se numără începând cu 0, de la dreapta spre stânga).

Pasul 3:

Deoarece bitul aflat pe poziția p în x_xor_y este egal cu 1, înseamnă că biții aflați pe poziția p în x și y sunt diferiți. În exemplul nostru, bitul aflat pe poziția $p = 2$ în $x = 2$ este egal cu 0, iar cel aflat pe poziția $p = 2$ în $y = 6$ este egal cu 1. Astfel, putem împărți toate numerele cuprinse între 1 și n în două grupe disjuncte, în funcție de valoarea bitului de pe poziția p , iar cele două numere lipsă x și y se vor găsi în grupe diferite. În tabelul de mai jos, puteți observa modul în care se împart numerele de la 1 la 10 în cele două grupe menționate, precum și faptul că x face parte din **grupa albastră** (bitul aflat pe poziția $p = 2$ este 0), iar y face parte din **grupa roșie** (bitul aflat pe poziția $p = 2$ este 1):

	Număr	Reprezentare binară
x	1	0001
	2	0010
	3	0011
	4	0100
	5	0101
y	6	0110
	7	0111
	8	1000
	9	1001
	10	1010

Utilizând această observație, putem să calculăm cele două numere lipsă x și y , aplicând operatorul \wedge între toate numerele cuprinse între 1 și n care fac parte din grupa sa și toate

numerele din lista v care fac parte din aceeași grupă. În exemplul dat, vom calcula valoarea $x = (1 \wedge 2 \wedge 3 \wedge 8 \wedge 9 \wedge 10) \wedge (3 \wedge 1 \wedge 8 \wedge 10 \wedge 9) = 2$, unde am utilizat prima pereche de paranteze doar pentru a pune în evidență numerele cuprinse între 1 și n care fac parte din grupa albastră, iar a doua pereche de paranteze am utilizat-o pentru numerele din lista v care fac parte din aceeași grupă.

Pasul 4:

Pentru a determina cel de-al doilea număr lipsă y , putem să procedăm la fel ca mai sus pentru grupa roșie sau, mai simplu, putem să-l calculăm cu expresia $y = x_xor_y \wedge x$. În exemplul dat, obținem $y = x_xor_y \wedge x = 4 \wedge 2 = 6$.

Înainte de a implementa acest algoritm în limbajul Python, vom prezenta o metodă eficientă de a testa din ce grupă face parte unui număr natural nenul t , în funcție de cel mai din dreapta bit egal cu 1 din reprezentarea binară a valorii x_xor_y (în cadrul algoritmului se poate utiliza orice alt bit nenul din reprezentarea binară a valorii x_xor_y , dar bitul cel mai din dreapta cu această proprietate se poate determina cel mai rapid). Practic, mai întâi vom crea o mască binară mb care va avea toți biții egali cu 0, mai puțin bitul de pe poziția p corespunzătoare celui mai din dreapta bit nenul din x_xor_y , astfel (am considerat $x_xor_y = 424 = 0b110101000$, deci $p = 3$):

<pre> x_xor_y = 0b110101000 x_xor_y - 1 = 0b110100111 ~(x_xor_y - 1) = 0b001011000 mb = x_xor_y & ~(x_xor_y - 1) mb = 0b000001000 </pre>	<p>toți biții nuli din dreapta poziției $p = 3$ și bitul nenul de pe poziția $p = 3$ își comută valorile, iar restul biților nu se modifică</p> <p>toți biții nenuli din dreapta poziției $p = 3$ și bitul nul de pe poziția $p = 3$ își comută valorile (deci revin la valorile inițiale din x_xor_y), iar restul biților își comută valorile (deci au valori complementare celor inițiale din x_xor_y)</p> <p>calculăm masca binară mb</p>
--	---

Folosind masca binară mb putem testa foarte ușor dacă un număr natural t face parte din prima grupă, respectiv verificăm dacă $mb \& t == 0$ (pentru a testa dacă t face parte din a doua grupă folosim condiția inversă $mb \& t != 0$).

Implementarea acestui algoritm în limbajul Python este următoarea:

```

n = int(input("n = "))

v = []
x_xor_y = 0
for i in range(1, n-1):
    t = int(input("v[" + str(i) + "] = "))
    v.append(t)
    x_xor_y = x_xor_y ^ t

```

```

for i in range(1, n+1):
    x_xor_y = x_xor_y ^ i

mb = x_xor_y & ~(x_xor_y - 1)

x = 0
for t in v:
    if t & mb == 0:
        x = x ^ t

for i in range(1, n+1):
    if i & mb == 0:
        x = x ^ i

y = x_xor_y ^ x

print("Numerele lipsă: ", x, " și ", y)

```

Complexitate computațională a acestui algoritm este $\mathcal{O}(n)$, iar spațiul de memorie utilizat este tot $\mathcal{O}(n)$.

Probleme propuse

1. Se citește un șir format din numere naturale cu proprietatea că fiecare valoare distinctă din șir apare de un număr par de ori, mai puțin una care apare de un număr impar de ori. Să se afișeze valoarea care apare de un număr impar de ori în șirul dat.

2. Să se calculeze numărul x obținut prin aplicarea operatorului XOR între toate elementele tuturor submulțimilor mulțimii $A = \{a_1, a_2, \dots, a_n\} \subset \mathbb{N}$, mai puțin mulțimea vidă. De exemplu, dacă $A = \{2, 7, 18\}$ vom obține numărul $x = (2) \wedge (7) \wedge (18) \wedge (2 \wedge 7) \wedge (2 \wedge 18) \wedge (7 \wedge 18) \wedge (2 \wedge 7 \wedge 18) = 0$ (am folosit parantezele doar pentru a pune în evidență elementele submulțimilor lui A).

3. Fie x și y două numere naturale nenule. Calculați numărul biților din reprezentarea binară a numărului x a căror valoare trebuie comutată pentru a obține numărul y .

Indicație de rezolvare: Se calculează numărul biților nenuli din numărul $t = x \wedge y$.

4. Fie n un număr natural. Să se determine cea mai mică putere a lui 2 mai mare sau egală decât numărul n .

Indicație de rezolvare: <https://www.geeksforgeeks.org/smallest-power-of-2-greater-than-or-equal-to-n/>

5. Să se determine în mod eficient numărul de biți nuli din reprezentarea binară a unui număr natural nenul. De exemplu, reprezentarea binară a numărului 600 este 0b1001011000 și conține 6 biți nuli.

Indicație de rezolvare:

Această problema poate fi redusă la problema rezolvată 4 prin comutarea fiecărui bit din reprezentarea binară a numărului natural nenul n dat (i.e., orice bit egal cu 0 devine 1 și orice bit egal cu 1 devine 0) și determinarea numărului de biți nenuli din noul număr obținut. Pentru a comuta toți biții numărului n vom aplica operatorul \wedge între numărul dat n și o mască binară mb având toți biții egali cu 1, așa cum se poate observa din exemplul de mai jos pentru $n = 600$:

```

n = 0b1001011000
mb = 0b1111111111
n ^ mb = 0b0110100111

```

Știind că lungimea reprezentării binare a numărului n este $k = 1 + \lceil \log_2 n \rceil$, se observă ușor faptul că masca binară mb este egală cu $(1 \ll k) - 1$.

SEMINAR 2

Șiruri de caractere

1. Să se verifice dacă un șir de caractere t apare ca subșir într-un șir s , iar în caz afirmativ să se afișeze toate pozițiile la care începe t în s . De exemplu, șirul $t = \text{"abc"}$ apare ca subșir în șirul $s = \text{"abccabcababc"}$ începând cu pozițiile 0, 4 și 9.

Rezolvare:

Vom căuta în mod repetat șirul t în șirul s folosind metoda `find(șir)`:

```
s = input("s = ")
t = input("t = ")

p = s.find(t)
if p == -1:
    print("Șirul '" + t + "' nu apare în șirul '" + s + "'!")
else:
    print("Pozițiile pe care începe șirul '" + t +
          "' în șirul '" + s + "':")
    while p != -1:
        print(p)
        p = s.find(t, p + len(t))
```

Dacă șirul t se găsește în șirul s începând cu o poziție p , atunci căutarea sa se poate relua cu una dintre următoarele două poziții (vezi ultima linie din programul de mai sus):

- $p + \text{len}(t)$, caz în care se vor găsi aparițiile nesuprapuse ale șirului t în s (de exemplu, pentru $s = \text{"ababababa"}$ și $t = \text{"aba"}$ se vor afișa pozițiile 0 și 4);
- $p + 1$, caz în care se vor găsi aparițiile suprapuse ale șirului t în s (de exemplu, pentru $s = \text{"ababababa"}$ și $t = \text{"aba"}$ se vor afișa pozițiile 0, 2, 4 și 6).

2. Se citește un cuvânt w , un număr natural nenul p și un șir format din n cuvinte. Să se afișeze toate cuvintele care sunt p -rime cu w , respectiv ultimele p caractere ale sale coincid cu ultimele p caractere ale lui w . De exemplu, pentru $w = \text{"mere"}$, $p = 2$ și cuvintele "pere", "teste" și "programare" (decă $n = 3$), trebuie să fie afișate cuvintele "pere" și "programare".

Rezolvare:

Extragem în șirul r ultimele p caractere din cuvântul w și apoi verificăm, pentru fiecare dintre cele n cuvinte citite, dacă ultimele p caractere ale sale coincid cu șirul r :

```
w = input("Cuvântul w: ")
p = int(input("Numărul p: "))
n = int(input("Numărul de cuvinte: "))
```

```

sol = ""
rima = w[-p:]
for i in range(n):
    cuv = input("Cuvânt: ")
    if rima == cuv[-p:]:
        sol = sol + cuv + " "
sol = sol.strip()

if sol != "":
    print(str(p) + "-rimele cuvântului '" + w + "': " + sol)
else:
    print("Niciun cuvânt dat nu este " + str(p) + "-rimă " +
        "a cuvântului '" + w + "':")

```

Pentru a verifica dacă ultimele p caractere ale cuvântului `cuv` coincid cu șirul `rima` putem folosi metoda `endswith(șir)`, astfel:

```

.....
if cuv.endswith(rima):
    sol = sol + cuv + " "
.....

```

3. Se citește o propoziție sau o frază în care cuvintele sunt despărțite între ele prin spații și semnele de punctuație uzuale. Să se afișeze toate cuvintele distincte de lungime maximă din propoziția sau fraza dată. De exemplu, în propoziția "Ana are prune și gutui verzi, dar mai multe prune decât gutui!" cuvintele distincte de lungime maximă, egală cu 5, sunt: "prune", "gutui", "verzi", "multe" și "decât".

Rezolvare:

Pentru a împărți propoziția sau fraza dată în cuvinte, vom folosi metoda `split(separator)`. Deoarece această metodă poate să împartă un șir în subșiruri doar în funcție de un singur separator, mai întâi vom înlocui toate semnele de punctuație uzuale cu spații, după care vom utiliza această metodă fără parametri. Cuvintele de lungime maximă le vom păstra în șirul de caractere `rez`, astfel: în cazul în care cuvântul curent `cuv` are o lungime strict mai mare decât lungimea maximă `lmax` determinată până în acel moment, atunci înlocuim `rez` cu `cuv` și `lmax` cu lungimea cuvântului curent, iar în cazul în care cuvântul curent are lungimea egală cu `lmax` în concatenăm la șirul `rez`, încadrat între două spații. Pentru a verifica dacă un cuvânt de lungime maximă a fost deja adăugat în șirul `rez`, îl vom căuta în acesta, încadrat între două spații, pentru a evita găsirea sa ca subșir al altui cuvânt.

```

text = input("Propoziția sau fraza: ")

separatori = ",.!:;?!"
for x in separatori:
    text = text.replace(x, " ")

rez = " "
lmax = 0

```

```

for cuv in text.split():
    if len(cuv) > lmax:
        rez = " " + cuv + " "
        lmax = len(cuv)
    elif len(cuv) == lmax:
        if " " + cuv + " " not in rez:
            rez += cuv + " "

rez = rez.strip()

print("Cuvintele distincte de lungime maximă:\n" +
      "\n".join(rez.split()))

```

O soluție mai eficientă se poate obține folosind în locul șirului rez o structură de date de tip mulțime.

4. Considerăm un șir de caractere reprezentând un titlu în limba engleză. Să se formateze titlul respectiv conform următoarelor reguli:

- primul și ultimul cuvânt se vor scrie întotdeauna cu majusculă;
- toate cuvintele formate din cel puțin 5 litere se vor scrie cu majusculă;
- toate cuvintele formate din cel mult 4 litere care nu fac parte dintr-o listă de cuvinte exceptate (articole, prepoziții, conjuncții etc.) se vor scrie cu majusculă.

Titlul poate să conțină și alte caractere în afară de litere mari sau mici (i.e., cifre sau semne de punctuație), iar cuvintele sunt despărțite între ele printr-un spațiu sau printr-un semn de punctuație și un spațiu. Lista cuvintelor exceptate poate conține, de exemplu, cuvintele "a", "an", "by", "on", "in", "at", "to", "for", "ago", "the", "past", "over", "into" și "onto".

Exemple:

- "where are WE GOING to?" -> "Where Are We Going To?"
- "GOING To California By My Car" -> "Going to California by My Car"
- "by my SIDE, at The SeaSide" -> "By My Side, at the Seaside"
- "walking over the rainbow" -> "Walking over the Rainbow"

Rezolvare:

Pentru formatarea unui titlu conform celor reguli de mai sus, vom identifica fiecare cuvânt al său (mai puțin primul) în funcție de spațiile care îl delimitează. Astfel, ultimul cuvânt va începe după ultimul spațiu din titlu (dacă acesta este format din cel puțin două cuvinte), iar orice alt cuvânt va fi delimitat de două spații consecutive (ale căror poziții le vom reține în două variabile p și q). Cuvintele exceptate de la cazul c) le vom păstra într-un șir de caractere, încadrate între două spații fiecare (de ce?).

```

s = input("Titlul: ")

# eliminăm spațiile de la începutul și sfârșitul șirului,
# după care transformăm toate literele în minuscule
s = s.strip().lower()

```

```

# transformăm prima literă în majusculă - cazul a)
s = s[0].upper() + s[1:]

# căutăm ultimul cuvânt din titlu (începe după ultimul spațiu)
# și îi transformăm prima literă în majusculă - cazul a)
p = s.rfind(" ")
if p != -1:
    s = s[:p+1] + s[p+1].upper() + s[p+2:]

exceptii = " a an by on in at to for ago the past over into onto "

# căutăm primul spațiu
p = s.find(" ")
while p != -1:
    # căutăm următorul spațiu
    q = s.find(" ", p + 1)
    if q != -1:
        # extragem în cuvântul curent și eliminăm eventualul semn
        # de punctuație de la sfârșitul său
        cuv = s[p + 1: q].strip(" ,. ; ? !")
        # transformăm în majusculă prima literă a cuvântului curent
        # dacă suntem într-unul dintre cazurile b) sau c)
        if len(cuv) >= 5 or (len(cuv) < 5 and
                           " " + cuv + " " not in exceptii):
            s = s[:p+1] + s[p+1].upper() + s[p+2:]
        p = q
    else:
        break

print("Titlul formatat: " + s)

```

Pentru formatare unui titlu în limba engleză sunt definite mai multe stiluri standard (APA, Chicago, NY Times etc.): <https://capitalizemytitle.com/>. Regulile de formatare prezentate în aceasta problemă sunt o variantă (simplificată) a celor mai des întâlnite reguli.

5. Se citește un șir de caractere s . Să se verifice dacă există un șir t , diferit de s , astfel încât șirul s să se poată obține prin concatenarea de un număr arbitrar de ori k a șirului t . De exemplu, pentru $s = "abcabc"$ avem $t = "abc"$ și $k = 2$.

Rezolvare:

Se observă faptul că șirul t poate fi doar un prefix al șirului s și lungimea sa trebuie să fie un divizor d al lungimii n a șirului s . De asemenea, divizorul d trebuie să fie cuprins între 1 și $\text{len}(s)//2$, deoarece șirul t trebuie să fie diferit de șirul s . În consecință, pentru fiecare divizor d al lui n , verificăm dacă prin concatenarea prefixului lui s de lungime d (i.e., subșirul $s[:d]$) cu el însuși de $n//d$ ori obținem șirul s sau nu:

```

s = input("Șirul s: ")
n = len(s)

```

```

for d in range(1, n//2 + 1):
    if n % d == 0:
        t = s[:d] * (n//d)
        if t == s:
            print("t = ", s[:d], "\nk = ", n//d)
            break
    else:
        print("Imposibil!")

```

Programul de mai sus va afișa șirul t de lungime minimă (de exemplu, pentru șirul $s = \text{"abababab"}$ va afișa $t = \text{"ab"}$ și $k = 4$), deoarece posibili divizori d ai lui n sunt considerați în ordine crescătoare. Pentru a afișa șirul t de lungime maximă (de exemplu, pentru șirul $s = \text{"abababab"}$ să se afișeze $t = \text{"abab"}$ și $k = 2$), vom considera posibili divizori d în ordine descrescătoare, respectiv vom înlocui `range(1, n//2 + 1)` cu `range(n//2, 0, -1)`.

6. Într-o propoziție, Ana a sintetizat informațiile despre cumpărăturile pe care le-a efectuat într-o anumită zi, pentru fiecare produs cumpărat ea precizând cantitatea și prețul unitar (e.g., "Astăzi am cumpărat 5 kg de mere cu 2.30 RON kilogramul și 2 pâini a câte 5 RON bucata."). Scrieți un program care să afișeze totalul cheltuielilor Anei din ziua respectivă.

Rezolvare:

Împărțim propoziția în cuvinte, considerând doar spațiul ca separator, și dacă avem un număr în cuvântul curent (i.e., dacă prin conversia cuvântului curent în număr nu apare o eroare) verificăm dacă o variabilă `cantitate` conține deja o valoare (i.e., nu conține valoarea `None`), reprezentând cantitatea dintr-un produs, sau nu. În caz afirmativ, numărul curent este păstrat în variabila `cantitate`, altfel el reprezintă un preț unitar, deci îl stocăm în variabila `pret` și adăugăm la totalul cheltuielilor valoarea produsului dintre cantitate și prețul unitar, după care setăm variabilele `cantitate` și `pret` la valoarea `None`.

```

s = input("Propozitia: ")
cuvinte = s.split(" ")

cantitate = pret = None
total = 0

for cuv in cuvinte:
    try:
        aux = float(cuv)
    except ValueError:
        continue

    if cantitate is None:
        cantitate = aux

```



```

else:
    pret = aux
    total += cantitate * pret
    cantitate = pret = None

print(f"Totalul cheltuielilor: {total}")

```

Probleme propuse

1. Se citește un cuvânt w și un șir format din n cuvinte. Să se afișeze toate cuvintele distincte care au aceeași lungime cu w . De exemplu, pentru $w = \text{"mere"}$ și cuvintele "pere", "teste" și "mure" (deci $n = 3$), trebuie să fie afișate cuvintele "pere" și "mure".
2. Se citește o propoziție sau o frază în care cuvintele sunt despărțite între ele prin spații și semnele de punctuație uzuale. Să se afișeze numărul cuvintelor distincte din propoziția sau fraza dată. De exemplu, în propoziția "Ana are prune și gutui verzi, dar mai multe prune decât gutui!" sunt 10 cuvinte distincte (cuvintele "prune" și "gutui" se iau în considerare o singură dată, deși apar de câte două ori în propoziție).
3. Să se afișeze numărul de litere mici, litere mari și semne de punctuație dintr-o propoziție citită de la tastatură.
4. Implementați cifrul lui Cezar (https://ro.wikipedia.org/wiki/Cifrul_Cezar) pentru a cripta/decripta o propoziție citită de la tastatură.

SEMINAR 3

Colecții de date și fișiere text

1. În fișierul text *numar_lipsa.txt* se găsesc pe prima linie $n - 1$ numere naturale distincte dintre primele n numere naturale nenule. Să se afișeze numărul lipsă. De exemplu, dacă prima linie din fișier este 2 1 5 4, numărul lipsă este 3.

Rezolvare:

În primul seminar am prezentat o modalitate de rezolvare a acestei probleme folosind operatori pe biți (vezi problema 7), având complexitatea computațională $\mathcal{O}(n)$ și spațiul de memorie utilizat $\mathcal{O}(n)$.

În continuare, vom prezenta alte variante de rezolvare a acestei probleme, folosind diverse colecții de date.

- a) calculăm diferența dintre suma t a primelor n numere naturale și suma s a numerelor din fișier:

```
try:
    f = open("numar_lipsa.txt")
    numere = [int(x) for x in f.readline().split()]
    s = sum(numere)
    n = len(numere) + 1
    t = n * (n+1) // 2
    print("Numarul lipsa este:", t - s)
    f.close()
except FileNotFoundError:
    print("Fișier inexistent!")
```

Complexitatea computațională a acestei soluții este $\mathcal{O}(n)$, iar spațiul de memorie utilizat este tot $\mathcal{O}(n)$.

- b) citim numerele din fișier într-o listă și apoi căutăm în ea fiecare număr de la 1 la n :

```
try:
    f = open("numar_lipsa.txt")
    numere = [int(x) for x in f.readline().split()]
    n = len(numere) + 1
    for x in range(1, n+1):
        if x not in numere:
            print("Numarul lipsa este:", x)
            break
    f.close()
except FileNotFoundError:
    print("Fișier inexistent!")
```

Complexitatea computațională a acestei soluții este $\mathcal{O}(n^2)$, iar spațiul de memorie utilizat este tot $\mathcal{O}(n)$.

- c) citim numerele din fișier într-o listă L pe care o sortăm crescător și apoi căutăm în ea primul indice pentru care $i + 1 \neq L[i]$, iar dacă nu există niciun indice având această proprietate, înseamnă că lipsește numărul n :

```
try:
    f = open("numar_lipsa.txt")
    L = [int(x) for x in f.readline().split()]
    L.sort()
    n = len(L) + 1
    for i in range(n-1):
        if L[i] != i+1:
            print("Numarul lipsa este:", i+1)
            break
    else:
        print("Numarul lipsa este:", n)
    f.close()
except FileNotFoundError:
    print("Fișier inexistent!")
```

Complexitatea computațională a acestei soluții este $\mathcal{O}(n \log_2 n)$, iar spațiul de memorie utilizat este tot $\mathcal{O}(n)$.

- d) determinăm numărului lipsă ca diferență dintre mulțimea T formată din primele n numere naturale nenule și mulțimea M a numerelor din fișier:

```
try:
    f = open("numar_lipsa.txt")
    M = {int(x) for x in f.readline().split()}
    n = len(M) + 1
    T = {x for x in range(1, n+1)}
    print("Numarul lipsa este:", *(T - M))
    f.close()
except FileNotFoundError:
    print("Fișier inexistent!")
```

Complexitatea computațională medie a acestei soluții este $\mathcal{O}(n)$, iar spațiu de memorie utilizat este $\mathcal{O}(n)$.

- e) marcăm cele $n - 1$ numere din fișier într-un dicționar având cheile $1, 2, \dots, n$ inițializate cu `False` și apoi căutăm primul număr nemarcat:

```
try:
    f = open("numar_lipsa.txt")
    L = [int(x) for x in f.readline().split()]
    n = len(L) + 1
    M = {x: False for x in range(1, n+1)}
```

```

for x in L:
    M[x] = True
for x in M:
    if M[x] == False:
        print("Numarul lipsa este:", x)
        break
f.close()
except FileNotFoundError:
    print("Fișier inexistent!")

```

Complexitatea computațională medie a acestei soluții este $O(n)$, iar spațiu de memorie utilizat este $O(n)$.

2. Fișierul text *numere.txt* conține numere naturale despărțite prin spații și scrise pe mai multe linii. Să se scrie în fișierul text *numere_comune.txt* numerele care apar pe toate liniile din fișier.

Exemplu: Dacă fișierul *numere.txt* are următorul conținut:

```

2 1 5 1 3
1 4 2 2
2 1 1 6 8

```

atunci fișierul *numere_comune.txt* trebuie să conțină numerele **1 2**, nu neapărat în această ordine.

Rezolvare:

Folosim o variabilă *C* de tip mulțime (set) pentru a păstra numerele comune tuturor liniilor din fișier. Inițial, mulțimea *C* va conține numerele de pe prima linie din fișier, după care va fi actualizată pentru fiecare linie nouă citită, respectiv va fi intersectată cu mulțimea *M* a numerelor de pe linia curentă:

```

f = open("numere.txt")

s = f.readline()
# C = mulțimea numerelor comune tuturor liniilor,
# inițial conținând numerele de pe prima linie
C = set([int(x) for x in s.split()])

# parcurgem restul liniilor din fișier
while s != "":
    s = f.readline()
    if s != "":
        # mulțimea numerelor de pe linia curentă
        M = set([int(x) for x in s.split()])
        # actualizăm mulțimea C, intersectând-o cu M
        C = C & M      # echivalent cu C = C.intersection(M)

f.close()

```

```
f = open("numere_comune.txt", "w")

if len(C) != 0:
    f.write("Numerele comune tuturor liniilor:\n" +
           " ".join([str(x) for x in C]))
else:
    f.write("Nu exista numere comune tuturor liniilor!")

f.close()
```

Modificați programul de mai sus astfel încât să întreruptă forțat citirea din fișier dacă mulțimea C a numerelor comune tuturor liniilor devine vidă. Care este complexitatea computațională a acestui program?

3. Fișierul text *exemplu.txt* conține un text pe mai multe linii, cuvintele fiind despărțite între ele prin spații și semnele de punctuație uzuale. Implementați un program care să scrie în fișierul text *grupe_cuvinte.txt* cuvintele distincte (fără duplicate) din fișierul dat, grupate după mulțimile de litere din care sunt formate (nu se va face distincție între literele mari și cele mici). Grupele se vor scrie în fișier în ordinea descrescătoare a numărului de elemente din mulțimile literelor, în fiecare grupă literele se vor scrie în ordine crescătoare, iar în fiecare grupă cuvintele vor fi scrise în ordine alfabetică.

Exemplu:

exemplu.txt	grupe_cuvinte.txt
Langa o cabana in munti, stand supus pe o banca si tastand, un bacan numit Pepe a spus un banc acum un minut.	Literele adnst: stand, tastand Literale imntu: minut, munti, numit Literale abcn: bacan, banc, banca, cabana Literale acmu: acum Literale agln: langa Literale psu: spus, supus Literale ep: pe, pepe Literale in: in Literale is: si Literale nu: un Literale a: a Literale o: o

Rezolvare:

Pentru a determina în mod eficient grupurile de cuvinte care sunt formate din aceleași litere (fără a fi neapărat anagrame!), vom utiliza un dicționar cu perechi de forma *mulțime de litere : lista cuvintelor formate din literele respective*. Deoarece cheile unui dicționar trebuie să fie imutabile, acestea vor fi obiecte de tipul *frozenset* (mulțimi imutabile). Pentru a putea sorta informațiile din dicționar în modul precizat în enunț, vom "exporta" perechile sale de forma *mulțime de litere : lista cuvintelor* într-o listă de tuple-uri de tipul (*șir, listă*) în care *șir* va fi un șir de caractere format dintr-o mulțime de litere sortate alfabetic, iar *listă* va conține, sortate alfabetic, cuvintele formate din literele memorate în *șir*.

```

try:
    f = open("exemplu.txt")
    text = f.read().lower()
    f.close()
except FileNotFoundError:
    print("Fisier inexistent!")
    exit(0)

for c in ",.;;?!":
    text = text.replace(c, " ")

d = {}
for cuvant in text.split():
    litere = frozenset(cuvant)
    if litere in d:
        d[litere].add(cuvant)
    else:
        d[litere] = set([cuvant])

aux = ["".join(sorted(k)), sorted(v)) for (k,v) in d.items()]

def cheieLitere(t):
    return -len(t[0]), t[0]

aux = sorted(aux, key=cheieLitere)

f = open("grupe_cuvinte.txt", "w")
for p in aux:
    f.write("Literele " + p[0] + ": " + ", ".join(p[1]) + "\n")
f.close()

```

Observați faptul că tuplurile din lista `aux` au fost sortate descrescător după lungimile șirurilor corespunzătoare mulțimilor de litere, iar în cazul unor lungimi egale au fost sortate alfabetic, folosind cheia definită în funcția `cheieLitere`!

4. Scrieți o funcție care să se verifice dacă două șiruri de caractere formate doar din litere mici sunt anagrame sau nu. Două șiruri sunt anagrame dacă sunt formate din aceleași litere, dar așezate în altă ordine (sau, echivalent, unul dintre șiruri se poate obține din celălalt printr-o permutare a caracterelor sale). De exemplu, șirurile `emerit` și `treime` sunt anagrame, dar șirurile `emerit` și `treimi` nu sunt!

Rezolvare:

Înainte de a prezenta mai multe metode de rezolvare a acestei probleme, utilizând diverse colecții de date, observăm faptul că două șiruri de caractere `s` și `t` pot fi anagrame doar dacă au aceeași lungime, deci putem considera faptul că $\text{len}(s) = \text{len}(t) = n$.

Varianta 1:

Căutăm fiecare literă din șirul s în șirul t și, dacă o găsim, atunci o ștergem din șirul t , altfel cele două șiruri nu sunt anagrame:

```
def anagrama(s, t):
    if len(s) != len(t):
        return False

    for litera in s:
        poz = t.find(litera)
        if poz == -1:
            return False
        t = t[:poz] + t[poz+1:]

    return True
```

O altă variantă de ștergere a unei anumite litere se poate implementa folosind metoda `replace`, setând valoarea parametrului opțional `count` la 1 pentru a șterge o singură apariție a literei respective:

```
def anagrama(s, t):
    if len(s) != len(t):
        return False

    for litera in s:
        if litera not in t:
            return False
        t = t.replace(litera, "", 1)

    return True
```

Evident, complexitatea computațională a ambelor funcții este $\mathcal{O}(n^2)$.

Varianta 2:

Verificăm dacă fiecare literă distinctă din șirul s apare de același număr de ori și în șirul s și în șirul t :

```
def anagrama(s, t):
    if set(s) != set(t):
        return False

    for litera in set(s):
        if s.count(litera) != t.count(litera):
            return False

    return True
```

Deoarece metoda `count` are complexitatea $\mathcal{O}(n)$, funcția de mai sus are complexitatea computațională maximă egală cu $\mathcal{O}(n^2)$. Observați faptul că am înlocuit testarea egalității lungimilor șirurilor `s` și `t`, care are complexitatea $\mathcal{O}(1)$, cu testarea egalității mulțimilor literelor celor două cuvinte, care are complexitatea maximă $\mathcal{O}(n)$! Noua condiție este mai puternică decât precedenta (de exemplu, va elimina direct cazul `s = "aabc"` și `t = "aabb"`), dar, totuși, este doar o condiție necesară, fără a fi și suficientă (de exemplu, șirurile `s = "aaab"` și `t = "abbb"` au aceeași mulțime a literelor, i.e. `{'a', 'b'}`, dar nu sunt anagrame deoarece frecvențele celor două litere sunt diferite)!

Varianta 3:

Construim pentru fiecare șir câte un dicționar cu perechi *literă distinctă : frecvență* (de exemplu, pentru șirul `s = "emerit"` dicționarul asociat va fi `{'e': 2, 'r': 1, 't': 1, 'i': 1, 'm': 1}`) și verificăm dacă dicționarele sunt egale sau nu:

```
def anagram(s, t):
    ds = {litera: s.count(litera) for litera in set(s)}
    dt = {litera: t.count(litera) for litera in set(t)}

    return ds == dt
```

Deoarece metoda `count` are complexitatea $\mathcal{O}(n)$, funcția de mai sus are complexitatea computațională maximă egală cu $\mathcal{O}(n^2)$. Observați faptul că nu am mai testat nici egalitatea lungimilor șirurilor `s` și `t` și nici egalitatea mulțimilor literelor celor două cuvinte (de ce?). Totuși, adăugarea uneia dintre cele două condiții ar putea îmbunătăți complexitatea computațională în mai multe cazuri!

Varianta 4:

Sortăm crescător listele formate din literele celor două șiruri și verificăm dacă listele sunt egale sau nu:

```
def anagram(s, t):
    return sorted(s) == sorted(t)
```

Deoarece metoda `sorted` are complexitatea $\mathcal{O}(n \log_2 n)$, funcția de mai sus are complexitatea computațională egală cu $\mathcal{O}(n \log_2 n)$. Observați faptul că, nici în această variantă, nu am mai testat nici egalitatea lungimilor șirurilor `s` și `t` și nici egalitatea mulțimilor literelor celor două cuvinte!

Varianta 5:

Construim pentru fiecare șir câte un dicționar cu perechi *literă distinctă : frecvență* (de exemplu, pentru șirul `s = "emerit"` dicționarul asociat va fi `{'e': 2, 'r': 1, 't': 1, 'i': 1, 'm': 1}`), dar fără a utiliza metoda `count`! Astfel, dicționarul asociat unui șir îl vom construi verificând, pentru fiecare literă `a` sa, dacă apare sau nu în dicționar. În caz afirmativ, vom crește frecvența literei respective cu 1, altfel vom adăuga litera în dicționar cu frecvența 1:

```
def anagram(s, t):
    ds = {}
    for litera in s:
```



```

    if litera in ds:
        ds[litera] = ds[litera] + 1
    else:
        ds[litera] = 1

dt = {}
for litera in t:
    if litera in dt:
        dt[litera] = dt[litera] + 1
    else:
        dt[litera] = 1

return ds == dt

```

Deoarece testarea apartenenței unei chei la un dicționar și accesarea valorii asociate unei chei se realizează cu complexitatea $\mathcal{O}(1)$, dacă se utilizează o funcție de dispersie bună (i.e., funcția generează puține coliziuni), rezultă faptul că funcția de mai sus are complexitatea computațională egală cu $\mathcal{O}(n)$.

Dicționarul asociat unui șir mai poate fi construit inițializând dicționarul cu perechi de forma *literă distinctă* : 0 și apoi crescând cu 1 frecvența fiecărei litere din șir:

```

def anagrama(s, t):
    ds = {litera: 0 for litera in set(s)}
    for litera in s:
        ds[litera] = ds[litera] + 1

    dt = {litera: 0 for litera in set(t)}
    for litera in t:
        dt[litera] = dt[litera] + 1

    return ds == dt

```

Evident, complexitatea computațională a acestei funcții este egală tot cu $\mathcal{O}(n)$!

Varianta 6:

Construim pentru fiecare șir câte o listă conținând frecvențele fiecăreia dintre cele 26 de litere mici (*vector de frecvențe*) și verificăm dacă ele sunt egale sau nu:

```

def anagrama(s, t):
    fs = [0] * 26
    ft = [0] * 26
    for litera in s:
        fs[ord(litera) - ord("a")] += 1
    for litera in t:
        ft[ord(litera) - ord("a")] += 1

    return fs == ft

```

Complexitatea acestei funcții este tot $\mathcal{O}(n)$, dar, spre deosebire de varianta cu dicționare, este stabilă (nu depinde de performanțele unei funcții de dispersie)!

Probleme propuse

1. Scrieți o funcție care verifică dacă două șiruri de caractere sunt anagrame sau nu utilizând un singur vector de frecvențe.

2. Scrieți un program care verifică dacă două șiruri de caractere s și t formate doar din litere mici sunt anagrame și, în caz afirmativ, afișează o permutare p prin care șirul t se obține din șirul s , respectiv o permutare q prin care șirul s se obține din șirul t . De exemplu, pentru șirurile $s = \text{"emerit"}$ și $t = \text{"treime"}$ o permutare p prin care se poate obține șirul t din șirul s este $p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 6 & 2 & 4 & 1 \end{pmatrix}$, deoarece prima literă din șirul s (litera 'e') trebuie să fie a treia literă din șirul t , a doua literă din șirul s (litera 'm') trebuie să fie a cincea literă din șirul t , ș.a.m.d.

Indicație de rezolvare:

Dacă șirurile s și t sunt anagrame, pentru a construi permutarea p , vom căuta poziția pe care apare în t fiecare literă din s , după care vom înlocui litera respectivă din t cu un spațiu (sau orice alt caracter care nu este literă mică). De ce nu putem să ștergem litera din t ? Folosiți dicționare pentru a construi cele două permutări!

3. Fișierul text *exemplu.txt* conține un text pe mai multe linii, cuvintele fiind despărțite între ele prin spații și semnele de punctuație uzuale. Implementați un program care să scrie în fișierul text *grupe_cuvinte.txt* cuvintele distincte (fără duplicate) din fișierul dat, grupate după lungimile lor. Grupele se vor scrie în fișier în ordinea descrescătoare a lungimilor cuvintelor, iar în fiecare grupă cuvintele vor fi scrise în ordine alfabetică.

Exemplu:

exemplu.txt	grupe_cuvinte.txt
Ana are multe pere, mere rosii, si mai multe prune!!!	Lungime 5: multe, prune, rosii Lungime 4: mere, pere Lungime 3: ana, are, mai Lungime 2: si

4. Fișierul text *numere.txt* conține numere naturale pe mai multe linii, numerele fiind despărțite între ele prin spații. Scrieți un program care să afișeze cel mai mare și cel mai mic număr care se poate obține din toate cifrele tuturor numerelor din fișier. Rezolvați problema în cel puțin 3 moduri diferite, respectiv folosind șiruri de caractere, liste, mulțimi și dicționare!

Exemplu:

numere.txt	ecran
215 13 1009 2377 1024 9099 2020 23	Minim: 10000009999775433322222111 Maxim: 99997754333222221111000000

SEMINAR 4

Funcții

1.

- a) Scrieți o funcție care returnează o matrice triunghiulară de dimensiune n , având forma următoare:
- prima coloană conține numerele $1, 2, 3, \dots, n$;
 - ultima linie conține numerele $n, n - 1, \dots, 2, 1$;
 - restul elementelor aflate în triunghiul de sub diagonala principală se calculează ca sumă a elementelor vecine de la vest, sud și sud-vest.
- b) Scrieți o funcție care afișează o matrice triunghiulară, memorată sub forma unei liste de liste, în formă matriceală, cu elementele de pe fiecare coloană aliniate la dreapta.

Exemplu: pentru $n = 4$ matricea cerută este $M = [[1], [2, 15], [3, 10, 15], [4, 3, 2, 1]]$ sau, afișată în formă matriceală:

$$M = \begin{pmatrix} 1 & & & \\ 2 & 15 & & \\ 3 & 10 & 15 & \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

Rezolvare:

Vom construi matricea triunghiulară de forma cerută pas cu pas, astfel:

- vom construi matricea fără ultima linie (exemplul este pentru $n = 4$) și elementele care nu se află pe prima coloană egale cu 0:

$$M = \begin{pmatrix} 1 & & & \\ 2 & 0 & & \\ 3 & 0 & 0 & \end{pmatrix}$$

Se observă faptul că fiecare linie i , unde $i \in \{0, 1, \dots, n - 2\}$, este o listă de forma $\left[i + 1, \underbrace{0, \dots, 0}_{\text{de } i \text{ ori}} \right]$.

- vom adăuga la matricea construită anterior ultima linie, adică lista $[n, n - 1, \dots, 1]$:

$$M = \begin{pmatrix} 1 & & & \\ 2 & 0 & & \\ 3 & 0 & 0 & \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

- pentru fiecare linie i , începând cu penultima ($i = n - 2$) și terminând cu a doua ($i = 1$), calculăm elementele aflate pe coloanele $\{1, 2, \dots, i\}$ ca sumă a elementelor vecine de la vest, sud și sud-vest.

Ținând cont de observațiile de mai sus, funcția cerută este următoarea:

```
def matrice(n):
    M = [[i] + [0]*(i-1) for i in range(1, n)]
    M.append([i for i in range(n, 0, -1)])

    for i in range(n-2, 0, -1):
        for j in range(1, i+1):
            M[i][j] = M[i][j-1] + M[i+1][j-1] + M[i+1][j]

    return M
```

Pentru a afișa o matrice triunghiulară, memorată sub forma unei liste de liste, în formă matriceală, cu elementele de pe fiecare coloană aliniate la dreapta, vom calcula mai întâi numărul maxim de cifre *ncmax* pe care îl are un element al matricei (de exemplu, pentru $n = 4$ vom obține $ncmax = 2$) și apoi vom afișa fiecare element al matricei aliniat la dreapta pe *ncmax* spații folosind metoda *rjust* pentru șiruri de caractere:

```
def afisare(M):
    ncmax = max([len(str(max(linie))) for linie in M])
    for linie in M:
        for elem in linie:
            print(str(elem).rjust(ncmax), end=' ')
        print()
```

Cum s-ar putea calcul mai eficient numărul maxim de cifre *ncmax* pe care îl are un element al matricei?

2. Scrieți o funcție și un generator care primesc ca parametrii un număr variabil de liste și o valoare *x* și furnizează toate listele care conțin valoarea *x*.

Rezolvare:

În cazul funcției, vom construi rezultatul sub forma unei liste *rez* care va conține listele în care apare valoarea *x*:

```
def cauta(x, *liste):
    rez = []
    for lcrt in liste:
        if x in lcrt:
            rez.append(lcrt)
    return rez

x = 7
r = cauta(x, [5, 1, 7, 3, 7], [2, 3], [-3, 7, 1])
```

```

if r == []:
    print("Valoarea", x, "nu se gaseste in nicio lista!\n")
else:
    print("Valoarea", x, "se gaseste in listele urmatoare:")
    print(*r, sep="\n")

```

În cazul generatorului, vom returna, pe rând, fiecare listă în care apare valoarea x:

```

def cauta(x, *liste):
    for lcrt in liste:
        if x in lcrt:
            yield lcrt

x = 7
r = cauta(x, [5, 1, 7, 3, 7], [2, 3], [-3, 7, 1])

lst = next(r, None)
if lst is None:
    print("Valoarea", x, "nu se gaseste in nicio lista!\n")
else:
    print("Valoarea", x, "se gaseste in listele urmatoare:")
    while lst is not None:
        print(lst)
        lst = next(r, None)

```

Observați modul în care am verificat faptul că generatorul nu a returnat nicio listă!

3. Pentru un student se cunosc următoarele informații: numele, grupa și o listă cu creditele obținute la toate examenele din anul respectiv.
 - a) Scrieți o funcție care să primească numele unui fișier de tip CSV care conține informații despre mai mulți studenți (i.e., informațiile despre un student se găsesc pe o linie și sunt despărțite între ele prin câte o virgulă) și furnizează o listă conținând informațiile despre studenți sub forma unei liste de tuple-uri (i.e., informațiile despre un student se vor păstra într-un tuplu de forma (*nume*, *grupa*, (*nota_1*, *nota_2*, ..., *nota_n*))).
 - b) Considerând o listă de studenți, scrieți o funcție care să adauge la informațiile despre un student situația sa școlară: promovat (True) sau nepromovat (False). Pentru a fi considerat promovat, un student trebuie să nu aibă nici un examen nepromovat (adică o valoare egală cu 0 în lista creditelor), iar suma creditelor obținute să fie mai mare sau egală decât un număr minim de credite dat.
 - c) Scrieți câte o funcție care să furnizeze cheia necesară sortării listei cu informații despre studenți în raport de fiecare dintre următoarele criterii:
 - crescător după grupă și în fiecare grupă în ordine alfabetică;
 - întâi studenții promovați, apoi cei nepromovați și în fiecare categorie în ordine alfabetică;

- descrescător după suma creditelor, iar în cazul unor sume egale în ordinea crescătoare a grupei și în ordine alfabetică în cadrul grupei;
- în ordinea crescătoare a grupelor, în cadrul fiecărei grupe mai întâi studenții promovați, iar apoi cei nepromovați, în fiecare categorie (promovat/nepromovat) în ordinea descrescătoare a sumei creditelor și, în cazul unor sume egale, în ordine alfabetică.

Rezolvare:

Pentru testarea celor 3 funcții cerute, vom considera următorul fișier de tip CSV:

studenti.csv

```
Popescu Ion Mihai,135,5,7,3,0,4
Popa Anca,131,5,7,3,3,5
Mihai Ana,135,7,3,3,2,4
Mihai Ana,132,7,0,3,0,0
Ionescu Clara Maria,131,3,3,3,3,3
Bobescu Iulia,135,5,5,5,3,3
```

Conținutul acestui fișier va fi încărcat într-o listă formată din tuple-uri cu următoarea structură:

```
lstud = [("Popescu Ion Mihai", 135, (5, 7, 3, 0, 4)),
         ("Popa Anca", 131, (5, 7, 3, 3, 5)),
         ("Mihai Ana", 135, (7, 3, 3, 2, 4)),
         ("Mihai Ana", 132, (7, 0, 3, 0, 0)),
         ("Ionescu Clara Maria", 131, (3, 3, 3, 3, 3)),
         ("Bobescu Iulia", 135, (5, 5, 5, 3, 3))]
```

În funcția pentru citirea informațiilor despre studenți vom parcurge fișierul text linie cu linie și vom împărți linia curentă în subșiruri considerând virgula ca separator:

```
def citireInformatiiStudenti(nume_fisier):
    fin = open(nume_fisier)

    studenti = []
    for linie in fin:
        aux = linie.split(",")
        student = (aux[0], int(aux[1]),
                  tuple(int(x) for x in aux[2:]))
        studenti.append(student)

    fin.close()

    return studenti
```

Pentru a calcula situația școlară a unui student, vom parcurge lista element cu element și vom verifica dacă studentul respectiv îndeplinește condițiile de promovare sau nu:

```
# ls = lista cu studenți
# minc = numărul minim de credite necesare
def situatie_scolara(ls, minc):
    for i in range(len(ls)):
        if 0 not in ls[i][2] and sum(ls[i][2]) >= minc:
            ls[i] += (True,)
        else:
            ls[i] += (False,)
```

Dacă vom testa funcția prin

```
situatie_scolara(lstud, 20)
print(*lstud, sep="\n")
```

se vor afișa următoarele informații:

```
('Popescu Ion Mihai', 135, (5, 7, 3, 0, 4), False)
('Popa Anca', 131, (5, 7, 3, 3, 5), True)
('Mihai Ana', 135, (7, 3, 3, 2, 4), False)
('Mihai Ana', 132, (7, 0, 3, 0, 0), False)
('Ionescu Clara Maria', 131, (3, 3, 3, 3, 3), False)
('Bobescu Iulia', 135, (5, 5, 5, 3, 3), True)
```

Atenție, următoare funcție este incorectă, respectiv nu va actualiza informațiile despre un student prin adăugarea la tuplul respectiv a valorii True sau False:

```
def situatie_scolara(ls, minc):
    for student in ls:
        if 0 not in student[2] and sum(student[2]) >= minc:
            student += (True,)
        else:
            student += (False,)
```

Prin parcurgerea unei colecții folosind instrucțiunea for nu se accesează direct un element al colecției, ci o copie a elementului respectiv (în acest caz, variabila student)! Pentru rezolvarea celei de-a doua cerințe, vom indica, în fiecare caz, funcția care va furniza cheia corespunzătoare și rezultatul sortării listei obținute după apelul situatie_scolara(lstud, 20) folosind cheia respectivă:

- crescător după grupă și în fiecare grupă în ordine alfabetică:

Funcția pentru cheie	Rezultatul sortării
<pre>def criteriu_1(t): return t[1], t[0] lstud.sort(key=criteriu_1) print(*lstud, sep="\n")</pre>	<pre>('Ionescu Clara Maria', 131, (3, 3, 3, 3, 3), False) ('Popa Anca', 131, (5, 7, 3, 3, 5), True) ('Mihai Ana', 132, (7, 0, 3, 0, 0), False) ('Bobescu Iulia', 135, (5, 5, 5, 3, 3), True) ('Mihai Ana', 135, (7, 3, 3, 2, 4), False) ('Popescu Ion Mihai', 135, (5, 7, 3, 0, 4), False)</pre>

- întâi studenții promovați, apoi cei nepromovați și în fiecare categorie în ordine alfabetică:

Funcția pentru cheie	Rezultatul sortării
<pre>def criteriu_2(t): return -t[3], t[0] lstud.sort(key=criteriu_2) print(*lstud, sep="\n")</pre>	<pre>('Bobescu Iulia', 135, (5, 5, 5, 3, 3), True) ('Popa Anca', 131, (5, 7, 3, 3, 5), True) ('Ionescu Clara Maria', 131, (3, 3, 3, 3, 3), False) ('Mihai Ana', 135, (7, 3, 3, 2, 4), False) ('Mihai Ana', 132, (7, 0, 3, 0, 0), False) ('Popescu Ion Mihai', 135, (5, 7, 3, 0, 4), False)</pre>

Observați faptul că am sortat descrescător după situația școlară, deoarece valoarea True este echivalentă cu 1, iar valoarea False cu 0!

- descrescător după suma creditelor, iar în cazul unor sume egale în ordinea crescătoare a grupei și în ordine alfabetică în cadrul grupei:

Funcția pentru cheie	Rezultatul sortării
<pre>def criteriu_3(t): return -sum(t[2]), \ t[1], t[0] lstud.sort(key=criteriu_3) print(*lstud, sep="\n")</pre>	<pre>('Popa Anca', 131, (5, 7, 3, 3, 5), True) ('Bobescu Iulia', 135, (5, 5, 5, 3, 3), True) ('Mihai Ana', 135, (7, 3, 3, 2, 4), False) ('Popescu Ion Mihai', 135, (5, 7, 3, 0, 4), False) ('Ionescu Clara Maria', 131, (3, 3, 3, 3, 3), False) ('Mihai Ana', 132, (7, 0, 3, 0, 0), False)</pre>

- în ordinea crescătoare a grupelor, în cadrul fiecărei grupe mai întâi studenții promovați, iar apoi cei nepromovați, în fiecare categorie (promovat/nepromovat) în ordinea descrescătoare a sumei creditelor și, în cazul unor sume egale, în ordine alfabetică:

Funcția pentru cheie	Rezultatul sortării
<pre>def criteriu_4(t): return t[1], -t[3], \ -sum(t[2]), t[0] lstud.sort(key=criteriu_4) print(*lstud, sep="\n")</pre>	<pre>('Popa Anca', 131, (5, 7, 3, 3, 5), True) ('Ionescu Clara Maria', 131, (3, 3, 3, 3, 3), False) ('Mihai Ana', 132, (7, 0, 3, 0, 0), False) ('Bobescu Iulia', 135, (5, 5, 5, 3, 3), True) ('Mihai Ana', 135, (7, 3, 3, 2, 4), False) ('Popescu Ion Mihai', 135, (5, 7, 3, 0, 4), False)</pre>

Observați faptul că în fiecare cheie (de fapt, un tuplu) am precizat componentele în ordinea cerută de criteriile de sortare, deoarece compararea a două tupluri se realizează lexicografic. De asemenea, o sortare descrescătoare în raport de o componentă numerică a cheii se poate realiza prin schimbarea semnului său.

4.

- Scriveți o funcție generică care să furnizeze numărul elementelor dintr-o colecție iterabilă care au o anumită proprietate, implementată sub forma unei funcții cu un singur parametru care returnează True dacă acesta are proprietatea cerută sau False în caz contrar.
- Scriveți câte o funcție care să implementeze proprietatea necesară pentru ca funcția de numărare să furnizeze:
 - numărul valorilor pare dintr-o listă/tuplu de numere întregi;
 - numărul vocalelor dintr-un șir de caractere;
 - numărul perechilor (x, y) cu proprietatea că $x = y$ dintr-o listă de tupluri;
 - numărul șirurilor de lungime k dintr-o listă de șiruri de caractere;

- numărul valorilor x dintr-o listă de numere întregi pentru care $\text{cmmdc}(x, y) = t$, unde y și t sunt date.

Rezolvare:

Funcția generică de numărare implementează algoritmul standard de numărare, respectiv se incrementează un contor dacă elementul curent al colecției verifică proprietatea cerută:

```
def numarare(colectie, proprietate):
    contor = 0
    for element in colectie:
        if proprietate(element) == True:
            contor = contor + 1
    return contor
```

Pentru rezolvarea celei de-a doua cerințe, vom indica, în fiecare caz, funcția care va implementa proprietatea cerută și un program de test:

- numărul valorilor pare dintr-o listă de numere întregi:

```
def esteNrPar(numar):
    return numar % 2 == 0

L = [15, 21, 700, 132, -8, 19, -10, 1, 10, 5, 133]

np = numarare(L, esteNrPar)
print("Numar valori pare:", np)          # Numar valori pare: 5
```

- numărul vocalelor dintr-un șir de caractere:

```
def esteVocala(litera):
    return litera in "aeiouAEIOU"

cuvant = "Exemplificare"
nv = numarare(cuvant, esteVocala)
print("Numar vocale:", nv)              # Numar vocale: 6
```

- numărul perechilor (x, y) cu proprietatea că $x = y$ dintr-o listă de tuple:

```
def areComponenteEgale(pereche):
    return pereche[0] == pereche[1]

n = 5
# L = {1, 2, ..., n} X {1, 2, ..., n}
L = [(i, j) for i in range(1, n+1) for j in range(1, n+1)]
print(L)

np = numarare(L, areComponenteEgale)
print("Numarul perechilor cu componente egale:", np)
```

Pentru un număr n dat, ce valoare va fi afișată întotdeauna în exemplul de mai sus?

- numărul șirurilor de lungime k dintr-o listă de șiruri de caractere, unde k este dat:

```
def verificareLungimeSir(k):
    def aux(sir):
        return len(sir) == k
    return aux

L = "Ana are mere si pere mai multe mere decat pere".split()
lmax = max([len(cuv) for cuv in L])
print(L)

for k in range(1, lmax+1):
    ncuv = numarare(L, verificareLungimeSir(k))
    print("Numarul de cuvinte cu lungimea", k, "este", ncuv)
```

Deoarece funcția care implementează proprietatea pe care trebuie să o îndeplinească un element al colecției pentru a fi numărat trebuie să aibă un singur parametru (i.e., elementul respectiv), nu putem să utilizăm următoarea funcție "naturală":

```
def verificareLungimeSir(sir, k):
    return len(sir) == k
```

Totuși, deoarece se va folosi aceeași valoare k pentru toate șirurile din listă în momentul în care vom apela funcția `numarare`, am utilizat o funcție imbricată în care am testat condiția `len(sir) == k`, eliminând astfel parametrul k !

- numărul valorilor x dintr-o listă de numere întregi pentru care $\text{cmmdc}(x, y) = t$, unde y și t sunt date:

```
def verificareCMMDC(y, t):
    def cmmdc(a, b):
        if b == 0:
            return a
        return cmmdc(b, a % b)

    def aux(x):
        return cmmdc(x, y) == t

    return aux

L = [10, 142, 24, 17, 18, 100, 13, 15]
print(numarare(L, verificareCMMDC(4, 2))) # 3
```

În exemplul de mai sus se va afișa valoarea 3, deoarece $y = 4$ și $t = 2$, iar în lista L sunt 3 valori x pentru care $\text{cmmdc}(x, 4) = 2$, respectiv 10, 142 și 18.

Probleme propuse

1. Scrieți o funcție care returnează o matrice pătratică de dimensiune n , având forma următoare:

- pe ultima linie și ultima coloană toate elementele sunt egale cu 1;
- restul elementelor se calculează ca sumă a elementelor vecine de la est și sud.

Exemplu: pentru $n = 4$ matricea M cerută este:

$$M = \begin{pmatrix} 10 & 10 & 4 & 1 \\ 10 & 6 & 3 & 1 \\ 4 & 3 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

2.

a) Scrieți o funcție generică care să furnizeze o listă cu pozițiile pe care apar într-o colecție iterabilă elementele care au o anumită proprietate, implementată sub forma unei funcții cu un singur parametru care returnează `True` dacă acesta are proprietatea cerută sau `False` în caz contrar.

b) Scrieți câte o funcție care să implementeze proprietatea necesară pentru ca funcția generică să furnizeze:

- pozițiile valorilor strict pozitive dintr-un tuplu de numere întregi;
- pozițiile semnelor de punctuație dintr-un șir de caractere;
- pozițiile cuvintelor care sunt anagrame ale unui cuvânt s dintr-o listă de cuvinte;
- pozițiile numerelor naturale x dintr-o listă care sunt formate din n cifre și au suma cifrelor egală cu s .

3.

a) Pentru o carte se cunosc următoarele informații: titlul, autorii, anul apariției și prețul. Considerând o listă de cărți, scrieți o funcție care să ieftinească cu 20% cărțile apărute înainte de anul 2000.

b) Scrieți câte o funcție care să furnizeze cheia necesară sortării listei de cărți în raport de fiecare dintre următoarele criterii:

- descrescător după anul apariției și în fiecare an în ordinea alfabetică a titlurilor;
- în ordinea crescătoare a numărului de autori și în fiecare categorie în ordinea descrescătoare a prețurilor;
- în ordinea alfabetică a numelui primului autor (se consideră faptul că orice autor este dat printr-un șir de forma "Prenume Nume"), apoi în ordinea alfabetică a prenumelui primului autor, apoi în ordinea alfabetică a titlului și, respectiv, în ordinea crescătoare a anului apariției.

4. Scrieți o funcție și un generator care primesc ca parametrii un număr variabil de șiruri de caractere și un număr natural nenul n și furnizează toate șirurile care au lungimea n .

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 $\left\lceil \frac{n}{2} \right\rceil + 1$ 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 secvențe cu lungimea strict mai mare decât $n//2$ are complexitatea maximă $\mathcal{O}(n)$, deci această funcție va avea complexitatea maximă $\mathcal{O}(n \log_2 n)$.

A treia variantă de rezolvare se obține observând faptul că în lista voturilor sortată crescător singurul posibil câștigător este elementul din mijlocul listei, pentru că orice secvență formată din cel puțin $n//2+1$ elemente egale trebuie să conțină și elementul de pe poziția $n//2$:

```

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 și această funcție va avea complexitatea maximă tot $\mathcal{O}(n \log_2 n)$.

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

```

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 computațională a acestei funcții este $\mathcal{O}(n)$, deci este optimă, dar are dezavantajul de a utiliza memorie suplimentară (dicționarul). Atenție, dicționarul `nr_voturi` ar putea fi creat și prin `nr_voturi = {x: voturi.count(x) for x in set(voturi)}`, dar complexitatea acestei operații ar fi $\mathcal{O}(n^2)$, deci complexitatea funcției ar crește 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/mjrtty.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 $L[i] \geq S/2$.

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 $L[i] \geq S//2$.

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 $x \geq S//2$ 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 $dr = \text{len}(L) - 1$), 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

```

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

3. Problema mulțimii de acoperire

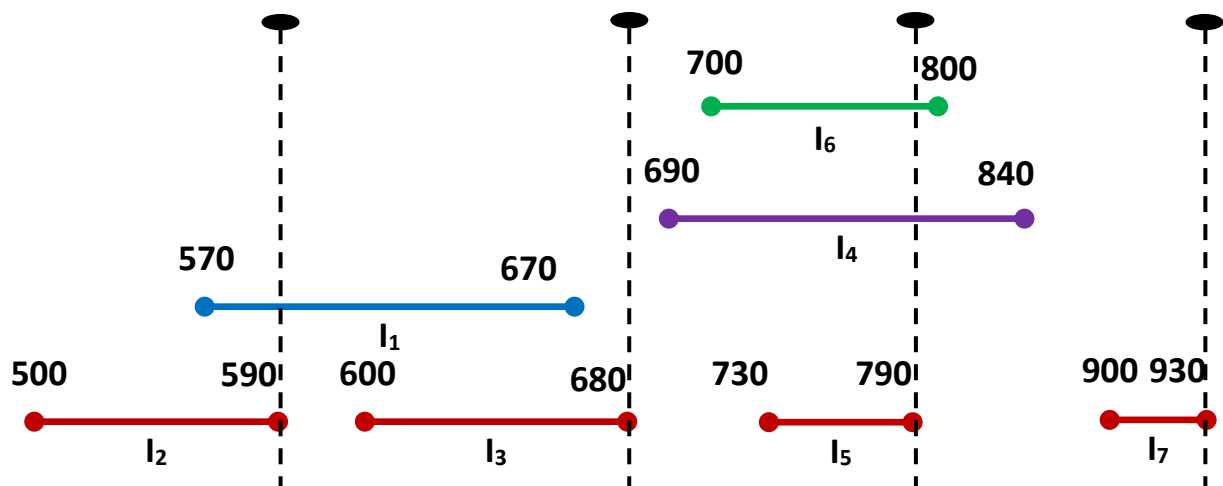
Fie n intervale închise $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$. Să se determine o mulțime M cu număr minim de elemente astfel încât $\forall k \in \overline{1, n}, \exists x \in M$ astfel încât $x \in I_k = [a_k, b_k]$. Mulțimea M se numește *mulțime de acoperire* a șirului de intervale respectiv.

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 și 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 cui pe care trebuie să le batem astfel încât în fiecare scândură să fie bătut cel puțin 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 extremităților 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 $\mathcal{O}(n \log_2 n)$. Observați 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 niște spectacole/activități, ce semnificație ar avea mulțimea 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 și 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 (atenție, reuniunea unor intervale nu este neapărat tot un interval!) și 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] [200, 750] => [100, 900]

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 operație suplimentară de reactualizare a tuturor intervalelor din reuniune. Această problemă apare deoarece intervalul [200, 750] este analizat prea târziu și, 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) și faptul că implementarea sa durează exact o zi. Să se găsească o modalitate de planificare a unor proiecte astfel încât profitul total să fie maxim.

Exemplu:

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

Rezolvare:

În primul rând, 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 și 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 și 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, deși el ar fi putut fi programat și 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 și 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 și 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 și Test2Test, obținând profitul maxim de 3550 RON. Argumentați 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 intrare 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 $\mathcal{O}(n \cdot zi_max)$ și nu este minimă, deși algoritmul Greedy este optim! Pentru a obține 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 priorități va fi prezentat în seminarul următor.

Probleme propuse

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

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

- Scrieți o funcție care să genereze o matrice dublu sortată de dimensiune $m \times n$ cu elemente aleatorii.
- Scrieți 3 funcții, având complexitățile $\mathcal{O}(mn)$, $\mathcal{O}(m \log_2 n)$ și $\mathcal{O}(m + n)$, care să verifice dacă un număr x se găsește sau nu într-o matrice dublu sortată. Funcția va furniza o poziție pe care apare valoarea x în matrice, sub forma unui tuplu (linie, coloană), sau valoarea None dacă x nu se găsește în matrice.

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

3. În fiecare zi, n șoferi transmit către Compania Națională de Administrare a Infrastructurii Rutiere informații referitoare la starea unei anumite autostrăzi, respectiv intervale închise de forma $[x, y]$ având semnificația "asfaltul de pe autostradă este avariat între kilometrii x și y ". Considerând faptul că toate informațiile transmise de către șoferi sunt corecte și cunoscând lungimea autostrăzii respective, scrieți un program care, la sfârșitul zilei respective, să furnizeze angajaților Companiei Naționale de Administrare a Infrastructurii Rutiere următoarele informații:

- porțiunile de autostradă care sunt avariate, sub forma unei reuniuni de intervale;
- porțiunile de autostradă care nu sunt avariate, sub forma unei reuniuni de intervale deschise;
- gradul de uzură al autostrăzii, respectiv raportul dintre numărul total de kilometri avariați de autostradă și lungimea autostrăzii.

Exemplu:

autostrada.in	autostrada.out	Explicații
200	[50, 68]	Lungimea autostrăzii este 200 km.
57 67	[69, 84]	
50 59	[100, 127]	Porțiunile avariate sunt [50, 68], [69, 84],
69 84	[160, 170]	[100, 127] și [160, 170].
100 121		
60 68	(0, 50)	Porțiunile neavariate sunt (0, 50), (68, 69),
73 79	(68, 69)	(84, 100), (127, 160) și (170, 200).
160 170	(84, 100)	
70 80	(127, 160)	Gradul de uzură al autostrăzii este 35%,
120 127	(170, 200)	deoarece numărul total de kilometri avariați
	35%	este 70.

4. Se consideră o mulțime A formată din $n + 1$ numere reale și un număr real x_0 . Considerând faptul că mulțimea A conține cel puțin un număr real nenul, scrieți un program care să determine un polinom $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ de grad n având toate elementele mulțimii A drept coeficienții pentru care valoarea $P(x_0)$ este maximă. De exemplu, pentru $A = \{2, -1, 7, 3\}$ și $x_0 = 2$, polinomul cerut este $P(x) = 7x^3 + 3x^2 + 2x - 1$.

SEMINAR 6

Metoda Greedy. Metoda Divide et Impera

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

Exemplu:

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

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

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

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

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

Exemplu:

```
import queue

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

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

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

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

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

```
import queue

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

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

```

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

    return rez

f = open("siruri.txt")

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

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

f.close()

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

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

```

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

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

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

Exemplu:

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

Rezolvare:

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

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

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

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

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

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

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

```
import queue

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

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

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

fin.close()

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

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

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

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

```

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

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

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

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

fout.close()

```

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

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

Exemple:

```

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

```

Rezolvare:

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

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

```

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

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

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

```

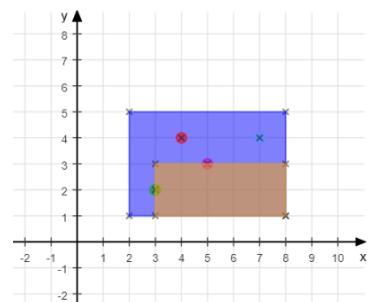
Evident, complexitatea acestei funcții este $\mathcal{O}(\log_2 n)$.

4. Problema debitării

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

Exemplu:

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

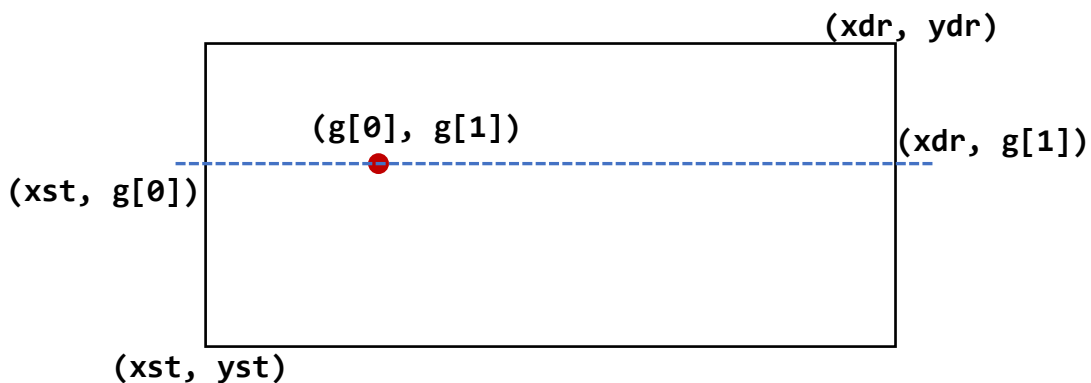


Rezolvare:

Vom utiliza tehnica Divide et Impera pentru a rezolva această problemă, respectiv vom defini o funcție dreptunghiArieMaxima(xst , yst , xdr , ydr) care să prelucreze dreptunghiului curent, indicat prin cei 4 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 tuple, fiecare tuplu fiind de forma *(abscisa, ordonata)*. Dacă dreptunghiul curent are coordonatele (xst, yst, xdr, ydr) și gaura curentă g are coordonatele $(g[0], g[1])$, atunci în urma unei tăieturi orizontale complete se vor forma următoarele două dreptunghiuri:



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

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

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

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

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

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

```

f.close()

return xst, yst, xdr, ydr, coordonateGauri

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

    # considerăm, pe rând, fiecare gaură
    for g in coordonateGauri:
        # dacă gaura curentă se găsește în interiorul dreptunghiului
        # curent, atunci reapelăm funcția pentru cele 4
        # dreptunghiuri care se formează aplicând o tăietură
        # orizontală și una verticală prin gaura curentă
        if xst < g[0] < xdr and yst < g[1] < ydr:
            # dreptunghiurile obținute după o tăietură orizontală
            dreptunghiArieMaxima(xst, yst, xdr, g[1])
            dreptunghiArieMaxima(xst, g[1], xdr, ydr)
            # dreptunghiurile obținute după o tăietură verticală
            dreptunghiArieMaxima(xst, yst, g[0], ydr)
            dreptunghiArieMaxima(g[0], yst, xdr, ydr)
            break

    # dacă dreptunghiul curent nu conține nicio gaură, atunci
    # comparăm aria sa cu aria maximă a unui dreptunghi fără găuri
    # determinată până în momentul respectiv
    else:
        if (xdr-xst)*(ydr-yst) > arieMaxima:
            arieMaxima = (xdr-xst)*(ydr-yst)
            dMaxim = (xst, yst, xdr, ydr)

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

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

```

Deoarece dimensiunile celor 4 subproblemele nu sunt egale, complexitatea acestui algoritm nu poate fi determinată folosind metodele prezentate la curs!

Probleme propuse

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

Indicație de rezolvare:

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

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

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 $O(n^2)$. Se observă foarte ușor 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 desfășurare a unui spectacol), deci poate fi rezolvată folosind algoritmul Greedy deja prezentat și care are complexitatea $O(n \log_2 n)$!

2. Partiționarea 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 partiționare a multisetului A în două submultiseturi X și Y (i.e., $X, Y \subseteq A, X \cup Y = A$ și $X \cap Y = \emptyset$) astfel încât valoarea absolută a diferenței dintre suma elementelor submultisetului X și suma elementelor submultisetului Y să fie minimă. Un *multiset* este o extindere a conceptului de mulțime, respectiv elementele unui multiset nu trebuie să mai fie distincte, ci se pot repeta.

Exemplu:

Multisetul $A = \{4, 6, 4, 6, 15\}$ poate fi partiționat în submultiseturile $X = \{4, 6, 6\}$ și $Y = \{4, 15\}$ astfel încât să se obțină valoarea minimă posibilă pentru diferența dintre sumele elementelor a două submultiseturi de partiție, respectiv 3. Atenție, 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 observație asemănătoare fiind valabilă și pentru reuniunea lor! Astfel, putem considera $X = \{a_1, a_2, a_4\}$ și $Y = \{a_3, a_5\}$.

Această problemă este o variantă a unei probleme NP-complete denumită *problema partiției* (https://en.wikipedia.org/wiki/Partition_problem), în care se cere să se verifice dacă un multiset poate fi partiționat în două submultiseturi având sumele elementelor egale, iar problema partiției este o variantă a altei probleme NP-complete, denumită *problema submulțimii cu sumă dată* (https://en.wikipedia.org/wiki/Subset_sum_problem), în care se cere să se verifice dacă un multiset are o submulțime 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 și cu S_Y suma elementelor submultisetului Y , cele două submultiseturi X și 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 și se obține când $E(x) = 0$, rezultă că suma S_X trebuie să fie cât mai apropiată de valoarea $\left\lfloor \frac{S}{2} \right\rfloor$, respectiv suma S_X trebuie să fie cea mai mare sumă mai mică sau egală decât $\left\lfloor \frac{S}{2} \right\rfloor$ care se poate obține folosind elementele multisetului A . În acest caz, valoarea absolută a diferenței dintre suma elementelor submultisetului X și suma elementelor submultisetului Y va fi minimă și egală cu $S - 2S_X$. Evident, submultisetul Y va fi egal cu $A \setminus X$, unde X este submultisetul determinat astfel încât S_X este cea mai mare sumă mai mică sau egală decât $\left\lfloor \frac{S}{2} \right\rfloor$ care se poate obține 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 și $\left\lfloor \frac{S}{2} \right\rfloor + 1$ coloane, în care un element $sume[i][j]$ va avea valoarea 1 dacă suma j se poate obține 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 observații:

- $sume[i][0] = 1$ pentru orice $i \in \{0, 1, \dots, n\}$ deoarece suma 0 se poate obține 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 $j \in \{1, \dots, \left\lfloor \frac{S}{2} \right\rfloor\}$ deoarece folosind 0 elemente din multisetul A nu se poate obține nicio sumă j nenulă (atenție, 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 obține suma j în următoarele două cazuri:
 - nu se utilizează elementul $a[i]$, ceea ce presupune ca suma j să poată fi obținută 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 $a[i] \leq j$ și suma $j - a[i]$ să poată fi obținută 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, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

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

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 $\left\lfloor \frac{S}{2} \right\rfloor$ 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 $O(n \binom{S}{2}) \approx O(nS) \approx O(n2^{\lceil \log_2 S \rceil})$.

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 cunoaște un interval de timp în care acesta trebuie executat (exprimat prin numerele de ordine a două zile din perioada respectivă), precum și bonusul pe care îl va obține echipa dacă proiectul este finalizat la timp (altfel, echipa nu va obține niciun bonus pentru proiectul respectiv). Să se determine o modalitate de planificare a unor proiecte care nu se suprapun astfel încât bonusul obținut 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 aceeași 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	

Deși 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 soluție corectă. De exemplu, dacă am planifica proiectele în ordinea descrescătoare a bonusurilor, atunci un proiect P_1 ([1,10], 1000 RON) cu bonus mare și durată mare ar fi programat înaintea a două proiecte P_2 ([1,5], 900 RON) și P_3 ([6,9], 800 RON) cu bonusuri și 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 și pentru alte criterii de selecție bazate pe ziua de început, pe ziua de sfârșit, pe durată sau pe raportul dintre bonusul și 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 sfârșit (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 obține 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 obține luând în considerare și 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) și 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		P ₄		P ₂		P ₁	
		1	3	2	6	4	12	7	13
		250		650		800		850	
ult	—	0		0		1		2	
		0		0		1		4	
bmax	0	250		650		1050		1500	
		0		250		650		1050	
		250		650		800+250		850+650	
		900		1000		900		300	
		13		18		15		22	
		25		27		25		27	
		1500		1500		1500		1500	
		1500		1500		1500		1500	
		1500		1500		1500		1500	
		1000+1500		900+1500		300+2500			
		1000+1500		900+1500		300+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 sqrt <= 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 sqrt == 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 $1 + 2 + \dots + 9 = 45$.

Pentru a rezolva problema vom utiliza metoda Backtracking, 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 $\min_k=1$ pentru $k=1$ (prima cifră a unui număr nu poate fi 0) sau $\min_k=0$ pentru $k \geq 2$, respectiv $\max_k=9$;
- $s[1], \dots, s[k-1], s[k]$ este soluție parțială dacă cifra curentă $s[k]$ nu a mai fost utilizată anterior, adică $s[k] \neq s[i]$ pentru orice $i \in \{1, \dots, k-1\}$, și $s[1] + \dots + s[k] \leq c$;
- pentru a testa dacă $s[1], \dots, s[k]$ este soluție vom ține cont de faptul că cifrele $s[1], \dots, s[k]$ sunt distincte (din condițiile de continuare), deci vom verifica doar condiția suplimentară $s[1] + \dots + s[k] == c$.

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	08:20-09:50 Vrajitorul din Oz
09:30-12:10 Punguta cu doi bani	10:00-11:20 Scufita Rosie
08:20-09:50 Vrajitorul din Oz	12:10-13:10 Micul Print
11:30-14:00 Capra cu trei iezi	15:00-15:30 Frumoasa si Bestia
12:10-13:10 Micul Print	
14:00-16:00 Povestea porcului	08:20-09:50 Vrajitorul din Oz
15:00-15:30 Frumoasa si Bestia	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 $A_n^1 + A_n^2 + \dots + A_n^n \gg A_n^n = n!$, 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 și S un număr natural nenul. Folosind metoda Backtracking, să se afișeze toate submultiseturile lui A care au suma elementelor egală cu S .
2. Fie A un multiset format din n numere naturale nenule și S un număr natural nenul. Folosind metoda programării dinamice, să se afișeze un submultiset a lui A care are suma elementelor egală cu S .
3. Folosind doar metoda backtracking, generați toate subșirurile crescătoare maximale ale un șir format din n numere întregi.
4. Optimizați algoritmul de la problema anterioară, utilizând metoda programării dinamice pentru a determina lungimea maximă a unui subșir crescător al șirului dat.