SEMINAR 3
Colecţii de date şi fişiere text

1. În fişierul text *numar_lipsa.txt* se găsesc pe prima linie $n-1$ numere naturale distincte dintre primele n numere naturale nenule. Să se afişeze numărul lipsă. De exemplu, dacă prima linie din fişier 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 biţi (vezi problema 7), având complexitatea computaţională $\mathcal{O}(n)$ şi spaţiul de memorie utilizat $\mathcal{O}(n)$.

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

a) calculăm diferenţa dintre suma t a primelor n numere naturale şi suma s a numerelor din fişier:

```
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("Fişier inexistent!")
```

Complexitatea computaţională a acestei soluţii este $\mathcal{O}(n)$, iar spaţiul de memorie utilizat este tot $\mathcal{O}(n)$.

b) citim numerele din fişier într-o listă şi 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("Fişier 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 $1, 2, \ldots, n$ 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 $\mathcal{O}(n)$, iar spațiu de memorie utilizat este $\mathcal{O}(n)$.

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()
```

Modificaţi programul de mai sus astfel încât să întreruptă forţat citirea din fişier dacă mulţimea `C` a numerelor comune tuturor liniilor devine vidă. Care este complexitatea computaţională a acestui program?

3. Fişierul text *exemplu.txt* conţine un text pe mai multe linii, cuvintele fiind despărţite între ele prin spaţii şi semnele de punctuaţie uzuale. Implementaţi un program care să scrie în fişierul text *grupe_cuvinte.txt* cuvintele distincte (fără duplicate) din fişierul dat, grupate după mulţimile de litere din care sunt formate (nu se va face distincţie între literele mari şi cele mici). Grupele se vor scrie în fişier în ordinea descrescătoare a numărului de elemente din mulţimile 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 Literele imntu: minut, munti, numit Literele abcn: bacan, banc, banca, cabana Literele acmu: acum Literele agln: langa Literele psu: spus, supus Literele ep: pe, pepe Literele in: in Literele is: si Literele nu: un Literele a: a Literele o: o

Rezolvare:
Pentru a determina în mod eficient grupurile de cuvinte care sunt formate din aceleaşi litere (fără a fi neapărat anagrame!), vom utiliza un dicţionar cu perechi de forma *mulţime de litere* : *lista cuvintelor formate din literele respective*. Deoarece cheile unui dicţionar trebuie să fie imutabile, acestea vor fi obiecte de tipul `frozenset` (mulţimi imutabile). Pentru a putea sorta informaţiile din dicţionar în modul precizat în enunţ, vom "exporta" perechile sale de forma *mulţime de litere* : *lista cuvintelor* într-o listă de tupluri de tipul (*şir*, *listă*) în care *şir* va fi un şir de caractere format dintr-o mulţime de litere sortate alfabetic, iar *listă* va conţine, sortate alfabetic, cuvintele formate din literele memorate în *şir*.

```
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 `len(s) = len(t) = n`.

Varianta 1:
Căutăm fiecare literă din şirul `s` în şirul `t` şi, dacă o găsim, atunci o ştergem din şirul `t`, altfel cele două şiruri nu sunt anagrame:

```
def anagrame(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 ştergere a unei anumite litere se poate implementa folosind metoda `replace`, setând valoarea parametrului opţional `count` la 1 pentru a şterge o singură apariţie a literei respective:

```
def anagrame(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 computaţională a ambelor funcţii este $\mathcal{O}(n^2)$.

Varianta 2:
Verificăm dacă fiecare literă distinctă din şirul `s` apare de acelaşi număr de ori şi în şirul `s` şi în şirul `t`:

```
def anagrame(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 anagrame(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 anagrame(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 anagrame(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 anagrame(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 anagrame(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 funcţii este tot $\mathcal{O}(n)$, dar, spre deosebire de varianta cu dicţionare, este stabilă (nu depinde de performanţele unei funcţii de dispersie)!

Probleme propuse

1. Scrieţi o funcţie care verifică dacă două şiruri de caractere sunt anagrame sau nu utilizând un singur vector de frecvenţe.

2. Scrieţi un program care verifică dacă două şiruri de caractere `s` şi `t` formate doar din litere mici sunt anagrame şi, în caz afirmativ, afişează o permutare `p` prin care şirul `t` se obţine din şirul `s`, respectiv o permutare `q` prin care şirul `s` se obţine din şirul `t`. De exemplu, pentru şirurile `s = "emerit"` şi `t = "treime"` o permutare `p` prin care se poate obţine şirul `t` din şirul `s` este $p = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 6 & 2 & 4 & 1 \end{pmatrix}$, deoarece prima literă din şirul `s` (litera `'e'`) trebuie să fie a treia literă din şirul `t`, a doua literă din şirul `s` (litera `'m'`) trebuie să fie a cincea literă din şirul `t`, ş.a.m.d.

Indicaţie de rezolvare:
Dacă şirurile `s` şi `t` sunt anagrame, pentru a construi permutarea `p`, vom căuta poziţia pe care apare în `t` fiecare literă din `s`, după care vom înlocui litera respectivă din `t` cu un spaţiu (sau orice alt caracter care nu este literă mică). De ce nu putem să ştergem litera din `t`? Folosiţi dicţionare pentru a construi cele două permutări!

3. Fişierul text *exemplu.txt* conţine un text pe mai multe linii, cuvintele fiind despărţite între ele prin spaţii şi semnele de punctuaţie uzuale. Implementaţi un program care să scrie în fişierul text *grupe_cuvinte.txt* cuvintele distincte (fără duplicate) din fişierul dat, grupate după lungimile lor. Grupele se vor scrie în fişier 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. Fişierul text *numere.txt* conţine numere naturale pe mai multe linii, numerele fiind despărţite între ele prin spaţii. Scrieţi un program care să afişeze cel mai mare şi cel mai mic număr care se poate obţine din toate cifrele tuturor numerelor din fişier. Rezolvaţi problema în cel puţin 3 moduri diferite, respectiv folosind şiruri de caractere, liste, mulţimi şi dicţionare!

Exemplu:

numere.txt	**ecran**

`215 13 1009`
`2377`
`1024 9099 2020 23`
	`Minim: 10000009999775433322222111`
	`Maxim: 99997754333222222111000000`