COMPLEXITATEA ALGORITMILOR. EXEMPLE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    if avantaj == 0:
        return 0
    if voturi.count(majoritar) > len(voturi) // 2:
        return majoritar
    return 0
```

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

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

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

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

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

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

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

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

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

```
                break
            elif S-L[i] < L[mij]:
                dr = mij-1
            else:
                st = mij+1
    return rez
```

Deoarece căutarea binară a valorii `S-L[i]` în sublista `L[i+1:]` are complexitatea maximă $\mathcal{O}(\log_2 n)$, această funcție va avea complexitatea $\mathcal{O}(n\log_2 n)$. La fel ca în varianta precedentă, putem îmbunătăți eficiența algoritmului oprind căutarea valorii `S-L[i]` în sublista `L[i+1:]` dacă `L[i] >= S//2`.

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

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

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

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

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

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

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

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

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 3 programe, având complexitatea $\mathcal{O}(mn)$, $\mathcal{O}(m \log_2 n)$ și $\mathcal{O}(m + n)$, care să verifice dacă un număr x se găsește sau nu într-o matrice dublu sortată. Un program va afișa 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. **Indicație de rezolvare:** Search in a row wise and column wise sorted matrix - GeeksforGeeks

2. Considerăm un șir format din n produse, așezate pe raftul cu promoții al magazinului CheapProds. Deoarece produsele de pe acest raft se vând foarte repede, Tom Buyer ar vrea să găsească o modalitate rapidă prin care să poată selecta o secvență de produse având prețul total egal cu suma de bani s pe care el este dispus să o cheltuiască. Puteți să-l ajutați pe Tom Buyer?

Exemplu:

produse.in	produse.out
14 7 5 5 10 2 1 1 6 30 12 2 2 4 10 20	2 4 4 8 10 13

Explicație: Sunt $n = 14$ produse având prețurile indicate și se dorește găsirea unei secvențe de produse având prețul total egal cu $s = 20$. Există 3 soluții: secvența formată din produsele 2, 3 și 4 (având prețurile 5, 5 și 10 RON), secvența formată din produsele 4, 5, 6, 7 și 8 (având prețurile 10, 2, 1, 1 și 6 RON), respectiv secvența formată din produsele 10, 11, 12 și 13 (având prețurile 12, 2, 2 și 4 RON). **Indicație de rezolvare:** Find Subarray with given sum | Set 1 (Non-negative Numbers) - GeeksforGeeks

3. Primăria municipiului Bucureşti a hotărât să scoată la licitaţie câte un chioşc în fiecare dintre cele n staţii de pe traseul autobuzului 2015. În acest scop, Primăria a efectuat un studiu de fezabilitate cu ajutorul căruia a fost estimat profitul lunar, în RON, ce poate fi obţinut de către fiecare chioşc. În unele staţii, aflate în zone mai puţin circulate, s-a constatat că este posibil să nu se obţină nici un profit, ci chiar să se înregistreze pierderi, adică profitul să fie negativ. Pentru a putea să vândă totuşi toate chioşcurile, Primăria a hotărât ca fiecare investitor interesat să fie obligat să cumpere o singură secvenţă de chioşcuri (aflate în staţii consecutive). Sarcina voastră este să determinaţi secvenţa de chioşcuri pentru care trebuie să liciteze un investitor astfel încât profitul estimat să fie maxim. Deoarece orice investitor doreşte să păstreze relaţii cordiale cu Primăria, în cazul în care nu există nici o secvenţă de chioşcuri care să aducă profit, el va licita o secvenţă de chioşcuri pentru care pierderile sunt minime.

Exemple:

profit.in	**profit.out**
7 10 9 -23 7 10 11 -3	28 4 6
10 -47 -5 90 -78 -10 -95 62 -20 54 90	186 6 9

Indicaţii de rezolvare:

- Subsecvenţa de sumă maximă (infoarena.ro)
- Secvenţa de sumă maximă | www.pbinfo.ro
- Largest Sum Contiguous Subarray (Kadane's Algorithm) - GeeksforGeeks