Tema 2: Analiza complexităţii algoritmilor. Algoritmi de căutare şi sortare. Aplicaţii ale tehnicilor de proiectare a algoritmilor (reducere, divizare, greedy, programare dinamică) - indicaţii şi variante de rezolvare.

---

1. Se consideră trei tablouri de numere întregi, $a[1..n]$, $b[1..n]$, $c[1..n]$. Se pune problema verificării dacă există cel puţin un element comun în cele trei tablouri. De exemplu tablourile $a = [3, 1, 5, 10]$, $b = [4, 2, 6, 1]$, $c = [5, 3, 1, 7]$ au un element comun, pe când $a = [3, 1, 5, 10]$, $b = [4, 2, 6, 8]$, $c = [15, 6, 1, 7]$ nu au nici un element comun.

   (a) Propuneţi un algoritm de complexitate $O(n^3)$ care returnează `True` dacă cele trei tablouri conţin cel puţin un element comun şi `False` în caz contrar. Justificaţi faptul că algoritmul are complexitatea cerută şi implementaţi algoritmul în Python.
   Indicaţie. Pentru fiecare triplet $(i, j, k) \in \{1, \ldots, n\} \times \{1, \ldots, n\} \times \{1, \ldots, n\}$ se verifică dacă $a[i] = b[j] = c[k]$ şi în caz afirmativ se returnează True. La ieşirea din prelucrarea repetitivă (cele trei cicluri suprapuse) se returnează False. Operaţia dominantă este comparaţia. In cazul cel mai favorabil ($a[1] = b[1] = c[1]$) se poate considera că se efectuează o singură operaţie, iar în cazul cel mai defavorabil (nu există elemente comune în cele trei tablouri) se efectuează n^3 operaţii, prin urmare algoritmul aparţine lui $\Omega(1)$ şi $\mathcal{O}(n^3)$.

   (b) Propuneţi un algoritm de complexitate $O(n^2)$ care returnează `True` dacă cele trei tablouri conţin cel puţin un element comun şi `False` în caz contrar. Justificaţi faptul că algoritmul are complexitatea cerută şi implementaţi algoritmul în Python.
   Indicaţie. Se parcurg tablourile $a[1..n]$ şi $b[1..n]$ şi se construieşte tabloul $d[1..k]$ ($k <= n$) care conţine elementele comune din a şi b. Construirea tabloului d are ordinul de complexitate $\mathcal{O}(n^2)$. Dacă tabloul d este nevid se parcurg $d[1..k]$ şi $c[1..n]$ pentru a identifica eventualele elemente comune. Această etapă are ordinul de complexitate $\mathcal{O}(kn)$.

   (c) Presupunând că elementele tablourilor sunt din $\{1, 2, \ldots, m\}$ propuneţi un algoritm de complexitate $O(\max(m, n))$ care returnează `True` dacă cele trei tablouri conţin cel puţin un element comun şi `False` în caz contrar. Justificaţi faptul că algoritmul are complexitatea cerută şi implementaţi algoritmul în Python.
   Indicaţie. este permisă utilizarea unei zone suplimentare de memorie de dimensiune $\mathcal{O}(m)$. Se construieşte un tabel de frecvenţ $f[1..m]$ care se completează în urma parcurgerii separate a fiecăruia dintre cele trei tablouri (ordin de complexitate $\Theta(n)$). După construirea lui $f[1..m]$ acesta se parcurge pentru a verifica dacă există cel puţin un element care are valoarea 3 (corespunde unei valori comune tuturor celor trei tablouri) - ordinul de complexitate este $\mathcal{O}(m)$.

   (d) Presupunând că toate cele trei tablouri sunt ordonate crescător propuneţi un algoritm de complexitate $O(n)$ care returnează `True` dacă cele trei tablouri conţin cel puţin un element comun şi `False` în caz contrar. Justificaţi faptul că algoritmul are complexitatea cerută şi implementaţi algoritmul în Python.
   Indicaţie. Se poate folosi ideea de la tehnica interclasării. Se construieşte (folosind tehnica interclasării - ordin de complexitate $\Theta(n)$) tabloul $d[1..k]$ care conţine elementele comune din a şi b. Dacă $d[1..k]$ este nevid se aplică tehnica de interclasare pentru d şi c, oprind parcurgerea în momentul în care este identificat un element comun în d şi c (ordinul de complexitate are fi $\mathcal{O}(\max\{k, n\})$.
   O altă variantă de rezolvare este de a parcurge "balansat" (ca în algoritmul de interclasare): (i) se compară cele trei elemente curente; (ii) dacă sunt identice se returnează

True, iar dacă nu sunt identice se progresează doar în tabloul/tablourile care conţin valori strict mai mici decât cea mai mare valoare din tripletul curent; (iii) prelucrarea continuă atât timp cât există încă elemente neanalizate în toate tablourile. Întrucât la fiecare etapă se progresează cel puţin într-un tablou, numărul de repetări ale ciclului este cel mult $3n$. La fiecare execuţie a corpului ciclului se efectuează cel mult 17 comparaţii. Ordinul de complexitate este $\mathcal{O}(n)$.

```
def elemcomun4(a,b,c):
    n=len(a)
    i=0
    j=0
    k=0
    while (i<n) and (j<n) and (k<n):
        if a[i]==b[j] and b[j]==c[k]:
            return True
        else:
            if a[i]>=b[j] and a[i]>=c[k]:
                if a[i]>b[j]:
                    j=j+1
                if a[i]>c[k]:
                    k=k+1
            elif b[j]>=a[i] and b[j]>=c[k]:
                if b[j]>a[i]:
                    i=i+1
                if b[j]>c[k]:
                    k=k+1
            elif c[k]>=a[i] and c[k]>=b[j]:
                if c[k]>a[i]:
                    i=i+1
                if c[k]>b[j]:
                    j=j+1
    return False
```

2. (a) Se consideră o matrice $A = (a_{ij})_{i=\overline{1,m},j=\overline{1,n}}$ cu m linii şi n coloane. Propuneţi (şi implementaţi în Python) un algoritm care rearanjează liniile matricii astfel încât să fie în ordine crescătoare în raport cu norma euclidiană a liniilor (norma euclidiană a liniei i a matricii A este $\sqrt{a_{i1}^2 + a_{i2}^2 + \ldots + a_{in}^2}$). Stabiliţi ordinul de complexitate al algoritmului propus considerând ca operaţii dominante toate operaţiile efectuate asupra elementelor matricii (comparaţii, adunări, înmulţiri).

   (b) Se consideră o listă cu date de naştere specificate prin triplete de forma (zi, luna, an). Propuneţi (şi implementaţi în Python) un algoritm care ordonează crescător datele de naştere. Stabiliţi ordinul de complexitate al algoritmului propus.

Indicaţie. (a) Intrucât transferul elementelor este o operaţie de cost $\Theta(n)$ se recomandă utilizarea unui algoritm care utilizează un număr mic de transferuri (sortare prin selecţie sau sortare rapidă). Pentru operaţia de comparare a două linii este suficient să se calculeze pătratul normei euclidiene a fiecărei linii (nu este necesar să se calculeze radicalul, întrucât nu se modifică relaţia de ordine prin aplicarea radicalului). In cazul în care se construieşte înainte de sortare un tablou ca valorile pătratelor distanţelor euclidiene, ordinul de complexitate al procesului de sortare este $\mathcal{O}(nm + m^2)$ sau $\mathcal{O}(nm + m \log m)$ (în acest caz valorile din acest

tablou sunt ordonate împreună cu liniile matricii). In cazul în care se calculează pătratul distanţei euclidiene de fiecare dată când o linie este implicată într-o comparaţie ordinul de complexitate este $\mathcal{O}(nm^2)$ sau $\mathcal{O}(nm \log m)$. O variantă de implementare este:

```
def norm2(x):
    n=len(x)
    s=0
    for i in range(n):
        s=s+x[i]*x[i]
    return s

def sortSelectie(a):
    m=len(a)
    v=[0]*m
    for i in range(m):  # Theta(m*n)
        v[i]=norm2(a[i])
    for i in range(1,m-1):
        imin=i
        for j in range(i+1,m):
            if v[j]<v[imin]: # Theta(m^2)
                imin=j
        if imin!=i:
            v[i],v[imin]=v[imin],v[i]
            a[i],a[imin]=a[imin],a[i] # O(m*n)
    return a
```

(b) O variantă de rezolvare este aplicarea sortării prin numărare (varianta stabilă) succesiv în raport cu diferite chei de sortare: zi, lună, an. In cazul unui tablou cu n elementete ordinul de complexitate este $\Theta(\max m, n)$ unde m este cea mai mare valoare din lista cu zile, luni, ani.

3. Se pune problema implementării unor operaţii aritmetice cu numere reprezentate prin tablouri de n cifre (de exemplu, numărul 65189218901567 este reprezentat (pentru $n = 15$) prin $[0, 6, 5, 1, 8, 9, 2, 1, 8, 9, 0, 1, 5, 6, 7]$. Pentru fiecare dintre operaţiile următoare propuneţi un algoritm, stabiliţi ordinul de complexitate şi implementaţi-l în Python:

   (a) compararea a două numere a şi b reprezentate prin tablouri cu n elemente (algoritmul va returna $+1$ dacă a este mai mare decât b şi -1 în caz contrar).

   (b) calculul sumei a două numere a şi b reprezentate prin tablouri cu n elemente (rezultatul va fi reprezentat tot folosind un tablou cu n elemente, iar dacă valoarea obţinută depăşeşte zona alocată atunci se va returna un tablou cu toate elementele egale cu -1).

   (c) calculul diferenţei dintre două numere a şi b reprezentate prin tablouri cu n elemente (rezultatul va fi reprezentat tot folosind un tablou cu n elemente). Se presupune că $a \geq b$.

   (d) calculul produsului a două numere a şi b reprezentate prin tablouri cu n elemente (rezultatul va fi reprezentat tot folosind un tablou cu n elemente, iar dacă valoarea obţinută depăşeşte zona alocată atunci se va returna un tablou cu toate elementele egale cu -1).

Indicaţie. (a) Se parcurg cele două valori începând cu primul element (cifra cea mai semnificativă) şi la prima pereche de valori diferite se returnează rezultatul (1 dacă cifra din tabloul a este mai mare decât cea din tabloul b, respectiv -1 în caz contrar. Dacă nu sunt cifre diferite în cele două tablouri se returnează 0. Ordinul de complexitate este $\mathcal{O}(n)$ (dimensiunea problemei este dată de dimensiunea tabloului, iar operaţia dominantă este comparaţia).

(b) Se parcurg cele două tablouri de la ultimul element (cifra cea mai puţin semnficativă) către primul şi pentru fiecare poziţie i se calculează $s[i] = (a[i] + b[i] + r)MOD10$. Reportul r se iniţializează cu 0 şi la fiecare etapă (după calculul lui $s[i]$) se actualizează $r = (a[i] + b[i]+r)DIV10$. Dacă reportul obţinut la adunarea elementelor corespunzătoare primei poziţii este nenul atunci se consideră depăşire. Ordinul de complexitate este $\Theta(n)$ (oricare dintre operaţiile ce intervin în calculul lui $s[i]$ poate fi considerată dominantă.

(c) Se parcurg cele două tablouri de la ultimul element (cifra cea mai puţin semnficativă) către primul şi pentru fiecare poziţie i se calculează $d[i]$ ţinând cont de relaţia dintre $a[i]$ şi $b[i]$: dacă $a[i] \geq b[i]$ atunci $d[i] = a[i] - b[i]$, iar în caz contrar $d[i] = 10 + a[i] - b[i]$ iar $a[i-1] = a[i-1] - 1$. Ordinul de complexitate este $\Theta(n)$.

(d) Se poate interpreta problema ca cea de înmulţire a două polinoame (coeficientul termenului de grad k al polinomului produs fiind $p[k] = \sum i + j = ka[i] \cdot b[j]$, în ipoteza că elementul de pe poziţia i corespunde coeficientului termenului de grad i) după care coeficienţii care depăşesc valoarea 9 se transformă prin transferul câtului împărţirii la 10 către elementul de grad imediat superior (indice cu 1 mai mic). La implementarea algoritmului pentru calculul produsului polinoamelor trebuie avut grijă la faptul că în cazul acestei probleme indicii cresc în ordinea inversă a gradelor. Ordinul de complexitate al algoritmului este $\mathcal{O}(n^2)$.

4. Se consideră o matrice $A[1..m][1..n]$ având elementele de pe fiecare linie şi elementele de pe fiecare coloană ordonate crescător. Se pune problema verificării prezenţei unei valori v în matrice, folosind un număr cât mai mic de comparaţii.
De exemplu matricea $[[3, 6, 10], [5, 9, 12], [11, 14, 16]]$ satisface proprietatea specificată. Descrieţi şi implementaţi algoritmul bazat pe următoarea idee:

   - se porneşte căutarea din colţul dreapta sus al matricii ($i = 1$, $j = n$)
   - dacă $v = A[i, j]$ atunci elementul a fost găsit şi se returnează True; dacă $v < A[i, j]$ atunci se continuă căutarea în stânga (se micşorează valoarea lui j); dacă $v > A[i, j]$ atunci se continuă căutarea în jos (se măreşte valoarea lui i); în cazul în care nu se mai poate continua căutarea (s-a ajuns la $i = m$ sau $j = 1$) atunci se returnează False.

   Este corect algoritmul? Argumentaţi. Stabiliţi ordinul de complexitate.

   Indicaţie. O valoare mai mică decât $a[i][j]$ nu se poate afla în submatricea care are pe $a[i][j]$ în colţul din stânga sus, deci căutarea trebuie continuată pentru valori mai mici ale lui j. Pe de altă parte, o valoare mai mare decât $a[i][j]$ nu se poate afla în submatricea care are pe $a[i][j]$ în colţul din dreapta jos, deci trebuie continuat cu valori mai mari ale lui i. In ceea ce priveşte ordinul de complexitate, se poate observa că la fiecare etapă se progresează fie pe orizontală (înspre stânga) fie pe verticală (în jos) prin urmare numărul de comparaţii efectuate este proporţional cu $m + n$, deci ordinul de complexitate este $\mathcal{O}(\max\{m, n\})$.

```
def caut(a,v):
    m=len(a)
    n=len(a[0])
    i=0
```

```
j=n-1
while (i<m) and (j>=0):
    if a[i][j]==v:
        return True
    elif v<a[i][j]:
        j=j-1
    else:
        i=i+1
return False
```

5. Se consideră doi algoritmi recursivi A şi B şi se presupune că timpii lor de execuţie satisfac următoarele relaţii de recurenţă: $T_A(n) = 7T_A(n/2) + n^2$ respectiv $T_B(n) = aT_B(n/4) + n^2$. Pentru ce valori ale lui a, algoritmul B are un ordin de complexitate mai mic decât algoritmul A?

   Indicaţie. Se poate folosi teorema Master pentru estimarea ordinului de complexitate al fiecărui algoritm. Pentru primul algoritm constantele care intervin în teorema Master sunt $k = 7$, $m = 2$ şi $d = 2$ deci conform cazului 3 al teoremei ordinul de complexitate este $\mathcal{O}(n^{\log 7/\log 2})$. Pentru al doilea algoritm, dacă $n \leq 16$ ordinul de complexitate (n^2 sau $n^2 \log n$) este mai mic decât $n^{\log 7/\log 2}$. Dacă $a > 16$ atunci algoritmul B aparţine lui $\mathcal{O}(n^{\log a/\log 4})$ deci pentru $a < 49$) ordinul de complexitate este mai mic decât cel al algoritmului A.

6. Se consideră un set de n obiecte caracterizate prin dimensiunile $d_1 < d_2 < \ldots < d_n$ şi valorile $v_1 > v_2 > \ldots > v_n$ (ordonarea crescătoare după dimensiune corespunde cu ordonarea descrescătoare după valoare). Stiind că dimensiunile d_i au valori reale, propuneţi un algoritm care să selecteze un subset de obiecte care să încapă într-un rucsac de capacitate $C \in \mathbf{R}$ şi care să aibă valoarea maximă.

   Indicaţie. Intrucât problema rucsacului are proprietatea de substructură optimă, este suficient să se demonstreze că strategia de selecţie a obiectelor în ordinea specificată în enunţ conduce la soluţii care satisfac proprietatea de alegere greedy. Considerăm o soluţie optimă $S = (s_1, s_2, \ldots, s_n)$. Dacă $s_1 = 1$ atunci proprietatea de alegere greedy este satisfăcută. Dacă $s_1 = 0$ atunci prin excluderea ultimului obiect inclus în rucsac (care are dimensiunea mai mare şi valoarea mai mică decât o_1) şi includerea lui o_1 s-ar obţine o soluţie de valoare mai mare, ceea ce contrazice faptul că o soluţie cu $s_1 = 0$ ar fi optimă. Prin urmare poate fi aplicată tehnica greedy prin selecţia succesivă a obiectelor până la întâlnirea primului obiect care nu mai încape în rucsac.

7. Se consideră o mulţime $A = \{a_1, a_2, \ldots, a_n\}$ cu elemente numere naturale, astfel încât $\sum_{i=1}^{n} a_i = 2k$. Să se determine o descompunere a lui A în două submulţimi disjuncte B şi C astfel încât $B \cup C = A$ şi suma elementelor din B, respectiv din C, să fie egală cu k.

   Indicaţie. Se reformulează problema ca una de optimizare cu restricţii pentru care poate fi aplicată tehnica programării dinamice: se caută o submulţime $S \subset A$ cu proprietatea că $\sum_{s \in A} s \leq k$ şi în acelaşi timp $\sum_{s \in A}$ este maximă. Problema este similară cu varianta discretă a problemei rucsacului pentru care criteriul de optimizat este să rămână cât mai puţin spaţiu liber în rucsac (echivalent cu cazul în care valoarea obiectelor coincide cu dimensiunea lor).

   Dacă $P(i, j)$ este problema selectării unei submulţimi din mulţimea $\{a_1, a_2, \ldots, a_i\}$ astfel încât suma elementelor submulţimii să fie cât mai apropiată de j atunci relaţia de recurenţă

corespunzătoare sumei elementelor submulţimii este:

$$S(i,j) = \begin{cases} 0 & \text{dacă } i = 0 \text{ sau } j = 0 \\ S(i-1,j) & \text{dacă } a_i > j \\ \max\{S(i-1,j), S(i-1,j-a_i)+a_i\} & \text{dacă } a_i \leq j \end{cases}$$

```
def matriceSuma(a,k):
    n=len(a)
    s=[[0 for j in range(k+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(k+1):
            if a[i-1]>j:
                s[i][j]=s[i-1][j]  # elementul i nu e selectat
            else:
                s[i][j]=max(s[i-1][j],s[i-1][j-a[i-1]]+a[i-1])
                # s[i][j]==s[i-1][j] - elementul i nu este selectat
                # s[i][j]!=s[i-1][j] - elementul i este selectat
    return(s)
def construireSolutie(i,j):
    #global solutie,a,s
    if (i==0 or j==0 or s[i][j]==0):
        return
    else:
        if (s[i][j]==s[i-1][j]):
            construireSolutie(i-1,j)

        else:
            solutie[i-1]=1
            construireSolutie(i-1,j-a[i-1])
```

8. Se consideră un set de activităţi A_1, A_2, ..., A_n, fiecare activitate, A_i, fiind caracterizată prin:

   (a) durata: $d_i \in \mathbf{N}$

   (b) termenul final de execuţie: $t_i \in \mathbf{N}$

   (c) profitul obţinut dacă se execută activitatea înainte de termenul final: $p_i \in \mathbf{R}$ (dacă activitatea nu este executată profitul este 0).

   Se pune problema determinării unei planificări a execuţiei activităţilor: $(T_1, T_2, \ldots, T_n)$, unde T_i reprezintă momentul startării activităţii A_i (dacă activitatea nu este executată atunci $T_i = -1$) astfel încât să fie îndeplinite condiţiile următoare:

   (a) activităţile planificate sunt compatibile între ele (nu pot fi executate două activităţi în acelaşi timp); două activităţi A_k şi A_l pot fi planificate dacă $T_k + d_k \leq \min\{t_k, T_l\}$ (în cazul în care $T_k < T_l$) respectiv $T_l + d_l \leq \min\{t_l, T_k\}$ (în cazul în care $T_l < T_k$);

   (b) profitul adus de activităţile planificate, $\sum_{k, T_k \neq -1} p_k$, este maxim.

   Propuneţi un algoritm bazat pe tehnica programării dinamice care permite determinare a uneia planificări optimale.

Indicaţie. Presupunem că activităţile sunt ordonate crescător după termenul final de execuţie: $t_1 \leq t_2 \leq \ldots \leq t_n$ şi considerăm problema generică $P(i,j)$ a determinării unei planificări optimale pentru activităţile A_1, A_2, ..., A_i care se finalizează până la momentul j.
(a) *Analiza unei soluţii optime şi identificarea subproblemelor.* Fie $(T_1, T_2, \ldots, T_i)$ o soluţie optimă a problemei $P(i,j)$. Sunt posibile două cazuri: (i) activitatea A_i nu este planificată ($T_i = -1$) ceea ce înseamnă că rezolvarea problemei $P(i,j)$ se reduce la rezolvarea problemei $P(i-1,j)$; (ii) activitatea A_i este planificată ($T_i \neq -1$) ceea ce înseamnă că rezolvarea problemei $P(i,j)$ se reduce la rezolvarea problemei $P(i-1, j-d_i)$;

(b) *Construirea relaţiei de recurenţă.* Fie $R(i,j)$ profitul maxim obţinut prin planificarea activităţilor A_1, A_2, ..., A_i până la momentul j.

$$R(i,j) = \begin{cases} 0 & i = 0 \text{ sau } j = 0 \\ R(i-1,j) & \min\{j, t_i\} < d_i \\ \max_{0 \leq k \leq j-d_i}\{R(i-1,j), R(i-1,k) + p_i\} & \min\{j, t_i\} \geq d_i \end{cases}$$

Pentru a facilita construirea soluţiei este util de reţinut pentru fiecare pereche (i,j) care este valoarea k pentru care se obţine valoarea maximă a profitului:

$$T(i,j) = \begin{cases} -1 & \text{dacă } R(i,j) = R(i-1,j) \\ kmax & \text{dacă } R(i-1,kmax) + p_i \geq R(i-1,k) + p_i, 0 \leq k \leq j - d_i \end{cases}$$

O variantă mai simplă de rezolvare este cea în care se foloseşte direct $\min\{j, t_i\} - d_i$ ca moment de start al activităţii i (cel mai târziu moment la care poate să înceapă activitatea astfel încât aceasta să se finalizeze înainte de termenul t_i dar şi înainte de finalul orizontului de timp stabilit, j). In acest caz este suficientă construirea matricii R iar profitul maxim care se poate obţine este pe ultima linie, ultima coloană a matricii.

```
def matriceProfit(d,p,t):
    n=len(d) # numarul total de activitati
    k=t[n-1] # ultimul termen de executie
    R=[[0 for j in range(k+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(k+1):
            dif=min(j,t[i-1])-d[i-1]
            # cel mai tarziu moment la care poate fi planificata activitatea (i-1)
            # astfel incat sa se finalizeze inainte de momentul j si
            # inainte de termenul de finalizare
            # daca dif<0 atunci activitatea nu poate fi planificata astfel incat sa
            # fie satisfacute restrictiile
            if dif<0:
                R[i][j]=R[i-1][j]  # activitatea i nu poate fi planificata
            else:
                R[i][j]=max(R[i-1][j],R[i-1][dif]+p[i-1])
    return(R)
def construireSolutie(d,t,R):
    n=len(R)-1
    k=len(R[0])-1
    i=n
    j=k
```

```
while(i>0 and j>0):
    if (R[i][j]==R[i-1][j]):
        i=i-1
    else:
        dif=min(j,t[i-1])-d[i-1]
        solutie[i-1]=dif
        i=i-1
        j=dif
```

9. Secvenţele ADN sunt succesiuni de simboluri "A", "C", "G", "T" corespunzătoare celor patru tipuri de nucleotide: "adenină", "citozină", "guanină" şi "timină". Căutarea în bazele de date ce conţin secvenţe ADN se bazează pe alinierea secvenţelor prin maximizarea unui scor de potrivire. Două secvenţe (care iniţial pot fi de lungimi diferite) se consideră aliniate dacă prin introducerea unor spaţii (gap-uri) ajung să aibă aceeaşi lungime, astfel că fiecare element din prima secvenţă (nucleotidă sau gap) este pus în corespondenţă cu un element din a doua secvenţă (nucleotidă sau gap). Scorul unei alinieri se determină calculând suma scorurilor de potrivire ale elementelor corespondente din cele două secvenţe. O aliniere se consideră optimă dacă scorul său este maxim.

   Exemplu. Considerăm secvenţele "CGTTA" şi "AGTA" şi scorurile de potrivire: $+2$ (nucleotide identice aliniate), -2 (nucleotide diferite aliniate) şi -1 (nucleotidă aliniată cu un gap). In acest caz alinierea optimă (de scor maxim) este:
   AG-TA
   CGTTA
   cu scorul 3 ($= -2 + 2 - 1 + 2 + 2$).

   (a) Scrieţi relaţia de recurenţă care permite calculul scorului corespunzător unei alinieri optime.

   (b) Descrieţi în pseudocod şi implementaţi în Python algoritmul pentru calculul scorului corespunzător unei alinieri optime (folosind scorurile de potrivire între elemente specificate în exemplul de mai sus). Nu e necesară construirea alinierii.

   Indicaţie. Se poate utiliza ideea, bazată pe tehnica programării dinamice, de la calculul distanţei de editare cu deosebirea că pentru elementele identice se adaugă scorul de potrivire, pentru cele diferite se adaugă scorul de nepotrivire iar în cazul ştergerii sau inserţiei se adaugă valoarea scorului corespunzător alinierii cu un gap. In plus, spre deosebire de calculul distanţei de editare unde se urmăreşte minimizarea numărului de operaţii în acest caz se urmăreşte maximizarea scorului de aliniere.

   Dacă cele două secvenţe sunt $a[1..m]$ şi $b[1..n]$ relaţia de recurenţă pentru calcului scorului de potrivire este:

$$S[i,j] = \begin{cases} -j & \text{dacă } i = 0 \\ -i & \text{dacă } j = 0 \\ S[i-1,j-1] + 2 & \text{dacă } a[i] = b[j] \\ \max\{S[i-1,j]-1, S[i,j-1]-1, S[i-1,j-1]-2\} & \text{dacă } a[i] \neq b[j] \end{cases}$$

```
def scorAliniere(a,b):
    m=len(a)
```

```
    n=len(b)
    s=[[0 for j in range(n+1)] for i in range(m+1)]
    for i in range(1,m+1):
        s[i][0]=-i
    for j in range(1,n+1):
        s[0][j]=-j
    for i in range(1,m+1):
        for j in range(1,n+1):
            if a[i-1]==b[j-1]:
                s[i][j]=s[i-1][j-1]+2 # potrivire
            else:
                s[i][j]=max(s[i-1][j]-1,max(s[i][j-1]-1,s[i-1][j-1]-2))

    return(s[m][n])
```

10. *Generarea codului Gray.* Codul Gray este o variantă de reprezentare binară caracterizată prin faptul că valori consecutive au asociate secvenţe binare care diferă într-o singură poziţie. De exemplu pentru $n = 3$ codificarea Gray este: $0 : (0, 0, 0)$; $1 : (0, 0, 1)$; $2 : (0, 1, 1)$; $3 : (0, 1, 0)$; $4 : (1, 1, 0)$; $5 : (1, 1, 1)$; $6 : (1, 0, 1)$; $7 : (1, 0, 0)$. Codul poartă numele unui cercetător (Frank Gray) de la AT&T Bell Laboratories care l-a utilizat pentru a minimiza efectul erorilor de transmitere a semnalelor digitale.

    (a) Propuneţi un algoritm iterativ prin care codul Gray se obţine pornind de la codificarea clasică în baza 2. De exemplu pentru $n = 3$ transformarea constă în: $(0, 0, 0) \to (0, 0, 0)$; $(0, 0, 1) \to (0, 0, 1)$; $(0, 1, 0) \to (0, 1, 1)$; $(0, 1, 1) \to (0, 1, 0)$; $(1, 0, 0) \to (1, 1, 0)$; $(1, 0, 1) \to (1, 1, 1)$; $(1, 1, 0) \to (1, 0, 1)$; $(1, 1, 1) \to (1, 0, 0)$;

    (b) Propuneţi un algoritm recursiv care generează codul Gray de ordin n direct, fără a folosi reprezentarea în baza 2 a valorilor din $\{0, 1, \ldots, 2^{n-1}\}$.

    Indicaţie. (a) Dacă $b[1..n]$ conţine secvenţa binră curentă atunci secvenţa corespunzătoare codului Gray se poate obţine astfel: $g[1] = b[1]$, $g[i] = (b[i-1] + b[i])\text{MOD}2$, pentru $i = \overline{2, n}$.

    (b) Ideea de rezolvare se bazează pe observaţia că având secvenţa de coduri Gray pentru $n - 1$ poziţii atâta timp cât se prefixează cu o aceeaşi valoare (0 sau 1) proprietatea de secvenţă Gray este conservată. Pentru a conserva proprietatea la trecerea de la prefiz 0 la prefix 1 este suficient să se inverseze ordinea elementelor din secvenţă astfel încât cele două secvenţe de lungime $n - 1$ care vor fi prefixate cu 0 respectiv cu 1 să fie identice. De exemplu pentru a obţine secvenţa Gray pentru $n = 3$ se porneşte de la cea corespunzătoare lui $n = 2$: $[0, 0], [0, 1], [1, 1], [1, 0]$ prin prefixarea elementelor din această secvenţă cu 0: $[0, 0, 0], [0, 0, 1], [0, 1, 1], [0, 1, 0]$ apoi prin prefixarea cu 1 a aceloraşi elemente dar parcurse în sens invers: $[1, 1, 0], [1, 1, 1], [1, 0, 1], [1, 0, 0]$.

```
def reverse(g): # inversarea ordinii elementelor
    n=len(g)
    grev=[]
    i=n-1
    while i>=0:
        grev.append(g[i])
        i=i-1
    return grev
```

```
def GrayRec(n):
    if (n==1):
        g=["0","1"] # lista cu codurile pentru n=1
        return g
    else:
        g1=GrayRec(n-1) # lista cu codurile avand n-1 elemente
        g2=reverse(g1) # aceeasi lista in ordine inversa
        lg=len(g1)
        for i in range(lg):
            g1[i]="0"+g1[i] # prefixarea cu 0 a elementelor din prima lista
            g2[i]="1"+g2[i] # prefixarea cu 1 a elementelor din a doua lista
        g=[]
        for i in range(lg): # concatenarea celor doua liste
                g.append(g1[i])
        for i in range(lg):
                g.append(g2[i])
    return g
```