Tema 1: Rezolvarea algoritmică a problemelor. Descrierea algoritmilor în pseudocod. Verificarea corectitudinii algoritmilor.

Termen: 9.11.2018

Submiterea temelor: Se uploadează prin Classroom (cf indicaţiilor primite de la cadrul didactic coordonator de seminar) o arhiva de tip *.zip având numele construit pe baza regulii: NumeStudent_seria_subgrupa_Tema1.zip (de exemplu `AdamEva_IR_sgr3_Tema1.zip`).

Arhiva trebuie să conţină următoarele fişiere:

- Un fişier în format PDF sau DOC care să conţină soluţiile propuse (răspunsuri la întrebări, algoritmi descrişi în pseudocod, explicaţii etc.)
- Câte un fişier cu codul sursă Python corespunzător implementărilor solicitate în enunţ denumit NumeStudent_seria_subgrupa_Tema1_Problema (de exemplu `AdamEva_IR_sgr3_Tema1_Pb1b.py`)

Punctaj total: 100p (+ bonus max 10p)

---

1. *Reprezentarea numerelor întregi în complement faţă de 2* pe k biţi se caracterizează prin: (i) primul bit este folosit pentru codificarea semnului (0 pentru valori pozitive respectiv 1 pentru valori negative); (ii) ceilalţi $(k-1)$ biţi sunt utilizaţi pentru reprezentarea în baza 2 a valorii (cifrele binare corespunzătoare în cazul numerelor pozitive, respectiv valori complementate după regula specifică în cazul numerelor negative). *Exemplu:* Pentru $k = 8$ reprezentarea lui 12 este 00001100 iar reprezentarea lui -12 este 11110100 (în cazul valorilor negative, şirul cifrelor binare este parcurs începând cu cifra cea mai puţin semnificativă până la întâlnirea primei valori egale cu 1 - toate cifrele binare parcurse sunt lăsate nemodificate, iar toate cele care vor fi parcurse ulterior vor fi complementate).

   (a)(2p) Care este cel mai mare şi cel mai mic număr întreg care pot fi reprezentate (în complement faţă de 2) pe 16 poziţii binare (biţi)? Argumentaţi răspunsul.

   (b)(5p) Propuneţi un algoritm care construieşte reprezentarea în complement faţă de 2 pe 16 poziţii binare a unui număr întreg primit ca parametru. *Exemplu:* pentru valoarea 12 se obţine $[0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0]$ iar pentru -12 se obţine $[1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0]$.

   (c)(5p) Propuneţi un algoritm care determină valoarea unui număr întreg (pozitiv sau negativ) pornind de la reprezentarea în complement faţă de 2 pe 16 biţi.

   (d)(8p) Propuneţi un algoritm care calculează suma a două numere întregi date prin reprezentările lor în complement faţă de 2 pe $k = 8$ poziţii binare. Identificaţi cazurile în care se produce *depăşire* (rezultatul nu poate fi reprezentat corect pe $k = 8$ poziţii binare). De exemplu, prin adunarea cifrelor binare aflate pe aceeaşi poziţie (începând de la cea mai puţin semnificativă poziţie) şi transferul reportului către poziţia imediat superioară (ca semnificaţie) pentru $[0,1,1,1,1,0,0,0]$ şi $[0,0,1,1,1,0,0,1]$ s-ar obţine $[1,0,1,1,0,0,0,1]$, ceea ce nu e corect însemnând că s-a produs o depăşire.

   Fiecare dintre algoritmii de mai sus va fi descris în pseudocod şi implementat în Python.

   Rezolvare.

   (a) Cel mai mare număr pozitiv este $2^{15} - 1 = 32627$, întrucât primul bit este folosit pentru semn astfel că pentru reprezentarea valorii propriu-zise rămân 15 biţi. În cazul valorilor negative se poate folosi în plus şirul de biţi $100\ldots0$ corespunzător lui $-2^{15} = -32768$ (întrucât

nu este necesar să fie reprezentat atât $+0$ cât şi -0), deci cel mai mic număr ce poate fi reprezentat pe 16 biţi este -32768.

(b) Se construieşte tabloul $b[0..k-1]$ care conţine cifrele binare obţinute prin conversia în baza doi a valorii absolute a numărului iniţial (cifra cea mai puţin semnificativă se plasează pe poziţia de indice $k-1$). Dacă numărul este negativ se parcurge şirul de cifre binare începând cu ultima cifră (elementul cu indice $k-1$) până la întâlnirea primei cifre egale cu 1 după care cifrele parcurse sunt complementate.

```
dec2bin(n,k)
    b[0..k − 1] ← 0
    i ← k − 1
    while n > 0 do
        bin[i] ← n MOD 2
        n ← n DIV 2
        i ← i − 1
    endwhile
    return bin[0..k − 1]
```

```
codificare(n)
    bin[0..k − 1] ← dec2bin(|n|, k)
    if n < 0 then
        i ← k − 1
// se ignoră toate valorile egale cu 0        while i ≥ 0 and bin[i] == 0 do
            i ← i − 1
        endwhile
// se ignoră primul 1 întâlnit în parcurgerea de la dreapta la stânga        i ← i − 1
        while i ≥ 0 do
            bin[i] ← 1 − bin[i]
            i ← i − 1
        endwhile
    endif
    return bin[0..k − 1]
```

(c) Se analizează bitul de semn (primul element din $bin[0..k-1]$). Dacă acesta este 1 atunci se aplică aceeaşi regulă de complementare după care se calculează valoarea pozitivă corespunzătoare reprezentării în baza doi.

```
decodificare(bin[0..k − 1])
    semn ← 1
    if bin[0] == 1 then
        i = k − 1
        while i ≥ 0 and b[i] == 0 do
            i ← i − 1
        endwhile
        i ← i − 1
        while i ≥ 0 do
            bin[i] ← 1 − bin[i]
            i ← i − 1
        endwhile
        semn ← −1
```

endif
$n \leftarrow bin[0]$
for $i \leftarrow 1, k-1$ **do**
  $n \leftarrow 2 * n + bin[i]$
return $n * semn$

(d) Se parcurg tablourile corespunzătoare celor două valori întregi ($t1[0..k-1]$, $t2[0..k-1]$) de la ultimul element către primul şi se adună elementele corespondente (împreună cu eventualul report de la pasul anterior) reţinând restul împărţirii la 2 ca bit în sumă iar câtul împărţirii la 2 ca valoare nouă pentru report. Regula se aplică pentru toţi biţii inclusiv pentru bitul de semn. Dacă termenii au semne contrare atunci nu se poate produce depăşire. Dacă au acelaşi semn şi după aplicarea regulii de calcul bitul de semn este schimbat atunci înseamnă că s-a produs depăşire.

adunare($t1[0..k-1]$, $t2[0..k-1]$, k)
  $report \leftarrow 0$
  $i \leftarrow k-1$
  $semn1 \leftarrow t1[0]$; $semn2 \leftarrow t2[0]$
  while $i \geq 0$ **do**
    $s \leftarrow t1[i] + t2[i] + report$
    $suma[i] \leftarrow s$ MOD 2
    $report \leftarrow s$ DIV 2
    $i \leftarrow i - 1$
  endwhile
  if $semn1 == semn2$ AND $semn1 \neq= suma[0]$ **then return** "Depasire"
  else return $suma[0..k-1]$

2. *Aproximarea funcţiei logaritmice prin serii.* Funcţia ln poate fi aproximată folosind următoarele serii:

$$f(x) = \begin{cases} \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}(x-1)^n & 0 < x \leq 1 \\ \sum_{n=1}^{\infty} \frac{1}{n}\left(\frac{x-1}{x}\right)^n & x > 1 \end{cases}$$

   (a)(2p) Identificaţi regula de calcul a termenului următor (T_{n+1}) din fiecare serie folosind valoarea termenului curent ($T_n = \frac{(-1)^{n+1}}{n}(x-1)^n$ respectiv $T_n = ((x-1)/x)^n/n$).

   (b)(8p) Pentru fiecare dintre cele două serii descrieţi algoritmul (şi funcţiile Python corespunzătoare) care aproximează suma seriei prin suma finită $T_1 + T_2 + \ldots + T_k$. Numărul de termeni din sumă se stabileşte în funcţie de valoarea ultimului termen adăugat (T_k este primul termen cu proprietatea că $|T_k| < \epsilon$, ϵ fiind o constantă cu valoare mică). Date de test: $\epsilon = 10^{-5}$, $x = 0.5$, $x = 1$,$x = 5$, $x = 10$. Determinaţi în fiecare caz numărul de termeni incluşi în sumă.

   (c)(5p) Pentru unul dintre algoritmii propuşi la punctul (b) (la alegere) identificaţi un invariant şi demonstraţi corectitudinea algoritmului.

   (d)(5p) Descrieţi un algoritm pentru estimarea erorii de aproximare $\sum_{i=1}^{m}(f(i \cdot h) - ln(i \cdot h))^2$. Date de test: $m = 100$, $h = 5/m$.

Rezolvare.

(a) Regulile de calcul a lui T_{n+1} în funcţie de T_n sunt:

- $T_{n+1} = -T_n \frac{n(x-1)}{n+1}$

- $T_{n+1} = T_n \dfrac{n(x-1)}{x(n+1)}$

(b) Ideea este să se pornească de la primul termen şi atât timp cât modulul termenului curent este mai mare decât ϵ se adaugă la sumă şi se construieşte un nou termen. Prelucrarea repetitivă se opreşte în momentul în care modulul ultimului termen adăugat este mai mic decât ϵ. De exemplu, pentru prima serie:

sumaSerie1(x, *epsilon*)
  $k \leftarrow 1$
  $T \leftarrow x - 1$
  $S \leftarrow T$
  while $abs(T) \geq epsilon$ **do**
    $T \leftarrow -T * (x-1) * k/(k+1)$
    $S \leftarrow S + T$
    $k \leftarrow k + 1$
  endwhile
  return S, k

Algoritmul pentru calculul celei de a doua serii este similar (doar regula de calcul a termenului curent diferă).

(c) Pentru algoritmul de calcul a sumei primei serii, precondiţia este $x \in (0, 1]$ iar postcondiţia este $S = \sum_{i=1}^{k} T_k$ cu proprietatea că $|T_k| < \epsilon$ iar $|T_{k-1}| \geq \epsilon$. Un candidat pentru invariant este următorul set de trei relaţii $I = \{S = \sum_{i=1}^{k} T_i, T_i = \frac{(-1)^{i+1}}{i}(x-1)^i, |T_{k-1}| \geq \epsilon\}$. Pentru a demonstra corectitudinea prelucrării repetitive verificăm cele trei condiţii:

(i) *Proprietatea invariantă este adevărată la intrarea în ciclu.* La intrarea în ciclul **while** starea algoritmului este $\{k = 1, T = x - 1, S = T\}$. In ipoteza că $|x - 1| \geq \epsilon$ cele trei relaţii din I sunt satisfăcute.

(ii) *Proprietatea invariantă rămâne adevărată prin execuţia corpului ciclului.* La intrarea în ciclu $S = \sum_{i=1}^{k} T_i$ şi $|T_k| \geq \epsilon\}$. Prin execuţia instrucţiunii $T \leftarrow -T * (x-1) * k/(k+1)$ variabila T va conţine termenul T_{k+1}, iar prin execuţia instrucţiunii $S \leftarrow S + T$ în S va fi $\sum_{i=1}^{k+1} T_i$. După incrementarea lui k, T va conţine pe T_k iar S pe $\sum_{i=1}^{k} T_i$, astfel că cele trei relaţii sunt din nou satisfăcute.

(iii) *La ieşirea din ciclu I implică postcondiţia.* La ieşirea din ciclu $S = \sum_{i=1}^{k} T_i$ iar $|T_k| < \epsilon$ (din condiţia de ieşire). In plus $|T_{k-1}| \geq \epsilon$ (altfel s-ar fi ieşit din ciclu la etapa anterioară).

Finitudinea algoritmului rezultă din faptul că şirul $|T_n(x)|$ este strict descrescător (pentru $x \neq 1$), deci pentru orice $\epsilon > 0$ există n_ϵ cu proprietatea că $|T_{n_\epsilon}| < \epsilon$. In aceste condiţii funcţia de terminare (în cazul în care contorul implicit al ciclului este notat cu p) este $F(p) = n_\epsilon - k_p$. F este strict descrescătoare (întrucât $k_{p+1} = k_p + 1$), este strict pozitivă (întrucât dacă $|T_{k_p}| \geq \epsilon$ înseamnă că $k < n_\epsilon$), iar când $F(p) = 0$ înseamnă că $k_p = n_\epsilon$ deci $|T_{k_p}| < \epsilon$, adică condiţia de continuare devine falsă.

(d) Este suficient să se calculeze suma apelând una din funcţiile de estimare a sumei (sumaSerie1 sau sumaSerie2):

eroareAproximare(m,h,*epsilon*)
  $E \leftarrow 0$
  for $i \leftarrow 1, m$ **do**

if $i * h \leq 1$ **then** $S \leftarrow sumaSerie1(i * h, epsilon)$
else $S \leftarrow sumaSerie2(i * h, epsilon)$
endif
$E \leftarrow E + (S - ln(i * h))^2$
endfor
return E

3. Se consideră algoritmii `alg1` şi `alg2`. Pentru fiecare dintre cei doi algoritmi:

   (a)(5p+5p) Implementaţi algoritmul în Python.

   (b)(5p+5p) Stabiliţi ce returnează fiecare dintre algoritmi atunci când este apelat pentru valori naturale nenule ale parametrilor. Identificaţi o proprietate invariantă şi demonstraţi corectitudinea fiecărui algoritm. *Indicaţie:* pentru `alg2` se poate folosi proprietatea că pentru orice număr natural nenul n există un număr natural $k > 0$ astfel încât $2^{k-1} \leq n < 2^k$.

```
1: alg1(int a, b)
2: if a < b then
3:     a ↔ b
4: end if
5: c ← 0
6: d ← a
7: while d > b do
8:     c ← c + 1
9:     d ← d − b
10: end while
11: return c, d
```

```
1: alg2(int n)
2: i ← 1
3: x ← 0
4: while i <= n do
5:     i ← 2 * i
6:     x ← x + 1
7: end while
8: return x
```

Rezolvare.

(a) Algoritmul returnează perechea de valori (c, d) care satisface condiţia $a = b * c + d$ cu $0 < d \leq b$, $a \geq b$. *Observaţie.* Dacă condiţia de continuare a ciclului **while** ar fi fost $d \geq b$ atunci algoritmul ar fi returnat câtul şi restul împărţirii întregi a maximului dintre a şi b la minimul dintre a şi b.

O proprietate invariantă este $I = \{a = b * c + d\}$. Verificarea celor trei condiţii care permit demonstrarea corectitudinii folosind proprietatea invariantă:

(i) *Proprietatea invariantă este adevărată la intrarea în ciclu.* La intrarea în ciclul **while** starea algoritmului este $\{a \geq b, c = 0, d = a\}$ deci $a = b * c + d$, adică proprietatea I este adevărată.

(ii) *Proprietatea invariantă rămâne adevărată prin execuţia corpului ciclului.* După execuţia instrucţiunii $c \leftarrow c+1$ se obţine că $a = b*c+d-b$ iar după execuţia instrucţiunii $d \leftarrow d-b$ se ajunge din nou la $a = b * c + d$.

(iii) *La ieşirea din ciclu I implică postcondiţia.* La ieşirea din ciclu $a = b*c+d$ iar $0 < d \leq b$ (din condiţia de ieşire), deci e satisfăcută postcondiţia.

Finitudinea algoritmului poate fi demonstrată folosind ca funcţie de terminare $F(p) = H(d_p - b)$ unde H e o funcţie cu proprietatea că $H(u) = u$ dacă $u > 0$ şi $H(u) = 0$ dacă $u \leq 0$.

Pentru $d_p - b > 0$ funcţia F este strict descrescătoare iar când devine 0 implică faptul că $d_p \leq b$ deci condiţia de continuare devine falsă.

(b) Algoritmul returnează numărul de poziţii binare pe care poate fi reprezentată valoarea naturală n (fără semn) adică $\lfloor \log_2(n) \rfloor + 1$. O proprietate invariantă este $i = 2^x$. Verificarea celor trei condiţii care permit demonstrarea corectitudinii folosind proprietatea invariantă:

(i) *Proprietatea invariantă este adevărată la intrarea în ciclu.* La intrarea în ciclul **while** starea algoritmului este $\{i = 1, x = 0\}$ deci $i = 1 = 2^0 = 2^x$, adică proprietatea I este adevărată.

(ii) *Proprietatea invariantă rămâne adevărată prin execuţia corpului ciclului.* După execuţia instrucţiunii $i \leftarrow 2 * i$ se obţine că $i = 2^{x+1}$ iar după execuţia instrucţiunii $x \leftarrow x + 1$ se ajunge din nou la $i = 2^x$.

(iii) *La ieşirea din ciclu, I implică postcondiţia.* La ieşirea din ciclu $i > n$ adică $2^x > n$, deci $2^{x-1} \leq n < 2^x$ adică $x - 1 \leq \log_2(n) < x$ de unde rezultă că $x - 1 = \lfloor \log_2(n) \rfloor$, adică $x = \lfloor \log_2(n) \rfloor + 1$ (postcondiţia).

Finitudinea algoritmului poate fi demonstrată folosind ca funcţie de terminare $F(p) = H(n - i_p)$ unde H e o funcţie cu proprietatea că $H(u) = u$ dacă $u \geq 0$ şi $H(u) = 0$ dacă $u < 0$. Pentru $i_p \leq n$ funcţia F este strict descrescătoare iar când devine 0 implică faptul că $i_p > n$ deci condiţia de continuare devine falsă.

4. Se consideră un tablou $x[1..n]$ şi se doreşte construirea unui tablou $m[1..n]$ care conţine pe poziţia i media aritmetică a elementelor din subtabloul $x[1..i]$ $(m[i] = (x[1] + \ldots + x[i])/i)$.

   (a)(5p) Propuneţi un algoritm de complexitate $\Theta(n^2)$ pentru construirea tabloului m. Justificaţi faptul că algoritmul are complexitatea cerută şi implementaţi algoritmul în Python.

   (b)(10p) Propuneţi un algoritm de complexitate $\Theta(n)$ pentru construirea tabloului m. Justificaţi faptul că algoritmul are complexitatea cerută şi implementaţi algoritmul în Python.

Rezolvare.

(a) Pentru fiecare poziţie i se calculează suma $x[1] + x[2] + ... + x[i]$ după care se împarte la i (algoritmul conţine două cicluri suprapuse). Pentru analiza complexităţii se stabileşte: (i) dimensiunea problemei este n; (ii) operaţia dominantă este adunarea $(m[i] + x[j])$; (iii) numărul de execuţii ale operaţiei dominante este: $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = n(n+1)/2$ deci $T(n) \in \Theta(n^2)$.

```
def medie1(x):
    n=len(x)
    m=[0]*n
    for i in range(n):
        for j in range(i+1):
            m[i]=m[i]+x[j]
        m[i]=m[i]/(i+1)
    return m
```

(b) Se iniţializează tabloul m cu tablul x după care, începând cu al doilea element se modifică valorile lui m folosind relaţia $m[i] = m[i-1] + m[i]$ (algoritmul conţine două cicluri independente). Pentru analiza complexităţii se stabileşte: (i) dimensiunea problemei este n;

(ii) operaţia dominantă este fie adunarea ($m[i-1]+m[i]$) fie înmulţirea ($m[i]/(i+1)$); (iii) numărul de execuţii ale operaţiei dominante este $T(n)=n-1$. Dacă se contorizează ambele operaţii se obţine $T(n)=2(n-1)$. In oricare dintre cazuri: $T(n)\in\Theta(n)$.

```
def medie2(x):
    n=len(x)
    m=x
    for i in range(1,n):
        m[i]=m[i-1]+m[i]
    for i in range(1,n):
        m[i]=m[i]/(i+1)
    return m
```

5. 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)(5p) 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.

   (b)(10p) 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.

   (c)(10p) 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 $\mathcal{O}(n)$.

   (d)(10p) 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.

Rezolvare.

(a) Se analizează toate tripletele $(a[i],b[j],c[k])$ pentru $i=\overline{1,n}$, $j=\overline{1,n}$, $k=\overline{1,n}$ şi la întâlnirea primului triplet de elemente identice se returnează True. La ieşirea din prelucrările repetitive se va returna False. Algoritmul va conţine trei cicluri suprapuse, iar numărul de comparaţii efectuate, $T(n)$, satisface $T(n)\geq 1$ şi $T(n)\leq\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}1=n^3$ deci ordinul de complexitate este $\mathcal{O}(n^3)$.

```
def elemcomun1(a,b,c):
    n=len(a)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                if a[i]==b[j] and b[j]==c[k]:
```

```
                    return True
    return False
```

(b) Se determină elementele comune din a şi b analizând toate perechile de forma $(a[i], b[j])$ pentru $i = \overline{1,n}$ şi $j = \overline{1,n}$ (numărul elementelor comune, k, va fi cel mult egal cu n). Se compară elementele din lista construită la pasul anterior cu toate elementele din tabloul c. Pentru stabilirea ordinului de complexitate se consideră dimensiunea problemei egală cu n iar operaţia dominantă este comparaţia. Construirea listei cu elemente comune necesită n^2 comparaţii iar a doua analiză necesită $nk < n^2$ comparaţii. Ordinul de complexitate este $\Theta(n^2)$.

```
def elemcomun2(a,b,c):
    n=len(a)
    ab=[]
    for i in range(n):
        for j in range(n):
            if a[i]==b[j]:
                    ab.append(a[i])
    k=len(ab)
    for i in range(k):
        for j in range(n):
            if ab[i]==c[j]:
                return True
    return False
```

(c) Dacă valorile din tablourile a, b şi c sunt din mulţimea $\{1, 2, \ldots, m\}$ înseamnă că pot fi utilizate tablouri de frecvenţe. In varianta cea mai simplă se utilizează trei tablouri de frecvenţe, câte unul pentru fiecare tablou iniţial. Construirea fiecărui tablou de frecvenţe are ordinul de complexitate $\Theta(n)$. La final se parcurg în paralel cele trei tablouri şi la întâlnirea unei poziţii în care toate cele trei tablouri de frecvenţe conţin valori nenule se returnează True. Ordinul de complexitate al etapei de parcurgere a tabelelor de frecvenţe este $\mathcal{O}(m)$. *Observaţie.* Se poate utiliza un singur tablou cu indicatori de prezenţă care să conţină informaţii despre toate cele trei tablouri. De exemplu se consideră indicatori de prezenţă cu maxim 3 cifre din $\{0, 1\}$: cifra unităţilor e asociată cu tabloul a, cifra zecilor cu tabloul b iar cea a sutelor cu tabloul c. Astfel, o valoare egală cu 10 indică faptul că elementul corespunzător se află doar în tabloul b pe când o valoare egală cu 101 indică faptul că elementul se află în a şi c.

```
def elemcomun3(a,b,c,m):
    n=len(a)
    fa=[0]*m
    fb=[0]*m
    fc=[0]*m
    for i in range(n):
        fa[a[i]-1]=fa[a[i]-1]+1
        fb[b[i]-1]=fb[b[i]-1]+1
        fc[c[i]-1]=fc[c[i]-1]+1
    for i in range(m):
        if fa[i]>0 and fb[i]>0 and fc[i]>0:
            return True
    return False
```

(d) In cazul în care tablourile sunt ordonate crescător se pot 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
```

6. Se consideră un tablou, $x[1..n]$, ordonat crescător, v o valoare de acelaşi tip ca elementele tabloului şi algoritmul `alg`.

1: `alg`($x[1..n]$,v)
2: $i \leftarrow n$
3: **while** $i \geq 1$ and $v < x[i]$ **do**
4: $\quad i \leftarrow i - 1$
5: **end while**
6: **return** $i + 1$

(a)(5p) Implementaţi algoritmul `alg` în Python şi stabiliţi ce returnează.

(b)(5p) Estimaţi numărul *mediu* de comparaţii efectuate (în ipoteza în care toate clasele de date de intrare au aceeaşi probabilitate de apariţie).

Rezolvare.

(a) Algoritmul returnează poziţia pe care poate fi inserată valoarea v în tabloul x astfel încât acesta să rămână ordonat crescător.

```
def alg(x,v):
    n=len(x)
    i=n-1
    while i>=0 and v<x[i]:
        i=i-1
    return i+1
```

(b) Numărul de comparaţii efectuate ($v < x[i]$) este cuprins între 1 şi n. Dacă fiecare din cele n clase de date de intrare are aceeaşi probabilitate de apariţie ($1/n$) atunci numărul mediu de comparaţii este $T(n) = (1 + 2 + \ldots + n)/n = (n + 1)/2$.