Tema 1: Rezolvarea algoritmică a problemelor. Descrierea algoritmilor în pseudocod. Verificarea corectitudinii algoritmilor. Stabilirea ordinului de complexitate.

1. *Operaţii exacte cu numere raţionale.* Un număr raţional poate fi întotdeauna specificat ca o fracţie ireductibilă (raportul a două numere întregi prime între ele). Se pune problema definirii unor funcţii corespunzătoare operaţiilor aritmetice (adunare, scădere, înmulţire, împărţire) care să permite obţinerea unor rezultate exacte (atât operanzii cât şi rezultatele vor fi specificate ca perechi de valori care definesc o fracţie ireductibilă). De exemplu pentru operaţia 1/2+3/4, funcţia de adunare va prelua numărătorul şi numitorul fiecărei fracţii (adică patru numere întregi: 1,2,3,4) şi va returna numărătorul şi numitorul rezultatului (5 şi 4).

   (a) Descrieţi un algoritm care primeşte ca parametri numărătorul şi numitorul unei fracţii şi returnează valorile corespunzătoare fracţiei ireductibile echivalente (pentru parametrii 6 şi 9 se vor returna valorile 2 şi 3).

   (b) Descrieţi algoritmi corespunzători operaţiilor de adunare, scădere, înmulţire şi împărţire a fracţiilor. Fiecare algoritm va primi ca parametri de intrare numărătorul şi numitorul fiecărei fracţii şi va returna numărătorul şi numitorul rezultatului.

   (c) Descrieţi un algoritm de calcul a valorii unui polinom cu coeficienţi întregi pentru un argument număr raţional. Se consideră că polinomul este reprezentat prin tabloul coeficienţilor (un polinom $c_nX^n + c_{n-1}X^{n-1} + \ldots c_1X + c_0$ este reprezentat printr-un tablou $c[0..n]$). De exemplu pentru polinomul $2X^3 - X^2 + X + 2$ valoarea calculată în 1/2 este 5/2. Pentru toate operaţiile aritmetice se vor folosi funcţiile descrise la punctul (b). `Indicaţie.` Se poate folosi ca punct de pornire algoritmul de la Problema 12/Seminar 4.

   (d) Descrieţi un algoritm care determină mulţimea tuturor rădăcinilor raţionale ale unui polinom cu coeficienţi numere întregi. *Indicaţie:* rădăcinile raţionale ale unui polinom cu coeficienţi întregi au proprietatea că sunt de forma d_0/d_n unde d_0 este un divizor (pozitiv sau negativ) al termenului liber (c_0) iar d_n este un divizor al coeficientului termenului de grad maxim (c_n). Mulţimea divizorilor luaţi în considerare include pe 1 şi numărul însuşi. Nu toate numerele raţionale de forma descrisă sunt rădăcini (trebuie selectate doar cele pentru care valoarea polinomului este 0).

Pentru toţi algoritmii descrişi mai sus se vor implementa funcţii Python corespunzătoare.

Rezolvare.

(a) Pentru cei doi parametri, a şi b, se determină cel mai mare divizor comun, d şi se returnează valorile obţinute împărţind a la d respectiv b la d. Orice variantă corectă de determinare a celui mai mare divizor comun este acceptată.

cmmdc(a,b)
  $deimpartit \leftarrow a$; $impartitor \leftarrow b$; $rest \leftarrow deimpartit$ MOD $impartitor$
  while $rest \neq 0$ **do**
    $deimpartit \leftarrow impartitor$
    $impartitor \leftarrow rest$
    $rest \leftarrow deimpartit$ MOD $impartitor$
  endwhile
  return $impartitor$

simplificare(a,b)
  $d \leftarrow$ cmmdc(a,b)
  return a/d, b/d

(b) Pentru fiecare operaţie se aplică regula de calcul corespunzătoare după care se apelează funcţia de simplificare. Reguli de calcul:

adunare: $a/b + c/d = (a * d + b * c)/(c * d)$

scădere: $a/b - c/d = (a * d - b * c)/(c * d)$

înmulţire: $(a/b)/(c/d) = (a * c)/(b * d)$

împărţire: $(a/b)/(c/d) = (a * d)/(b * c)$

Exemplu pentru adunare:

adunare(a,b,c,d)
  $numarator \leftarrow a * d + b * c$
  $numitor \leftarrow c * d$
  return simplificare($numarator$, $numitor$)

(c) Pentru calculul valorii unui polinom se poate folosi ideea de la schema lui Horner. Coeficienţii polinomului fiind numere întregi vor fi interpretaţi ca fracţii cu numitorul egal cu 1 (de exemplu coeficientul $c[i]$ va fi reprezentat prin perechea $(c[i], 1)$).

evaluarePolinom($c[0..n]$,a,b)
  $rezNumarator \leftarrow c[n]$; $rezNumitor \leftarrow 1$; $i \leftarrow n - 1$
  while $i \geq 0$ **do**
    $numarator, numitor \leftarrow inmultire(rezNumarator, rezNumitor, a, b)$
    $rezNumarator, rezNumitor \leftarrow adunare(numarator, numitor, c[i], 1)$
    $i \leftarrow i - 1$
  endwhile
  return simplificare($rezNumarator$,$rezNumitor$)

(d) Se generează toate fracţiile ireductibile obţinute utilizând toţi divizorii termenului liber (pozitivi şi negativi) cu rol de numărător şi toţi divizorii pozitivi ai coeficientului termenului de grad maxim cu rol de numitor. Pentru fiecare fracţie astfel generată se evaluează

polinomul folosind algoritmul de la punctul (c) iar în cazul în care valoarea corespunzătoare numărătorului este nulă se reţine (sau se afişează) perechea de valori corespunzătoare fracţiei.

```
radaciniPolinom(c[0..n])
    i ← 0
    for d ← 1, |c[0]| do
        if c[0] MOD d = 0
            then for e ← 1, |c[n]| do
                    if |c[n]| MOD e = 0
                        rezNumarator, rezNumitor ← evaluarePolinom(c, d, e)
                        if rezNumarator = 0 then i ← i + 1; radacini[i] ← (d, e) endif
                        rezNumarator, rezNumitor ← evaluarePolinom(c, −d, e)
                        if rezNumarator = 0 then i ← i + 1; radacini[i] ← (−d, e) endif
                    endif
                endfor
        endif
    endfor
```

2. *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 (toate zerourile de la sfârşit şi primul 1 întâlnit) sunt lăsate nemodificate, iar toate cele care vor fi parcurse ulterior vor fi complementate).

   (a) 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) 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) 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) 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 ← bin[0]
    for i ← 1, k − 1 do
      n ← 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 ← 0
    i ← k − 1
    semn1 ← t1[0]; semn2 ← t2[0]
    while i ≥ 0 do
      s ← t1[i] + t2[i] + report
      suma[i] ← s MOD 2
      report ← s DIV 2
      i ← i − 1
    endwhile
    if semn1 == semn2 AND semn1 ≠= suma[0] then return "Depasire"
    else return suma[0..k − 1]
```

3. *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) 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) 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) Pentru unul dintre algoritmii propuşi la punctul (b) (la alegere) identificaţi un invariant şi demonstraţi corectitudinea algoritmului.

   (d) 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:

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

(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 ← 1
    T ← x − 1
    S ← T
    while abs(T) ≥ epsilon do
        T ← −T ∗ (x − 1) ∗ k/(k + 1)
        S ← S + T
        k ← 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 ← 0
    for i ← 1, m do
        if i * h ≤ 1 then S ← sumaSerie1(i * h, epsilon)
            else S ← sumaSerie2(i * h, epsilon)
        endif
        E ← E + (S − ln(i * h))^2
    endfor
    return E
```

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

   (a) Implementaţi fiecare algoritm în Python.

   (b) 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.

5. Se consideră următoarea relaţie de recurenţă:

$$\begin{cases} x_0 & = & a \\ x_{k+1} & = & \frac{1}{3}\left(2x_k + \frac{a}{x_k^2}\right), \quad k > 0 \end{cases}$$

(a) Propuneţi un algoritm care, pentru o valoare dată a, afişează valorile x_0, x_1, ..., x_k până când este îndeplinită condiţia $|x_k - x_{k-1}| < \epsilon$. Descrieţi algoritmul în pseudocod, implementaţi-l în Python şi testaţi-l pentru $\epsilon = 0.00001$ şi $a = 8$, $a = -27$, $a = 64$.

(b) Ce returnează algoritmul pentru o valoare a?

(c) Propuneţi o proprietate invariantă cu ajutorul căreia să se demonstreze că algoritmul propus returnează valoarea specificată la punctul (b).

6. 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
```

7. 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) Implementaţi algoritmul `alg` în Python şi stabiliţi ce returnează.

(b) Identificaţi un invariant pentru prelucrarea repetitivă şi demonstraţi că algoritmul este corect (inclusiv finit).

(c) 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) In cazul în care $v \geq x[n]$ atunci nu se intră în prelucrarea iterativă şi algoritmul retunează $n+1$, adică poziţia pe care poate fi plasat v astfel încât vectorul să rămână ordonat crescător. Dacă $v < x[n]$atunci se intră în prelucrarea repetitivă, iar proprietatea invariantă este $v < x[j]$ for $j \in \{i+1, i+2, \ldots, n\}$.

(c) 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 $T(n) = (1 + 2 + \ldots + n)/n = (n + 1)/2$.

8. Se consideră următoarea tehnică de compresie a unui şir de litere: "dacă o literă este precedată şi urmată de o altă literă atunci ea este transferată ca atare în textul comprimat; dacă o literă apare pe mai multe poziţii consecutive atunci în textul comprimat subsecvenţa care conţine aceeaşi litera va fi înlocuită cu o pereche formată din litera respectivă şi numărul de repetări". De exemplu, textul "urrrraa" se va înlocui cu "ur4a2". Descrieţi ideea de rezolvare şi implementaţi în Python câte o funcţie pentru:

   (a) compresia unui şir de litere;

   (b) decompresia pornind de la un şir "comprimat".

Rezolvare. (a) O variabilă contor (iniţializată cu 1) este incrementată ori de câte ori caracterul de pe poziţia urmţoare este identic cu cel de pe poziţia curentă şi se setează din nou pe 1 dacă pe poziţia următoare în şirul iniţial se află un caracter diferit. La întâlnirea unui caracter diferit de cel curent, valoarea contorului se salvează (înainte de resetare) în tabloul rezultat după care se plasează şi noul caracter. Un exemplu de implementare este:

```
def compresie(sir):
    n = len(sir)
    c = [sir[0]]
    nr = 1
    for i in range (n-1):
        if sir[i] != sir[i+1]:
            c.append(str(nr))
            c.append(sir[i+1])
            nr = 1
        else:
            nr = nr+1
    c.append(str(nr))
    return c
```

(b) Se parcurge tabloul cu şirul "comprimat" şi pentru fiecare caracter întâlnit pe poziţiile 0, 2, 4 ... în tabloul cu varianta comprimată se pun atâtea copii ale acestuia câte sunt specificate pe poziţia următoare din tablou (1, 3, 5 ...). Un exemplu de implementare este:

```
def decompresie(c):
    n = len(c)
    sir = ""
    i = 0
    while i < n-1:
        for j in range(int(c[i+1])):
            sir = sir + str(c[i])
        i = i+2
    return sir
```

9. (10p) Implementaţi în Python un generator aleator de adrese IP de forma "a.b.c.d" unde fiecare componentă este un număr natural cuprins între 0 şi 255.
   Indicaţie: se pot folosi funcţiile din modulul (pachetul) `random` din Python. Pentru fiecare dintre cele 4 componente se apelează funcţia `randint`. Pentru a stabili ordinul de complexitate al generatorului ar trebui stabilită operaţia dominantă. Având în vedere că prelucrarea cea mai costisitoare este efectuată de către funcţia de generare a unei valori aleatoare. Pentru a estima costul acesteia în raport cu o operaţie elementară, se calculează raportul dintre durata medie de execuţie a apelului funcţiei `randint` şi durata medie de execuţie a unei operaţii elementare (de exemplu înmulţire). Mediile se calculează pentru cel puţin 100 de repetări ale operaţiei analizate calculând diferenţa dintre apeluri ale funcîei `time.perf_counter()`. De exemplu estimarea duratei aferente unui apel al generatorului de numere aleatoare poate fi realizată prin:

```
import random, time
def profilareRand(n):
    tmed=0
    for i in range(n):
        start = time.perf_counter()
        r = random.randint(0,255)
        stop = time.perf_counter()
        tmed = tmed+(stop-start)
    return (tmed/n)
```