ALGORITMICĂ. Seminar 3: Analiza eficienţei algoritmilor - estimarea timpului de execuţie şi notaţii asimptotice.

Problema 1 (L) Să se determine numărul de operaţii efectuate de către un algoritm care determină numărul de inversiuni ale unei permutări.

Rezolvare. Pentru o permutare $(a_1, \ldots, a_n)$, $a_i \in \{1, \ldots, n\}$, $i = \overline{1, n}$ o inversiune este o pereche de indici (i, j) cu proprietatea că $i < j$ şi $a_i > a_j$. Considerând permutarea ca fiind reprezentată prin tabloul $a[1..n]$ numărul de inversiuni poate fi determinat prin algoritmul `inversiuni` descris în Tabelul 1.

```
inversiuni(integer
a[1..n])
1: k ← 0
2: for i ← 1, n − 1 do
3:    for j ← i + 1, n do
4:       if a[i] > a[j] then
5:          k ← k + 1
6:       end if
7:    end for
8: end for
9: return k
```

Linie	Cost operaţie	Repetări
1	1	1
2	$2n$	1
3	$2(n-i+1)$	$i = \overline{1, n-1}$
4	1	$i = \overline{1, n-1}$, $j = \overline{i+1, n}$
5	τ	$i = \overline{1, n-1}$, $j = \overline{i+1, n}$

Table 1: Algoritmul pentru determinarea numărului de inversiuni şi tabelul costurilor

Dimensiunea problemei este reprezentată de numărul de elemente ale tabloului a. Numărul (costul) operaţiilor elementare executate pentru fiecare linie a algoritmului este detaliat în tabelul de costuri (pentru liniile corespunzătoare instrucţiunilor **for** este specificat costul gestiunii contorului corespunzător tuturor execuţiilor corpului ciclului). τ reprezintă numărul de incrementări ale lui k. Acesta depinde de proprietăţile datelor de intrare şi poate fi 0 (operaţia nu se execută) respectiv 1 (operaţia se execută). Însumând costurile operaţiilor elementare se obţine:

$$T(n) = 1 + 2n + 2\sum_{i=1}^{n-1}(n-i+1) + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \tau = \frac{3n(n-1)}{2} + 4n - 1 + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \tau$$

deci

$$\frac{3n^2 + 5n - 2}{2} \leq T(n) \leq 2n^2 + 2n - 1.$$

Observaţie. In practică nu este necesară contorizarea tuturor operaţiilor ci este suficient să se determine ordinul de mărime al numărului de operaţii, motiv pentru care este suficient să se contorizeze operaţia care induce costul cel mai mare. De regulă aceasta este operaţia care se repetă cel mai frecvent (numită operaţie dominantă). În cazul exemplului de mai sus, aceasta este comparaţia care apare pe linia 4, care se execută de $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2$. Deci $T(n) = n(n-1)/2$, adică $T(n) \in \Theta(n^2)$.

Problema 2 (S+L) Să se analizeze eficienţa unui algoritm care verifică dacă un tablou este ordonat crescător sau nu atât în cazurile extreme (cel mai favorabil, cel mai defavorabil) cât şi în cel mediu.

Rezolvare. Rezultatul algoritmului se colectează într-o variabilă booleană ce va conţine **true** dacă $a[i] \leq a[i+1]$ pentru $i = \overline{1, n-1}$ respectiv **false** dacă există i cu $a[i] > a[i+1]$. Algoritmul şi tabelul costurilor sunt prezentate în Tabel 2.

Cazul cel mai favorabil (din punctul de vedere al numărului de operaţii executate) corespunde situaţiei în care $a[1] > a[2]$ ceea ce conduce la $\tau_1(n) = 1$ iar $\tau_2(n) = 1$. Numărul de operaţii efectuate în acest caz este

```
ordonat(a[1..n])
1: r ← true
2: i ← 0
3: while (i < n − 1) and (r =true)
   do
4:    i ← i + 1
5:    if a[i] > a[i + 1] then
6:       r ← false
7:    end if
8: end while
9: return r
```

Linie	Cost operaţiei	Repetări
1	1	1
2	1	1
3	3	$\tau_1(n) + 1$
4	1	$\tau_1(n)$
5	1	$\tau_1(n)$
6	1	$\tau_2(n)$

Table 2: Algoritm pentru a verifica dacă un tablou este ordonat crescător şi costurile operaţiilor

$2 + 3 \cdot 2 + 3 = 10$. Cazul cel mai defavorabil este cel în care tabloul este ordonat crescător. În acest caz $\tau_1(n) = n - 1$ iar $\tau_2(n) = 0$, deci numărul de operaţii executate este: $2 + 3n + 2(n - 1) = 5n$. Prin urmare $10 \leq T(n) \leq 5n$. Dependenţa timpului de execuţie de dimensiunea problemei în cazul cel mai favorabil este diferită de cea corespunzătoare cazului cel mai defavorabil. Prin urmare nu putem spune despre $T(n)$ că aparţine lui $\Theta(n)$ ci doar că $T(n) \in \Omega(1)$ şi $T(n) \in \mathcal{O}(n)$.

În cazul în care considerăm comparaţia din interiorul ciclului ca fiind operaţia dominantă şi facem abstracţie de celelalte operaţii obţinem că $1 \leq T(n) \leq n - 1$.

Să analizăm cazul mediu în ipoteza că există n clase distincte de date de intrare. Clasa C_i ($i \in \{1, \ldots, n-1\}$) corespunde vectorilor pentru care prima pereche de valori consecutive care încalcă condiţia de ordonare crescătoare este $(a[i], a[i + 1])$. Când se ajunge la $i = n$ considerăm că tabloul este ordonat crescător. Numărul de comparaţii corespunzător clasei i este i. Notând cu p_i probabilitatea de apariţie a unei instanţe din clasa C_i rezultă că timpul mediu de execuţie este:

$$T_m(n) = \sum_{i=1}^{n-1} ip_i + (n - 1)p_n$$

Întrucât $\sum_{i=1}^{n} p_i = 1$ şi $i \leq n$ rezultă $T_m(n) \leq n - 1$.

In ipoteza că toate cele n clase au aceeaşi probabilitate de apariţie se obţine următoarea estimare pentru timpul mediu:

$$T_m(n) = \sum_{i=1}^{n-1} \frac{1}{n} i + \frac{n-1}{n} = \frac{1}{n} \cdot \frac{n(n-1) + 2(n-1)}{2} = \frac{(n-1)(n+2)}{2n}$$

În realitate cele n clase nu sunt echiprobabile însă probabilităţile corespunzătoare nu sunt uşor de determinat. Ceea ce se poate observa este faptul că $p_1 \geq p_2 \geq \ldots \geq p_n$ deci pe măsură ce probabilitatea corespunzătoare unei instanţe creşte numărul de operaţii scade. Analiza cazului mediu este în general dificilă, fiind fezabilă doar în cazuri particulare, când se impun ipoteze simplificatoare asupra distribuţiei de probabilitate.

Problema 3 (S) Se consideră un şir de valori întregi (pozitive şi negative - cel puţin un element este pozitiv). Să se determine suma maximă a elementelor unei secvenţe a şirului (succesiune de elemente consecutive).

Rezolvare. Pentru fiecare $i \in \{1, \ldots, n\}$ se calculează succesiv sumele elementelor subtablourilor $a[i..j]$ cu $j \in \{1, \ldots, n\}$ şi se reţine cea maximă. O primă variantă a algoritmului (`secventa1`) este descrisă în continuare.

```
secventa1(integer a[1..n])
1: max ← 0; imax ← 1; jmax ← 0
2: for i ← 1, n do
3:    s ← 0
4:    for j ← i, n do
5:       s ← s + a[j]
6:       if s > max then
7:          max ← s; imax ← i; jmax ← j
8:       end if
9:    end for
10: end for
11: return max, imax, jmax
```

Operaţiile dominante sunt cele specificate pe liniile 5 şi 6, iar numărul de repetări ale acestora este $T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = n(n+1)/2$.

Aceeaşi problemă poate fi rezolvată şi printr-un algoritm de complexitate liniară calculând succesiv suma elementelor din tablou şi reiniţiind calculul în momentul în care suma devine negativă (în acest caz reiniţializând cu 0 variabila s valoarea obţinută este mai mare decât cea curentă, care este negativă). Algoritmul `secventa2` este descris în continuare.

```
secventa2(integer a[1..n])
max ← 0; icrt ← 1; imax ← 1
jmax ← 0
s ← 0
for i ← 1, n do
   s ← s + a[i]
   if s < 0 then
      s ← 0; icrt ← i + 1
   else if s > max then
      max ← s; imax ← icrt; jmax ← i
   end if
end for
return max, imax, jmax
```

Numărul de execuţii ale operaţiei dominante ($s \leftarrow s + a[i]$) este $T(n) = n$.

Problema 4 (S) Să se estimeze timpul de execuţie al unui algoritm care generează toate numerele prime mai mici decât o valoare dată n ($n \geq 2$).

Rezolvare. Vom analiza două variante de rezolvare: (a) parcurgerea tuturor valorilor cuprinse între 2 şi n şi reţinerea celor prime; (b) algoritmul lui Eratostene.

(a) Presupunem că algoritmul `prim(i)` returnează **true** dacă n este prim şi **false** în caz contrar (analizând divizorii potenţiali dintre 2 şi $\lfloor\sqrt{i}\rfloor$).

(b) *Algoritmul lui Eratostene.* Se porneşte de la tabloul $a[2..n]$ iniţializat cu valorile naturale cuprinse între 2 şi n şi se marchează succesiv toate valorile ce sunt multipli ai unor valori mai mici. Elementele rămase nemarcate sunt prime.

Ambele variante sunt descrise în Algoritmul 1.

In ambele cazuri considerăm dimensiunea problemei ca fiind n.

In prima variantă considerăm că operaţiile dominante sunt cele efectuate în cadrul subalgoritmului `prim`. Luăm în considerare doar operaţia de analiză a divizorilor potenţiali. Pentru fiecare i numărul de comparaţii

Algoritmul 1 Algoritmi pentru generarea numerelor prime

```
generare(integer n)
k ← 0
for i ← 2, n do
    if prim(i) =true then
        k ← k + 1
        p[k] ← i
    end if
end for
return p[1..k]
```

```
eratostene(integer n)
for i ← 2, n do
    a[i] ← i
end for
for i ← 2, ⌊√n⌋ do
    if a[i] ≠ 0 then
        j ← i ∗ i
        while j ≤ n do
            a[j] ← 0; j ← j + i
        end while
    end if
end for
k ← 0
for i ← 2, n do
    if a[i] ≠ 0 then
        k ← k + 1; p[k] ← a[i]
    end if
end for
return p[1..k]
```

efectuate este $\lfloor\sqrt{i}\rfloor - 1$. Deci:

$$T_1(n) = \sum_{i=2}^{n}(\lfloor\sqrt{i}\rfloor - 1) \leq \sum_{i=2}^{n}\sqrt{i} - (n-1)$$

Pentru a estima suma de mai sus se aproximează cu integrala $\int_2^n \sqrt{x}dx = 2/3n\sqrt{n} - 4\sqrt{2}/3$ obţinându-se că:

$$T_1(n) \leq 2/3n\sqrt{n} - 4\sqrt{2}/3 - (n-1)$$

Pe de altă parte, întrucât $\sqrt{i} - 1 < \lfloor\sqrt{i}\rfloor \leq \sqrt{i}$, rezultă că $T_1(n) \geq 2/3n\sqrt{n} - 4\sqrt{2}/3 - 2(n-1)$.

In cazul algoritmului lui Eratostene operaţia dominantă poate fi considerată cea de marcare a elementelor tabloului a. Pentru fiecare i se marchează cel mult $\lfloor(n-i^2)/i+1\rfloor$ elemente astfel că numărul total de astfel de operaţii satisface:

$$T_2(n) \leq \sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\lfloor\frac{n-i^2}{i}+1\rfloor \leq \sum_{i=2}^{\lfloor\sqrt{n}\rfloor}(\frac{n-i^2}{i}+1) = n\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\frac{1}{i} - \sum_{i=2}^{\lfloor\sqrt{n}\rfloor} i + \lfloor\sqrt{n}\rfloor - 1$$

Se aplică din nou tehnica aproximării unei sume prin integrala corespunzătoare şi se obţine:

$$T_2(n) \leq n\ln\sqrt{n} - n\ln\sqrt{2} - \sqrt{n}(\sqrt{n}+1)/2 + \sqrt{n}$$

Se observă că varianta bazată pe algoritmul lui Eratostene are un ordin de complexitate mai mic decât prima variantă însă necesită utilizarea unui tablou suplimentar de dimensiune $n-1$.

Problema 5 (S) Să se determine ordinul de creştere al valorii variabilei x (în funcţie de n) după execuţia algoritmilor `alg1`, `alg2`, `alg3`, `alg4`, `alg5` şi `alg6`.

```
alg1(integer n)
1: x ← 0
2: for i ← 1, n do
3:    for j ← 1, i do
4:       for k ← 1, j do
5:          x ← x + 1
6:       end for
7:    end for
8: end for
9: return x
```

```
alg2(integer n)
1: x ← 0
2: i ← n
3: while i ≥ 1 do
4:    x ← x + 1
5:    i ← iDIV2
6: end while
7: return x
```

```
alg3(integer n)
1: x ← 0
2: i ← n
3: while i ≥ 1 do
4:    for j ← 1, n do
5:       x ← x + 1
6:    end for
7:    i ← iDIV2
8: end while
9: return x
```

```
alg4(integer n)
1: x ← 0
2: i ← n
3: while i ≥ 1 do
4:    for j ← 1, i do
5:       x ← x + 1
6:    end for
7:    i ← iDIV2
8: end while
9: return x
```

Indicaţie. `alg1`:

$$x = \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n}\sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{n(n+1)(n+2)}{6} \in \Theta(n^3)$$

`alg2`: x va conţine numărul de cifre ale reprezentării în baza 2 a lui n, adică $\lfloor \log_2(n) \rfloor + 1 \in \Theta(\lg n)$.

`alg3`: Spre deosebire de exemplul anterior, pentru fiecare valoare a lui i variabila x este mărită cu n, deci valoarea finală va fi $n(\lfloor \log_2(n) \rfloor + 1) \in \Theta(n \lg n)$.

`alg4`: Este similar algoritmului `alg3` însă pe linia 4 limita superioară a ciclului **for** este i în loc de n. Variabila x va conţine suma $n + \lfloor \frac{n}{2} \rfloor + \ldots + \lfloor \frac{n}{2^k} \rfloor$ cu k având proprietatea că $2^k \leq n < 2^{k+1}$. Ordinul de mărime al lui x se poate stabili pornind de la observaţia că $2^{k+1} - 1 \leq x < 2(2^{k+1} - 1)$. Deci $x \in \Theta(2^k) = \Theta(n)$.

```
alg5(integer n)
x ← 0
i ← 1
while i < n do
   x ← x + 1
   i ← 2 * i
end while
return x
```

```
alg6(integer n)
x ← 0
i ← 2
while i < n do
   x ← x + 1
   i ← i * i
end while
return x
```

`alg5`: Variabila x va conţine cel mai mare număr natural k cu proprietatea că $2^k \leq n$, deci $x = \lfloor \log_2 n \rfloor \in \Theta(\lg n)$.

`alg6`: Variabila x va conţine cel mai mare număr natural k cu proprietatea că $2^{2^k} \leq n$ deci $x = \lfloor \log_2 \log_2 n \rfloor \in \Theta(\lg \lg n)$.

Problema 6 (S) Să se determine termenul dominant şi ordinul de creştere pentru expresiile:

(a) $2\lg n + 4n + 3n\lg n$

(b) $2+4+6+\ldots 2n$

(c) $\frac{(n+1)(n+3)}{n+2}$

(d) $2+4+8+\ldots+2^n$

Indicaţie. (a) Termenul dominant în $2\lg n+4n+3n\lg n$ este evident $3n\lg n$ iar ordinul de creştere este $n\lg n$.
(b) Termenul dominant al sumei $2+4+6+\ldots 2n$ nu este $2n$ ci n^2 întrucât $2+4+6+\ldots 2n = n(n+1)$. Deci ordinul de creştere este chiar n^2.
(c) Termenul dominant este $n^2/(n+2)$ iar ordinul de creştere este n.
(d) Întrucât $2+4+8+\ldots+2^n = 2(1+2+4+\ldots+2^{n-1}) = 2(2^n-1)$, termenul dominant este $2\cdot 2^n$ iar ordinul de creştere este 2^n.

Problema 7 (S) Să se arate că:

(a) $n! \in \mathcal{O}(n^n)$, $n! \in \Omega(2^n)$

(b) $\lg n! \in \Theta(n\lg n)$

(c) $2^n \in \mathcal{O}(n!)$

(d) $\sum_{i=1}^{n} i\lg i \in \mathcal{O}(n^2\lg n)$

(e) $\lg(n^k+c) \in \Theta(\lg n)$, $k>0, c>0$.

Indicaţie. (a) Întrucât $n! \le n^n$ pentru orice n natural, rezultă imediat că $n! \in \mathcal{O}(n^n)$. Pe de altă parte, $n! \ge 2^{n-1}$ pentru orice n, deci $n! \in \Omega(2^n)$.
(b) Se porneşte de la aproximaţia Stirling $n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ care este adevărată pentru valori mari ale lui n. Mai exact, există $c_1, c_2 \in \mathbb{R}_+$ şi $n_0 \in \mathbb{N}$ astfel încât

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{c_1}{n}\right) \le n! \le \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{c_2}{n}\right)$$

pentru orice $n \ge n_0$.

Prin logaritmare se obţine:

$$\ln(2\pi n)/2 + \ln(1+c_1/n) + n\ln n - n \le \ln n \le \ln(2\pi n)/2 + \ln(1+c_1/n) + n\ln n - n$$

deci $\ln n! \in \Theta(n\ln n)$. Facând abstracţie de baza logaritmului se obţine $\lg n! \in \Theta(n\lg n)$
(c) Cum $2^n < n!$ pentru orice $n \ge 4$ rezultă că $2^n \in \mathcal{O}(n!)$.
(d) $\sum_{i=1}^{n} i\lg i \le \lg n \sum_{i=1}^{n} i \le n^2\lg n$, deci $\sum_{i=1}^{n} i\lg i \in \mathcal{O}(n^2\lg n)$.
(e) Pentru valori suficient de mari ale lui n are loc $\lg n^k \le \lg(n^k+c) \le \lg(2n^k)$, adică $k\lg n \le \lg(n^k+c) \le k\lg(n) + \lg 2$, deci $\lg(n^k+c) \in \Theta(\lg n)$.

Problema 8 (S) Care dintre următoarele afirmaţii sunt adevărate ? Demonstraţi.

(a) $\sum_{i=1}^{n} i^2 \in \Theta(n^2)$, $\sum_{i=1}^{n} i^2 \in \mathcal{O}(n^2)$, $\sum_{i=1}^{n} i^2 \in \Omega(n^2)$

(b) $cf(n) \in \Theta(f(n))$, $f(cn) \in \Theta(f(n))$

(c) $2^{n+1} \in \mathcal{O}(2^n)$, $2^{2n} \in \mathcal{O}(2^n)$?

Indicaţie. (a) Întrucât $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ rezultă că $\sum_{i=1}^{n} i^2 \in \Theta(n^3)$, deci $\sum_{i=1}^{n} i^2 \in \Omega(n^2)$ însa celelalte două afirmaţii sunt false.
(b)Întrucât $c_1 f(n) \le cf(n) \le c_2 f(n)$ pentru constante c_1 şi c_2 satisfăcând $0 < c_1 \le c \le c_2$ rezultă că $cf(n) \in \Theta(f(n))$. In schimb $f(cn) \in \Theta(f(n))$ nu este adevărată pentru orice funcţie f şi orice constantă c. De exemplu $f(n) = \exp(n)$ şi $c > 1$ nu satisfac această proprietate întrucât nu există c' astfel încât $\exp(cn) \le c'\exp(n)$ pentru valori mari ale lui n.
(c) Întrucât $2^{n+1} = 2\cdot 2^n$ rezultă că $2^{n+1} \in \Theta(2^n)$ deci implicit şi $2^{n+1} \in \mathcal{O}(2^n)$. Pe de altă parte, $\lim_{n\to\infty} 2^{2n}/2^n = \infty$ prin urmare $2^{2n} \notin \mathcal{O}(2^n)$ dar $2^{2n} \in \Omega(2^n)$.

Problema 9 (L) Propuneţi un algoritm pentru determinarea celor mai mici două valori dintr-un tablou. Determinaţi numărul de comparaţii efectuate asupra elementelor tabloului şi stabiliţi ordinul de complexitate al algoritmului.

Indicaţie. O variantă de algoritm este descrisă în `valori_minime`.

```
valori_minime(integer a[1..n])
if a[1] < a[2] then
   min1 ← a[1]; min2 ← a[2];
else
   min1 ← a[2]; min2 ← a[1];
end if
for i ← 3, n do
   if a[i] < min1 then
      min2 ← min1; min2 ← a[i]
   else if a[i] < min2 then
      min2 ← a[i]
   end if
end for
return min1, min2
```

În cazul cel mai nefavorabil numărul de comparaţii efectuate asupra elementelor tabloului este $T(n) = 1 + 2(n-2) = 2n-3$ deci algoritmul aparţine lui $\mathcal{O}(n)$.

Problema 10 (L) Propuneţi un algoritm de complexitate liniară pentru a determina tabloul frecvenţelor cumulate pornind de la tabloul frecvenţelor simple. Pentru un tablou $(f_1, \ldots, f_n)$ de frecvenţe, tabloul frecvenţelor cumulate $(fc_1, \ldots, fc_n)$ se caracterizează prin $fc_i = \sum_{j=1}^{i} f_j$.

Rezolvare. Este uşor de văzut că un algoritm care calculează pentru fiecare i valoarea sumei $fc_i = \sum_{j=1}^{i} f_j$ necesită efectuarea a $\sum_{i=2}^{n} \sum_{j=1}^{i} 1 = \sum_{i=2}^{n} i = n(n+1)/2-1 \in \Theta(n^2)$. Observând că $fc_i = fc_{i-1}+f_i$ pentru $i = \overline{2,n}$ şi $fc_1 = f_1$ rezultă că frecvenţele cumulate pot fi calculate prin algoritmul `frecvente_cumulate` descris în continuare.

```
frecvente_cumulate(integer f[1..n])
integer i,fc[1..n]
fc[1] ← f[1]
for i ← 2, n do
   fc[i] = fc[i − 1] + f[i]
end for
return fc[1..n]
```

Probleme suplimentare seminar/laborator

1. Scrieţi un algoritm pentru a verifica dacă elementele unui tablou (cu valori întregi) sunt distincte sau nu. Demonstraţi corectitudinea algoritmului şi estimati timpul de execuţie considerând comparaţia între elementele tabloului drept operaţie dominantă.

2. Scrieţi un algoritm pentru a verifica dacă elementele unui tablou (cu valori întregi) sunt toate identice sau nu. Demonstraţi corectitudinea algoritmului şi timpul de execuţie considerând comparaţia între elementele tabloului drept operaţie dominantă.

3. Se consideră o matrice A cu m linii şi n coloane şi o valoare x. Scrieţi un algoritm care verifică dacă valoarea x este prezentă sau nu în matrice. Stabiliţi ordinul de complexitate al algoritmului considerând comparaţia cu elementele matricii drept operaţie dominantă.

4. Se consideră un tablou $x[1..n-1]$ care conţine valori distincte din mulţimea $\{1, 2, \ldots, n\}$. Propuneţi un algoritm de complexitate liniară care să identifice valoarea din $\{1, 2, \ldots, n\}$ care nu este prezentă în tabloul x.

   Observaţie. Incercaţi să rezolvaţi problema folosind spaţiu de memorie auxiliar de dimensiune $\mathcal{O}(1)$ (aceasta înseamnă să nu folosiţi vectori auxiliari).

5. Se consideră un tablou $x[1..n]$ ordonat crescător cu elemente care nu sunt neapărat distincte. Să se transforme tabloul x astfel încât pe primele k poziţii să se afle elementele distincte în ordine crescătoare. De exemplu pornind de la $[1, 2, 2, 4, 4, 4, 7, 8, 9, 9, 9, 9]$ se obţine tabloul ce conţine pe primele $k = 6$ poziţii valorile: $[1, 2, 4, 7, 8, 9]$ (restul elementelor nu interesează ce valori au).

6. Se consideră un tablou $x[1..n]$ şi o valoare $x0$ care este prezentă (pe o singură poziţie) în tablou. Presupunem că avem un algoritm de căutare care verifică elementele tabloului într-o ordine aleatoare (dar verifică fiecare element o singură dată). Procedura este similară celei în care avem într-o cutie $n-1$ bile negre şi o singură bilă albă şi efectuăm extrageri (fără a pune bila extrasă înapoi) până la extragerea bilei albe. Estimaţi numărul mediu de comparaţii (sau extrageri de bile) până la identificarea valorii căutate (extragerea bilei albe).

   Indicaţie. Se presupune că elementele (bilele) disponibile pot fi selectate cu aceeaşi probabilitate.

7. Scrieţi un algoritm care calculează produsul dintre o matrice $A[1..m, 1..n]$ cu m linii şi n coloane şi un vector $x[1..n]$ cu n elemente şi estimaţi numul de operaţii de înmulţire efectuate.

   Indicaţie. Produsul va fi un vector $y[1..m]$ în care $y[i] = \sum_{j=1}^{n} A[i, j] * x[j]$, $i = \overline{1, m}$.

8. Se consideră un şir de caractere $s[1..n]$ şi un şablon (subşir de caractere) $t[1..m]$ (cu $m < n$). Scrieţi un algoritm care verifică dacă şablonul t este sau nu prezent în s şi stabiliţi ordinul de complexitate al algoritmului (considerând comparaţia dintre elementele şirului şi cele ale şablonului ca operaţie dominantă).

   Indicaţie. Cel mai simplu algoritm (nu şi cel mai eficient) se bazează pe parcurgerea şirului s de la poziţia $i = 1$ până la poziţia $i = n - m$ şi compararea element cu element al lui $s[i..i+m-1]$ cu $t[1..m]$.

9. Se consideră un tablou de valori întregi $x[1..n]$ şi o valoare dată, s. Să se verifice dacă există cel puţin doi indici i şi j (nu neapărat distincţi) care au proprietatea că $x[i] + x[j] = s$. Analizaţi complexitatea algoritmului propus.

10. Care este valoarea variabilei x după execuţia prelucrărilor:

```
x ← 0
j ← n
while j ≥ 1 do
    for i ← 1, j do
        x ← x + 1
    endfor
    j ← j DIV 3
endwhile
```

11. Să se determine ordinul de mărime al variabilei x după execuţia următoarelor prelucrări:

```
x ← 0
j ← n
while j ≥ 1 do
    for i ← 1, j do
        x ← x + 1
    end for
    j ← j DIV 3
end while
```

```
return x
```

12. Se consideră următorii doi algoritmi pentru calculul puterii unui număr natural, x ($p = x^n$):

`putere1`(**real** x, **integer** n)	`putere1`(**real** x, **integer** n)
$p \leftarrow 1$	$p \leftarrow 1$
for $i \leftarrow 1, n$ **do**	**for** $i \leftarrow 1, n$ **do**
$\quad np \leftarrow 0$	$\quad p \leftarrow p * x$
$\quad$ **for** $j \leftarrow 1, x$ **do**	**end for**
$\quad\quad np \leftarrow np + p$	**return** p
$\quad$ **end for**	
$\quad p \leftarrow np$	
end for	
return p	

Să se stabilească ordinele de complexitate considerând că toate operaţiile aritmetice au acelaşi cost.