Algoritmi şi structuri de date (I). Seminar 5: Analiza eficienţei algoritmilor. Identificarea celui mai favorabil şi a celui mai defavorabil caz. Estimarea timpului de execuţie prin contorizarea operaţiei dominante.

Problema 1 Scrieţi un algoritm care determină numărul de inversiuni ale unei permutări (pentru o permutare reprezentată printr-un tablou $a[1..n]$ o inversiune este o pereche de indici (i, j) cu proprietăţile $i, j \in \{1, \dots, n\}$, $i < j$ şi $a[i] > a[j]$. Determinaţi numărul de operaţii de comparare (asupra elementelor tabloului a) efectuate de către algoritm şi stabiliţi ordinul de complexitate al algoritmului.

Indicaţie. Numărul de inversiuni poate fi calculat prin algoritmul de mai jos:

```
inversiuni(a[1..n])
k ← 0
for i ← 1, n − 1 do
   for j ← i + 1, n do
      if a[i] > a[j] then
         k ← k + 1
      end if
   end for
end for
return k
```

Pentru stabilirea ordinului de complexitate se consideră n ca fiind dimensiunea problemei şi $a[i] > a[j]$ ca fiind operaţia dominantă. Numărul de execuţii ale operaţiei dominante este $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = n(n-1)/2$. Deci $T(n) = n(n-1)/2$, adică $T(n) \in \Theta(n^2)$.

Problema 2 Se consideră algoritmul de mai jos care verifică dacă un tablou este ordonat crescător. Demonstraţi (identificând un invariant şi o funcţie de terminare) că algoritmul este corect şi analizaţi eficienţa atât în cazurile extreme (cel mai favorabil, cel mai defavorabil) cât şi în cel mediu.

```
ordonat(a[1..n])
r ← true
i ← 0
while (i < n − 1) and (r = true) do
   i ← i + 1
   if a[i] > a[i + 1] then
      r ← false
   end if
end while
return r
```

Indicaţie. (a) *Analiza corectitudine.* E uşor de remarcat că proprietatea "r conţine valoarea de adevăr a afirmaţiei $a[1..i]$ este ordonat crescător" este invariantă în raport cu ciclul **while** şi permite demonstrarea corectitudinii algoritmului de mai sus.

(b) *Analiza eficienţei în cazul mediu.* Să analizăm cazul mediu în ipoteza că există n clase distincte de date de intrare. Clasa C_i ($i \in \{1, \dots, n-1\}$) 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:

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

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

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 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 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.

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]
```

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 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 $n-1$.

Probleme suplimentare/teme

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. 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.

3. 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).

4. 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).