
Algoritmi și structuri de date (I). Seminar 5: Analiza eficienței algoritmilor. Identificarea celui mai favorabil și a celui mai defavorabil caz. Estimarea timpului de execuție prin contorizarea operației dominante.

Problema 1 Scrieți 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 proprietățile $i, j \in \{1, \dots, n\}$, $i < j$ și $a[i] > a[j]$). Determinați numărul de operații de comparare (asupra elementelor tabloului a) efectuate de către algoritm și stabiliți ordinul de complexitate al algoritmului.

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

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

Pentru stabilirea ordinului de complexitate se consideră n ca fiind dimensiunea problemei și $a[i] > a[j]$ ca fiind operația dominantă. Numărul de execuții ale operației 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. Demonstrați (identificând un invariant și o funcție de terminare) că algoritmul este corect și analizați eficiența atât în cazurile extreme (cel mai favorabil, cel mai defavorabil) câtși în cazul mediu.

```
ordonat(a[1..n])
1:  $r \leftarrow \text{true}$ 
2:  $i \leftarrow 0$ 
3: while  $(i < n - 1)$  and  $(r = \text{true})$  do
4:    $i \leftarrow i + 1$ 
5:   if  $a[i] > a[i + 1]$  then
6:      $r \leftarrow \text{false}$ 
7:   end if
8: end while
9: return  $r$ 
```

Indicație. (a) *Analiza corectitudinii.* E ușor de remarcat că proprietatea ” r conține valoarea de adevăr a afirmației $a[1..i]$ este ordonat crescător” este invariantă în raport cu ciclul **while** și permite demonstrarea corectitudinii algoritmului de mai sus.

(b) Dimensiunea problemei este n , iar operația dominantă este $a[i] > a[i + 1]$.

2 *Analiza în cazul cel mai favorabil.* Cazul cel mai favorabil corespunde situației în care $a[1] > a[2]$, caz în care se efectuează o singură comparație. Prin urmare algoritmul este din clasa $\Omega(1)$.

Analiza în cazul cel mai defavorabil. Cazul cel mai defavorabil corespunde situației în care tabloul $a[1..n]$ este ordonat crescător sau $a[i] \leq a[i + 1]$ pentru $i = \overline{1, n - 2}$ iar $a[n - 1] > a[n]$, caz în care se $n - 1$ comparații. Algoritmul este din clasa $O(n)$.

Analiza eficienței 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 condiția de ordonare crescătoare este $(a[i], a[i + 1])$. Când se ajunge la $i = n$ considerăm că tabloul

este ordonat crescător. Numărul de comparații corespunzător clasei i este i . Notând cu p_i probabilitatea de apariție a unei instanțe din clasa C_i rezultă că timpul mediu de execuție este:

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

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

În ipoteza că toate cele n clase au aceeași probabilitate de apariție se obține 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 probabilitățile corespunzătoare nu sunt ușor de determinat. Ceea ce se poate observa este faptul că $p_1 \geq p_2 \geq \dots \geq p_n$ deci pe măsură ce probabilitatea corespunzătoare unei instanțe crește numărul de operații scade. Analiza cazului mediu este în general dificilă, fiind fezabilă doar în cazuri particulare, când se impun ipoteze simplificatoare asupra distribuției de probabilitate.

Problema 3 Se consideră un șir de valori întregi (pozitive și negative - cel puțin un element este pozitiv). Să se determine suma maximă a elementelor unei secvențe a șirului (succesiune de elemente consecutive).

Rezolvare.

Varianta 1. Pentru fiecare $i \in \{1, \dots, n\}$ se calculează succesiv sumele elementelor subtablourilor $a[i..j]$ cu $j \in \{1, \dots, n\}$ și se reține 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
```

Operațiile dominante sunt cele specificate pe liniile 5 și 6, iar numărul de repetări ale acestora este $T(n) = \sum_{i=1}^n \sum_{j=i}^n 1 = n(n+1)/2$, prin urmare acest algoritm aparține clasei $\Theta(n^2)$.

Varianta 2. Aceeași problemă poate fi rezolvată și printr-un algoritm de complexitate liniară calculând succesiv suma elementelor din tablou și reinițiind calculul în momentul în care suma devine negativă (în acest caz reinițializând cu 0 variabila s valoarea obținută este mai mare decât cea curentă, care este negativă). Algoritmul **secventa2** este descris în continuare.

secventa2(integer $a[1..n]$)

```
1:  $max \leftarrow 0$ ;  $icrt \leftarrow 1$ ;  $imax \leftarrow 1$ 
2:  $jmax \leftarrow 0$ 
3:  $s \leftarrow 0$ 
4: for  $i \leftarrow 1, n$  do
5:    $s \leftarrow s + a[i]$ 
6:   if  $s < 0$  then
7:      $s \leftarrow 0$ ;  $icrt \leftarrow i + 1$ 
8:   else if  $s > max$  then
9:      $max \leftarrow s$ ;  $imax \leftarrow icrt$ ;  $jmax \leftarrow i$ 
10:  end if
11: end for
12: return  $max, imax, jmax$ 
```

Numărul de execuții ale operației dominante ($s \leftarrow s + a[i]$) este $T(n) = n$, prin urmare algoritmul aparține clasei $O(n)$.

Problema 4 Să se estimeze timpul de execuție 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 și n și reținerea celor prime; (b) algoritmul lui Eratostene.

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

(b) *Algoritmul lui Eratostene.* Se pornește de la tabloul $a[2..n]$ inițializat cu valorile naturale cuprinse între 2 și n și 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)

```
1:  $k \leftarrow 0$ 
2: for  $i \leftarrow 2, n$  do
3:   if prim( $i$ ) = true then
4:      $k \leftarrow k + 1$ 
5:      $p[k] \leftarrow i$ 
6:   end if
7: end for
8: return  $p[1..k]$ 
```

eratostene(integer
 n)

```
1: for  $i \leftarrow 2, n$  do
2:    $a[i] \leftarrow i$ 
3: end for
4: for  $i \leftarrow 2, \lfloor \sqrt{n} \rfloor$  do
5:   if  $a[i] \neq 0$  then
6:      $j \leftarrow i * i$ 
7:     while  $j \leq n$  do
8:        $a[j] \leftarrow 0$ ;  $j \leftarrow j + i$ 
9:     end while
10:  end if
11: end for
12:  $k \leftarrow 0$ 
13: for  $i \leftarrow 2, n$  do
14:   if  $a[i] \neq 0$  then
15:      $k \leftarrow k + 1$ ;  $p[k] \leftarrow a[i]$ 
16:   end if
17: end for
18: return  $p[1..k]$ 
```

În ambele cazuri considerăm dimensiunea problemei ca fiind n .

În prima variantă considerăm că operațiile dominante sunt cele efectuate în cadrul subalgoritmului **prim**. Luăm în considerare doar operația de analiză a divizorilor potențiali. Pentru fiecare i numărul de comparații 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 $\int_2^n \sqrt{x} dx = 2/3 n\sqrt{n} - 4\sqrt{2}/3$ obținându-se că:

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

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

Se obține astfel că $T_1(n) \in \Theta(n\sqrt{n})$.

În cazul algoritmului lui Eratostene operația dominantă poate fi considerată cea de marcarea 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 operații 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 și se obține:

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

deci $T_2(n) \in \mathcal{O}(n \lg n)$.

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. Scrieți un algoritm pentru a verifica dacă elementele unui tablou (cu valori întregi) sunt distincte sau nu. Demonstrați corectitudinea algoritmului și estimați timpul de execuție considerând comparația între elementele tabloului drept operație dominantă.
2. Scrieți un algoritm pentru a verifica dacă elementele unui tablou (cu valori întregi) sunt toate identice sau nu. Demonstrați corectitudinea algoritmului și timpul de execuție considerând comparația între elementele tabloului drept operație dominantă.
3. Se consideră o matrice A cu m linii și n coloane și o valoare x . Scrieți un algoritm care verifică dacă valoarea x este prezentă sau nu în matrice. Stabiliți ordinul de complexitate al algoritmului considerând comparația cu elementele matricii drept operație dominantă.
4. Se consideră un tablou $x[1..n-1]$ care conține valori distincte din mulțimea $\{1, 2, \dots, n\}$. Propuneți un algoritm de complexitate liniară care să identifice valoarea din $\{1, 2, \dots, n\}$ care nu este prezentă în tabloul x .

Observație. Încercați să rezolvați problema folosind spațiu de memorie auxiliar de dimensiune $\mathcal{O}(1)$ (aceasta înseamnă să nu folosiți 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 poziții să se afle elementele distincte în ordine crescătoare. De exemplu pornind de la $[1, 2, 2, 4, 4, 4, 7, 8, 9, 9, 9]$ se obține tabloul ce conține pe primele $k = 6$ poziții valorile: $[1, 2, 4, 7, 8, 9]$ (restul elementelor nu interesează ce valori au).