Proiectarea algoritmilor: complexitatea medie

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2015/2016

1. Timpul mediu: algoritmi deterministi
   - Quicksort determinist
2. Timpul mediu: algoritmi probabiliști
   - Agoritmi nedeterminiști în general
   - Agoritmi nedeterminiști pentru probleme de decizie
   - Algoritmi probabiliști
   - Quicksort probabilist
3. *k*-mediana

Plan

Motivație

- A rezolvă P, $p \in P$
- timpul în cazul cel mai nefavorabil:
  $T_A(n) = \sup\{time(A, p) \mid p \in P \wedge g(p) = n\}$
- $T_A(n)$ este irelevant dacă numărul instanțelor p cu $g(p) = n \wedge time(A, p) = T_A(n)$ (sau $time(A, p) = T_A(n) - \varepsilon$) este foarte mic

Definiție

- $time(A, p)$ ca variabilă aleatorie:
  - o experiență = execuția algoritmului pentru o instanță p,
  - valoarea experienței = durata execuției algoritmului
- legea de repartiție a acestei variabile aleatorie
- timpul de execuție mediu = media acestei variabile aleatoare

$$T_A^{med}(n) = M(\{time(A, p) \mid p \in P \wedge g(p) = n\})$$

- Caz particular: $time(A, p) = \{x_0, x_1, \ldots\}$, $P(p \mid time(A, p) = x_i) = p_i$

$$T_A^{med}(n) = \sum_i x_i \cdot p_i$$

Exemplu

Problema FIRST OCCURRENCE

Input: $n, a = (a_0, \ldots, a_{n-1}), z$, toate numere întregi.

Output: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Algoritm pentru FIRST OCCURRENCE

Algoritmul FOAlg descris de următorul program rezolvă FIRST OCCURRENCE:

```
//@input: un tablou a cu n elemente, z
//@output:  pozitia primului element din a egal cu z,
//          -1 daca nu exista un astfel de element
i =  0;
while (a[i] != z) && (i < n-1) {
  i =  i+1
if (a[i] = z)  poz =  i;
else poz =  -1;
```

Timpul mediu pentru FOAlg 1/2

dimensiune instanță: $n = \texttt{a.size}()$

operații măsurate: atribuiri și comparații

$time(\mathrm{FOAlg}, \{p \mid size(p) = n\}) = \{3i + 2 \mid 1 \leq i \leq n\}$

Presupuneri:

- probabibilitatea ca $z \in \{a_0, \ldots, a_{n-1}\}$ este q și
- probabilitatea ca z să apară prima dată pe poziția $i - 1$ este $\frac{q}{n}$ (indicii i candidează cu aceeași probabilitate pentru prima apariție a lui z).

Timpul mediu pentru FOAlg 2/2

$P(z \notin \{a_0, \ldots, a_{n-1}\}) = 1 - q$

$P(time(\mathrm{FOAlg}, p) = 3i + 2) = \dfrac{q}{n} = p_i,\ 1 \leq i < n$

$P(time(\mathrm{FOAlg}, p) = 3n + 2) = \dfrac{q}{n} + (1 - q) = p_n$

Timpul mediu de execuție este:

$$
\begin{aligned}
T^{med}_{\mathrm{FOAlg}}(n) &= \sum_{i=1}^{n} p_i x_i \\
&= \sum_{i=1}^{n-1} \frac{q}{n} \cdot (3i + 2) + (\frac{q}{n} + (1 - q)) \cdot (3n + 2) \\
&= 3n - \frac{3nq}{2} + \frac{3q}{2} + 2
\end{aligned}
$$

Plan

Quicksort: descriere

Este proiectat pe paradigma divide-et-impera.

Algoritmul Quicksort

Input: $S = \{a_0, \ldots, a_{n-1}\}$

Output: o secvență cu elementele a_i în ordine crescătoare

1. se alege $x \in S$
2. calculează
   $S_< = \{a_i \mid a_i < x\}$ $S_= = \{a_i \mid a_i = x\}$ $S_> = \{a_i \mid a_i > x\}$
3. sortează recursiv $S_<$ și $S_>$ producând $Seq_<$ și $Seq_>$, respectiv
4. întoarce secvența $Seq_<$, $S_=$, $Seq_>$

Quicksort: partiționarea

Presupunem că S este memorată într-un tablou a.

Se determină prin interschimbări a unui indice k cu proprietățile:

- $p \leq k \leq q$ și $a[k] = x$;
- $\forall i : p \leq i \leq k \implies a[i] \leq a[k]$;
- $\forall j : k < j \leq q \implies a[k] \leq a[j]$;

- inițial: $x = a[p]$, $i = p + 1$ și $j = q$
- proprietățile menținute invariante:

$$\forall i' : p \leq i' < i \implies a[i'] \leq x \tag{1}$$

și

$$\forall j' : j < j' \leq q \implies a[j'] \geq x \tag{2}$$

Quicksort: partiționarea 1/4

Presupunem că la momentul curent sunt interogate elementele a[i] și a[j] cu $i < j$. Distingem următoarele cazuri:

1. $a[i] \leq x$. Transformarea `i = i+1` păstrează 1.
2. $a[j] \geq x$. Transformarea `j = j-1` păstrează 2.
3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea a[i] $\leftrightarrow$ a[j] și se face `i = i+1` și `j = j-1`, atunci ambele predicate 1 și 2 sunt păstrate.

Quicksort: partiționarea 2/4

Operațiile de mai sus sunt repetate până când *i* devine mai mare decât *j*:

```
while (i ≤ j) {
  if (a[i] ≤ x) i = i+1;
  else if (a[j] ≥ x) j = j-1;
  else if ((i < j) && (a[i] > x) && (x > a[j])) {
    swap(a, i, j);
    i = i+1;
    j = j-1;
  }
}
```

Quicksort: partiționarea 3/4

Analiza terminării lui `while`:

- $i = j + 1$:
- din 1, 2 avem $a[i - 1] \leq x$ și $a[i] = a[j + 1] \geq x$
- deci interschimbând `a[p]` cu `a[i` $-$ `1]` obținem partiționarea dorită a tabloului $\implies k = i - 1$

```
k = i-1;  a[p] = a[k];  a[k] = x;
```

- analiza cazurilor limită:
  $i = p + 1$ – relațiile de mai sus au sens
  $j = q \implies k = q$, i.e. $a[p..k] = a[p..q]$ ce conduce la recursie infinită; în acest caz k trebuie decrementat

Quicksort: algoritmul de partiționare 4/4

@input: $a = (a[p], \ldots, a[q])$
@output: k, a cu proprietatea
$\forall\ i : p \leq\ i \leq\ k \implies\ a[i] \leq\ a[k]$ şi $\forall\ j : k <\ j \leq\ q \implies\ a[k] \leq\ a[j]$

```
partitioneaza(a, p, q, k) {
  x = a[p] ;
  i = p + 1;   j = q;
  while (i ≤ j) {
    if (a[i] ≤ x) i = i+1;
    else if (a[j] ≥ x) j = j-1;
    else if ((i < j) && (a[i] > x) && (x > a[j])) {
      swap(a, i, j);
      i = i+1;
      j = j-1;
    }
  }
  k = i-1;  a[p] = a[k];  a[k] = x;
  if (j == q) --k;
}
```

Quicksort: algoritm

```
@input:   a = (a[p], ..., a[q])
@output:  elementele secvenţei  a în ordine crescătoare
qsort(a, p, q) {
  if (p < q) {
    partitioneaza(a, p, q, k)
    qSort(a, p, k)
    qSort(a, k+1, q)
  }
}
```

Quicksort: timpul în cazul cel mai nefavorabil

- dimensiune instanță: $\texttt{n} = \texttt{a.size}()$
- operații măsurate: comparații care implică elementele tabloului
- cazul cel mai nefavorabil: tabloul deja ordonat
- numărul de comparații pentru acest caz:
  $(n-1)+(n-2)+\cdots+1 = O(n^2)$

Quicksort: timpul mediu

- lungimea secvenței: $q + 1 - p = n$
- probabilitatea ca x sa fie al k-lea element: $\dfrac{1}{n}$
- dimensiuni subprobleme: $k - p = i - 1$ și $q - k = n - i$
- numărul mediu de comparații:

$$T^{med}(n) = \begin{cases} (n-1) + \dfrac{1}{n}\sum\limits_{i=1}^{n}(T^{med}(i-1) + T^{med}(n-i)) & \text{, dacă } n \geq 1 \\ 1 & \text{, dacă } n = 0 \end{cases}$$

Teoremă

Complexitatea medie a algoritmului QuickSort este $O(n \log_2 n)$.

Plan

Plan

Extensia limbajului

`choose` x `in` S`;` – întoarce un element din S ales arbitrar

`choose` x `in` S `s.t.` B`;` – întoarce un element din S care satisface condiția B ales arbitrar

`failure;` – semnalează terminarea fără succes (e.g., o instrucțiune `choose` nu s-a putut executa)

Demo cu noile instructiuni

```
choose x1 in { 1 ..  5 };

$ ~/k-3.6/bin/krun tests/choose.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    x1 |-> 3
</state>
$ ~/k-3.6/bin/krun tests/choose.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    x1 |-> 1
</state>
```

Demo cu noile instructiuni

```
choose x1 in { 1 ..  5 };
```

```
$ ~/k-3.6/bin/krun tests/choose.alk -cINIT=".Map" --search
Search results:
Solution 1:
<k>
    .K
</k>
<state>
    x1 |-> 1
</state>
...
Solution 5:
<k>
    .K
</k>
<state>
    x1 |-> 5
</state>
```

Demo cu noile instructiuni

```
odd(x) {
  return x % 2 == 1;
}
choose x1 in  1 .. 5  s.t. odd(x1);
```

```
$ ~/k-3.6/bin/krun tests/choosest.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    x1 |-> 5
</state>
$ ~/k-3.6/bin/krun tests/choosest.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    x1 |-> 1
</state>
```

Demo cu noile instructiuni

```
$ ~/k-3.6/bin/krun tests/choosest.alk -cINIT=".Map" --search
Search results:
Solution 1:
<k>
    .K
</k>
<state>
    x1 |-> 1
</state>
Solution 2:
<k>
    .K
</k>
<state>
    x1 |-> 3
</state>
Solution 3:
<k>
    .K
</k>
<state>
    x1 |-> 5
</state>
```

Demo cu noile instructiuni

```
odd(x) {
  return x % 2 == 1;
}

s = emptySet;
for (i = 0; i < 8; i = i+2)
  s.pushBack(i);
choose x in s s.t. odd(x);

$ ~/k-3.6/bin/krun tests/failure.alk -cINIT=".Map"
<k>
    failure ;
</k>
<state>
    i |-> 8
    s |->  0, 2, 4, 6
    x |-> 6
</state>
```

Execuția unui program nedeterminist

- un program nedeterminist are mai multe fire de execuție
- un program nedeterminist rezolvă P dacă $\forall\, p \in P\ \exists$ un fir de execuție care se termină și a cărui configurție finală include $P(p)$

Exemplu: problema celor N regine

Input: o tablă de șah $n \times n$.
Output: o așezare a n piese de tip regină pe tablă a.î. nicio regină nu atacă o altă regină.

```
attacked(i, j, b) {
  attack = false;
  for (k = 0;  k < i; ++k)
     if ((b[k] == j) || ((b[k]-j) == (k-i)) || ((b[k]-j) == (i-k)))
       attack = true;
  return(attack);
}

nqueens (n) {
   for (i = 0; i < n; ++i) {
     choose j in { 0 .. n-1 } s.t. ! (attacked(i, j, b));
     b[i] = j;
    }
}
```

Exemplu: problema celor N regine

```
$ ~/k-3.6/bin/krun tests/nqueens.alk -cINIT="n |-> 4"
<k>
    failure ;
</k>
<state>
    b |-> [ 0, 3, 1, -1 ]
    i |-> 3
    j |-> 3
    n |-> 4
</state>
<stack>
    .List
</stack>
```

Exemplu: problema celor N regine

```
$ ~/k-3.6/bin/krun tests/nqueens.alk -cINIT="n |-> 4" --search
Search results:
Solution 1:
<k>
    .K              i.e. success
</k>
<state>
    b |-> [ 1, 3, 0, 2 ]
    n |-> 4
</state>
Solution 2:
<k>
    .K              i.e. success
</k>
<state>
    b |-> [ 2, 0, 3, 1 ]
    n |-> 4
</state>
```

Exemplu: problema celor N regine

```
Solution 3:
<k>
    failure ;
</k>
<state>
    b |-> [ 0, 2, -1, -1 ]
    i |-> 2
    j |-> 3
    n |-> 4
</state>

Solution 4:
<k>
    failure ;
</k>
<state>
    b |-> [ 0, 3, 1, -1 ]
    i |-> 3
    j |-> 3
    n |-> 4
</state>
```

Exemplu: problema celor N regine

```
Solution 5:
<k>
    failure ;
</k>
<state>
    b |-> [ 3, 0, 2, -1 ]
    i |-> 3
    j |-> 3
    n |-> 4
</state>

Solution 6:
<k>
    failure ;
</k>
<state>
    b |-> [ 3, 1, -1, -1 ]
    i |-> 2
    j |-> 3
    n |-> 4
</state>
```

Plan

1. Timpul mediu: algoritmi deterministi
   - Quicksort determinist
2. Timpul mediu: algoritmi probabiliști
   - Agoritmi nedeterminiști în general
   - Agoritmi nedeterminiști pentru probleme de decizie
   - Algoritmi probabiliști
   - Quicksort probabilist
3. *k*-mediana

Definiție

- Activitatea unui algoritm nedeterminist pentru o problemă de decizie:
  - "se ghicește" o anumită structură S
  - verifică dacă S satisface o proprietățile din *output*
- Extensia limbajului:

  `success` – semnalează terminarea verificării (și a a algoritmului) cu succes

Agoritmi nedeterminiști pentru probleme de decizie: exemplu

SAT

Instance: O mulțime finită de variabile și o formulă propozițională F în formă normală conjunctivă. *Question*: Este F adevărată pentru o anume atribuire de variabile? (i.e., este F satisfiabilă?)

```
// guess
for (i = 0; i < n; ++i) {
  choose z in {false, true};
  x[i] = z;
}
// check
if (f(x)) success;
else failure;
```

Exemplu de instanță SAT

```
f(x) {
  return (x[0] || x[1]) &&
         (!x[0] || x[3] || x[2]) &&
         (x[2] || !x[3]) &&
         (!x[1] || !x[2] || x[3]);
}
```

Execuție nedeterministă

```
$ ~/k-3.6/bin/krun tests/sat.alk -cINIT="n |-> 4"
<k>
    failure ;
</k>
<state>
    i |-> 4
    n |-> 4
    x |-> [ false, true, false, true ]
    z |-> true
</state>
<stack>
    .List
</stack>
```

Execuție nedeterministă

```
$ ~/k-3.6/bin/krun tests/sat.alk -cINIT="n |-> 4"
<k>
    failure ;
</k>
<state>
    i |-> 4
    n |-> 4
    x |-> [ false, false, true, true ]
    z |-> true
</state>
<stack>
    .List
</stack>
```

Execuție exhaustivă

```
$ ~/k-3.6/bin/krun tests/sat.alk -cINIT="n |-> 4" --search
Search results:

Solution 1:
<k>
    failure ;
</k>
...
Solution 12:
<k>
    success ;
</k>
<state>
    i |-> 4
    n |-> 4
    x |-> [ false, true, false, false ]
    z |-> false
</state>
<stack>
    .List
</stack>
```

Plan

Definiții

Exista două puncte de vedere:

- 1. algoritmul probabilist este văzut ca un algoritm nedeterminist pentru care există o distribuție de probabilitate peste alegerile nedetermiste
- 2. algoritmul probabilist este un algoritm care are o intrare suplimentară ce constă într-o secvență de biți aleatorii;

Această diferență poate fi proiectată în complexitate sau ieșire:

- timpul de execuție văzut ca o variabilă aleatorie
- ieșirea văzută ca o variabilă aleatorie

La acest curs consideră doar prima variantă (numiți și algoritmi Las Vegas).

Timpul mediu de execuție al algoritmilor probabiliști

$Prob_{A,x}(C)$ = probabilitatea cu care algoritmul A execută calculul C pentru intrarea x

$Prob(A(x) = y)$ = probabilitatea cu care algoritmul A produce ieșirea y pentru intrarea x
$Prob(A(x) = y) = \sum_{C, A_C(x)=y} Prob_{A,x}(C)$

$Time(C)$ = timpul necesar lui A ca să execute C

timpul mediu de execuție a lui A pentru intrarea x este
$Exp\text{-}Time_A(x) = M[Time] = \sum_{C, A_C(x)=y} Prob_{A,x}(C) \cdot Time(C)$.

$Time$ este variabilă aleatorie.

timpul mediu de execuție a lui A în cazul cel mai nefavorabil este
$Exp\text{-}Time_A(n) = max\{Exp\text{-}Time_A(x) \mid g(x) = n\}$

Plan

"Randomized Quicksort"

- exemplul canonic pentru algoritmii Las Vegas

Algoritmul RQS

Intrare: $S = \{a_0, \ldots, a_{n-1}\}$

Ieșire: elementele a_i în ordine crescătoare

1. dacă $n = 1$ întoarce a_0, altfel alege aleatoriu $k \in \{0, \ldots, n-1\}$
2. calculează
   $S_< = \{a_i \mid a_i < a_k\}$
   $S_= = \{a_i \mid a_i = a_k\}$
   $S_> = \{a_i \mid a_i > a_k\}$
3. sortează recursiv $S_<$ și $S_>$
4. întoarce secvența $S_<$, $S_=$, $S_>$

"Randomized Quicksort"

Se modifică doar algoritmul de partiționare:

```
partitioneaza(out a, p, q) {
  m = p + random(q-p);
  swap(a, p, m);
  in rest la fel ca în cazul determinist
}
```

Evident că se poate optimiza pentru a face mai puține interschimbări ... (exercițiu)

Analiza algoritmului RQS

Fie funcția *rank* astfel încât $a_{rank(0)} \leq \ldots \leq a_{rank(n-1)}$.

Definim $X_{ij} = \begin{cases} 1 & a_{rank(i)} \text{ şi } a_{rank(j)} \text{ sunt comparate} \\ 0 & altfel \end{cases}$

X_{ij} numără comparațiile dintre $a_{rank(i)}$ şi $a_{rank(j)}$

X_{ij} este variabilă aleatorie

numărul mediu de comparații este

$M[\sum_{i=0}^{n-1} \sum_{j>i} X_{ij}] = \sum_{i=0}^{n-1} \sum_{j>i} M[X_{ij}]$

Analiza algoritmului RQS

p_{ij} probabiltatea ca $a_{rank(i)}$ și $a_{rank(j)}$ să fie comparate într-o execuție

$M[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij}$

$p_{ij} = \frac{2}{j - i + 1}$

$$\begin{aligned}
\sum_{i=0}^{n-2} \sum_{j>i} p_{ij} &= \sum_{i=0}^{n-2} \sum_{j>i} \frac{2}{j - i + 1} \\
&\le \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{2}{k} \\
&\le 2 \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{1}{k}
\end{aligned}$$

Analiza algoritmului RQS

Teoremă

Numărul mediu de comparații într-o execuție al algoritmului RQS este cel mult $2nH_n = O(n \log n)$.

Plan

k-mediana: problema

Definiție

Fie S o listă cu n elemente dintr-o mulțime univers total ordonată. k-mediana este cel de-al k-lea element din lista sortată a elementelor din S.

Presupunem S memorată într-un tablou.
Considerăm următoarea problemă:

Input	un tablou $(a[i] \mid 0 \leq i < n)$ și un număr $k \in \{0, 1, \ldots, n-1\}$,
Output	k-mediana

k-mediana: descriere algoritm

Condiția pe care trebuie să o satisfacă la ieșire tabloul a este formulată de:

$$(\forall i)(i < k \implies a[i] \leq a[k]) \wedge (i > k \implies a[i] \geq a[k])$$

Fie j o poziție în a astfel încât:

$$(\forall i)(i < j \implies a[i] \leq a[j]) \wedge (i > j \implies a[i] \geq a[j])$$

Un asemenea j poate fi obținut cu algoritmul de partiționare de la `quickSort`.

1. $j = k \implies$ problema este rezolvată
2. $j < k \implies$ caută în `a[j+1..n]`
3. $j > k \implies$ caută în `a[1..j-1]`

k-mediana: algoritmul recursiv

Aceasta conduce la următoarea formulare recursivă a algoritmului de selectare:

```
@input: un tablou a cu n elemente, 0 ≤ k < n
@output: k-mediana
selecteaza(a, p, q, k) {
  partitioneaza(a, p, q, j);
  if (j == k) return a[k];
  if (j < k) selecteaza(a, j + 1, q, k);
  else selecteaza(a, p, j - 1, k);
}
```

k-mediana: algoritmul nerecursiv

Descrierea recursivă nu este avantajoasă deoarece produce un consum de memorie suplimentară (stiva apelurilor recursive) ce poate fi eliminat prin derecursivare:

```
@input: un tablou a cu n elemente, 0 ≤ k < n
@output: k-mediana
selecteaza(a, n, k) {
  p = 0;  1 = n-1;
  repeat
    partitioneaza(a, p, q, j);
    if (j < k) p = k1 + 1;
    if (k < j) q = k1 - 1;
  until (j == k);
  return a[k];
}
```

k-mediana: analiza

Analiza algoritmului `selecteaza` este asemănătoare cu cea a algoritmului `quickSort`. În cazul cel mai nefavorabil necesită $O(n^2)$ timp, iar pentru complexitatea medie se cunoaște următorul rezultat:

Teoremă

Complexitatea timp medie a algoritmului `Selecteaza` *este* $O(n)$*. Mai precis, numărul de comparații este aproximativ:*

$$n + k \log\left(\frac{n}{k}\right) + (n-k)\log\left(\frac{n}{n-k}\right).$$

Mediana in general: analiza

Proprietate: Fie dată o bară de lungime 1, care se taie arbitrar în două. Lungimea medie a bucății mai lungi este $\frac{3}{4}$.

Concluzie: daca se împarte în două un tablou de lungime n, lungimea celui mare subtablou este $\frac{3}{4}n$.

Mediana in general: analiza

Intuitiv: $\mathit{Exp\text{-}Time}(n) \leq (n-1) + \mathit{Exp\text{-}Time}_A(\frac{3}{4}n)$

Formal: Presupunem ca pivotul este ales aleatoriu.Avem
$\mathit{Exp\text{-}Time}(n) = \max_k \mathit{Exp\text{-}Time}(n, k)$

Afirmăm că $\mathit{Exp\text{-}Time}(n) \leq 4n$. Demonstrăm prin inducție.

$\mathit{Exp\text{-}Time}(n) \leq (n-1) + \sum_{i=n/2}^{n-1} \mathit{Exp\text{-}Time}(i) =$
$(n-1) + M(\mathit{Exp\text{-}Time}(\frac{n}{2}), \ldots, \mathit{Exp\text{-}Time}(n-1))$

Ipoteza inductivă: $\mathit{Exp\text{-}Time}(i) \leq 4i$, $i = n/2, \ldots, n-1$

Avem $\mathit{Exp\text{-}Time}(n) \leq (n-1) + M(4\frac{n}{2}, \ldots, 4(n-1)) \leq (n-1) + 4\frac{3}{4}n < 4n$

Un algoritm determinist liniar

1. grupează tabloul în $\frac{n}{5}$ grupe de 5 elemente și calculează mediana fiecărei grupe;
2. calculează recursiv mediana medianelor p
3. utilizează p ca pivot și separă elementele din tablou
4. apelează recursiv pentru subtabloul potrivit (în care se află k-mediana)

Un algoritm determinist liniar: analiza

Notații: $T(n, k)$ timpul pentru cazul cel mai nefavorabil pentru k-mediana,
$T_n = \max_k T(n, k)$
Pasul 1: $O(n)$
Pasul 2: $T(n/5)$
Pasul 3: $O(n)$
pasul 4: presupunând că cel puțin $\frac{3}{10}$ din tablou este $\leq p$ și că cel puțin $\frac{3}{10}$ din tablu este $\geq p$, pasul recursiv ia cel mult $T(\frac{7n}{10})$.

Se arată că
"cel puțin $\frac{3}{10}$ din tablou este $\leq p$ și că cel puțin $\frac{3}{10}$ din tablou este $\geq p$".

Comparați experimental cei doi algoritmi pentru mediană.