Outline

Cuprins

1 Timpul mediu: algoritmi determinişti

Motivaţie

Fie P o problemă, $p \in P$ instanţă a problemei P, A un algoritm care rezolvă P.

Reamintim formula pentru complexitatea timp în cazul cel mai nefavorabil:

$$T_A(n) = \sup\{\mathit{time}(A,p) \mid p \in P \land \mathit{size}(p) = n\}$$

Uneori, numărul instanţelor p cu $\mathit{size}(p) = n$ şi pentru care $\mathit{time}(A,p) = T_A(n)$ sau $\mathit{time}(A,p)$ are o valoare foarte apropiată de $T_A(n)$ este foarte mic.

Pentru aceste cazuri este preferabil să calculăm comportarea în medie a algoritmului.

Definiţie

Pentru a putea calcula comportarea în medie este necesar să privim mărimea $\mathit{time}(A,x)$ ca fiind o variabilă aleatorie:

- o experienţă = execuţia algoritmului pentru o instanţă x,
- valoarea experienţei = durata execuţiei algoritmului pentru instanţa p

şi să precizăm legea de repartiţie a acestei variabile aleatorie. Apoi, comportarea în medie se calculează ca fiind media acestei variabile aleatoare (considerăm numai cazul timpului de execuţie):

$$T_A^{med}(n) = M(\{\mathit{time}(A,x) \mid x \in P \land \mathit{size}(x) = n\})$$

Dacă mulţimea valorilor variabilei aleatoare $\mathit{time}(A,x) = \{t_0, t_1, \dots\}$ este finită sau numărabilă şi probabilitatea ca $\mathit{time}(A,x) = t_i$ este p_i, atunci media variabilei aleatorie $\mathit{time}(A,\_)$ (timpul mediu de execuţie) este:

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

Exemplu
Considerăm problema căutării unui element într-o secvenţă de numere întregi:

Problema FIRST OCCURRENCE
Input: $n, a = (a_0, \ldots, a_{n-1}), z$, toate numere întregi.
Output: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Presupunem că secvenţa $(a_0, \ldots, a_{n-1})$ este memorată în tabloul `a`. Considerăm ca dimensiune a problemei P_1 numărul n al elementelor din secvenţa în care se caută.

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
Mulţimea valorilor variabilei aleatorii $time(\text{FOAlg}, p)$ este $\{3i + 2 \mid 1 \leq i \leq n\}$ (s-au numărat doar atribuirile şi comparaţiile). În continuare trebuie să stabilim legea de repartiţie. Facem următoarele presupuneri:

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

Timpul mediu pentru FOAlg 2/2
Rezultă că probabilitatea ca $z \notin \{a_0, \ldots, a_{n-1}\}$ este $1 - q$. Acum probabilitatea ca $time(\text{FOAlg}, p) = 3i + 2$ $(poz = i - 1)$ este $\frac{q}{n}$, pentru $1 \leq i < n$, iar probabilitatea ca $time(\text{FOAlg}, p) = 3n + 2$ este $p_n = \frac{q}{n} + (1 - q)$.
Timpul mediu de execuţie este:

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

Pentru $q = 1$ (z apare totdeauna în secvenţă) avem $T^{med}_{\text{FOAlg}}(n) = \frac{3n}{2} + \frac{7}{2}$ şi pentru $q = \frac{1}{2}$ avem $T^{med}_{A}(n) = \frac{9n}{4} + \frac{11}{4}$.

1.1 Quicksort determinist

Quicksort: descriere
Este proiectat pe paradigma divide-et-impera.
Algoritmul Quicksort *Input*: $S = \{a_0, \ldots, a_{n-1}\}$ *Output*: o secvenţă cu elementele a_i în ordine crescătoare

1. Divizarea problemei constă în alegerea unei valori x din S. Elementul x este numit *pivot*. În general se alege pivotul $x = a_0$, dar nu este obligatoriu.
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 $S_<$ şi $S_>$ producând $Seq_<$ şi $Seq_>$, respectiv
4. întoarce secvenţa $Seq_<$, $S_=$, $Seq_>$

Quicksort: partiţionarea
Presupunem că S este memorată într-un tablou `a`. Următoarea soluţie utilizează un tablou suplimentar:

```
partition(out a, p, q, out k) {
  l = 0;
  for (i = p; i <= q; ++i) {
     if (a[i] <= a[p]) {
      b[l] = a[i]; l = l + 1;
    }
  }
  k = l - 1;
  for (i = p; i <= q; ++i) {
     if (a[i] > a[p]) {
      b[l] = a[i]; l = l + 1;
    }
  }
  a[p..q] = b;   // pseudocode
}
```

Quicksort: partiţionarea, eliminarea tabloului auxiliar, 1/5
Se determină prin interschimbări a unui indice k cu proprietăţile:

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

Partiţionarea tabloului se face prin interschimbări care menţin invariante proprietăţi asemănătoare cu cele de mai sus. Se consideră două variabile index: `i` cu care se parcurge tabloul de la stânga la dreapta şi `j` cu care se parcurge tabloul de la dreapta la stânga. Iniţial se ia $i = p + 1$ şi $j = q$. Proprietăţile menţinute invariante în timpul procesului de partiţionare sunt:

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

şi

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

Quicksort: partiţionarea, eliminarea tabloului auxiliar, 2/5
Presupunem că la momentul curent sunt interogate elementele `a[i]` şi `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]` şi se face `i = i+1` şi `j = j-1`, atunci ambele predicate (1) şi (2) sunt păstrate.

Quicksort: partiţionarea, eliminarea tabloului auxiliar, 3/5
Operaţiile 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 ((a[i] > x) && (x > a[j])) {
    swap(a, i, j);
    i = i+1;
    j = j-1;
  }
}
```

Quicksort: partiţionarea, eliminarea tabloului auxiliar, 4/5
Analiza terminării lui `while`:

- $i = j + 1$:
- din 1, 2 avem $a[i-1] \leq x$ şi $a[i] = a[j+1] \geq x$
- deci interschimbând `a[p]` cu `a[i−1]` obţinem partiţionarea dorită a tabloului $\Longrightarrow k = i - 1$

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

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

Quicksort: algoritmul de partiţionare, eliminarea tabloului auxiliar, 5/5

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

```
partition(out a, p, q, out 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 ((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

După sortarea recursivă a subtablourilor $\texttt{a}[\texttt{p}..\texttt{k}-1]$ şi $\texttt{a}[\texttt{k}+1..\texttt{q}]$ se observă că tabloul este sortat deja. Astfel partea de asamblare a soluţiilor este vidă.

`@input:` $a = (a[p], \ldots, a[q])$
`@output:` elementele secvenţei `a` în ordine crescătoare

```
qsort(out a, p, q) {
  if (p < q) {
    partition(a, p, q, k)
    qsort(a, p, k-1)
    qsort(a, k+1, q)
  }
}
```

Quicksort: timpul în cazul cel mai nefavorabil

- dimensiune instanţă: $\texttt{n} = \texttt{a.size}()$
- operaţii măsurate: comparaţii care implică elementele tabloului

Cazul cel mai nefavorabil se obţine atunci când la fiecare partiţionare se obţine una din subprobleme cu dimensiunea 1.

Deoarece operaţia de partiţionare necesită $q-p$ comparaţii, rezultă că pentru acest caz numărul de comparaţii este $(n-1)+(n-2)+\cdots+1 = O(n^2)$.

Acest rezultat este oarecum surprinzător, având în vedere că numele metodei este "sortare rapidă".

Aşa cum vom vedea, într-o distribuţie normală cazurile pentru care QuickSort execută n^2 comparaţii sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului.

Quicksort: timpul mediu

În continuare determinăm numărul mediu de comparaţii. Presupunem că $q+1-p = n$ (lungimea secvenţei) şi că probabilitatea ca pivotul x sa fie al k-lea element este $\frac{1}{n}$ (fiecare element al tabloului poate fi pivot cu aceeaşi probabilitate $\frac{1}{n}$). Rezultă că probabilitatea obţinerii subproblemelor de dimensiuni $k-p = i-1$ şi $q-k = n-i$ este $\frac{1}{n}$. În procesul de partiţionare, un element al tabloului (pivotul) este comparat cu toate celelalte, astfel că sunt necesare $n-1$ comparaţii. Acum numărul mediu de comparaţii se calculează prin formula:

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

Rezolvăm această recurenţă. Avem:

$$\begin{aligned} T^{med}(n) &= (n-1) + \frac{2}{n}(T^{med}(0) + \cdots + T^{med}(n-1)) \\ nT^{med}(n) &= n(n-1) + 2(T^{med}(0) + \cdots + T^{med}(n-1)) \end{aligned}$$

Trecem pe n în $n-1$:

$$(n-1)T^{med}(n-1) = (n-1)(n-2) + 2(T^{med}(0) + \cdots + T^{med}(n-2))$$

Scădem:

$$nT^{med}(n) = 2(n-1) + (n+1)T^{med}(n-1)$$

Împărţim prin $n(n+1)$ şi rezolvăm recurenţa obţinută:

$$\begin{aligned} \frac{T^{med}(n)}{n+1} &= \frac{T^{med}(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\ &= \frac{T^{med}(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} - \left(\frac{2}{(n-1)n} + \frac{2}{n(n+1)}\right) \\ &= \cdots \\ &= \frac{T^{med}(0)}{1} + \frac{2}{1} + \cdots + \frac{2}{n+1} - \left(\frac{2}{1\cdot 2} + \cdots + \frac{2}{n(n+1)}\right) \\ &= 1 + 2\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n+1}\right) - 2\left(\frac{1}{1\cdot 2} + \cdots + \frac{1}{n(n+1)}\right) \end{aligned}$$

Deoarece $1 + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log_2 n)$ şi seria $\sum \frac{1}{k(k+1)}$ este convergentă (şi deci şirul sumelor parţiale este mărginit), rezultă că $T(n) = O(n \log_2 n)$.
Am demonstrat următorul rezultat:

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

2 Timpul mediu: algoritmi probabilişti

2.1 Agoritmi nedeterminişti în general

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 condiţia B ales arbitrar

`failure;` – semnalează terminarea fără succes (e.g., o instrucţiune `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>
```

Problemă rezolvată de un program nedeterminist

- un program nedeterminist are mai multe *fire de execuţie*
- un program nedeterminist rezolvă P dacă $\forall x \in P$ $\exists$ un fir de execuţie care se termină şi a cărui configurţie finală include $P(x)$

Exemplu: problema celor N regine

Input: o tablă de şah $n \times n$. *Output:* o aşezare 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>
```

2.2 Agoritmi nedeterminişti pentru probleme de decizie

Definiţie

Agoritmi nedeterminişti pentru probleme de decizie: exemplu

SAT *Instance*: O mulţime finită de variabile şi o formulă propoziţională 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 instanţă SAT

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

Execuţie 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>
```

Execuţie 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>
```

Execuţie 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>
```

2.3 Algoritmi probabilişti

Definiţii
Exista două puncte de vedere:

- 1. algoritmul probabilist este văzut ca un algoritm nedeterminist pentru care există o distribuţie de probabilitate peste alegerile nedetermiste
- 2. algoritmul probabilist este un algoritm care are o intrare suplimentară ce constă într-o secvenţă de biţi aleatorii;
  - e echivalent cu a spune ca algoritmul probabilist constă în o mulţime de algoritmi determinişti din care un algoritm este ales aleatoriu pentru o intrare dată
  - pentru o intrare x a problemei date, calculele algoritmului probabilist pot diferi în funcţie de e secvenţa actuală de biţi aleatori

Această diferenţă poate fi proiectată în complexitate sau ieşire:

- timpul de execuţie văzut ca o variabilă aleatorie
- ieşirea văzută ca o variabilă aleatorie

La acest curs consideră doar prima variantă (numiţi şi algoritmi Las Vegas).

Timpul mediu de execuţie al algoritmilor probabilişti 1/2
Notaţii:

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

$time(A, C)$ = timpul necesar lui A ca să execute calculul C (un pic diferit faţă de cazul determinist)

Timpul mediu de execuţie al algoritmilor probabilişti 2/2
timpul mediu de execuţie a lui A pentru intrarea x este $\textit{exp-time}(A, x) = M[time] = \sum_C prob_{A,x}(C) \cdot time(A, C)$.

$time(A, \_)$ este variabilă aleatorie.

timpul mediu de execuţie a lui A în cazul cel mai nefavorabil este $\textit{exp-time}(A, n) = max\{\textit{exp-time}(A, x) \mid g(x) = n\}$

Dacă A este subînţeles din context, atunci scriem numai $\textit{exp-time}(n)$ ($\textit{exp-time}(x)$) în loc de $\textit{exp-time}(A, n)$ (resp. $\textit{exp-time}(A, x)$).

2.4 Quicksort probabilist

"Randomized Quicksort"
- exemplul canonic pentru algoritmii Las Vegas

Algoritmul RQS *Input*: $S = \{a_0, \ldots, a_{n-1}\}$ *Output*: 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 $S_<$ şi $S_>$ producând $Seq_<$ şi $Seq_>$, resp
4. întoarce secvenţa $Seq_<$, $Seq_=$, $Seq_>$

"Randomized Quicksort"
Se modifică doar algoritmul de partiţionare:

```
partition(out a, p, q, out k) {
  l = 0;
  k = p + random(q-p);
  for (i = p; i <= q; ++i) {
     if (a[i] <= a[k]) {
      b[l] = a[i]; l = l + 1;
    }
  }
  k = l - 1;
  for (i = p; i <= q; ++i) {
     if (a[i] > a[k]) {
      b[l] = a[i]; l = l + 1;
    }
  }
  a[p..q] = b;  // pseudocode
}
```

Analiza algoritmului RQS
Fie funcţia *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 comparaţiile dintre $a_{rank(i)}$ şi $a_{rank(j)}$

X_{ij} este variabilă aleatorie

Numărul mediu de comparaţii 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 $a_{rank(i)}$ şi $a_{rank(j)}$ să fie comparate într-o execuţie

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

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

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

Analiza algoritmului RQS

Theorem

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

3 k-mediana

k-mediana: problema

Definition

Fie S o listă cu n elemente dintr-o mulţime univers total ordonată. k-mediana este cel de-al k-lea element din lista sortată a elementelor din S. În alte cuvinte, k-mediana este un element $x \in \{a[0], \ldots, a[n-1]\}$ cu proprietăţile $|\{i \mid 0 \leq i < n \wedge a[i] < x\}| < k$ şi $|\{i \mid 0 \leq i < n \wedge a[i] \leq x\}| \geq k$ (dacă toate lelementele din S sunt distincte, atunci avem egalitate în ultima relaţie).

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

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

Evident, orice algoritm de sortare rezolvă problema de mai sus. Deoarece cerinţele pentru selecţie sunt mai slabe decât cele de la ordonare, se pune firesc întrebarea dacă există algoritmi mai performanţi decât cei utilizaţi la sortare.

k-mediana: descriere algoritm

Mediana poate fi obţinută cu algoritmul de partiţionare de la `quickSort`.

Condiţia pe care trebuie să o satisfacă la ieşire 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 poziţia calculată de algoritmul de partiţionare, i.e:

$$(\forall i)(i < j \implies a[i] \le a[j]) \wedge (i > j \implies a[i] \ge a[j])$$

Dacă $j = k$ atunci problema este rezolvată. Dacă $j < k$ atunci cel de-al k-lea cel mai mic element trebuie căutat în subtabloul `a[j+1..n]`, iar dacă $j > k$ atunci cel de-al k-lea cel mai mic element trebuie căutat în subtabloul `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
qselect(out a, p, q, k) {
  partition(a, p, q, j);
  if (j == k) return a[k];
  if (j < k) qselect(a, j + 1, q, k);
  else qselect(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
qselect(out a, n, k) {
  p = 0;  1 = n-1;
  repeat
    partition(a, p, q, j);
    if (j < k) p = k1 + 1;
    if (k < j) q = k1 - 1;
  until (j == k);
  return a[k];
}
```

qselect: analiza 1/3

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

Spaţiul continuu: Considerăm variabila aleatorie $Y = \max(u, 1-u)$.

$M(Y) = \int_0^1 M(Y \mid Y = u)du = \int_0^{\frac{1}{2}}(1-u)du + \int_{\frac{1}{2}}^1 udu = \frac{3}{4}$

Spaţiul discret: se împarte bara în n părţi egale de lungime $\frac{1}{n}$

n impar: lungimile segmentelor mai lungi sunt $\frac{k}{n}$, $k = \frac{n}{2} + 1, \ldots, n-1$, cu probabilitatea $\frac{2}{n-1}$. Rezultă media egală cu $\sum_{k=(n/2+1}^{n-1} \frac{k}{n} \cdot \frac{2}{n-1} = \frac{3n-1}{4n} \le \frac{3}{4}$

n par: lungimile segmentelor mai lungi sunt $\frac{k}{n}$, $k = \frac{n+1}{2}, \ldots, n-1$, cu probabilitatea $\frac{2}{n-1}$. Rezultă media egală cu $\sum_{k=(n+1)/2}^{n-1} \frac{k}{n} \cdot \frac{2}{n-1} = \frac{3n-4}{4n} \leq \frac{3}{4}$

La limită se obţine în ambele cazuri $\frac{3}{4}$.

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

qselect: analiza 2/3
exp-time(n, k) - timpul mediu pentru a găsi k-mediana

exp-time$(n) = \max_k$ *exp-time*(n, k)

$$\begin{aligned}
\textit{exp-time}(n) &\leq (n-1) + \frac{2}{n} \sum_{i=\frac{n}{2}}^{n-1} \textit{exp-time}(i) \\
&= (n-1) + \text{avg}[\textit{exp-time}(\frac{n}{2}), \ldots, \textit{exp-time}(n-1)]
\end{aligned}$$

(un raţionament similar ca la qsort)

qselect: analiza 3/3
Afirmăm că *exp-time*$(n) \leq 4n$. Demonstrăm prin inducţie.

Baza: $n = 1$

Pasul inductiv:

Ipoteza inductivă: *exp-time*$(i) \leq 4i$, $i = n/2, \ldots, n-1$

Avem

$$\begin{aligned}
\textit{exp-time}(n) &\leq (n-1) + \text{avg}[\textit{exp-time}(\frac{n}{2}), \ldots, \textit{exp-time}(n-1)] \\
&\leq (n-1) + \text{avg}[4\frac{n}{2}, \ldots, 4(n-1)] \\
&\leq (n-1) + 4\frac{3}{4}n \\
&< 4n
\end{aligned}$$

Un algoritm determinist liniar

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

Un algoritm determinist liniar: analiza

Notaţii: $T(n,k)$ 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 puţin $\frac{3}{10}$ din tablou este $\leq p$ şi că cel puţin $\frac{3}{10}$ din tablu este $\geq p$, pasul recursiv ia cel mult $T(\frac{7n}{10})$.
Însumând obţinem:

$$
\begin{aligned}
T(n) &\leq cn + T(\frac{n}{5}) + T(\frac{7n}{10}) \\
&\leq cn + c\frac{n}{5} + T(\frac{n}{5^2}) + T(\frac{7n}{5 \cdot 10}) + c\frac{7n}{10} + T(\frac{7n}{5 \cdot 10}) + T(\frac{7^2 n}{10^2}) \\
&\leq \ldots \\
&= O(n)
\end{aligned}
$$

(similar teoremei de master).

Demonstrarea afirmaţiei "cel puţin $\frac{3}{10}$ din tablou este $\leq p$ şi că cel puţin $\frac{3}{10}$ din tablou este $\geq p$": Fie $g = \frac{n}{5}$. Cel puţin $\lceil \frac{g}{2} \rceil$ dintre grupuri (cele cu mediana $\leq p$) au cel puţin trei elemente $\leq p$. Rezultă că numărul de elemente $\leq p$ este cel puţin $3\lceil \frac{g}{2} \rceil \geq \frac{3n}{10}$. Analog pentru numărul de elemente $\geq p$.

Comparaţi experimental cei doi algoritmi pentru mediană.