Outline

Cuprins

1 Reamintire cursul precedent

Algoritmi nedeterminişti

- pentru anumite stări există mai multe posibilităţi de a continua calculul (presupunem că acestea sunt în număr finit)
- în consecinţă, pentru aceeaşi intrare, algoritmul poate da rezultate (configuraţii finale) diferite
- instrucţiuni specifice:

  `choose` x `in` S`;` – întoarce un element din S ales arbitrar
  Timp de execuţie (uniform): $O(1)$

  `choose` x `in` S `s.t.` B`;` – întoarce un element din S care satisface condiţia B ales arbitrar
  Timp de execuţie (uniform): $T(B)$
- 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 configuraţie finală include $P(x)$

Agoritmi nedeterminişti pentru probleme de decizie

- Activitatea unui algoritm nedeterminist pentru o problemă de decizie:
  - "se ghiceşte" o anumită structură S
  - verifică dacă S satisface o proprietăţile cerute de întrebare
  - dacă proprietatea este satisfăcută, se apeleaza instrucţiunea de terminare `success`, altfel instrucţiunea de terminare `failure;`
- Extensia limbajului:

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

  `failure;` – semnalează terminarea fără succes

Algoritmi probabilişti

- caz particular de algoritmi nedeterminişti când există distribuţie de probabilitate peste mulţimea alegerilor
- `random(`n`)` - întoarce unumăr $x \in \{0, 1, \ldots, n-1\}$ ales aleatoriu uniform (i.e. cu probabilitatea $\frac{1}{n}$)
  Timp de execuţie: $O(1)$

- O variabilă aleatorie este o funcţie X definită peste un câmp de evenimente Ω
- Valoarea medie a lui X: $M(X) = \sum_i x_i \cdot p_i$
  Proprietăţi ale mediei:
  $M(X+Y) = M(X) + M(Y)$
  $M(X \cdot Y) = M(X) \cdot M(Y)$
  (X şi Y independente)

Algoritmi Monte Carlo
Pot produce rezultate incorecte cu o mică probabilitate, i.e. componenta de "output" este variabilă aleatorie.
Dacă un astfel de algoritm este executat de mai multe ori peste alegeri independente alese aleatoriu de fiecare dată, probabilitatea de eşec poate fi oricât de mică dar cu un timp de execuţie mai mare. [3ex] Exemplu: test de compoziţionalitate (primalitate)

Algoritmi Las Vegas
Nu produc niciodată rezultate incorecte, dar timpul poate varia de la o execuţie la alta (este variabilă aleatorie).
Alegerile aleatorii sunt făcute pentru a stabili timpul mediu de execuţie, care în esenţă este independent de intrare. [3ex] Exemplu: k-mediana

k-mediana: algoritm Las Vegas

```
randSelectRec(out a, p, q, k)
{
  j = randPartition(a, p, q);
  if (j == k) return a[j];
  if (j < k) return randSelectRec(a, j+1, q, k);
  return randSelectRec(a, p, j-1, k);
}

randSelect(out a, k)
{
  return randSelectRec(a, 0, a.size()-1, k);
}
```

randSelect: analiza 1/3
exp-time(n, k) - timpul mediu pentru a găsi k-mediana într-un tablou de lungime n

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

Pentru că analizăm timpul mediu în cazul cel mai nefavorabil, presupunem apelul recursiv alege tot timpul subtabloul mai mare

Reamintim că $M(CB) < \frac{3}{4}$

Rezultă că dacă se împarte aleatoriu în două un tablou de lungime n, lungimea medie a celui mare subtablou este cel mult $\frac{3}{4}n$.

randSelect: analiza 2/3

Lemă 1. *Lungimea medie a tabloului după i apeluri recursive este cel mult* $\left(\frac{3}{4}\right)^i n$.

Demonstrarea lemei.
L_i variabila aleatorie care dă lungimea tabloului după i apeluri.
P_j variabila aleatorie care dă partea fracţionară de elemente păstrate la nivelul j.
X_j variabila aleatorie care dă lungimea celui mare subtablou la nivelul j.
Avem: $L_i = n\prod_{j=1}^{i} P_j$, $P_j = \frac{X_j}{n}$ (presupunem că la fiecare apel recursiv se alege subtabloul cel mai lung), $M(P_j) = M\left(\frac{X_j}{n}\right) = \frac{M(X_j)}{n} \leq \frac{\frac{3}{4}n}{n} = \frac{3}{4}$,
$P_1, \ldots, P_n$ sunt independente,
$M(L_i) = M(n\prod_{j=1}^{i} P_j) = n\prod_{j=1}^{i} M(P_j) \leq \left(\frac{3}{4}\right)^i n$
Acum lema e demonstrată.

randSelect: analiza 3/3
La nivelul i, numărul de operaţii este liniar, pp. $\leq aX_i + b$ (reamintim că X_i dă lungimea celui mare subtablou).
Fie $r \leq n$ numărul de apeluri recursive. Timpul mediu:

$$\begin{aligned}
\textit{exp-time}(n) &= M\left(\sum_{i=1}^{r}(aX_i + b)\right) \\
&= \sum_{i=1}^{r} M(aX_i + b) \\
&\leq \sum_{i=1}^{n}(aM(X_i) + b) \\
&\leq \sum_{i=1}^{n}\left(a\left(\frac{3}{4}\right)^i n + b\right) \\
&= an\sum_{i=1}^{n}\left(\frac{3}{4}\right)^i + bn \\
&\leq 4an + bn \\
&= O(n)
\end{aligned}$$

2 Timpul mediu pentru algoritmi determinişti

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

Reamintim formula pentru complexitatea timp în cazul cel mai nefavorabil:

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

Uneori, numărul instanţelor p cu $\textit{size}(p) = n$ şi pentru care $\textit{time}(A, p) = T_A(n)$ sau $\textit{time}(A, p)$ 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 $time(A, x)$ 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):

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

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

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

3 Studii de caz

3.1 Prima apariţie într-o listă

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

Problema FIRST OCCURRENCE
Input: $n, a = (a_0, \dots, a_{n-1}), z$, 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 $(a_0, \dots, a_{n-1})$ 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;
```

Am văzut în cursul 3 că numărul de comparaţii în cazul cel mai nefavorabil este n.

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:

- probabilitatea ca $z \in \{a_0, \ldots, a_{n-1}\}$ este q şi
- probabilitatea ca z să apară prima dată pe poziţia i 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 $time^n_{\text{FOAlg}}(p) = i$ ($poz = i-1$) este $\frac{q}{n}$, pentru $2 \leq i < n$, iar probabilitatea ca $time^n_{\text{FOAlg}}(p) = n$ este $p_n = \frac{q}{n} + (1-q)$.
Timpul mediu de execuţie este:

$$\begin{aligned} \textit{exp-time}(n) &= \sum_{i=2}^{n} p_i x_i \\ &= \sum_{i=2}^{n-1} \frac{q}{n} \cdot i + (\frac{q}{n} + (1-q)) \cdot n \\ &= n - \frac{q}{n} - \frac{n-1}{2} \cdot q \end{aligned}$$

Pentru $q = \frac{1}{2}$ avem $\textit{exp-time}(n) = \frac{3n+1}{4} - \frac{1}{2n}$.

Pentru $q = 1$ avem $\textit{exp-time}(n) = \frac{n+1}{2} - \frac{1}{n}$.

3.2 Quicksort

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 $x = a_k$ 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
Putem utiliza algoritmul Lomuto pentru k-mediană:

```
partition(out a, p, q)
{
  pivot = a[q];
  i = p - 1;
  for (j = p; j < q; ++j)
    if (a[j] <= pivot) {
        i = i + 1;
        swap(a, i , j);
    }
  swap(a, i+1, q);
  return i + 1;
}
```

Quicksort: algoritm
După sortarea recursivă a subtablourilor `a`$[$`p..k` $-$ `1`$]$ şi `a`$[$`k` $+$ `1..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

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

qsort(out a)
{
  qsortRec(a, 0, n-1);
}
```

Quicksort: timpul în cazul cel mai nefavorabil

- dimensiune instanţă: `n` $=$ `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
Timpul mediu pentru `qsort` este egal cu timpul mediu pentru varianta probabilistă ("randomized quicksort").

"Randomized Quicksort", intuitiv
- 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* $x = a_k \in S$
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", algoritmic
Se utilizează algoritmul de partiţionare aleatorie:

```
randPartition(out a, p, q) {
  if (p < q) {
    i = p + random(q - p);
    swap(a, i, q);
    return partition(a, p, q);
  }
}

randQsortRec(out a, p, q) {
  if (p < q) {
    k = randPartition(a, p, q);
    randQsortRec(a, p, k-1);
    randQsortRec(a, k+1, q);
  }
}

RQS(out a) { randQsortRec(a, 0, n-1); }
```

Analiza algoritmului RQS: varianta 1 (1/4)
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 şi se numeşte indicator de variabilă

$C(n)$ - variabilă aleatorie care dă numărul de comparaţii

$C(n) = \sum_{j>i} X_{ij}$

Numărul mediu de comparaţii este

$M(C(n)) = M(\sum_{i=0}^{n-1} \sum_{j>i} X_{ij}) = \sum_{i=0}^{n-1} \sum_{j>i} M(X_{ij})$

Analiza algoritmului RQS: varianta 1 (2/4)
p_{ij} probabiltatea ca $a_{rank(i)}$ şi $a_{rank(j)}$ să fie comparate într-o execuţie

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

Presupunem $i < j$; $a_{rank(i)}$ şi $a_{rank(j)}$ sunt comparate dacă şi numai dacă primul pivot din secvenţa

$$a_{rank(i)}, \ldots, a_{rank(j)}$$

este $a_{rank(i)}$ sau $a_{rank(j)}$. Altfel pivotul separă cele două elemente în subsecvenţe diferite.

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

Analiza algoritmului RQS: varianta 1 (3/4)

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

Analiza algoritmului RQS: varianta 1 (4/4)
Avem:

$\sum_{k=1}^{n-i-1} \dfrac{1}{k} = H_{n-i-1} = \Theta(\log(n - i - 1))$,

$\sum_{i=0}^{n-2} \Theta(\log(n - i - 1)) = \Theta(\log(n-1)!) = \Theta(n \log n)$.

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

Exerciţii
Să se determine în aceeaşi manieră complexităţile medii pentru insertSort şi bubbleSort. Se vor considera două cazuri:

- se numără interschimbările (inversiunile) din tablou
- se numără deplasările elementelor din tablou

Indicii:
1. Probabilitatea ca perechea (i, j) $(i < j)$ să formeze o inversiune este $\dfrac{1}{2}$.
2. Numărul de deplasări este direct proporţional cu numărul de inversiuni.
În care dintre cele două cazuri există diferenţă?

Analiza algoritmului RQS: varianta 2 (recursiv) 1/2

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:

Analiza algoritmului RQS: varianta 2 (recursiv) 2/2

$$\begin{aligned} \textit{exp-time}(n) = M(C(n)) &= M((n-1) + \sum_{i=1}^{n} Y_i) \\ &= \begin{cases} (n-1) + \frac{1}{n}\sum_{i=1}^{n}(M(C(i-1)) + M(C(n-i)) & \text{, dacă } n \geq 1 \\ 1 & \text{, dacă } n = 0 \end{cases} \end{aligned}$$

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 `RQS` este $O(n \log_2 n)$.

Rezolvare recurenţă
Notaţie: $c(n) = M(C(n))$

$$c(n) = (n-1) + \frac{2}{n}(c(0) + \cdots + c(n-1))$$
$$nc(n) = n(n-1) + 2(c(0) + \cdots + c(n-1))$$

Trecem pe n în $n-1$:

$$(n-1)c(n-1) = (n-1)(n-2) + 2(c(0) + \cdots + c(n-2))$$

Scădem:

$$nc(n) = 2(n-1) + (n+1)c(n-1)$$

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

$$\begin{aligned}
\frac{c(n)}{n+1} &= \frac{c(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\
&= \frac{c(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} - \left(\frac{2}{(n-1)n} + \frac{2}{n(n+1)}\right) \\
&= \cdots \\
&= \frac{c(0)}{1} + \frac{2}{1} + \cdots + \frac{2}{n+1} - \left(\frac{2}{1\cdot 2} + \cdots + \frac{2}{n(n+1)}\right) \\
&= O(\log n) - O(1)
\end{aligned}$$

3.3 Şuruburi şi piuliţe (Nuts and Bolts)

Domeniul problemei

- n şuruburi şi n piuliţe de mărimi aproximativ egale
- dar nu se poate decide care şurub (piuliţă) e mai mare şi care e mai mic(ă) (nu se pot compara direct două şuruburi între ele; la fel pentru piuliţe)
- fiecare piuliţă se potriveşte exact cu un şurub
- când se compară un şurub cu o piuliţă, se poate vedea dacă se potrivesc sau dacă şurubul este mai mare sau mai mic decât piuliţă
- sarcina de rezolvat constă în găsirea piuliţei potrivite pentru fiecare şurub

Obs. Abia în 1995 s-a proiectat un algoritm care rezolvă problema în timpul $O(n \log n)$:
Janos Komlos, Yuan Ma, and Endre Szemeredi. Sorting nuts and bolts in $O(nlogn)$ time, SIAM J. Discrete Math 11(3):347-372, 1998. Technical Report MPI-I-95-1-025, Max-Planck-Institut für Informatik, September 1995.

Versiunea simplificată
Presupunem că trebuie să găsim piuliţa potrivită pentru un şurub dat.

Fie `cmp`(N, B) o funcţie care întoarce:
-1 dacă piuliţa N este mai mică decât şurubul B,
0 dacă piuliţa N se potriveşte cu şurubul B,
1 dacă piuliţa N este mai mare decât şurubul B.

`cmp`(N, B) este defintă pentru toate datele de intrare.

```
NutsAndBolt (NL, B)
{
  n = NL.size();
  for (i=0; i < n - 1; ++i)
    if (cmp(NL[i], B) == 0) return i;
  return n - 1;
}
```

Numărul mediu de comparaţii: varianta 1
Presupunem că ordinea piuliţelor în lista NL (care coincide cu ordinea în care sunt testate) este o variabilă aleatorie uniformă.
Rezultă că probabilitatea ca piuliţa căutată să fie pe poziţia i este $\frac{1}{n}$ (de ce?).
Fie $C(n)$ variabila aleatorie care dă numărul de comparaţii.
Presupunem că NL este nevidă şi totdeauna există o piuliţă care se potriveşte cu B.
Valorile posibile pentru $C(n)$: $1, 2, \ldots, n-1$.
Probabilitatea ca $C(n) = i$, $1 \leq i < n-1$: $\frac{1}{n}$
Probabilitatea $C(n) = n-1$: $\frac{2}{n}$

Calculul numărului mediu de comparaţii

$$\begin{aligned} M(C(n)) &= \sum_{i=1}^{n-2} i \cdot \frac{1}{n} + (n-1) \cdot \frac{2}{n} \\ &= \sum_{i=1}^{n-1} \frac{i}{n} + \frac{n-1}{n} \\ &= \frac{n(n-1)}{2n} + \frac{n-1}{n} \\ &= \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

Numărului mediu de comparaţii: varianta 2 (recursiv) 1/2
Dacă $n > 1$, primul element este totdeauna testat (i.e. probabilitatea de testare a lui este 1). Reamintim că probabilitatea ca algoritmul să se oprească după primul test este $\frac{1}{n}$, rezultă că restul listei este procesată cu probabilitatea $1 - \frac{1}{n} = \frac{n-1}{n}$.
Y - variabila aleatoare care dă numărul mediu de comparaţii $M(C(n-1))$ pentru apelul recursiv dacă prima piuliţă nu e potrivită, 0 altfel.

Avem relaţia:
$C(1) = 0$, $C(n) = 1 + Y$ pentru $n > 0$
de unde rezultă
$M(C(n)) = 1 + M(Y) = 1 + \frac{n-1}{n} M(C(n-1))$ pentru $n > 1$.

Numărul mediu de comparaţii: varianta 2 (recursiv) 2/2
Notăm $c(n) = nM(C(n))$.

Rezultă: $c(1) = 0$, $c(n) = n + c(n-1)$ pentru $n > 1$.

Avem: $c(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$.

care implică $M(C(n)) = \frac{c(n)}{n} = \frac{n+1}{2} - \frac{1}{n}$.

Cazul general, soluţia 1 (naivă)

```
NutsAndBolts (NL, BL) {
  M = emptyList;  n = NL.size();
  forall B in BL {
    i = 0;
    while (i < n)
      if (cmp(NL[i], B) == 0) {
        M.pushBack([B, NL[i]]);
        i = n;
      }
  }
  return M;
}
```

Numărul de comparaţii în cazul cel mai nefavorabil =
Numărul mediu de comparaţii =
$O(n^2)$
Soluţia de mai sus nu ţine seama de semnificaţiile valorilor -1 şi 1 întoarse de `cmp`.

Cazul general, soluţia 2: ideea
Soluţia e de natură recursivă.
Pasul curent:

1. se alege un şurub ca pivot şi se compară cu toate piuliţele;
2. se ia piuliţa care se potriveşte cu şurubul pivot şi se compară cu toate celelalte şuruburi;
3. după cele $2(n-1)$ comparaţii, se împarte fiecare dintre cele două mulţimi (liste) în două: mulţimea şuruburilor|piuliţelor mai mici, respectiv mai mari, decât şurubul|piuliţa pivot;

Apeluri recursive:

- un apel pentru potrivirea piuliţelor mai mici cu şuruburi mai mici
- un apel pentru potrivirea piuliţelor mai mari cu şuruburi mai mari

Exerciţiu. Să se scrie în Alk şi să se testeze algoritmul de mai sus.

Analiza în cazul cel mai nefavorabil

$$\begin{aligned} C(0) &= 0 \\ C(n) &= 2n - 1 + \max_{k=1,\ldots,n} (C(k-1) + C(n-k)) \\ &= 2n - 1 + C(n-1) \end{aligned}$$

Rezolvând recurenţa, obţinem

$$C(n) = \sum_{i=1}^{n} (2n-1) = O(n^2)$$

Numărul mediu de comparaţii 1/3
Presupunem că şurubul pivot este ales aleatoriu uniform.
Fie B_i al i-lea şurub în lista sortată. Similar N_j.
X_{ij} variabila aleatorie (indicatorul de variabilă) care întoarce 1 dacă B_i şi N_j sunt comparate, 0 altfel.
Presupunem $i < j$. B_i şi N_j sunt comparate ddacă primul pivot din $B_i, \ldots, B_j$ este B_i sau B_j (de ce?).
Rezultă $M(X_{ij}) = Pr(X_{ij} = 1) = \dfrac{2}{j+1-i}$, $i < j$.
Relaţia simetrică este obţinută la fel: $M(X_{ij}) = \dfrac{2}{i+1-j}$, $i > j$.
Dacă $i = j$, $M(X_{ij}) = 1$ (deşi aceste comparaţii ar putea fi evitate).

Numărul mediu de comparaţii 2/3
Deoarece $C(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}$, rezultă

$$\begin{aligned} M(C(n)) &= M(\sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} M(X_{ij}) \\ &= \sum_{i=1}^{n} M(X_{ii}) + 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} M(X_{ij}) \\ &= n + 4 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j+1-i} \\ &= n + 4(nH_n - 2n + H_n) \end{aligned}$$

Numărul mediu de comparaţii 3/3

S-a utilizat:

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j+1-i} &= \sum_{i=1}^{n}\sum_{k=2}^{n+1-i}\frac{1}{k}\\ &= \sum_{k=2}^{n}\sum_{i=1}^{n+1-k}\frac{1}{k}\\ &= \sum_{k=2}^{n}\frac{n+1-k}{k}\\ &= (n+1)\sum_{k=2}^{n}\frac{1}{k}-\sum_{k=2}^{n}1\\ &= (n+1)(H_n-1)-(n-1)\end{aligned}$$

Deoarece $H_n = \sum_{k=1}^{n}\frac{1}{k} = \Theta(\log n)$, rezultă

$$M(C(n)) = \Theta(n\log n)$$

Numărul mediu de comparaţii, recursiv 1/3

Presupunem că pivot este ales aleatoriu uniform şi că un şurub B este egal probabil al k-lea cel mai mic element în listă, i.e. B apare pe poziţia k în lista sortată cu probabilitatea $\frac{1}{n}$.

Fie Y_k variabila aleatoare care întoarce numărul mediu de comparaţii dat de apelurile recursive dacă pivotul este al k-lea element, 0 altfel:

$$Y_k = \begin{cases} M(C(k-1)) + M(C(n-k)) & \text{dacă pivotul este } B_k\\ 0 & \text{altfel}\end{cases}$$

Avem

$$C(n) = 2n-1+\sum_{k=1}^{n}Y_k$$

$M(Y_k) = \frac{1}{n}(M(C(k-1)) + M(C(n-k)))$.

Numărul mediu de comparaţii, recursiv 2/3

$$\begin{aligned}M(C(n)) &= M(2n-1+\sum_{k=1}^{n}Y_k)\\ &= 2n-1+\sum_{k=1}^{n}M(Y_k)\\ &= 2n-1+\frac{1}{n}\sum_{k=1}^{n}(M(C(k-1))+M(C(n-k))\\ &= 2n-1+\frac{2}{n}\sum_{k=0}^{n-1}M(C(k))\end{aligned}$$

de unde obţinem

$$nC(n) = n(2n-1) + 2\sum_{k=0}^{n-1} M(C(k))$$

care implică

$$nC(n) = (n+1)C(n-1) + 4n - 3$$

Numărul mediu de comparaţii, recursiv 3/3

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{4}{n+1} - \frac{3}{n(n+1)}$$
$$\frac{C(n-1)}{n} = \frac{C(n-2)}{n-1} + \frac{4}{n} - \frac{3}{(n-1)n)}$$
$$\ldots$$
$$\frac{C(2)}{3} = \frac{C(1)}{2} + \frac{4}{2} - \frac{3}{1(2)}$$

de unde obţinem

$$\frac{C(n)}{n+1} = \frac{C(1)}{2} + \Theta(\log n) - \Theta(1)$$

care implică

$$C(n) = \Theta(n \log n)$$

Reducerea la sortare

- două şuruburi se pot compara indirect prin intermediul piuliţelor (cum?)
- la fel, două piuliţe se pot compara indirect prin intermediul şuruburilor (cum?)
- se definesc funcţii de comparare (care este timpul pentru comparare?)
- se apelează un algoritm de sortare pentru ordonarea crescătoare a şuruburilor şi, respectiv, piuliţelor cu funcţiile de comparare corespunzătoare
- se întorc perechile $(BL[i], NL[i])$ din listele sortate

Reducerea sortării la NutsAndBolts

- se duplică lista de sortat
- o copie joacă rolul şuruburilor, cealaltă a piuliţelor
- se apelează algoritmul NutsAndBolts
- se "contopeşte" lista sortată de perechi întoarsă de NutsAndBolts într-o listă sortată a elementelor iniţiale

3.4 Treaps

Definiţie

- combină structura de arbore binar de căutare cu cea de max-heap
- structura de arbore binar de căutare se face pa baza unei chei `key` asociate fiecărei valori
- structura de max-heap se face pa baza unei priorităţ `pri` asociate fiecărei valori (diferită de cheie)
- operaţia de căutare se face ca în arborii binari de căutare

Exemplu pe tablă.

Raimund Seidel and Cecilia R. Aragon. Randomized search trees. Algorithmica, 16(4/5):464-497, 1996. Available at `https://faculty.washington.edu/aragon/pubs/rst96.pdf`.

Operaţia de inserare

- se inserează elementul într-un nod frunză ca la arborii de căutare
- restaurare proprietate max-heap: prin operaţii de rotaţie se merge înapoi spre rădăcină până când se întâlneşte un nod cu prioritate mai mare sau egală (ca la max-heap)

Exemplu pe tablă.

Operaţia de ştergere

- este inversă operaţiei de inserare
- prin operaţii de rotaţie se merge spre frontieră (interschimbând de fiecare dată cu nodul copil cu prioritate mai mare)

Numărul mediu de comparaţii pentru căutare 1/3

- presupunem că valorile cheilor sunt $1, 2, \ldots, n$
- indicator de variabilă: A_{ij} este variabila aleatorie care întoarce 1 ddacă i este un predecesor al lui j
- i va avea prioritatea maximă pentru toate cheile din intervalul $[\min(i,j)..\max(i,j)]$
- rezultă $M(A_{ij}) = \dfrac{1}{|i-j|+1}$
- numărul de operaţii pentru un nod j (căutat/inserat/şters) este proporţional cu adâncimea în arbore a lui j
- definim $depth(j) = \sum_i A_{ij} - 1$ (se consideră că rădăcina are adâncimea 0)

Numărul mediu de comparaţii pentru căutare 2/3

$$\begin{aligned} M(\mathit{depth}(j)) &= M(\sum_i A_{ij} - 1) \\ &= \sum_i M(A_{ij}) - 1 \\ &= \sum_i \left(\frac{1}{|i-j|+1}\right) - 1 \\ &= \sum_{i=1}^{j} \left(\frac{1}{|i-j|+1}\right) + \sum_{i=j+1}^{n} \left(\frac{1}{|i-j|+1}\right) - 1 \\ &= \sum_{k=1}^{j} \frac{1}{k} + \sum_{k=2}^{n-j+1} \frac{1}{k} - 1 \\ &= H_j + H_{n-j+1} - 2 \end{aligned}$$

Numărul mediu de comparaţii pentru căutare 3/3

$\max_j \mathit{depth}(j)$ este atins pentru $j = \dfrac{n+1}{2}$, care este egală cu

$$2H_{\frac{n+1}{2}} - 2 = 2\ln n + O(1)$$

Aceasta ne dă o margine superioară pentru numărul mediu de comparaţii.

Un treap aleatoriu (randomized) este un treap un obţinut prin generarea aleatorie uniformă a priorităţilor (numere reale în $[0, 1]$) la inserare;

Dacă cheile sunt inserate în ordinea descrescătoare a priorităţilor, atunci rezultatul este un treap.

Analiza de mai sus este valabilă şi pentru arbori binari aleatori (construiţi cu inserţii aleatorii, fără priorităţi).

Quicksort din nou

Se consideră următorul algoritm de sortare:

```
T = un arbore binar de căutare vid
inserează cheile în T în ordine aleatorie
returnează secvenţa inordine a lui T
```

Să se arate că `RQS` poate fi uşor transformat într-un algoritm probabilist care construieşte arborel binar de căutare. [2ex] Se obţine aceeaşi complexitate medie?