Sortare

SD 2017/2018

Conținut

Problema sortării

- Forma 1:
  - Intrare: n, $(v_0, ..., v_{n-1})$
  - Ieșire: $(w_0, ..., w_{n-1})$ astfel încât $(w_0, ..., w_{n-1})$ este o permutare a $(v_0, ..., v_{n-1})$ si $w_0 \leq ... \leq w_{n-1}$
- Forma 2:
  - Intrare: n, $(R_0, ..., R_{n-1})$ cu cheile $k_0, ..., k_{n-1}$
  - Ieșire: $(R'_0, ..., R'_{n-1})$ astfel încât $(R'_0, ..., R'_{n-1})$ este o permutare a $(R_0, ..., R_{n-1})$ și $R'_0.k_0 \leq ... \leq R'_{n-1}.k_{n-1}$
- Structura de date
  Tablou $a[0..n-1]$
  $a[0] = v_0, ..., a[n-1] = v_{n-1}$

Conținut

Sortare prin interschimbare (*bubble-sort*)

- Principiul de bază:
  - (i, j) cu $i < j$ este o inversiune dacă $a[i] > a[j]$
  - Cât timp există o inversiune $(i, i+1)$ interschimbă $a[i]$ cu $a[i+1]$

- Algoritm:

Procedure *bubbleSort(a, n)*
begin
    $ultim \leftarrow n - 1$
    while $(ultim > 0)$ **do**
        $n1 \leftarrow ultim–1; ultim \leftarrow 0$
        for $i \leftarrow 0$ **to** $n1$ **do**
            if $(a[i] > a[i+1])$ **then**
                swap$(a[i], a[i+1])$
                $ultim \leftarrow i$
end

Sortare prin interschimbare - exemplu

3 7 2 1 4 (n1 = 3)
3 **7 2** 1 4
3 **2 7** 1 4
3 2 **7 1** 4
3 2 **1 7** 4
3 2 1 **7 4**
3 2 1 **4 7**
3 2 1 4 **7**

3 2 1 4 7 (n1 = 2)
2 3 1 4 7
2 **3 1** 4 7
2 **1 3** 4 7
2 1 **3 4** 7
2 1 **3 4** 7

2 1 3 4 7 (n1 = 0)
1 2 3 4 7
1 2 3 4 7

Sortare prin interschimbare

- Analiza
  - Cazul cel mai nefavorabil
    $a[0] > a[1] > ... > a[n-1]$
    Timp căutare: $O(n-1+n-2+...+1) = O(n^2)$
    $T_{bubbleSort}(n) = O(n^2)$
  - Cazul cel mai favorabil: $O(n)$

Conținut

Sortare prin inserție directă

- Principiul de bază:
  presupunem $a[0..i-1]$ sortat
  inserează $a[i]$ astfel încât $a[0..i]$ devine sortat

- Algoritm (căutarea poziției lui $a[i]$ secvențial):

```
Procedure insertSort(a, n)
begin
    for i ← 1 to n − 1 do
        j ← i–1 // a[0..i − 1] sortat
        temp ← a[i] // caut locul lui temp
        while ((j ≥ 0) and (a[j] > temp)) do
            a[j + 1] ← a[j]
            j ← j–1
        if (a[j + 1]! = temp) then
            a[j + 1] ← temp
end
```

Sortare prin inserție directă

- Exemplu
  3 **7** 2 1
  3 7 **2** 1
  2 3 7 **1**
  1 2 3 7
- Analiza
  - căutarea poziției i în $a[0..j-1]$ necesită $O(j-1)$ pași
  - cazul cel mai nefavorabil $a[0] > a[1] > ... > a[n-1]$
    Timp căutare: $O(1+2+...+n-1) = O(n^2)$
    $T_{insertSort}(n) = O(n^2)$
  - Cazul cel mai favorabil: $O(n)$

Conținut

Sortare prin selecție

- Se aplică următoarea schemă:
  - pasul curent: selectează un element și-l duce pe poziția sa finală din tabloul sortat;
  - repetă pasul curent până când toate elementele ajung pe locurile finale.

- După modul de selectare a unui element:
  - Selecție naivă: alegerea elementelor în ordinea în care se află inițial (de la $n-1$ la 0 sau de la 0 la $n-1$)
  - Selecție sistematică: utilizare max-heap

Sortare prin selecție naivă

- În ordinea $n-1, n-2, ..., 1, 0$, adică:
  $(\forall i) 0 \leq i < n \implies a[i] = max\{a[0], ..., a[i]\}$

```
Procedure naivSort(a, n)
begin
    for i ← n − 1 downto 1 do
        imax ← i
        for j ← i − 1 downto 0 do
            if (a[j] > a[imax]) then
                imax ← j
        if (i! = imax) then
            swap(a[i], a[imax])
end
```

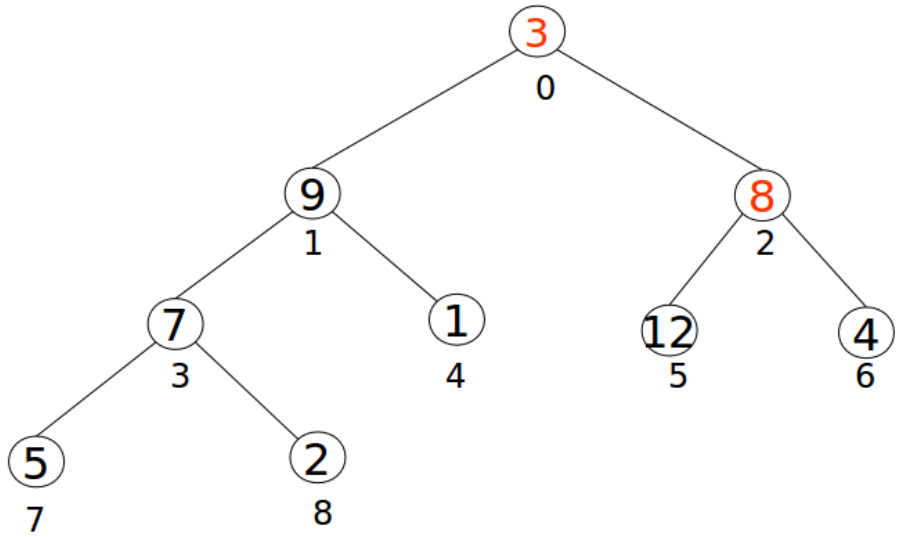
- Complexitatea timp în toate cazurile este $O(n^2)$

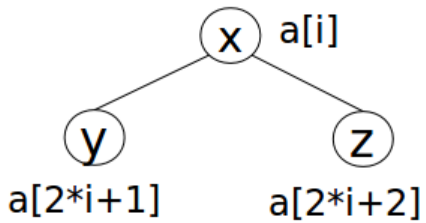
Heap sort (sortare prin selecție sistematică)

Etapa I

- organizează tabloul ca un max-heap: $(\forall k) 1 \leq k \leq n-1 \implies a[k] \leq a[(k-1)/2]$;
- inițial tabloul satisface proprietatea max-heap începând cu poziția $n/2$;
- introduce în max-heap elementele de pe pozițiile $n/2-1, n/2-2, \cdots, 1, 0$.

3	9	8	7	1	12	4	5	2
0	1	2	3	4	5	6	7	8

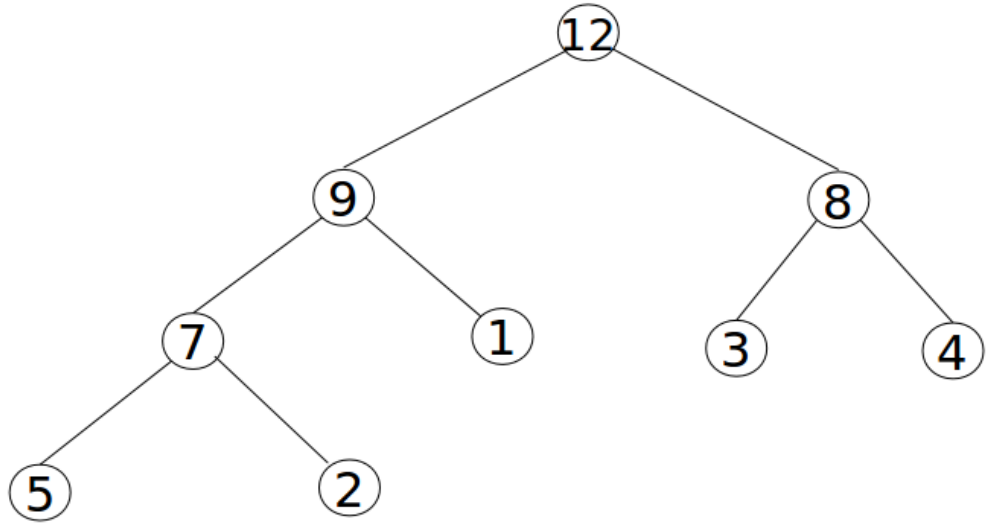




Heap sort (sortare prin selecție sistematică)

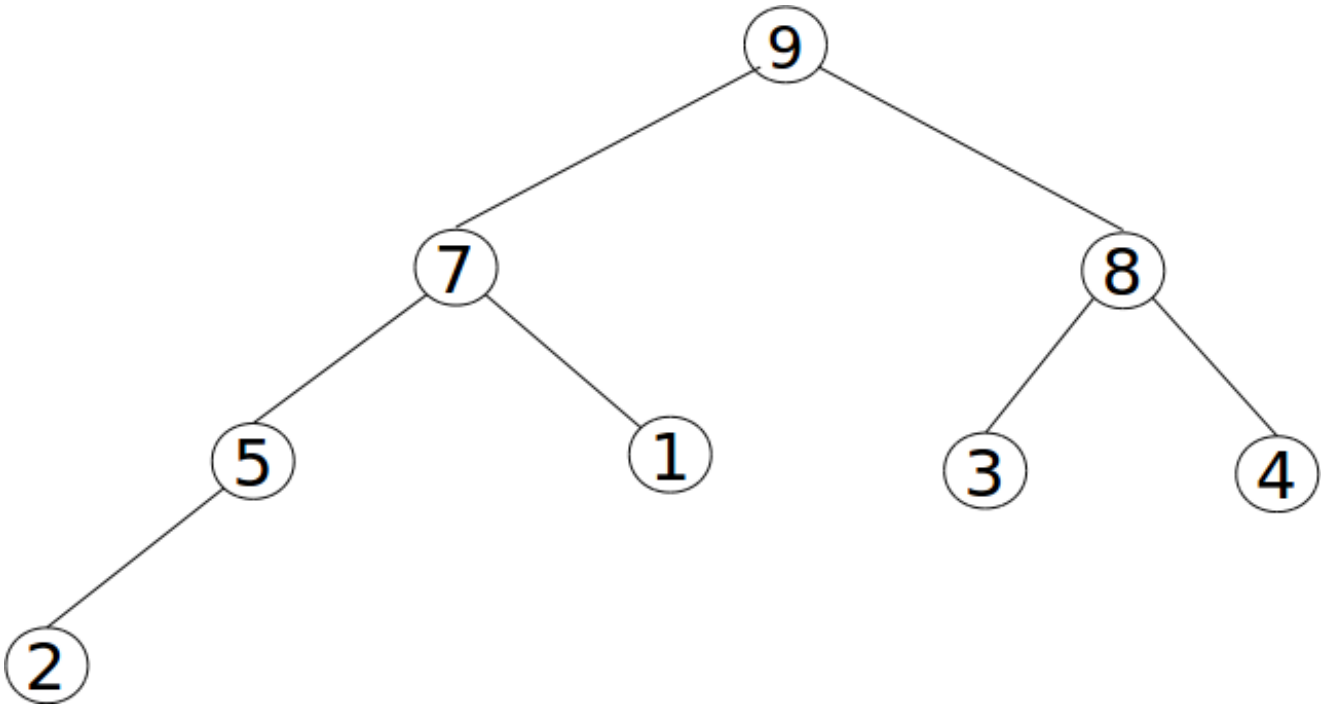
Etapa II

- selectează elementul maxim și-l duce la locul lui prin interschimbare cu ultimul;
- micșorează n cu 1 și apoi reface max-heapul;
- repetă pașii de mai sus până când toate elementele ajung pe locul lor.



Heap sort (sortare prin selecție sistematică)

9	7	8	5	1	3	4	2	12
0	1	2	3	4	5	6	7	8



Operația de introducere în heap

```
Procedure insereazaAlTlea(a, n, t)
begin
    j ← t
    heap ← false
    while ((2 * j + 1 < n) and not heap) do
        k ← 2 * j + 1
        if ((k < n − 1) and (a[k] < a[k + 1])) then
            k ← k + 1
        if (a[j] < a[k]) then
            swap(a[j], a[k])
            j ← k
        else
            heap ← true
end
```

Heap sort (sortare prin selecție sistematică)

```
Procedure heapSort(a, n)
begin
    // construieste maxheap-ul
    for t ← (n − 1)/2 downto 0 do
        insereazaAITlea(a, n, t)
    // elimina
    r ← n − 1
    while (r > 0) do
        swap(a[0], a[r])
        insereazaAITlea(a, r, 0)
        r ← r − 1
end
```

Heap sort - Exemplu

10	17	5	23	7	(n = 5)
10	**17**	5	23	7	
10	23	5	17	7	
23	10	5	17	7	
23	17	5	10	7	(max-heap n)

Heap sort - Exemplu

23	17	5	10	7	(max-heap n)
7	17	5	10	23	
17	10	5	7	23	(max-heap n-1)
7	10	5	17	23	
10	7	5	17	23	(max-heap n-2)
5	7	10	17	23	
7	5	10	17	23	(max-heap n-3)
5	7	10	17	23	
5	7	10	17	23	(max-heap n-4)
5	7	10	17	23	

Heap sort - complexitate

- formarea heap-ului (pp. $n = 2^k - 1$)
  $\sum_{i=0}^{k-1} 2(k-i-1)2^i = 2^{k+1} - 2(k+1)$
- eliminarea din heap si refacerea heap-ului
  $\sum_{i=0}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$
- complexitate algoritm de sortare
  $T_{heapSort}(n) = 2nlogn - 2n = O(nlogn)$

Conținut

Paradigma *divide-et-impera*

- $P(n)$: problemă de dimensiune n
- baza:
  - dacă $n \leq n_0$ atunci rezolvă P prin metode elementare
- *divide-et-impera*:
  - **divide** P în a probleme $P_1(n_1), ..., P_a(n_a)$ cu $n_i \leq n/b, b > 1$
  - **rezolvă** $P_1(n_1), ..., P_a(n_a)$ în aceeași manieră și obține soluțiile $S_1, ..., S_a$
  - **asamblează** $S_1, ..., S_a$ pentru a obține soluția S a problemei P

Paradigma *divide-et-impera*: algoritm

Procedure *DivideEtImpera(P, n, S)*
begin
    if $(n \leq n_0)$ **then**
        determină S prin metode elementare
    else
        Împarte P in $P_1, ..., P_a$
        DivideEtImpera(P_1, n_1, S_1)
        ...
        DivideEtImpera(P_a, n_a, S_a)
        Asambleaza$(S_1, ..., S_a, S)$
end

Sortare prin interclasare (*Merge sort*)

- generalizare: $a[p..q]$
- baza: $p \geq q$
- *divide-et-impera*
  - divide: $m = [(p + q)/2]$
  - subprobleme: $a[p..m]$, $a[m + 1..q]$
  - asamblare: interclasează subsecvențele sortate $a[p..m]$ și $a[m + 1..q]$
    - inițial memorează rezultatul interclasării în *temp*
    - copie din $temp[0..q - p + 1]$ în $a[p..q]$
- complexitate:
  - timp: $T(n) = O(n \ log \ n)$
  - spațiu suplimentar: $O(n)$

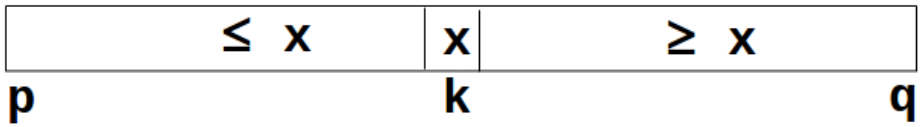
Interclasarea a două secvențe sortate

- problema:
  - date $a[0] \leq a[1] \leq \cdots \leq a[m-1]$, $b[0] \leq b[1] \leq \cdots \leq b[n-1]$, să se construiască $c[0] \leq c[1] \leq \cdots \leq c[m+n-1]$ a.î. $(\forall k)((\exists i)c[k] = a[i]) \vee (\exists j)c[k] = b[j])$ iar pentru $k! = p$, $c[k]$ și $c[p]$ provin din elemente diferite
- soluția
  - inițial: $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$
  - pasul curent:
    - daca $a[i] \leq b[j]$ atunci $c[k] \leftarrow a[i]$, $i \leftarrow i + 1$
    - daca $a[i] > b[j]$ atunci $c[k] \leftarrow b[j]$, $j \leftarrow j + 1$
    - $k \leftarrow k + 1$
  - condiția de terminare: $i > m - 1$ sau $j > n - 1$
  - daca e cazul, copie în c elementele din tabloul neterminat

Conținut

Sortare rapidă (*Quick sort*)

- generalizare: $a[p..q]$
- baza: $p \geq q$
- *divide-et-impera*
  - divide: determină k între p și q prin interschimbări a.î. după determinarea lui k avem:
    - $p \leq i \leq k \implies a[i] \leq a[k]$
    - $k < j \leq q \implies a[k] \leq a[j]$



  - subprobleme: $a[p..k-1]$, $a[k+1..q]$
  - asamblare: nu există

Quick sort: partiționare

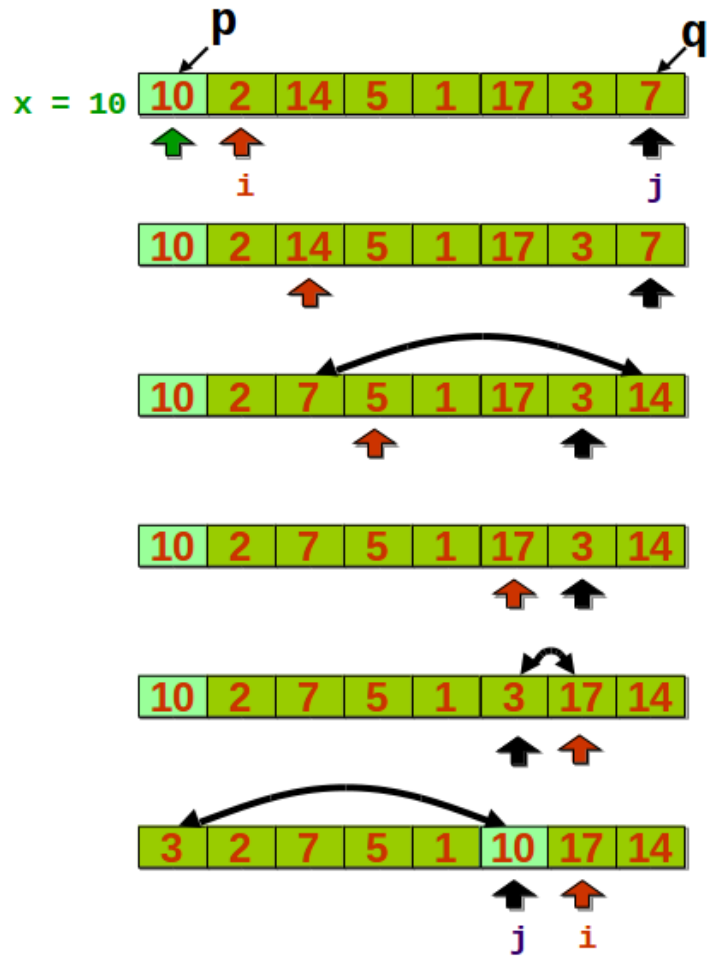
- inițial:
  - $x \leftarrow a[p]$ (se poate alege x arbitrar din $a[p..q]$)
  - $i \leftarrow p+1; j \leftarrow q$
- pasul curent:
  - dacă $a[i] \leq x$ atunci $i \leftarrow i+1$
  - dacă $a[j] \geq x$ atunci $j \leftarrow j-1$
  - dacă $a[i] > x > a[j]$ si $i < j$ atunci
    $swap(a[i], a[j])$
    $i \leftarrow i+1$
    $j \leftarrow j-1$
- terminare:
  - condiția $i > j$
  - operații
    $k \leftarrow i-1$
    $swap(a[p], a[k])$

Quick sort: partiționare - exemplu

Procedure *partitioneaza(a, p, q, k)*
begin

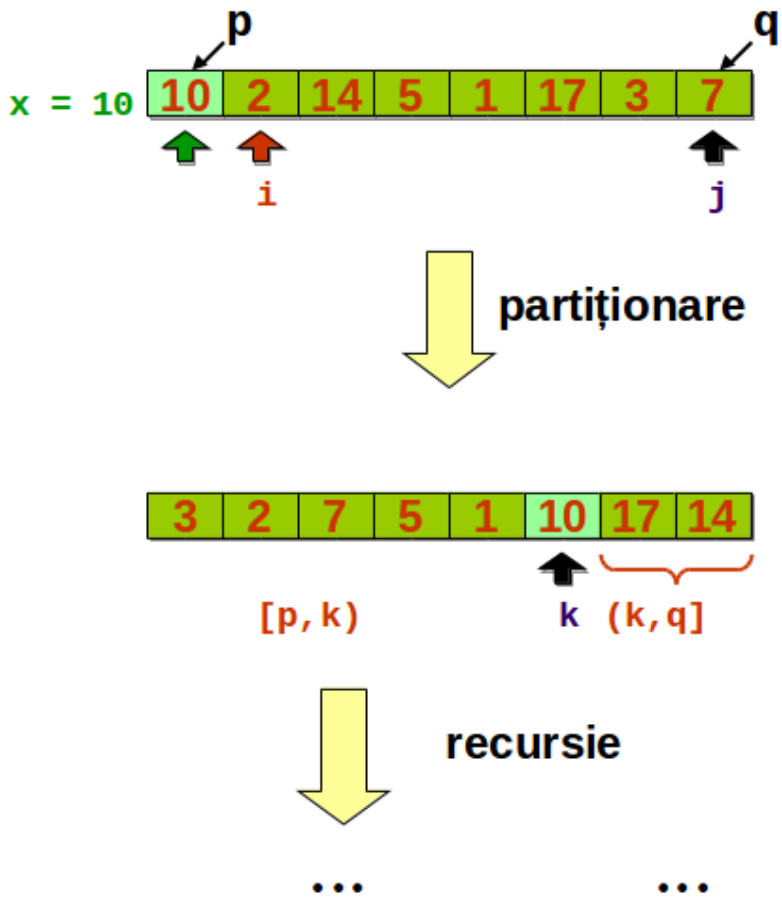
$x \leftarrow a[p]$
$i \leftarrow p + 1$
$j \leftarrow q$
while $(i <= j)$ **do**
if $(a[i] <= x)$ **then**
$i \leftarrow i + 1$
if $(a[j] >= x)$ **then**
$j \leftarrow j - 1$
if $(i < j)$ *and* $(a[i] > x)$ *and* $(x > a[j])$
then
$swap(a[i], a[j])$
$i \leftarrow i + 1$
$j \leftarrow j - 1$
$k \leftarrow i - 1$
$a[p] \leftarrow a[k]$
$a[k] \leftarrow x$

end

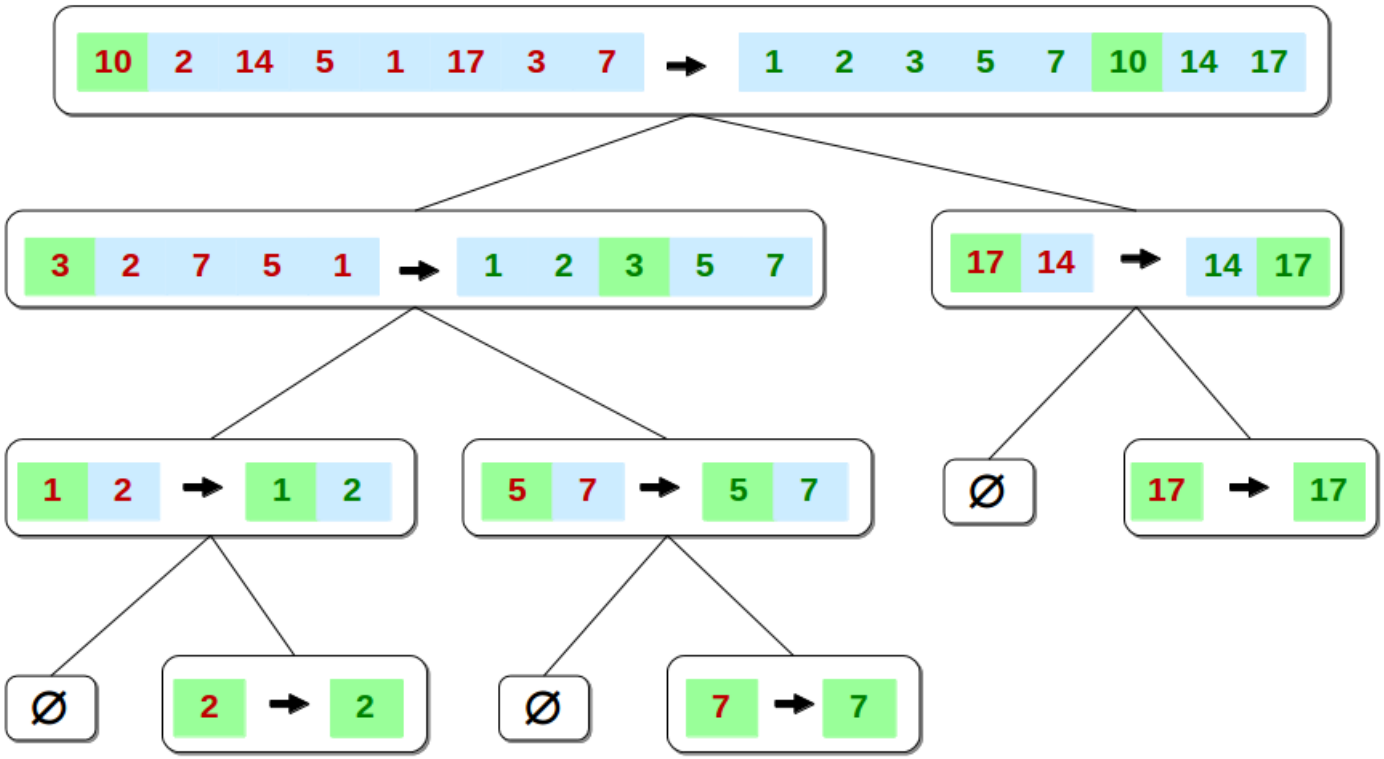


Quick sort: recursie - exemplu

Procedure *quickSort(a, p, q)*
begin
    while $(p < q)$ **do**
        $partitioneaza(a, p, q, k)$
        $quickSort(a, p, k-1)$
        $quickSort(a, k+1, q)$
end

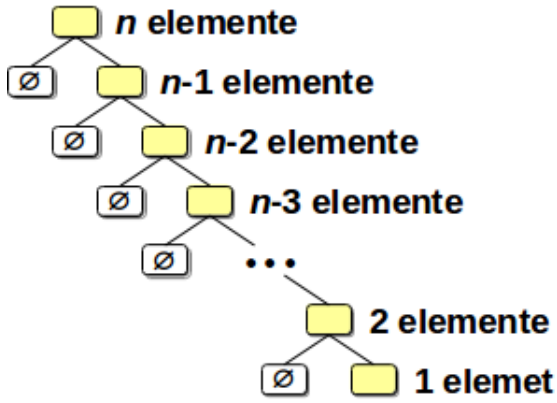


Quick sort: arbore de recursie



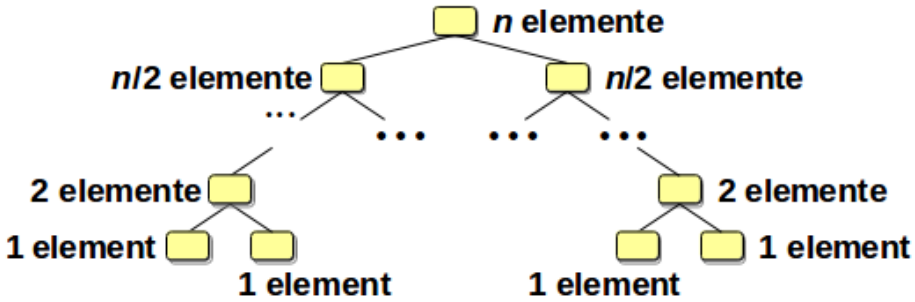
Quick sort - complexitate

- Alegerea pivotului influențează eficiența algoritmului
- Cazul cel mai nefavorabil: pivotul este cea mai mică (cea mai mare) valoare. Timp proporțional cu $n + n - 1 + ... + 1$.
- $T_{quickSort}(n) = O(n^2)$
- Arborele de recursie:



Quick sort - complexitate

- Un pivot "bun" împarte tabloul în două subtablouri de dimensiuni comparabile
- Înălțimea arborelui de recursie este $O(log\ n)$
- Complexitatea medie este $O(n\ log\ n)$



Conținut

Sortare prin numărare

- Ipoteză: $a[i] \in \{1, 2, \ldots, k\}$
- Se determină poziția fiecărui element în tabloul sortat numărând câte elemente sunt mai mici decât acesta

```
1  Procedure countingSort(a, b, n, k)
2  begin
3      for i ← 1 to k do
4          c[i] ← 0
5      for j ← 0 to n − 1 do
6          c[a[j]] ← c[a[j]] + 1
7      for i ← 2 to k do
8          c[i] ← c[i] + c[i − 1]
9      for j ← n − 1 downto 0 do
10         b[c[a[j]] − 1] ← a[j]
11         c[a[j]] ← c[a[j]] − 1
12 end
```

Complexitate: $O(k + n)$

Sortare prin numărare – exemplu ($k = 6$)

	0	1	2	3	4	5	6	7
a	3	6	4	1	3	4	1	4

liniile 5-6

	1	2	3	4	5	6
c	2	0	2	3	0	1

liniile 7-8

	1	2	3	4	5	6
c	2	2	4	7	7	8

	0	1	2	3	4	5	6	7
b								4

	1	2	3	4	5	6
c	2	2	4	6	7	8

liniile 9-11, j = 7

	0	1	2	3	4	5	6	7
b		1						4

	1	2	3	4	5	6
c	1	2	4	6	7	8

liniile 9-11, j = 6

	0	1	2	3	4	5	6	7
b		1					4	4

	1	2	3	4	5	6
c	1	2	4	5	7	8

liniile 9-11, j = 5

---

tabloul sortat:

	0	1	2	3	4	5	6	7
b	1	1	3	3	4	4	4	6

Conținut

Sortare prin distribuire

- Ipoteză: Elementele $a[i]$ sunt distribuite uniform peste intervalul $[0, 1)$

- Principiu:
  - se divide intervalul $[0, 1)$ în n subintervale de mărimi egale, numerotate de la 0 la $n - 1$;
  - se distribuie elementele $a[i]$ în intervalul corespunzător: $\lfloor n \cdot a[i] \rfloor$;
  - se sortează fiecare pachet folosind o altă metodă;
  - se combină cele n pachete într-o listă sortată.

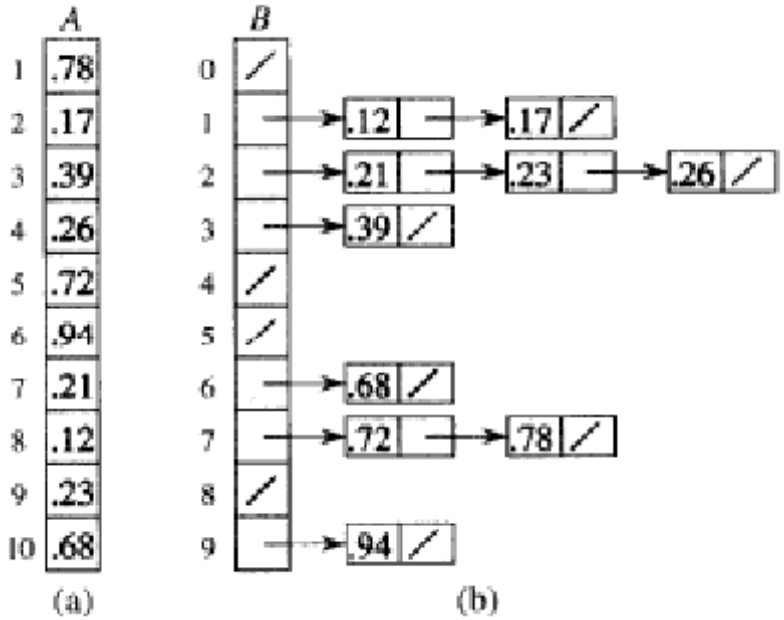
Sortare prin distribuire

- Algoritm:

Procedure *bucketSort(a, n)*
begin
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $insereaza(B[\lfloor n \cdot a[i] \rfloor], a[i])$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        sortează lista $B[i]$
    concatenează în ordine listele $B[0], B[1], \cdots, B[n-1]$
end

Complexitatea medie: $O(n)$

Sortare prin distribuire – exemplu



(Cormen T.H. et al., Introducere în algoritmi)

Sortare - complexitate

Algoritm	**Caz**		
	favorabil	mediu	nefavorabil
bubbleSort	n	n^2	n^2
insertSort	n	n^2	n^2
naivSort	n^2	n^2	n^2
heapSort	$n \log n$	$n \log n$	$n \log n$
mergeSort	$n \log n$	$n \log n$	$n \log n$
quickSort	$n \log n$	$n \log n$	n^2
countingSort	$-$	$n + k$	$n + k$
bucketSort	$-$	n	$-$

Când utilizăm un anumit algoritm de sortare?

- O metodă de sortare este *stabilă* dacă păstrează ordinea relativă a elementelor cu chei identice

- Recomandări
  - *Quick sort*: când nu e nevoie de o metodă stabilă și performanța medie e mai importantă decât cea în cazul cel mai nefavorabil; $O(n \ log \ n)$ complexitatea timp medie, $O(log \ n)$ spațiu suplimentar
  - *Merge sort*: când este necesară o metodă stabilă; complexitate timp $O(n \ log \ n)$; dezavantaje: $O(n)$ spațiu suplimentar, constanta mai mare decât cea a QuickSort
  - *Heap sort*: când nu e nevoie de o metodă stabilă și ne interesează mai mult performanța în cazul cel mai nefavorabil decât în cazul mediu; timp $O(n \ log \ n)$, spațiu $O(1)$
  - *Insert sort*: când n e mic

Când utilizăm un anumit algoritm de sortare?

- In anumite condiții, este posibilă o sortare în $O(n)$
- *Counting sort*: valori dintr-un interval
- *Bucket sort*: valorile sunt distribuite aproximativ uniform