Sortare

SD 2014/2015

Conținut

- Sortare bazată pe comparații
  - sortare prin interschimbare
  - sortare prin inserţie
  - sortare prin selecţie
    - naivă
    - sistematică (“heap sort”)
  - sortare prin interclasare (“merge sort”)
  - sortare rapidă (“quick sort”)
- Sortare prin numărare
- Sortare prin distribuire

Problema sortării

- Forma 1
  - Intrare: n, $(v_0, \ldots, v_{n-1})$
  - Ieşire: $(w_0, \ldots, w_{n-1})$ astfel încât
    $(w_0, \ldots, w_{n-1})$ este o permutare a $(v_0, \ldots, v_{n-1})$, şi
    $w_0 \leq \ldots \leq w_{n-1}$

- Forma 2
  - Intrare: n, $(R_0, \ldots, R_{n-1})$ cu cheile $k_0, \ldots, k_{n-1}$
  - Ieşire: $(R'_0, \ldots, R'_{n-1})$ astfel încât
    $(R'_0, \ldots, R'_{n-1})$ este o permutare a $(R_0, \ldots, R_{n-1})$,
    şi
    $R'_0.k_0 \leq \ldots \leq R'_{n-1}.k_{n-1}$

- structura de date
  Tablou **`a[0..n-1]`**
  `a[0]` $= v_0, \ldots,$ **`a[n-1]`** $= v_{n-1}$

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 ← n-1
    while (ultim > 0) do
        n1 ← ultim - 1; ultim ←0
        for i ← 0 to n1 do
            if (a[i] > a[i+1]) then
                swap(a[i], a[i+1])
                ultim ← 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)

2 **1** 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)

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 poziţiei lui a[i] secvenţial):

```
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 inserţie directă

- Exemplu

3	**7**	2	1
3	7	**2**	1
2	3	7	**1**
1	2	3	7

- Analiza
  - căutarea poziţiei i în a[0..j-1] necesită O(j-1) paşi
  - 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)

Sortare prin selecţie

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

- După modul de selectare a unui element:
  - Selecţie naivă: alegerea elementelor în ordinea în care se află iniţial (de la n-1 la 0 sau de la 0 la n-1);
  - Selecţie sistematică: utilizarea unui max-heap.

Sortare prin selecţie naivă

- în ordinea n-1, n-2,..., 1, 0, adică:

  $(\forall i)\ 0 \le i < n \Rightarrow a[i] = \max\{a[0],\ldots,a[i]\}$

```
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
```

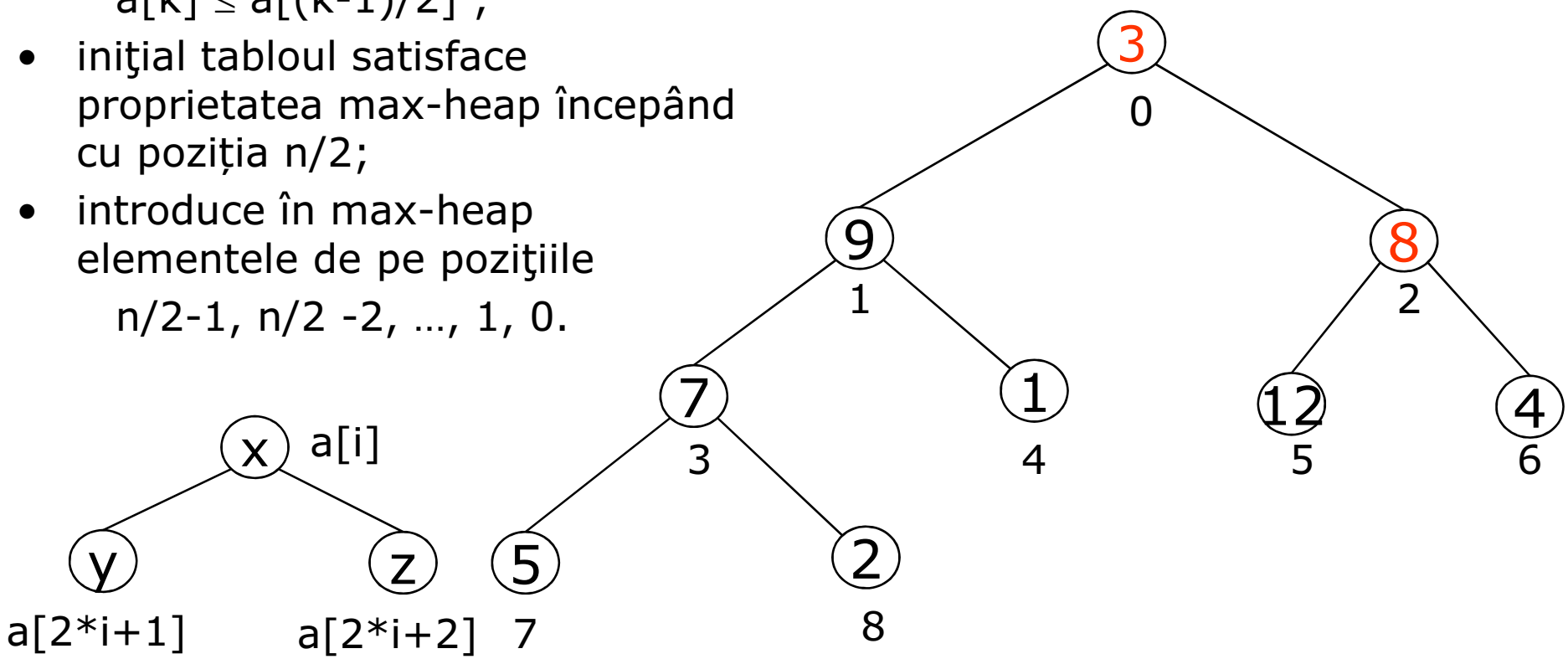
- complexitatea timp toate cazurile este $O(n^2)$

“Heap sort”
(sortare prin selecţie sistematică)

Etapa I

- organizează tabloul ca un max-heap: $(\forall k)\ 1 \le k \le n-1 \Rightarrow$
  $a[k] \le a[(k-1)/2]$;
- iniţial tabloul satisface proprietatea max-heap începând cu poziția n/2;
- introduce în max-heap elementele de pe poziţiile
  n/2-1, n/2 -2, …, 1, 0.

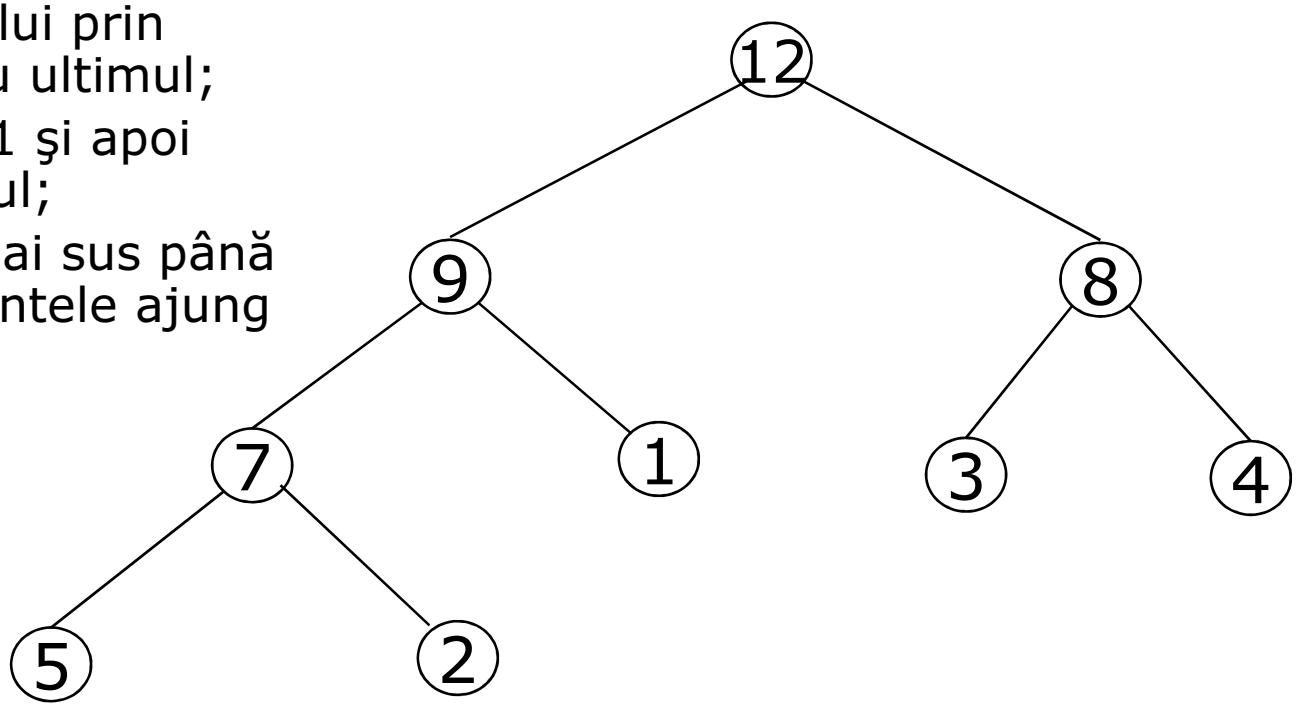
3	9	8	7	1	12	4	5	2
0	1	2	3	4	5	6	7	8



“Heap sort”
(sortare prin selecţie sistematică)

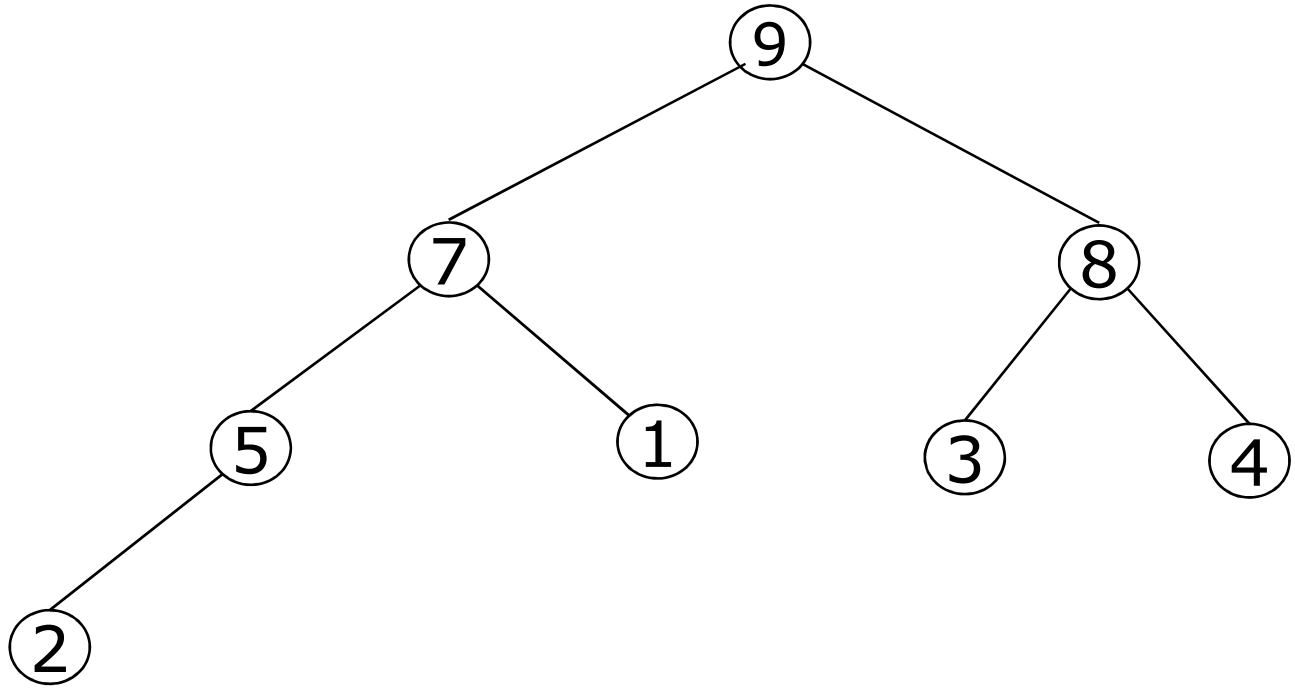
Etapa II

- selectează elementul maxim şi îl duce la locul lui prin interschimbare cu ultimul;
- micşorează n cu 1 şi apoi reface max-heapul;
- repetă paşii de mai sus până când toate elementele ajung pe locul lor.



“Heap sort”
(sortare prin selecţie sistematică)

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



Operaţia 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 selectie sistematică)

```
procedure heapSort(a, n)
begin
   // construieste maxheap-ul
   for t ← (n-1)/2 downto 0 do
      insereazaAlTlea(a, n, t)
   // elimina
   r ← n-1
   while (r > 0) do
      swap(a[0], a[r])
      insereazaAlTlea(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 și refacerea heap-ului

$$\sum_{i=0}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$$

- complexitate algoritm de sortare

$$T_{\text{heapSort}}(n) = 2n\log n - 2n = O(n\log n)$$

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 aceeaşi manieră şi obţine soluţiile S_1, ..., S_a
  - ***asamblează*** S_1, ..., S_a pentru a obţine soluţia S a problemei P

Paradigma divide-et-impera: algoritm

```
procedure DivideEtImpera(P, n, S)
begin
   if (n <= n0) then
      determina S prin metode elementare
   else
      imparte P in P1, ..., Pa
      DivideEtImpera(P1, n1, S1)
      ...
      DivideEtImpera(Pa, na, Sa)
      Asambleaza(S1, ..., Sa, S)
end
```

Sortare prin interclasare ("Merge sort")

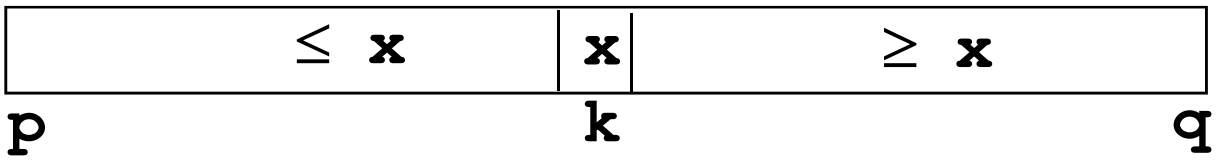
- generalizare: **`a[p..q]`**
- baza: **`p` ≥ `q`**
- divide-et-impera
  - divide: **`m = [(p + q)/2]`**
  - subprobleme: **`a[p..m], a[m+1..q]`**
  - asamblare: interclasează subsecvenţele sortate **`a[p..m]`** şi **`a[m+1..q]`**
    - iniţial memorează rezultatul interclasării în **`temp`**
    - copie din **`temp[0..p+q-1]`** în **`a[p..q]`**
- complexitate:
  - timp : T(n) = O(n log n)
  - spaţiu suplimentar: O(n)

Interclasarea a două secvenţe sortate

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

Sortare rapidă ("Quick sort")

- generalizare: **a[p..q]**
- baza: **p ≥ q**
- divide-et-impera
  - divide: determină **k** între **p** şi **q** *prin interschimbări* a.î. după determinarea lui **k** avem:
    - **p ≤ i ≤ k ⇒ a[i] ≤ a[k]**
    - **k < j ≤ q ⇒ a[k] ≤ a[j]**



⇨subprobleme: **a[p..k-1], a[k+1..q]**

⇨asamblare: nu există

Quick sort: partiţionare

- iniţial:
  - **x** ← **a[p]** (se poate alege **x** arbitrar din a**[p..q]**)
  - **i** ← **p+1** ; **j** ← **q**
- pasul curent:
  - dacă **a[i]** ≤ **x** atunci **i** ← **i+1**
  - dacă **a[j]** ≥ **x** atunci **j** ← **j-1**
  - dacă **a[i] > x > a[j]** si **i < j** atunci
    - **swap(a[i], a[j])**
    - **i** ← **i+1**
    - **j** ← **j-1**
- terminare:

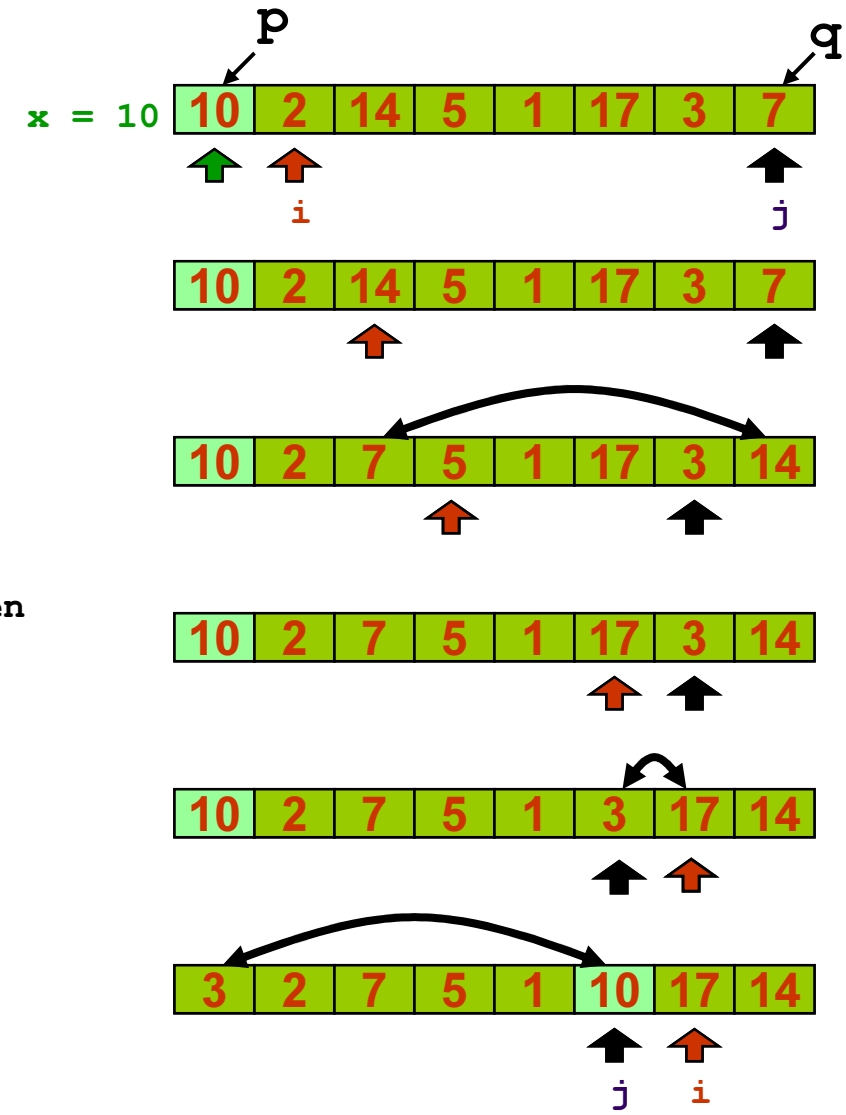
  condiţia **i > j**

  operaţii **k** ← **i-1**

  swap(a[p], a[k])

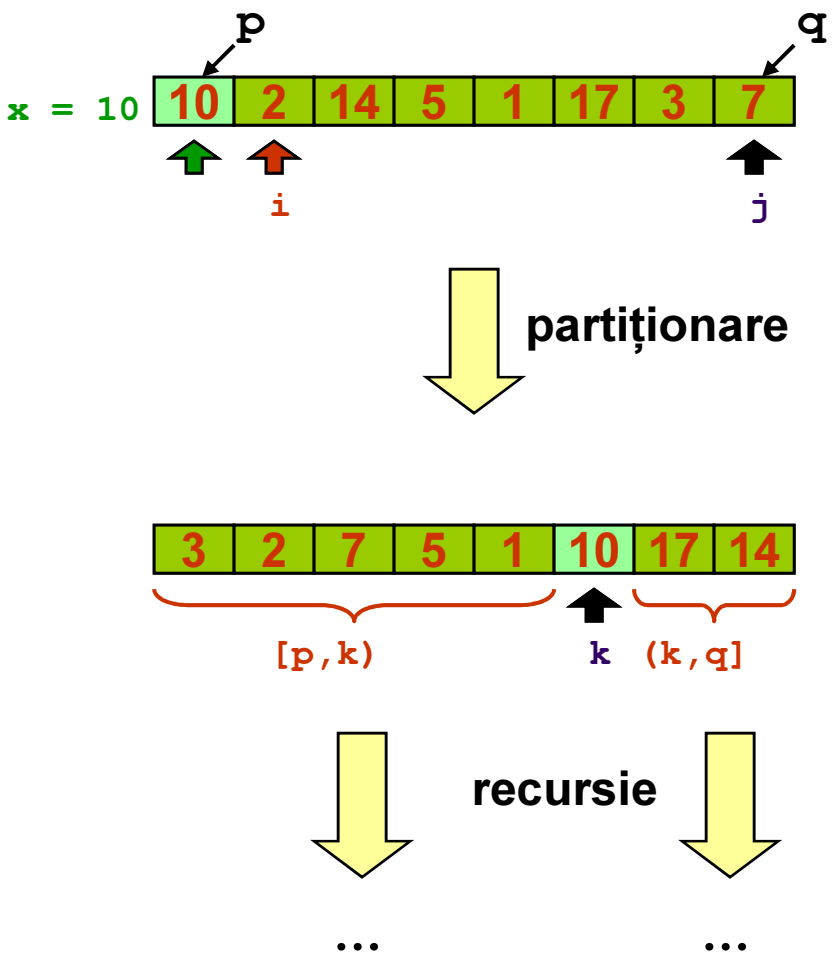
Quick sort: partiționare - exemplu

```
procedure partitioneaza(a, p, q, k)
begin
  x ← a[p]
  i ← p+1
  j ← q

  while (i <= j) do
    if (a[i] <= x) then i ← i+1
    if (a[j] >= x) then j ← j-1
    if (i<j) and (a[i]>x) and (x>a[j]) then
       swap(a[i], a[j])
       i ← i+1
       j ← j-1

  k ← i-1
  a[p] ← a[k]
  a[k] ← x
end
```

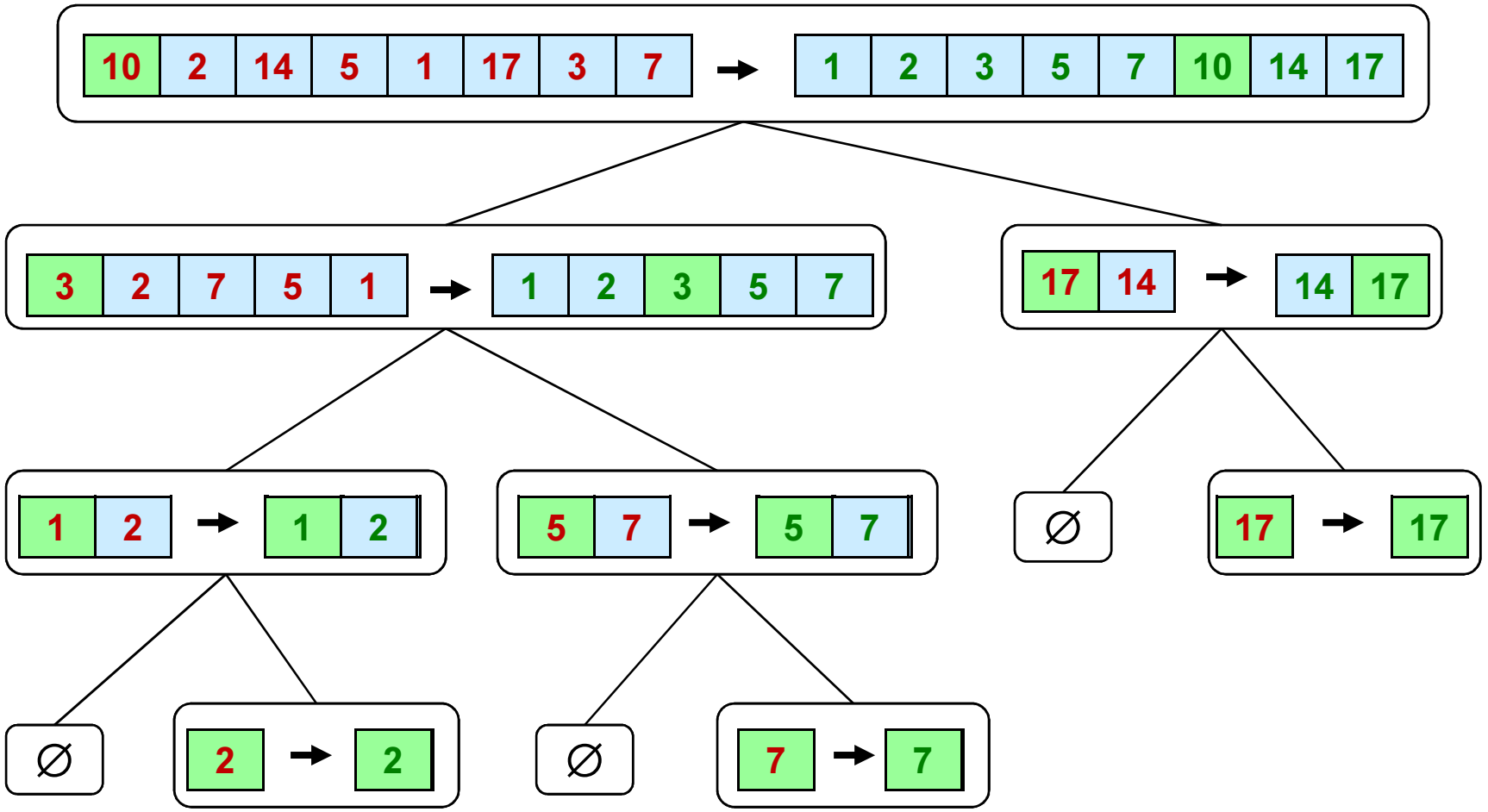


Quick sort: recursie - exemplu

```
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
```

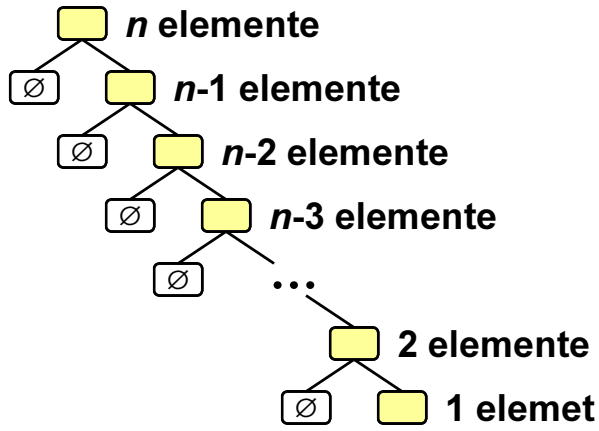


Quick sort: arbore de recursie



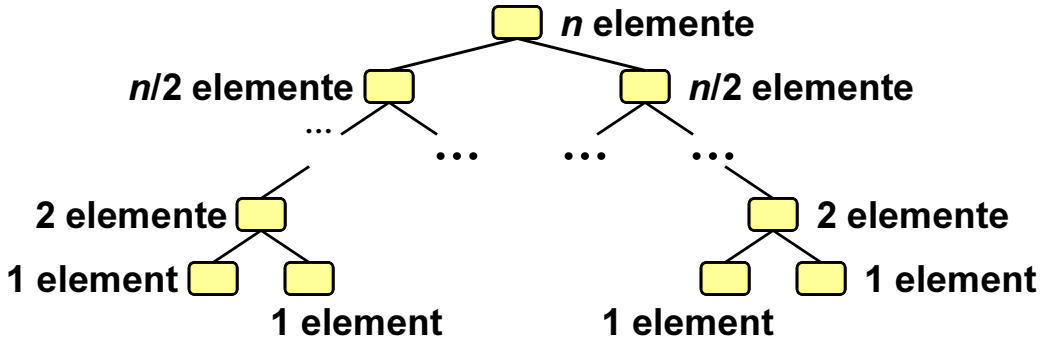
Quick sort - complexitate

- 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" imparte tabloul în două subtablouri de dimensiuni comparabile
- Înălțimea arborelui de recursie este *O(log n)*
- Complexitatea medie este *O(n log n)*



Sortare prin numărare

- Ipoteză: a[i] ϵ {1, 2,..., k}
- Se dermină poziția fiecărui element în tabloul sortat numărând câte elemente sunt mai mici decât el.
- Algoritm:

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

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

linia 4

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

linia 5

	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 6-8, 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 6-8, 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 6-8, j = 5

---

tabloul sortat:

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

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.

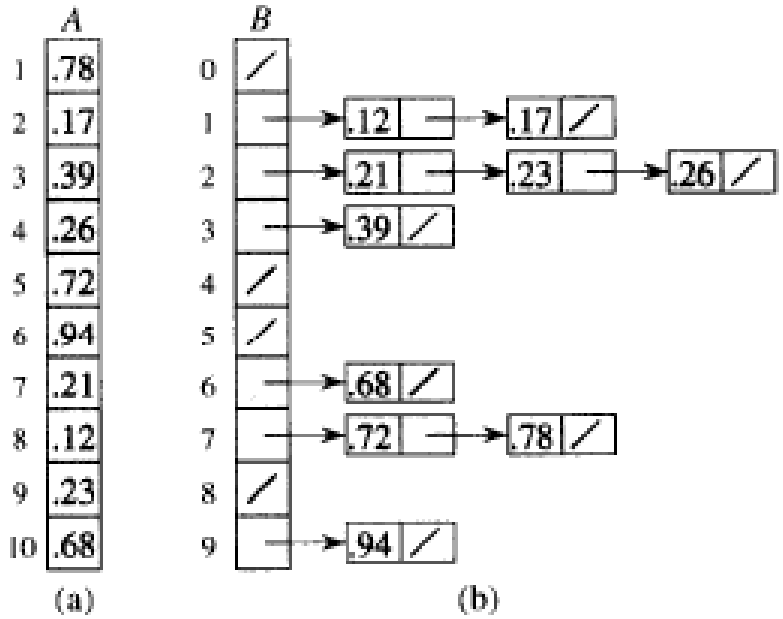
Sortare prin distribuire

- Algoritm:

```
1. procedure bucketSort(a, n)
2. begin
3.    for i ← 0 to n-1 do
4.    insereaza( B[⌊n · a[i]⌋ ], a[i])
5.    for i ← 0 to n-1 do
6.       sorteaza lista B[i]
7.    concatenează în ordine listele B[0],
   B[1], ..., B[n-1]
8. 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	-