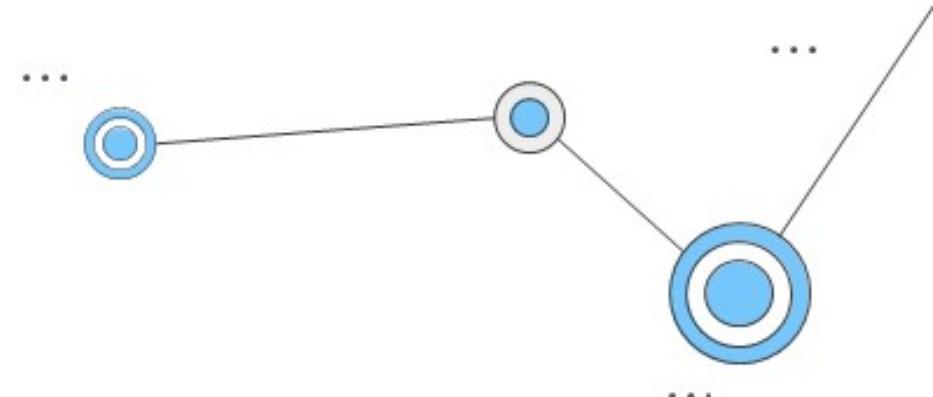


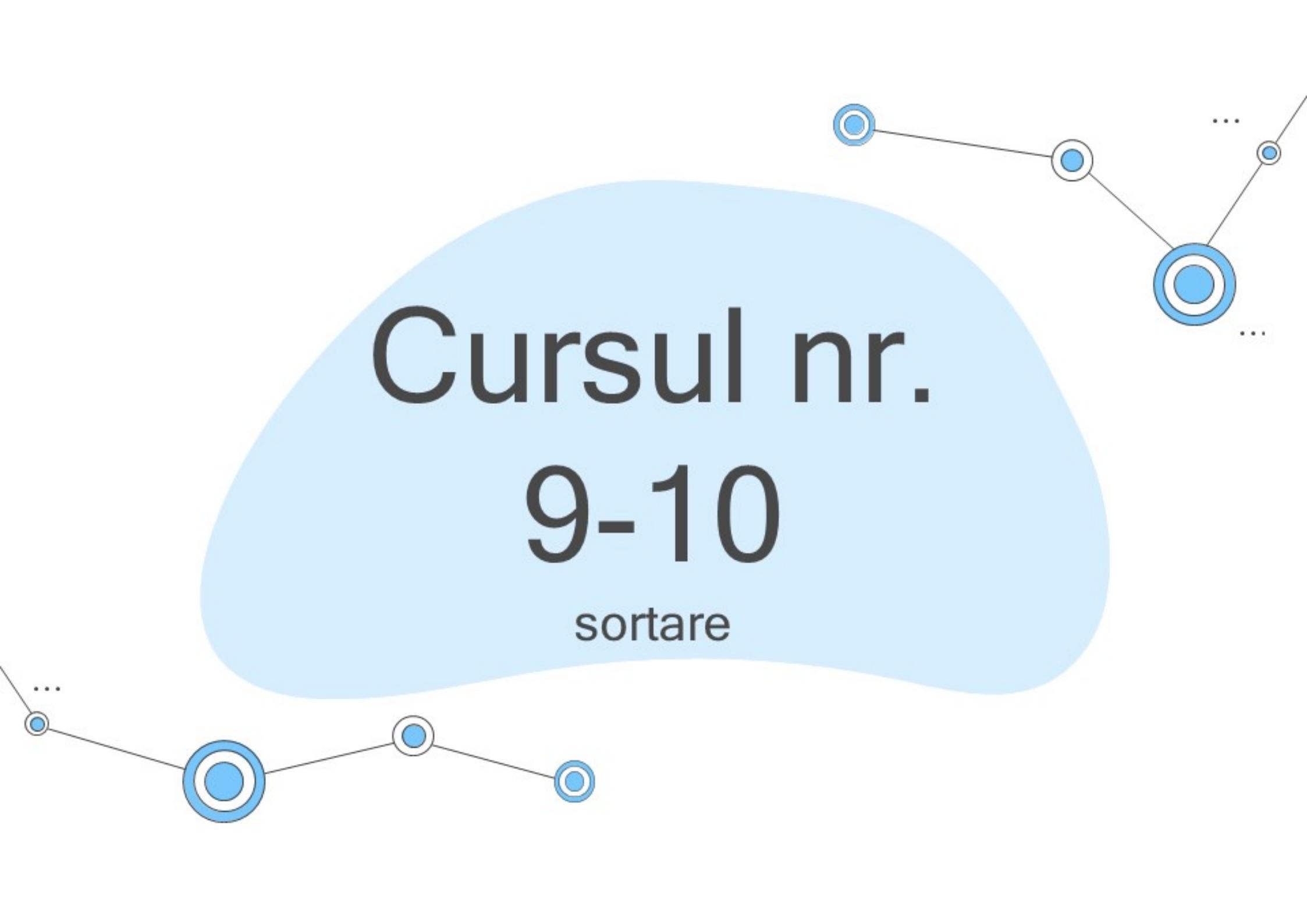
Algoritmi Fundamentali

Lector dr.
Dorin IORDACHE



Cursul nr. 9-10

sortare



Agenda



Sortarea prin interclasare



Sortarea prin pivotare



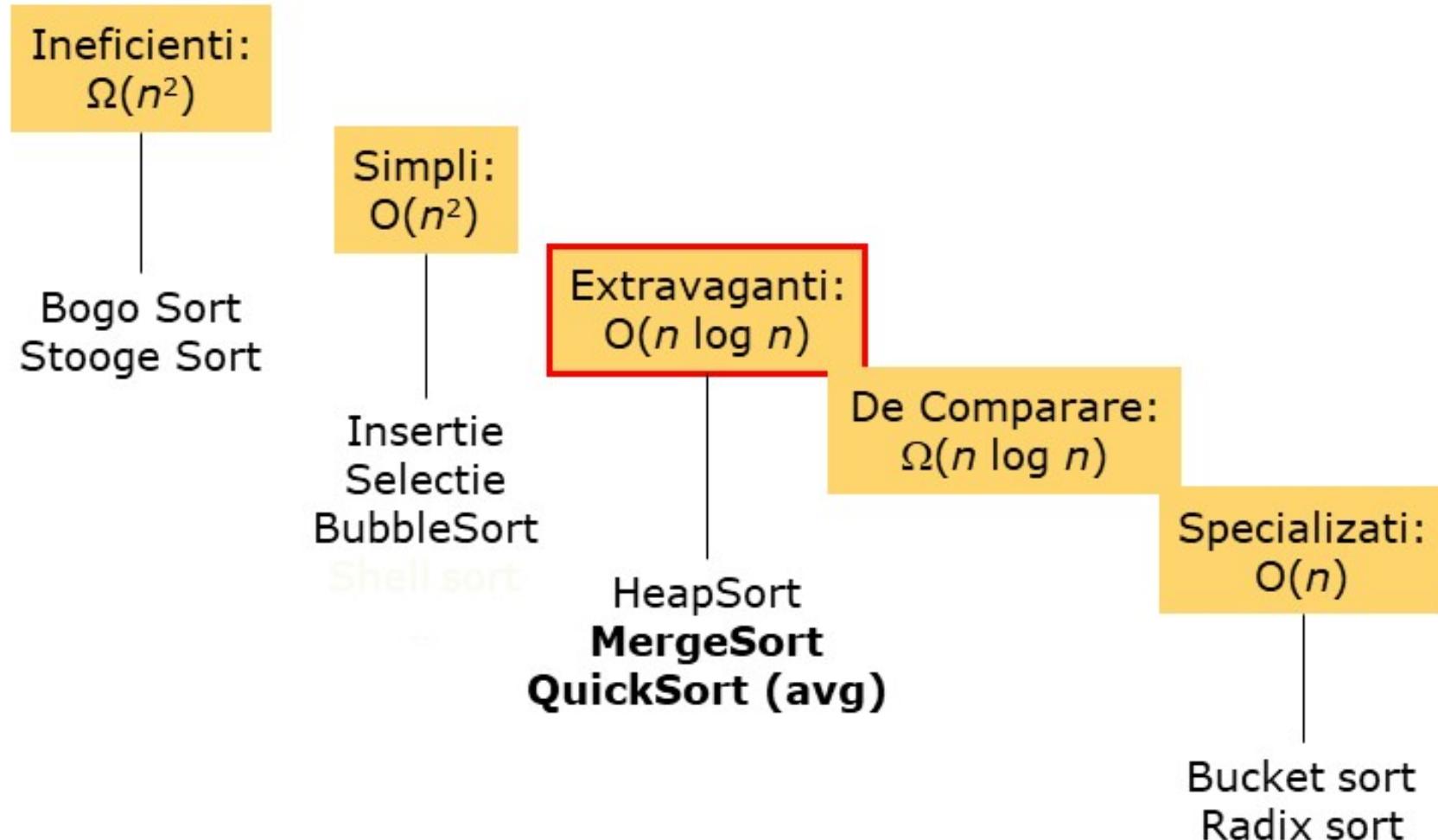
HeapSort



RadixSort



Sortare: algoritmi



Divide et Impera

O tehnică foarte importantă în proiectarea algoritmilor

1. Divide - împărțirea în probleme unitare(sub-probleme)
2. Rezolvarea independentă a sub-problemelor rezultate în etapa Divide
 - Utilizare Recursivitate
 - sau procesare paralelă
3. Combinarea soluțiilor sub-problemelor pentru a determina soluția problemei

Divide et Impera(D&I) - Sortare

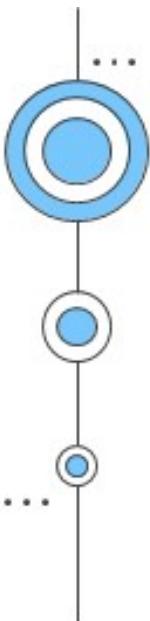
2 mari metode de ordonare D&I

Mergesort: Sortare recursiv jumătatea stângă
Sortare recursiv jumătatea dreaptă
Îmbinarea cele două jumătăți sortate

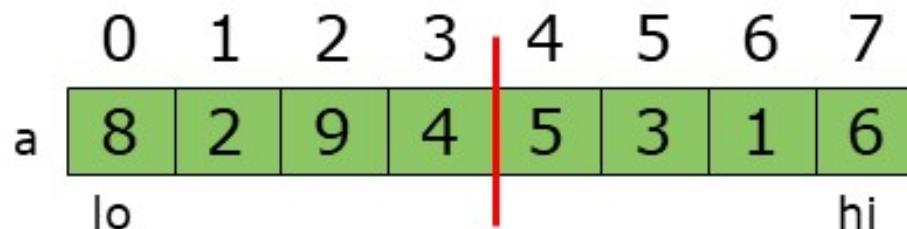
Quicksort: Alegerea unui element “pivot”
Separarea elementelor după (< și >)
Execuție recursivă asupra separărilor
Return < pivot, pivot, > pivot]

01

Sortarea prin interclasare **MergeSort**



Mergesort



Ordonarea elementelor vectorului de la pozitia **lo** la pozitia **hi**:

- Daca intervalul este unitar(1 element), atunci este sortat! (cel mai bun scenariu)
- Altfel, împărțirea elementelor în 2 submulțimi:
 - Sortare de la **lo** la **(hi+lo)/2**
 - Sortare de la **(hi+lo)/2** la **hi**
 - Interclasarea celor 2 jumătăți

Interclasarea a 2 părți sortate va avea ca rezultat o mulțime sortată

- $O(n)$ dar necesită spațiu auxiliar...

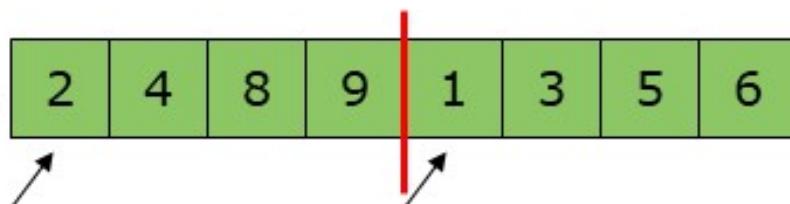
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

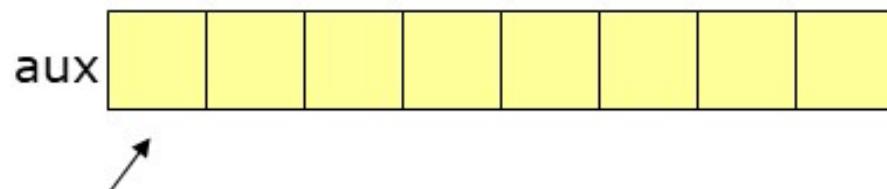
a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



Interclasare:

Utilizare vector
aditional

aux								
-----	--	--	--	--	--	--	--	--



După interclasare,
copiem conținutul în
vectorul original

Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
	↑			↑				

Interclasare:

Utilizare vector
aditional

aux	1							
	↑							

După interclasare,
copiem conținutul în
vectorul original

Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
	↑			↑				

Interclasare:

Utilizare vector
aditional

aux	1	2						
	↑							

După interclasare,
copiem conținutul în
vectorul original

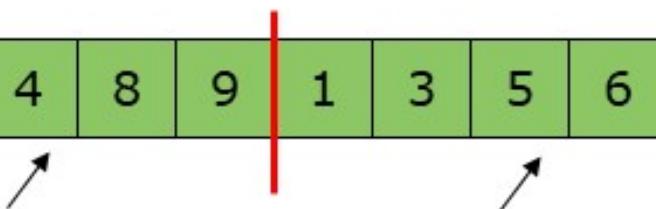
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



Interclasare:

Utilizare vector
aditional

aux	1	2	3					
-----	---	---	---	--	--	--	--	--



După interclasare,
copiem conținutul în
vectorul original

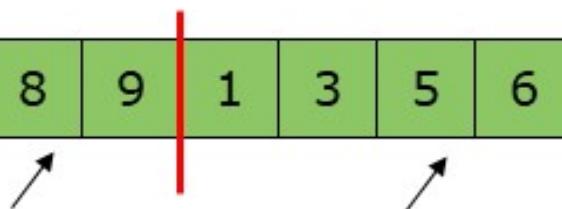
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

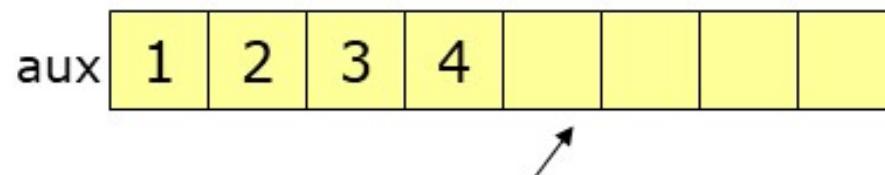
a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



Interclasare:

Utilizare vector
aditional

aux	1	2	3	4				
-----	---	---	---	---	--	--	--	--



După interclasare,
copiem conținutul în
vectorul original

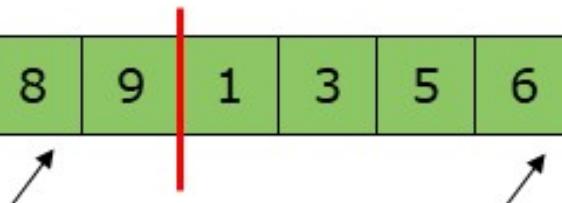
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



Interclasare:

Utilizare vector
aditional

aux	1	2	3	4	5			
-----	---	---	---	---	---	--	--	--



După interclasare,
copiem conținutul în
vectorul original

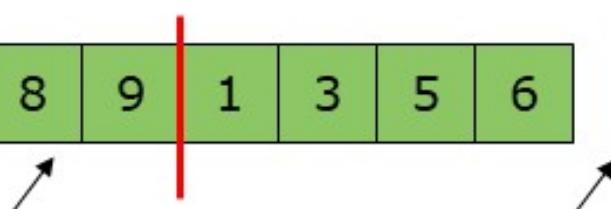
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

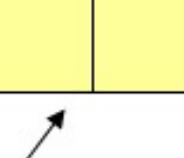
a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



Interclasare:

Utilizare vector
aditional

aux	1	2	3	4	5	6		
-----	---	---	---	---	---	---	--	--



După interclasare,
copiem conținutul în
vectorul original

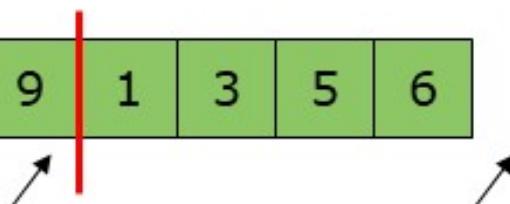
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



A red vertical line is positioned between the 4th and 5th elements (9 and 1). Two black arrows point upwards from below the array, one pointing to the 5th element (1) and another pointing to the 8th element (6), indicating the boundaries of the current partition.

Interclasare:

Utilizare vector
aditional

aux	1	2	3	4	5	6	8	
-----	---	---	---	---	---	---	---	--



A yellow arrow points to the last empty slot in the auxiliary array, indicating where the next element will be placed.

După interclasare,
copiem conținutul în
vectorul original

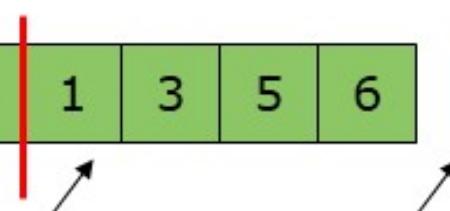
Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---



Interclasare:

Utilizare vector
aditional

aux	1	2	3	4	5	6	8	9
-----	---	---	---	---	---	---	---	---



După interclasare,
copiem conținutul în
vectorul original

Exemplu: Interclasare

Start cu:

a	8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---	---

După apel recursiv:

a	2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---	---

A diagram illustrating the state of the array after one recursive call. A red vertical line is drawn through the element at index 4 (value 9). Two black arrows point to the right from index 5 (value 1) and index 8 (value 6), indicating the range of elements to be moved.

Interclasare:

Utilizare vector
aditional

aux	1	2	3	4	5	6	8	9
-----	---	---	---	---	---	---	---	---

A diagram illustrating the state of the auxiliary array ('aux') after one recursive call. A red arrow points to the right from index 5 (value 1), and another red arrow points to the right from index 8 (value 6).

După interclasare,
copiem conținutul în
vectorul original

a	1	2	3	4	5	6	8	9
---	---	---	---	---	---	---	---	---

A diagram illustrating the final state of the array ('a'). The values have been copied back from the auxiliary array ('aux'). A red arrow points to the right from index 5 (value 1), and another red arrow points to the right from index 8 (value 6).

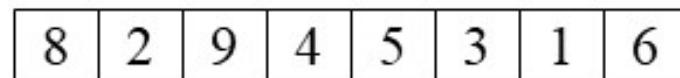
Interclasarea a 2 vectori unidimensionali

```
interclasare(a; b;c)
  k ← 0, i ← 0, j ← 0
  while i < n AND j < m do
    if a[i] ≤ b[j] then
      c[k++] ← a[i++]
    else
      c[k++] ← b[j++]
    endif
  endwhile
```

```
if i < n then
  for p ← i,n do
    c[k++] ← a[p]
  endfor
endif
if j < m then
  for p ← j,m do
    c[k++] ← b[p]
  endfor
endif
end interclasare
```

Exemplu: Mergesort

Divide



Divide



Divide



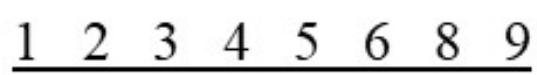
1 Element



Interclasare



Interclasare

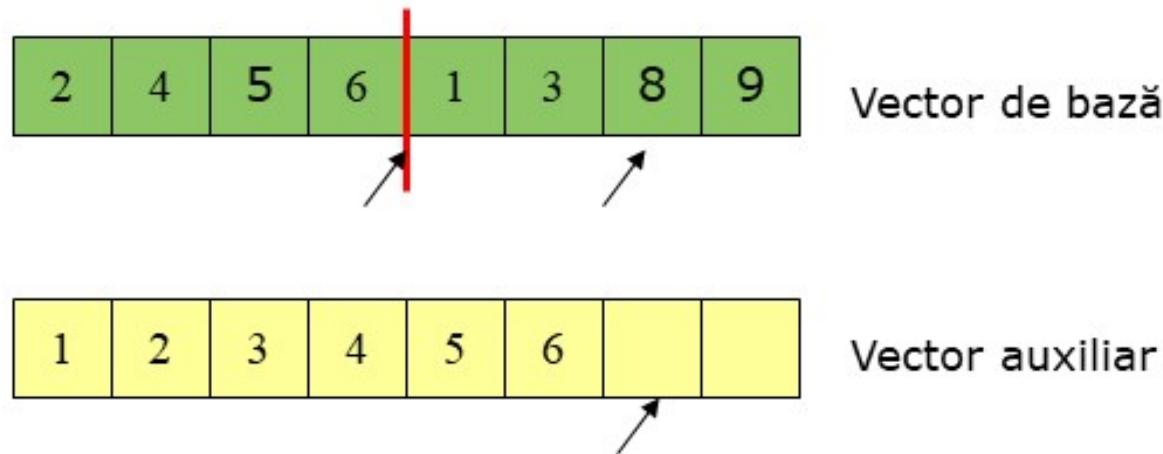


MergeSort

```
mergesort(v[]; w[]; st; dr)
    if stanga < dreapta then
        mijloc ← (stanga + dreapta)/2
        mergesort(v; w; stanga; mijloc)
        mergesort(v; w; mijloc + 1; dreapta)
        interclasare(v; w; stanga; mijloc;
                     dreapta)
    end if
end mergesort
```

Mergesort: Eficiență timp

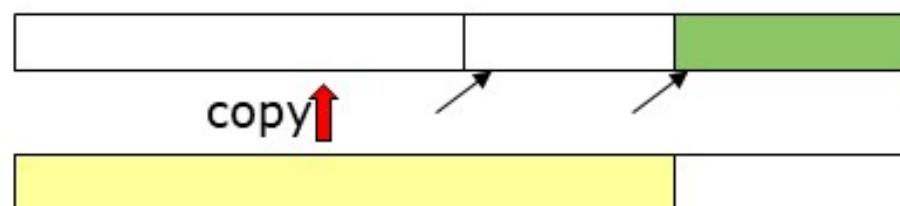
Ce se întâmplă dacă pașii finali ai interclasării ar arăta astfel?



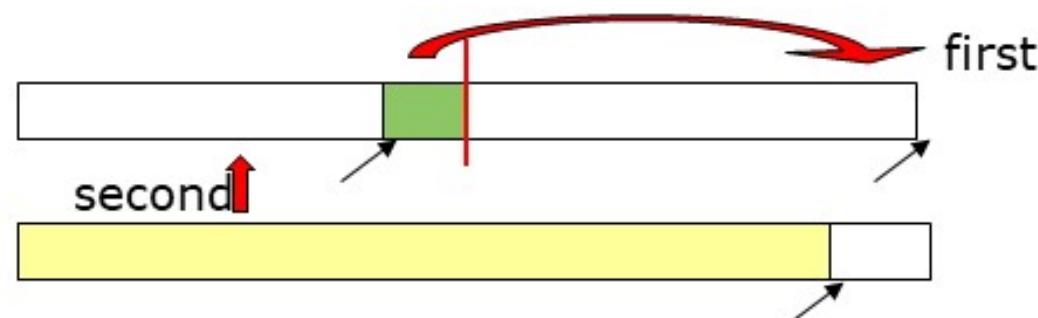
Nu este irositor să se copieze în vectorul auxiliar doar pentru a copia înapoi...

Mergesort: Eficiență timp

Dacă partea stângă se finalizează întâi, doar oprim îmbinarea și copiem înapoi:



Dacă partea dreaptă se finalizează prima, copiem partea rămasă în dreapta apoi copiem înapoi:



Mergesort: Eficiență spațiu

Simplu / Cea mai slabă implementare:

- Utilizăm un vector auxiliar de dimensiune (hi-lo) pentru fiecare interclasare

O implementare mai bună

- Utilizăm un vector auxiliar de dimensiune n pentru fiecare interclasare

O implementare și mai bună

- Reutilizăm același vector auxiliar de dimensiune n pentru fiecare interclasare

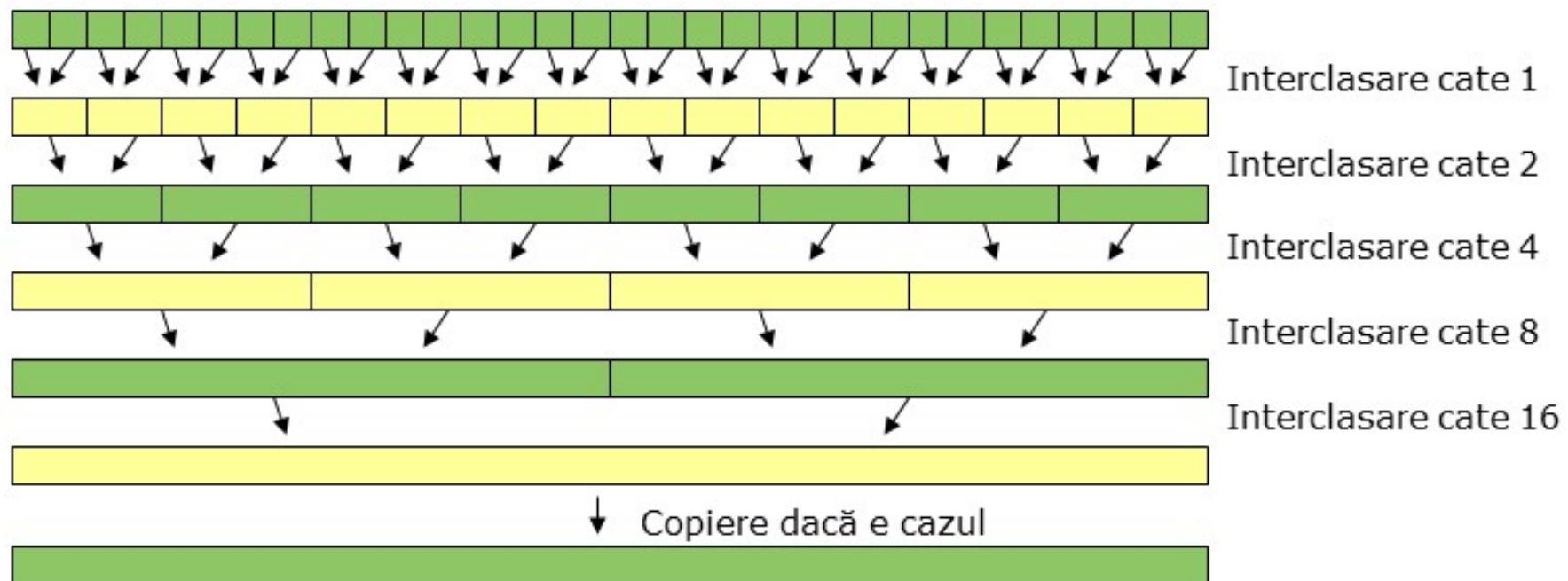
Cea mai bună implementare:

- Nu se execută copierea înapoi după interclasare
- Interschimbare folosire vector de bază și auxiliari
- Vom avea nevoie de o copie la final dacă numărul de etape este impar

Interschimbare Vector Original & Auxiliar

Recursiv pana la elemente unitar

După recursivitate, interschimbare între vectori



Probabil mai ușor de programat fără a utiliza deloc recursivitatea

Analiza complexității Mergesort

Poate fi stabil și in-place (complexitate!)

Performanță:

Pentru ordonarea a n elemente,

- Return imediat dacă $n=1$
- Altfel

do 2 subprobleme cu dimensiunea $n/2$ și
apoi interclasare $O(n)$

- Relația de recurență:

$$T(1) = c_1$$

$$T(n) = 2T(n/2) + c_2n$$

Analiza MergeSort

Pentru simplitate, fie constantele 1, fără efect asupra răspunsului asimptotic

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n$$

$$= 2(2T(n/4) + n/2) + n$$

$$= 4T(n/4) + 2n$$

$$= 4(2T(n/8) + n/4) + 2n$$

$$= 8T(n/8) + 3n$$

... (după k iterări)

$$= 2^k T(n/2^k) + kn$$

total este $2^k T(n/2^k) + kn$
unde $n/2^k = 1$, i.e., $\log n = k$

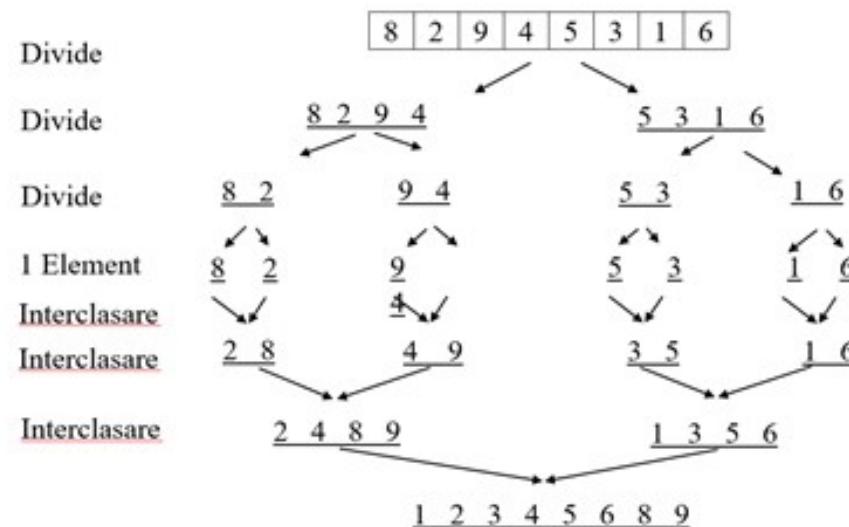
$$\begin{aligned} \text{Atunci, } & 2^{\log n} T(1) + n \log n \\ & = n + n \log n \\ & = O(n \log n) \end{aligned}$$

Analiza Mergesort

Acest algoritm este cunoscut sub notația $O(n \log n)$

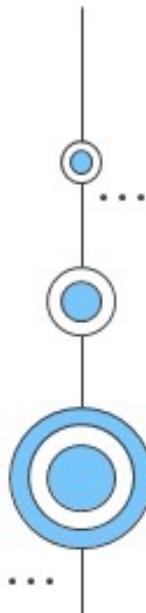
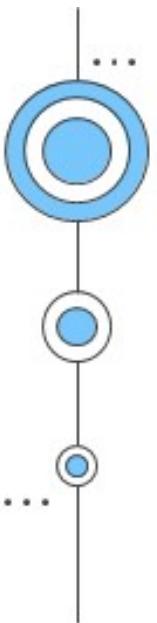
Sortarea prin interclasare (mergesort) este relativ ușor de intuit (cazul cel mai favorabil, cel mai defavorabil și mediu):

- Stiva de recursivitate va avea înălțimea $\log n$
- La fiecare nivel numărul total de interclasări este n



02

Sortarea prin pivotare **QuickSort**



Quicksort – Sortarea prin pivotare

Se bazează pe tehnica Divide et Impera

- Recursiv se împarte mulțimea elementelor în jumătăți
- În loc să realizăm execuția de interclasare, se execută activitatea recursiv împărțind în jumătăți
- Față de MergeSort, nu este nevoie de spațiu auxiliar

$O(n \log n)$ în cazul mediu, dar $O(n^2)$ în cazul cel mai nefavorabil

- MergeSort este întotdeauna $O(n \log n)$
- De ce să utilizăm QuickSort atunci?

Poate fi mai rapidă decât Mergesort

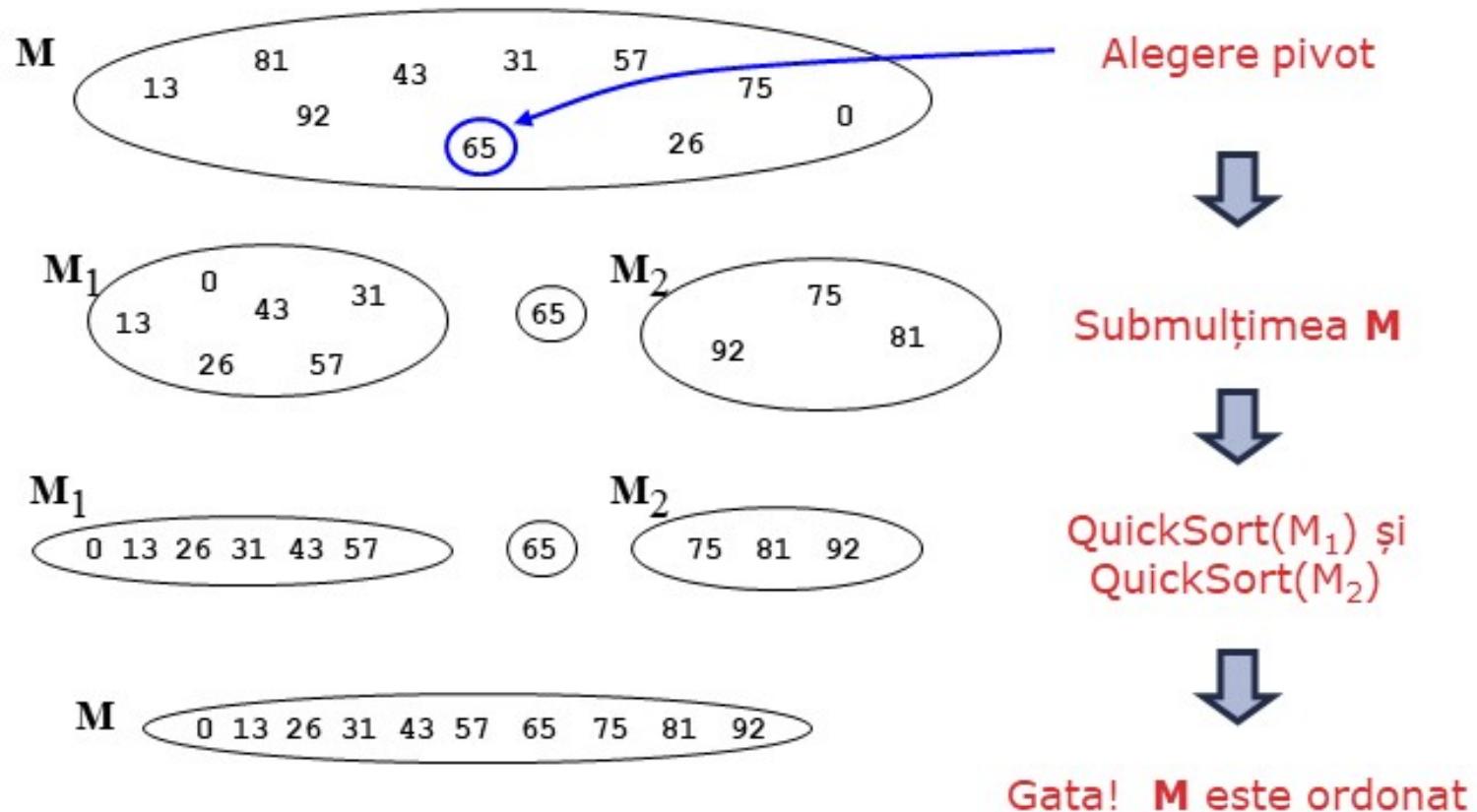
- Crezut de mulți că este mai rapid !!!
- Quicksort execută mai puține copii și mai multe comparații, aşa că depinde de costul relativ al acestor două operațiuni!

Quicksort

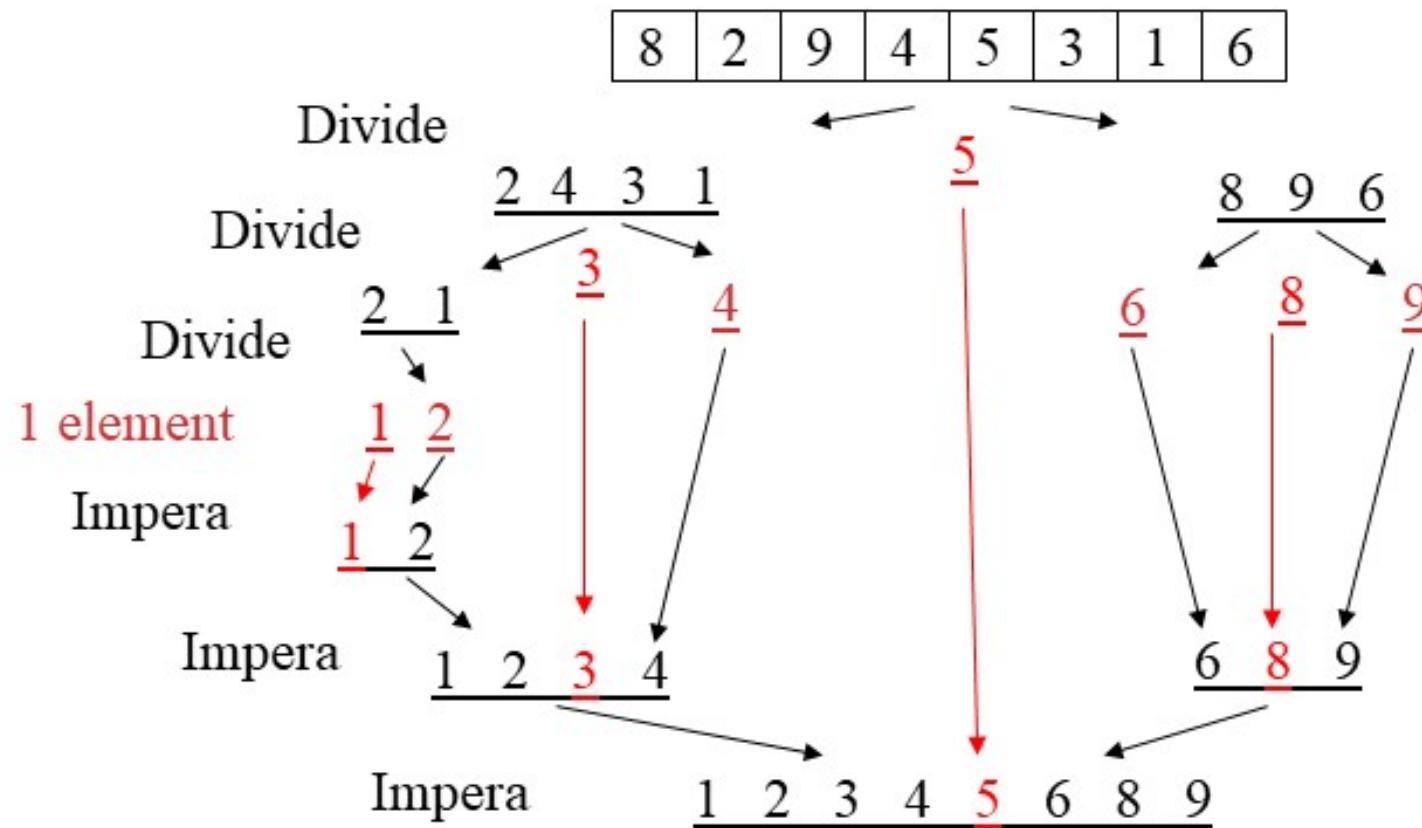
1. Alegerea unui element **pivot**
2. Partiționarea datelor în:
 - A. Elementele mai mici decât pivotul
 - B. Pivotul
 - C. Elementele mai mari decât pivotul
3. Recursiv ordonare A și C
4. Rezultatul obținut este "A, B, C"

Pare simplu dar detaliile sunt "șmechere"!

Quicksort: vom gândi totul în termeni de mulțimi



Exemplu: Quicksort Recursiv



Quicksort

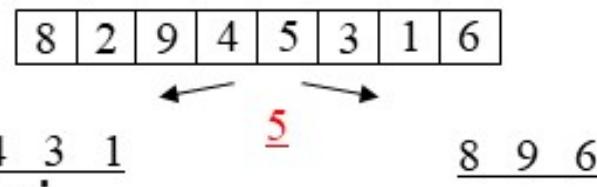
Nu am discutat :

- Cum alegem elementul pivot
 - Orice variantă este corectă: la final elementele vor fi ordonate
 - Ne dorim ca cele două partiții să fie aproximativ egale ca dimensiune (nu întotdeauna reușim)
- Cum implementăm partiționarea
 - În timp liniar
 - In-place

Pivot

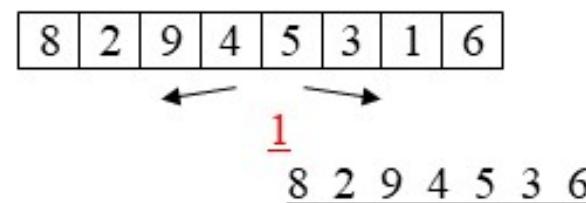
- Cel mai bun pivot?

- Median
- Elementul din mijlocul intervalului



- Cel mai defavorabil?

- Cel mai mare/mic element
- Dimensiunea problemei $n - 1$
- $O(n^2)$



Quicksort: reguli pentru alegerea pivotului

Atunci când avem un vector în intervalul $\text{arr}[\text{lo}]$ la $\text{arr}[\text{hi}-1]$

Alegem $\text{arr}[\text{lo}]$ sau $\text{arr}[\text{hi}-1]$

- Eficient, dar poate fi cazul cel mai nefavorabil când elementele sunt aproape sortate

Alegerea aleatoare

- Se execută similar cu orice altă variantă
- Generarea aleatoare poate încetini procesul
- Probabil o abordare elegantă

Determinarea elementului din mijlocul intervalului

- $O(n)$ timp!

Median of 3, (e.g., $\text{arr}[\text{lo}]$, $\text{arr}[\text{hi}-1]$, $\text{arr}[(\text{hi}+\text{lo})/2]$)

- O abordare heuristică poate funcționa la fel de bine

Partiționarea

Teoretic ușor, dar destul de complex de programat

- Este necesară partiționarea într-un timp liniar *in-place*

O sugestie de abordare:

Interschimbare pivot cu elementul arr[stanga]

Utilizare 2 indecsări i și j, de la stanga+1 la dreapta-1

```
while (i < j) do
    if (arr[j] >= pivot) then
        j--
    else
        if (arr[i] <= pivot) then
            i++
        else
            swap arr[i], arr[j]
        endif
    endif
    swap pivot, arr[i]
```

QuickSort

```
quicksort(v[]; stanga; dreapta)
    if stanga < dreapta then
        poz ← pivotare(v; stanga; dreapta)
        quicksort(v; stanga; poz - 1)
        quicksort(v; poz + 1; dreapta)
    end if
end quicksort
```

Quicksort Exemplu

Pas 1:

Alegerea elementului din mijloc ca *pivot*
stanga = 0, dreapta = 9

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

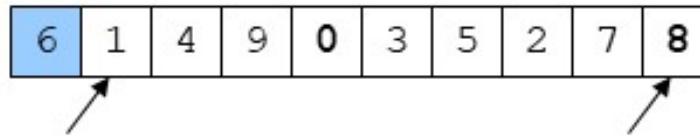
Pas 2: Mutare *pivot* in pozitia stanga

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

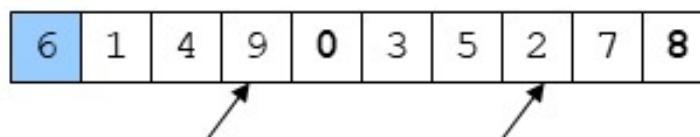


Quicksort Exemplu

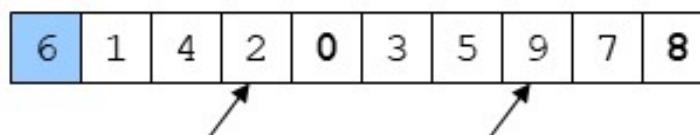
Pivotare elemente



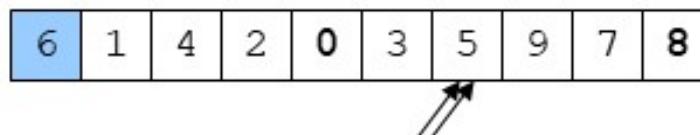
Mutare stanga dreapta



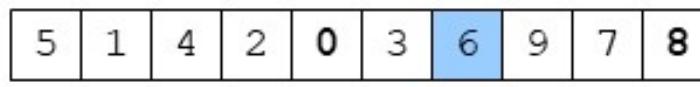
Interschimbare



Mutare stanga dreapta



Mutare pivot



Se execută mai multe iterații până când
elementele sunt ordonate

Pivotare

```
pivotare(v[]; stanga; dreapta)
pivot ← v[stanga]
s ← stanga + 1
d ← dreapta
while s <= d do
    while v[d] > pivot do
        d ← d - 1
    end while
```

```
while v[s] < pivot AND s <= d do
    s ← s + 1
end while
if s <= d then
    v[s] ↔ v[d]
    d ← d - 1
    s ← s + 1
end if
end while
v[stanga] ↔ v[d]
return d
end pivotare
```

Analiza Quicksort

Cazul favorabil: Pivotul este întotdeauna elementul din mijloc

$$T(0)=T(1)=1$$

$$T(n)=2T(n/2) + n \quad \text{partiționare liniară}$$

Aceeași complexitate ca Mergesort: $O(n \log n)$

Cazul nefavorabil: Pivotul este intodeauna cel mai mic sau cel mai mare

$$T(0)=T(1)=1$$

$$T(n) = 1T(n-1) + n$$

Aceeași complexitate ca sortarea prin Selecție: $O(n^2)$

Cazul mediu (pivot aleas aleator):

$$O(n \log n)$$

Liste și Big Data

Mergesort poate funcționa foarte bine direct pe listele înlățuit

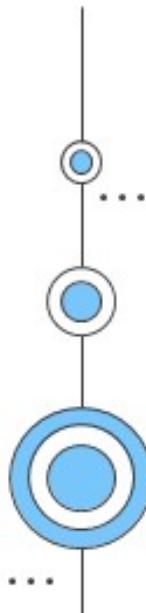
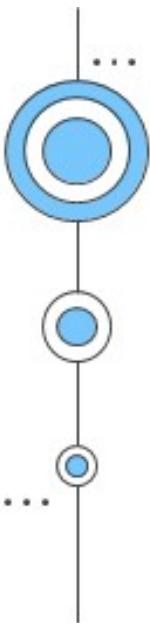
- Heapsort și Quicksort nu
- Sortările prin Inserție și Seleție pot de asemenea, dar mai lent

Mergesort, de asemenea, pentru sortarea externă

- Quicksort și Heapsort acolo unde sunt vectori
- Mergesort scanează liniar vectorii
- Sortarea în-memorie a blocurilor poate fi combinată cu sortări mai mari
- Mergesort poate folosi mai multe discuri de memorie

03

Sortarea heap HeapSort



Heapsort

Sortarea heap = o tehnică de sortare prin comparație bazată pe structura de date Binary Heap.

- similară cu sortarea prin selecție în care găsim mai întâi elementul minim și plasăm elementul minim la început.

Algoritm Heapsort

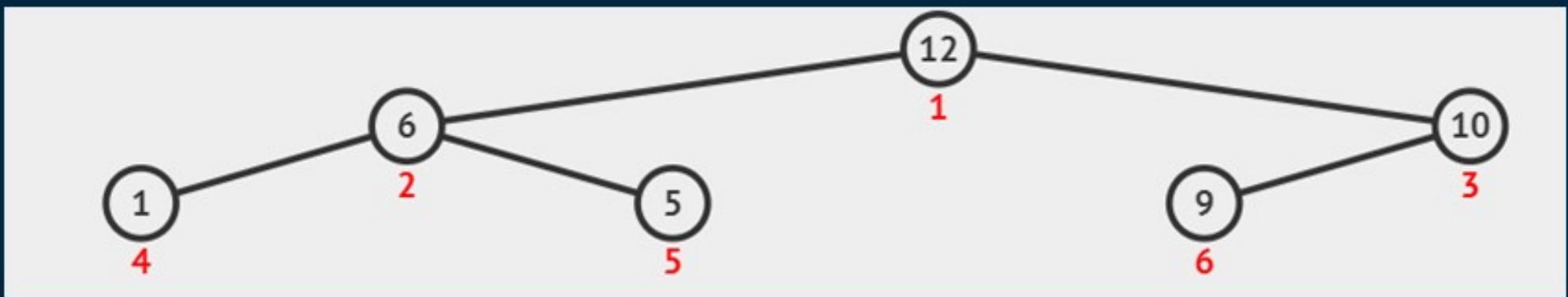
Considerăm arborele Max-Heap (cel mai mare element este root).

1. **Interschimbare:** Interschimbare ultimul element din arbore cu nodul root. Stocare la sfârșitul vectorului (poziția a n-a).
2. **Eliminați:** decrementăm dimensiunea stivei cu 1.
3. **Heapify:** Heapify elementul rădăcină din nou, astfel încât să avem cel mai înalt element la rădăcină.
4. Repetăm pașii 1-3 până toate elementele din listă sunt sortate.

Heapsort

1	12	9	5	6	10
0	1	2	3	4	5

Max Heap

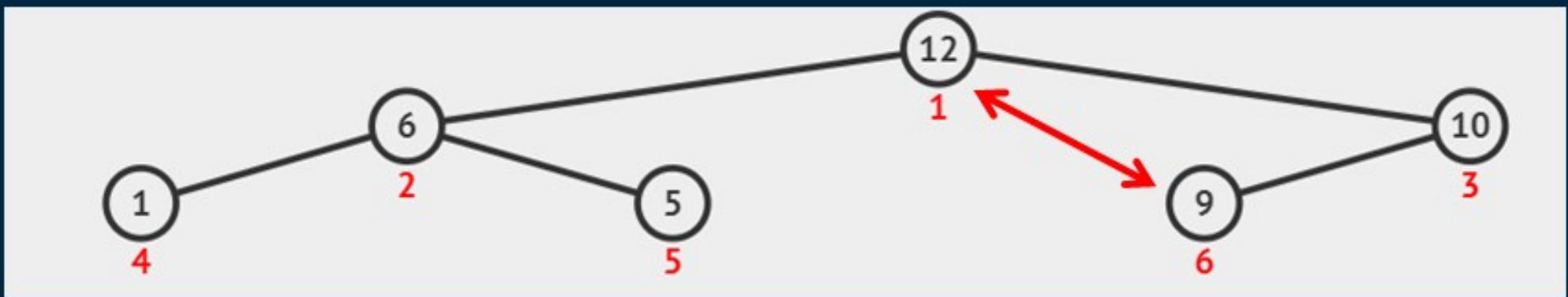


12	6	10	1	5	9
0	1	2	3	4	5

Heapsort

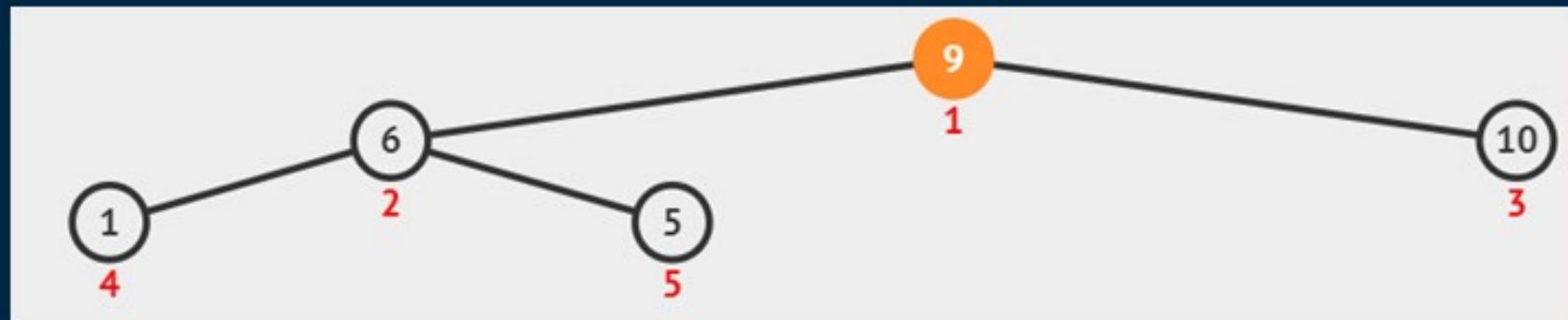
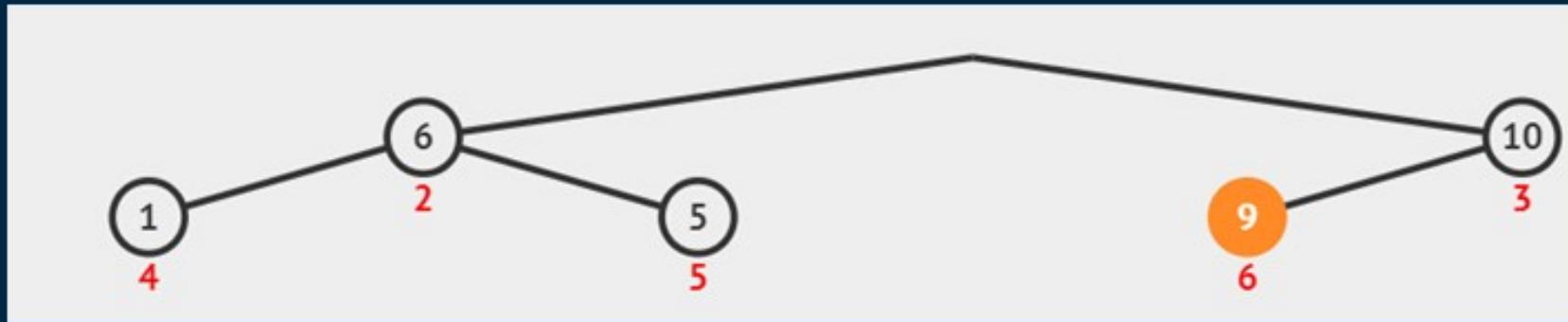
12	6	10	1	5	9
0	1	2	3	4	5

Max Heap



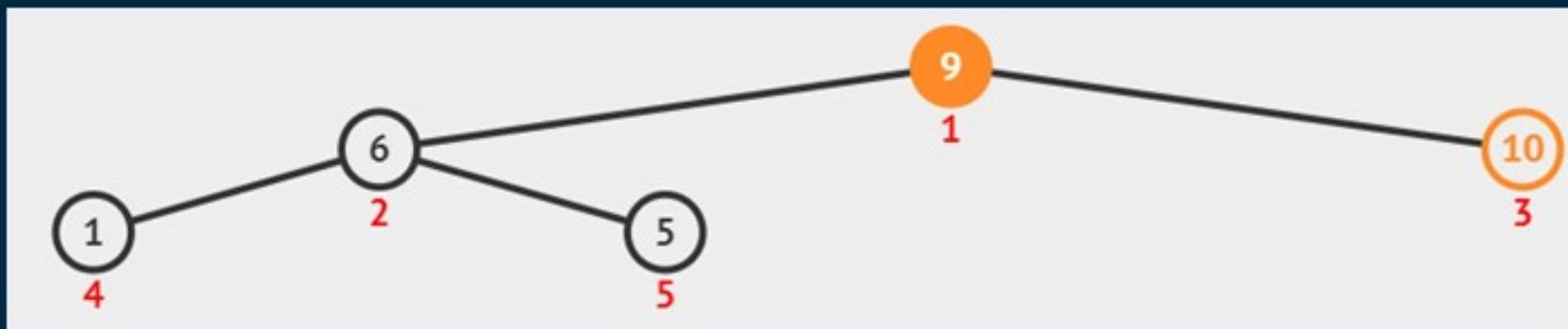
9	6	10	1	5	12
0	1	2	3	4	5

Heapsort

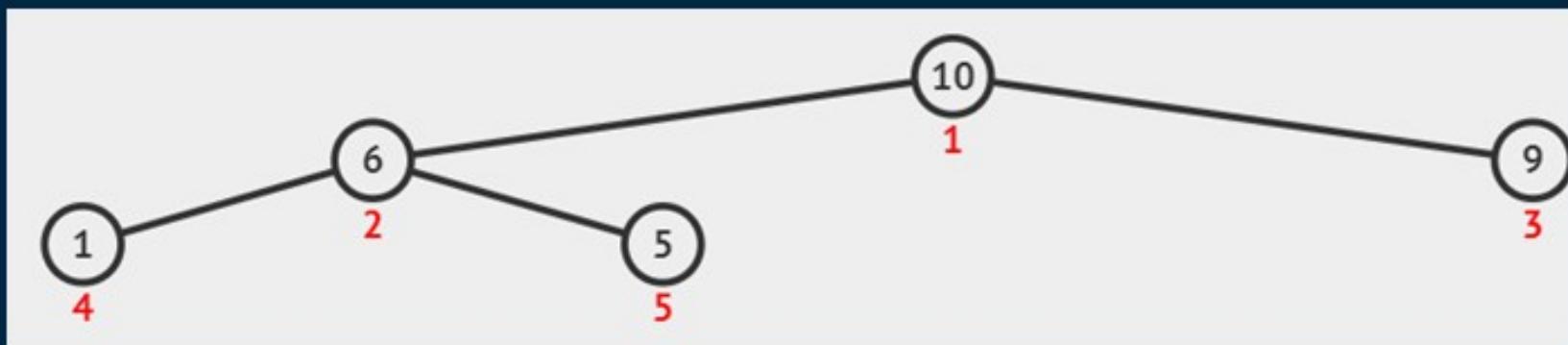


9	6	10	1	5	12
0	1	2	3	4	5

Heapsort



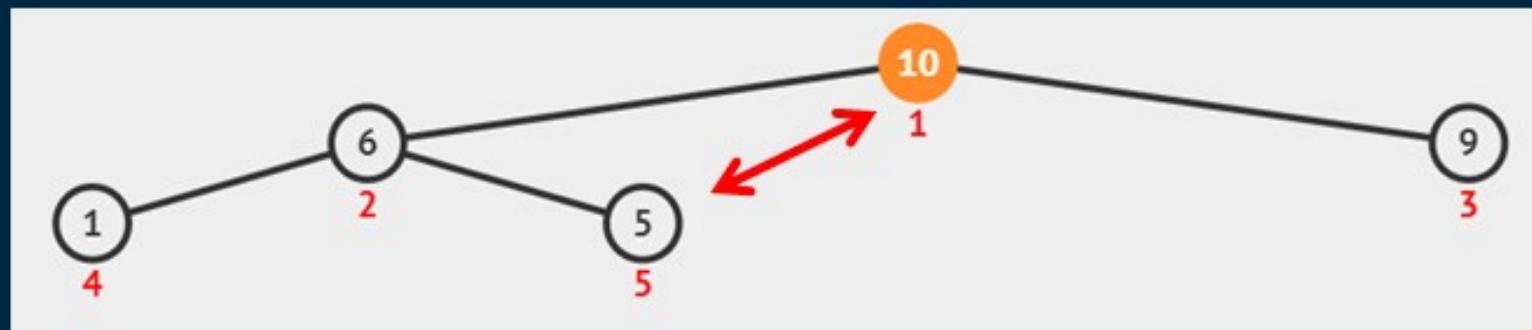
Heapify



9	6	10	1	5	12
0	1	2	3	4	5

Heapsort

9	6	10	1	5	12
0	1	2	3	4	5



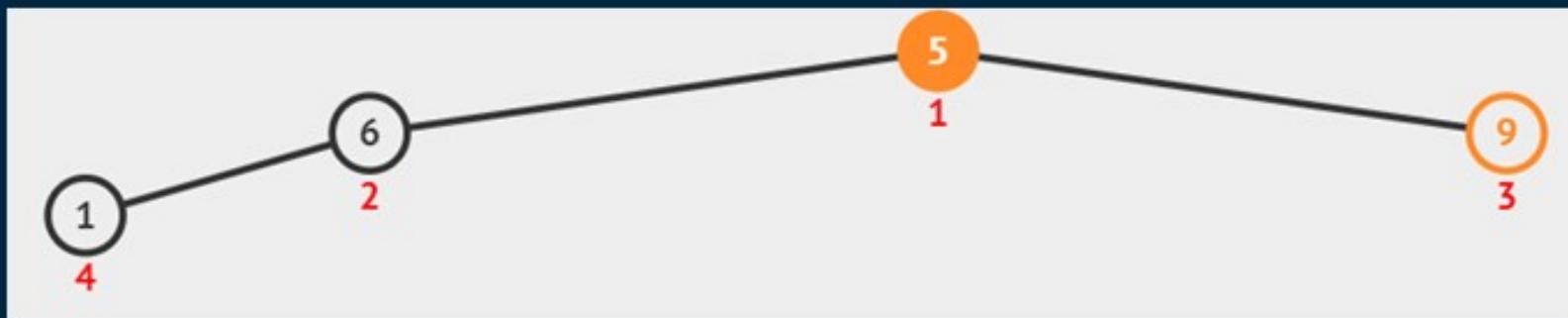
9	6	10	1	5	12
0	1	2	3	4	5

Heapsort

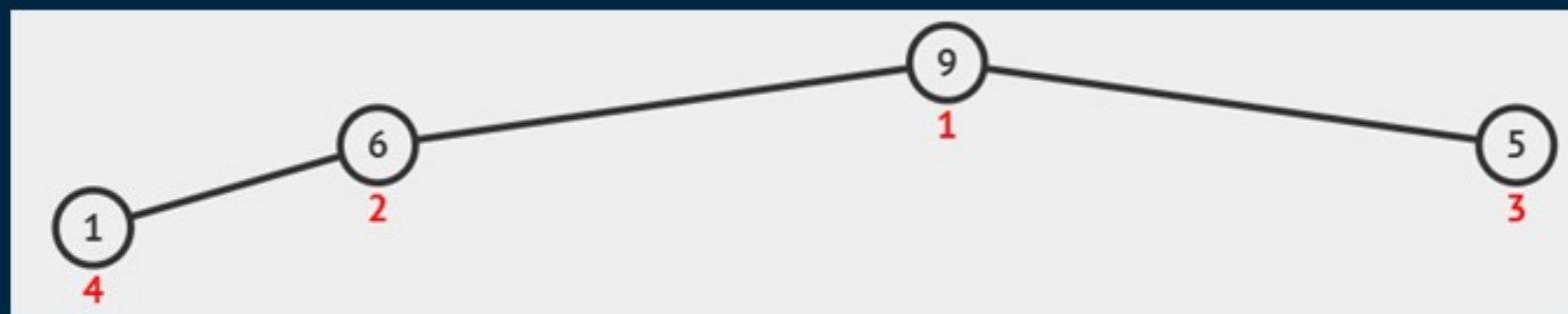


9	6	5	1	10	12
0	1	2	3	4	5

Heapsort



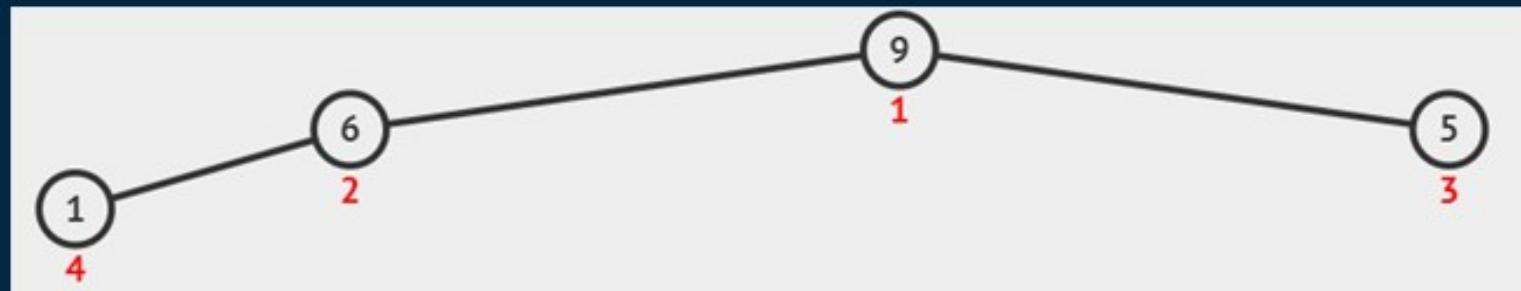
Heapify



9	6	5	1	10	12
0	1	2	3	4	5

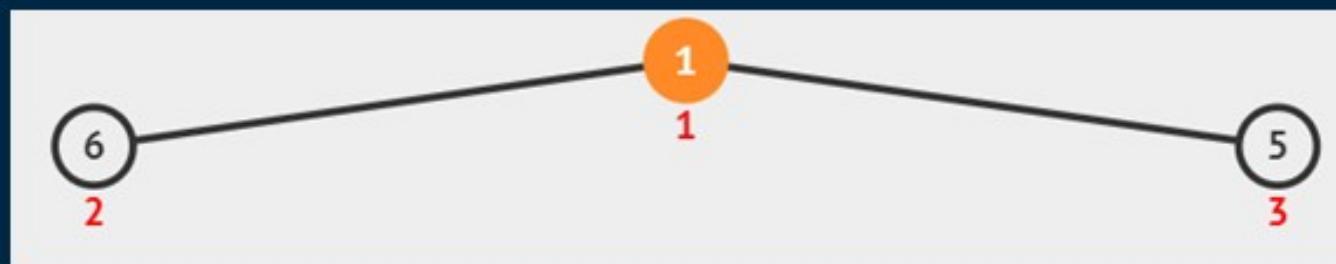
Heapsort

9	6	5	1	10	12
0	1	2	3	4	5

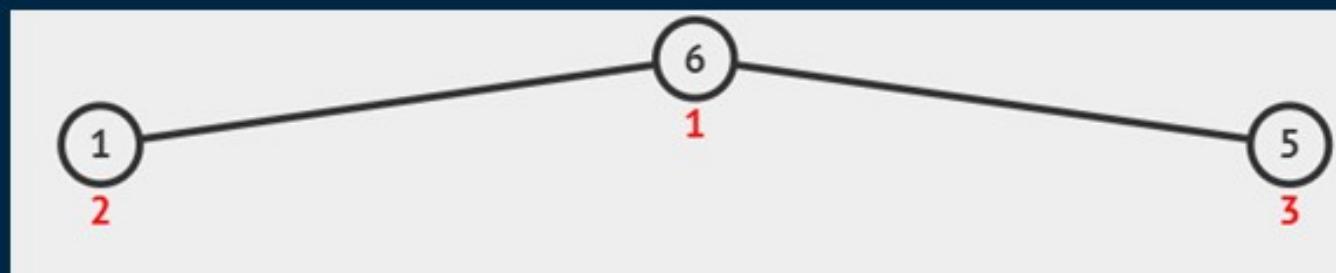


1	6	5	9	10	12
0	1	2	3	4	5

Heapsort



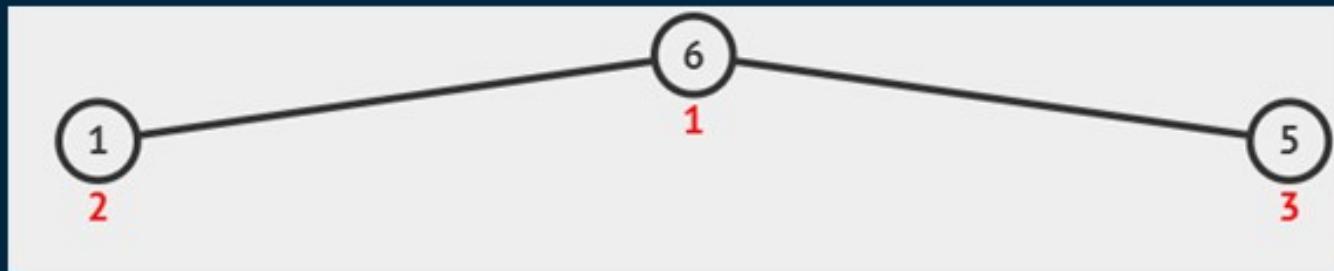
Heapify



1	6	5	9	10	12
0	1	2	3	4	5

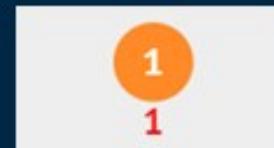
Heapsort

1	6	5	9	10	12
0	1	2	3	4	5



1	5	6	9	10	12
0	1	2	3	4	5

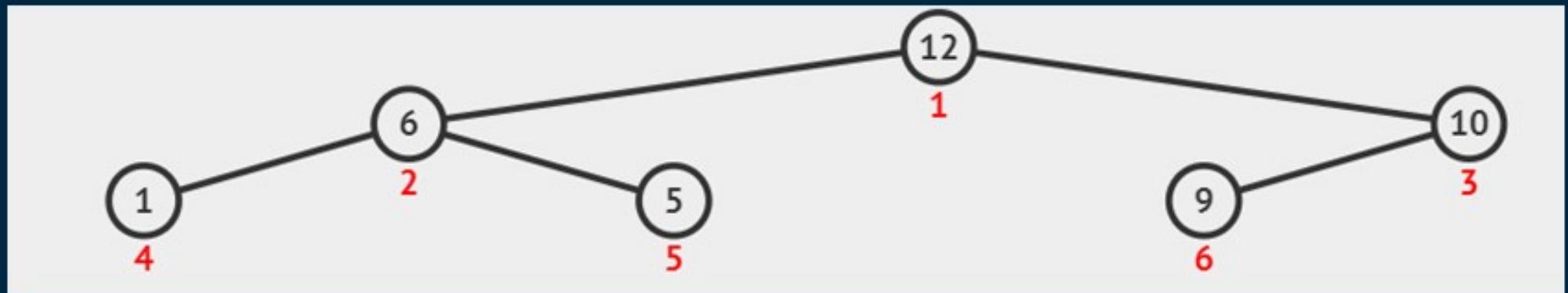
Heapsort



1	5	6	9	10	12
0	1	2	3	4	5

Heapsort

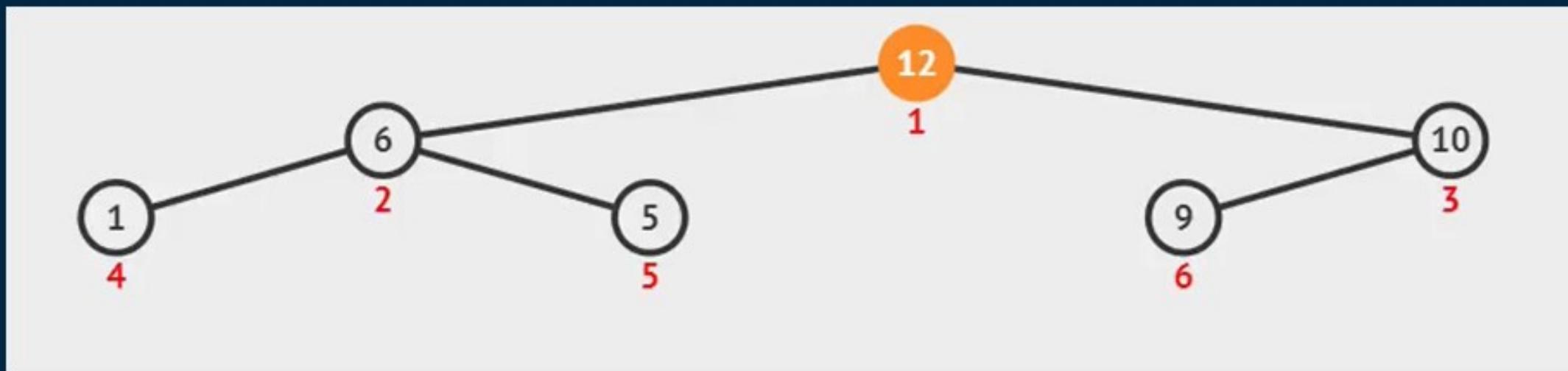
1	12	9	5	6	10
0	1	2	3	4	5



Vector sortat crescător

1	5	6	9	10	12
0	1	2	3	4	5

Heapsort



Vector sortat crescător

1	5	6	9	10	12
0	1	2	3	4	5

Complexitate - Heapsort

Complexitate timp execuție

Cazul cel mai favorabil $O(n \log n)$

Cazul cel mai nefavorabil $O(n \log n)$

Medie $O(n \log n)$

Heapsort

```
Heapsort(A) {
    BuildHeap(A)
    for i <- length(A) downto 2 {
        exchange A[1] <-> A[i]
        heapsize <- heapsize -1
        Heapify(A, 1)
    }
```

```
BuildHeap(A) {
    heapsize <- length(A)
    for i <- floor( length/2 ) downto 1
        Heapify(A, i)
}
```

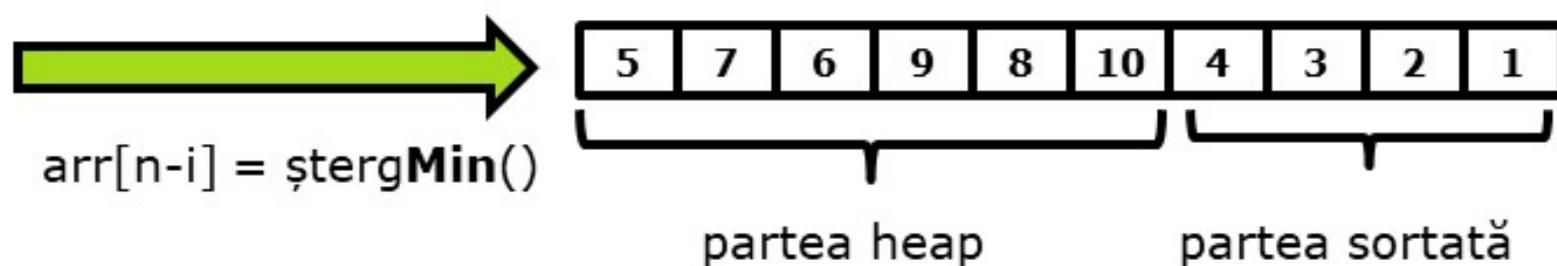
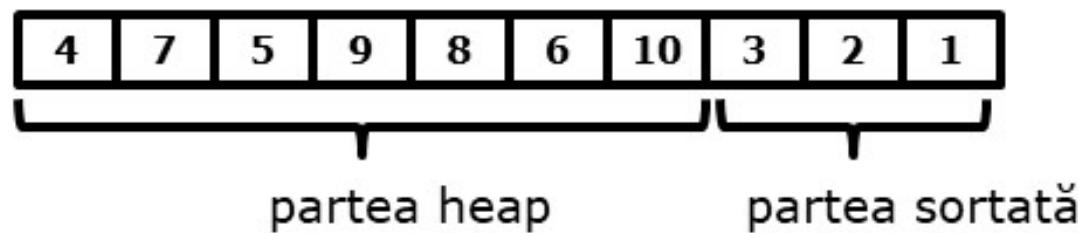
```
Heapify(A, i) {
    le <- left(i)
    ri <- right(i)
    if (le<=heapsize) and (A[le]>A[i])
        largest <- le
    else
        largest <- i
    if (ri<=heapsize) and (A[ri]>A[largest])
        largest <- ri
    if (largest != i) {
        exchange A[i] <-> A[largest]
        Heapify(A, largest)
    }
}
```

<https://fullyunderstood.com/pseudocodes/heap-sort/>

In-Place Heap Sort

Tratați vectorul inițial ca un heap (prin buildHeap)

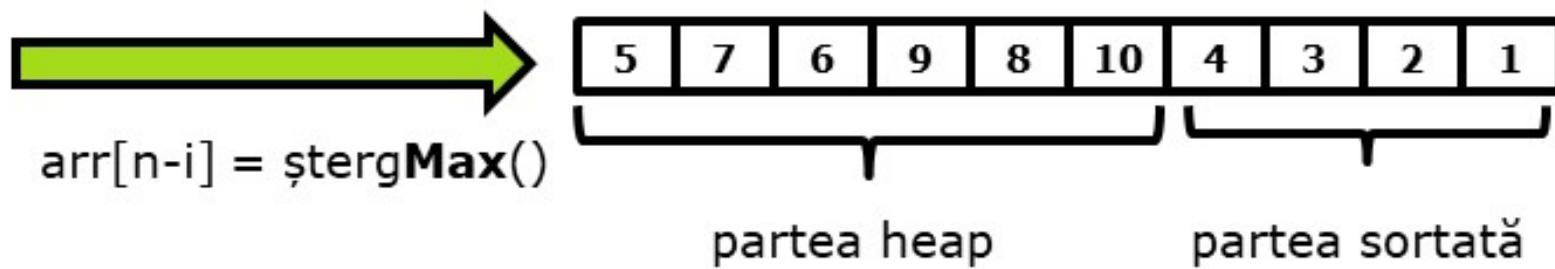
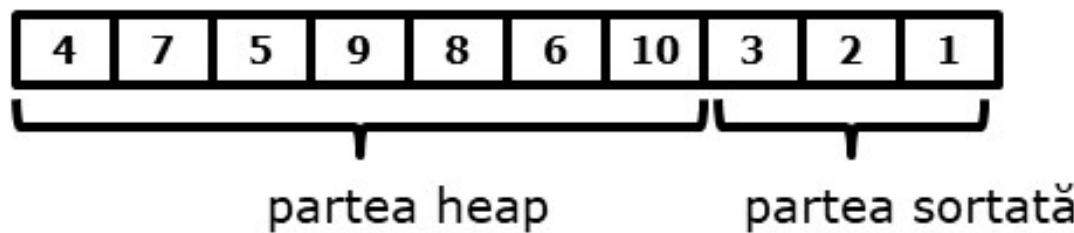
Când se șterge al i-lea element, se plasează la $\text{arr}[n-i]$, deoarece locația matricei nu mai face parte din heap!



In-Place Heap Sort

Dar dacă schimbăm criteriul de ordonare invers?

Construim un maxHeap!!!



"Dictionary Sort"

Se poate utiliza un arbore echilibrat pentru:

- `insert` fiecărui element: timp total $O(n \log n)$
- Repetat `ștergMin`: timp total $O(n \log n)$

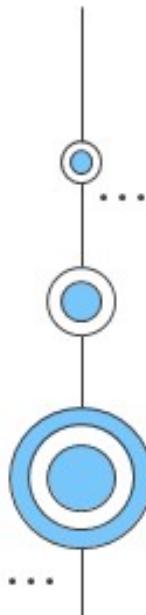
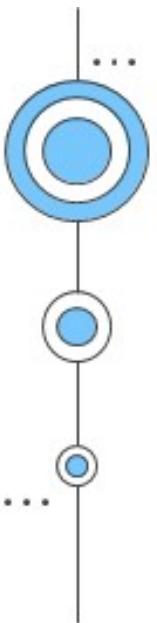
Dar acest lucru nu poate fi realizat in-place și are factori constanți mai rău decât Heap Sort

- Amândouă metode $O(n \log n)$ în toate cazurile
- Nu se pot paraleliza acțiunile
- Heap sort este mai bună

NU va gândiți vreodată sa utilizați hash table pentru sortare

04

RadixSort



BinSort (BucketSort)

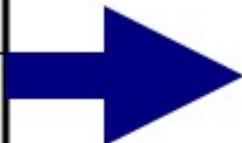
- Dacă toate elementele sunt de la $1 \dots K$
- Utilizează un vector de dimensiune K
- Plasarea elementelor în forma ordonată în vector

exemplu BinSort

- K=5. lista=(5,1,3,4,3,2,1,1,5,4,5)



în Vector	
cheia = 1	1,1,1
cheia = 2	2
cheia = 3	3,3
cheia = 4	4,4
cheia = 5	5,5,5



Lista sortată:
1,1,1,2,3,3,4,4,5,5,5

pseudocod BinSort

```
procedure BinSort (List L,K)

LinkedList vector[1..K]
/*
Fiecare element al matricei vector este din listă.
Se poate utiliza și un vector bidimensional
*/

for x <- 1,L do
    vector[x] <- x
endfor
for i <- 1,K do
    for each number x in vector[i]
        print x
    endfor
endfor
```

Concluzii BinSort :

- K este o constantă
 - BinSort se execută în timp liniar
- K este variabilă
 - Nu este un timp liniar simplu
- K este mare(de exemplu 2^{32})
 - Ineficient, neutilizabil

RadixSort

- Radix = “Baza unui sistem de numerație”
 - utilizat în 1890 U.S. Hollerith
- Ideea: BinSort pentru fiecare cifră, bottom-up

https://www.census.gov/history/www/innovations/technology/the_hollerith_tabulator.html.

RadixSort

- Date de intrare:
126, 328, 636, 341, 416, 131, 328
- BinSort cifre cel mai puțin semnificative:
341, 131, 126, 636, 416, 328, 328
- BinSort după următoarele cifre:
416, 126, 328, 328, 131, 636, 341
- BinSort după următoarele cifre:
126, 131, 328, 328, 341, 416, 636

Magic, nu ?!

- Cheile
 - Numere cu N-cifre
 - Baza de numerație B
- Cerință: după i^a BinSort, cele mai puțin semnificative i cifre sunt sortate.
 - i.e. $B=10$, $i=3$, cheile sunt 2667 și 7345. 7345 este ulterior lui 2667 după ultimele 3 cifre.

Analiza Radixsort

- Concluzii
 - Nu este un timp liniar pentru K mare.
- În practică
 - RadixSort - numere largi de date, dar care au puține chei
 - Ineficient față de MergeSort/QuickSort

Ce tipuri de date T putem ordonat cu RadixSort?

- Orice tip de dată T
- Orice tip de dată T care poate fi “spartă” în părți A și B, dacă:
 - Se poate reconstrui T din A și B
 - A poate fi ordonat cu RadixSorted
 - B poate fi ordonat RadixSorted
 - A este întotdeauna partea cea mai semnificativă a T

Exemplu:

- Numere cu 1 cifră BinSort
- Numere cu 2 la 5 cifre BinSort fără a utiliza memorie în exces
- Numere cu 6 cifre, “*sparte*” în numere a câte 3 cifre, A și B.
 - A și B pot reconstrui numărul original
 - A și B ordonate cu RadixSortable
 - dar, A este partea mai semnificativă a numărului, față de B

Exemplu cod implementare:

<https://www.cs.yale.edu/homes/aspnes/pinewiki/RadixSort.html>

Ordonarea sirurilor de caractere

RadixSort

- 1 caracter poate fi ordonat BinSort
- “spargerea” sirurilor in caractere
- Este necesara cunoasterea lungimii celui mai mare sir de caractere (sau se determina).

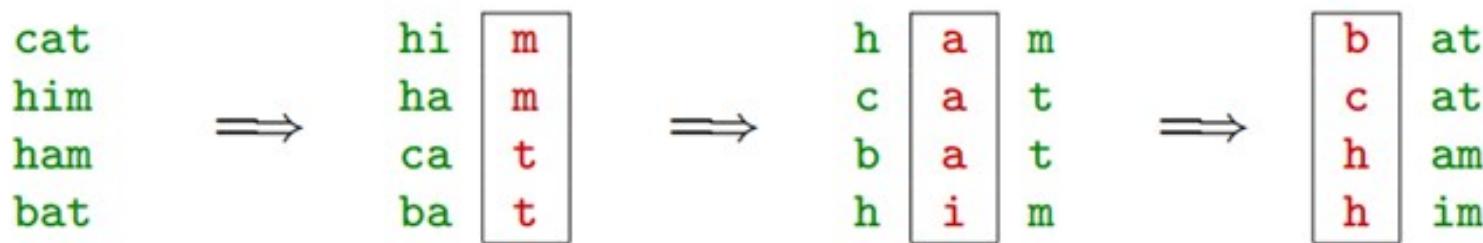
RadixSort - sir de caractere exemplu

	5 th pass	4 th pass	3 rd pass	2 nd pass	1 st pass
String 1	z	i	p	p	y
String 2	z	a	p		
String 3	a	n	t	s	
String 4	f	l	a	p	s

NULLs are
just like fake
characters

RadixSort - sir de caractere exemplu

$T = \{cat, him, ham, bat\}$



Este vizibil că după **i** iterații, sirurile sunt ordonate după **sufixul de lungimea **i****. Astfel, sunt complet ordonate la final.

RadixSort - sir de caractere timp de execuție

- N numărul de siruri de caractere
- L lungimea celui mai mare sir de caractere
- RadixSort $\in O(N \cdot L)$

RadixSort IEEE numere virgulă mobilă

- RadixSort oronează numerele reale, în majoritatea reprezentărilor
- IEEE real, utilizate în C/C++ (float / double)

Reprezentarea numerelor reale

Semn
(+ sau -)

-1.3892*10²⁴

Exponent

+1.507*10⁻¹⁷

Partea întreagă (mantisa)

IEEE float în binar

-1.0110100111*2¹⁰¹¹
+1.101101001*2⁻¹

- Semn: 1 bit
- Partea întreagă: întotdeauna 1.*fraction*. *fraction* folosește 23 biți
- Baza exponentului: 8 biți
 - Baza: reprezintă -127 la +127 prin adăugarea 127 (0-254)

Observații

- Partea întreagă întotdeauna începe cu 1
→ doar un singur mod de reprezentare a oricărui număr
- Exponent întotdeauna mai semnificativ decât partea întreagă
- Semnul este cel mai semnificativ

RadixSort – intreg

- Exemplu – intreg pe 16 biti
- 45289 - 1011000011101001
- 7786 - 0001111001101010
- 58640 - 1110010100010000

$$2^{16} = 65,536 \text{ chei distințe } ???$$

RadixSort – intreg

- Să presupunem că examinăm $d = 3$ biți din fiecare cheie per trecere

- 45289 - 1|011|000|011|101|001
- 7786 - 0|001|111|001|101|010
- 58640 - 1|110|010|100|010|000

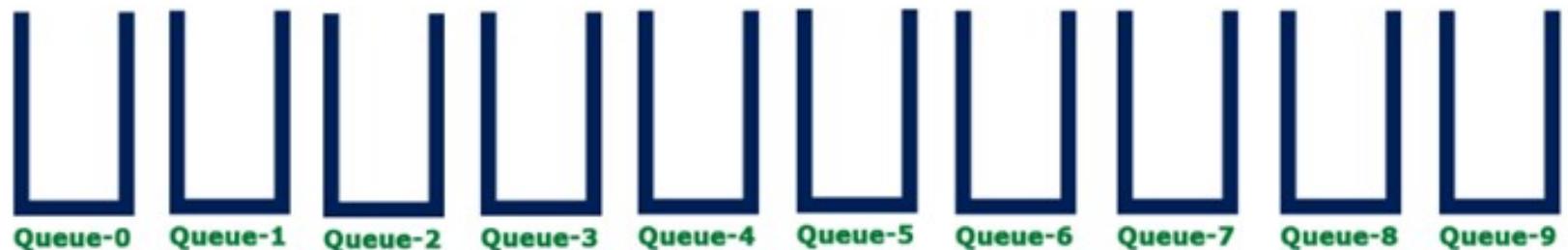
Zecimal	Binar
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

$2^3 = 8$ grupe distințe

RadixSort – intreg

82, 901, 100, 12, 150, 77, 55 & 23

1. Coadă/cifră



82, 901, 100, 12, 150, 77, 55 & 23

2. Less S D



100, 150, 901, 82, 12, 23, 55 & 77

RadixSort – intreg

100, 150, 901, 82, 12, 23, 55 & 77

100, 150, 901, 82, 12, 23, 55 & 77

3. Next Digit



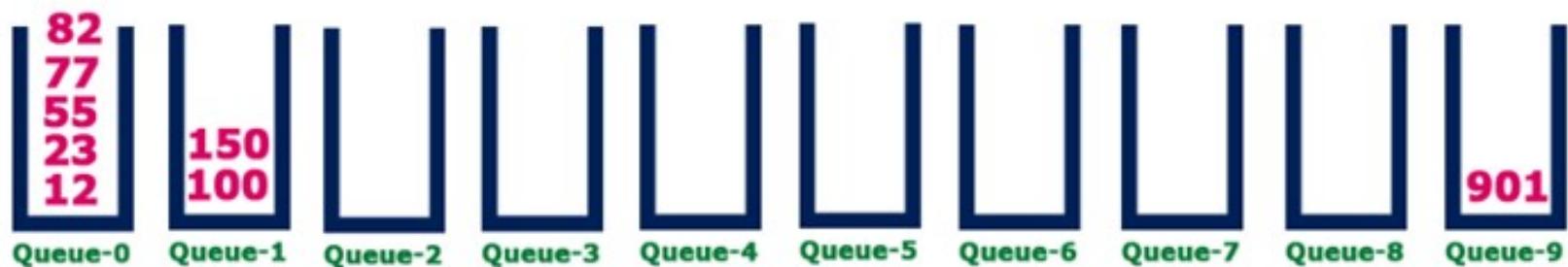
100, 901, 12, 23, 150, 55, 77 & 82

RadixSort – intreg

100, 901, 12, 23, 150, 55, 77 & 82

100, 901, 12, 23, 150, 55, 77 & 82

4. Next Digit



5. Final

12, 23, 55, 77, 82, 100, 150, 901

Worst Case : O(n)

Best Case : O(n)

Average Case : O(n)

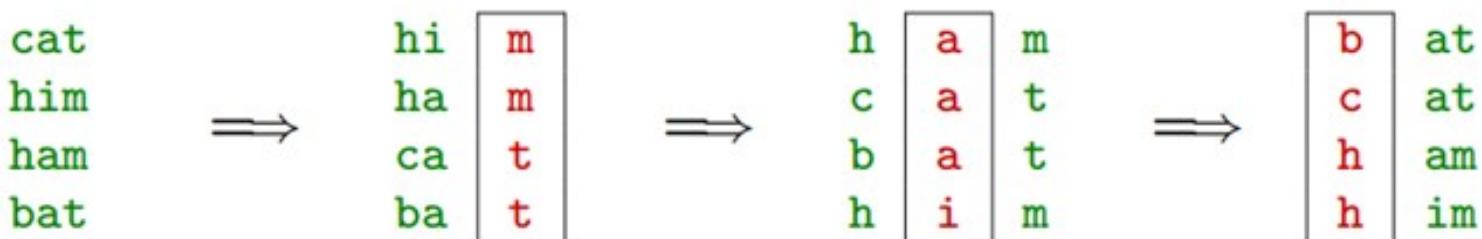
Radix Sort

- <https://www.hackerearth.com/practice/algorithms/sorting/radix-sort/visualize/>

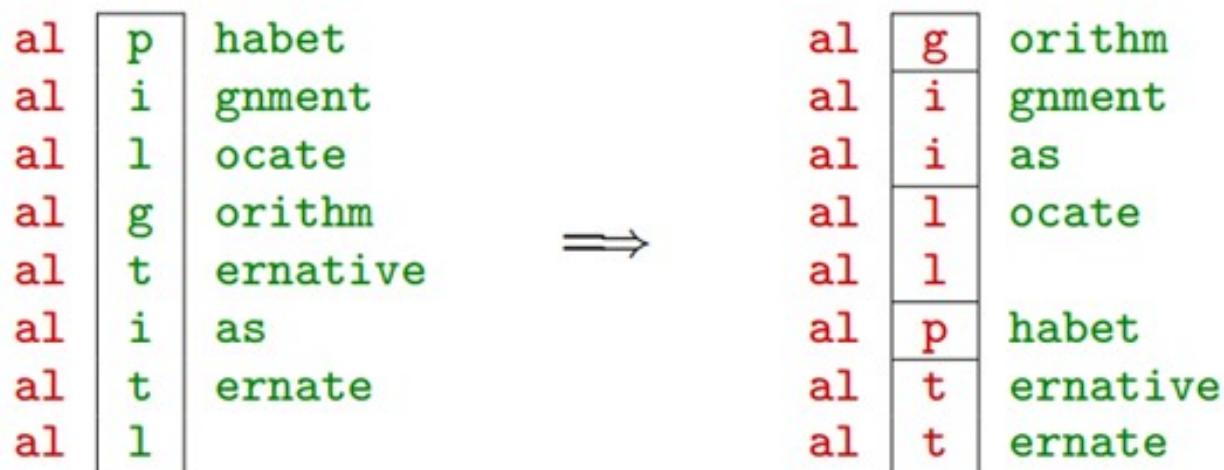
RadixSort – sir de caractere

LSD

$$\mathcal{R} = \{\text{cat}, \text{him}, \text{ham}, \text{bat}\}.$$

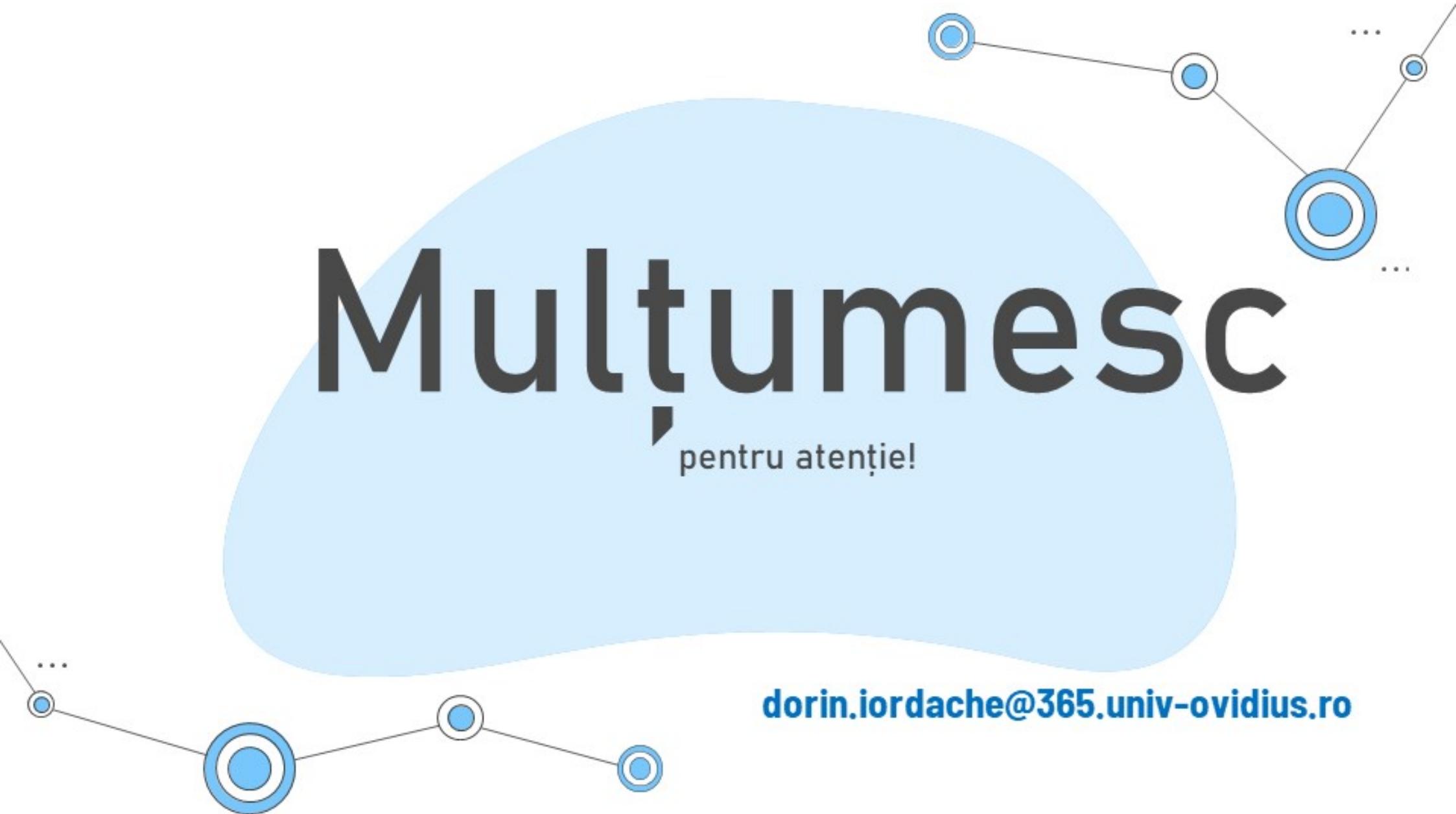


MSD radix sort partitioning.





Multumesc
pentru atenție!



dorin.iordache@365.univ-ovidius.ro