M. Caramihai, © 2020

STRUCTURI DE DATE & ALGORITMI

CURS 9

Algoritmi de sortare

Algoritmi de sortare (1)

- **Sortarea** reprezinta operatia de ordonare a unui set de elemente intr'o ordine crescatoare / descrescatoare.
- Ipoteze:
  - Elementele sunt comparabile
  - Elementele se gasesc stocate intr'un vector
  - Fiecare element se gaseste stocat intr'o singura componenta a vectorului
  - Pentru simplitate, elementele se considera a fi numere intregi; metodele pot fi utilizate insa pentru orice tip de elemente (ce pot fi ordonate).

Algoritmi de sortare (2)

- Exista diferiti algoritmi de sortare:
  - Algoritmi patratici: $O(N^2)$
    - Selection sort, Bubble sort, Insertion sort, etc.
  - Algoritmi logaritmici: O(NlogN)
    - Quick Sort, Merge Sort, etc.

- In general, un algoritm de sortare necesita **Ω(NlogN)** comparatii.

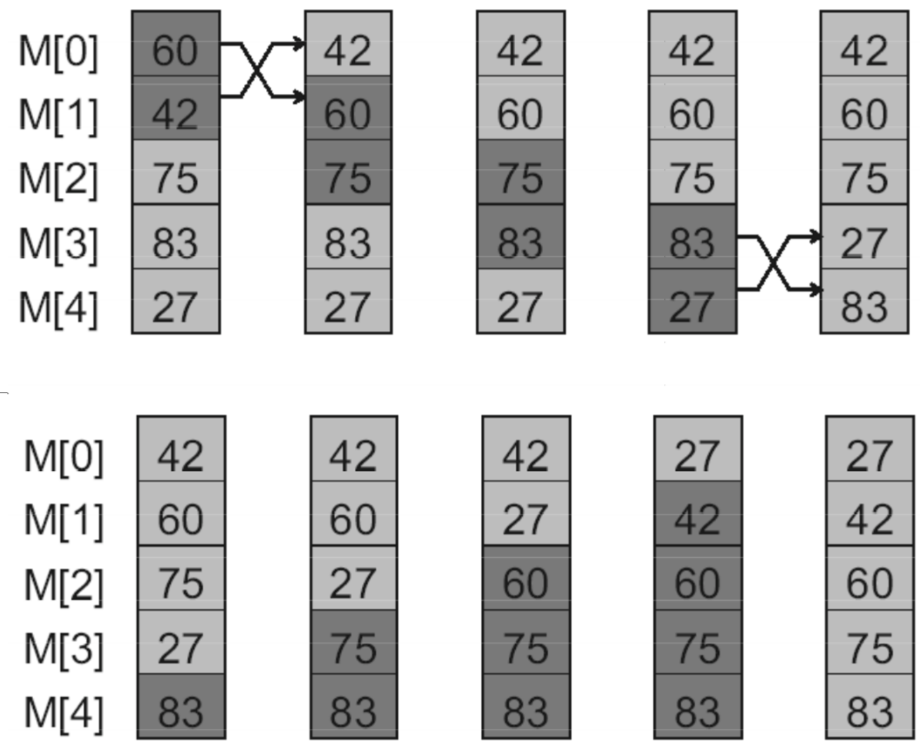
ALGORITMI DE SELECTIE PATRATICI

Bubble sort

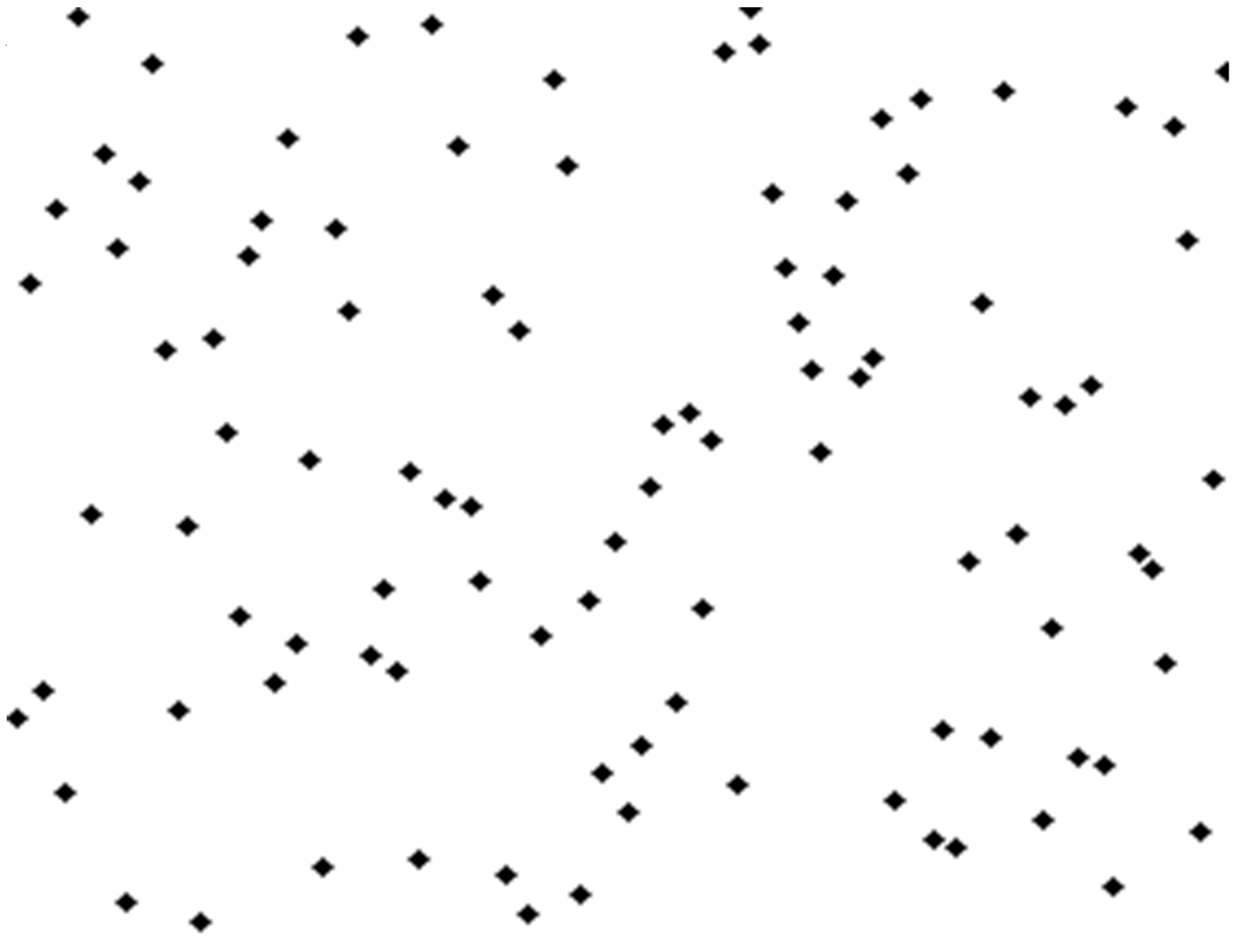
- Algoritmul ***Bubble sort***:
  - Compara elementele adiacente; daca primul este mai mare decat al doilea, ele isi schimba pozitiile (locurile).
  - Operatia se repeta pentru fiecare pereche de elemente adiacente (incepand cu primele doua si sfarsind cu ultimele doua). In acest moment, ultimul element trebuie sa fie cel mai mare.
- Complexitate: **O(*n*²)** → $O(n^2)$
- Exemplu: Fie lista {60, 42, 75, 83, 27}

Bubble sort – algoritm

```
for pass = 1 .. n-1
    exchange = false
    for position = 1 .. n-pass
        if element at position < element at position +1
            exchange elements
            exchange = true
        end if
    next position
    if exchange = false BREAK
next pass
```



Bubble sort – exemplu



Bubble sort – analiza

- Numar de comparatii (cazul nefericit):

  $(n-1) + (n-2) + ... + 3 + 2 + 1 \rightarrow O(n^2)$

- Numar de comparatii (cazul fericit):

  $n - 1 \rightarrow O(n)$

- Numar de schimbari (cazul nefericit):

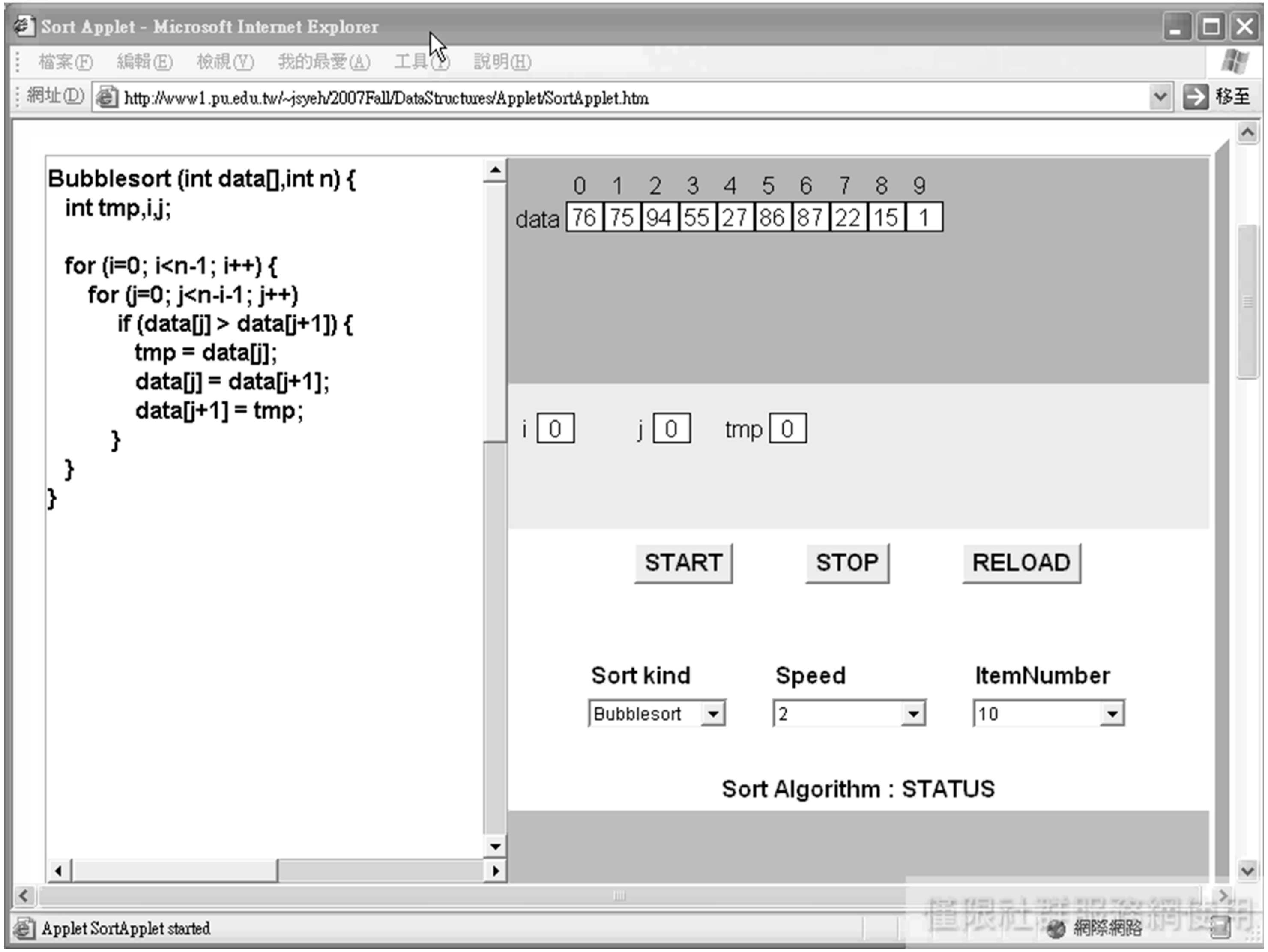
  $(n-1) + (n-2) + ... + 3 + 2 + 1 \rightarrow O(n^2)$

- Numar de schimbari (cazul fericit):

  $0 \rightarrow O(1)$

Cazul nefericit (global): $O(n^2) + O(n^2) = O(n^2)$

Bubble sort – simulare



Sursa: http://www1.pu.edu.tw

Insertion sort (1)

- **Algoritm**:
  - se presupune ca subsecventa (X[1], … , X[k-1]) este sortata. Se cauta in aceasta subsecventa locul **i** al elementului **X[k]** si se insereaza **X[k]** în pozitia **i**. Pozitia **i** este determinata astfel:
    - i = 1 dacă X[k] < X[1];
    - 1 < i < k si satisface X[i-1] <= X[k] < X[i];
    - i=k dacă X[k] >= X[k-1].
  - Pozitia lui **i** este determinată prin cautare secventiala de la dreapta la stanga simultan cu deplasarea elementelor mai mari decat X[j] cu o pozitie la dreapta. Cu alte cuvinte:
    - lista se parcurge de la stanga spre dreapta
    - la fiecare pas al parcurgerii listei elementul curent este memorat intr-o variabila distincta

Insertion sort (2)

- procesul de parcurgere incepe cu cel de-al doilea element al listei; daca este mai mare decat primul – ramane pe loc, iar daca nu, primul trece pe locul celui de-al doilea, iar cel de-al doilea trece pe prima pozitie
- pentru al treilea element al listei procesul se reia:
  - se memoreaza in variabila element
  - se parcurge lista inapoi pana se gaseste un element <= element; in acest moment locul valorii memorate in element este imediat dupa elementul mai mic decat element; odata cu parcurgerea inapoi, elementele > element se deplasează cu o pozitie la dreapta.

Insertion sort (3)

  - Daca, in urma cautarii, in submultimea din stanga nu se gaseste nici un element <=element, inseamna ca valoarea memorata in element este cea mai mica la momentul actual si se plaseaza pe prima pozitie
  - Procesul se repeta pentru al patrulea element, etc

- Complexitate: **O(*n*^{2})**

- Exemplu: fie lista {1, 12, 39, 3, 42}

Insertion sort – algoritm

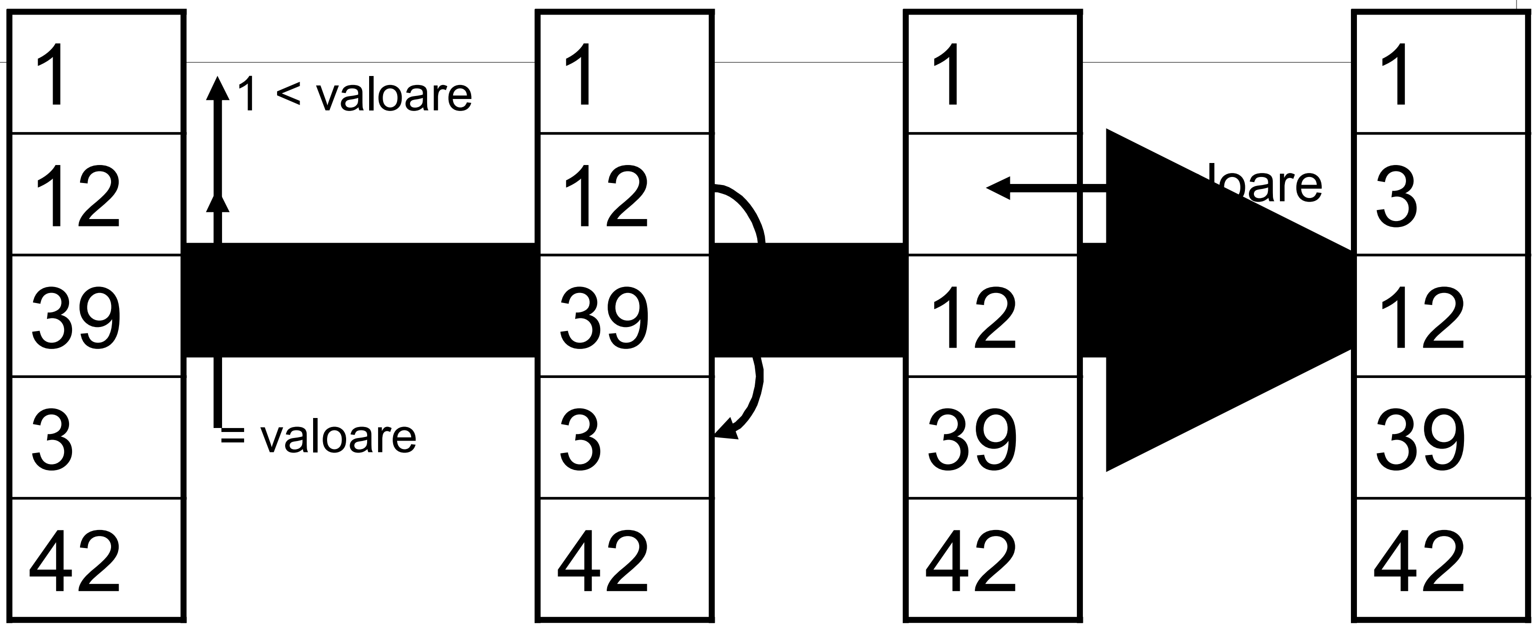
for pass = 2 .. n-1

value = element at pass

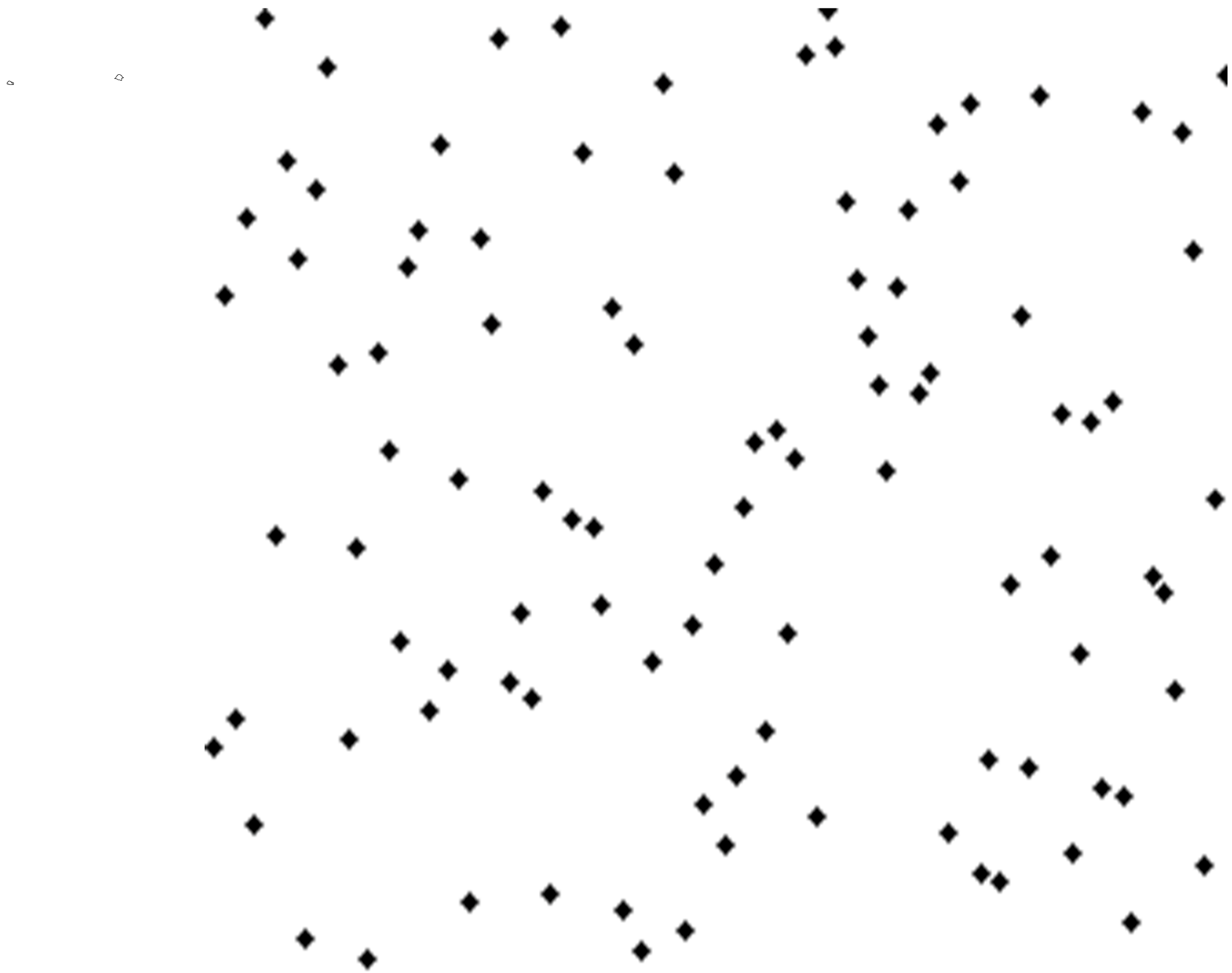
shift all elements > value in array 1..pass-1 one pos. right

place value in the array at the ‘vacant’ position

next pass



Insertion sort – exemplu



Sursa: Wikipedia.org

Insertion sort – analiza

- Numar de comparatii (cazul nefericit):

   $(n-1) + (n-2) + ... + 3 + 2 + 1 \rightarrow O(n^2)$

- Numar de comparatii (cazul fericit):

   $n - 1 \rightarrow O(n)$

- Numar de schimbari (cazul nefericit):

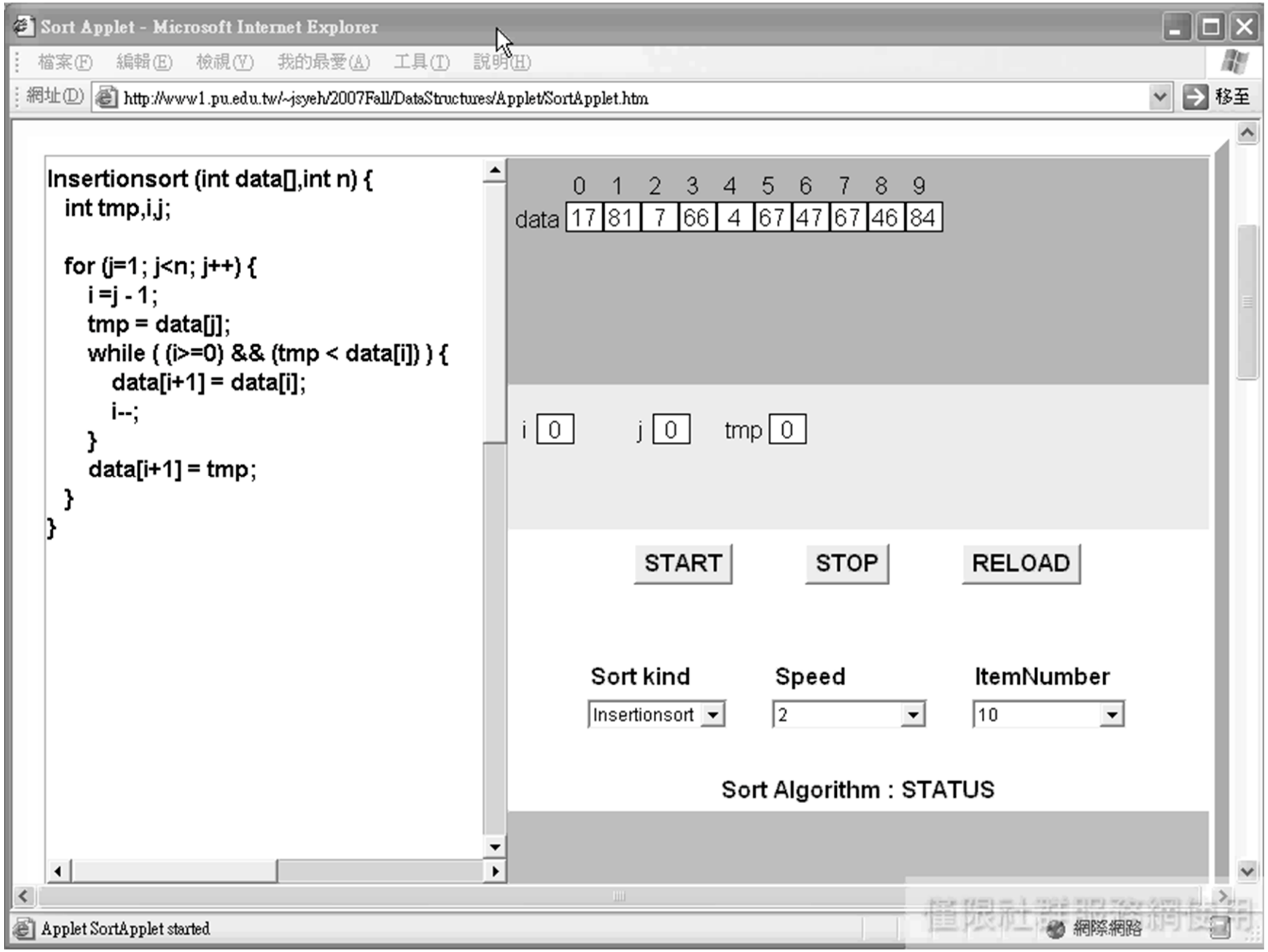
   $(n-1) + (n-2) + ... + 3 + 2 + 1 \rightarrow O(n^2)$

- Numar de schimbari (cazul fericit):

   $0 \rightarrow O(1)$

Cazul nefericit (global): $O(n^2) + O(n^2) = O(n^2)$

Insertion sort – simulare



Sursa: http://www1.pu.edu.tw

Analiza comparativa (1)

	Comparatie		Schimb	
	Cel mai bun	Cel mai rau	Cel mai bun	Cel mai rau
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$

Analiza comparativa (2)

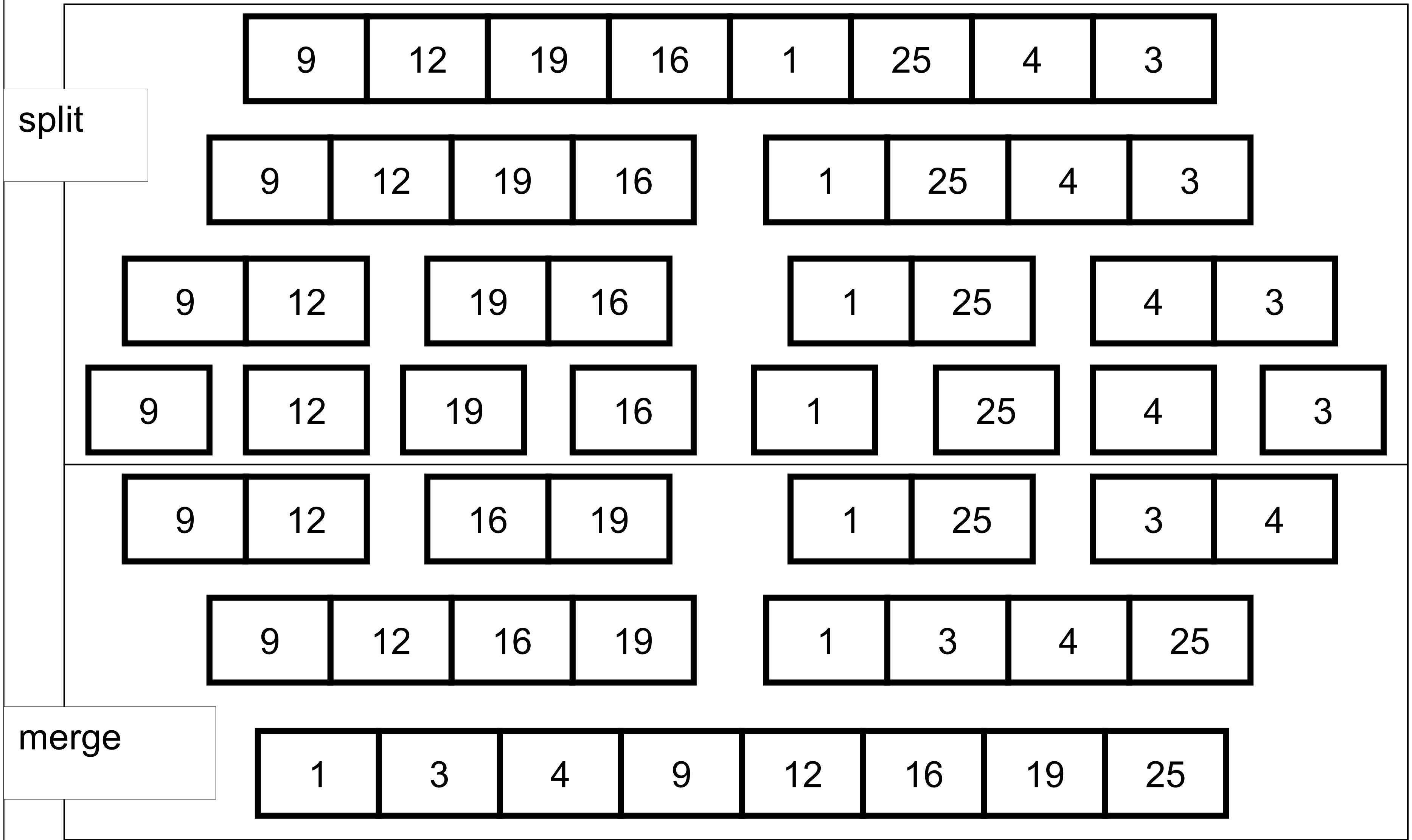
	Pro	Contra
Bubble Sort	Cand vectorul este pre-sortat	Cand vectorul se afla intr'o dezordine totala
Insertion Sort	Cand vectorul este pre-sortat	Cand vectorul se afla intr'o dezordine totala

ALGORITMI DE SELECTIE LOGARITMICI

Merge sort

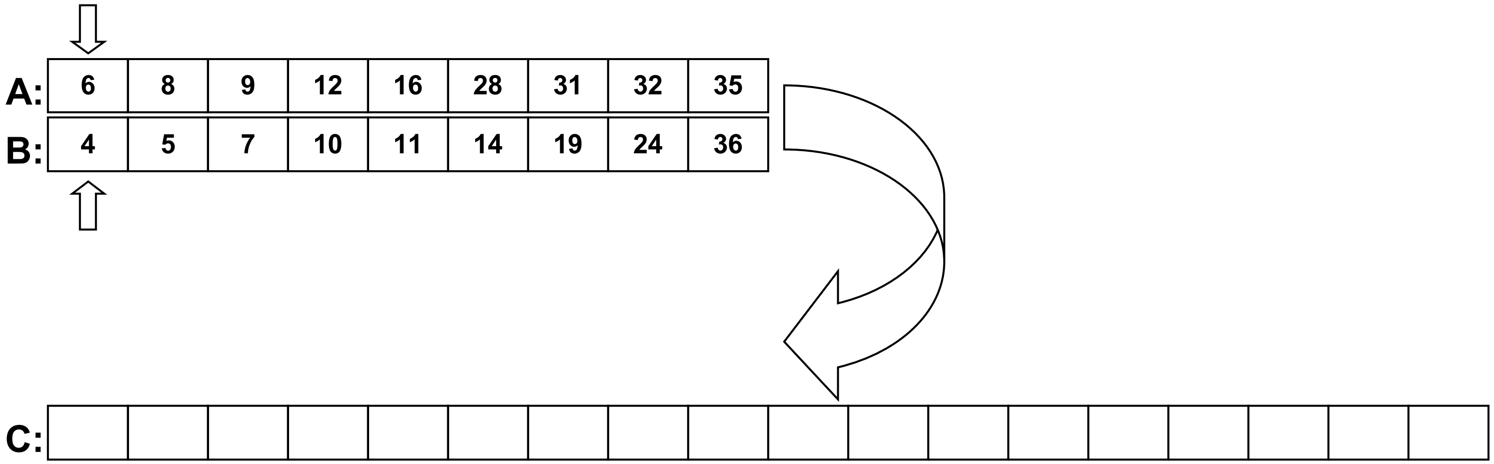
- **Algoritm**:
  - Lista nesortata se imparte in doua sub-liste aproximativ egale
  - Fiecare din aceste doua sub-liste se imparte la randul ei in cate doua sub-liste, in mod recursiv, pana se obtine lista de lungime 1 (caz in care lista se returneaza).
  - Cele doua sub-liste sunt apoi asociate intr'o lista noua.
- **Merge sort** inglobeaza doua idei ce permit imbunatatirea timpului de rulare:
  - O lista de lungime mica va necesita un timp de lucru mai mic (/ un numar mai mic de pasi) decat o lista de lungime mai mare.
  - Constructia unei liste (ordonate) din doua subliste ordonate necesita un numar mai mic de pasi decat daca cele doua liste ar fi ne-ordonate.

Merge sort – exemplu



Merge sort – implementare (1)

- Implementare
  - Fie secventele A si B (ordonate crescator); se cere sa se obtina secventa rezultat C (ordonata tot crescator)



Merge sort – implementare (2)

- Fie cate un pointer (indicator) catre primul element al fiecarei secvente
- La fiecare pas se compara cele 2 elemente (spre care indica pointerii)
- Minimul dintre cele 2 elemente este copiat in secventa rezultat si pointerul corespunzator este mutat o pozitie mai la dreapta (celalalt pointer ramanand nemiscat)
- Daca vreunul dintre pointeri depaseste limita dreapta a secventei asociate, se copiaza elementele ramase in cealalta secventa, in secventa rezultat
- Algoritmul se incheie in momentul in care ambii pointeri au depasit limita dreapta a secventelor asociate

Merge sort – implementare (3)

A:	6	8	9	12	16	28	31	32	35
B:	4	5	7	10	11	14	19	24	36

4 < 6, trece 4

C:	4																	

A:	6	8	9	12	16	28	31	32	35
B:	4	5	7	10	11	14	19	24	36

5 < 6, trece 5

C:	4	5																

Merge sort – implementare (4)

A:	6	8	9	12	16	28	31	32	35
B:	4	5	7	10	11	14	19	24	36

35 < 36, trece 35

C:	4	5	6	7	8	9	10	11	12	14	16	19	24	28	31	32	35	

A:	6	8	9	12	16	28	31	32	35
B:	4	5	7	10	11	14	19	24	36

trece 36

C:	4	5	6	7	8	9	10	11	12	14	16	19	24	28	31	32	35	36

Merge sort – implementare (5)

... etc ... Se obtine:

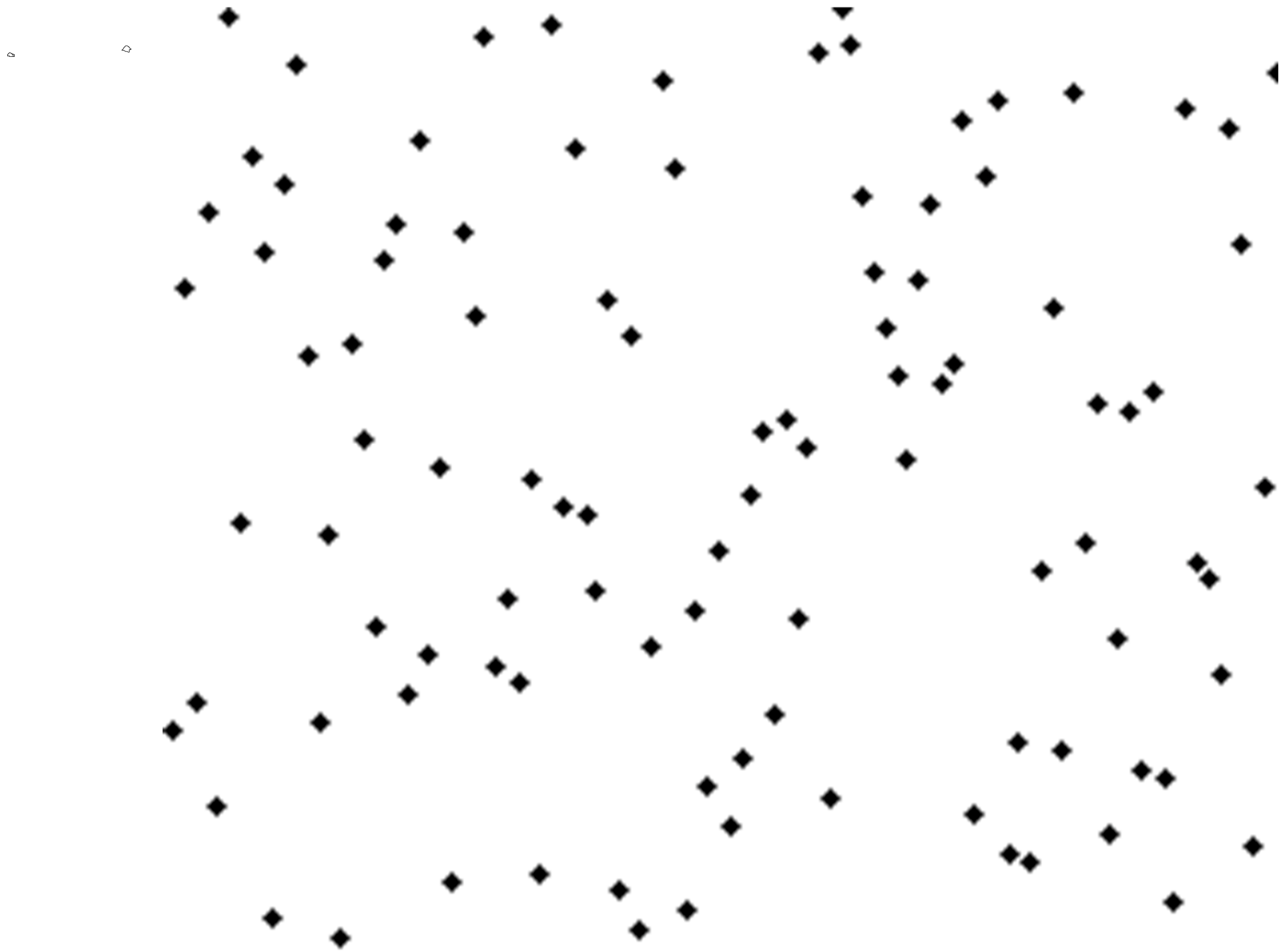
A:	6	8	9	12	16	28	31	32	35
B:	4	5	7	10	11	14	19	24	36

C:	4	5	6	7	8	9	10	11	12	14	16	19	24	28	31	32	35	36

- Ambii pointeri au depasit limita dreapta a secventelor asociate, → **STOP** algoritm.

- Mecanismul pointerilor poate fi implementat explicit, cu ajutorul unor variabile intregi care memoreaza indicii la care s-a ajuns in cadrul fiecarei secvente (daca este vorba despre tablouri), sau implicit, cu ajutorul pointerilor de fisier – in cazul fisierelor.

Merge sort – simulare



Sursa: Wikipedia.org

Merge sort – concluzii

- Algoritmul este foarte rapid, complexitatea sa fiind proportionala cu suma lungimilor celor 2 secvente

- Nu este necesar accesul aleator in cadrul secventelor, ci doar accesul secvential – secventele fiind parcurse de la inceput la sfarsit, fara reveniri

- **Observatie**: Aceasta caracteristica (care nu se regaseste la algoritmi de sortare specifici tablourilor), face ca ***merge sort*** sa se preteze, in special, la sortarea fisierelor secventiale

- **Dezavantaj**: **Merge Sort** necesita memorie suplimentara !

Heap sort (1)

- **HeapSort** este unul din algoritmii de sortare foarte performanti, fiind de clasa **O(N·log$_2$N)**; este cunoscut si sub denumirea de “sortare prin metoda ansamblelor”
- Desi nerecursiv, **HeapSort** este aproape la fel de performant ca si algoritmii de sortare recursivi (**QuickSort** fiind cel mai cunoscut)
- **HeapSort** este un algoritm de sortare “in situ”, adica nu necesita structuri de date suplimentare, ci sortarea se face folosind numai spatiul de memorie al tabloului ce trebuie sortat
- Exista si implementari **HeapSort** care nu sunt “in situ”

Heap sort (2)

- Algoritmul **HeapSort** se aseamana, in unele privinte, cu sortarea prin selectie (**SelSort**)
- Astfel, la fiecare pas, cel mai mic element din tablou este identificat si mutat in spatele tabloului, fiind ignorat de pasii urmatori, care vor continua cu restul tabloului
- Diferenta fata de **SelSort** este ca pasii urmatori ai algoritmului vor depune un efort *mai mic* pentru a depista minimul din tabloul ramas
- Fiecare pas al algoritmului are rolul de a usura sarcina pasilor ce urmeaza, ceea ce duce la o performanta foarte buna a algoritmului

Heap sort (3)

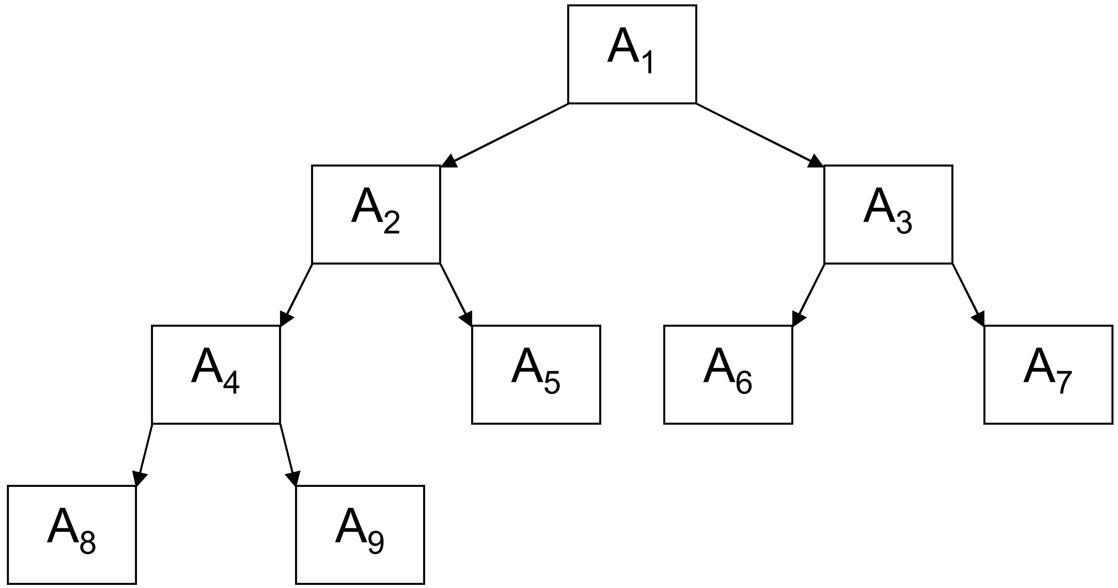
- **Algoritm**:
  - Gasirea minimului din tablou, operatie ce are loc la fiecare pas, se bazeaza pe aducerea tabloului la forma de ***ansamblu***
  - Un ***ansamblu*** este un sir A_i (i = 1 … N) care indeplineste urmatoarele conditii pentru fiecare i:
    - $A_i \leq A_{2 \cdot i}$
    - $A_i \leq A_{2 \cdot i+1}$
  - Evident, pentru valori ale lui *i* mai mari decat N/2 nu se pune problema indeplinirii conditiilor de mai sus

Sursa: Calin Jebelean, Algoritmul Heap Sort

Heap sort – implementare (1)

- Orice tablou poate fi transformat intr-un arbore binar

Index:	1	2	3	4	5	6	7	8	9
A:	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9



Heap sort – implementare (2)

- Daca tabloul este un ***ansamblu***, se observa ca arborele binar obtinut indeplineste urmatoarea conditie: “fiecare nod are cheia mai mare sau egala cu a parintelui sau”.
- Astfel, A_2 si A_3 sunt mai mari sau egale cu A_1, A_4 si A_5 sunt mai mari sau egale cu A_2, etc
- Dar A_1 este radacina arborelui binar, ceea ce inseamna ca A_1 trebuie sa fie elementul minim al tabloului
- *Ergo*, intr-un ***ansamblu***, elementul minim se afla intotdeauna pe prima pozitie
- In cadrul algoritmului **HeapSort**, daca la fiecare pas se aduce tabloul pe care se lucreaza la forma unui ***ansamblu*** , inseamna ca a fost localizat in acelasi timp si minimul din tablou

Heap sort – implementare (3)

- Aducerea unui tablou la forma de ***ansamblu*** se face urmarind exemplul de mai jos:

Index:	1	…	i	…	2·i	2·i+1	…	N
A:	A_1	…	A_i	…	$A_{2\cdot i}$	$A_{2\cdot i+1}$	…	A_N

- Daca nu este indeplinita una din conditiile $A_i \le A_{2\cdot i}$ si $A_i \le A_{2\cdot i+1}$ atunci se va interschimba A_i cu minimul dintre $A_{2\cdot i}$ si $A_{2\cdot i+1}$
- Elementele astfel interschimbate vor indeplini conditia de ***ansamblu***
- Pentru o eficienta maxima, urmarirea acestui tip de situatii trebuie facuta de la dreapta la stanga, in caz contrar fiind nevoie de reveniri repetate chiar si dupa ce o situatie de neconcordanta a fost rezolvata

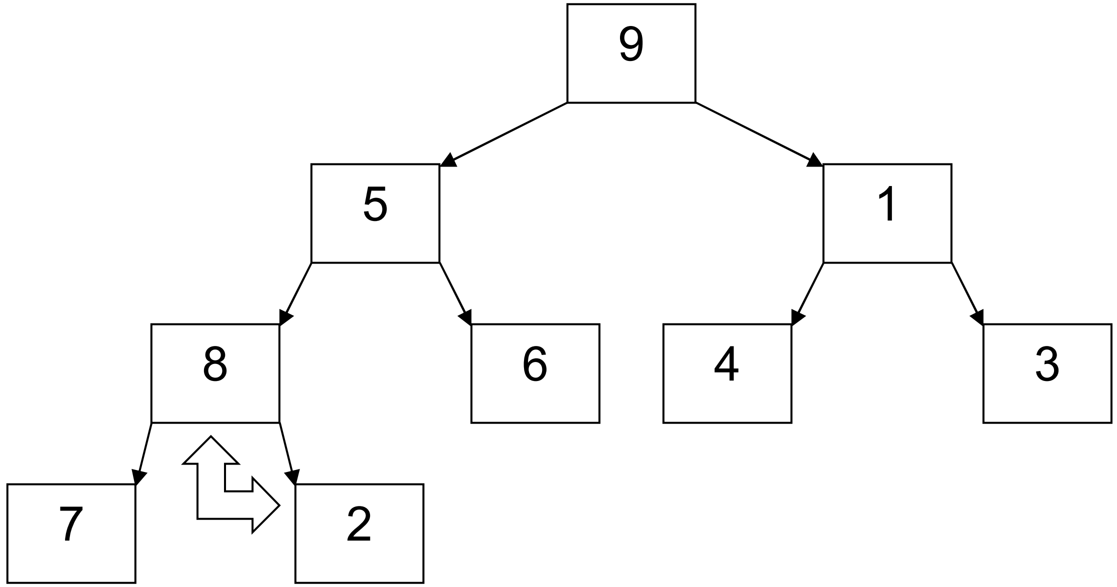
Heap sort – implementare (4)

- Transformarea in ***ansamblu*** a unui tablou se va aplica la fiecare pas in cadrul algoritmului **HeapSort**, pe un tablou din ce in ce mai mic (deoarece dupa fiecare pas, primul element al tabloului, care este elementul minim, va fi eliminat si “pus la pastrare”, algoritmul continuand cu restul tabloului)

- Astfel, se ia in considerare reprezentarea sub forma de arbore a tabloului:

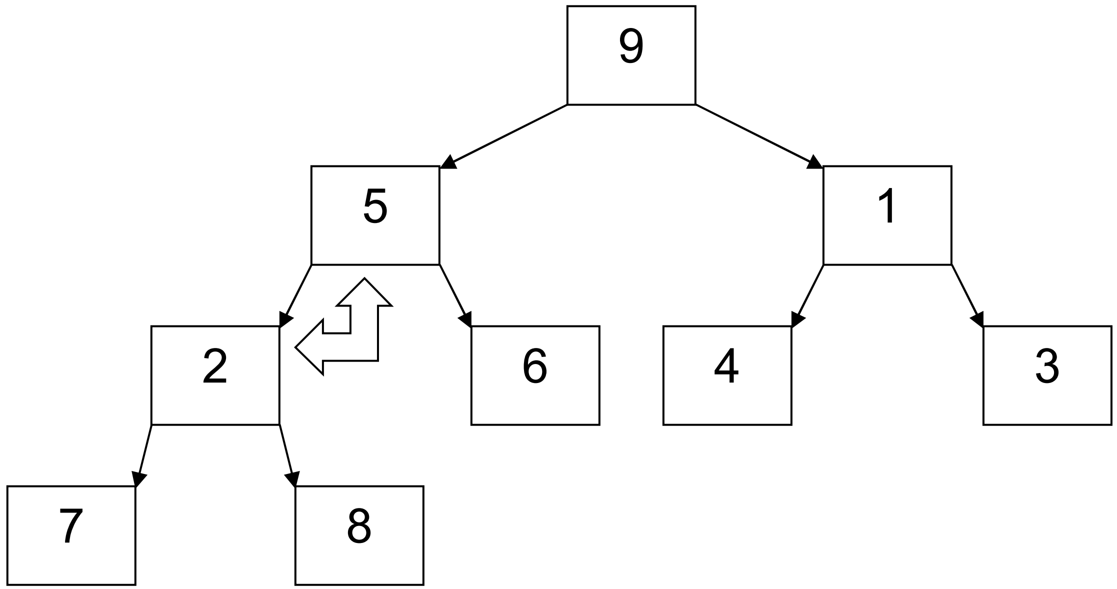
Index:	1	2	3	4	5	6	7	8	9
A:	9	5	1	8	6	4	3	7	2

Heap sort – implementare (5)



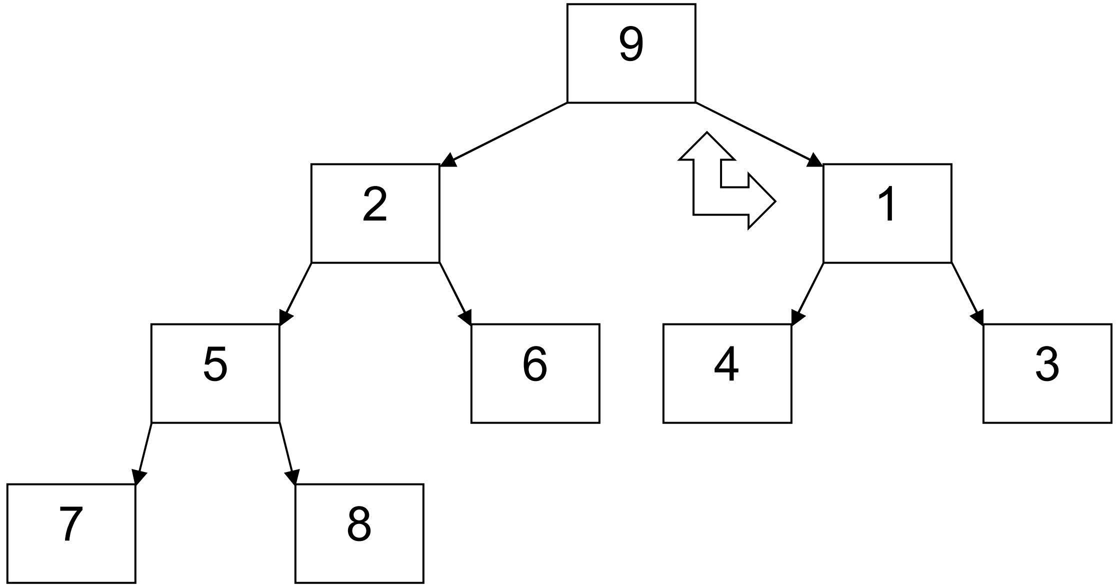
- Problema se pune numai pentru noduri *neterminale*
- Trebuie localizat cel mai de jos nod neterminal, si in caz ca sunt mai multe astfel de noduri, se considera cel mai din dreapta – i.e. 8
- Deaoarece 8 are fiii 7 si 2 si este mai mare decat ambii, se va interschimba cu cel mai mic dintre ei, adica cu 2

Heap sort – implementare (6)



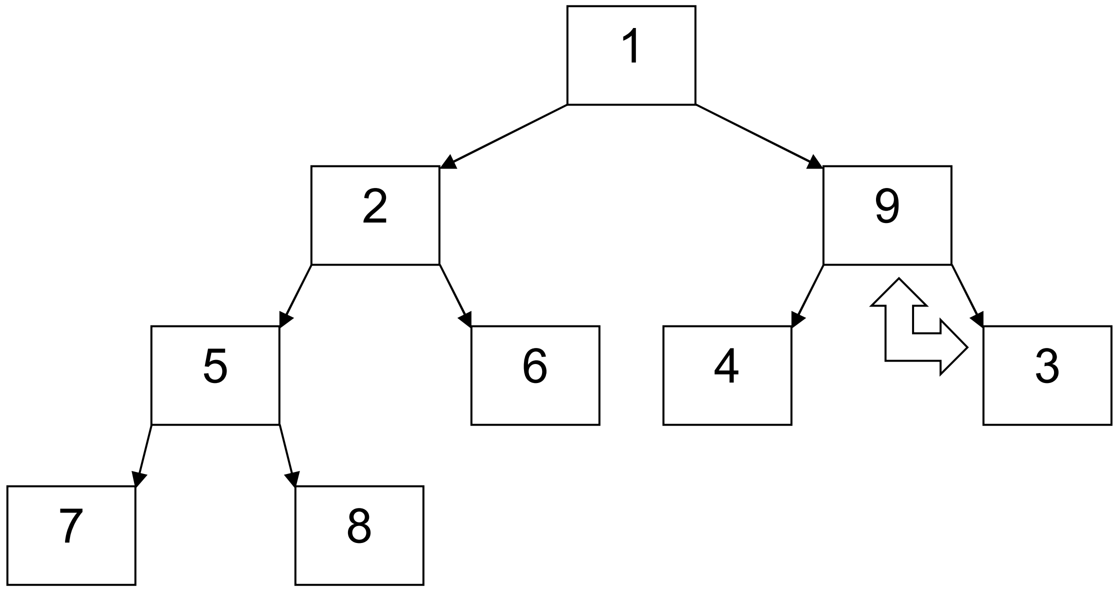
- Urmatorul nod neterminal este 1 (nodul 5 este pe acelasi nivel, dar va fi ales intotdeauna cel mai din dreapta in astfel de cazuri)
- Nodul 1 este mai mic decat fiii sai, deci nu va face obiectul vreunei interschimbari
- Urmeaza nodul 5: acesta nu indeplineste conditiile → va fi interschimbat cu cel mai mic fiu al sau, i.e. 2

Heap sort – implementare (7)



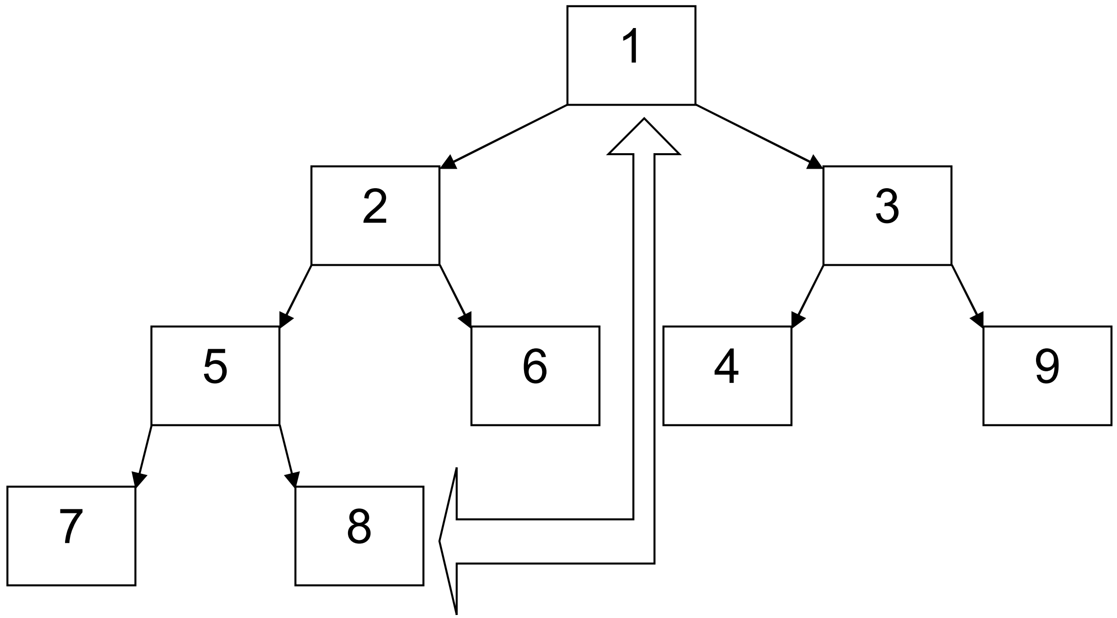
- Inainte de a trece la noul nod neterminal, se verifica faptul ca ultimul nod interschimbat (5) indeplineasca conditia referitoare la fiii sai (7 si 8) – se observa ca o indeplineste
- Noul nod neterminal este 9
- Acesta nu indeplineste conditiile, fiind mai mare si decat 2 si decat 1 → 9 va fi interschimbat cu cel mai mic, deci cu 1

Heap sort – implementare (8)



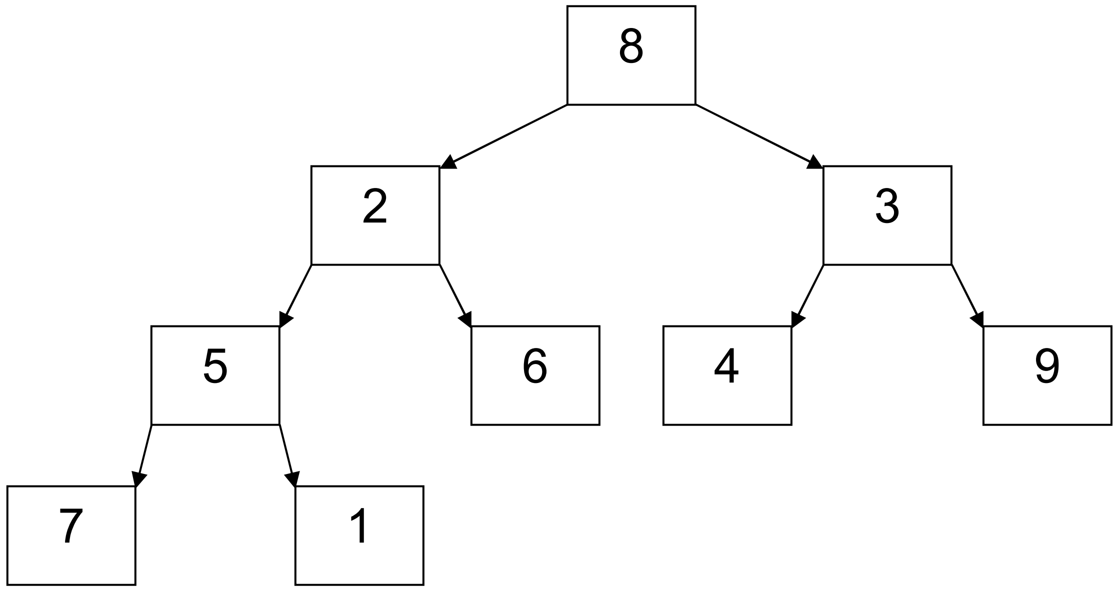
- Daca ultimul nod interschimbat (9) inca nu indeplineste conditiile referitoare la fiii sai, atunci: 9 fiind mai mare si decat 4 si decat 3, se va interschimba cu 3 (cel mai mic)
- Astfel de interschimbari repetate vor avea loc pana cand 9 ajunge pe un nivel pe care fiii sai sunt mai mari sau egali cu el (sau pe un nivel unde nu mai are fii)

Heap sort – implementare (9)



- 9 a ajuns pe un nivel terminal (nu mai are fii) → STOP
- Tabloul a ajuns la forma de ansamblu, fiecare nod avand cheia mai mica sau egala decat cheile fiilor sai
- Cel mai mic element al tabloului a ajuns pe post de radacina
- Se interschimba radacina cu ultimul element al tabloului (i.e. 1 cu 8)

Heap sort – implementare (10)

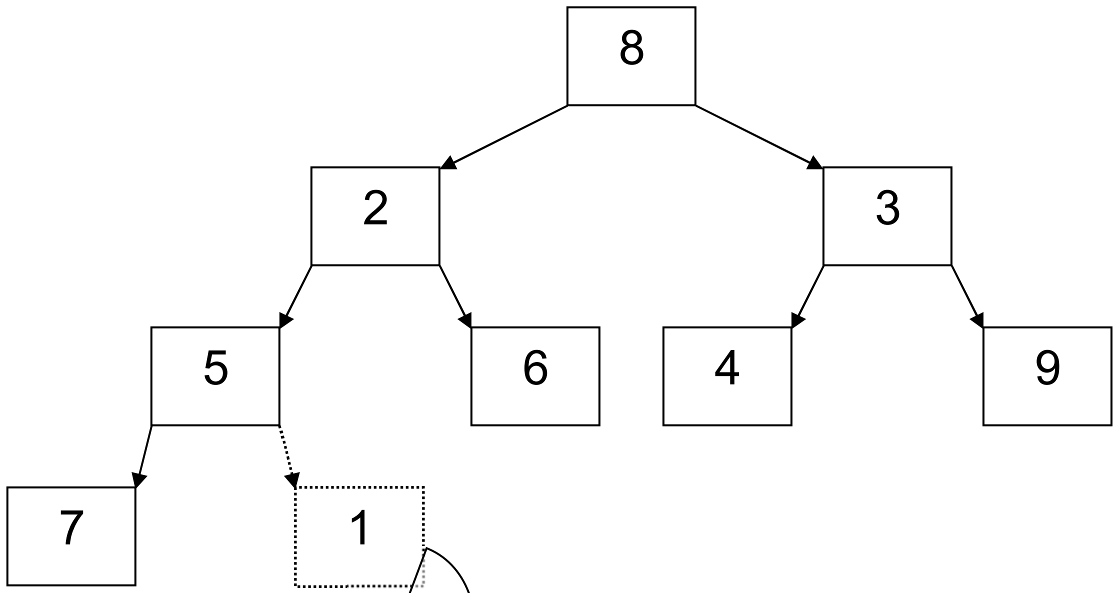


- Elementul minim (1) se elimina si se adauga la un tablou auxiliar (initial gol) care va contine la final elementele sortate

→ primul pas al algoritmului de sortare **HeapSort**

Heap sort – implementare (11)

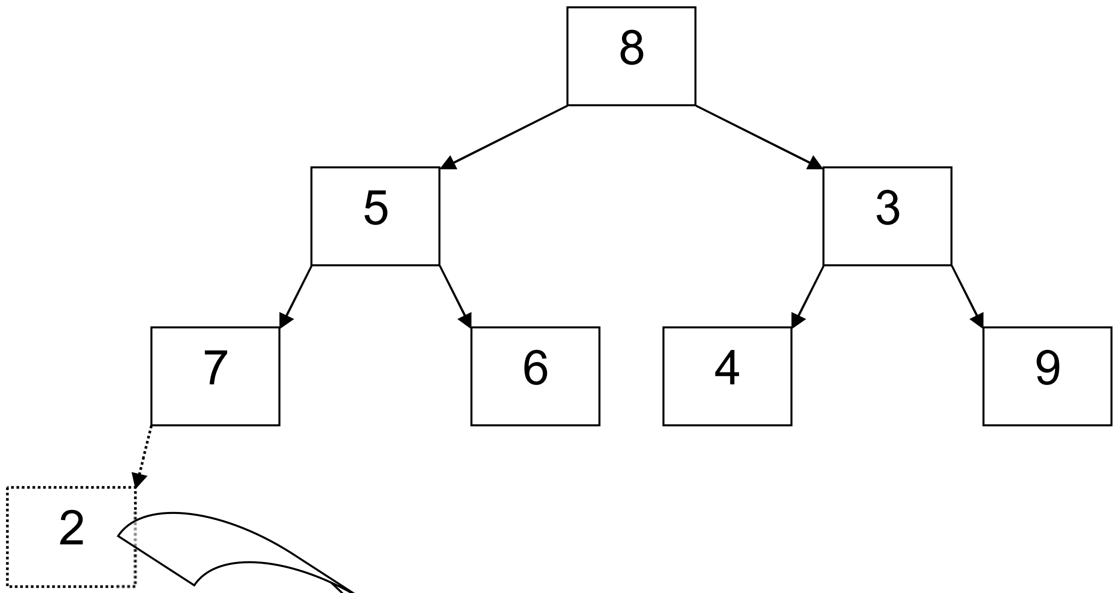
- Situatia actuala:



Index:	1	2	3	4	5	6	7	8	9
Tablou auxiliar:	1	-	-	-	-	-	-	-	-

Heap sort – implementare (12)

- Situatia actuala (ulterior):

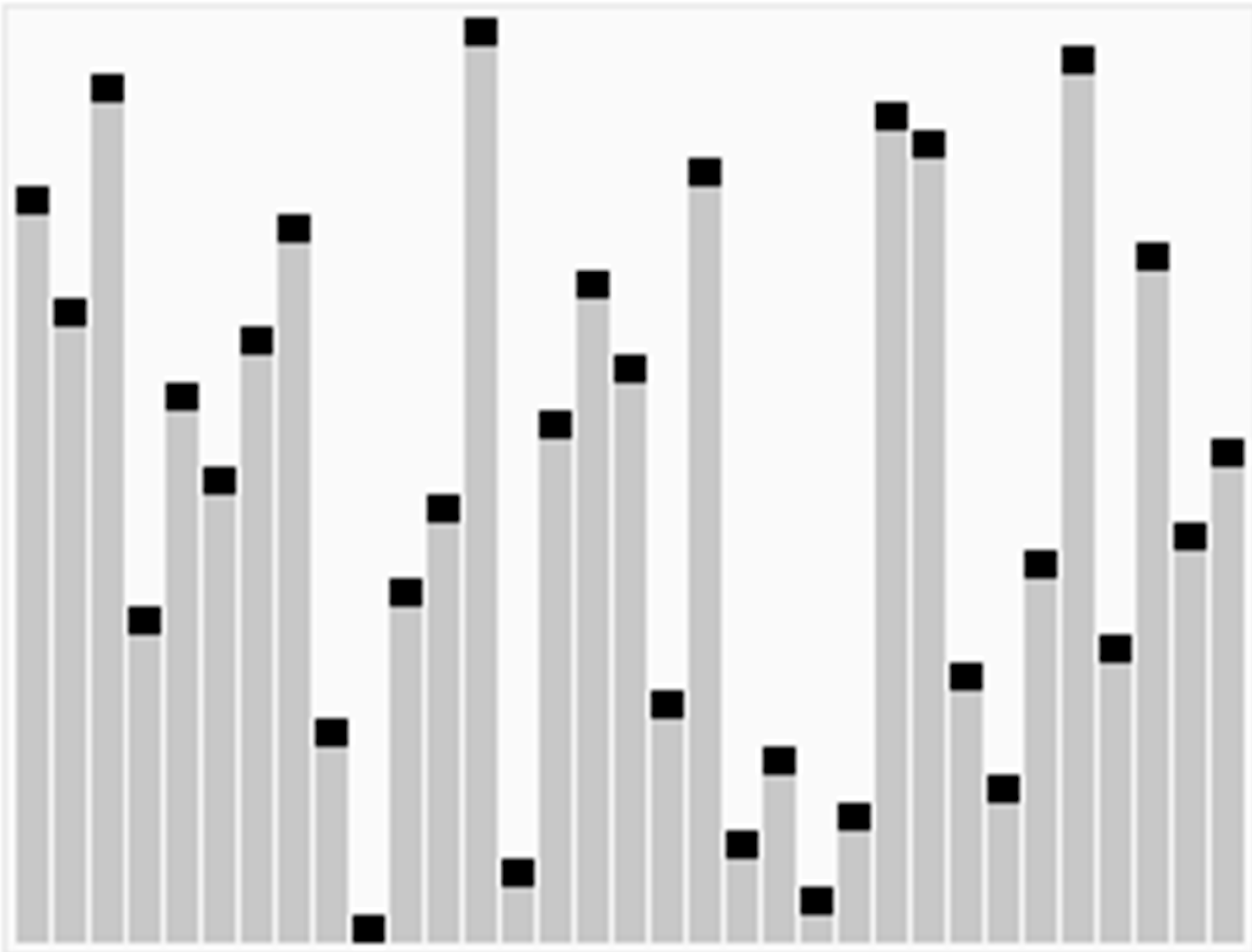


Index:	1	2	3	4	5	6	7	8	9
Tablou auxiliar:	1	2	-	-	-	-	-	-	-

Heap sort – concluzii

- Algoritmul **HeapSort** este cel mai slab algoritm de clasa **O(N·log$_2$N)**
- Este mai slab (dar nu cu mult) decat algoritmii din familia **QuickSort**, dar are marele avantaj fata de acestia ca nu este recursiv
- Algoritmii recursivi ruleaza rapid, dar consuma o mare cantitate de memorie, ceea ce nu le permite sa sorteze tablouri de dimensiuni oricat de mari
- **HeapSort** este un algoritm care "impaca" viteza cu consumul relativ mic de memorie

Heap sort – simulare



Sursa: Wikipedia.org

Shell sort

- **ShellSort** este un algoritm de sortare performant, bazat pe sortarea prin insertie (**InsertSort**), fiind supranumit si “insertie cu diminuarea incrementului”
- Algoritmul lucreaza pe tablouri de lungime **N**, fiind de clasa **O(N^2),** clasa ce caracterizeaza, in mod normal, algoritmii de sortare mai putin performanti
- Cu toate acestea, algoritmul este vizibil mai rapid decat algoritmii obisnuiti din clasa **O(N^2)**: **InsertSort, BubbleSort, ShakerSort, SelSort**, etc., fiind de circa 2 ori mai rapid decat **InsertSort,** cel mai apropiat competitor din clasa **O(N^2)**
- **ShellSort** nu este un algoritm de sortare “in situ”, adica necesita structuri de date suplimentare, in plus fata de spatiul de memorie al tabloului ce trebuie sortat – spatiul suplimentar este revendicat de un tablou de incrementi necesar rularii algoritmului

Shell sort – implementare (1)

- Fie urmatorul tablou ce trebuie sortat:

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	17	8	3	21	14	24	2	12	30	9	4	19	6	18	23	15	7	13	1

- Algoritmul **Shellsort** propune alegerea in prealabil a unui tablou H, numit “tablou de incrementari”:
- Tabloul de incrementari trebuie sa indeplineasca conditiile:
  - H are M elemente (0 … M-1) cu M>1
  - H[M-1] = 1 (ultimul element trebuie sa fie 1)
  - H[i] > H[i+1] pentru i = 0 .. M-2 (H trebuie sa fie un tablou strict descrescator)
- Exemplu: H = [7, 4, 3, 2, 1]

Shell sort – implementare (2)

- Algoritmul va face M treceri asupra tabloului A, (M este lungimea tabloului de incrementare)
- Dupa fiecare pas i, elementele aflate in tabloul A la distanta H[i] unul de altul vor fi sortate
- Sortarea acestor elemente se va face folosind algoritmul **InsertSort**

Shell sort – implementare (3)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	17	8	3	21	14	24	2	12	30	9	4	19	6	18	23	15	7	13	1

- Pasul 0: sortare (folosind **InsertSort**) a elementelor aflate la distanta H[0]=7 unul de celalalt, i.e.
  - 17, 12, 23
  - 8, 30, 15
  - 3, 9, 7
  - s. a. m. d.

- Aceste elemente ocupa celulele colorate la fel in reprezentarea de mai sus

Shell sort – implementare (4)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	17	8	3	21	14	24	2	12	30	9	4	19	6	18	23	15	7	13	1

- Elementele tabloului se copiaza intr’o matrice avand 7 coloane si se sorteaza pe coloane (sortarea coloanelor foloseste tehnica **InsertSort**):

17	8	3	21	14	24	2
12	30	9	4	19	6	18
23	15	7	13	1		

12	8	3	4	1	6	2
17	15	7	13	14	24	18
23	30	9	21	19		

- Ulterior, se reface tabloul initial din liniile matricii, (observatie: daca se iau elementele din 7 in 7, se obtin siruri sortate)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	12	8	3	4	1	6	2	17	15	7	13	14	24	18	23	30	9	21	19

Shell sort – implementare (6)

4 4 4

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	12	8	3	4	1	6	2	17	15	7	13	14	24	18	23	30	9	21	19

- Pasul 1: sortare (folosind **InsertSort**) a elementelor aflate la distanta H[1]=4 unul de celalalt, i.e.
  - 12, 1, 15, 24, 9
  - 8, 6, 7, 18, 21
  - 3, 2, 13, 23, 19
  - 4, 17, 14, 30
- Aceste elemente ocupa celulele colorate la fel in reprezentarea de mai sus

Shell sort – implementare (7)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	12	8	3	4	1	6	2	17	15	7	13	14	24	18	23	30	9	21	19

- Elementele tabloului vor fi copiate intr-o matrice avand 4 coloane si vor fi sortate pe coloane (sortarea coloanelor foloseste tehnica **InsertSort**):

12	8	3	4
1	6	2	17
15	7	13	14
24	18	23	30
9	21	19	

$\Rightarrow$

1	6	2	4
9	7	3	14
12	8	13	17
15	18	19	30
24	21	23	

- Se reface tabloul:

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	6	2	4	9	7	3	14	12	8	13	17	15	18	19	30	24	21	23

Shell sort – implementare (8)

3 3 3 3 3

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	6	2	4	9	7	3	14	12	8	13	17	15	18	19	30	24	21	23

- Elementele tabloului vor fi copiate intr-o matrice avand 3 coloane si vor fi sortate pe coloane (sortarea coloanelor foloseste tehnica **InsertSort**):
  - 1, 4, 3, 8, 15, 30, 23
  - 6, 9, 14, 13, 18, 24
  - 2, 7, 12, 17, 19, 21

- Aceste elemente ocupa celulele colorate la fel in reprezentarea de mai sus

Shell sort – implementare (9)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	6	2	4	9	7	3	14	12	8	13	17	15	18	19	30	24	21	23

- Elementele tabloului vor fi copiate intr-o matrice avand 3 coloane si vor fi sortate pe coloane (sortarea coloanelor foloseste tehnica **InsertSort**):

1	6	2
4	9	7
3	14	12
8	13	17
15	18	19
30	24	21
23		

1	6	2
3	9	7
4	13	12
8	14	17
15	18	19
23	24	21
30		

- Se reface tabloul:

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	6	2	3	9	7	4	13	12	8	14	17	15	18	19	23	24	21	30

Shell sort – implementare (10)

			2		2		2		2		2		2		2		2		
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	6	2	3	9	7	4	13	12	8	14	17	15	18	19	23	24	21	30

- Pasul 3: sortare (folosind **InsertSort**) a elementelor aflate la distanta H[3]=2 unul de celalalt, i.e.
  - 1, 2, 9, 4, 12, 14, 15, 19, 24, 30
  - 6, 3, 7, 13, 8, 17, 18, 23, 21

- Aceste elemente ocupa celulele colorate la fel in reprezentarea de mai sus

Shell sort – implementare (11)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	6	2	3	9	7	4	13	12	8	14	17	15	18	19	23	24	21	30

- Elementele tabloului vor fi copiate intr’o matrice avand 2 coloane si vor fi sortate pe coloane (sortarea coloanelor foloseste tehnica **InsertSort**):

1	6
2	3
9	7
4	13
12	8
14	17
15	18
19	23
24	21
30	

$\Rightarrow$

1	3
2	6
4	7
9	8
12	13
14	17
15	18
19	21
24	23
30	

Shell sort – implementare (12)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A:	1	3	2	6	4	7	9	8	12	13	14	17	15	18	19	21	24	23	30

- La pasul 4, vom sorta folosind InsertSort elementele aflate la distanta H[4]=1 unul de celalalt
- Cu alte cuvinte, vom aplica un InsertSort obisnuit pe intreg tabloul A
- Faptul ca ultimul element al lui H este 1 garanteaza ca tabloul A sfarseste prin a fi sortat
- Se observa ca pasii anteriori au adus tabloul A la o forma aproape ordonata, deci ultimul InsertSort va reusi sa sorteze tabloul foarte rapid, chiar daca este o metoda putin performanta, fiind de clasa **O(N^2)**

Shell sort – concluzii

- Nu se recomanda alegerea puterilor lui 2 pe post de incrementi, deoarece aceasta ar insemna ca elementele de la indici pari nu vor fi sortate cu elementele de la indici impari decat in cadrul ultimei treceri
- Algoritmul **ShellSort** este cel mai rapid algoritm de clasa **O(N^2)**
- Totusi, nu se poate compara cu algoritmii de sortare super-performanti (**QuickSort** sau **HeapSort**)

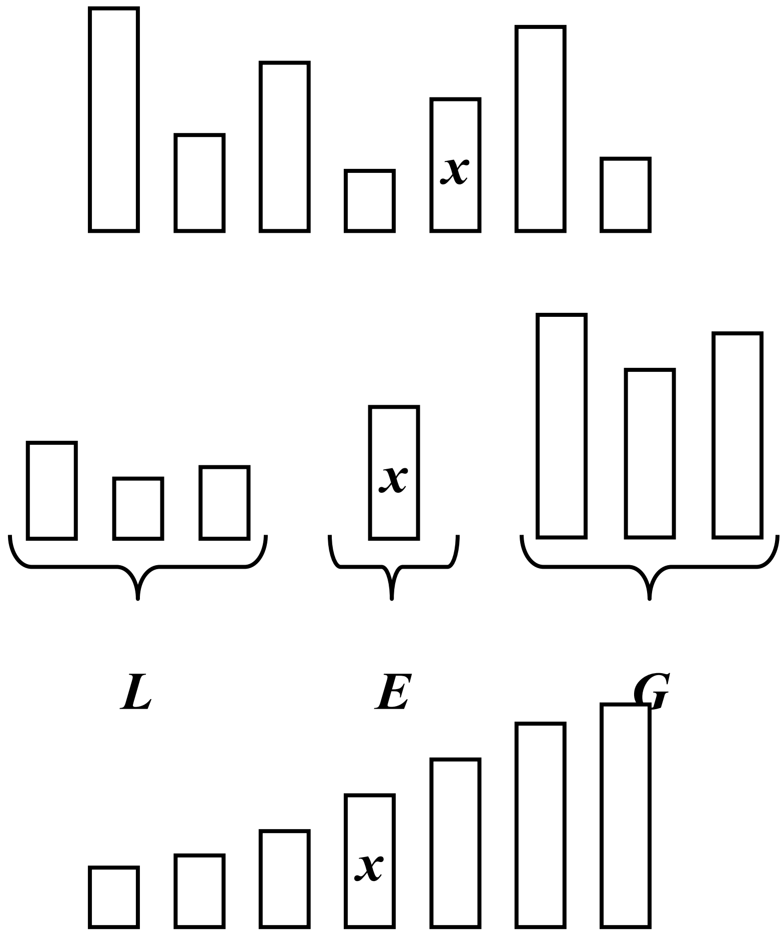
Shell sort – simulare

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

Sursa: Lai – Jay, Shell sort (CS141)

Quick sort

- **Quick-sort** este un algoritm (aleator) de sortare bazat pe paradigma “divide & impera”:
  - **Divide**: se alege in mod aleator un element x (numit *pivot*) si o partitie S si se imparte in:
    - L elemente mai mici decat x
    - E elemente egale cu x
    - G elemente mai mari decat x
  - **Recurent**: sorteaza L si G
  - **Impera**: join L, E si G



Quick sort

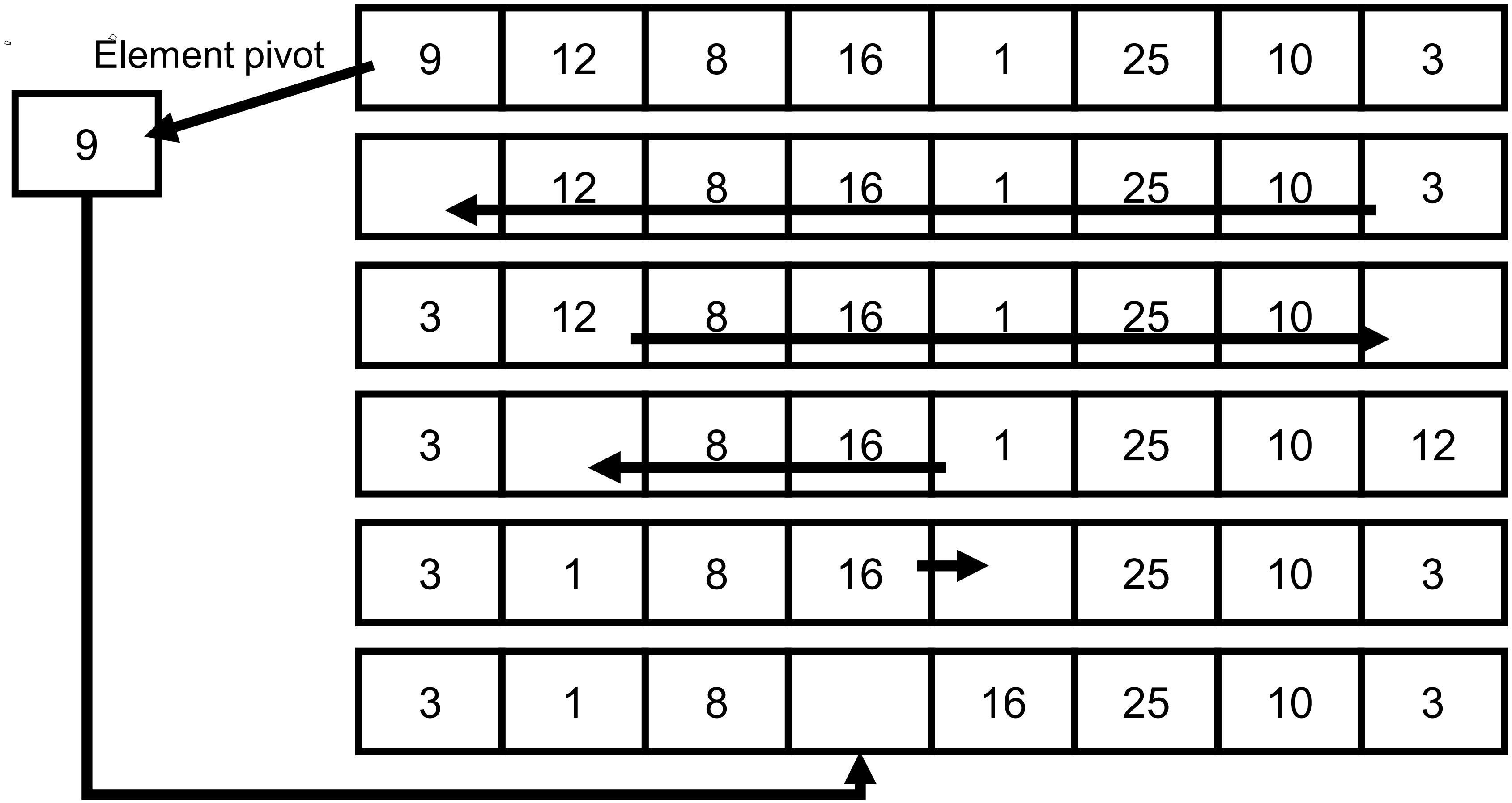
- **Quick sort**, (intalnit si sub numele de **partition sort**), lucreaza pe baza unei strategii **divide & impera** (**divide-and-conquer**).
- Algoritm:
  - Se alege un element pivot din vectorul de intrare;
  - Toate celelalte elemente ale vectorului de intrare se aseaza in felul urmator: elementele mai mici decat pivotul se aseaza in fata acestuia, iar cele mai mari decat acesta se aseaza in urma lui. Elementele egale cu pivotul se aseaza in jurul acestuia;
  - In mod recursiv, celelalte elemente ale vectorului de intrare se sorteaza in acelasi fel;
  - The recursion terminates when a list contains zero or one element.
- **Complexitate: O($n$log$n$) sau O(n^2)**
- Exemplu: fie lista {25, 57, 48, 37, 12}

Partitii

- Realizarea partitiilor se face in felul urmator:
  - Se elimina (in ordine) fiecare element y din S, si
  - Se insereaza y in L, E sau G, in functie de rezultatul compararii lui y cu pivotul x
- Fiecare inserare / eliminare se face la inceputul / sfarsitul vectorului si are o durata $O(1)$
- Prin urmare, procesul de partitionare a algoritmului **quick sort** dureaza $O(n)$

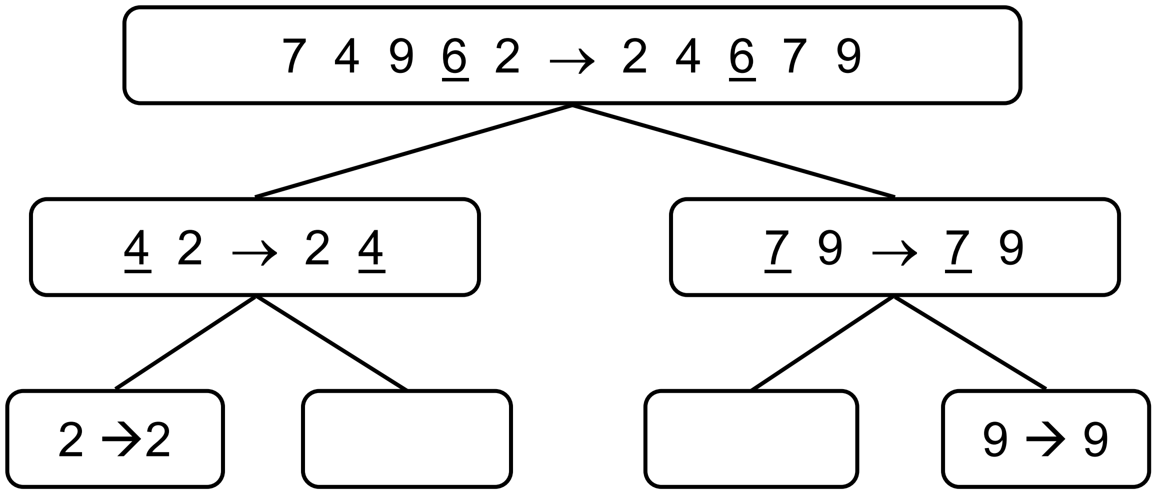
Algorithm ***partition***(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.insertLast(y)$
    else if $y = x$
      $E.insertLast(y)$
    else $\{ y > x \}$
      $G.insertLast(y)$
  return L, E, G

Quick sort **– exemplu**



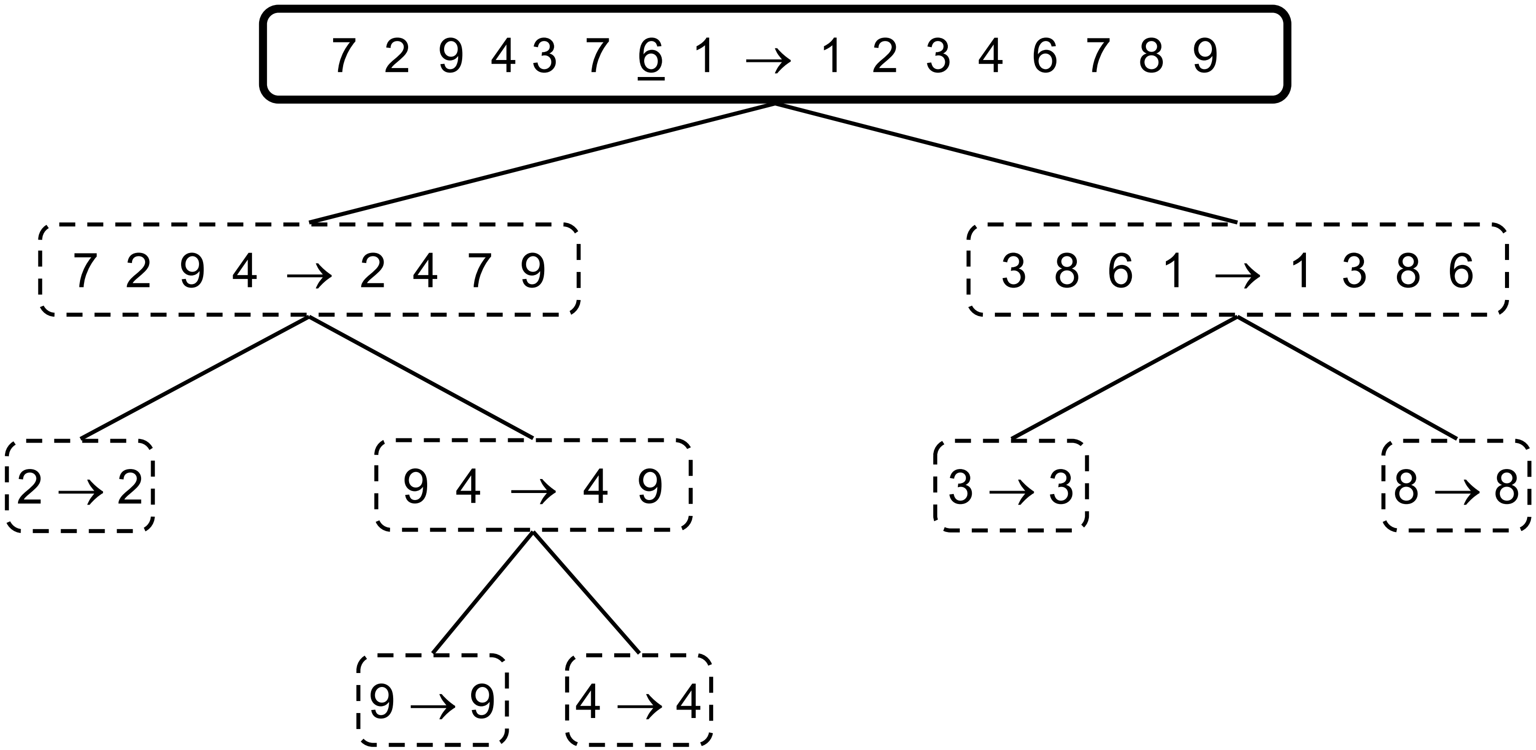
Arborele *Quick sort*

- Executia algoritmului **quick-sort** poate fi reprezentata prin intermediul unui arbore binar:
  - Fiecare nod reprezinta un apel recursiv al lui **quick-sort** si permite stocarea:
    - Secventa nesortata inainte de cea sortata si pivotul atasat
    - Secventa sortata la sfarsit
  - Radacina reprezinta apelul initial
  - Frunzele reprezinta apeluri sau sub-secvente de marime 0 sau 1



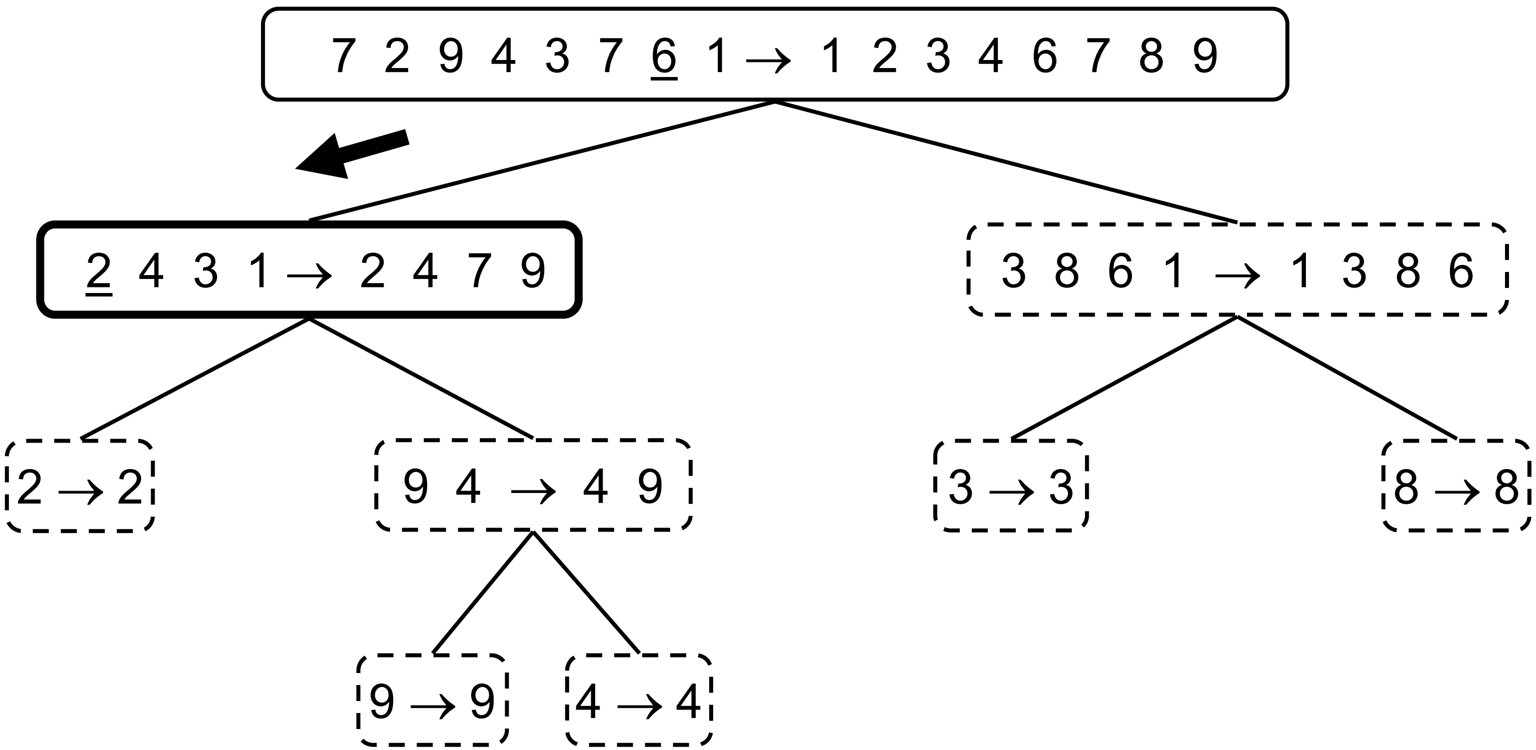
Quick sort – exemplu (1)

- Selectie pivot



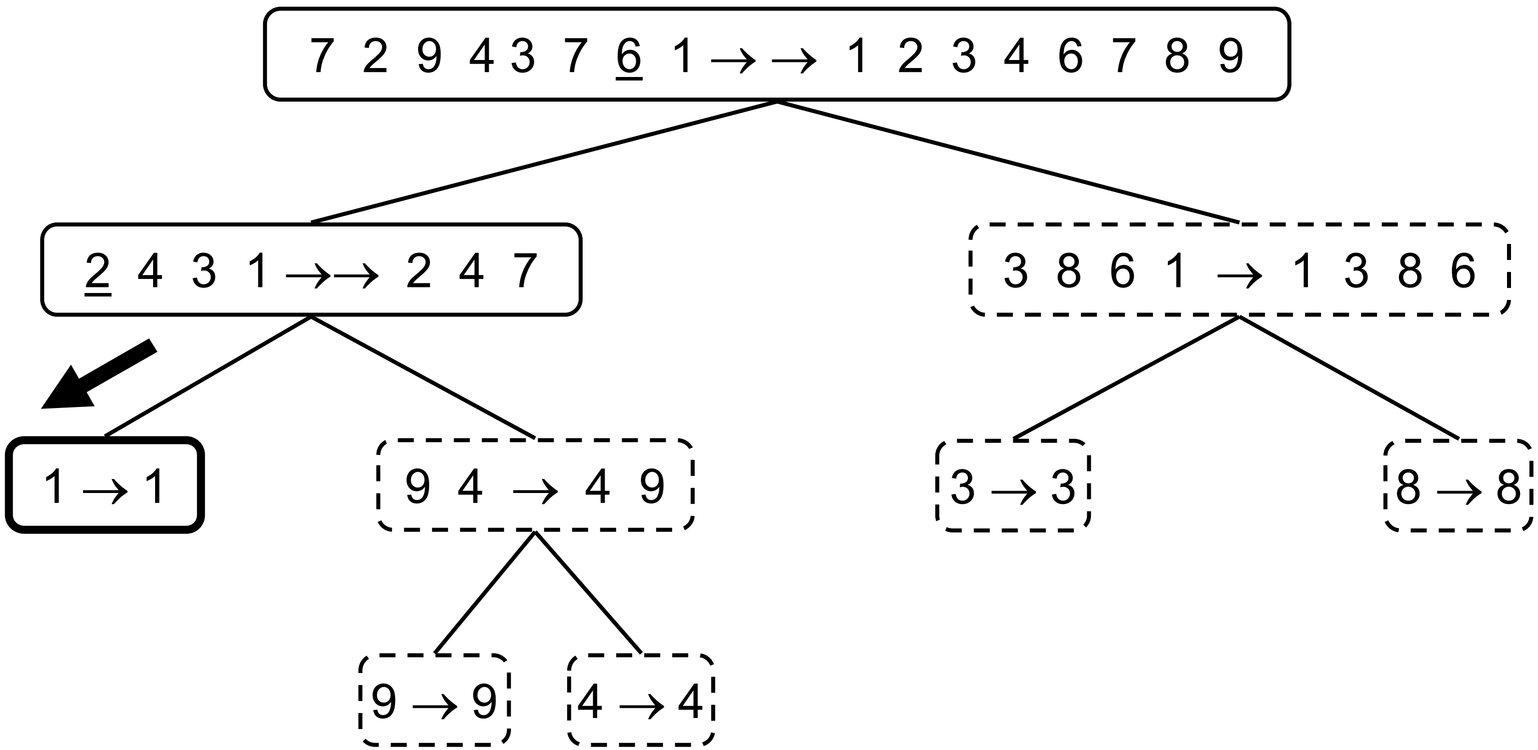
Quick sort – exemplu (2)

- Partitionare, apel recursiv, selectie pivot



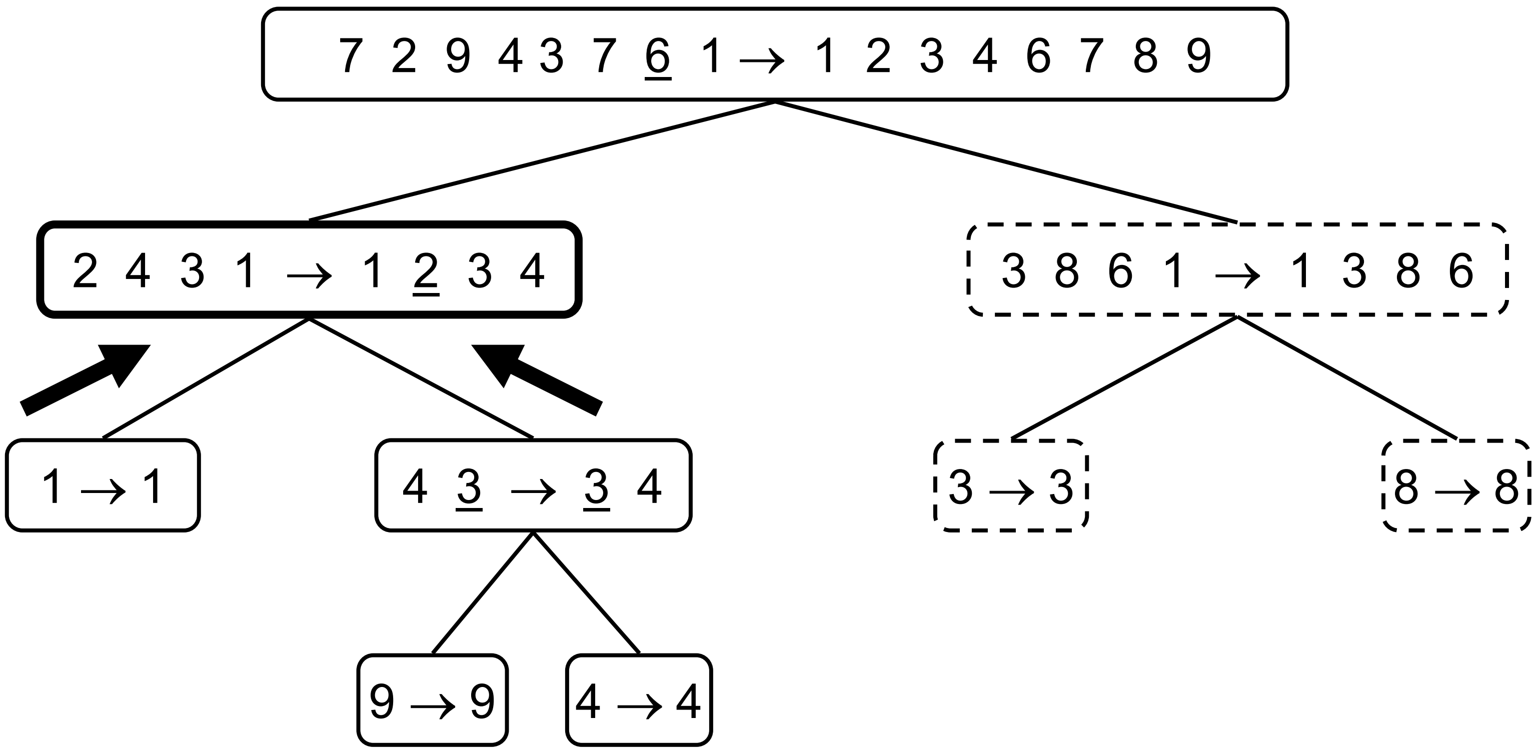
Quick sort – exemplu (3)

- Partitionare, apel recursiv, *base case*



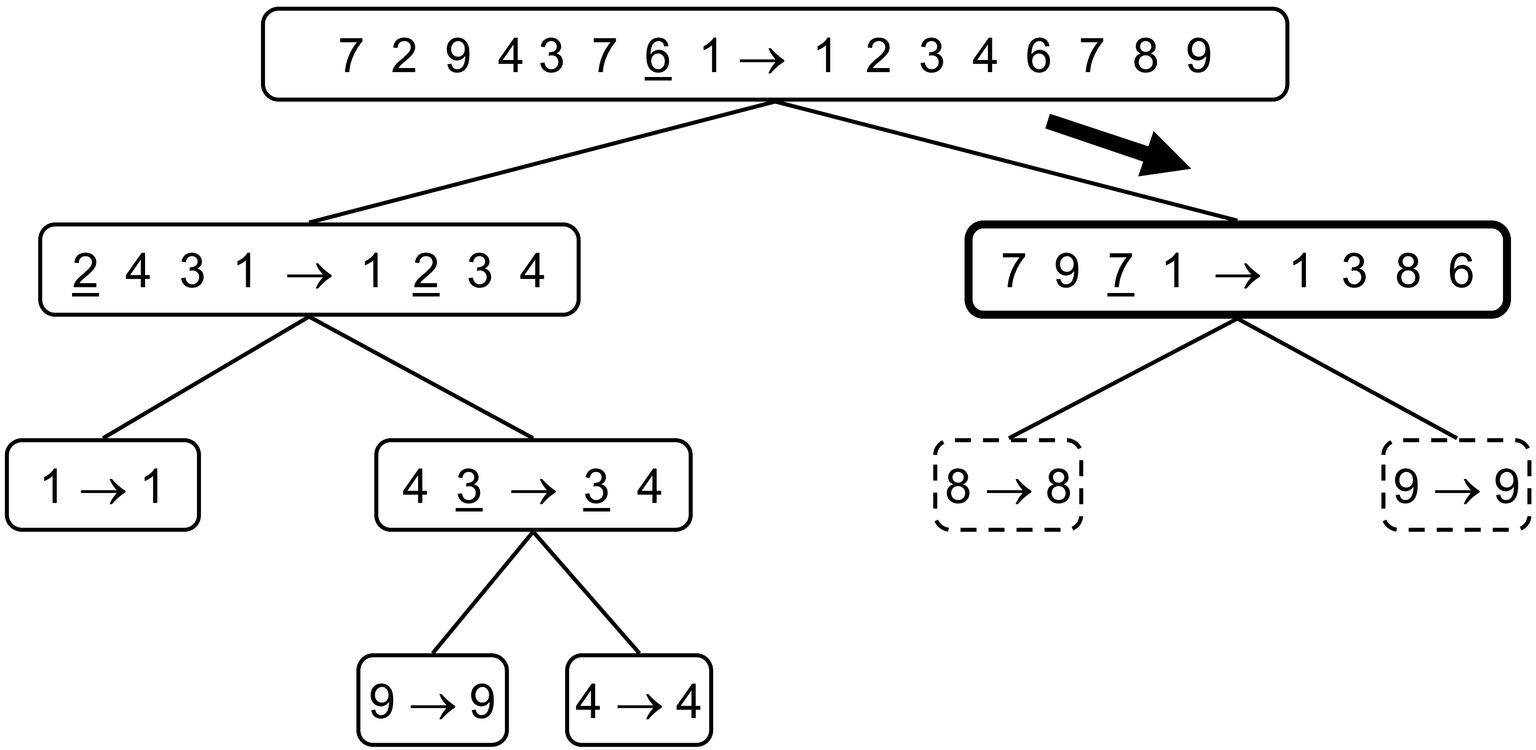
Quick sort – exemplu (4)

- Apel recursiv, ..., *base case*, *join*



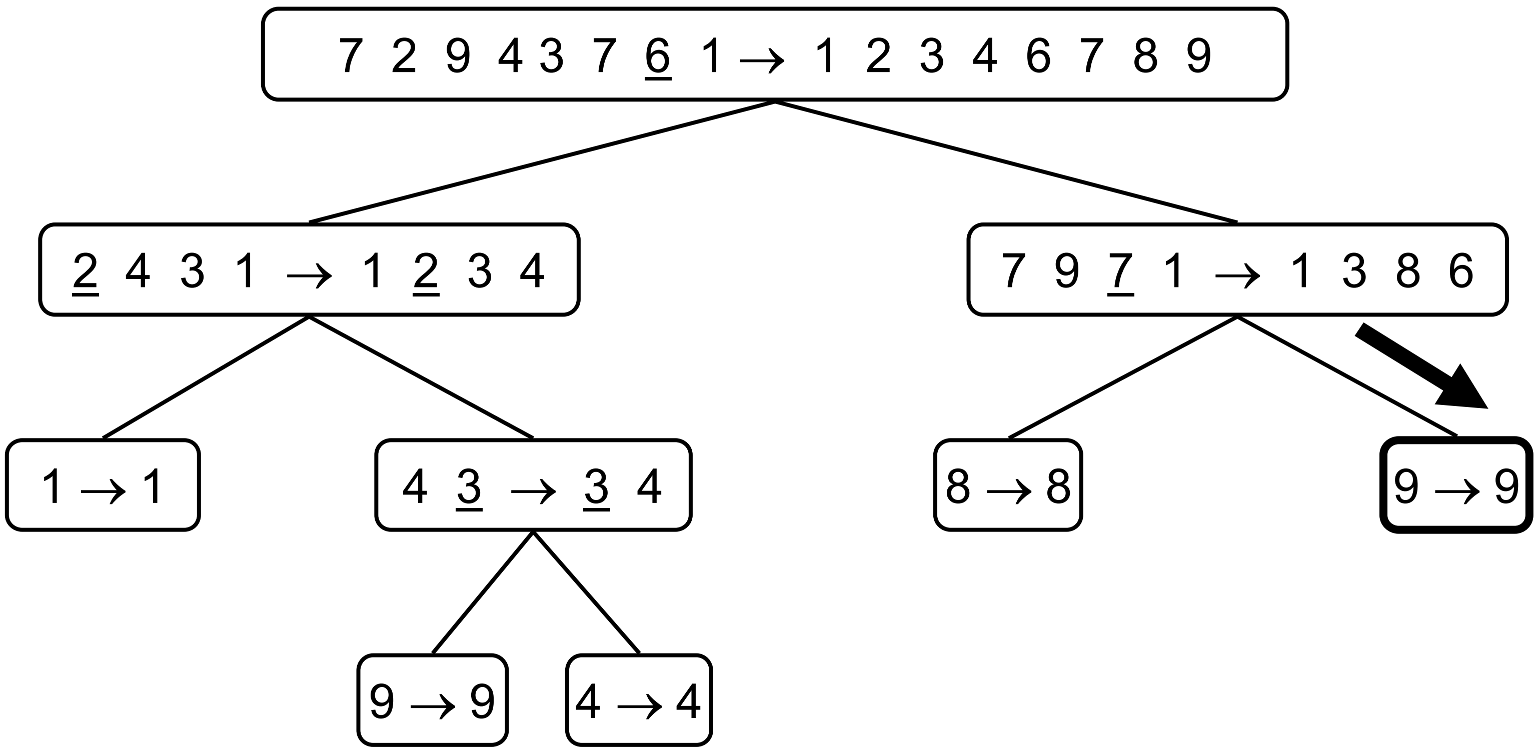
Quick sort – exemplu (5)

- Apel recursiv, selectie pivot



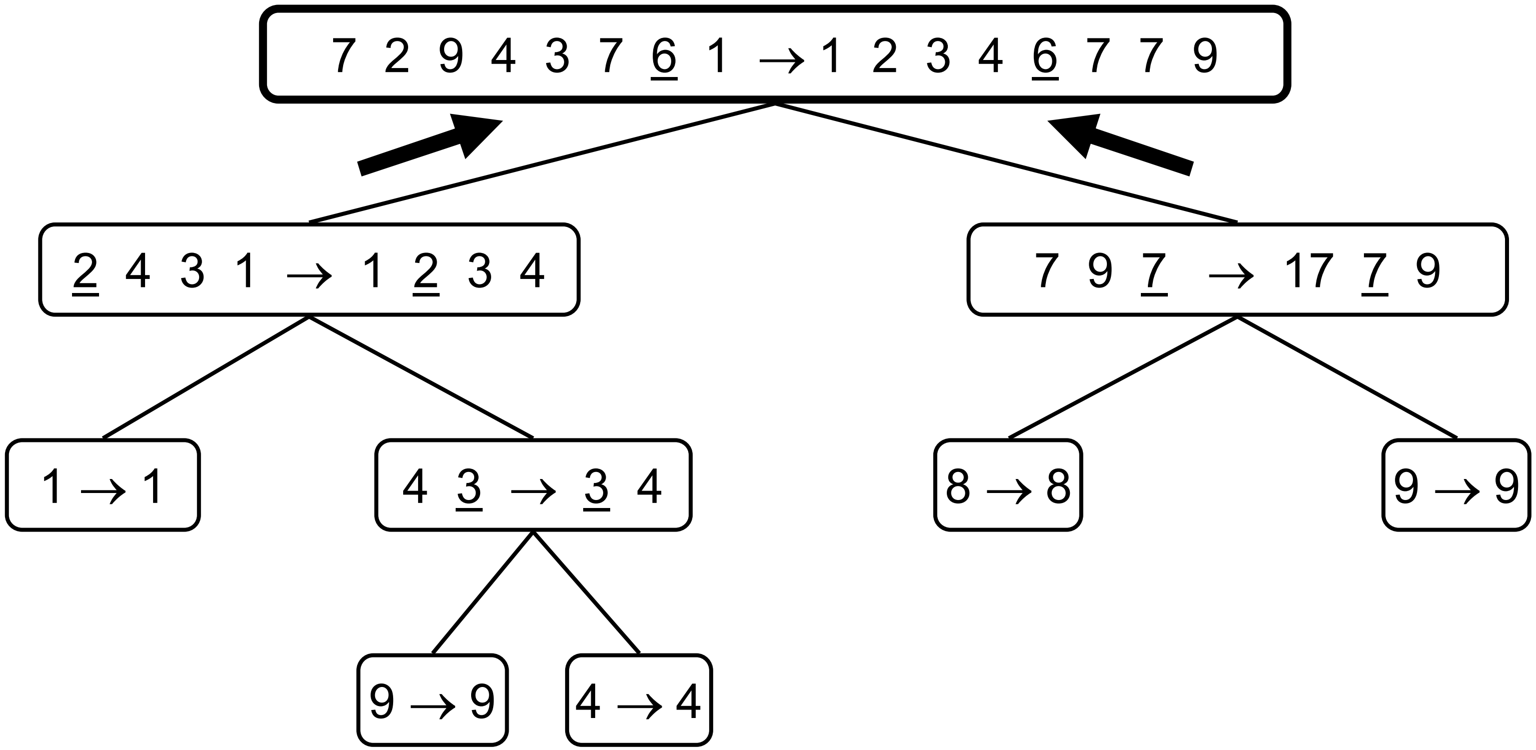
Quick sort – exemplu (6)

- Partitionare, …, apel recursiv, *base case*



Quick sort – exemplu (7)

- *Join, join*



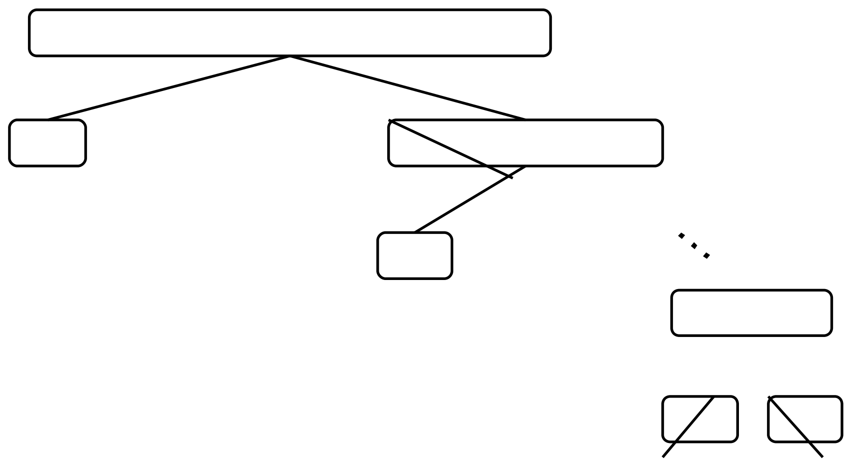
Quick sort – cazul nefericit

- *Cazul nefericit* pentru algoritmul **quick-sort** apare atunci cand pivotul reprezinta elementul (unic) minim / maxim
- Timpul de rulare este proportional cu suma:

$$\boldsymbol{n} + (\boldsymbol{n} - 1) + \ldots + 2 + 1$$

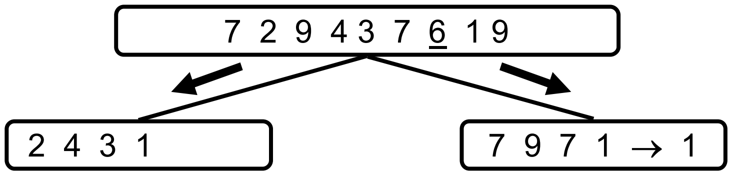
- Prin urmare, pentru *cazul nefericit*, timpul de rulare este proportional cu $\boldsymbol{O}(\boldsymbol{n}^2)$

nivel	timp
0	$\boldsymbol{n}$
1	$\boldsymbol{n} - 1$
…	…
$\boldsymbol{n} - 1$	1

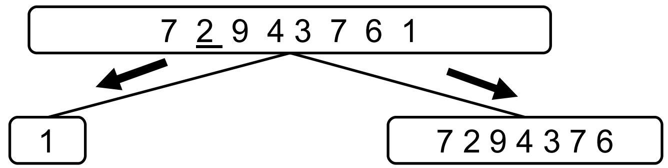


Quick sort – cazuri diferite

- Fie un apel recursiv al lui **quick-sort** pe o secventa de marime s
  - **Cazul fericit (apel bun):** marimea lui L si G este fiecare < $3s/4$
  - **Cazul nefericit (apel rau):** fie L, fie G are marimea > $3s/4$

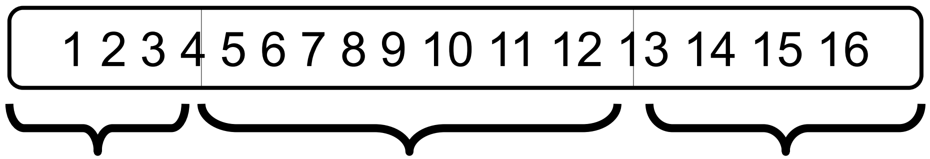


Cazul fericit



Cazul nefericit

- Un apel recusiv este bun cu probabilitatea $1/2$
  - 1/2 din pivotii posibili genereaza cazurile fericite :



Pivoti rai **Pivoti buni** **Pivoti rai**

Quick sort – implementare *in situ*

- **Quick-sort** poate fi implementat pentru rulare *in situ*
- In faza de partitionare, se utilizeaza operatii de rearanjare pentru elementele din secventa de intrare:
  - Elementele mai mici decat pivotul vor avea rangul < h
  - Elementele egale cu pivotul vor avea rangul intre h si k
  - Elementele mai mari decat pivotul vor avea > k
- Apelurile recursive considera:
  - Elementele de rang < h
  - Elementele de rang > k

Algorithm ***inPlaceQuickSort***(***S, l, r***)
    Input sequence ***S***, ranks ***l*** and ***r***
    Output sequence ***S*** with the elements of rank between ***l*** and ***r*** rearranged in increasing order
    if $l \geq r$
        return
    $i \leftarrow$ a random integer between ***l*** and ***r***
    $x \leftarrow$ ***S.elemAtRank***(***i***)
    (***h, k***) $\leftarrow$ ***inPlacePartition***(***x***)
    inPlaceQuickSort(***S, l, h*** – 1)
    inPlaceQuickSort(***S, k*** + 1***, r***)

Partitionare – *in situ*

- Partitionarea se poate face prin utilizarea a doi indici pentru splitarea lui S in L si E U G (similar pentru splitarea lui E U G in E si G).

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- *Repeat until j and k cross*:
  - *Scan j to the right until finding an element* $\geq$ *x.*
  - *Scan k to the left until finding an element < x.*
  - *Swap elements at indices j and k*

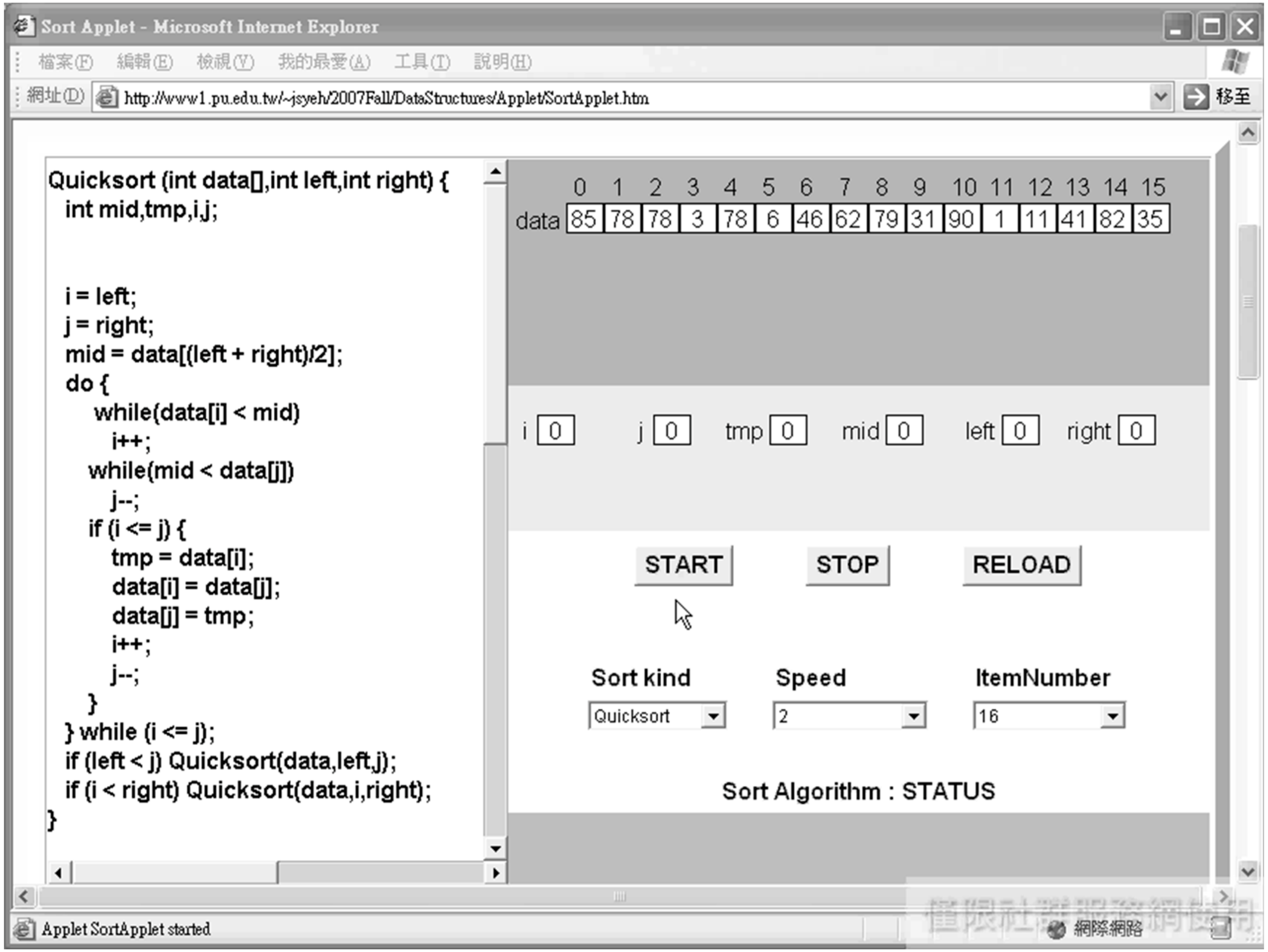
j k

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

Quick sort – exemplu

Sursa: Wikipedia.org

Quick sort – simulare



Sursa: http://www1.pu.edu.tw

Recapitulare

Algoritm	Timp	Observatii
selection-sort	$O(n^2)$	◆ *in-place* ◆ lent (bun pentru intrari de mici dimensiuni)
insertion-sort	$O(n^2)$	◆ *in-place* ◆ lent (bun pentru intrari de mici dimensiuni)
quick-sort	$O(n \log n)$	◆ *in-place*, aleator ◆ cel mai rapid (excelent pentru intrari de mari dimensiuni)
heap-sort	$O(n \log n)$	◆ *in-place* ◆ rapid (bun pentru intrari de mari dimensiuni)
merge-sort	$O(n \log n)$	◆ acces secvential al datelor ◆ rapid (bun pentru intrari de foarte mari dimensiuni)