

# Structuri de Date și Algoritmi

## Algoritmi de sortare

Mihai Nan

Departamentul de Calculatoare  
Facultatea de Automatică și Calculatoare  
Universitatea POLITEHNICA din București



Anul Universitar 2022–2023

# Conținutul cursului

- ① Metode de sortare – Generalități
- ② Sortarea prin selecție
- ③ Sortarea prin inserție
- ④ Sortarea prin metoda bulelor
- ⑤ Sortare indirectă
- ⑥ Algoritmul ShellSort
- ⑦ Algoritmul QuickSort
- ⑧ Sortare fără comparare
  - Sortarea prin numărare
  - Algoritmul Radix Sort

# Metode de sortare

- **Metode elementare** – potrivite pentru un număr mic de elemente (tipic  $< 500 - 1000$ )

De obicei, metodele elementare au complexitatea  $O(N^2)$

- **Metode avansate de sortare** — potrivite pentru fișiere mari

Complexitatea pentru metodele avansate de sortare este  $O(N \log N)$ .

## Criterii de performanță pentru metodele de sortare

- ① Timpul de rulare
- ② Cantitatea de memorie suplimentară

## Două categorii importante

- ① **Sortare internă**
- ② **Sortare externă**

# Metode de sortare

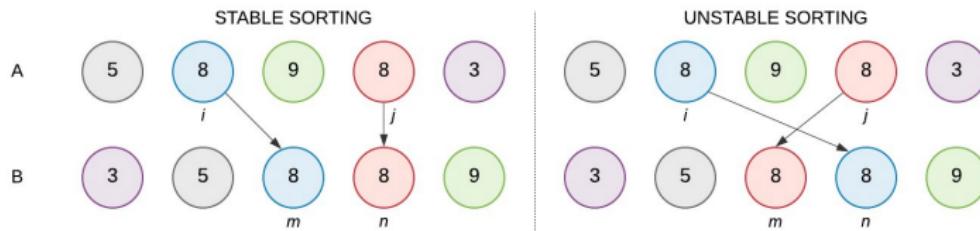
## Metodă de sortare stabilă

O astfel de metodă menține ordinea relativă a înregistrărilor cu chei egale. Dacă stabilitatea este importantă, se poate crea o nouă cheie – cheie extinsă.

## Sortare indirectă

Dacă înregistrările sunt mari este bine să nu le schimbăm poziția.

- Mulți algoritmi folosesc o santinelă (celulă suplimentară).



# Sortarea prin Selectie

## Pașii aplicați de algoritm

Pornim de la vectorul  $a[0], a[1], \dots, a[N-1]$  pe care dorim să îl sortăm crescător.

- ① Găsește cel mai mic element și îl schimbă cu cel de pe prima poziție.
- ② Pentru fiecare  $i = 1, N-1$  schimbă  $a[i]$  cu elementul minim din  $a[i], a[i+1] \dots a[N-1]$ .

## Observație

Deoarece fiecare element este mutat cel mult o dată, este o metodă bună pentru a sorta secvențe cu înregistrări mari și număr mic de chei.

# Sortarea prin Selectie – Exemplu

0	1	2	3	4	5	6	
42	16	84	12	77	26	53	– vectorul initial
12	16	84	42	77	26	53	– pasul 1
12	16	84	42	77	26	53	– pasul 2
12	16	26	42	77	84	53	– pasul 3
12	16	26	42	77	84	53	– pasul 4
12	16	26	42	53	84	77	– pasul 5
12	16	26	42	53	77	84	– pasul 6
12	16	26	42	53	77	84	– vectorul sortat

# Sortarea prin Selectie – Implementare

```
void selection(int a[], int N) {  
    int i, j, min, t;  
    for (i = 0; i < N-1; i++) {  
        min = i;  
        for (j = i + 1; j < N; j++) {  
            if (a[j] < a[min])  
                min = j;  
        }  
        t = a[min];  
        a[min] = a[i];  
        a[i] = t;  
    }  
}
```

**Complexitate:** ??

# Sortarea prin Selectie – Implementare

```
void selection(int a[], int N) {  
    int i, j, min, t;  
    for (i = 0; i < N-1; i++) {  
        min = i;  
        for (j = i + 1; j < N; j++) {  
            if (a[j] < a[min])  
                min = j;  
        }  
        t = a[min];  
        a[min] = a[i];  
        a[i] = t;  
    }  
}
```

**Complexitate:**  $O(N^2)$

# Sortare prin Insertie

## Ideea algoritmului

Consideră fiecare element pe rând și inserează elementul la locul potrivit printre cele deja sortate.

Pornim de la vectorul  $a[0], a[1], \dots, a[N-1]$  pe care dorim să îl sortăm crescător.

- ① Considerăm primul element deja inserat.
- ② Pentru fiecare  $i = 1, N-1$ , elementele  $a[0], \dots, a[i]$  vor deveni sortate prin plasarea pe poziția corectă a lui  $a[i]$  (îl vom insera printre elementele deja sortate  $a[0], \dots, a[i-1]$ ).

# Sortare prin Insertie – Implementare

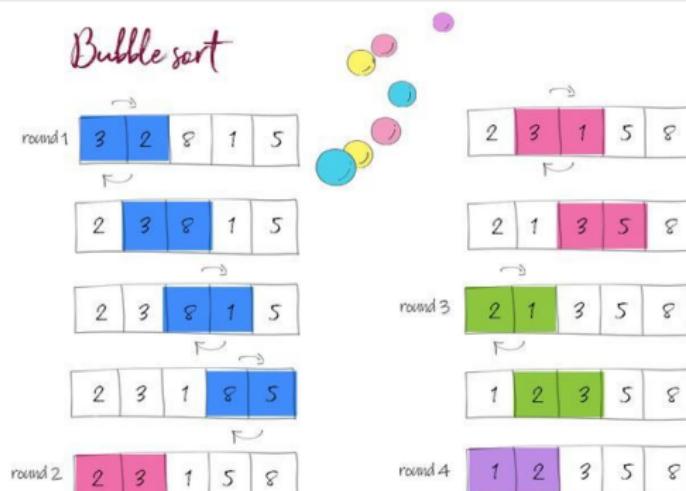
```
// Indexarea de la 1 in vector
a[0] = INT_MIN;
void insertion(int a[], int N) {
    int i, j, v;
    for (i = 2; i <= N; i++) {
        v = a[i];
        j = i;
        while (a[j-1] > v) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
    }
}
```

# Sortarea prin metoda bulelor

## Ideea algoritmului

Parcurgem vectorul și pentru oricare două elemente învecinate care nu sunt în ordinea dorită, le interzicăm valorile. După o singură parcurgere, vectorul nu se va sorta, dar putem repeta parcurgerea.

**Când ne oprim?** Dacă la o parcurgere nu se face nicio interzicere, vectorul este sortat.



# Sortarea prin metoda bulelor – Implementare

```
// Considerăm indexarea de la 1 în vector
void bubble(int a[], int N) {
    int i, j, t;
    for (i = N; i >= 1; i--) {
        for (j = 2; j <= i; j++) {
            if (a[j-1] > a[j]) {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}
```

# Sortarea prin metoda bulelor – Varianta optimizată

```
void bubbleSortOptimized(int a[], int N) {  
    int i, j, t, ok;  
    for (i = 0; i < N - 1; i++) {  
        ok = 0;  
        for (j = 0; j < N - i - 1; j++) {  
            if (arr[j] > arr[j+1]) {  
                t = arr[j];  
                arr[j] = arr[j+1];  
                arr[j+1] = t;  
                ok = 1;  
            }  
        }  
        if (!ok)  
            break;  
    }  
}
```

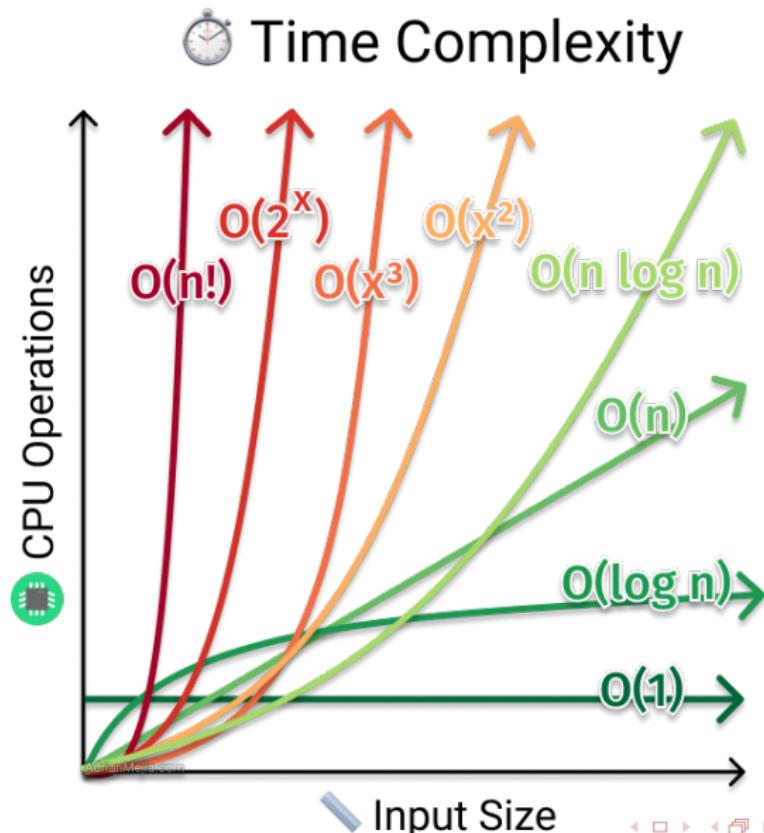
# Performanța metodelor elementare

- **Sortarea prin selecție** folosește aproximativ  $\frac{N^2}{2}$  comparații și  $N$  interschimbări.
- **Sortarea prin inserție** folosește aproximativ  $\frac{N^2}{4}$  comparații și  $\frac{N^2}{8}$  interschimbări în medie, și dublu în cazul cel mai nefavorabil.
- **Sortarea prin inserție** este liniară pentru fișiere *aproape sortate*.
- **Sortarea prin metoda bulelor** folosește aproximativ  $\frac{N^2}{2}$  comparații și  $\frac{N^2}{2}$  interschimbări pentru cazul mediu și pentru cazul cel mai nefavorabil.

## Important

Toate metodele elementare au complexitatea  $O(N^2)$  cazul cel mai nefavorabil.

# Complexitatea algoritmilor



# Comparație metodelor elementare

Algoritm	Timp			Spațiu
	Cel mai bun (Best)	Mediu (Average)	Cel mai rău (Worst)	Cel mai rău (Worst)
<b>Bubble Sort</b>	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
<b>Insertion Sort</b>	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
<b>Selection Sort</b>	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$

# Sortare indirectă

- Utilă când avem înregistrări de dimensiune mare
- Folosim un vector de indici  $p: p[0] \dots p[N-1]$
- Algoritmii trebuie modificați pentru a se referi la  $a[p[i]]$  în loc de  $a[i]$  când se compara  $a[i]$  și să se refere la  $p$  în loc de  $a$  când se face sortarea.
- După ce s-a realizat sortarea indirectă, dacă se dorește se pot rearanja înregistrările.
- Se poate utiliza și un vector de pointeri.

# Sortarea prin Insertie – varianta indirectă

```
// Considerăm indexarea în vector de la 1
a[0] = INT_MIN; // sentinelă
void insertion(int a[], int p[], int N) {
    int i, j, v;
    for (i = 1; i <= N; i++)
        p[i] = i;
    for (i = 2; i <= N; i++) {
        v = p[i];
        j = i;
        while (a[p[j-1]] > a[v]) {
            p[j] = p[j-1];
            j--;
        }
        p[j] = v;
    }
}
```

# Sortarea Shell

- Extindere a Insertion Sort — crește viteza permitând interschimbări între elemente neadiacente;
- Rearanjăm vectorul de sortat a.î. să aibă proprietatea:

Dacă se ia fiecare al  $h$ -lea element (începând de oriunde) obținem un vector sortat.

- Un astfel de vector se numește  $h$ -sortat iar  $h$  se numește în engleză gap.
- Un vector  $h$ -sortat poate fi văzut ca o colecție de  $h$  vectori independenți întrețesuți.
- Pentru fiecare  $h$  folosim Insertion Sort, independent pentru fiecare din cele  $h$  subșiruri.

## Observație

Nu putem folosi santinele în acest caz, deoarece ar trebui să avem  $h$  santinele.

# Sortarea Shell

- Există multe variante de alegere a valorilor  $h$
- De ex  $h = 1$  și  $h = h \cdot 3 + 1$
- Fie  $h_1 = 1, h_{s+1} = 3 \cdot h_s + 1$  și ne oprim cu  $h_t$  pentru care  $h_{t+2} \geq N$ .

## Exemplu

$$h_1 = 1 \tag{1}$$

$$h_2 = (3 \cdot 1) + 1 = 4 \tag{2}$$

$$h_3 = (3 \cdot 4) + 1 = 13 \tag{3}$$

$$h_4 = (3 \cdot 13) + 1 = 40 \tag{4}$$

$$h_5 = (3 \cdot 40) + 1 = 121 \tag{5}$$

(6)

Pentru  $N = 100$ , vom selecta  $h_3$ .

# Sortarea Shell

```
void shellSort(int a[], int N) {  
    int i, j, h, v;  
    for (h = 1; h <= N; h = 3 * h + 1);  
    h /= 3;  
    h /= 3;  
    for ( ; h > 0; h /= 3)  
        for (i = h + 1; i <= N; i++) {  
            v = a[i];  
            j = i;  
            while (j > h && a[j - h] > v) {  
                a[j] = a[j-h];  
                j -= h;  
            }  
            a[j] = v;  
        }  
}
```

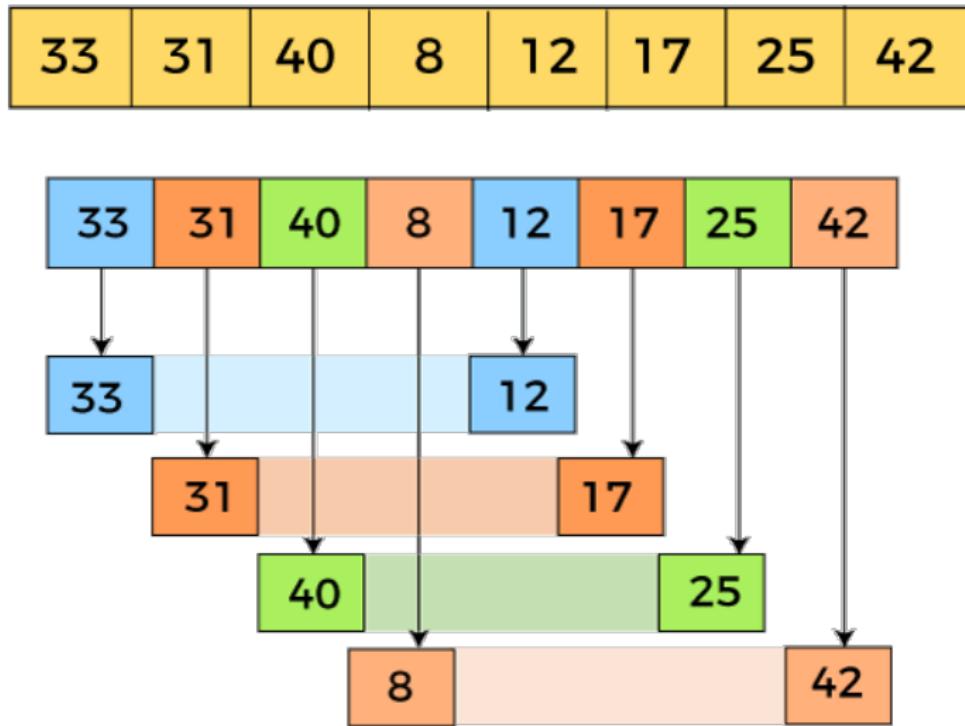
# Sortarea Shell – Exemplu (Pasul 1 – $h = 4$ )

0	1	2	3	4	5	6	7	8	9	10	11	12
45	36	75	20	5	90	80	65	30	50	10	75	85
20			45			50			80			85
	5			10			36			65		
		30			75			75			90	
20	5	30	45	10	75	50	36	75	80	65	90	85

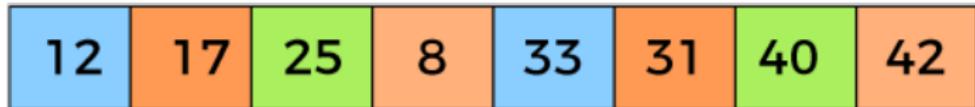
# Sortarea Shell

33	31	40	8	12	17	25	42
----	----	----	---	----	----	----	----

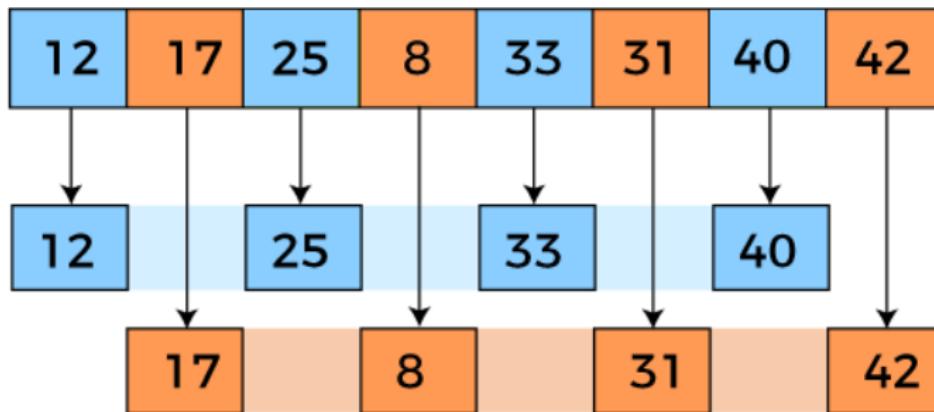
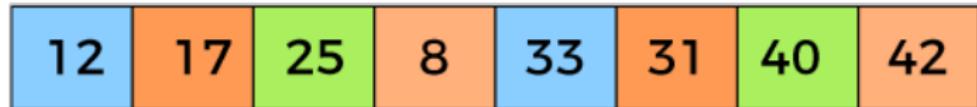
# Sortarea Shell



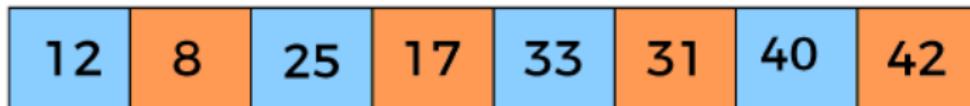
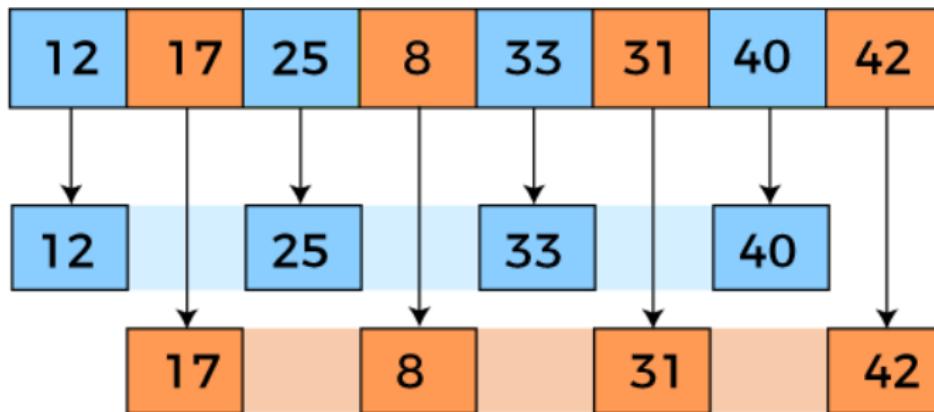
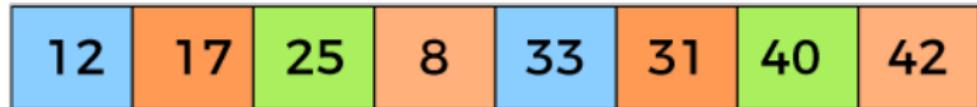
# Sortarea Shell



# Sortarea Shell



# Sortarea Shell



# Sortarea Shell

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

# Metode de sortare avansate

① Pentru vectori:

- Heapsort (discutat anterior)
- Quicksort
- Mergesort

② Pentru grafuri:

- Sortare topologică (discutat anterior)

## Time Complexities of Sorting Algorithms:

Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$



# Algoritmul Quicksort

- Probabil cel mai utilizat algoritm de sortare
- Descoperit în 1960 de C.A.R. Hoare
- Consumă resurse mai puține decât alte metode
- În medie, folosește  $N \log N$  operații pentru a sorta  $N$  elemente, dar  $N^2$  pentru cazul cel mai nefavorabil.



**Figure 1:** C.A.R. Hoare, englez (1934)

# Algoritmul Quicksort

Algoritmul are la bază tehnica de programare **Divide et Impera**.

## Pasul Divide

Dacă S (șirul de sortat) are cel puțin două elemente, selectează un element  $x$  din S – pivot (tipic ultimul). Împarte elementele din S în 3 părți:

- ① Elemente mai mici decât  $x$  (L)
- ② Elemente egale cu  $x$  (E)
- ③ Elemente mai mari ca  $x$  (G)

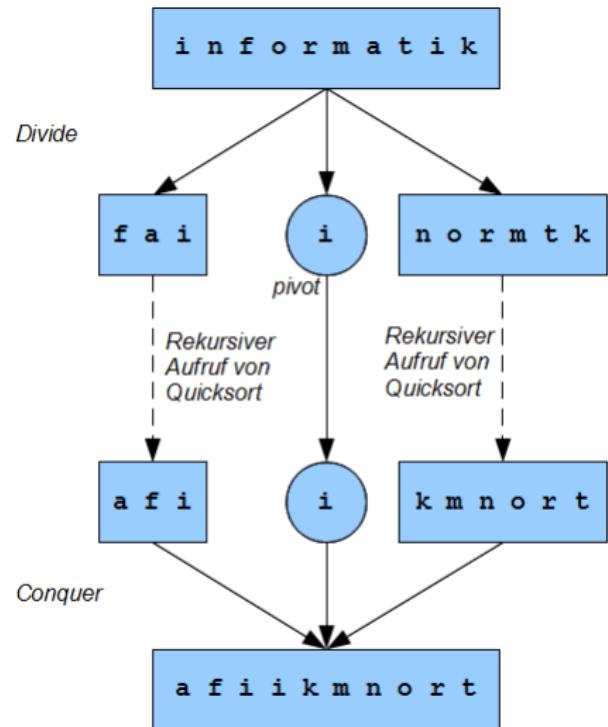
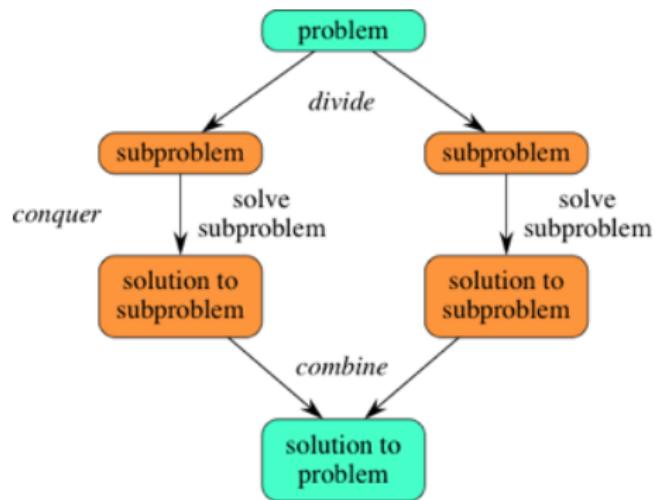
## Pasul Impera

Sortează L și G

## Pasul Combină

Reface S punând în S, în ordine: L, E și G

# Algoritmul Quicksort



# Algoritmul Quicksort – Pseudocod

QuickSort

```
1: procedure QUICK-SORT( $V, start, end$ )
2:   if  $start < end$  then
3:      $pivot \leftarrow partition(V, start, end)$ 
4:     quick_sort( $V, start, pivot - 1$ )
5:     quick_sort( $V, pivot + 1, end$ )
6:   end if
7: end procedure
```

Partition

```
1: procedure PARTITION( $V, start, end$ )
2:    $pivot \leftarrow start$ 
3:    $index \leftarrow start$ 
4:   for  $i \leftarrow start + 1$  to  $end$  do
5:     if  $V[i] < V[pivot]$  then
6:        $index \leftarrow index + 1$ 
7:       swap( $V[i], V[index]$ )
8:     end if
9:   end for
10:  swap( $V[start], V[index]$ )
11:  return  $index$ 
12: end procedure
```

# Algoritmul Quick Sort – Exemplu



**Pas 1:** Alegere pivot



**Pas 2:** Valorile mai mici în stânga pivotului, cele mai mari sau egale în dreapta



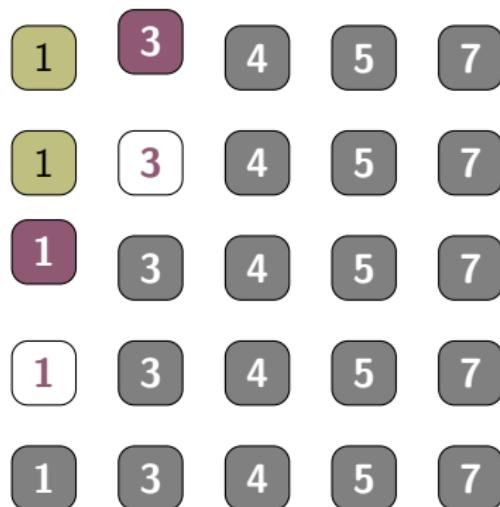
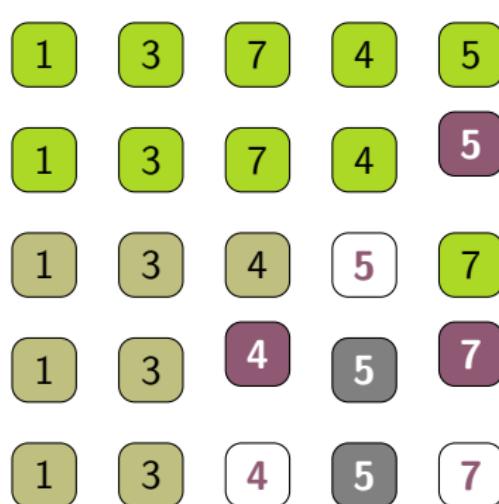
**Pas 3:** Repetă pasul 1 pentru cele două subintervale



**Pas 4:** Repetă pasul 2 pentru cele două subintervale



# Algoritmul Quick Sort – Exemplu



# Algoritmul Quicksort – Implementare

```
void swap(int* a, int* b) {  
    int t = *a;  
    *a = *b;  
    *b = t;  
}  
  
int partition (int arr[], int low, int high) {  
    int pivot = arr[high]; // pivot  
    int i = (low - 1);  
    for (int j = low; j <= high - 1; j++) {  
        if (arr[j] < pivot) {  
            i++;  
            swap(&arr[i], &arr[j]);  
        }  
    }  
    swap(&arr[i + 1], &arr[high]);  
    return (i + 1);  
}
```

# Algoritmul Quicksort – Implementare

```
void quickSort(int arr[], int low, int high) {  
    if (low < high) {  
        int pivot = partition(arr, low, high);  
        quickSort(arr, low, pivot - 1);  
        quickSort(arr, pivot + 1, high);  
    }  
}  
  
int main() {  
    int arr[] = {10, 7, 8, 9, 1, 5};  
    int n = sizeof(arr) / sizeof(arr[0]);  
    quickSort(arr, 0, n - 1);  
    printArray(arr, n);  
    return 0;  
}
```

# Algoritmul Quicksort – Caracteristici

- **Limitare** — ineficient dacă secvența este deja sortată;
- Folosește aproximativ  $2N \log N$  comparații pentru cazul mediu;
- $O(N^2)$  cazul cel mai defavorabil;
- $O(N \log N)$  cazul mediu.

Folosește spațiu suplimentar în stivă!

Pentru a scăpa de limitarea anterioară, putem să transformăm implementarea într-o variantă iterativă pentru a nu mai folosi spațiu suplimentar în stivă!

# Algoritmul Quicksort – Iterativ

```
void Quicksort(int a[], int N) {  
    int i, l = 0, r = N-1;  
    Stack s = initStack();  
    for( ; ; ) {  
        while (r > l) {  
            i = partition(a,l,r);  
            if (i-l > r-i) {  
                push(s, l); push(s, i-1); l = i + 1;  
            } else {  
                push(s, i+1); push(s, r); r = i - 1;  
            }  
        }  
        if (isEmptyStack(s)) break;  
        r = pop(s); l = pop(s);  
    }  
}
```

# Algoritmul Mergesort

- Folosește tehnica de programare Divide et Impera
  - Poate fi implementat a.î. să acceseze elementele într-o manieră secvențială
  - Avantaje: potrivit pentru sortarea secvențelor reprezentate prin liste înlăntuite
  - Potrivit ca bază pentru extindere la sortări externe
- ① Two-way merging
- ② Multi-way merging

# Algoritmul Mergesort

Algoritmul are la bază tehnica **Divide et Impera**

## Pasul Divide

Dacă S are 0 sau 1 element, întoarce soluția. Altfel (S are cel puțin 2 elemente) împarte S în 2 părți disjuncte aproximativ egale:

- ① S<sub>1</sub> cu primele N/2 elemente
- ② S<sub>2</sub> cu ultimele N/2 elemente

## Pasul Impera

Sortează S<sub>1</sub> și S<sub>2</sub>

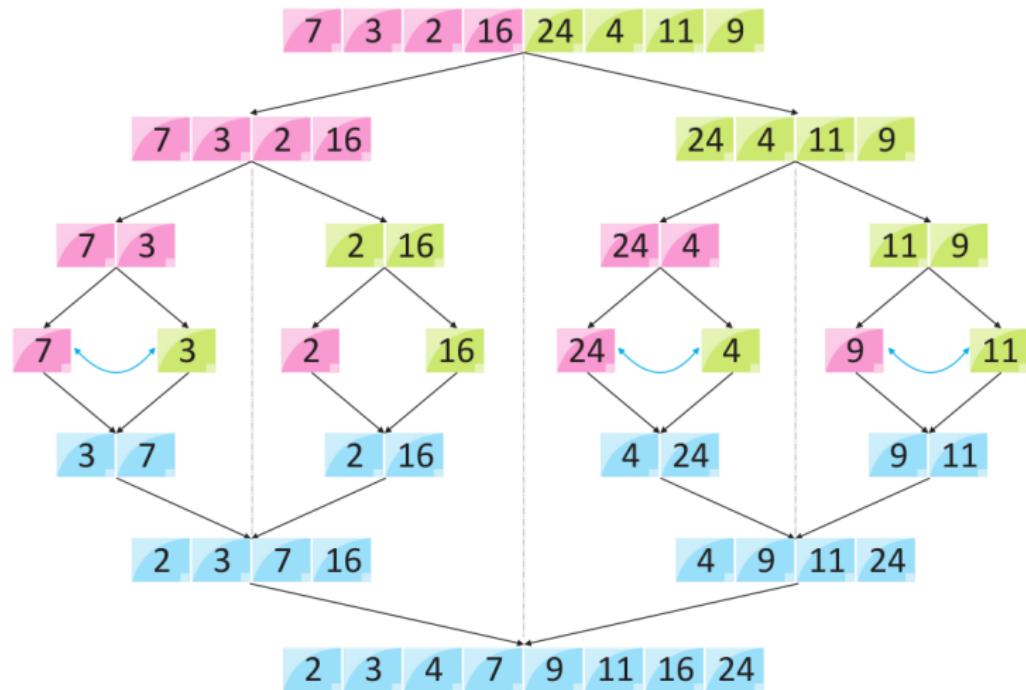
## Pasul Combină

Reface S prin combinarea (îmbinarea) lui S<sub>1</sub> și S<sub>2</sub> într-o secvență sortată = merge

# Algoritmul Mergesort – Implementare

```
/* nu are nevoie de santinela */
void mergesort(int a[], int l, int r) {
    int i,j,k,m;
    if (r > l) {
        m = (r+l)/2;
        mergesort(a,l,m);
        mergesort(a,m+1,r);
        for(i = m+1; i>l; i--) /* 1 */
            b[i-1] = a[i-1];
        for(j = m; j<r; j++) /* 2 */
            b[r+m-j] = a[j+1];
        for(k = l; k<=r; k++) /* 3 */
            a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
    }
}
```

# Algoritmul Mergesort – Exemplu



# Sortare fără comparare

- Metodele prezentate până acum sunt metode de sortare bazate pe compararea elementelor.
- Pentru timpul mediu și timpul cel mai defavorabil, nu putem obține o valoare a complexității timp sub  $O(N \log N)$ .
- Există algoritmi de sortare care nu presupun compararea elementelor și care au timp mediu și defavorabil proporțional cu  $N$  dar care se pot aplica numai pentru anumite categorii de date.
  - ① Sortarea prin numărare
  - ② Sortarea Radix

# Sortarea prin numărare

- Este un algoritm de sortare care se aplică dacă în vector cheile sunt într-un anumit interval.
- Funcționează prin numărarea numărului de obiecte care au valori de chei distincte (un fel de hashing).
- Apoi realizează calcule aritmetice pentru a calcula poziția fiecărui obiect din secvența de ieșire.

# Sortarea prin numărare

```
int getMax(int a[], int n) {  
    int max = a[0];  
    for(int i = 1; i < n; i++) {  
        if(a[i] > max)  
            max = a[i];  
    }  
    return max;  
}  
  
void printArr(int a[], int n) {  
    int i;  
    for (i = 0; i < n; i++)  
        printf("%d ", a[i]);  
    printf("\n");  
}
```

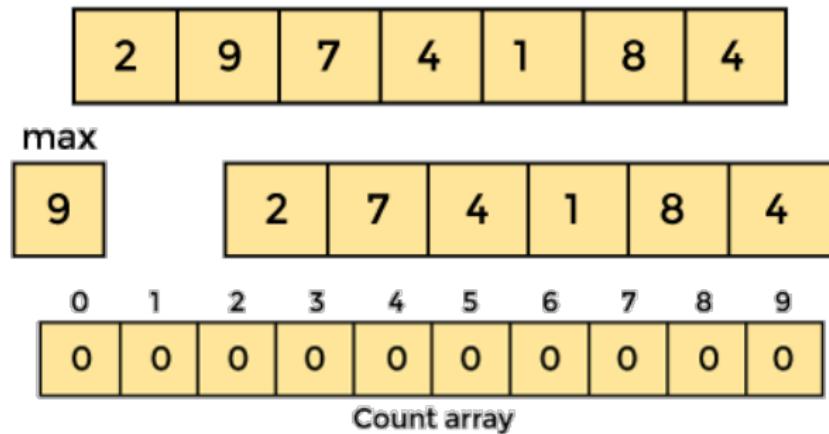
# Sortarea prin numărare

```
void countSort(int a[], int n) {  
    int output[n+1];  
    int max = getMax(a, n);  
    int count[max+1];  
    for (int i = 0; i <= max; ++i)  
        count[i] = 0;  
    for (int i = 0; i < n; i++)  
        count[a[i]]++;  
    for (int i = 1; i <= max; i++)  
        count[i] += count[i-1];  
    for (int i = n - 1; i >= 0; i--) {  
        output[count[a[i]] - 1] = a[i];  
        count[a[i]]--;  
    }  
    for (int i = 0; i < n; i++)  
        a[i] = output[i];
```

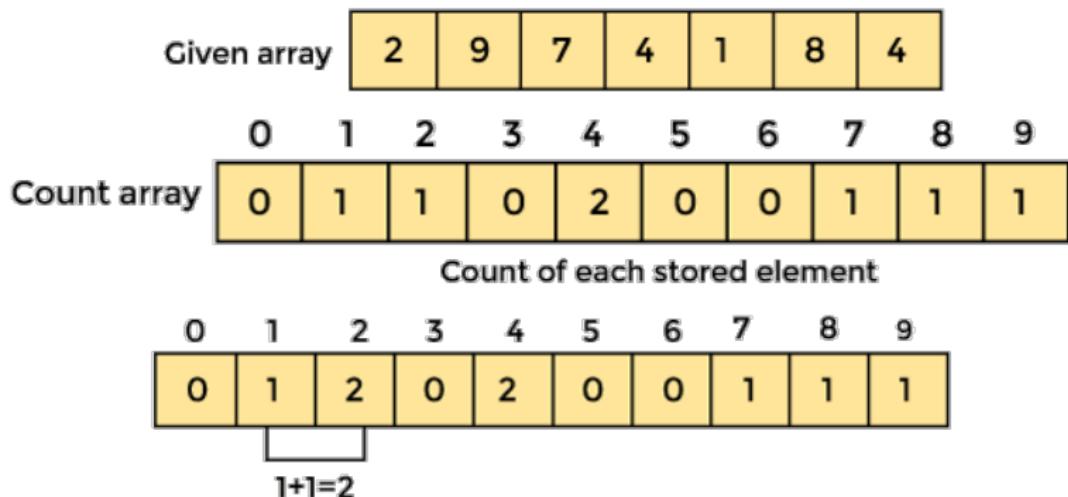
# Sortarea prin numărare

```
int main() {  
    int a[] = { 11, 30, 24, 7, 31, 16 };  
    int n = sizeof(a)/sizeof(a[0]);  
    printf("Before sorting array elements are - \n");  
    printArr(a, n);  
    countSort(a, n);  
    printf("\nAfter sorting array elements are - \n");  
    printArr(a, n);  
    return 0;  
}
```

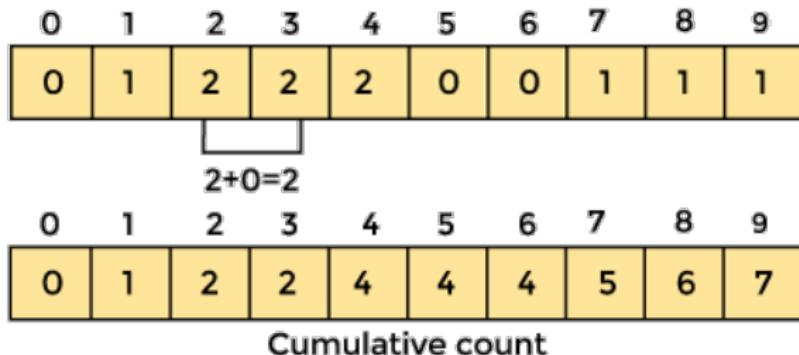
# Sortarea prin numărare



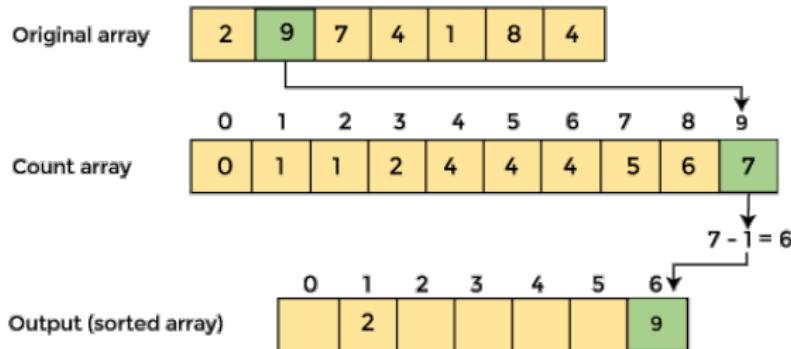
# Sortarea prin numărare



# Sortarea prin numărare

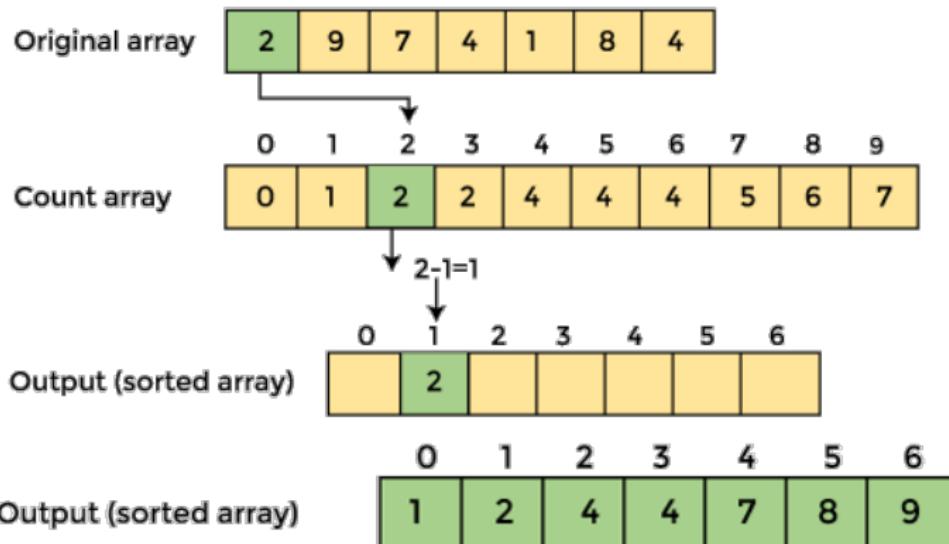


For element 9



# Sortarea prin numărare

For element 2



# Algoritmul Radix Sort

- Radix Sort este un algoritm de sortare care compara individual cifrele elementelor din vectorul care urmează să fie sortat.
- Cea mai întâlnită variantă pentru algoritmul Radix Sort este cea care folosește 10 cozi (câte una pentru fiecare cifră de la 0 la 9).
- Aceste cozi vor reține la fiecare pas numerele care au cifra corespunzătoare rangului curent.
- După ce este realizată această împărțire, elementele sunt preluate din cozi în ordinea crescătoare a indicelui cozii (pornind de la 0, 1, ..., 9) și sunt reintroduse în vector.

## Ideeă generală

La sfârșitul fiecarui pas aceste cozi pot fi concatenate usor în ordinea potrivita. De remarcat este faptul că, la fiecare pas  $k$  al algoritmului, numerele formate cu cifre de pana la pozitia  $k$  sunt deja sortate.

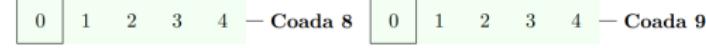
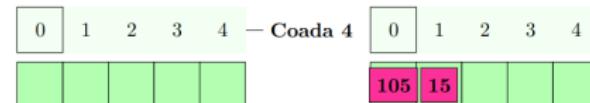
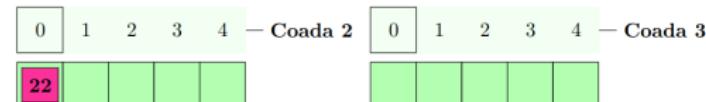
# Algoritm Radix Sort

```
1: procedure RADIXSORT(v)
2:   Queue v_queue[10]
3:   max  $\leftarrow$  maximumNumberOfDigits(v)
4:   size  $\leftarrow$  length(v)
5:   for i  $\leftarrow$  0; i  $<$  max; i  $\leftarrow$  i + 1 do
6:     for j  $\leftarrow$  0; j  $<$  size; j  $\leftarrow$  j + 1 do
7:       digit  $\leftarrow$  getDigit(v[j], i)
8:       if digit  $\geq$  0 and digit  $\leq$  9 then
9:         v_queue[digit]  $\leftarrow$  enqueue(v_queue[digit], v[j])
10:        end if
11:      end for
12:      k  $\leftarrow$  0
13:      for j  $\leftarrow$  0; j  $\leq$  9; j  $\leftarrow$  j + 1 do
14:        while !isEmpty(v_queue[j]) do
15:          value  $\leftarrow$  front(v_queue[j])
16:          v_queue[j]  $\leftarrow$  dequeue(v_queue[j])
17:          v[k]  $\leftarrow$  value
18:          k  $\leftarrow$  k + 1
19:        end while
20:      end for
21:    end for
22: end procedure
```

# Algoritmul Radix Sort



Pasul 1



Vectorul a devenit:



# Algoritm Radix Sort



Pasul 2

<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 0	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 1
0	1	2	3	4									
0	1	2	3	4									
<table border="1"><tr><td>105</td><td></td><td></td><td></td><td></td></tr></table>	105						<table border="1"><tr><td>15</td><td></td><td></td><td></td><td></td></tr></table>	15					
105													
15													
<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 2	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 3
0	1	2	3	4									
0	1	2	3	4									
<table border="1"><tr><td>120</td><td>220</td><td>22</td><td></td><td></td></tr></table>	120	220	22				<table border="1"><tr><td></td><td></td><td></td><td></td><td></td></tr></table>						
120	220	22											
<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 4	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 5
0	1	2	3	4									
0	1	2	3	4									
<table border="1"><tr><td>48</td><td></td><td></td><td></td><td></td></tr></table>	48						<table border="1"><tr><td></td><td></td><td></td><td></td><td></td></tr></table>						
48													
<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 6	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 7
0	1	2	3	4									
0	1	2	3	4									
<table border="1"><tr><td>760</td><td></td><td></td><td></td><td></td></tr></table>	760						<table border="1"><tr><td></td><td></td><td></td><td></td><td></td></tr></table>						
760													
<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 8	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	2	3	4	— Coada 9
0	1	2	3	4									
0	1	2	3	4									
<table border="1"><tr><td>86</td><td></td><td></td><td></td><td></td></tr></table>	86						<table border="1"><tr><td></td><td></td><td></td><td></td><td></td></tr></table>						
86													

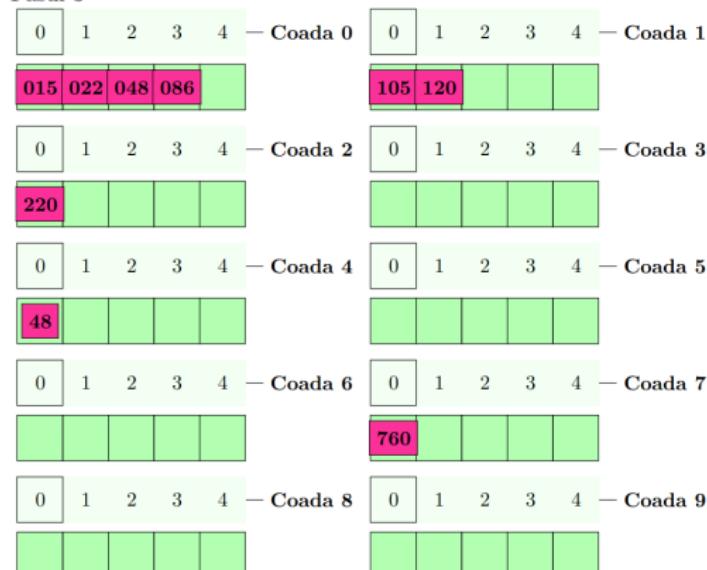
Vectorul a devenit:



# Algoritmul Radix Sort



Pasul 3



Vectorul a devenit:



# Întrebări?!

