MINISTERUL EDUCAȚIEI ȘI CERCETĂRII
ROMÂNIA
UNIVERSITATEA DE MEDICINĂ,
FARMACIE, ȘTIINȚE ȘI TEHNOLOGIE
„GEORGE EMIL PALADE"
DIN TÂRGU MUREȘ

FACULTATEA DE INGINERIE ȘI TEHNOLOGIA INFORMAȚIEI

# Algoritmi fundamentali

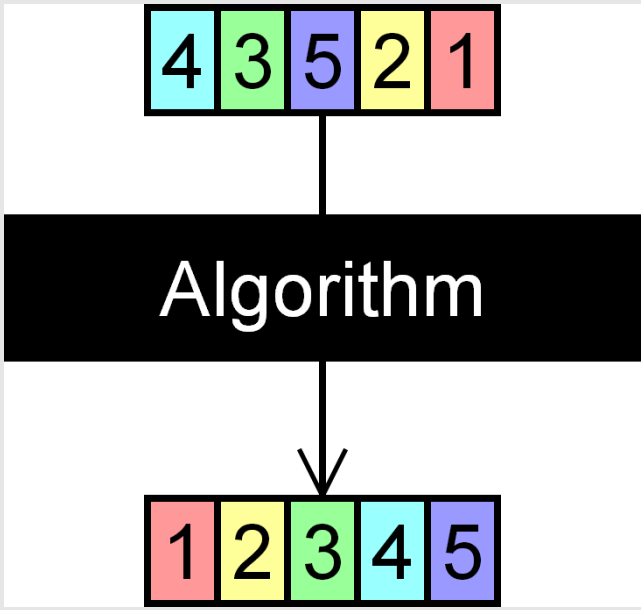
## Curs 7

## Algoritmi de sortare

Dr. ing. Kiss Istvan

istvan.kiss@umfst.ro

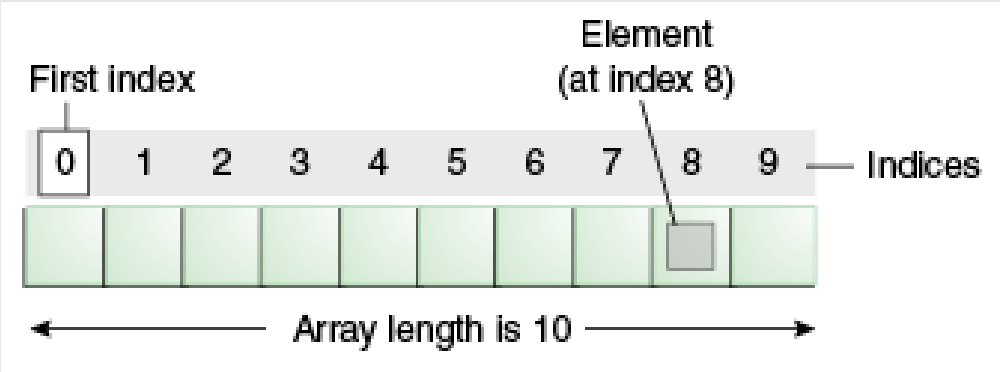
# Cuprins



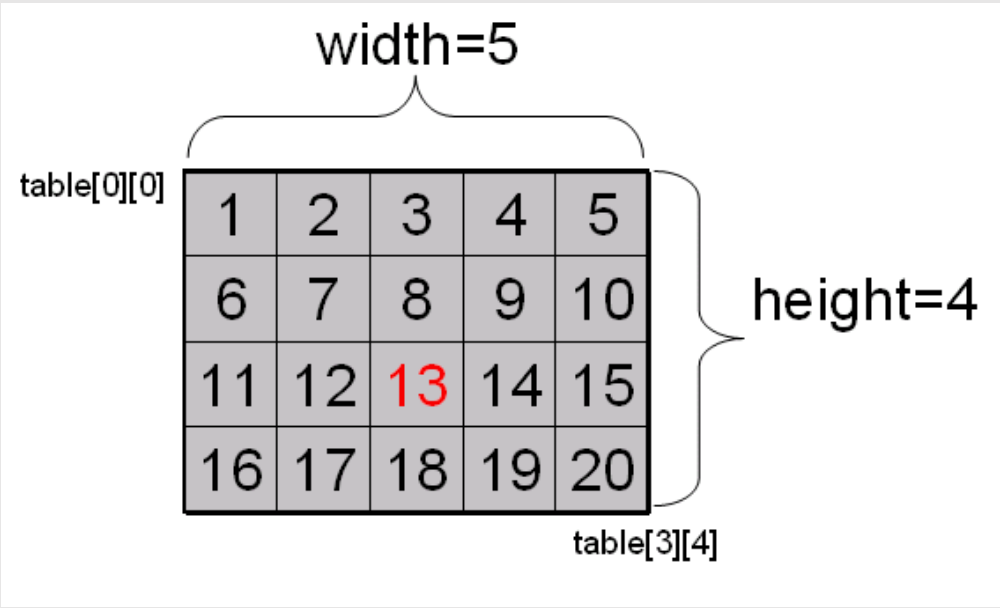

1. Tablouri uni-,bidimensionale
   1. Introducerea si afisarea elementelor
2. Sortare prin compararea elementelor
   1. Sortare prin insertie (Insertion Sort)
   2. Sortare prin selectie cu pas variabil (Shell sort)
   3. Sortare prin selectie directa (Selection sort)
   4. Sortare prin numarare
   5. Sortare prin metoda bulelor (Bubble sort)
3. Sortare liniara
   1. Sortare pe baza unui tabel de frecvente (Counting sort)
   2. Sortare pe baza cifrelor (Radix sort)
   3. Bucket sort (Bin sort)
4. Metode avansate de sortare
   1. Sortare rapida (QuickSort)
   2. Sortare prin interclasare (Merge sort)
   3. Heapsort
5. Probleme

- Knuth, Donald Ervin. *The art of computer programming*. Vol. 3. Pearson Education, 1997.
  - Descrie 33 de metode de sortare…

# 1. Tablouri




- Unidimensionale – vectori, array
  - Un singur index pentru accesarea elementelor
- Bidimensionale – matrice, 2D array
  - Doi indecsi pentru accesarea elementelor
- Multidimensionale
  - n indecsi

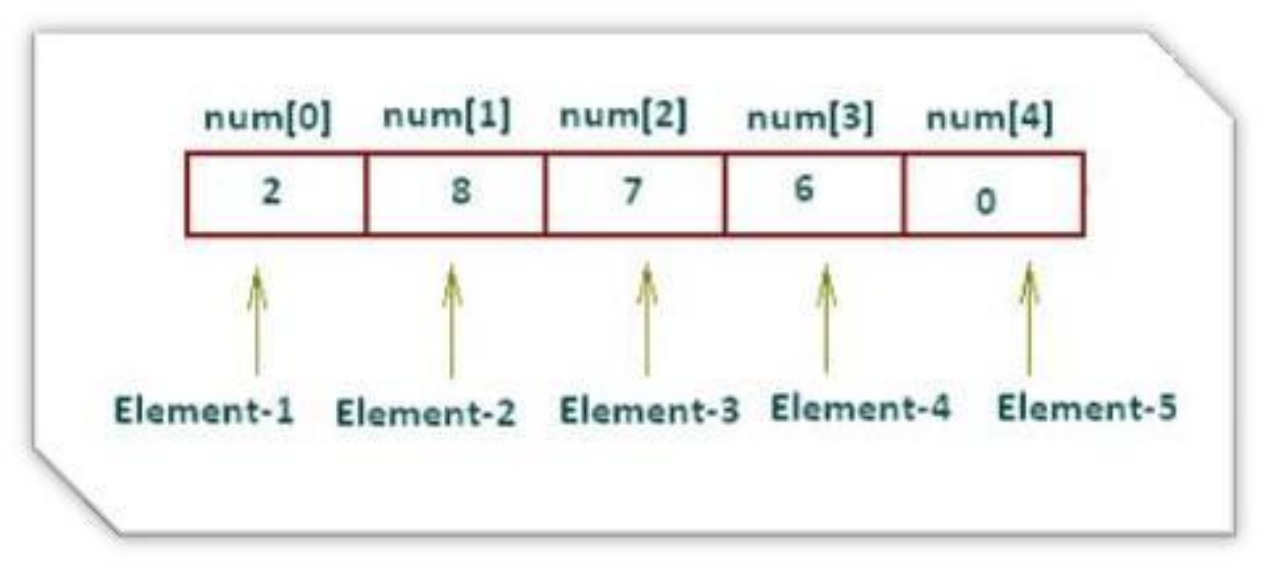


# 1. Tablouri unidimensionale

- Tabloul este o colectie finita de elemente de acelasi tip, numit tip de baza al tabloului, care ocupa un spatiu continuu de memorie.
- Un element este in mod unic identificat prin pozitia ocupata in cadrul structurii.
- Descrierea in pseudocod:
  - Nume, tipul comun al elementelor, tipul tabloului
  - Primul element are indice *0*, air elementul cel mai superior are indice (*dimensiune-1*)
  - vect[0] primul element
  - .....
  - vect[9] ultimul element

# 1. Tablouri unidimensionale

- Declaratia generala de tablou
  - **tip nume_tabl [lim_1][lim_2]...[lim_n] = {lista valori}**
- Vectori – tablouri unidimensionale
  - **tip nume_tabl[lim_1] = {lista valori}**
- **lim_1 – dimensiunea vectorului**



# 1. Tablouri unidimensionale

- **Citirea elementelor**

```
Subalgoritmul citire(n,a) este:
        Intreg i;
        citeste n;
        pentru i:=0;i<=n
                citeste a[i];
        sf.pentru
Sf.citire
```

- **Afisarea elementelor**

```
Subalgoritmul afisare(n,a) :
        Intreg i;
        pentru i:=0;i<=n
                scrie a[i];
        sf.pentru
Sf.citire
```

# 1. Tablouri unidimensionale

- **Produsul scalar a doi vectori**

```
Intreg n, v[100], w[100];
Citeste n; citire(n,v); citire(n,w);
Intreg p=0;
Pentru i:=0;n-1
        p:=p+v[i]*w[i];
Sf.pentru
Afiseaza p;
```

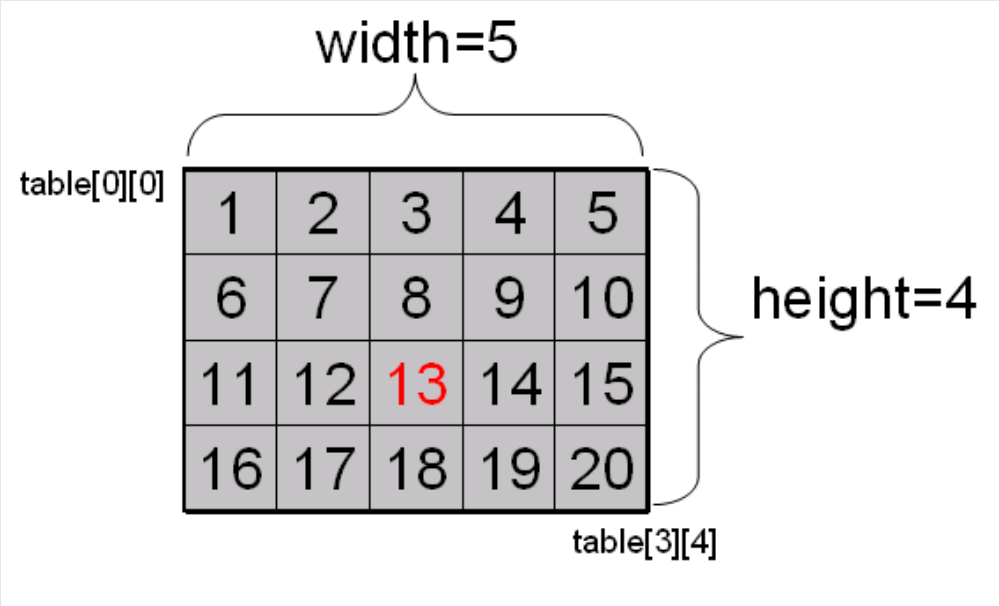
# 1. Tablouri unidimensionale

- **Sortarea crescatoare a elementelor unui vector**

1. Intreg n, v[100];
2. Citeste n; citire(n,v);
3. m=n, sort=0
4. CÂT TIMP (m>0) ŞI (sort=0)
    - 4.1. sort=1
    - 4.2. PENTRU i=0,m-1
        - DACĂ (v[i]>v[i+1])
            - 4.2.1. sort=0
            - 4.2.2. aux=v[i]
            - 4.2.3. v[i]=v[i+1]
            - 4.2.4. v[i+1]=aux
    - 4.3. m=m-1
5. afisare(n,v)

# 1. Tablouri bidimensionale

- **Matrici**
- Tip nume[dim1][dim2]
- Intreg mat[4,5];
  - 4 linii
  - 5 coloane
- Acces la elemente
  - mat[1,2] = 8
  - mat[0,0] = 1



# 1. Tablouri bidimensionale

- **Citirea elementelor**

```
Subalgoritmul citire(n,m,a) este:
        Intreg i,j;
        citeste n,m;
        pentru i:=0;n-1
                pentru j:=0;m-1
                        citeste a[i,j];
                sf.pentru
        sf.pentru
Sf.citire
```

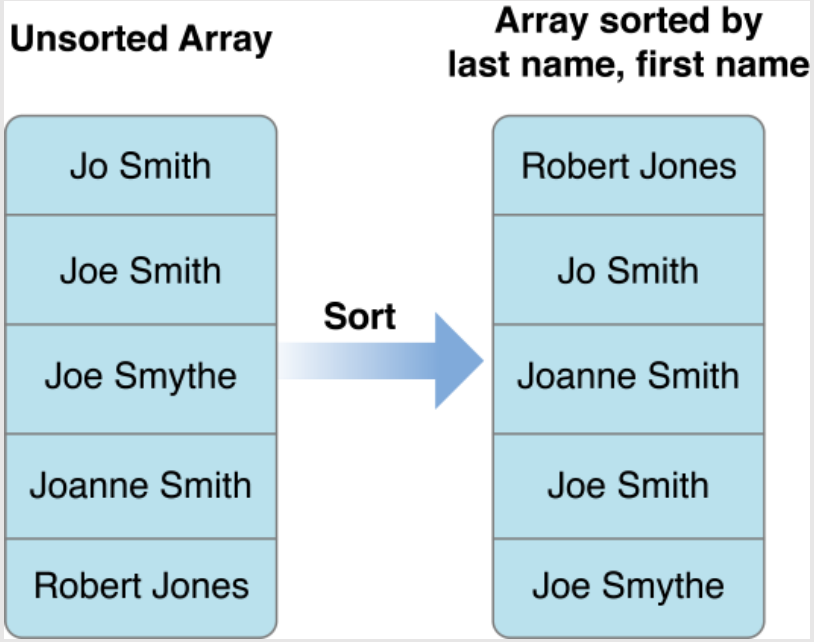
# 1. Tablouri bidimensionale

- **Afisarea elementelor**

```
Subalgoritmul afisare(n,m,a) este:
        Intreg i,j;
        pentru i:=0;n-1
                pentru j:=0;m-1
                        afiseaza a[i,j];
                sf.pentru
                afiseaza {linie noua}
        sf.pentru
Sf.citire
```

# 2. Sortare

- Este o operatie asupra unui sir finit de elemente
- Se bazeaza pe compararea unei caracteristici bine definite, numita ***cheie***.
- Este procesul prin care obiectele sunt rearanjate astfel incat cheile lor sa respecte o ordine.

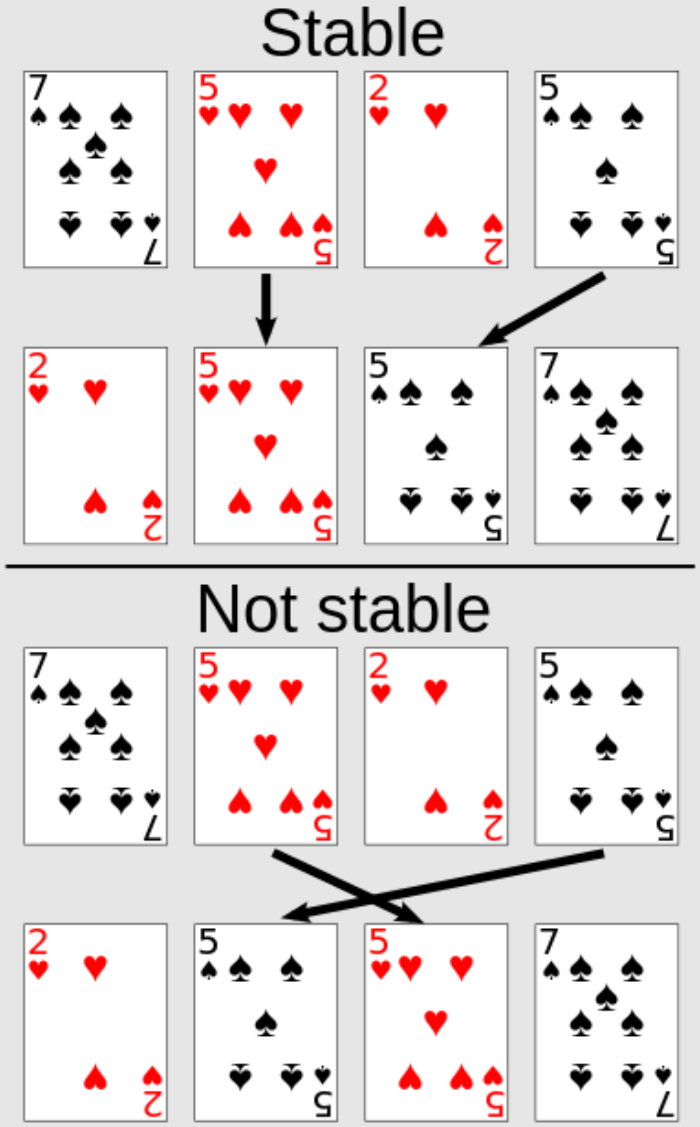


# 2. Sortare – definitie matematica

- Fiind date N inregistrari, sa se aranjeze aceste inregistrari in functie de valoarea campului K, numit cheie de sortare, astfel incat intre oricare doua inregistrari vecine $R_i$ si $R_{i+1}$ sa existe una dintre relatiile urmatoare intre cheile de sortare:
  - $K_i < K_{i+1}$ sau $K_i > K_{i+1}$
- Relatia de ordine < sau > trebuie sa satisfaca:
  - Una si numai una din variantele: a<b, a=b, a>b este adevarata (legea trihotomiei)
  - Daca a<b si b<c atunci a<c (legea tranzitivitatii).

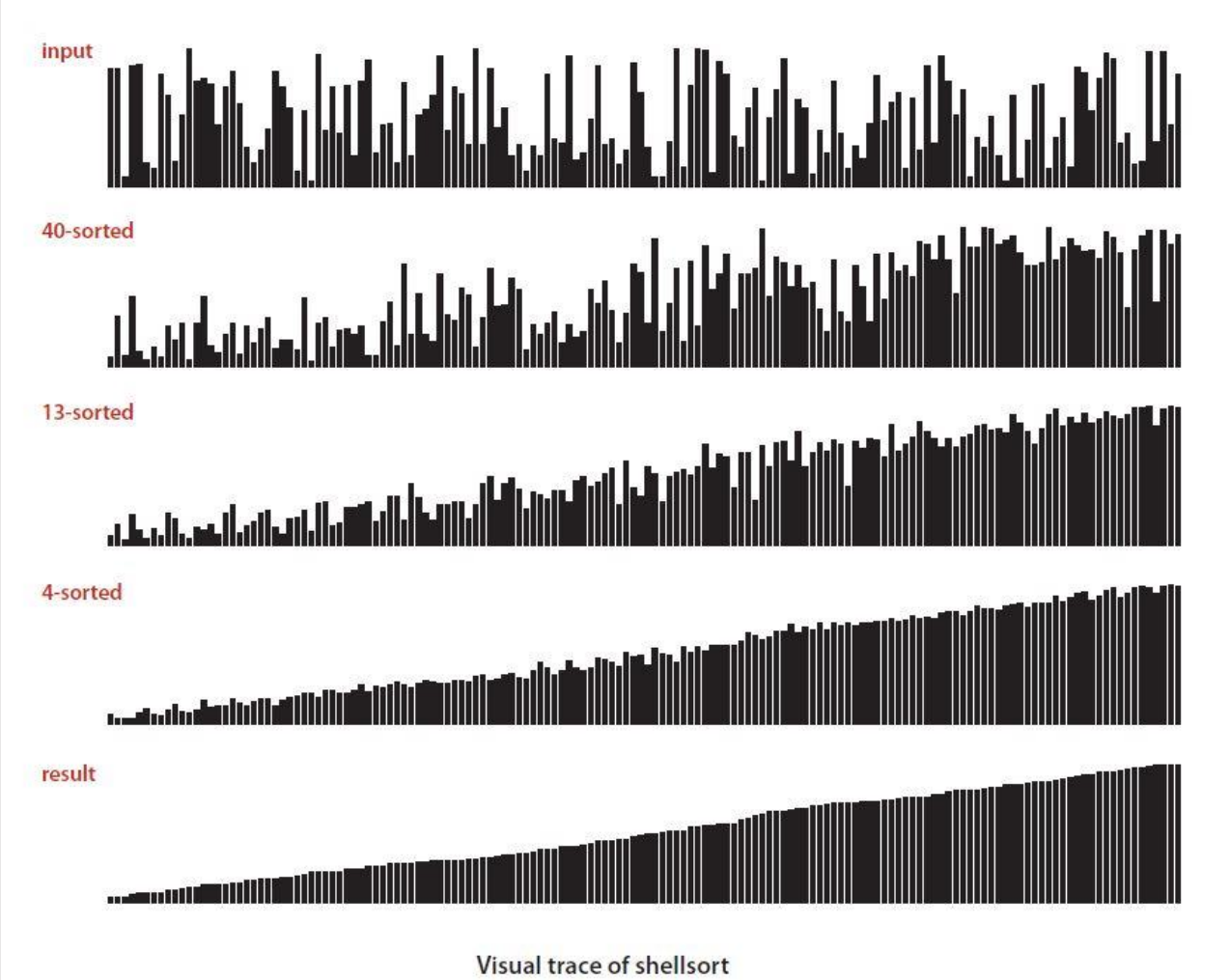
# 2. Sortare – caracteristici

- Sortare interna
  - Pe parcursul sortarii inregistrarile sunt salvate in memoria interna
- Sortare externa
  - Inregistrarile sunt stocate pe memorii externe
- Sortare stabila
  - Inregistrarile cu chei egale trebuie sa-si pastreze pozitiile relative.
- Eficienta, simplicitate



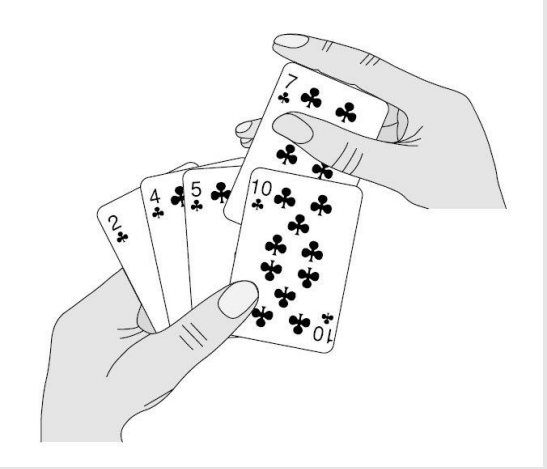
# 2. Ilustratii vizuale

- https://www.toptal.com/developers/sorting-algorithms



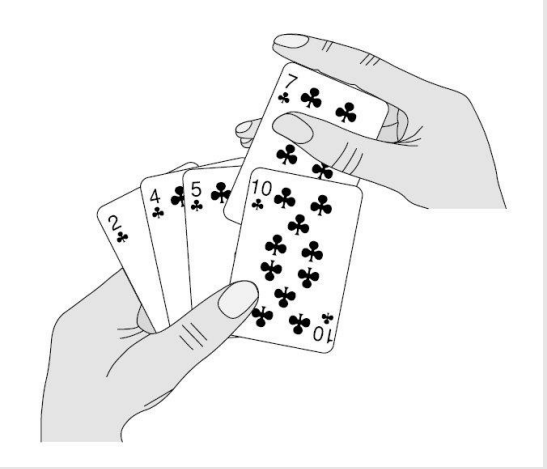
Visual trace of shellsort

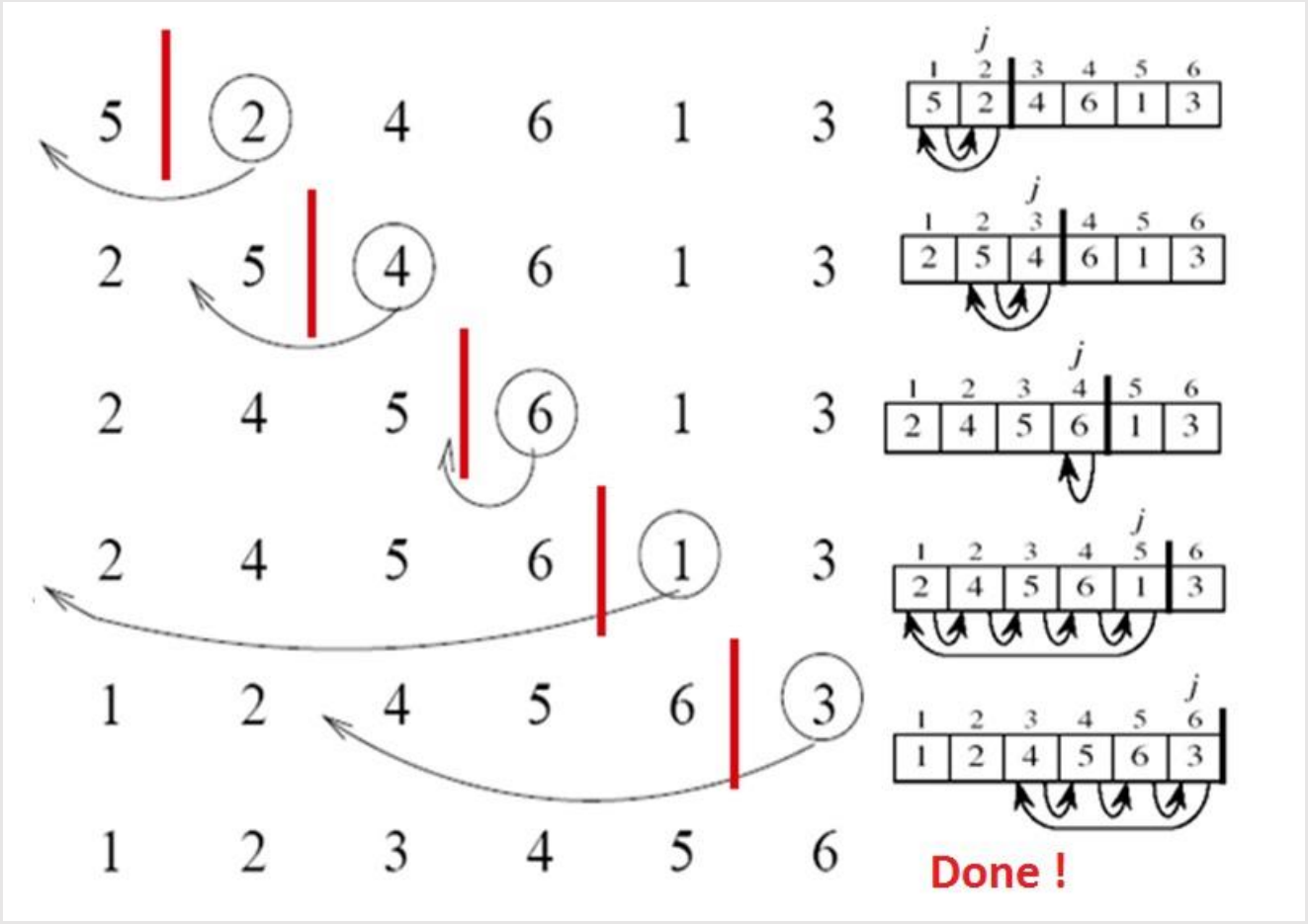
# 2.1. Sortare prin insertie
# Insertion sort



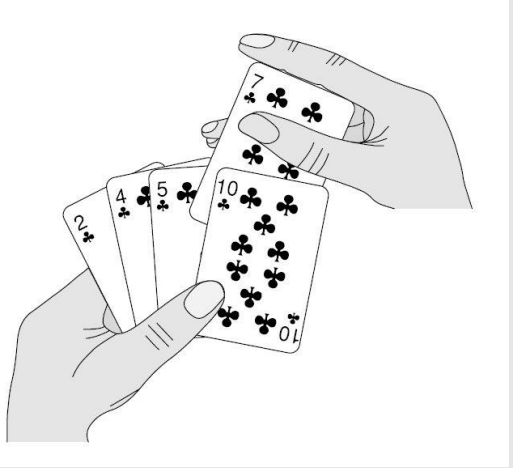
- Incepand cu al doilea element fiecare element este inserat pe pozitia corespunzatoare, presupunand ca elementele anterioare sunt deja sortate.
- Memoria necesara: vectorul elementelor si variabila temporara pentru elementul care urmeaza sa fie inserat.
- Pentru fiecare element v[i] se cauta pozitia corespunzatoare in intervalul [i-1,0] (interval ordonat deja), comparand pe v[i] cu elementele precedente de la i-1 pana la 0. Se creeaza loc in vector prin deplasarea elementelor mai mari decat v[i] cu o pozitie spre dreapta.

# 2.1. Sortare prin insertie
# Insertion sort



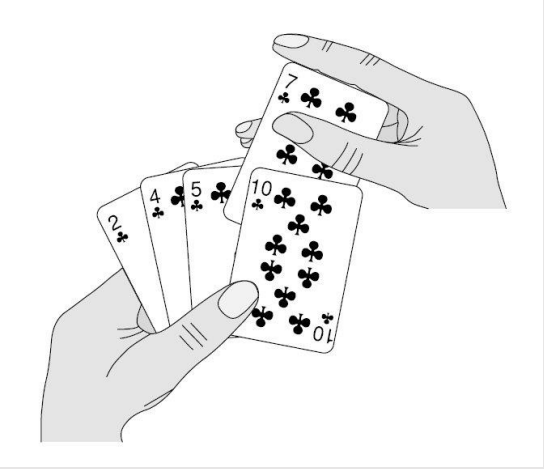


# 2.1. Sortare prin insertie
# Insertion sort



- sortare_insertie(v,n)
- pt. i:=1..n-1 exec
- j:=i-1;
- aux:=v[i];
- cat timp j>=0 si aux<v[j] exec.
- v[j+1]:=v[j];
- j:=j-1;
- sf. cat timp
- v[j+1]:=aux;//de ce e v[j+1] și nu v[j]?
- sf. pt.

# 2.1. Sortare prin insertie Complexitate



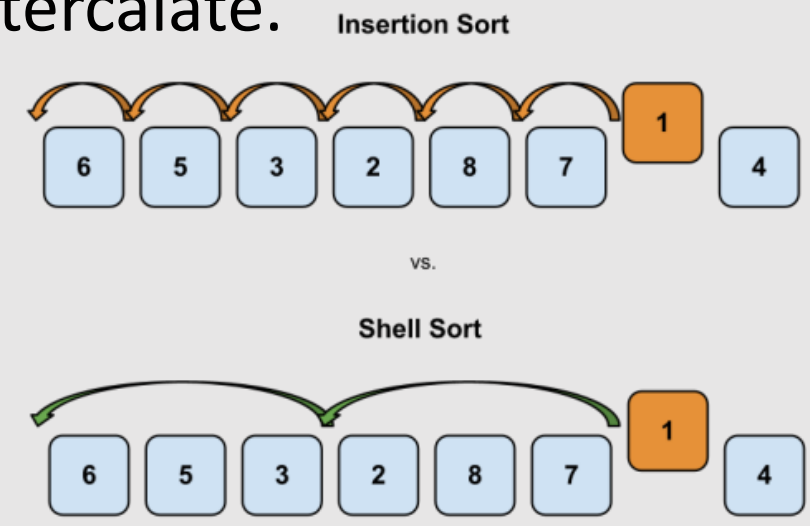
- Operatii elementare: comparative si mutare
- Cel mai favorabil caz cand vectorul este ordonat deja
  - O(3(n-1)) ~ O(n)
- Cel mai nefavorabil caz, vectorul in ordine descrescatoare

$$\sum_{i=2}^{n}(i-1+i+2)=\sum_{i=2}^{n}2i+1=\frac{2(n-1)(n+2)}{2}+n-1=n^2+n-3\Rightarrow O(n^2)$$

- Deci in general algoritmul are complexitatea $O(n^2)$.

# 2.2. Sortare prin selectie cu pas variabil (Shell sort)

- Este o extensie simpla a Insertion sort, care castiga viteza permitand schimbarea elementelor aflate departe.
- Rearanjeaza elementele in asa fel incat, luand fiecare a *h*-a element (incepand de oriunde), sa obtinem un vector sortat. Astfel vectorul devine *h-sortat.*
- Un vector *h-sortat* este format din *h* subvectori sortati independent, intercalate.



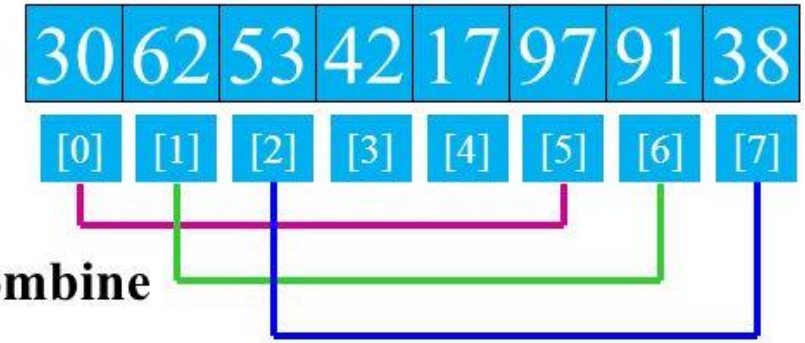
# 2.2. Sortare prin selectie cu pas variabil (Shell sort)

- Let's sort the following list given the sequence (gaps) numbers are 5, 3, 1

30 62 53 42 17 97 91 38

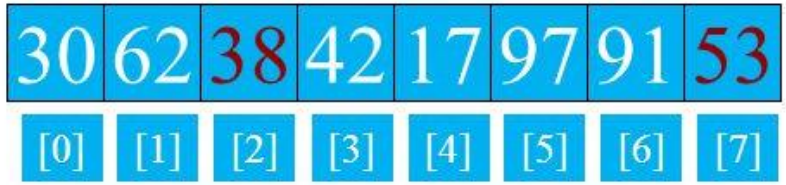
**Step 1: Create the sub list k = 5**

S[0] S[5]
S[1] S[6]
S[2] S[7]
S[3]
S[4]




**Step 2 - 3: Sort the sub list & combine**

S[0] < S[5] This is OK
S[1] < S[6] This is OK
S[2] > S[7] This is not OK.
Swap them



# 2.2. Sortare prin selectie cu pas variabil (Shell sort)

- v[i0], v[i0+h], v[i0+2h]… este sortat
- Cu alegerea adecvata a lui h se poate reduce ordinea complexitatii de la $O(n^2)$ la $O(n^{3/2})$.
- $h_k=2^k-1$, unde $1<=k<=ln(n)$
- Sau h ales altfel, h={5,3,1}
  - Sortare din 3 etape; ultima etapa cu h=1 este defapt sortarea obisnuita prin insertie.

# 2.2. Sortare prin selectie cu pas variabil (Shell sort)

- Fie h={5,3,1} – pasii variabili
- v={62 83 18 53 7 17 95 86 47 59 25 28}

1. 17 28 18 47 7 25 83 86 53 69 62 95
2. 17 28 18 47 7 25 83 86 53 69 62 95
3. 17 7 18 47 28 25 69 62 53 83 86 95
4. 17 7 18 47 28 25 69 62 53 83 86 95
5. 7 17 18 25 28 47 53 59 62 83 86 95

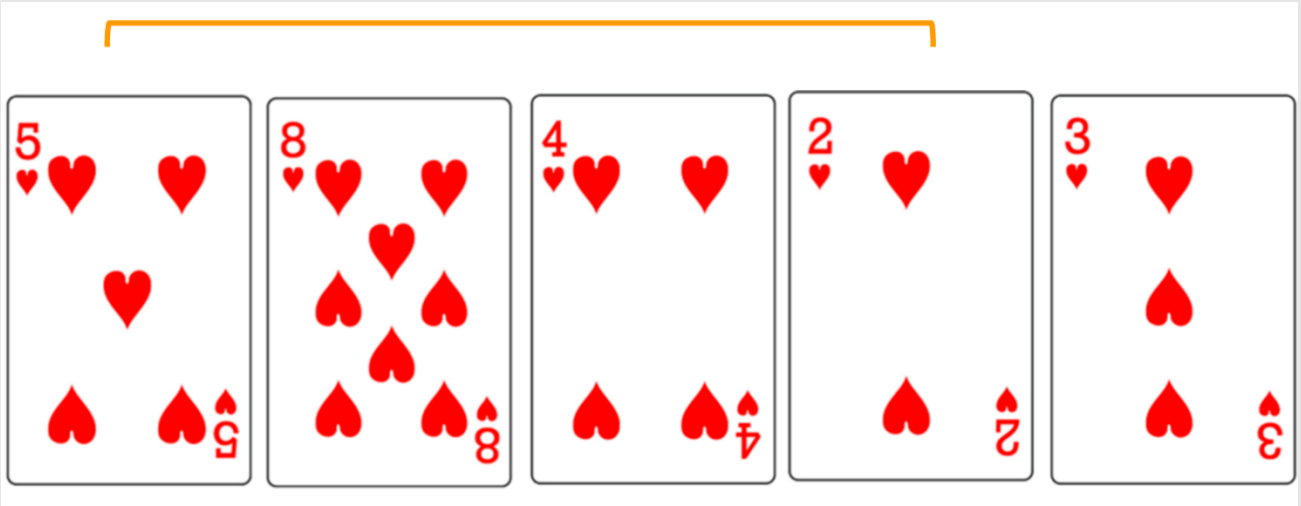
- Cum functioneaza?

# 2.2. Sortare prin selectie cu pas variabil (Shell sort)

```
h[] = {5, 3, 1};  nh:=3;
pt. ih:=0..nh-1
    pas=h[ih]; //o valoare din h
    pt. i = pas..n-1
        temp := v[i];
        pt. (j := i; j >= pas si v[j - pas] > temp; j -= pas)
            v[j] := v[j - pas];
        sf. pt.
        v[j] := temp;
    sf. pt.
sf. pt
```

# 2.3. Sortare prin selectie directa

- Primul element se compara pe rand cu toate elementele de dupa el si daca ordinea de sortare nu este respectata, cele doua elemente se interchimba.
- Al doilea element se compara pe rand cu toate elementele de dupa el…
- Se repeta operatia pentru fiecare element din vector.



# 2.3. Sortare prin selectie directa

| v[0] | v[1] | v[2] | v[3] | v[4] | index |
|---|---|---|---|---|---|
| **64** | **25** | **12** | **22** | **11** | i=0, j=1 v[0]>v[1] 64>25 isch |
| **25** | 64 | **12** | 22 | 11 | i=0, j=2 v[0]>v[2] 25>12 isch |
| **12** | 64 | 25 | **22** | 11 | i=0, j=3 v[0]>v[3] 12>22 nu |
| **12** | 64 | 25 | 22 | **11** | i=0, j=4 v[0]>v[4] 12>11 isch |
| 11 | **64** | **25** | 22 | 12 | i=1, j=2 v[1]>v[2] 64>25 isch |
| 11 | **25** | 64 | **22** | 12 | i=1, j=3 v[1]>v[3] 25>22 isch |
| 11 | **22** | 64 | 25 | **12** | i=1, j=4 v[1]>v[4] 22>12 isch |
| 11 | 12 | **64** | **25** | 22 | i=2,j=3 v[2]>v[3] 64>25 isch |
| 11 | 12 | **25** | 64 | **22** | i=2,j=4 v[2]>v[4] 25>22 isch |
| 11 | 12 | 22 | **64** | **25** | i=3, j=4 v[3]>v[4] 64>25 isch |
| 11 | 12 | 22 | 25 | **64** | i=4 stop |

# 2.3. Sortare prin selectie directa

```
pt. i:=0..n-2  //elem. care compară
      pt. j:=i+1..n-1//el. după el cu care
          daca v[i]>v[j]                    se compară
             aux=v[i]; //interschimbăm
             v[i]=v[j];  //elementele
             v[j]=aux;
          sf. daca
      sf. pt
sf. pt.
```

# 2.3. Sortare prin selectie directa

- Complexitatea algoritmului
  - Primul element $v_0$ se compara cu $v_1$ pana la $v_{n-1}$
  - Al doilea element $v_1$ se compara cu $v_2$ pana la $v_{n-1}$
  - …
  - Penultimul $v_{n-2}$ se compara cu $v_{n-1}$

$$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

  - Cazul cel mai nefavorabil $O(n^2)$

# 2.4. Sortare prin numarare

- Necesita zone de memorie auxiliare
  - Vectorul destinatie
  - Vectorul de numarator (contor)
- Elementele vectorului sursa se copiaza in vectorul destinatie prin inserarea in pozitia corespunzatoare, astfel incat vectorul destinatie sa respecte relatia de ordine.
- Pentru a se cunoaste pozitia in care se va insera fiecare element, se parcurge vectorul sursa si se numara in vectorul contor pentru fiecare element, cate elemente au valoarea mai mica.
- Valoarea vectorului contor[i] pentru elementul v[i] reprezinta cate elemente sunt mai mici decat el si arata pozitia in care trebuie copiat in vectorul destinatie.
  - dest[contor[i]]:=v[i]

# 2.4. Sortare prin numarare

```
pt. i:=0..n-2  exec
    pt. j:=i+1..n-1 exec
        daca v[i]>v[j] atunci contor[i]++;
        altfel contor[j]++;
         sf. daca
    sf. pt
sf. pt.
pt. i:=0..n exec
     dest[contor[i]]:=v[i];
sf. pt.
```

# 2.4. Sortare prin numarare

- v[]={7,2,3,3}
- Contor={3,0,1,1}
- Dupa atribuire pe baza contorului dest={2,3,0,7}
- 0 apare din cauza ca elemental 3 se repeta in vectorul sursa
- Pentru rezolvare mai trebuie parcursa odata sirul si la elementul care nu respecta relatia de ordine se atribuie valoarea elementului precedent.

```
pt. i:=0..n-1 exec
daca dest[i]>=dest[i+1] atunci
        dest[i+1]:=dest[i];
sf. daca
sf. pt.
```

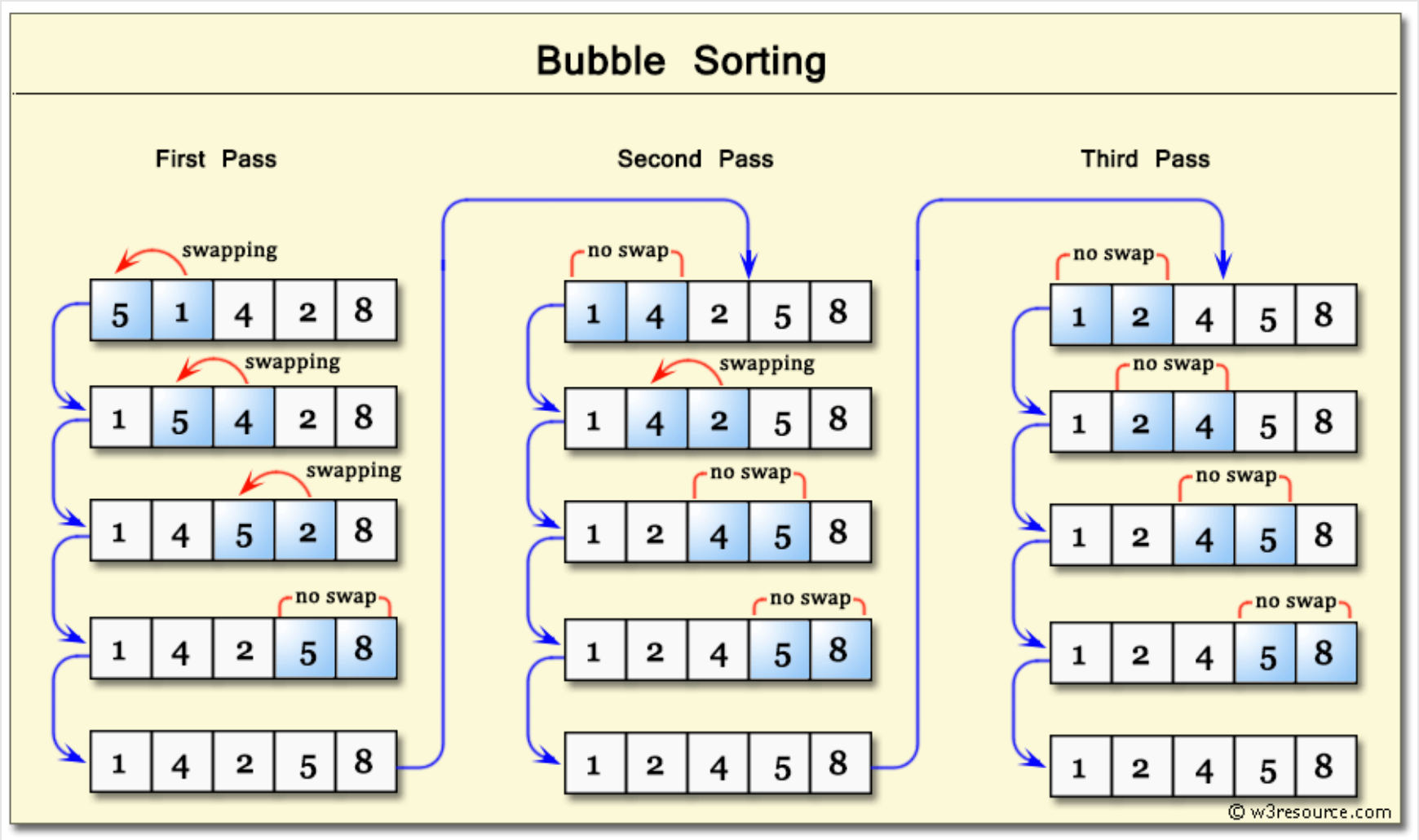
# 2.4. Sortare prin numarare

- Complexitatea algoritmului
  - $O(n^2)$ – nu exista caz favorabil sau nefavorabil

# 2.5. Sortare prin metoda bulelor (Interschimbarea elementelor vecine)

- Se parcurge tabloul si se compara elementele vecine, iar daca acestea nu se afla in ordine corecta se interschimba.
- Parcurgerea se reia pana cand nu mai este necesara nicio interchimbare!!!
- Literatura prezinta ca si un algoritm care nu merita interes.

# 2.5. Sortare prin metoda bulelor (Interschimbarea elementelor vecine)



# 2.5. Sortare prin metoda bulelor (Interschimbarea elementelor vecine)

- Varianta 1 – complexitate $O(n^2)$

```
repeta
    interschimbare=false;
    pt. i:=0..n-2
        daca (v[i]>v[i+1])
            aux:=v[i];
            v[i]:=v[i+1];
            v[i+1]:=aux;
            interschimbare:=true;
        sf. daca
    sf. pt.
cat timp interschimbare=true; //repeta pana cand
//nu se mai face intreschimbare (devine fals)
```

# 2.5. Sortare prin metoda bulelor (Interschimbarea elementelor vecine)

- Varianta 2 – complexitate $O(n^2)$ (Best case $O(n)$)
  - Se retine pozitia interschimbarii care s-a facut ultima data intr-o parcurge; in asa fel se poate limita regiunea analizata la urmatoarea iteratie.

```
pozitieinterschimbare=n-1;
    repeta
        interschimbare=false;
        pozitie:=0;
        pt. i:=0..pozitieinterschimbare
        daca v[i]>v[i+1] atunci
            aux:=v[i];
            v[i]:=v[i+1];
            v[i+1]:=aux;
            interschimbare:=true;
            pozitie:=i;
        sf. daca
        sf. pt.
        pozitieinerschimbare:=pozitie;
    cat timp interschimbare=true;
```

# 2.5. Sortare prin metoda bulelor (Interschimbarea elementelor vecine)

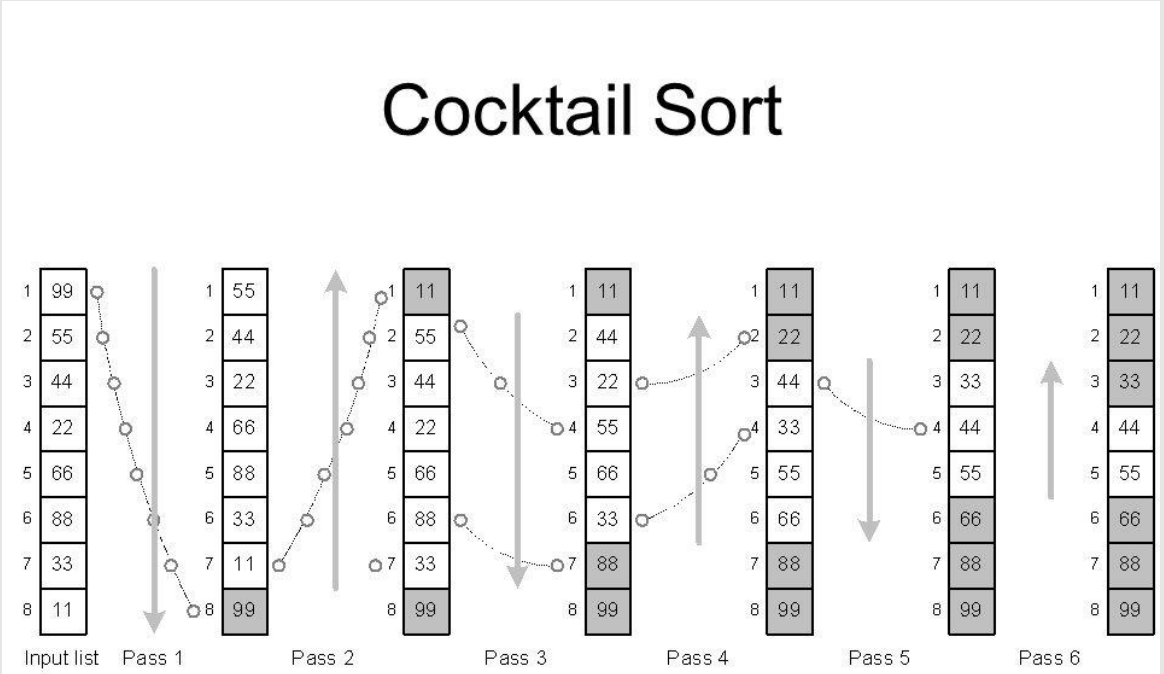
- Varianta 3 – complexitate $O(n^2)$ (Best case $O(n)$)
  - ***Shaker sort – Cocktail sort***
  - Parcurge sirul in mod alternativ: in prima iteratie de la stanga la dreapta iar in al doilea de la dreapta la stanga si asa mai departe…

```
repeta
    interschimbare := false;
    pt. i :=0..n - 2 exec //stânga dreapta
      daca v[i] > v[i+1] atunci
           aux:=v[i];
           v[i]:=v[i+1];
           v[i+1]:=aux;
          interschimbare := true;
      sf. daca
    sf. pt.
    daca interschimbare=false
      // nu s-a facut interschimbare iesim din ciclu
      break;
    sf. daca
```

```
interschimbare := false;
    pt.i:=n-2..0 exec // dreapta stânga
     daca v[i] > v[i+1] atunci
        aux:=v[i];
       v[i]:=v[i+1];
        v[i+1]:=aux;
        interschimbare := true;
      sf. daca
    sf. pt.
  pana cand interschimbare=true;
```

# 2.5. Sortare prin metoda bulelor (Interschimbarea elementelor vecine)

- Varianta 3 – complexitate $O(n^2)$ (Best case O(n))
  - ***Shaker sort – Cocktail sort***
  - Exemplu: (2,3,4,5,1) necesita numai o singura parcurgere cu cocktail sort.



# 3. Sortare - continuare

1. Sortare liniara
    1. Sortare pe baza unui table de frecvente (Counting sort)
    2. Sortare pe baza cifrelor (Radix sort)
    3. Bucket sort (Bin sort)
2. Metode avansate de sortare
    1. Sortare rapida (QuickSort)
    2. Sortare prin interclasare (Merge sort)
    3. Heapsort

# 3.1. Sortare pe baza tabelului de frecvente (Counting sort)

- Conditii preliminare:
  - Vectorul v[n] are elementele in intervalul [1,m]
  - Se construieste tabelul de frecvente fr[m]
  - n>m – n este mult mai mare decat m
- Se numara frecventa elementelor individuale.
- Se modifica tabelul fr astfel incat sa contine pentru fiecare element numarul de aparitie a elementelor mai mici → acesta se face prin frecvente cumulate si **nu** prin comparare (sortare prin numarare).
- Tabelul ordonat y se creeaza prin folosirea tabelului frecventelor cumulate.

# 3.1. Sortare pe baza tabelului de frecvente (Counting sort)

**Input Data**

| 0 | 4 | 2 | 2 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 4 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Count Array**

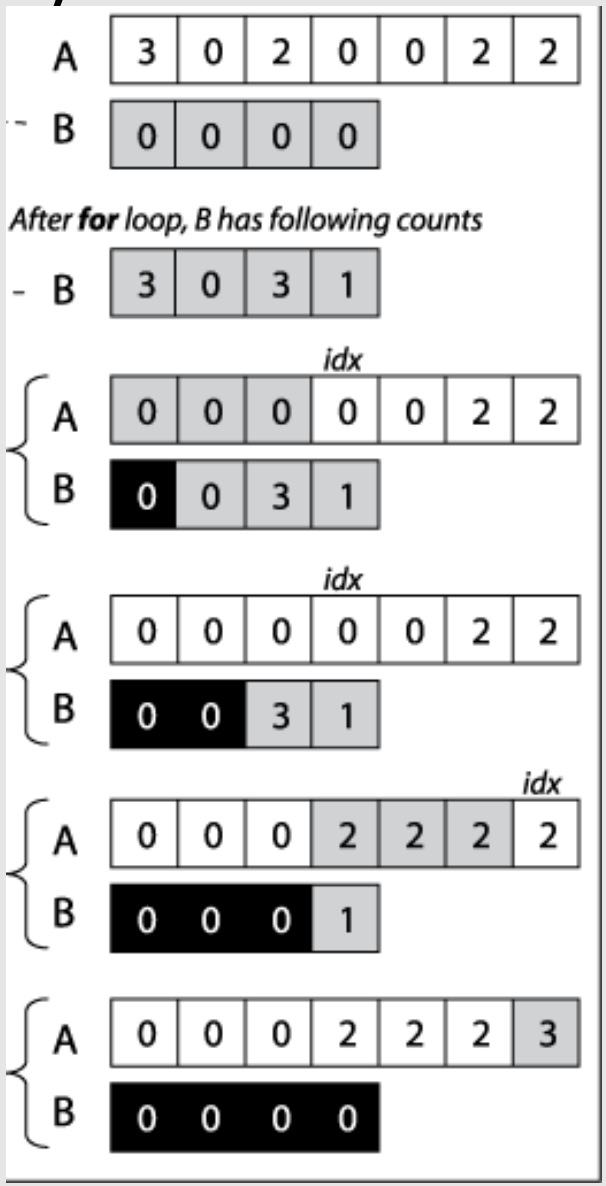
| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 3 | 4 | 0 | 2 |

**Sorted Data**

| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# 3.1. Sortare pe baza tabelului de frecvente (Counting sort)

- Complexitate O(m+n)
- Memorie necesare:
  - Tabloul de frecvente fr
  - Tabloul ordonat y



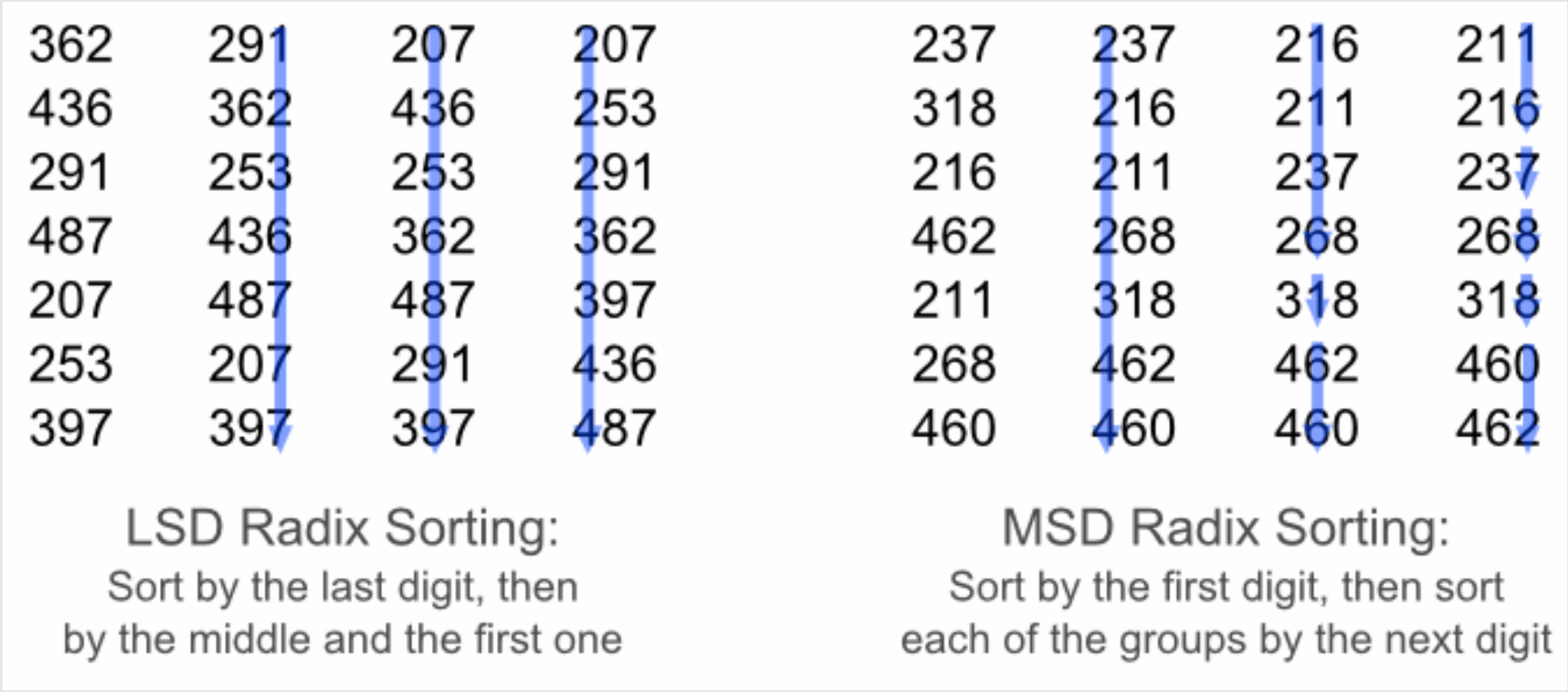
# 3.1. Sortare pe baza tabelului de frecvente (Counting sort)

- Algoritm CountingSort(vector,n)
- m=maxim(vector,n);
- pt. i=1..m exec. fr[i]=0; sf. pt.
- pt. i=0..n-1 exec. fr[v[i]]++;sf. pt. //calc. fr.
- pt. i=1..m exec. fr[i]=fr[i-1]+fr[i] sf.pt. //fr.cum
- pt. i=n-1..0 exec.
- y[fr[v[i]]-1]=v[i];
- fr[v[i]]--;
- sf. pt.
- pt. i=0..n-1 exec. //copiere y înapoi în v
- v[i]=y[i];
- sf. pt.
- sf. algoritm

# 3.2. Sortare pe baza cifrelor (Radix sort)

- Conditii preliminare
  - Elementele vectorului sunt numere natural de cel mult k cifre
  - k<n
- **1.** Se sorteaza elementele folosind ca si cheia de sortare cifra cea mai putina semnificativa
- **2.** Se sorteaza elementele folosind ca si cheia de sortare cifra urmatoare
- …
- **k.** Se sorteaza elementele folosind ca si cheia de sortare cifra cea mai semnificativa
- **La fiecare pas se foloseste CountingSort cu m=10**
  - **Trebuie sa fie stabil**

# 3.2. Sortare pe baza cifrelor (Radix sort)



# 3.2. Sortare pe baza cifrelor (Radix sort)

- **Implementare:**
  - **Foloseste subalgoritmul CountingSortCifra.**

```
Algoritm RadixSort(vector,n)
pt. pozcifra=0..k-1 exec.
    CountingSortCifra(v,n,pozcifra);
sf. pt.
sf. algoritm
```

# 3.2. Sortare pe baza cifrelor (Radix sort)

- **Implementare:**

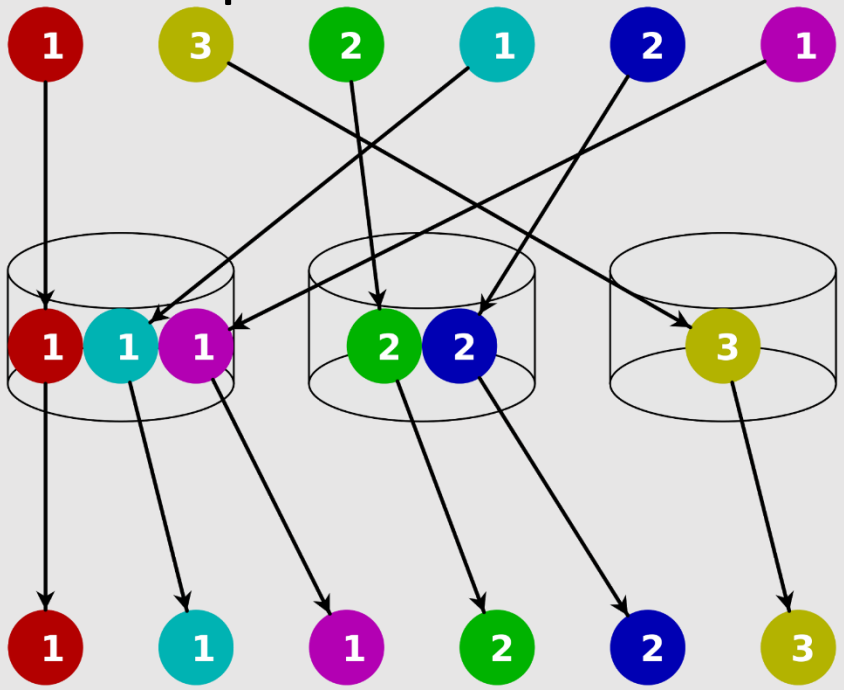
```
Subalgoritm CountingSortCifra(v,n,pozcif)
        m:=9;
        pentru i:=1, m exec. fr[i]:=0; sf.pentru
        pentru i:=0, n-1 exec.
                cifra:=(v[i]/10^pozcif) mod 10;
                fr[cifra]++;
        sf.pentru
        pentru i:=1, m exec. fr[i]=fr[i-1]+fr[i]; sf.pentru
        pentru i:=n-1, 0 exec.
                cifra:=(v[i]/10^pozcif) mod 10;
                y[fr[cifra]-1]:=v[i];
                fr[cifra]--;
        sf.pentru
        pentr i:=0, n-1 exec. v[i]:=y[i]; sf.pentru
Sf.Subalg.
```

# 3.2. Sortare pe baza cifrelor (Radix sort)

- Complexitatea algoritmului RadixSort este O(k*n)
- k – numarul de cifre
- n – numarul de elemente

# 3.3. Sortare cu galeti - Bucket sort (Bin sort)

- Sortare compartimentata
- Conditii preliminare:
  - vector de lungime n cu elemente distribuite uniform in intervalul [a,b]
- Creeaza m compartimente de marimi egale
- Fiecare compartiment va contine aproximativ un numar de n/m elemente

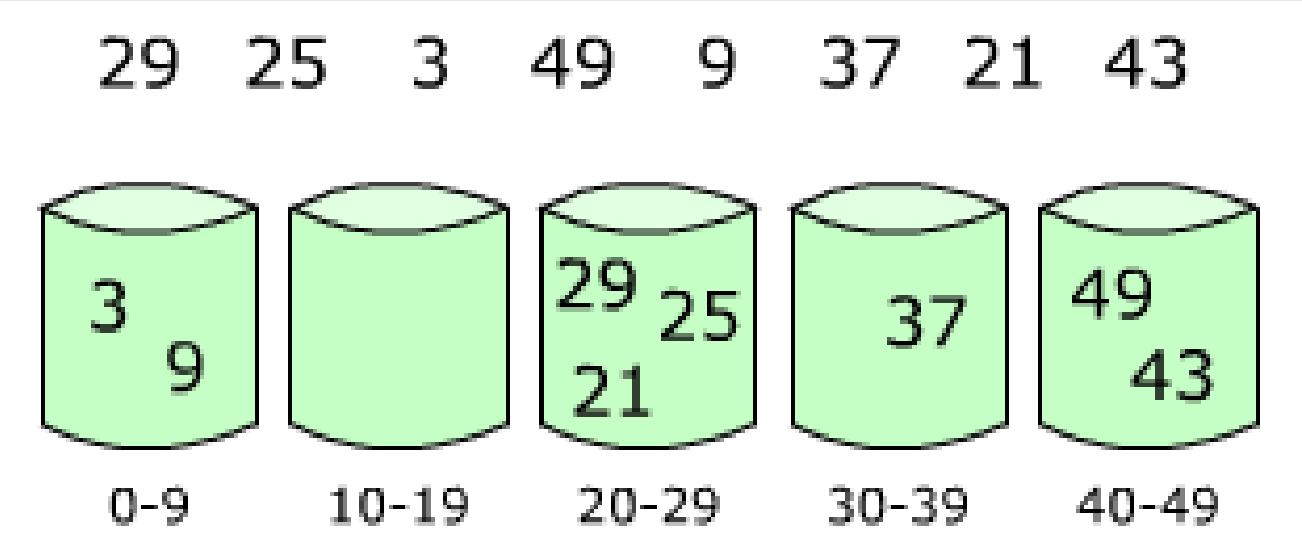


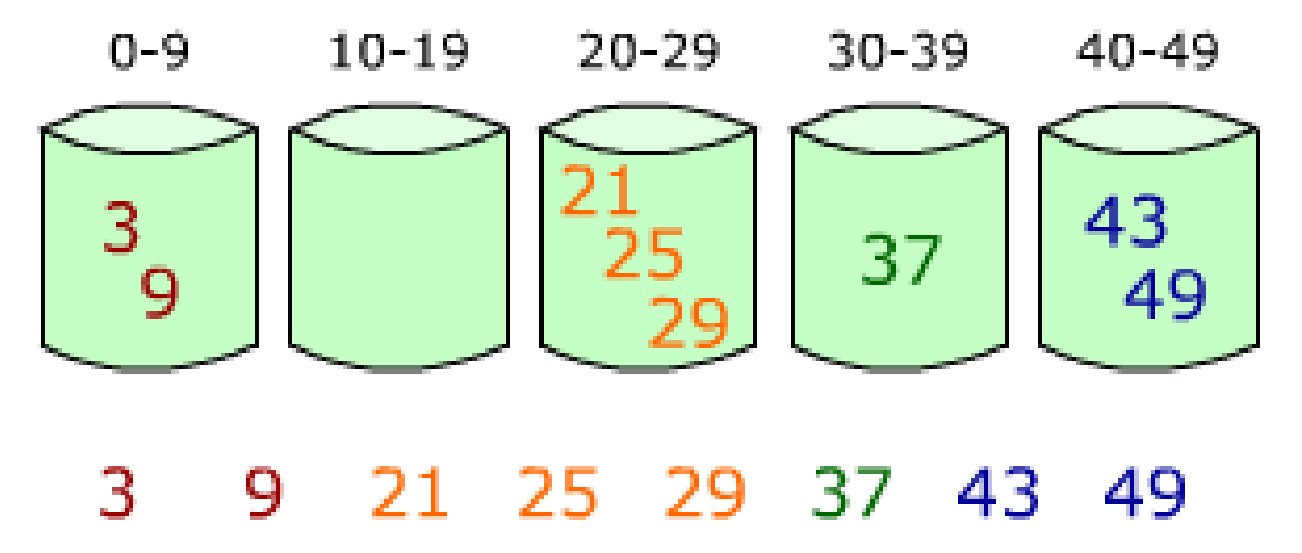
# 3.3. Sortare cu galeti - Bucket sort (Bin sort)

- Cele m intervale (galeti) se aleg astfel:
  - primul interval [a,a+n/m]
  - al doile [a+n/m+1,a+2*n/m]
  - al i-lea interval [a+(i-1)*n/m+1,a+i*n/m]
  - ultimul interval [a+(m-1)*n/m+1, b]

- Intervalele se sorteaza separate.
- In final, elementele se copiaza pe rand in vectorul rezultat.
- Complexitate: **$O(n^2)$**
  - Caz general cand elementele sunt distribuite uniform: **O(k+n)**

# 3.3. Sortare cu galeti - Bucket sort (Bin sort)



# 3.3. Sortare cu galeti - Bucket sort (Bin sort)

- **Implementare:**
  - se poate definii o structura pentru reprezentarea galetilor

```
struct Bucket{
        int inf,sup; //limitele intervalului
        nod* LIST;
}
```

  - se defineste subalgoritmul BucketSort(vector,n,m)
    - alocare spatiu pentru m galeti
    - calcul interval pentru fiecare galeata
    - elementele din vector se insereaza in galeata corespunzatoare
    - elementele din fiecare galeata se ordoneaza folosind CountingSort
    - Varianta 2 optimizata: elementele intervalelor se copiaza in vectorul initial si se aplica InsertionSort

# 3.3. Sortare cu galeti - Bucket sort (Bin sort)

- **Implementare: exemplu in C++:**

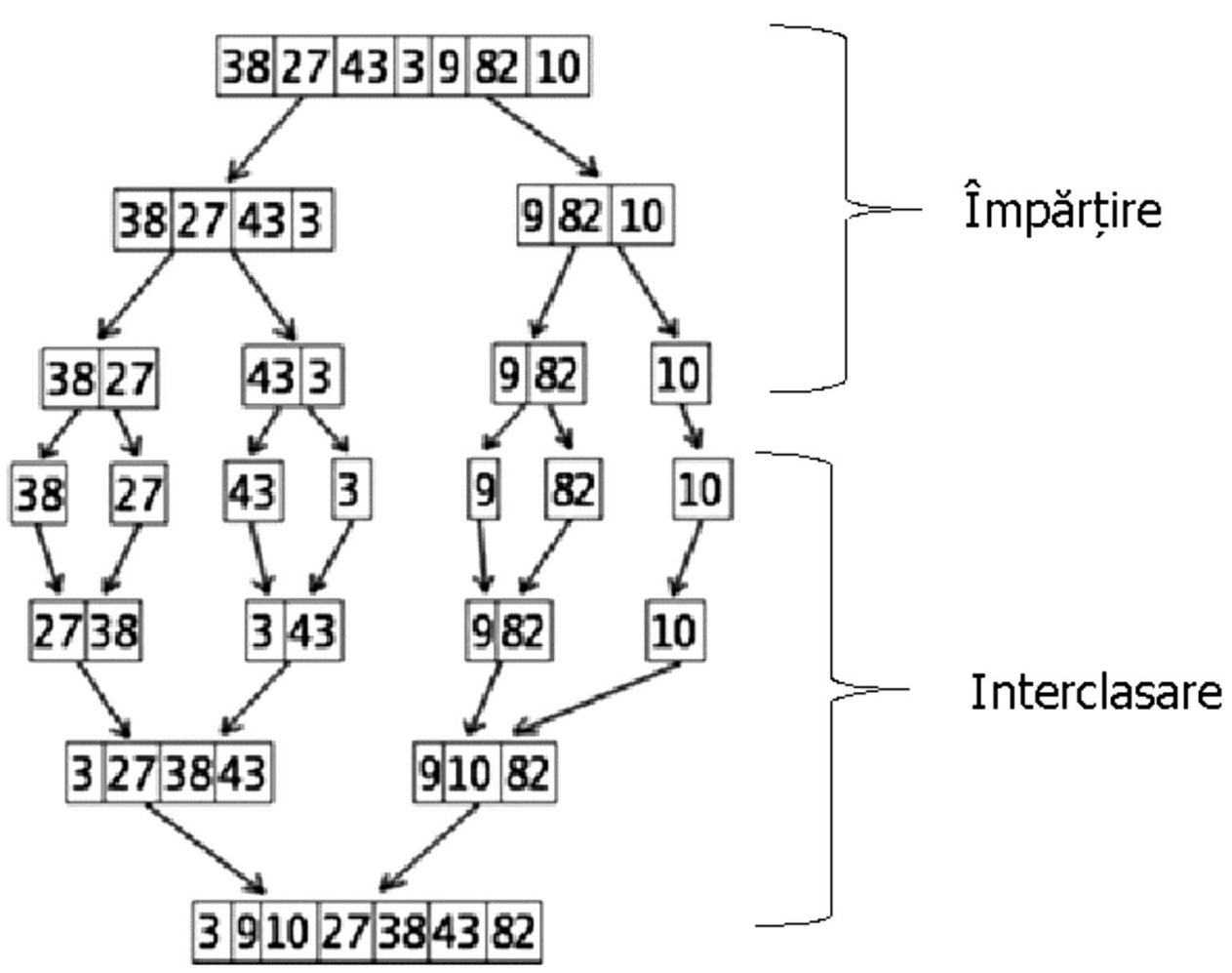
```cpp
// C++ program to sort an array using bucket sort
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

// Function to sort arr[] of size n using bucket sort
void bucketSort(float arr[], int n)
{
    // 1) Create n empty buckets
    vector<float> b[n];

    // 2) Put array elements in different buckets
    for (int i=0; i<n; i++)
    {
       int bi = n*arr[i]; // Index in bucket
       b[bi].push_back(arr[i]);
    }

    // 3) Sort individual buckets
    for (int i=0; i<n; i++)
       sort(b[i].begin(), b[i].end());

    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
          arr[index++] = b[i][j];
}

/* Driver program to test above funtion */
int main()
{
    float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(arr)/sizeof(arr[0]);
    bucketSort(arr, n);

    cout << "Sorted array is \n";
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
    return 0;
}
```

# 4. Metode avansate de sortare

1. Sortare prin interclasare (Merge sort)
2. Sortare rapida (QuickSort)
3. Heapsort

# 4.1. Sortare prin interclasare (Merge sort)

- Inventat de John von Neumann in 1945
- Foloseste tehnica Divide eit Impera, Complx. O(nlogn)



# 4.1. Sortare prin interclasare (Merge sort)

- **Interclasarea**: fie doi vectori v1 si v2 cu elemente sortate; se cere imbinarea lor intr-un v3 astfel incat si v3 sa fie sortat.

- Exemplu:
- v1={3,27,38,43}  i=0
- v2={9,10,82}  j=0  k=0
- dacă v1[i]<v2[j] atunci v3[k]=v1[i]; i++; k++;
- altfel v3[k]=v2[j]; j++; k++;
- Pasul 1. Se compară 3 cu 9 , 3<9 =>v3={3}, i=1
- Pasul 2. Se compară 27 cu 9, 27>9 =>v3={3,9} j=1
- Paul 3. Se compară 10 cu 27, 10<27 => v3={3,9,10}, i=2
- ...
- Ultimul pas

# 4.1. Sortare prin interclasare (Merge sort)

- **Implementare**:

```
Subalgoritm Interclasare(inc,mijl,sf,vector)
   i:=inc; j:=mijl+1; k:=0;//inițializarea indecșilor
   //parcurgerea până la sf. unuia dintre vectori
   cât timp i<=mijl și j<=sf exec.
      dacă vector[i]<vector[j] atunci
         //transfer din prima subsecv. în vectorul rezultat(vect temp)
         temp[k]:=vector[i]; k:=k+1; i:=i+1;
      altfel
        //transfer din a doua  subsecv.în vectorul rezultat(vect temp)
        temp[k]:=vector[j]; k:=k+1; j:=j+1;
      sf. daca
sf. cât timp
```

```
cât timp i<=mijl exec //transferul elementelor rămase în prima subsecv.
 temp[k]:=vector[i]; k:=k+1; i:=i+1;
sf. cât timp
cât timp j<=sf exec //transferul elementelor rămase în a doua subsecv.
 temp[k]:=vector[j]; k:=k+1; j:=j+1;
sf. cât timp
//copiere din temp înapoi în vector
pt. i=inc..sf exec vector[i]:=temp[i-inc];//vector[inc]:=temp[0];
sf. subalgoritm                          //vector[inc+1]:=temp[1];

Acest subalgoritm se apelează în subalgoritmul MergeSort.
```

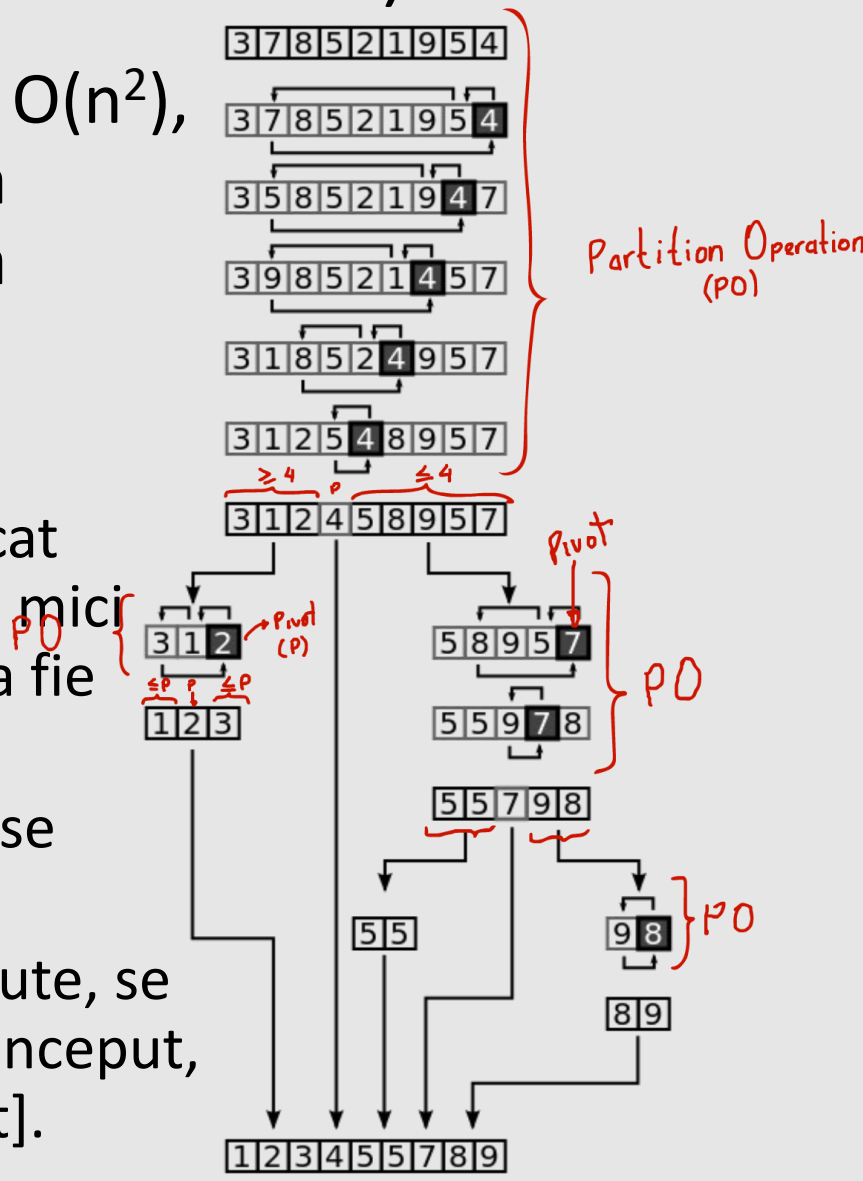
# 4.1. Sortare prin interclasare (Merge sort)

- **Implementare**:

```
Subalgoritm MergeSort(inc,sf,vector)
daca inc>=sf atunci return;
altfel
    mijl:=(inc+sf)/2;
    MergeSort(inc,mijl,vector);
    MergeSort(mijl+1,sf,vector);
    Interclasare(inc,mijl,sf,vector);//slide-ul
sf. daca
sf. subalgoritm
În algoritmul principal MergeSort(0,n-1,v);
```

# 4.2. Sortare rapida (QuickSort)

- Tony Hoare 1960, complexitate $O(n^2)$, însă este mai efficientă datorită implementărilor mai eficiente a buclei interne de repetiții.
- Elimină interclasarea
  - se alege o valoare pivot astfel incat elementele inaintea sa sa fie mai mici decat el si elementele de dupa sa fie mai mari.
  - daca nu exista astfel de element se creeaza unul prin partitionare.
  - se sorteaza cele doua parti obtinute, se apeleaza recursiv QSort pentru [inceput, pozpivot-1] si [pozpivot+1, sfarsit].



# 4.2. Sortare rapida (QuickSort)

- **Implementare:**

```
Subalgoritm QuickSort(inc,sf,vector)
daca inceput<sfarsit atunci
        pozpivot=Partitionare(inc,sf,vector);
        QuickSort(inc,pozpivot-1);
        QuickSort(pozpivot+1,sf);
sf. subalgoritm
```

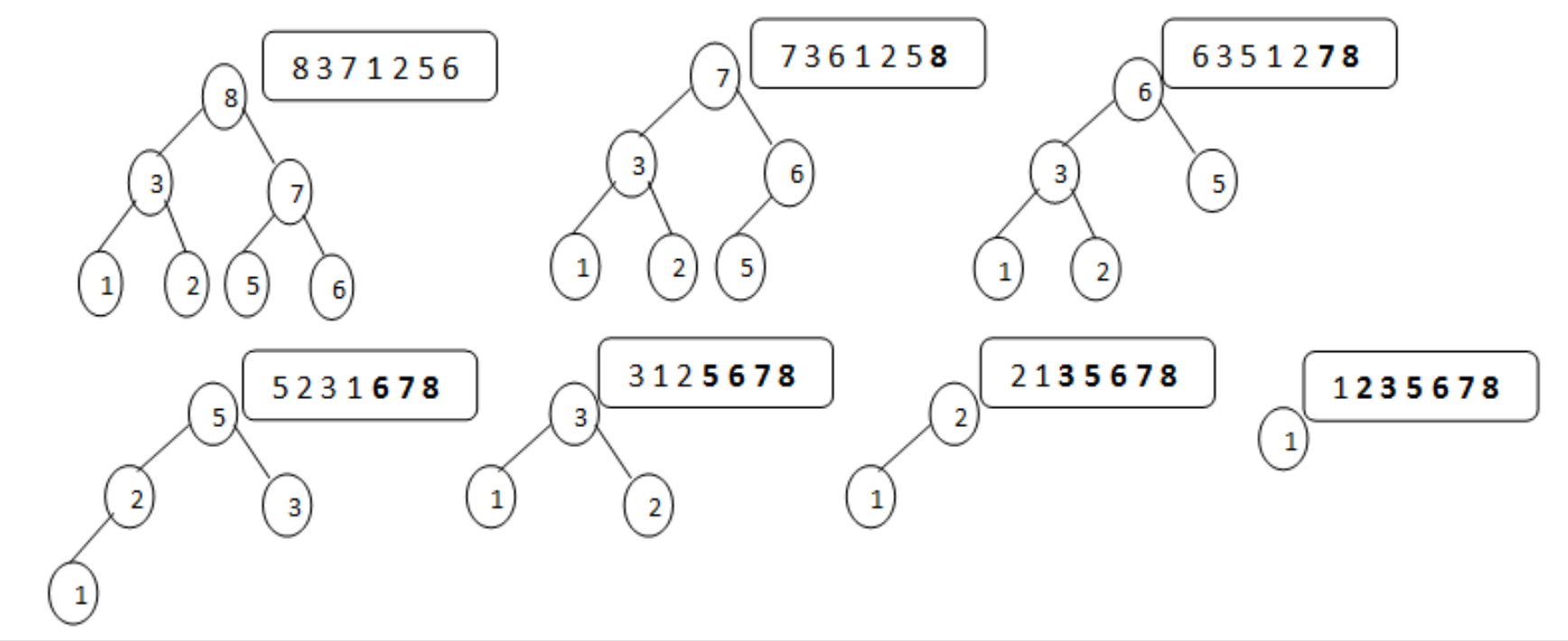
- **Partinionarea se rezolvă prin partiționare Lomuto.**
- **Sau partiționare Hoare…**

# 4.2. Sortare rapida (QuickSort)

- **Implementare: partitionare Lomuto**

```
Subalg. partitionare_Lomuto(inc,sf,vector)
P:=vector[inc];
pozP:=inc;
pt. i:=inc+1..sf exec.
daca vector[i]<P
    PozP:=PozP+1;
    interchimba(vector[i],vector[PozP]);
sf. daca
sf. pt.
interchimba(vector[inc],vector[PozP]);
return pozP;
sf. subalg.
```

# 4.3. HeapSort
# O(nlogn)




https://medium.com/basecs/heapify-all-the-things-with-heap-sort-55ee1c93af82

# 4.3. 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)
  }
}
```

# 5. Probleme sortare

- **Vezi documentul lab_problem.pdf**