

Capitolul 6 - Sortari avansate

Introducere

Limita $O(n^2)$ pentru sortari este mult prea mare și trebuie depășită pentru a reduce timpii de rulare

O metodă generică de a îmbunătății performanța algoritmilor (în acest caz a algoritmilor de sortare) ar fi apelarea algoritmului pe subdiviziuni ale setului de date de intrare (metoda Divide and Conquer)

De la această idee de bază pornesc o serie de algoritmi avansați

Shellsort

Shell Sort este o variantă a algoritmului de **Insertion Sort** care îmbunătățește performanța prin utilizarea unei tehnici numite „sortare pe secvențe”. Algoritmul împarte lista în subliste mai mici, sortând elementele aflate la o anumită distanță („gap”) unul de celălalt. Distanța scade treptat până când ajunge la 1, moment în care lista devine complet sortată.

Ideea principală este că, prin utilizarea acestei distanțe inițiale mari, elementele sunt aduse mai aproape de pozițiile lor finale mult mai rapid decât în cazul unei inserții directe.

Pentru algoritmi de sortare **metoda generală**, de îmbunătățire a performanțelor bazată pe Divide and Conquer ar putea fi:

împarte tabloul a în h subtablouri

pentru i de la 1 la h

sorteaza subtabloul a_i

sorteaza intregul tablou

Dacă h este prea mic, atunci subtablourile a_i pot fi prea mari => ineficiență

Dacă h este prea mare, atunci se creează prea multe subtablouri => ineficiență

-> O soluție ar fi alegerea unui set de valori, în locul unei valori unice

-> Metoda reprezintă o perfectionare a metodei de sortare prin inserție directă

-> În aceasta metoda de sortare, se alege un set de valori pentru h , acesta devenind un tablou, numit și tablou de incrementi

Algoritmul devine:

se determină valorile $h_{t-1} \dots h_0$ pentru divizarea tabloului a în subtablouri
pentru fiecare element h_j de la h_{t-1} la h_0

imparte tabloul a în h_j subtablouri

pentru i de la 1 la h_j

sortează subtabloul a_i

sortează tabloul a

Pentru sortarea subtablourilor algoritmul original folosește sortarea prin inserție, aceasta fiind și cea mai întâlnită variantă pentru Shellsort. Mai rămâne o problemă de rezolvat. Cum se aleg valorile pentru tabloul de incrementi (h)?

Ideea algoritmului este să se compare inițial elementele aflate la o distanță mai mare, apoi elementele din ce în ce mai apropiate, până la o comparare a tuturor elementelor adiacente într-o sortare finală. Astfel elementele tabloului h se vor alege în ordine descrescătoare, elementul final fiind obligatoriu 1.

Care sunt valorile cele mai potrivite pentru h ?

Problema este încă deschisă cercetării. Nu există o soluție care să fie demonstrată ca fiind optimă.

Donald Knuth a demonstrat totuși că și cu doar doi incrementi: $\lceil \frac{16n}{\pi} \rceil^{\frac{1}{3}}$ și 1, Shellsort este mai eficientă decât sortarea prin inserție deoarece este $O(n^{\frac{5}{3}})$ în loc de $O(n^2)$.

Eficiența poate fi îmbunătățită folosind un număr mai mare de incrementi.

Studii empirice arată o eficiență crescută pentru următorul șir de valori pentru h , unde $t = \lfloor \log_2 n \rfloor - 1$:

$$h_{t-1} = 1$$

$$h_i = 3h_{i+1} + 1$$

Ex : $h = \{13, 4, 1\}$

Dacă Shellsort folosește sortarea prin inserție, de ce nu avem $O(n^2)$?

Ne bazăm pe două principii

- Cazul cel mai favorabil pentru sortarea prin inserție este $O(n)$, pentru șiruri gata sortate
- Prinț-o alegere corectă a valorilor incrementilor se evită compararea în repede rânduri a acelorași perechi de elemente

Obținem tablouri parțial sortate, după fiecare pas

Evităm valorile incrementilor care se divid una pe celalătă

De ex. 8,4,2,1 – nu este o alegere eficientă, deoarece se compară în repede rânduri aceleasi perechi de elemente și elementele pare se compară cu cele impare doar în ultimul pas

Pasii algoritmului

1. Începe cu o valoare mare a distanței („gap”) și sortează elementele aflate la această distanță unul de celălalt.
2. Pe măsură ce distanța scade (de obicei la jumătate la fiecare pas), listele de elemente considerate devin mai mici.
3. Când gap-ul ajunge la 1, algoritmul execută o ultimă sortare similară cu Insertion Sort, dar acum elementele sunt deja aproape de pozițiile lor finale.

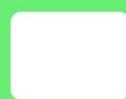
Shellsort

h:	5	3	1
----	---	---	---

a	10	8	6	20	4	3	22	1	0	15	16
Sub-tablourile înainte de sortarea cu h[0]=5	10	-	-	-	-	3	-	-	-	-	16
		8	-	-	-	-	22				
			6	-	-	-	-	1			
				20	-	-	-	-	0		
					4	-	-	-	-	15	
Sub-tablourile după sortarea cu h[0]=5	3	-	-	-	-	10	-	-	-	-	16
		8	-	-	-	-	22				
			1	-	-	-	-	6			
				0	-	-	-	-	20		
					4	-	-	-	-	15	

Dupa sortarea anterioara	3	8	1	0	4	10	22	6	20	15	16
Sub-tablourile inainte de sortarea cu h[1]=3	3	-	-	0	-	-	22	-	-	15	
		8	-	-	4	-	-	6	-	-	16
			1	-	-	10	-	-	20		
Sub-tablourile dupa sortarea cu h[1]=3	0	-	-	3	-	-	15	-	-	22	
		4	-	-	6	-	-	8	-	-	16
			1	-	-	10	-	-	20		
Dupa sortarea anterioara	0	4	1	3	6	10	15	8	20	22	16
Sub-tablourile dupa sortarea cu h[2]=1	0	1	3	4	6	8	10	15	16	20	22

12 | 34 | 54 | 2 | 3



Temp

Start with gap = n/2 (2 in this case)

One by one select elements to the right of gap and place them at their appropriate position.



54
Temp

Elements left of 54 are already smaller, so no change.

One by one select elements to the right of gap and place them at their appropriate position.

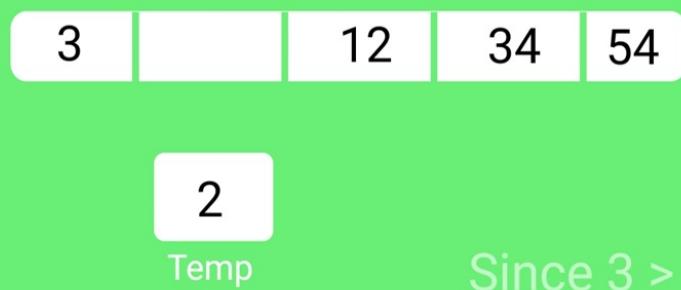


2
Temp

Compare 2 with arr[3-2] = 34 and shift it to arr[gap+1 = 3].



Compare 2 with arr[3-2] = 34 and shift it to arr[gap+1 = 3].



Now gap reduces to 1(n/4).

Select all elements starting from arr[1] and compare them with elements within the distance of gap.

2

3

12

34

54

Now gap reduces to 0

Sorting stops and array is sorted.

Implementarea algoritmului

```
// T este dimensiunea
//tabloului de incrementi
#define T 3
int h[T];
void generare_incrementi(int h[], int t) {
    int i; //generare tablou de incrementi
    h[T - 1] = 1;
    for (i = T - 2; i >=0; i--) {
        h[i] = 3 * h[i + 1] + 1;
    }
}

void shell_sort(tip_element a[], int n) {
    int i, j, k, hCnt, H;
    tip_element tmp;
    generare_incrementi(h, T);
    //pentru fiecare increment
    for (i = 0; i < T; i++) {
        H = h[i]; //incrementul curent
        //pentru fiecare subtablou
        for (hCnt = 0; hCnt < H; hCnt++) {
            //insetion_sort pentru subtablou
            for (j = H+hCnt; j < n;j+=H){
                //se selecteaza elementele cu pasul H
                tmp = a[j];
                for (k = j; (k-H>=0) && tmp.cheie < a[k - H].cheie; k-=H)
                    a[k] = a[k - H];
            }
        }
    }
}
```

```

        a[k] = tmp;
    }
}
}

//geeks for geeks
// C++ implementation of Shell Sort
#include <iostream>
using namespace std;

/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
    // Start with a big gap, then reduce the gap
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in gapped order
        // keep adding one more element until the entire array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
            // add a[i] to the elements that have been gap sorted
            // save a[i] in temp and make a hole at position i
            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct
            // location for a[i] is found
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            // put temp (the original a[i]) in its correct location
            arr[j] = temp;
        }
    }
    return 0;
}

void printArray(int arr[], int n)
{
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}

int main()
{

```

```

int arr[] = {12, 34, 54, 2, 3}, i;
int n = sizeof(arr)/sizeof(arr[0]);

cout << "Array before sorting: \n";
printArray(arr, n);

shellSort(arr, n);

cout << "\nArray after sorting: \n";
printArray(arr, n);

return 0;
}

```

Analiza ShellSort

Time Complexity: Time complexity of the above implementation of Shell sort is $O(n^2)$. In the above implementation, the gap is reduced by half in every iteration. There are many other ways to reduce gaps which leads to better time complexity. See [this](#) for more details.

Worst Case Complexity

The worst-case complexity for shell sort is $O(n^2)$

Best Case Complexity

When the given array list is already sorted the total count of comparisons of each interval is equal to the size of the given array.

So best case complexity is $\Omega(n \log(n))$

Average Case Complexity

The Average Case Complexity: $O(n * \log n) \sim O(n^{1.25})$

Space Complexity

The space complexity of the shell sort is $O(1)$.

Avantaje

-> O sortare mai eficientă decât cele din categoria sortărilor directe (simple)

Dezavantaje

-> Performanță mai slabă față de sortările avansate studiate în continuare
-> nu este stabil

Quicksort

-> cea mai rapida sortare "in situ" pentru caz general, considerand cazul mediu

Este cea mai răspândită metodă de sortare, implementată în general în bibliotecile standard (qsort)

Este numită și sortare **prin partitie** pentru că se bazează pe tehnica **Divide and Conquer**

Shellsort împărtea tabloul în subtablouri pe care le sorta separat, ca mai apoi să reconsideră tabloul întreg, care era din nou împărțit în alte subtablouri ce urmau să fie sortate individual, s.a.m.d.

Quicksort împarte într-un mod mult mai eficient tabloul.

Tabloul este împărțit în două subtablouri, în funcție de valoarea elementelor raportată la un element ales, numit **pivot**.

Pasii algoritmului

1. Se alege un **pivot** din lista nesortată. Există mai multe strategii pentru a alege pivotul, cele mai comune fiind primul element, ultimul element, sau un element ales aleatoriu.
2. Lista este apoi împărțită astfel încât toate elementele mai mici decât pivotul să fie plasate în stânga acestuia, iar cele mai mari în dreapta.
3. Se aplică recursiv algoritmul pe sublistele din stânga și din dreapta pivotului, până când fiecare sublistă conține doar un singur element sau niciunul (cazul în care este deja sortată).
4. În final, listele rezultate sunt combinate, formând lista sortată.

Principiul de funcționare:

În tablou se formează două subtablouri în funcție de valoarea elementelor raportată la pivot în felul următor: primul subtablou conținând elemente mai mici sau egale cu valoarea pivotului și al doilea subtablou conținând elemente egale sau mai mari decât valoarea pivotului. Acest proces poartă numele de **partitie**

Se apelează algoritmul pe fiecare subtablou în parte. Acestea la rândul lor se partionează în două subtablouri fiecare, față de un pivot nou ales, algoritmul continuând într-un mod recursiv până se ajunge la subtablouri de un singur element

Algoritmul procedurii de partitōnare:

Fie x un element oarecare al tabloului de sortat a_1, \dots, a_n .

Se parurge tabloul de la stānga spre dreapta pānă se găsește primul element $a_i > x$.

În continuare se parurge tabloul de la dreapta spre stānga pānă se găsește primul element $a_j < x$.

Se interschimbă între ele elementele a_i și a_j .

Se continuă parcurgerea tabloului de la stānga respectiv de la dreapta, din punctele în care s-a ajuns anterior, pānă se găsesc alte două elemente care se interschimbă, ș.a.m.d.

Procesul se termină când cele două parcurgeri se "întâlnesc" undeva în interiorul tabloului.

Efectul final este acela că sirul inițial este partitōnat într-o partitōie stānga cu chei mai mici sau egale cu x și o partitōie dreapta cu chei mai mari sau egale cu x .

În continuare, cu ajutorul partitōnării, **sortarea** se realizează simplu:

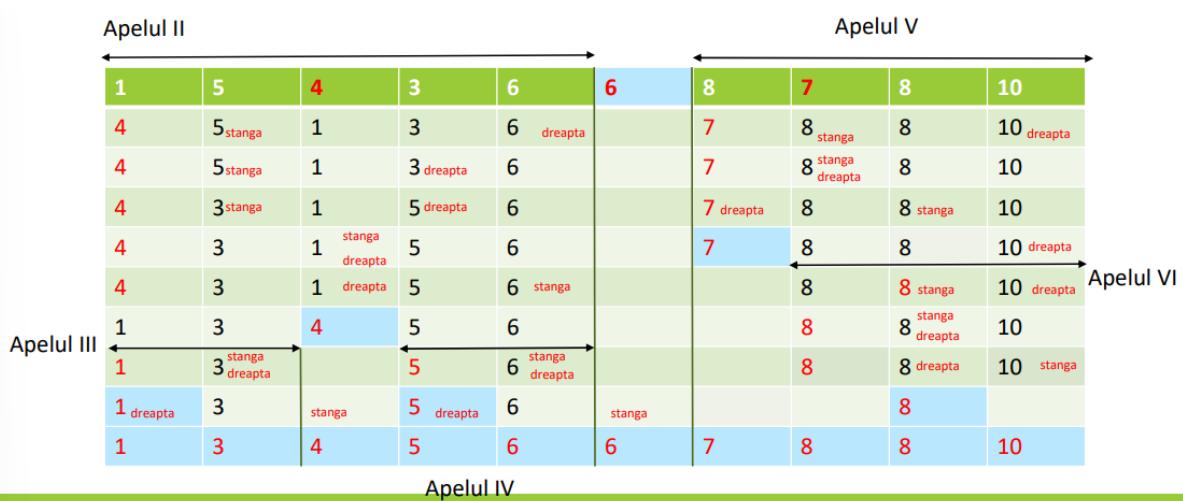
După o primă partitōnare a secvenței de elemente se aplică același procedeu celor două partitōii rezultate.

Apoi celor patru partitōii ale acestora, ș.a.m.d.

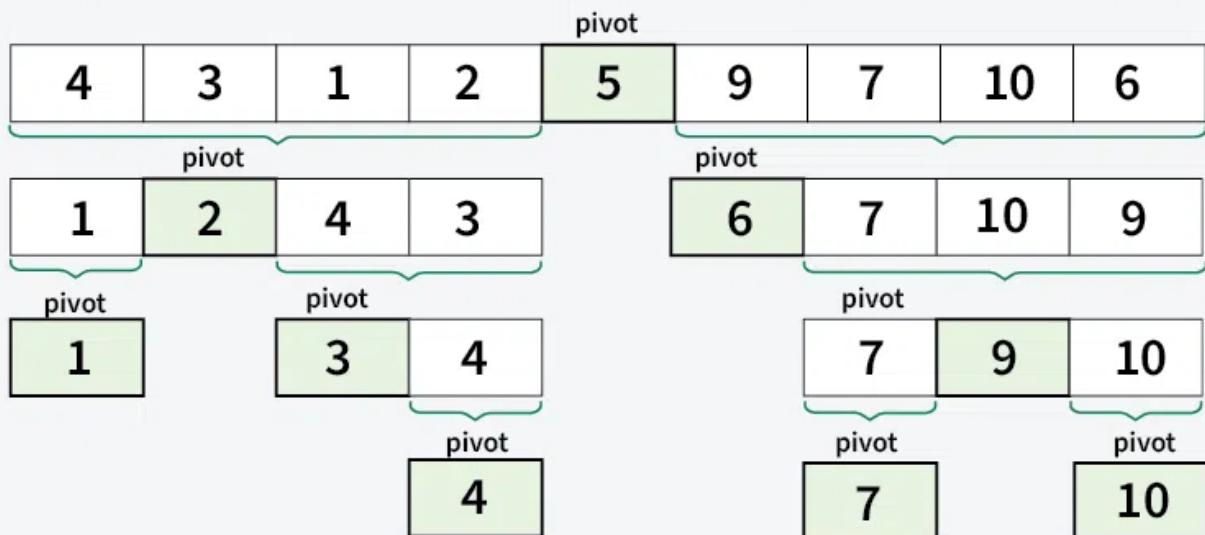
Procesul se termină când fiecare partitōie se reduce la un singur element.

Apelul I

8	5	4	7	6	1	6	3	8	10
6	5 _{stānga}	4	7	8	1	6	3	8	10 _{dreapta}
6	5	4	7 _{stānga}	8	1	6	3 _{dreapta}	8	10
6	5	4	3	8 _{stānga}	1	6 _{dreapta}	7	8	10
6	5	4	3	6 _{stānga}	1	8 _{dreapta}	7	8	10
6	5	4	3	6	1 _{stānga dreapta}	8	7	8	10
1	5	4	3	6	6	8	7	8	10

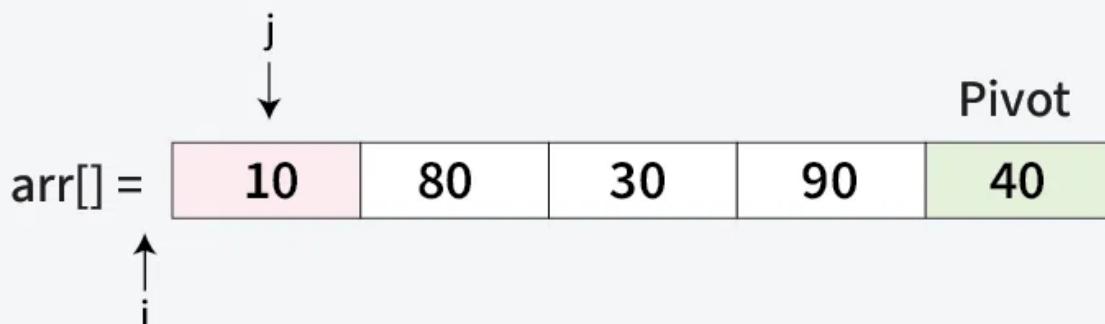


Here, we have represented the recursive call after each partitioning step of the array.



01
Step

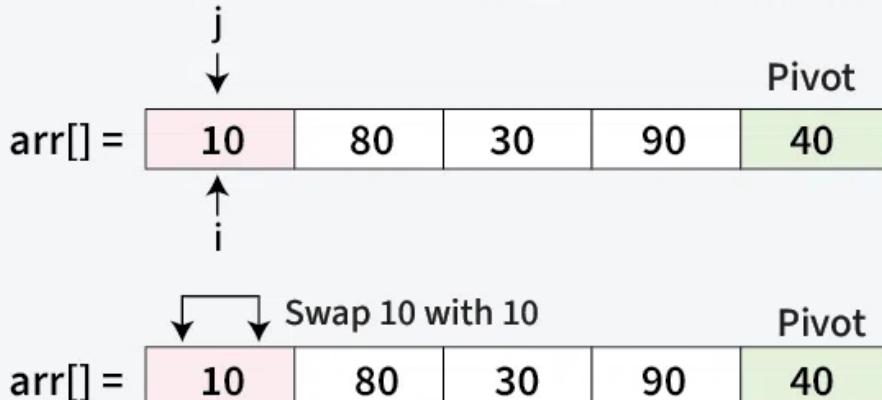
Pivot Selection: The last element arr[4] = 40 is chosen as the pivot.
Initial Pointers: i = -1 and j = 0.



Quick sort

02
Step

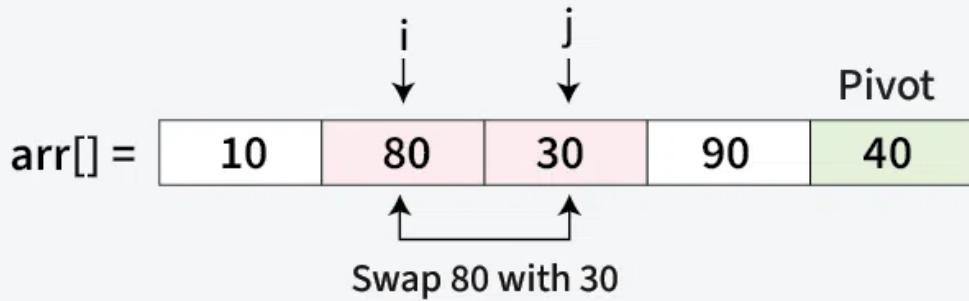
Since, arr[j] < pivot (10 < 40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1



Quick sort

04
Step

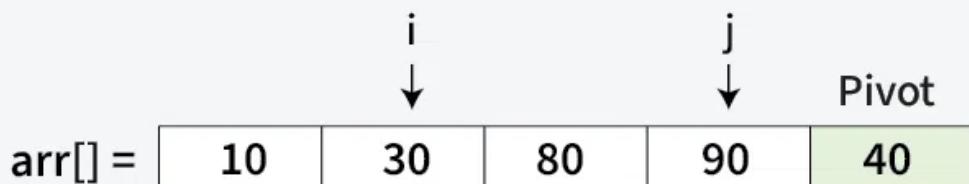
Since, $\text{arr}[j] < \text{pivot}$ ($30 < 40$)
Increment i by 1 and swap $\text{arr}[i]$ with $\text{arr}[j]$. Increment j by 1



Quick sort

05
Step

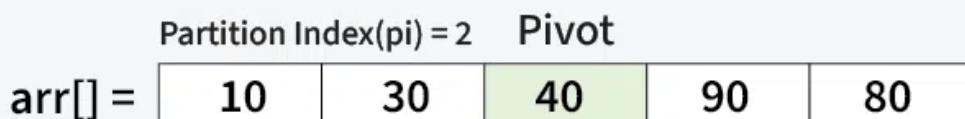
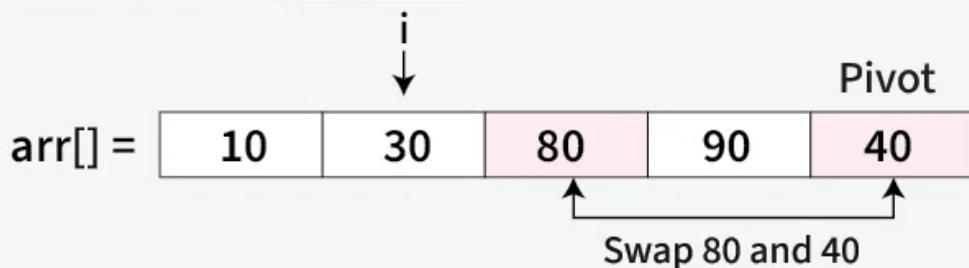
Since, $\text{arr}[j] > \text{pivot}$ ($90 < 40$)
No swap needed. Increment j by 1



Quick sort

06
Step

Since traversal of j has ended. Now move pivot to its correct position, Swap $\text{arr}[i + 1] = \text{arr}[2] = 80$ with $\text{arr}[4] = 40$.



Quick sort

Implementare

Pseudocod:

```
quicksort (vector[])
    daca lungime(vector) > 1
        alege pivot;
        cat timp mai sunt elemente in vector
            include elemental fie in subtabloul1 = { el: el <= pivot};
            sau in subtabloul2={el: el>=pivot};
        quicksort (subtablou1);
        quicksort (subtablou2);
```

Problema alegării pivotului nu este simplă

În primul rând se dorește o împărțire cât mai egală a celor două subtablouri din punct de vedere al numărului de elemente

Dacă se alege unul din capetele intervalului, împărțirea este dezechilibrată și duce la o degradare a performanțelor din punct de vedere al timpului de execuție

În funcție de metoda de alegere a pivotului avem mai multe variante de implementare pentru Quicksort

```
void quicksort(tip_element a[], int prim, int ultim) {
    int stanga = prim + 1;
    int dreapta = ultim;
    //alegere pivot
    swap(&a[prim], &a[(prim + ultim) / 2]);
    //mutare pivot pe prima pozitie tip_element
    pivot = a[prim];
    while (stanga <= dreapta) //partitionare
    {
        while (a[stanga].cheie < pivot.cheie)
            stanga++;
        while (pivot.cheie < a[dreapta].cheie)
            dreapta--;
        if (stanga < dreapta)
            swap(&a[stanga++], &a[dreapta--]);
        else stanga++;
    } //mutare pivot la locul sau final
    swap(&a[dreapta], &a[prim]);
    //apelurile recursive
    if (prim < dreapta - 1)
        quicksort(a, prim, dreapta - 1);
    if (dreapta + 1 < ultim)
```

```

        quicksort(a, dreapta + 1, ultim);
    }

#include <stdio.h>

void swap(int* a, int* b);

// Partition function
int partition(int arr[], int low, int high) {

    // Choose the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[low..high] and move all smaller
    // elements to the left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Move pivot after smaller elements and
    // return its position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void swap(int* a, int* b) {
    int t = *a;

```

```

*a = *b;
*b = t;
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}

```

Ca orice program implementat într-o variantă recursivă, și funcția propusă anterior poate fi implementat și într-o variantă nerecursivă

Mai multe despre tehniciile de implementare a funcțiilor recursive în variante nerecursive, veți găsi în capitolul de recursivitate

Putem avea mai multe variante de implementare a algoritmului de sortare Quicksort în funcție de modul în care alegem pivotul și în funcție de modul în care facem partitōnarea

Analiza algoritmului

Analiza algoritmului Quicksort

Cel mai **favorabil** caz îl avem când pivotul împarte tabloul în subtablouri de aproximativ $n/2$ elemente

$$C_{min} = n + 2 \frac{n}{2} + 4 \frac{n}{4} + \dots + n \frac{n}{n} = n(\log n + 1) = O(n \log n)$$

$$M_{min} = O(n \log n)$$

Cel mai **defavorabil** caz îl avem când pivotul are valoarea maximă sau minimă, astfel fiecare pas va partaja secvența formată din n elemente, într-o partiție cu $n - 1$ elemente și o partiție cu un singur element.

$$C_{max} = n + C(n - 1) = n + n - 1 + C(n - 2) = \dots = n + n - 1 + n - 2..+1 = \frac{n(n + 1)}{2} = O(n^2)$$

$$M_{max} = O(n^2)$$

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\Theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space: $O(n)$, due to recursive call stack

Avantaje

0 sortare eficientă pentru caz general $O(n \log n)$

0 sortare "in situ"

Dezavantaje

Un algoritm recursiv, se folosește de stivă

Performanță slabă pentru cazul cel mai defavorabil $O(n^2)$

HeapSort

Face parte tot din categoria sortărilor avansate, dar față de Quicksort, acest algoritm nu se bazează pe recursivitate

Algoritmul se bazează pe un principiu asemănător cu algoritmul de sortare prin selecție (selection sort) și anume acela de a determina elementul cu cheia minimă, respectiv maximă

Acest algoritm selectează algoritmul cu cheia maximă și îl împinge spre finalul tabloului, spre deosebire de sortarea prin selecție care căuta elementul cu cheia minimă pentru a-l aduce spre începutul tabloului

O altă diferență semnificativă este că Heapsort aduce o îmbunătățire față de algoritmul de sortare prin selecție, prin faptul că la fiecare trecere se vor reține mai multe informații, nu doar elementul cu cheia minimă, ca în cazul sortării prin selecție

-> imbunatatire la selection sort

Heapsort se folosește de conceptul de ansamblu (heap), după care este și denumit

Un ansamblu este un arbore binar cu următoarele proprietăți (max-heap):

- Valoarea cheii fiecărui nod este mai mare sau egală cu valorile cheilor fiilor săi
- Arborele este perfect echilibrat, nodurile frunză de pe ultimul nivel fiind situate în cele mai din stânga poziții

Dacă vorbim despre o relație inversă: valoarea cheii fiecărui nod este mai mică sau egală cu valorile cheilor fiilor săi, acesta se numește min-heap

Altfel spus, un ansamblu (max-heap) este definit ca o secvență de chei h_0, h_1, \dots, h_{n-1} care se bucură de proprietățile:

$$\begin{aligned} h_i &\geq h_{2i+1} \\ h_i &\geq h_{2i+2} \end{aligned}$$

pentru toți $i = 0, \dots, (n-1)/2-1$

Pentru min-heap, inegalitățile vor fi invers ($h_i \leq h_{2i+1}$ și $h_i \leq h_{2i+2}$)

Un **ansamblu** poate fi asimilat cu un **arbore binar parțial ordonat** și reprezentat printr-un **tablou**

Pentru ca implementarea algoritmului să fie eficientă și din punct de vedere al spațiului, nu doar al timpului, scopul este să nu folosim alte structuri suplimentare, ci să alegem o variantă de implementare "in situ"

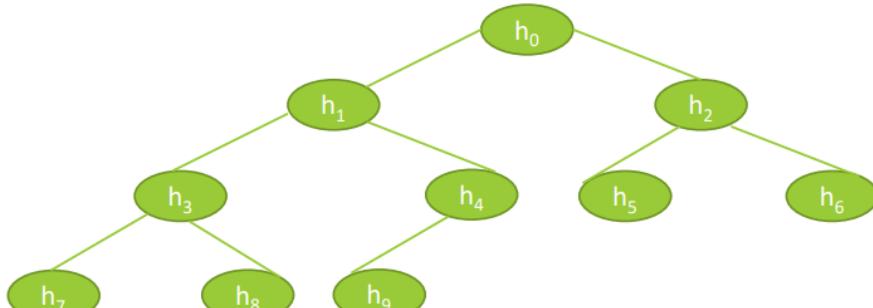
Atfel, nu vor fi folosite nici structuri suplimentare de tip arbore, nici alte tablouri

Cum ne vom folosi de structura arborescentă în acest caz?

Pasii algoritmului

1. **Construirea heap-ului:** Transformăm lista într-un max-heap. În acest pas, toate elementele respectă proprietatea de max-heap, unde fiecare nod părinte este mai mare decât copiii săi.
2. **Sortarea heap-ului:** Odată ce lista este transformată într-un max-heap, interzicem rădăcina (elementul cel mai mare) cu ultimul element al heap-ului și reducem dimensiunea heap-ului cu 1. Apoi reconstruim heap-ul pentru a respecta din nou proprietatea de max-heap.
3. **Repetăm procesul** până când toate elementele sunt extrase din heap, obținând o listă sortată

Tabloul de sortat va fi privit ca un arbore, pe care trebuie să îl ducem la o structură de tip ansamblu



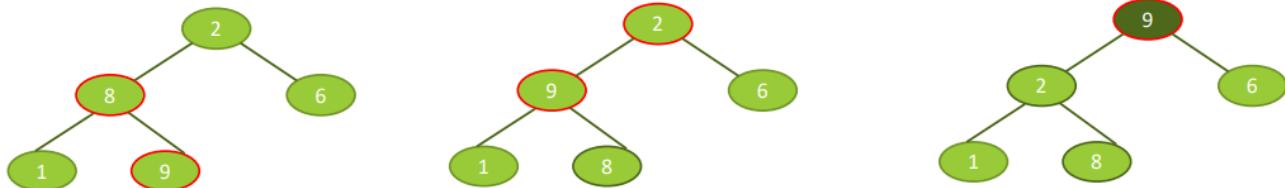
0	1	2	3	4	5	6	7	8	9
h ₀	h ₁	h ₂	h ₃	h ₄	h ₅	h ₆	h ₇	h ₈	h ₉

Elementele dintr-un ansamblu nu sunt toate ordonate

Ştim doar că elementul cu cheia maximă este rădăcina arborelui (pentru max-heap) și că pentru fiecare nod, cheile fiilor săi sunt mai mici sau egale cu cea a părintelui

Heapsort pornește astfel de la un ansamblu, pune elementul cu cheia cea mai mare la sfârșit, îl ignoră și reface ansamblul care acum are cu un element mai puțin

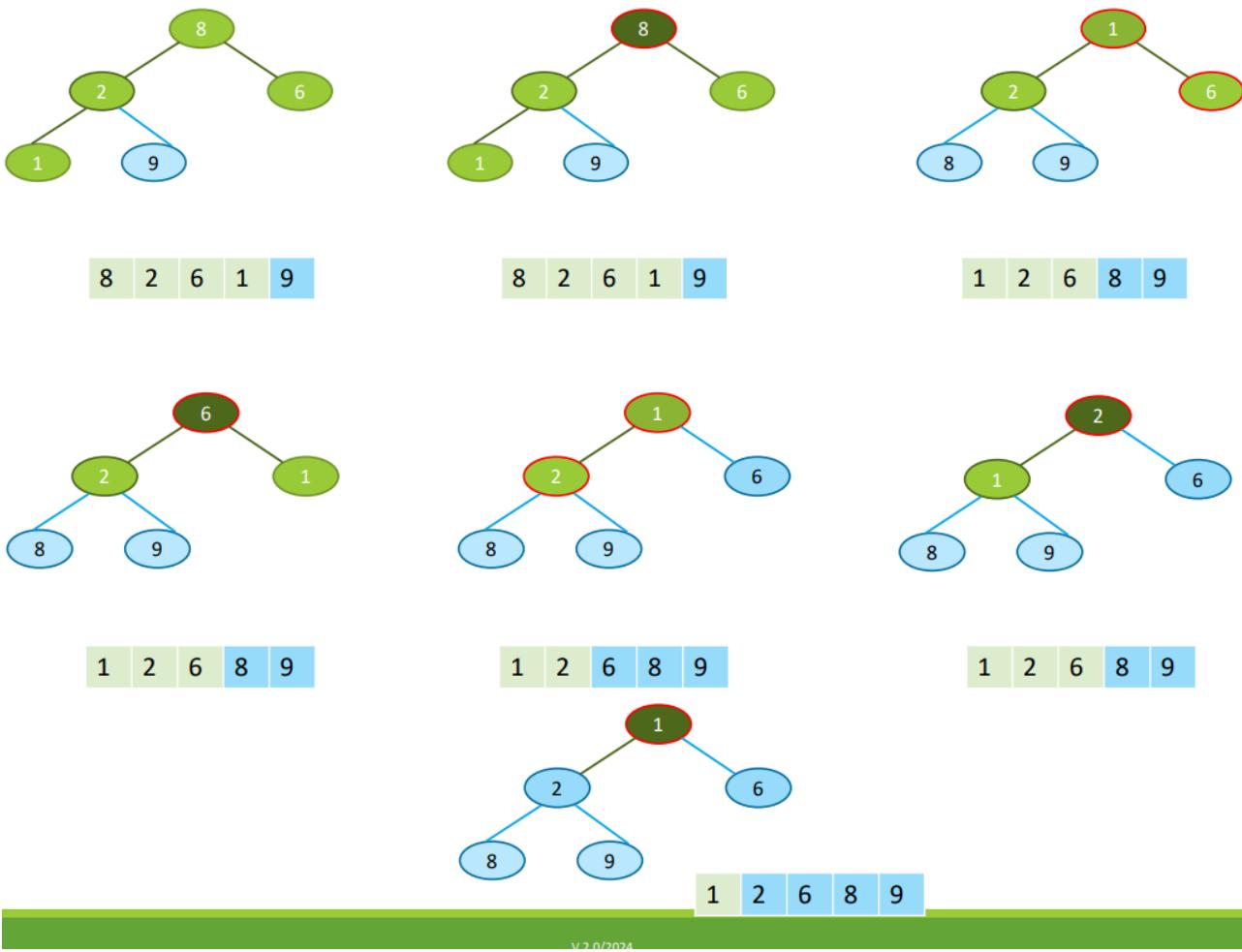
Atfel în fiecare tură un element ajunge în poziția sa finală și ansamblul devine mai mic cu un element



2	8	6	1	9
---	---	---	---	---

2	9	6	1	8
---	---	---	---	---

9	2	6	1	8
---	---	---	---	---



Implementare

Pseudocod:

heapsort (a[], n)

transforma datele intr-un ansamblu

pentru $i = n-1 ; i \geq 1 ; i--$

schimba elemental radacina cu elemental de pe pozitia i

refa ansamblul pentru $a[0], \dots, a[i-1]$

Pentru formarea ansamblului se va folosi o metodă de tip bottom-up, "in situ", propusă de R.W.Floyd:

- Se consideră un tablou h_0, \dots, h_{n-1} care conține cele n elemente din care se va construi ansamblul.
- În mod evident, elementele $h_{\lfloor n/2 \rfloor}, \dots, h_{n-1}$ formează deja un **ansamblu** deoarece **nu** există nici o pereche de indici i și j care să satisfacă relația $j=2*i+1$ (sau $j=2*i+2$).
- Aceste elemente formează cea ce poate fi considerat drept **șirul de bază** al **ansamblului** asociat.
- În continuare, ansamblul $h_{\lfloor n/2 \rfloor}, \dots, h_{n-1}$ este **extins spre stânga**, la fiecare pas cu câte un element, introdus în vîrful ansamblului și deplasat până la locul său.

Prin urmare, considerând că tabloul inițial este memorat în h, procesul de generare "in situ" al unui **ansamblu** poate fi descris prin secvența următoare:

Pseudocod:

Floyd(a[])

```
stanga = n/2-1  
pentru i = stanga; i>=0; i--  
    stanga=stanga-1  
    deplasare (a,i,n-1)
```

```
void deplasare(tip_element a[], int stanga, int dreapta) {  
    int fiu = 2 * stanga + 1;  
  
    while (fiu <= dreapta) {  
        //daca al doilea fiu are cheia cea mai mare  
        if (fiu < dreapta && a[fiu].cheie < a[fiu + 1].cheie)  
            fiu++;  
        //retinem al doilea fiu  
        if (a[stanga].cheie < a[fiu].cheie) {  
            //schimba parinte cu fiu  
            swap(&a[stanga], &a[fiu]);  
            //si deplaseaza in jos  
            stanga = fiu;  
            fiu = 2 * stanga + 1;  
        }  
        else  
            fiu = dreapta + 1;  
    }  
}  
  
void heapsort(tip_element a[], int n) {  
    int i; //algoritmul lui Floyd  
    for (i = n / 2 - 1; i >= 0; i--)  
        //se creaza ansamblul  
        deplasare(a, i, n - 1);  
        //extragerea maximului si refacerea ansamblului  
    for (i = n-1; i >= 1; i--) {  
        swap(&a[0], &a[i]);  
        //mutare element maxim pe pozitia a[i]  
        deplasare(a, 0, i - 1);  
        //se reface proprietatea de ansamblu  
    }  
}
```

Analiza algoritmului

Analiza algoritmului Heapsort:

La prima vedere nu rezultă în mod evident faptul că această metodă conduce la rezultate bune

Analiza detaliată a performanței metodei heapsort contrazice însă această părere

La **faza de construcție a ansamblului** sunt necesari $n/2$ pași de deplasare

În **fiecare pas** se mută elemente de-a lungul a respectiv $\log(n/2), \log(n/2 + 1), \dots, \log(n-1)$ poziții, (în cel mai defavorabil caz), unde logaritmul se ia în baza 2 și se trunchiază la prima valoare întreagă

În continuare, **faza de sortare** necesită $n-1$ deplasări fiecare cu cel mult respectiv $\log(n-2), \log(n-1), \dots, 1$ mișcări.

În plus mai sunt necesare $3 \cdot (n-1)$ mișcări pentru a **așeza** elementele sortate în ordine.

Toate acestea dovedesc că în cel mai defavorabil caz, tehnica **heapsort** are nevoie de un număr de pași de ordinul $O(n \cdot \log n)$

$$M_{\max} = O(n) + O(n \log n) + 3(n - 1) = O(n \log n)$$
$$C_{\max} = O(n \log n)$$

-> la orice tip de date, heapsort are prea mulți pași

În ceea ce privește cazul cel mai **favorabil**, acesta este reprezentat de un tablou cu elemente identice

În acest caz deplasare are $n/2$ pași și nu avem mutări de elemente

În ceea ce privește a doua etapă, heapsort efectuează $n-1$ mutări ale rădăcinii la locul potrivit

În ceea ce privește comparațiile avem n în prima fază și $2(n-1)$ în a doua, în cazul unui tablou cu elemente **identice**

$$C_{\min} = M_{\min} = O(n)$$

Pentru tablouri cu elemente diferite avem:

$$C_{\min} = M_{\min} = O(n \log n)$$

Avantaje

-> Are complexitate $O(n \log n)$ chiar și în cel mai rău caz, ceea ce îl face o alegere bună pentru seturi mari de date.

-> Este *in situ* și nu necesită memorie suplimentară, în afară de câteva variabile temporare.

Dezavantaje

-> Nu este stabil (ordinea relativă a elementelor egale poate fi schimbată).

-> Este mai lent decât Quick Sort în practică pentru majoritatea datelor, datorită costului ridicat pentru construirea și menținerea heap-ului.

Binsort si counting sort

În general algoritmii de sortare bazați pe metode avansate au nevoie de $O(n \cdot \log n)$ pași pentru a sorta n elemente

Trebuie precizat însă faptul că acest lucru este valabil în situația în care:

Nu există nici o altă informație suplimentară referitoare la chei, decât faptul că pe mulțimea acestora este definită o **relație de ordonare**, prin intermediul căreia se poate preciza dacă valoarea unei chei este mai **mică** respectiv mai **mare** decât o alta

După cum se va vedea în continuare, sortarea se poate face și **mai rapid** decât în limitele performanței $O(n \cdot \log n)$, dacă:

- Există și **alte informații** referitoare la **cheile** care urmează a fi sortate
- Se **renunță** la constrângerea de sortare "*in situ*"

->>> BINSORT

Spre exemplu:

Se cere să se sorteze un set de n chei de tip întreg, ale căror **valori sunt unice și aparțin intervalului de la 0 la $n-1$** .

Dacă a și b sunt **tablouri** cu câte n elemente, a conținând cheile care urmează a fi sortate, atunci sortarea se poate realiza direct în tabloul b , într-o singură trecere (o sortare liniară), conform secvenței următoare:

```
void sortare_liniara(tip_element a[], tip_element b[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        b[a[i].cheie] = a[i];
}
```

Principiul metodei:

- Se determină locul elementului $a[i]$ și se plasează elementul exact la locul potrivit în tabloul b
- Întregul ciclu necesită $O(n)$ pași.

Rezultatul este însă corect numai în cazul în care există **un singur element** cu cheia x , pentru fiecare valoare cuprinsă între $[0, n-1]$.

Un al doilea element cu aceeași cheie va fi introdus tot în $b[x]$ distrugând elementul anterior.

Acest tip de sortare poate fi realizat și "*in situ*"

Astfel, fiind dat tabloul a de dimensiune n , ale cărui elemente au respectiv cheile $0, \dots, n-1$, se balează pe rând elementele sale (bucla **for** exterioară)

Dacă elementul $a[i]$ are cheia j , atunci se realizează interschimbarea lui $a[i]$ cu $a[j]$.

Fiecare interschimbare plasează elementul aflat în locația i exact la locul său în tabloul ordonat, fiind necesare în cel mai rău caz $3 \cdot n$ mișcări pentru întreg procesul de sortare

---> COUNTING SORT

Problema se formulează astfel:

- Se cere să se sorteze un tablou cu n articole ale căror chei sunt cuprinse în intervalul $[0, m-1]$.
- Dacă m nu este prea mare pentru rezolvarea problemei poate fi utilizat algoritmul de "**determinare a distribuției cheilor**".

Pasii algoritmului

---> BINSORT

1. Creează un număr de buckets goale (de obicei egale ca interval).
2. Parcurge lista și distribuie fiecare element în bucket-ul corespunzător (în funcție de valoare).
3. Sortează fiecare bucket individual (de obicei cu Insertion Sort sau Quick Sort).
4. Concatenază toate bucket-urile sortate pentru a obține lista finală sortată.

index	0	1	2	3	4	5
	2	0	1	5	3	4
	1	0	2	5	3	4
0	1	2	5	3	4	
0	1	2	5	3	4	
0	1	2	5	3	4	
0	1	2	5	3	4	
0	1	2	5	3	4	
0	1	2	5	3	4	
0	1	2	5	3	4	
0	1	2	5	3	4	

Tehnica sortării este simplă:

- Se examinează fiecare element de sortat
- Se introduce în bin-ul corespunzător valorii cheii
- În cazul exemplului sortării liniare, bin-urile sunt chiar elementele tabloului b , unde $b[i]$ este binul cheii având valoarea i
- În cazul exemplului anterior de binsort, bin-urile sunt chiar elementele tabloului a după reașezare

---> COUNTING SORT

1. Creează un array de frecvențe, unde fiecare element din array reprezintă numărul de apariții al valorii corespunzătoare.
2. Calculează array-ul cumulativ de frecvențe pentru a determina pozițiile fiecărui element în lista sortată.

3. Construieste lista sortată folosind array-ul de frecvențe.

Idea algoritmului este următoarea:

1. Se contorizează într-o primă trecere **numărul de chei** pentru fiecare valoare de cheie care apare în tabloul a.
 2. Se ajustează valorile **contoarelor**.
 3. Într-o a doua trecere, utilizând aceste conțoare, se **mută** direct articolele în poziția lor ordonată în tabloul b.

Tabloul a													
0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	0	2	0	3	0	1	1	0	3	3	0

index

Tabloul numar

0	1	2	3
0	0	0	0

Initializare

Tabloul numar

0	1	2	3
6	10	11	14

Ajustare valori

Tabloul numar

0	1	2	3
6	4	1	3

Contorizare valori

Tabloul numar

0	1	2	3
5	10	11	14

i

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

A horizontal line representing an array of 14 elements (index 0 to 13). A green shaded segment highlights the elements from index 5 to index 8, inclusive. The number '0' is written below the line at index 5, indicating the start of the slice.

0	1	2	3
5	10	11	13

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

..... to be continued

Tabloul numar

0	1	2	3
0	6	10	11

Tabloul a

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	1	1	0	2	0	3	0	1	1	0	3	3	0

Tabloul b

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	0	0	0	0	0	1	1	1	1	2	3	3	3

Implementare

--> BINSORT

```
void swap(tip_element *el1, tip_element *el2) {
    tip_element tmp;
    tmp = *el1;
    *el1 = *el2;
    *el2 = tmp;
}

void binsort(tip_element a[], int n) {
    int i;
    for (i = 0; i < n; i++)
        while (a[i].cheie != i)
            //daca elementul a[i] nu are cheia i
            //interschimba elementele
            swap(&a[i], &a[a[i].cheie]);
}
```

--> COUNTING SORT

```
tip_cheie numar[m];
void countingsort(tip_element a[], tip_element b[], int n) {
    int i, j;
    for (j = 1; j <= m - 1; j++)
        numar[j] = 0;
    //initializare tablou contorizare distributie elemente
    for (i = 1; i <= n; i++)
        numar[a[i - 1].cheie] = numar[a[i - 1].cheie] + 1;
    //numararea valorilor
    for (j = 1; j <= m - 1; j++)
        numar[j] = numar[j - 1] + numar[j];
    //ajustare valori contoare
```

```

        for (i = n; i >= 1; i--) {
            //mutare elemente pe pozitia ordonata in tabloul temporar b
            b[numar[a[i - 1].cheie] - 1] = a[i - 1];
            numar[a[i - 1].cheie] = numar[a[i - 1].cheie] - 1;
        }
        //copiere elemente in tabloul initial
        for (i = 1; i <= n; i++)
            a[i - 1] = b[i - 1];
    }
}

```

Funcționarea algoritmului:

- Contoarele asociate cheilor sunt memorate în tabloul numar de dimensiune m .
- Inițial locațiile tabloului numar sunt inițializate pe zero (prima buclă **for**).
- Se contorizează cheile în tabloul numar (a doua buclă **for**).
- În continuare sunt ajustate valorile contoarelor tabloului numar (a treia buclă **for**).

Funcționarea algoritmului (continuare):

- Se parurge tabloul a de la sfârșit spre început, iar cheile sunt introduse exact la locul lor în tabloul b cu ajutorul contoarelor memorate în tabloul numar (a patra buclă **for**).
- Concomitent cu introducerea cheilor are loc și **decrementarea** contoarelor specifice astfel încât în final, cheile identice apar în binul specific în **ordinea relativă** în care apar în secvența inițială.
- Ultima buclă **for** realizează **mutarea** integrală a elementelor tabloului b în tabloul a, (dacă acest lucru este necesar).

Analiza algoritmului

---> BINSORT

Analiza performanță:

Avem în vedere că în fiecare pas, un element este pus la locul lui în tabloul ordonat

Comparății

$$C_{min} = C_{max} = C_{med} = O(n)$$

Mutări de elemente

$$M_{min} = O(1)$$

$$M_{max} = O(n)$$

Tehnica aceasta simplă și performantă se bazează pe următoarele **cerințe apriorice**:

- **Domeniul limitat** al cheilor ($0, n-1$)
- **Unicitatea** fiecărei chei

Dacă cea de-a doua cerință **nu** este respectată, și de fapt acesta este cazul obișnuit, este necesar ca într-un **bin** să fie memorate **mai multe elemente** având aceeași cheie

Acest lucru se realizează fie prin **înșiruire**, fie prin **concatenare**, fiind utilizate în acest scop **structuri listă**

Această situație **nu** deteriorează prea mult performanțele acestei tehnici, efortul de sortare ajungând egal cu $O(n+m)$, unde n este numărul de elemente iar m numărul de chei.

Din acest motiv, această metodă reprezintă punctul de plecare al mai multor tehnici de sortare a structurilor listă

Spre exemplu, o metodă de rezolvare a unei astfel de situații este cea bazată pe **determinarea distribuției cheilor** ("distribution counting sort" sau "counting sort")

--- COUNTING SORT

Analiza algoritmului:

Deși se realizează mai multe treceri prin elementele tabloului totuși în ansamblu, **performanța algoritmului de sortare bazat pe determinarea distribuției cheilor** este $O(n)$.

Aceasta metodă de sortare pe lângă faptul că este rapidă are avantajul de a fi **stabilă**, motiv pentru care ea stă la baza mai multor metode de sortare de tip **radix**.

Avantaje

--- BINSORT

Este foarte rapid pentru distribuții uniforme de date într-un interval restrâns.

Este stabil dacă fiecare bucket este sortat cu un algoritm stabil

--- COUNTING SORT

- Foarte rapid pentru liste mari de numere întregi dintr-un interval mic.
- Este stabil, deci păstrează ordinea elementelor egale.

Dezavantaje

--- BINSORT

- Performanța depinde de distribuția datelor și de alegera numărului de buckets.
- Nu este eficient pentru liste mari cu o gamă largă de valori.

--- COUNTING SORT

- Nu funcționează pentru numere reale sau pentru intervale foarte mari de valori.
- Necesară spațiu suplimentar pentru array-ul de frecvențe și nu este in-situ.

RadixSort

O metodă pe care o putem folosi pentru sortarea unor facturi, ar fi să sortăm facturile pe luni, știind că putem avea doar 12 luni și mai apoi fiecare teanc corespunzând fiecărei luni calendaristice să îl sortăm după ani. În acest mod nu ar trebui să ne preocupăm de faptul că numărul anilor poate fi variabil, deoarece numărul maxim de luni este fix.

Într-un mod asemănător, putem sorta numere sortând pe rând după câte o cifră, indiferent de numărul de cifre pe care îl pot avea aceste numere.

Un **exemplu** sugestiv îl reprezintă sortarea unui teanc de cartele care au inscripționate pe ele **numere formate din trei cifre**.

- Se grupează cartelele în 10 grupe distincte, prima cuprinzând cheile mai mici decât 100, a doua cheile cuprinse între 100 și 199, etc., adică se realizează o sortare după **cifra sutelor**.
- În continuare se sortează pe rând grupele formate aplicând aceeași metodă, după **cifra zecilor**.
- Apoi fiecare grupă nou formată, se sortează după **cifra unităților**.
- Acesta este un exemplu simplu de **sortare radix** cu baza de numerație $m = 10$.
- Metodele de sortare prezentate până în prezent, concep cheile de sortat ca entități pe care le prelucrează integral prin comparare și interschimbare.
- În unele situații însă se poate profita de faptul că aceste chei sunt de fapt **numere** exprimate prin **cifre** aparținând unui **domeniu mărginit**.
- Metodele de sortare care iau în considerare **proprietățile digitale** ale numerelor sunt **metodele de sortare bazate pe baze de numerație ("radix sort")**.
- Algoritmii de tip **bază de numerație**:
 - (1) Consideră cheile ca și numere reprezentate într-o **bază de numerație m** , unde m poate lua diferite valori ("radix").
 - (2) Procesează **cifrele individuale** ale numărului.

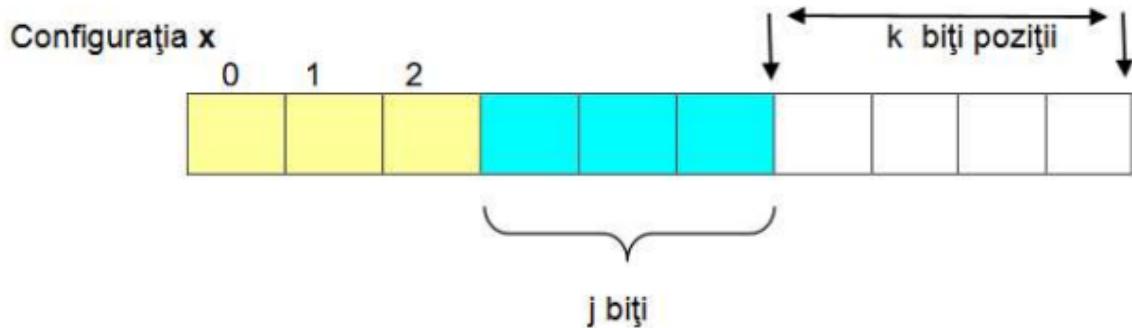
Pentru sistemele de calcul, unde prelucrările se fac exclusiv în baza 2, se pretează cel mai bine metodele de **sortare radix** care operează cu numere **binare**.

- În general, în **sortarea radix** a unui set de numere, **operația fundamentală** este extragerea unui **set contigu de biți** din numărul binar care reprezintă **cheia**.
- Spre **exemplu**:
 - Pentru a extrage primii 2 biți ai unui număr binar format din 10 cifre binare:
 - (1) Se realizează o **deplasare la dreapta** cu 8 poziții a reprezentării binare a numărului.
 - (2) Se operează configurația obținută, printr-o **operătie "și"** cu masca 0000000011.

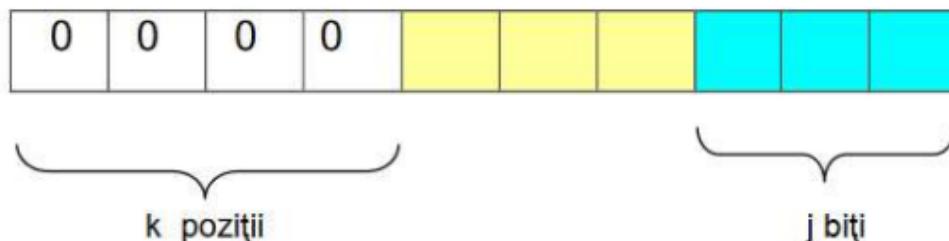
În continuare, pentru implementarea sortărilor radix, se consideră definit un operator **biti(x,k,j:integer):integer** care combină cele două operații returnând valorile a j biți care apar la k poziții de la marginea dreaptă a lui x.

- O posibilă implementare în limbajul C a acestui operator

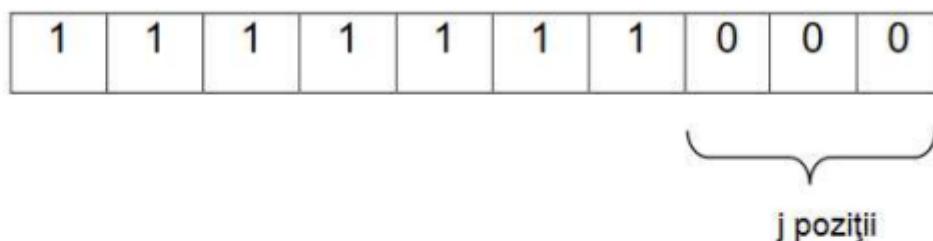
```
-----  
/*operator care returnează j biți care apar la k poziții de  
marginea dreaptă a lui x */  
unsigned biti(unsigned x, int k, int j)  
{  
    return (x>>k) &~ (~0<<j);  
}  
-----
```



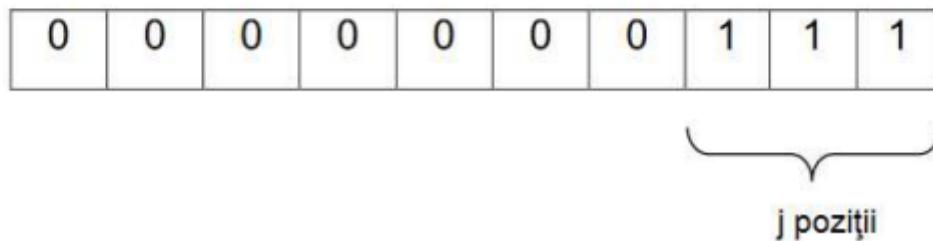
Configurația $x>>k$



Masca $\sim 0<<j$



Masca $\sim(\sim 0<<j)$



Configurația $(x>>k) \& \sim(\sim 0<<j)$



Pasii algoritmului

1. Determinăm numărul maxim de cifre (sau „lungimea” maximă) pentru elementele de sortat.

- Începem sortarea de la cifra cea mai puțin semnificativă (de exemplu, unitățile) și avansăm către cifrele mai semnificative.
- Pentru fiecare poziție a cifrei, aplicăm o sortare stabilă, cum ar fi **Counting Sort** (adaptată pentru a sorta după cifrele individuale).

A	00001
S	10011
O	01111
R	10010
T	10100
I	01001
N	01110
G	00111
E	00101
X	11000
A	00001
M	01101
P	10000
L	01100
E	00101

A	00001	A	0	0001	A	0	0	001	A	00	0	01	A	000	0	1	A	0000	1	A
S	10011	E	0	0101	E	0	0	101	A	00	0	01	A	000	0	1	A	0000	1	A
O	01111	O	0	1111	A	0	0	001	E	00	1	01	E	001	0	1	E	0010	1	E
R	10010	L	0	1100	E	0	0	101	E	00	1	01	E	001	0	1	E	0010	1	E
T	10100	M	0	1101	G	0	0	111	G	00	1	11	G	001	1	1	G	0011	1	G
I	01001	I	0	1001	I	0	1	001	I	01	0	01	I	010	0	1	I	0100	1	I
N	01110	N	0	1110	N	0	1	110	M	01	1	10	L	011	0	0	L	0110	0	L
G	00111	G	0	0111	M	0	1	101	M	01	1	01	M	011	0	1	M	0110	1	M
E	00101	E	0	0101	L	0	1	100	L	01	1	00	N	011	1	0	N	0111	0	N
X	11000	A	0	0001	O	0	1	111	O	01	1	11	O	011	1	1	O	0111	1	O
A	00001	X	1	1000	S	1	0	011	S	10	0	11	P	100	0	0	P	1000	0	P
M	01101	T	1	0100	T	1	0	100	R	10	0	10	R	100	1	0	R	1001	0	R
P	10000	P	1	0000	P	1	0	000	P	10	0	00	S	100	1	1	S	1001	1	S
L	01100	R	1	0010	R	1	0	010	T	10	1	00	T	101	0	0	T	1010	0	T
E	00101	S	1	0011	X	1	1	000	X	11	0	00	X	110	0	0	X	1100	0	X

Implementare

Există două metode de bază pentru implementarea sortării radix.

- (1) Prima metodă examinează biții cheilor de la **stânga la dreapta** și se numește **sortare radix prin interschimbare ("radix exchange sort")**.
- (2) A doua metodă se numește **sortare radix directă ("straight radix sort")**.

Sortarea radix prin interschimbare ("radix exchange sort")

- **Ideea sortării radix prin interschimbare** este următoarea:
- Se sortează elementelor tabloului a astfel încât toate elementele ale căror chei încep cu un bit zero să fie trecute în fața celor ale căror chei încep cu 1.
- Aceast proces va avea drept consecință formarea a două **partiții** ale tabloului inițial.
- Cele două partiții la rândul lor se sortează independent, conform aceleiași metode **după cel de-al doilea bit** al celor elementelor, rezultând 4 partiții.
- Cele 4 partiții rezultate se sortează similar după al 3-lea bit, și.a.m.d.
- Acum mod de lucru sugerează abordarea recursivă a implementării metodei de sortare.

Procesul de sortare radix prin interschimbare se desfășoară exact ca și la **partiționare**:

- Se balează tabloul de la **stânga spre dreapta** până se găsește un element a cărui cheie începe cu 1.
- Se balează tabloul de la **dreapta spre stânga** până se găsește un element a cărui cheie începe cu 0.
- Se **interschimbă** cele două elemente.
- Procesul **continuă** până când idicatorii de parcurgere **se întâlnesc** formând două partiții.
- Se reia aceeași procedură pentru **cel de-al doilea bit** al celor elementelor în cadrul fiecareia dintre cele două partiții rezultate și.a.m.d.

Pseudocod:

RadixInterschimb (stanga,dreapta: TipIndice, b: INTEGER)

//stânga,dreapta - limitele curente ale tabloului de sortat, b - lungimea în biți a cheii de sortat dacă (dreapta>stanga) și (b>=0) atunci

i:= stanga; j:= dreapta; b:= b-1;

executa

 cât timp (biti(a[i].cheie,b,1)=0) și (i<j)

 i:= i+1;

 cât timp(biti(a[j].cheie,b,1)=1) și (i<j)

 j:= j-1;

 interschimba a[i] și a[j]

cât timp j!=i;

dacă biti(a[dreapta].cheie,b,1)= 0

 j:= j+1; {dacă ultimul bit testat este 0 se reface lungimea partiției}

 RadixInterschimb(stanga,j-1,b-1);

 RadixInterschimb(j,dreapta,b-1);

```
void radinters(int stanga, int dreapta, int b, int a[])
/*stanga, dreapta - limitele curente ale tabloului de sortat b - lungimea în
biți a cheii de sortat -1*/
{
    int i, j;
    if ((dreapta > stanga) && (b >= 0)) {
        /* Implementație pseudocod */
        i = stanga;
        j = dreapta;
        b--;
        while (i < j) {
            if (biti(a[i].cheie, b, 1) == 0) {
                i++;
            } else if (biti(a[j].cheie, b, 1) == 1) {
                j--;
            } else {
                /* Interschimbă a[i] și a[j] */
                /* Implementație pseudocod */
                /* Înlocuiește a[i] cu a[j] și viceversă */
            }
        }
        /* Rezolvă cazul în care j > i */
        if (j > i) {
            /* Implementație pseudocod */
            /* Înlocuiește a[i] cu a[j] și viceversă */
        }
        /* Recursează pentru partea stângă */
        RadixInterschimb(stanga, j-1, b-1);
        /* Recursează pentru partea dreaptă */
        RadixInterschimb(j, dreapta, b-1);
    }
}
```

```

        i = stanga;
        j = dreapta;
        do {
            while ((biti(a[i], b, 1) == 0) && (i < j))
                i = i + 1;
            while ((biti(a[j], b, 1) == 1) && (i < j))
                j = j - 1;
            swap(&a[i], &a[j]);
        } while (!(j == i));
        if (biti(a[dreapta], b, 1) == 0)
            j = j + 1;
        /* dacă ultimul bit testat este 0 se reface lungimea
partiției */
        radinters(stanga, j - 1, b - 1, a);
        radinters(j, dreapta, b - 1, a);
    }
}

```

O altă variantă de implementare a **sortării radix** este aceea de a examina biții cheilor elementelor de la **dreapta la stânga**.

Aceată metodă se numește **sortare radix directă**.

- **Ideea** metodei de **sortare radix directă**:
- Se sortează cheile după un bit examinând biții lor **de la dreapta spre stânga**.
- Sortarea după bitul i constă în extragerea tuturor elementelor ale căror chei au zero pe poziția i și plasarea lor în fața celor care au 1 pe aceeași poziție.
- Când se ajunge la bitul i venind dinspre dreapta, cheile sunt gata sortate pe ultimii $i-1$ biți ai lor.

Nu este ușor de demonstrat că metoda este corectă: de fapt ea este corectă **numai** în situația în care **sortarea după 1 bit** este o **sortare stabilă**.

Datorită acestei cerințe, **interschimbarea normală nu** poate fi utilizată deoarece **nu** este o **metodă de sortare stabilă**.

Am observat la radix prin interschimbare că eficiența în termeni de $O(f(n))$ este $b \cdot n$, unde b este numărul de biți care ne dă și numărul de parcurgeri

Pentru a îmbunătății această eficiență ne dorim să reducem numărul de parcurgeri, astfel putem sorta cheile nu doar după un bit, ci după un număr de m biți folosind o sortare stabilă

Ne amintim faptul că sortarea prin determinarea distribuției cheilor (counting sort) este o sortare stabilă.

Aceasta poate fi utilizată cu succes în sortarea radix directă

În continuare, urmează un exemplu folosind counting sort în care $m = 2$ și baza de numerație este tot 2

index	0	1	2	3	4	5	6	7	
Baza 10	10	8	3	2	1	7	6	9	Sortare dupa cei mai puțin semnificativi 2 biți
Baza 2	1010	1000	0011	0010	0001	0111	0110	1001	

$m = 2$

Tabloul numar

0	1	2	3
00	01	10	11
0	0	0	0

Initializare

0	1	2	3
1	2	3	2

Contorizare valori

Tabloul numar

0	1	2	3
1	3	6	8

Ajustare valori

index	0	1	2	3	4	5	6	7
Baza 10	8	1	9	10	2	6	3	7
Baza 2	1000	0001	1001	1010	0010	0110	0011	0111

-> aici avem vectorul sortat in functie de ultimii 2 biti

index	0	1	2	3	4	5	6	7	
Baza 10	8	1	9	10	2	6	3	7	Sortare dupa cei mai semnificativi 2 biți
Baza 2	1000	0001	1001	1010	0010	0110	0011	0111	

$m = 2$

Obs: După biții cei mai puțini semnificativi tabloul este deja sortat din pasul anterior

Tabloul numar

0	1	2	3
00	01	10	11
0	0	0	0

Initializare

0	1	2	3
3	2	3	0

Contorizare valori

Tabloul numar

0	1	2	3
3	5	8	8

Ajustare valori

index	0	1	2	3	4	5	6	7
Baza 10	1	2	3	6	7	8	9	10
Baza 2	0001	0010	0011	0110	0111	1000	1001	1010

Pentru a crește performanța procesului de sortare, nu este indicat să se lucreze cu $m = 2$, ci este convenabil ca m să fie cât mai mare.

- În acest fel se reduce numărul de treceri.
- Dacă se prelucrează m biți odată:
 - Timpul de sortare scade prin reducerea numărului de treceri.
 - Tabela de distribuții însă crește ca dimensiune ea trebuind să conțină $m_1 = 2^m$ locații, deoarece cu m biți se pot alcătui 2^m configurații binare.
- În acest mod, sortarea radix-directă devine o generalizare a sortării bazate pe determinarea distribuțiilor.

```
void radixdirect(int a[], int n, int b)
// b-numarul de biti pe care este reprezentata cheia -1
{
    int i, j, trecere;
    int numar[m1];
    int aux;
    /*m1:=2^m*/
    for (trecere = 0; trecere <= (b / m) - 1; trecere++)
        for (j = 0; j <= m1 - 1; j++)
            numar[j] = 0;
```

```

        for (i = 0; i <= n - 1; i++) {
            aux = biti(a[i], trecere * m, m);
            numar[aux] = numar[aux] + 1;
        }
        for (j = 1; j <= m1 - 1; j++)
            numar[j] = numar[j - 1] + numar[j];
        for (i = n - 1; i >= 0; i--) {
            aux = biti(a[i], trecere * m, m);
            t[numar[aux] - 1] = a[i];
            numar[aux] = numar[aux] - 1;
        }
        for (i = 0; i < n; i++)
            a[i] = t[i];
    }
}

```

```

#include <stdio.h>

// A utility function to get the maximum
// value in arr[]
int getMax(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[]
// according to the digit represented by exp
void countSort(int arr[], int n, int exp) {
    int output[n]; // Output array
    int count[10] = {0}; // Initialize count array as 0

    // Store count of occurrences in count[]
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
}

// Change count[i] so that count[i] now
// contains actual position of this digit
// in output[]
for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

// Build the output array
for (int i = n - 1; i >= 0; i--) {
    output[count[(arr[i] / exp) % 10] - 1] = arr[i];
    count[(arr[i] / exp) % 10]--;
}

```

```

        // Copy the output array to arr[],
        // so that arr[] now contains sorted
        // numbers according to current digit
        for (int i = 0; i < n; i++)
            arr[i] = output[i];
    }

    // The main function to sort arr[] of size
    // n using Radix Sort
    void radixSort(int arr[], int n) {

        // Find the maximum number to know
        // the number of digits
        int m = getMax(arr, n);

        // Do counting sort for every digit
        // exp is 10^i where i is the current
        // digit number
        for (int exp = 1; m / exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }

    // A utility function to print an array
    void printArray(int arr[], int n) {
        for (int i = 0; i < n; i++)
            printf("%d ", arr[i]);
        printf("\n");
    }

    // Driver code
    int main() {
        int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
        int n = sizeof(arr) / sizeof(arr[0]);

        // Function call
        radixSort(arr, n);
        printArray(arr, n);
        return 0;
    }
}

```

Analiza algoritmului

Din punctul de vedere al **performanței**, metoda de sortare radix prin interschimbare sortează n chei de b biți utilizând un număr de comparații de biți egal cu $n \cdot b$.

- Cu alte cuvinte, **sortarea radix prin interschimbare** este liniară cu **numărul de biți ai unei chei**.
- Pentru o **distribuție normală** a bițiilor cheilor, **sortarea radix prin interschimbare** este ceva mai rapidă decât metoda **quicksort**.

Performanța sortării radix directe:

- Sortează n elemente cu chei de b biți în b/m treceri.

Dezavantajele metodei de **sortare radix directă**:

- Utilizează un **spațiu suplimentar de memorie** pentru 2^m contoare.
- Utilizează un **buffer** pentru rearanjarea tabloului cu dimensiunea egală cu cea a tabloului original.

Timpul de execuție al celor două metode de sortare radix fundamentale, pentru n elemente având chei de b biți este în esență proporțional cu $n \cdot b$.

Pe de altă parte, timpul de execuție poate fi aproimat ca fiind $n \cdot \log(n)$, deoarece dacă toate cheile sunt diferite, b trebuie să fie cel puțin $\log(n)$.

Avantaje

- Este foarte rapid pentru sortarea unui număr mare de numere întregi, în special atunci când numerele au lungimi similare.
- Este stabil, păstrând ordinea relativă a elementelor egale.

Dezavantaje

- Nu este in situ, deoarece necesită spațiu suplimentar.
- Funcționează eficient doar pentru numere întregi sau siruri de caractere de lungime fixă, fiind mai puțin potrivit pentru alte tipuri de date.

MergeSort

Mergesort sau sortarea prin interclasare, se bazează, la fel ca și quicksort și shellsort pe principiul "divide et impera" (divide and conquer).

Principiul de sortare este simplu:

- Se împarte secvența de elemente de ordonat în două subsecvențe egale (sau care diferă prin cel mult un element, în cazul unui număr impar de elemente)
- Fiecare subsecvență se împarte la rândul său în alte două subsecvențe până se ajunge la subsecvențe de un element, care se consideră ordonate
- Fiecare două subsecvențe ordonate se combină într-o subsecvență ordonată printr-un proces care se numește interclasare ("merging").
- Procesul se reia pe subsecvențele deja ordonate până când întreaga secvență este ordonată

Interclasarea presupune combinarea a două sau mai multe secvențe ordonate într-o singură secvență ordonată, prin selecții repetitive ale componentelor curent accesibile.

Metoda de sortare presupune folosirea de structuri de memorie suplimentare

Pasii algoritmului

- Divide:** Împarte lista în două jumătăți până când obținem liste de lungime 1.
- Conquer:** Sortează fiecare jumătate recursiv.
- Combine:** Îmbină cele două jumătăți sortate într-o listă unică, păstrând ordinea.

Interclasarea a două secvențe ordonate într-o altă secvență (tot ordonată)

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

1<2 , se avansează în secvența s1

1								
---	--	--	--	--	--	--	--	--

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

2<3 , se avansează în secvența s2

1	2							
---	---	--	--	--	--	--	--	--

1	3	6	7
---	---	---	---

s1

2	4	5	8	11
---	---	---	---	----

s2

3<4 , se avansează în secvența s1

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

1	3	6	7	
---	---	---	---	--

s1

2	4	5	8	11	
---	---	---	---	----	--

s2

$4 < 6$, se avansează în secvența s2

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

1	3	6	7		
---	---	---	---	--	--

s1

2	4	5	8	11	
---	---	---	---	----	--

s2

$5 < 6$, se avansează în secvența s2

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

1	3	6	7	
---	---	---	---	--

s1

2	4	5	8	11	
---	---	---	---	----	--

s2

$6 < 8$, se avansează în secvența s1

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

1	3	6	7	
---	---	---	---	--

s1

2	4	5	8	11	
---	---	---	---	----	--

s2

$7 < 8$, se avansează în secvența s1 care se termină

1	2	3	4	5	6	7		
---	---	---	---	---	---	---	--	--

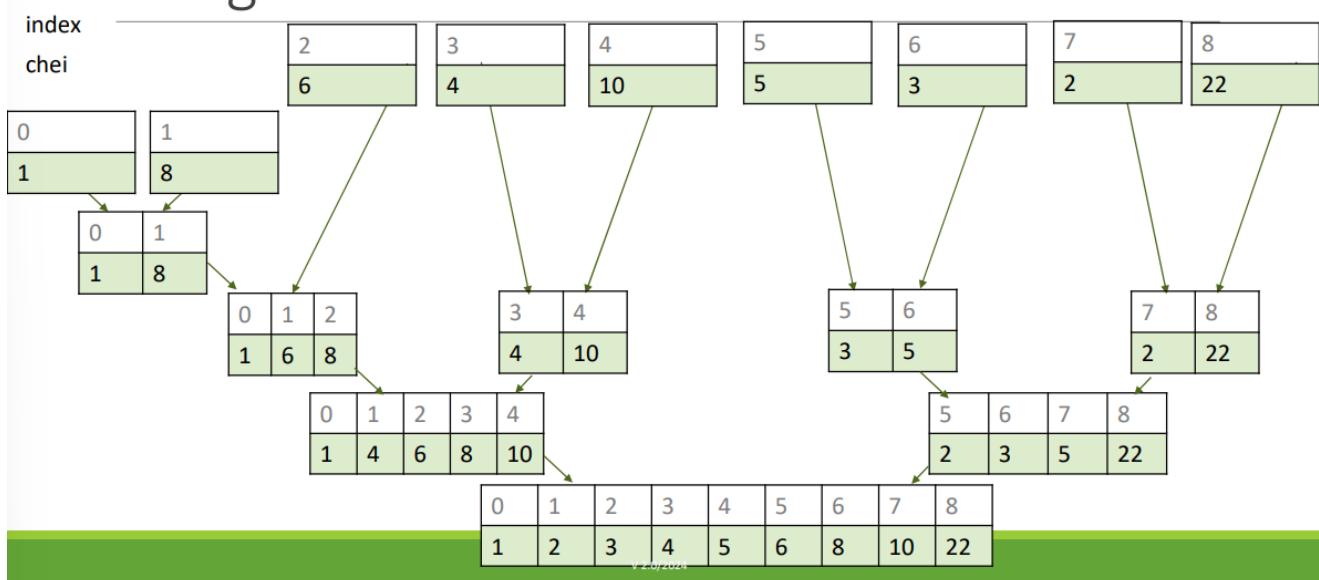
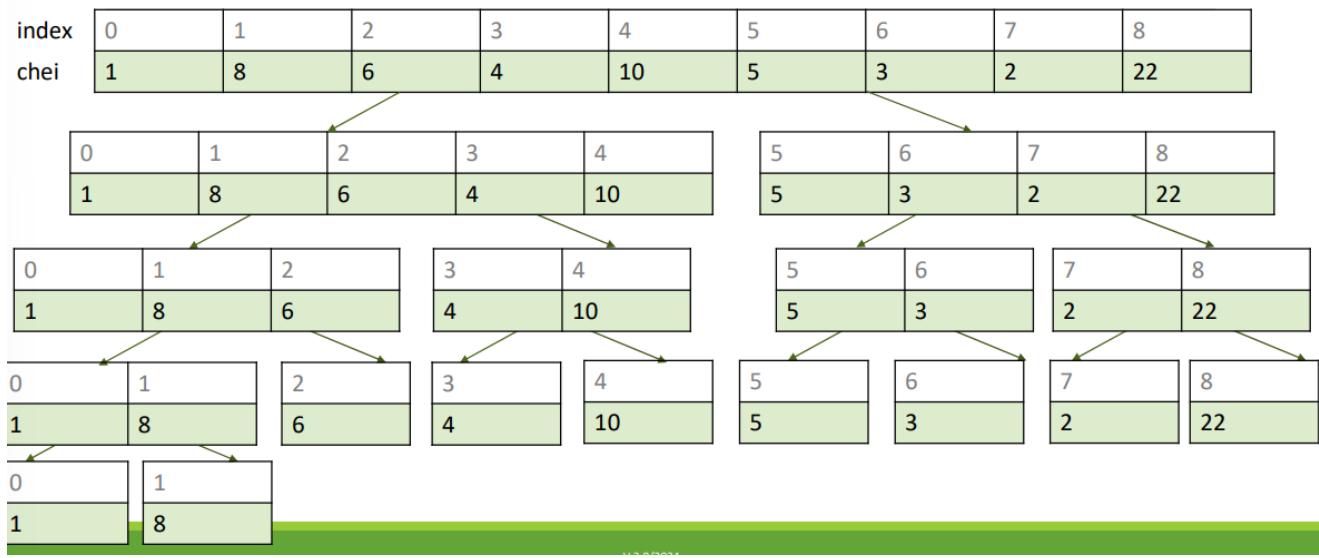
1	3	6	7
---	---	---	---

s1				
2	4	5	8	11

s2

restul elementelor din s2 se copiază în secvența rezultat

1	2	3	4	5	6	7	8	11
---	---	---	---	---	---	---	---	----



Implementare

Pseudocod:

```
tablou mergesort(tablou inlist) {
    daca (inlist.length() <= 1) returneaza inlist;
    tablou T1 = jumata din elementele din inlist;
    tablou T2 = cealalta jumata din elementele din inlist;
    returneaza interclasare(mergesort(T1), mergesort(T2));
}
```

```
void mergesort(tip_element A[], tip_element temp[], int left, int right) {
    if (left == right)
        return;
    // secenta de un element
    int mid = (left + right) / 2;
    mergesort(A, temp, left, mid);
    mergesort(A, temp, mid + 1, right);

    for (int i = left; i <= right; i++)
        temp[i] = A[i];
    // se copiaza secenta in temp
    int i1 = left;
    int i2 = mid + 1;
    // interclasare inapoi in A

    for (int index = left; index <= right; index++) {
        if (i1 == mid + 1)
            A[index] = temp[i2++];
        // s-a epuizat secenta din stanga
        else if (i2 > right)
            A[index] = temp[i1++];
        // din dreapta
        else if (temp[i1].cheie < temp[i2].cheie)
            A[index] = temp[i1++];
        else A[index] = temp[i2++];
    }
}
```

```
// C program for Merge Sort
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
```

```

{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temp arrays
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// l is for left index and r is right index of the
// sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r)

```

```

{
    if (l < r) {
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

// Function to print an array
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}

```

Analiza algoritmului

Analiza algoritmului este directă, în ciuda faptului că avem un algoritm recursiv

Interclasarea durează $2k$ timpi, deci are eficiență $\Theta(k)$, unde k este lungimea secvențelor care sunt interclasate

Algoritmul împarte secvența de sortat în jumătate până când avem subsecvențe de un element, astfel numărul de apeluri recursive este $\log n$, pentru n elemente (indiferent de valoarea lor).

Astfel numărul de mutări devine:

$$M(1) = 0$$

$$M(n) = 2M\left(\frac{n}{2}\right) + 2n$$

$$\begin{aligned} M(n) &= 2\left(2M\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right)\right) + 2n = 4M\left(\frac{n}{4}\right) + 4n \\ &= 4\left(2M\left(\frac{n}{8}\right) + 2\left(\frac{n}{4}\right)\right) + 4n = 8M\left(\frac{n}{8}\right) + 6n = \dots = 2^i M\left(\frac{n}{2^i}\right) + 2 \cdot i \cdot n \end{aligned}$$

Dacă alegem $i = \log n$, atunci

$$M(n) = 2^i M\left(\frac{n}{2^i}\right) + 2 \cdot i \cdot n = nM(1) + 2n \cdot \log n = 2n \cdot \log n = O(n \cdot \log n)$$

Formula este valabilă pentru toate cazurile (cel mai favorabil, cel mai defavorabil și cazul mediu)

Numărul comparațiilor este dat de o relație similară

$$C(1) = 0$$

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1$$

Numărul comparațiilor fiind tot $O(n \cdot \log n)$

The recurrence relation of merge sort is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- $T(n)$ Represents the total time taken by the algorithm to sort an array of size n .
- $2T(n/2)$ represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has $n/2$ elements, we have two recursive calls with input size as $(n/2)$.
- $\Theta(n)$ represents the time taken to merge the two sorted halves

Complexity Analysis of Merge Sort:

- **Time Complexity:**
 - **Best Case:** $O(n \log n)$, When the array is already sorted or nearly sorted.
 - **Average Case:** $O(n \log n)$, When the array is randomly ordered.
 - **Worst Case:** $O(n \log n)$, When the array is sorted in reverse order.
- **Auxiliary Space:** $O(n)$, Additional space is required for the temporary array used during merging.

Avantaje

Este foarte eficient pentru liste mari.

Este stabil, menținând ordinea relativă a elementelor egale.

Este garantat să aibă complexitate $O(n \log n)$, indiferent de cum sunt inițial ordonate elementele.

Advantages of Merge Sort:

- **Stability :** Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of $O(N \log N)$, which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.
- **Naturally Parallel :** We independently merge subarrays that makes it suitable for parallel processing.

Dezavantaje

- Nu este un algoritm in situ, deci necesită spațiu suplimentar pentru a păstra sublistele intermediiare.
- În practică, poate fi mai lent decât Quick Sort datorită costurilor de memorie suplimentară și manipulării datelor în afara cache-ului CPU.

Disadvantages of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Slower than QuickSort in general.** QuickSort is more cache friendly because it works in-place.

Sortari externe

Metodele de sortare prezentate anterior se aplicau pe structuri de date de tip tablou, folosindu-se de proprietățile acestuia prin care elementele se pot accesa în mod direct

În continuare vom adresa problema sortării unor secvențe de înregistrări care nu pot fi stocate în memoria operativă din cauza dimensiunilor acestora și nu pot fi accesate în mod direct

În acest caz, înregistrările sunt conținute în fișiere stocate în memoria externă (ex. hard disk)

Atunci când numărul înregistrărilor este prea mare pentru a fi reținute în memoria operativă, sortarea se face citind o serie de înregistrări, ordonându-le într-un anumit mod și scriindu-le înapoi în memoria externă. Acest proces se repetă până când întreg fișierul este sortat

Astfel, tehniciile de sortare prezentate anterior și aplicate tablourilor nu pot fi aplicate și în cazul sortărilor externe

Din punct de vedere al structurilor de date abstracte vom considera în continuare secvența

Ne amintim:

Particularități ale tablourilor:

- au cardinalitate finită
- pot fi accesate direct

Particularități ale secvențelor:

- au cardinalitate infinită
- pot fi accesate doar în mod secvențial

Ținând cont de proprietățile secvențelor și de costul de citire/scriere din/în memoria externă scopul sortărilor externe este de a minimiza numărul de citiri/scrieri (de) pe disc

Modelul de acces pe disc împarte fișierele în blocuri de dimensiune fixă

Un bloc poate conține una sau mai multe înregistrări, în funcție de dimensiunea acestora

În continuare vom considera înregistrările de dimensiuni egale, astfel fiecare bloc conține același număr de înregistrări

În general citirea din fișier în ordine secvențială este mai eficientă decât citirea unor blocuri de memorie în ordine aleatoare

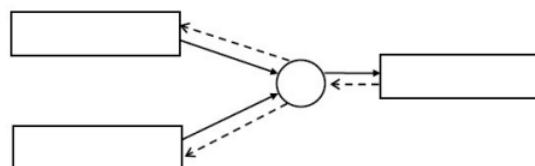
Cu toate că majoritatea algoritmilor de sortare prezentați anterior se bazează pe accesul direct la elemente, Mergesort (sortarea prin interclasare) este un algoritm care poate fi adaptat ușor la o sortare externă

Pasii algoritmului

Mergesort extern (sortare prin interclasare secvențială)

Principiul de funcționare:

1. Se **împarte** secvența A (fișierul original) în două alte secvențe (fișiere) B și C de dimensiune (aproximativ) egală
2. Se **interclasează** subsecvențe de câte o înregistrare din fiecare secvență B și C, obținând subsecvențe ordonate de câte două elemente, care se vor stoca în noua secvență A (rescriind fișierul original)
3. Se **repetă** cu noua secvență A, pașii anteriori, de această dată interclasând subsecvențele ordonate de două elemente în subsecvențe ordonate de patru elemente.
4. Se repetă pașii inițiali, **dublând mereu dimensiunea subsecvențelor ordonate** până la interclasarea întregii secvențe



Exemplu – sortare cu **3 secvențe** (benzi):

Secvența

A: 34 | 65 | 12 | 22 | 83 | 18 | 4 | 67 | 9 | 11

Pasul 1 – distribuire subsecvențe ordonate

B: 34 | 12 | 83 | 4 | 9

C: 65 | 22 | 18 | 67 | 11

Pasul 2 – interclasare

A: 34 | 65 | 12 | 22 | 18 | 83 | 4 | 67 | 9 | 11

Pasul 1 – distribuire

B: 34 | 65 | 18 | 83 | 9 | 11

C: 12 | 22 | 4 | 67

Pasul 2 – interclasare

A:	12	22	34	65	4	18	67	83	9	11
----	----	----	----	----	---	----	----	----	---	----

Pasul 1 – distribuire

B:	12	22	34	65	9	11
----	----	----	----	----	---	----

C:	4	18	67	83
----	---	----	----	----

Pasul 2 – interclasare

A:	4	12	18	22	34	65	67	83	9	11
----	---	----	----	----	----	----	----	----	---	----

Pasul 1 – distribuire

B:	4	12	18	22	34	65	67	83
----	---	----	----	----	----	----	----	----

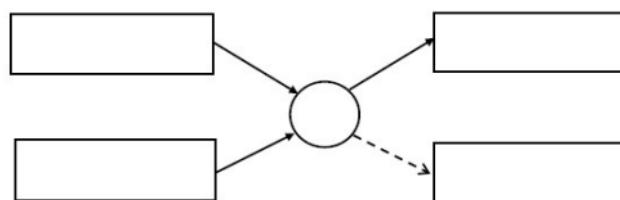
C:	9	11
----	---	----

Pasul 2 – interclasare

A:	4	9	11	12	18	22	34	65	67	83
----	---	---	----	----	----	----	----	----	----	----

Faza de înjumătățire care de fapt nu contribuie direct la sortare (în sensul că ea nu permutează nici un element), consumă jumătate din operațiile de copiere

- Acest neajuns poate fi remediat prin combinarea fazei de înjumătățire cu cea de interclasare
- Astfel simultan cu interclasarea se realizează și redistribuirea n-uplelor interclasate pe două secvențe care vor constitui sursa trecerii următoare.
- Acest proces se numește interclasare cu o singură fază sau interclasare echilibrată cu **4 secvențe**



Exemplu, sortare echilibrată cu **4 secvențe** :

	34	65	12	22	83	18	4	67	9	11
--	----	----	----	----	----	----	---	----	---	----

A:	34	12	83	4	9
----	----	----	----	---	---

B:	65	22	18	67	11
----	----	----	----	----	----

C:	34	65	18	83	9	11
----	----	----	----	----	---	----

D:	12	22	4	67
----	----	----	---	----

A:	12	22	34	65	9	11
----	----	----	----	----	---	----

B:	4	18	67	83
----	---	----	----	----

C:	4	12	18	22	34	65	67	83
----	---	----	----	----	----	----	----	----

D:	9	11
----	---	----

A:	4	9	11	12	18	22	34	65	67	83
----	---	---	----	----	----	----	----	----	----	----

Tehnica de sortare prin interclasare nu ia în considerare faptul că datele inițiale pot fi parțial sortate, subsecvențele interclasate având o lungime fixă predeterminată adică $2k$ în trecerea k .

- De fapt, oricare două subsecvențe ordonate de lungimi m și n pot fi interclasate într-o singură subsecvență ordonată de lungime $m+n$.
- Tehnica de interclasare care în fiecare moment combină cele mai lungi secvențe ordonate posibile se numește sortare prin **interclasare naturală**.
- În cadrul acestei tehnici un rol central îl joacă noțiunea de monotonie

Formal, se înțelege prin monotonie orice secvență parțială a_i, \dots, a_j care satisfac următoarele condiții:

-
- 1) $1 \leq i \leq j \leq n$;
 - 2) $a_k \leq a_{k+1}$ pentru orice $i \leq k < j$;
 - 3) $a_{i-1} > a_i$ sau $i = 1$;
 - 4) $a_j > a_{j+1}$ sau $j = n$;
-

- Această definiție include și monotonile cu un singur element, deoarece în acest caz $i=j$ și proprietatea 2) este îndeplinită, neexistând nici un k cuprins între i și $j-1$.

Exemplu, sortare prin **interclasare naturală** :

	34	65	12	22	83	18	4	67	9	11
A:	34	65	18	9	11					
B:	12	22	83	4	67					
C:	12	22	34	65	83	9	11			
D:	4	18	67							
A:	4	12	18	22	34	65	67	83		
B:	9	11								
C:	4	9	11	12	18	22	34	65	67	83

- Complexitatea metodelor de sortare externă prezentate nu permite formularea unor concluzii generalizatoare, cu atât mai mult cu cât evidențierea performanțelor acestora este dificilă.
- Se pot formula însă următoarele **observații**:
 - Există o legătură **indisolubilă** între un anumit **algoritm** care rezolvă o anumită problemă și **structurile de date** pe care acesta le utilizează, influența celor din urmă fiind uneori decisivă pentru algoritm, acest lucru este evidențiat cu preponderență în cazul sortărilor externe care sunt diferite ca mod de abordare în raport cu metodele de sortare internă
 - De multe ori trebuie să facem un compromis între performanța de timp și cea de memorie

Concluzii

Nume	Caz mediu	Cazul cel mai defavorabil	Memorie auxiliară	Algoritm stabil
Shellsort	Depinde de alegerea lui H	$O(n \log^2 n)$	$O(1)$	Nu
Quicksort	$O(n \log n)$	$O(n^2)$	Depinde de implementare	Depinde de implementare
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Nu
Mergesort	$O(n \log n)$	$O(n \log n)$	Depinde de implementare	Da

- Algoritmii de sortare se pot clasifica în funcție de următorii parametri:
 - Numărul de comparații – pentru sortările generale putem obține $O(n \log n)$ pentru cazul cel mai favorabil și $O(n^2)$ pentru cazul cel mai defavorabil.
 - Numărul de interschimbări/ mutări în memorie
 - Memoria utilizată – algoritmii "in situ" necesită $O(1)$ sau cel mult $O(\log n)$ spații auxiliare de memorie
 - Recursivitate – algoritmi recursivi (ex. variante de Quicksort, Mergesort) sau nerecursivi (ex. Selection Sort, Insertion Sort).
 - Stabilitate – pentru algoritmii stabili, elementele egale își păstrează ordinea relativă pe care au avut-o în sirul inițial.
 - Tipul de memorie folosit pentru stocarea datelor - algoritmi de sortare interni și externi.
- Sortarea elementelor este una dintre cele mai vechi și mai studiate probleme în domeniul informaticii
- Sortările pe bază de comparații au o limitare inferioară (lejeră) $\omega(f(n)) = n \log n$, ceea ce înseamnă că în cazul acestor sortări nu putem obține o performanță mai bună de $O(n \log n)$
- Mergesort și Heapsort au $O(n \log n)$ și pentru cazul cel mai defavorabil, în timp ce Quicksort doar pentru cazul mediu
- Avem și sortări în timp liniar (binsort și radix), dar fac parte dintr-o altă categorie de sortări față de cele bazate pe comparații și nu pot fi utilizate în orice caz
- Decizia cu privire la un algoritm de sortare anume se ia în funcție de mai mulți factori (nu doar în funcție de performanța de timp)