**Bubble Sort**

* Fie un array cu n elemente.

Sortarea va incepe mereu de la primul element pana la ultimul, si elementul cel mai mare se va tot misca la dreapta pana ajunge la urma, si va fi pe ultima pozitie. Apoi de la primul element mergem la penultimul, si al 2 cel mai mare element va ajunge in peunultima pozitie si tot asa.

De ex:

4 2 3 8 1

1. Prima iteratie

2 4 3 8 1

2 3 4 8 1

2 3 4 1 8 – deci 8 a ajuns la final.

1. Acum, array va fi iterat pana la penultima pozitie, caci 8 deja e la urma

2 3 1 4 8

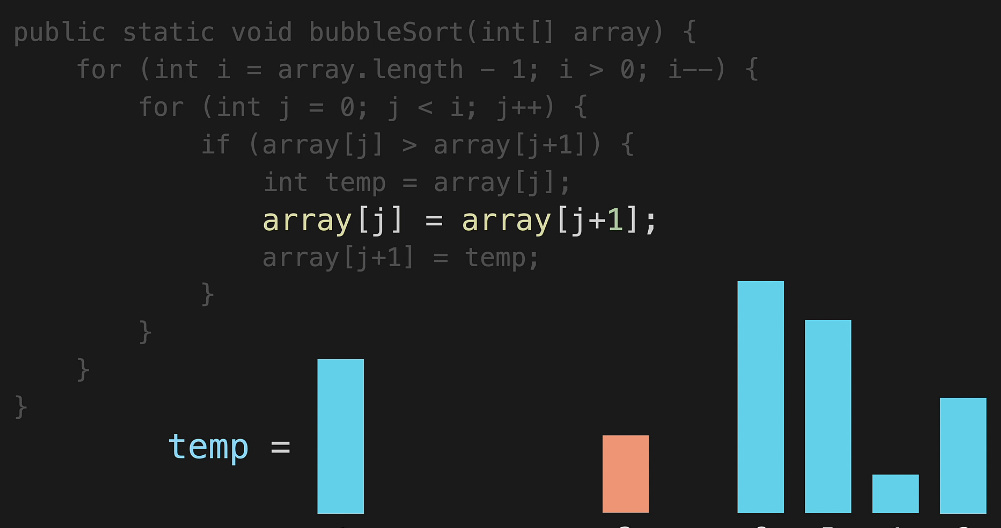
2 1 3 4 8

1 2 3 4 8 – 4, al 2 cel mai mare e pe pozitia sa

1. acum for va ignora deja ultimele 2 pozitii

......

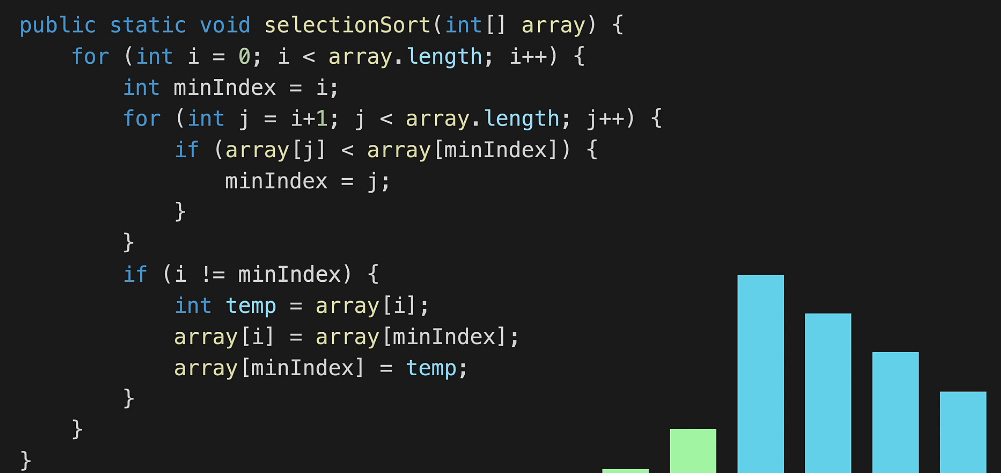
array deja e sortat, asa ca am terminat.



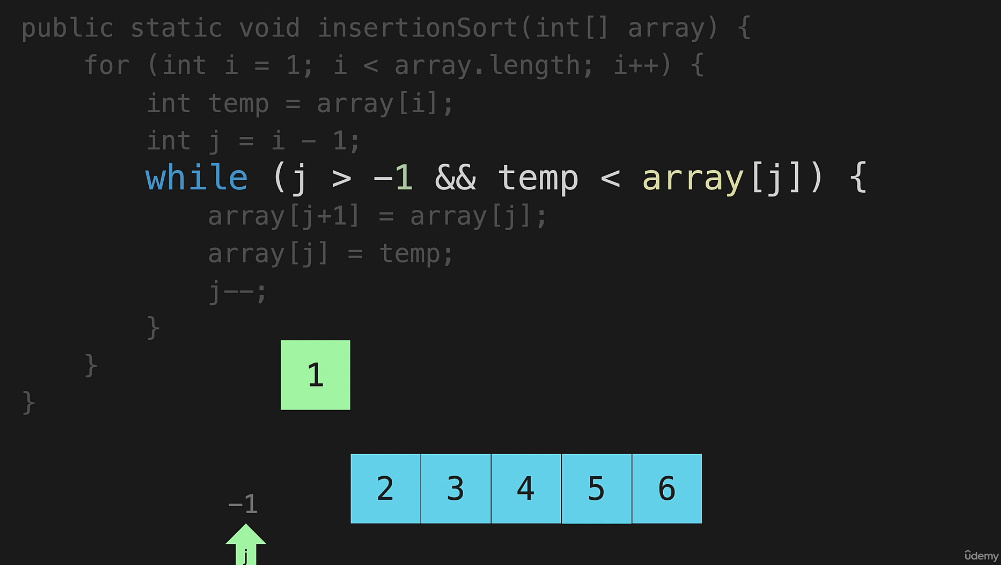
* BubbleSort e bun ca nu trebuie sa facem copii la array, de asta space complexity e O(1)

**Selection Sort**

* Se bazeaza pe aceea ca la fiecare iteratie determinam pe ce index se gaseste cel mai mic element curent.
* Deci, vom incepe sa cautam elementul cel mai mic, care trebuie sa stea pe index 0, apoi urmatorul cel mai mic, ce sa stea pe index 1 si tot asa



**Insertion Sort**

* Mereu incepem cu elementul al 2 din array, si il comparam cu cel din urma si tot asa mergem
* Metoda se bazeaza de a lua cate un subarray din array, si de a muta elementul selectat pe ultima pozitie din subarray in pozitia pe care ar trebui sa o ocupe in subarray. Deci, intai am selectat elementul cu indexul 1 din subarray din 2 elemente, apoi elementul cu indexul 2 din subarray cu 3 elemente apoi elementul cu indexul 3 din subarray cu 4 elemente si tot asa, si am determinat unde ar trebui sa stea elementul selectat in subarray.
* 
* Ex:

1. 5 3 6 1 2

3 5 6 1 2

1. 3 5 6 1 2
2. 3 5 6 1 2

3 5 1 6 2

3 1 5 6 2

1 3 5 6 2

1. 1 3 5 6 2

1 3 5 2 6

1 3 2 5 6

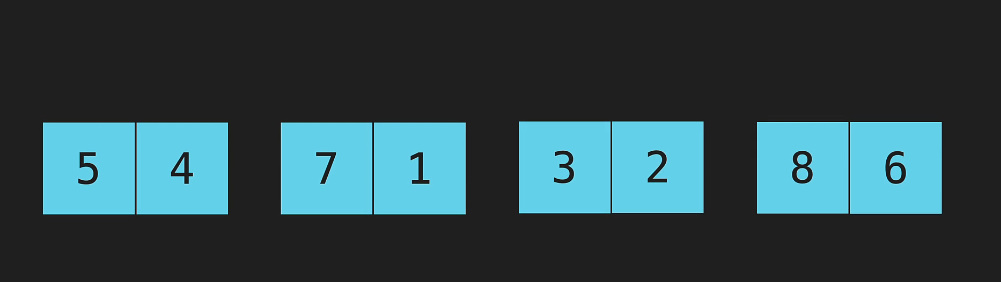
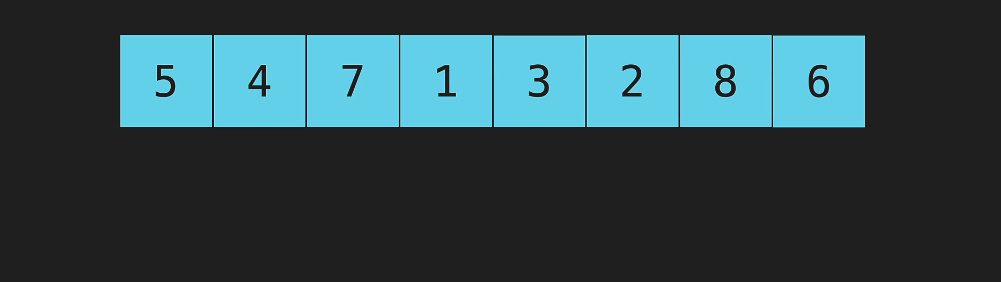
1 2 3 5 6

* Best case este o(n)

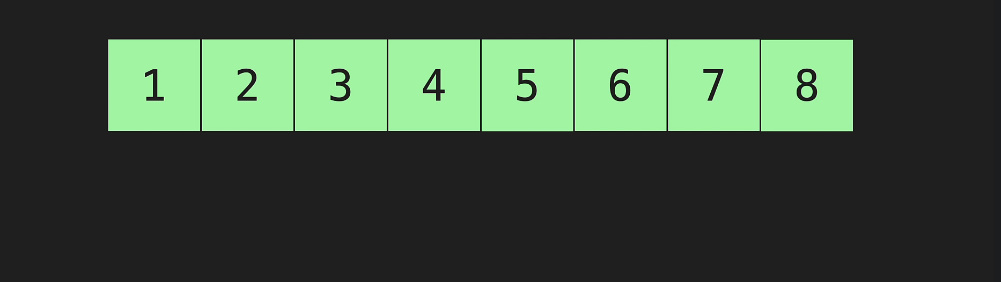
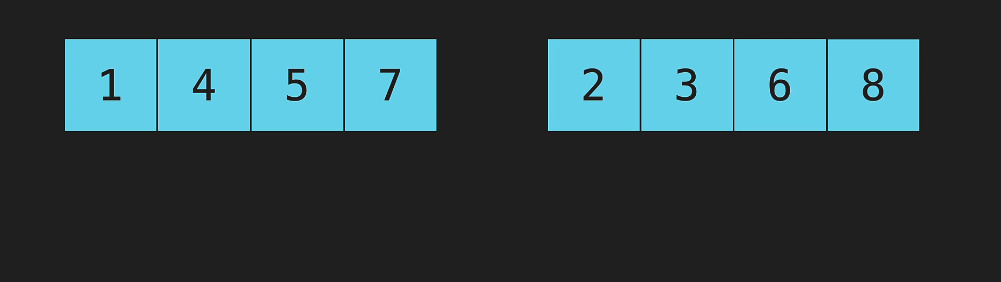
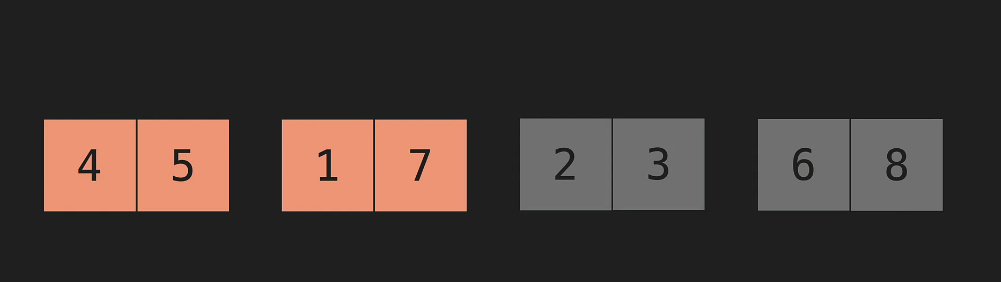
**Algoritmurile de sortare de mai sus au complexitatea O(n^2) si sunt forte bune daca sansele ca datele sa nu fie sortate de loc sunt mari. Urmatoarele algoritmuri au complexitatea O(nlogn), dar daca arrays sunt practic sortate, ele vor fi mai lente**

**Merge Sort**

* Merge sort foloseste recursivitatea
* El se bazeaza pe aceea ca impartim array in 2, apoi acele 2 bucati iar fiecare in 2 si tot asa pana ajungem la single item arrays

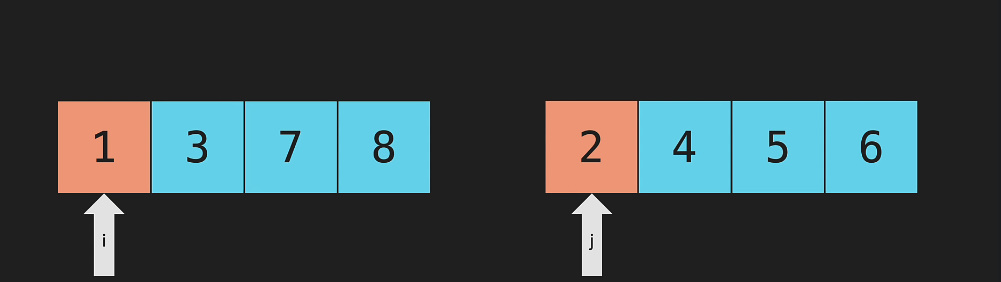


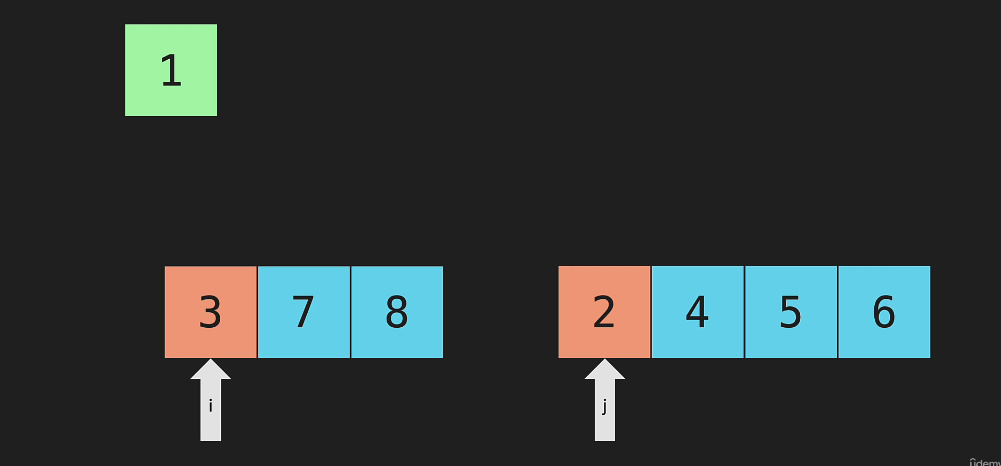
* Apoi, luam cate 2 single element arrays si le sortam, si cream asa arrays de 2 elemente, apoi aceste arrays de 2 elemente le luam cate 2 si le sortam impreuna si cream arrays de 4 elemente sortate si tot asa:

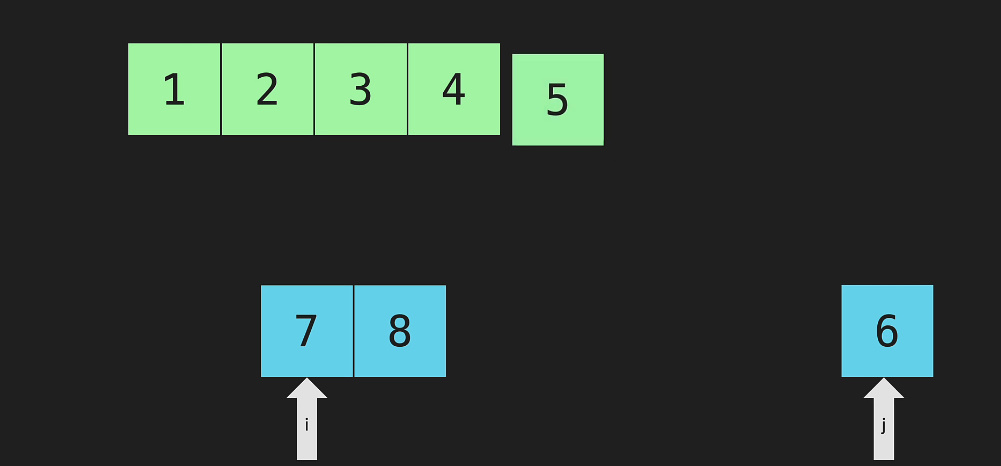


* **merge function:**

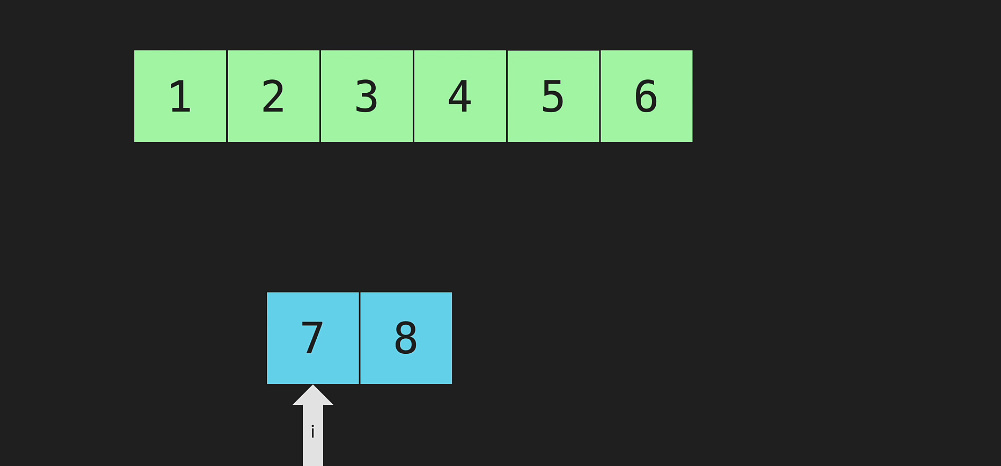
deci, luam cate un element din primul subarray si il comparam cu elementele din al 2 subarrays. Elementul care e mai mic se duce in noul arrays pe pozitia respectiva:







Daca se intampla ca un subarray mai are elemente, dar celalalt nu, pur si simplu punem la final la noul array elementele ramase, caci ele tot sun sortate:



public static int[] merge(int[] array1, int[] array2){

int[] newArray = new int[array1.length + array2.length];

int indexArray1 = 0;

int indexArray2 = 0;

int indexNewArray = 0;

while(indexArray1 < array1.length && indexArray2 < array2.length){

if(array1[indexArray1] < array2[indexArray2]){

newArray[indexNewArray] = array1[indexArray1];

indexArray1++;

}

else{

newArray[indexNewArray] = array2[indexArray2];

indexArray2++;

}

indexNewArray++;

}

while(indexArray1 < array1.length){

newArray[indexNewArray] = array1[indexArray1];

indexArray1++;

indexNewArray++;

}

while(indexArray2 < array2.length){

newArray[indexNewArray] = array2[indexArray2];

indexArray2++;

indexNewArray++;

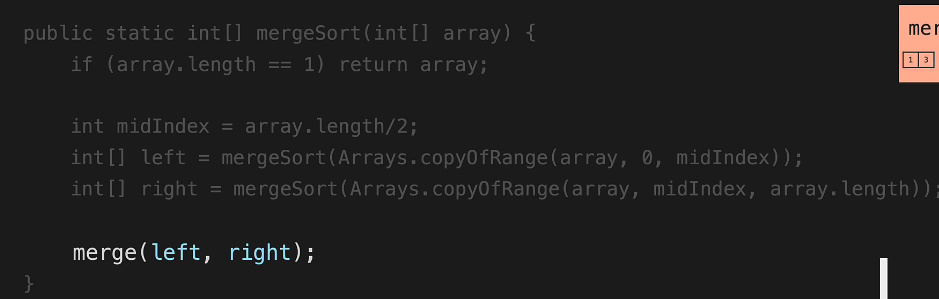
}

return newArray;

}

* **merge sort function:**

Functia data trebuie sa separa arrays in subarrays si sa returneze tot array sortat:





* Vedem ca merge sort returneaza un nou array sortat, nu cel initial, de aceea desi ea e buna ca complexity time, e mai rea ca space complexity.
* Space Complexity: O(n)
* Time complexity: n\*logn

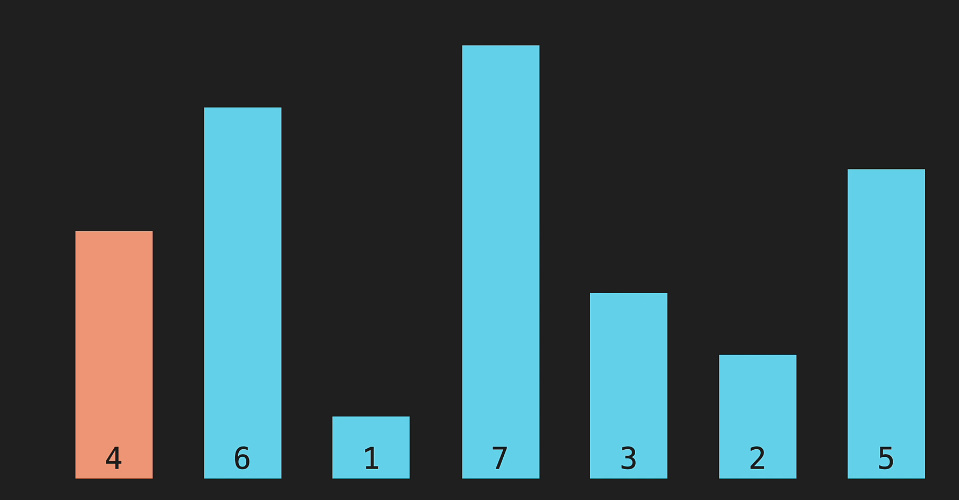
logn – de la separarea in subarrays

n – deoarece avem bucla for care va face ca sa se treaca prin fiecare element ca sa le puna la loc in noul array.

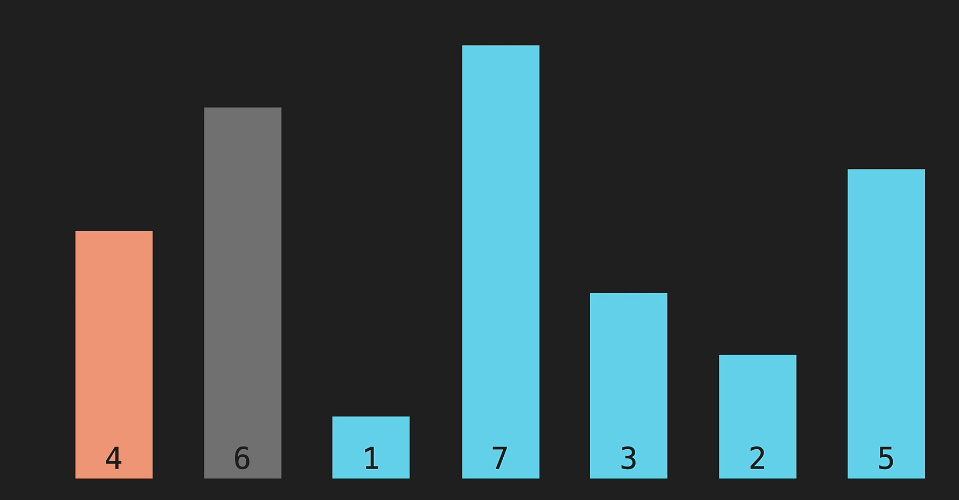
**Quick Sort**

* Quick sort prevede ca sa luam un element pivot, si sa determinam pozitia sa in array sortat si sa punem elementele mai mici ca el in stanga lui, cele mai mari ca el in dreapta lui
* Apoi, elementele din stanga lui vor forma un array si va fi si el sortat tot asa. Iar se va lua primul element din subarray ca pivot, se va determina pozitia lui care ar trebui sa fie, si ce e mai mare se va duce in dreapta, ce e mai mic in stanga si tot asa
* Primul element care e mai mic ca pivotul va fi pus dupa elementul pivot, adica el va fi schimbat cu primul element care e dupa cel pivot, daca nu el e, si pozitia acestui prim element mai mic va fi memorata. Dupa aceea, fiecare nou element care e mai mic ca pivotul va fi schimbat cu urmatorul element de dupa elementul mai mic ca cel pivot, adica cel care a fost pus dupa pivot si am memorat pozitia. Apoi, vom memora pozitia acestui element nou care e mai mic ca pivotul.Elementele mai mari ca pivotul stau acolo unde sunt.

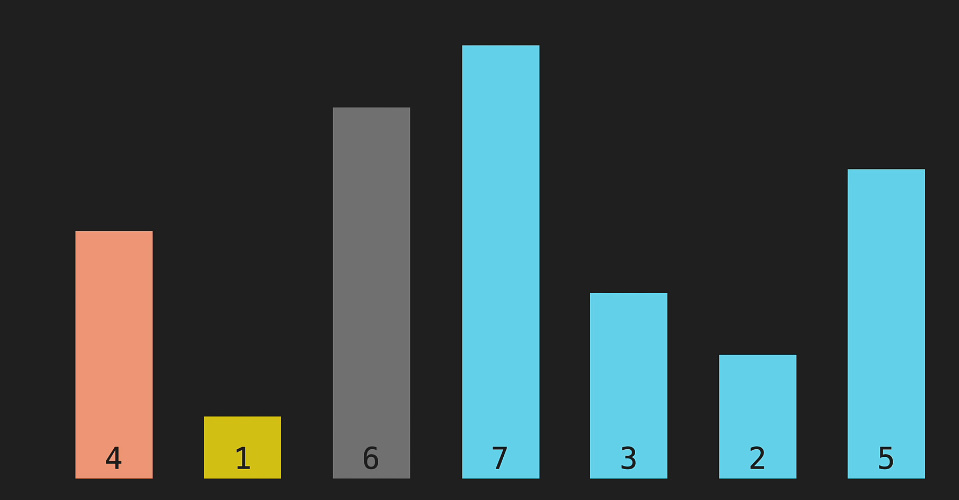
De ex:

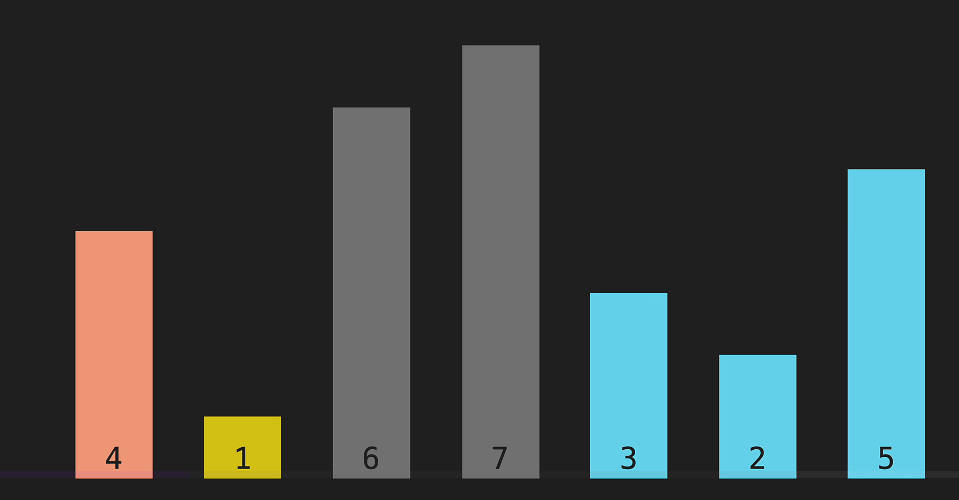


* Deci, luam pivot primul element, adica 4
* acum fiecare element trebuie comparat cu 4. Este necesar sa memorizam pozitia elementului curent care e mai mic ca 4, ca sa stim unde sa punem urmatoarele elemente. Deocamdata nu avem un element mai mic ca 4 gasit, deci vom salva pozitia lui 4 in loc de acel element negasit inca, caci oricum primul element mai mic va sta dupa 4
* Deci, 6 e mai mare ca 4. Ramane pe loc.

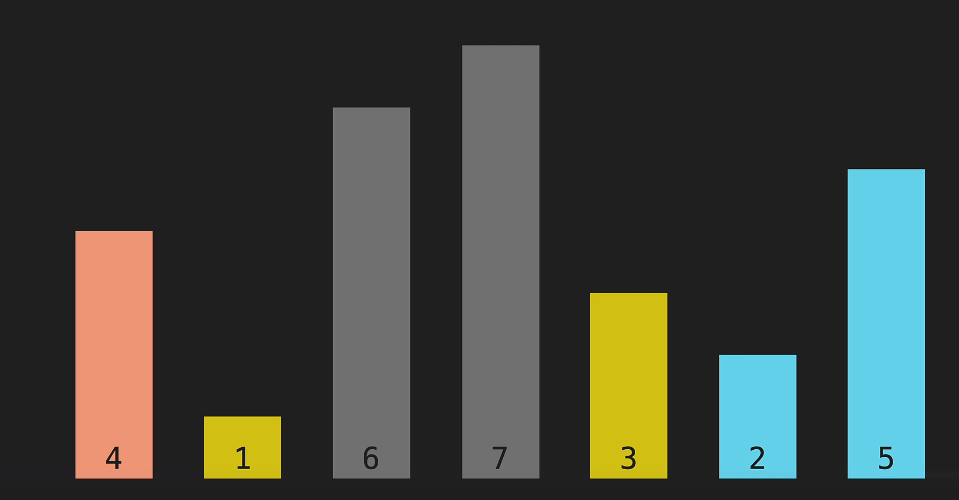


* acum, comparam 1 cu 4. 1 este mai mic, si nu avem niciun element mai mic ca 4 salvat, deci el e primul, si el va fi pus in locul elementului ce este dupa 4, adica va trece in locul lui 6 si 6 in locul lui. Acum 1 e primul element mai mic ca 4 si-i memoram pozitia

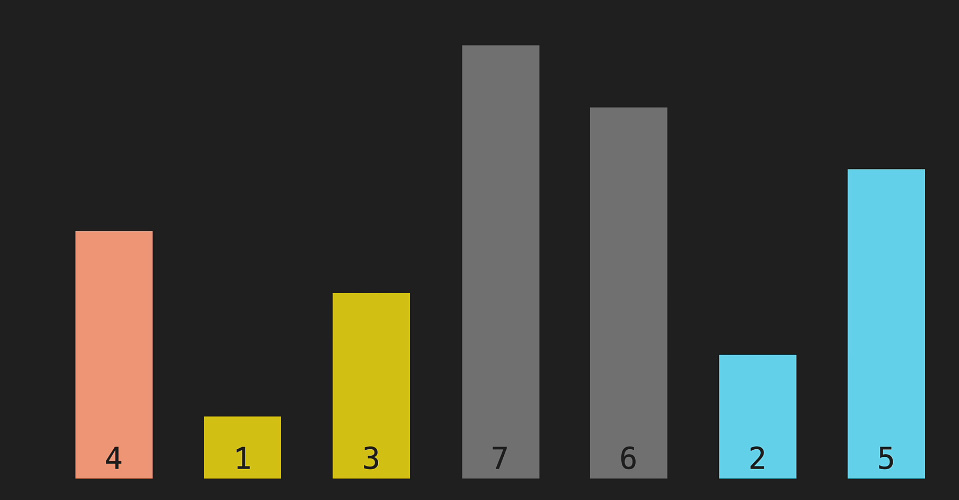


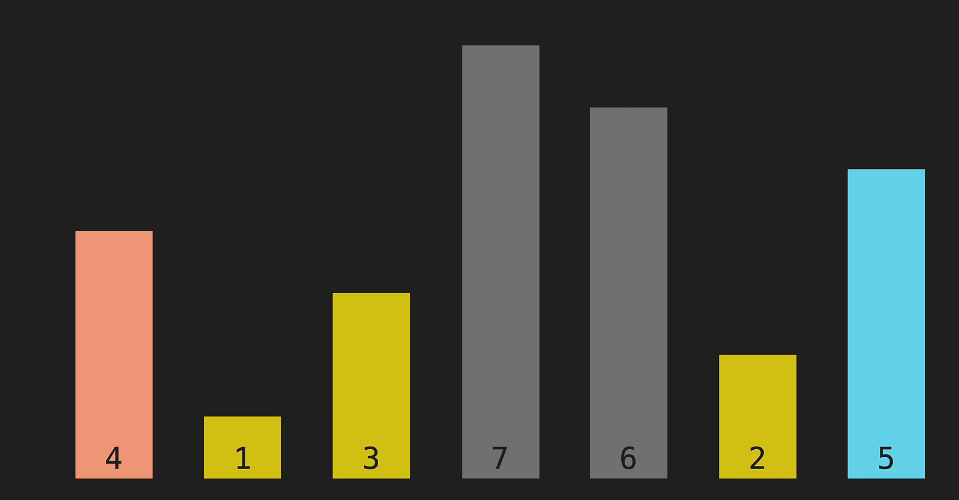


* 7 > 4
* Deci 7 sta pe loc



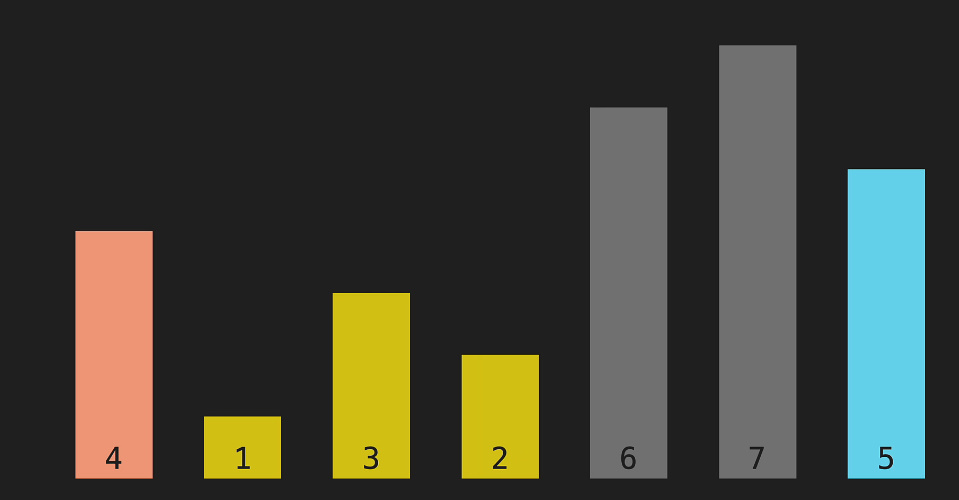
3 < 4, deci il schimbam cu urmatorul element ce e dupa ultimul mai mic element gasit mai mic ca 4, adica dupa 1, si acesta e 6, asa dar il schimbam pe 3 cu 6



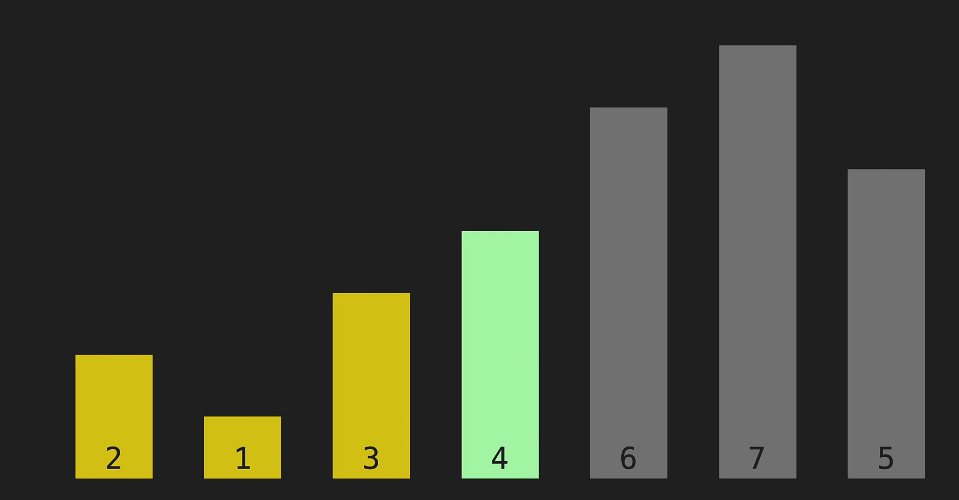


2 < 3

deci, 2 trece in locul la elementul care este dupa elementul mai mic ca pivotul gasit, adica 3, deci 2 trece in locul lui 7 si 7 in locul lui 2



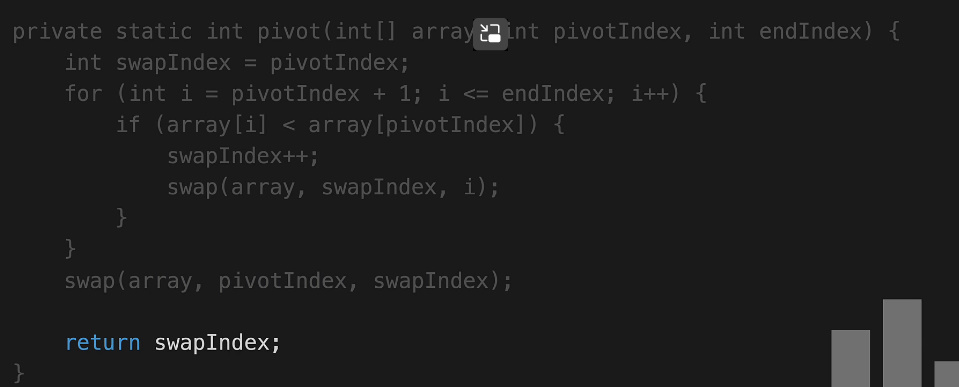
5 e mai mare deci sta pe loc

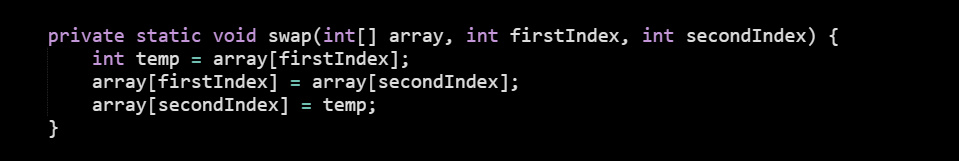


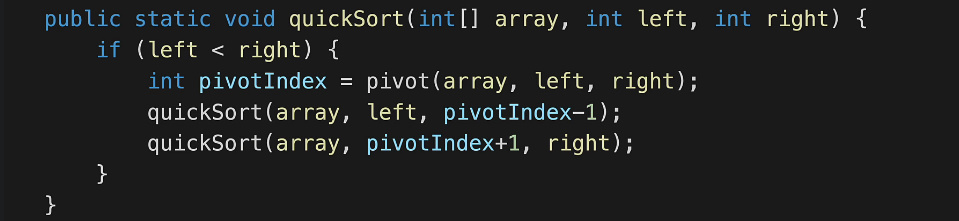
Acum 4 si ultimul element mai mic ca 4 gasit se schimba cu locul.

Acum sortam sub array din stanga lui 4 si dreapta tot asa

* Avem nevoie de o metoda care sa aranjeze fiecare element pivot din array la locul lui, cu elementele plasate in stanga si dreapta corespunzator
* **Pivote() method**







* Quick Sort e mai bun ca merge sort deoarece nu creaza niciun nou array, ci lucreaza cu cel original, asa dar space complexity e O(1)
* Time complexity:

O(nlogn)

n – selectarea si compararea itemelor din array intr-o bucla for

logn – despartirea array in subarrays

Totusi, O(nlogn) e best case, caci cel mai rau caz e cand avem deja toate datele sortate, si deja vom avea O(n^2), asa dar average case O(nlogn). Asta e din cauza ca daca de ex avem 8 iteme, si toate sortate, va trebui sa parcurgem fiecare item si il comparam cu toate elementele si asa vor fi 8 foruri cu inca un for in ele, deci n\*n, dar cand sunt nesortate, vom avea doar 3 splituri cu comparari, asa cum cand avem un array cu un element, nu se compara nimic.