Grzegorz Łoszewski

12.2022

SORTING ALGORITHMS REPORT

BUBBLE SORT

Bubble Sort is the most simple form of a sorting algorithm. It works by comparing the value of current element(n) with the one after it(n+1) and swapping the values with each other if n > n+1, then - it compares an element n+1 with an element n+2, etc.

The algorithm runs through the entire list of values for a couple of times until it is fully sorted.

That is what makes this algorithm very unefficient for large data, as its time complexity is very high.

EXAMPLE:

Input: Array[] = {6, 2, 3, 1}

1st run:

First, we compare values of first two elements
{6, 2, 3, 1} - 6 > 2, so the program swaps the values **->** {2, 6, 3, 1}
Then, we move on to next elements (6 and 3)
{2, 6, 3, 1} - 6 > 3, swap the values **->** {2, 3, 6, 1}
{2, 3, 6, 1} - 6 > 1, swap the values **->** {2, 3, 1, 6}

2nd run:

{2, 3, 1, 6} - 2 < 3, don't swap
{2, 3, 1, 6} - 3 > 1, swap the values **->** {2, 1, 3, 6}
{2, 1, 3, 6} - 3 < 6, don't swap

3rd run:

{2, 1, 3, 6} - 2 > 1, swap the values **->** {1, 2, 3, 6}
{1, 2, 3, 6} - 2 < 3, don't swap
{1, 2, 3, 6} - 3 < 6, don't swap

4th run:

{1, 2, 3, 6} - 1 < 2, don't swap
{1, 2, 3, 6} - 2 < 3, don't swap
{1, 2, 3, 6} - 3 < 6, don't swap

Algorithm didn't swap anything on the 4th run which means that we have a fully sorted array **->** {1, 2, 3, 6}

INSERTION SORT

Insertion Sort is similar to sorting adeck of playing cards.
It divides the input array into sorted and unsorted part.
First, we take any element from the **unsorted array** (usually the first one) and place in in the **sorted array.**
Then we take any element from **unsorted array**, compare it with all values in the **sorted array** until we reach an element that is equal or bigger in value and we place it in a sorted manner

EXAMPLE:

Input: Array[] = **{6, 2, 3, 1}**

We pick the first element and put it as a "sorted element"
{6, 2, 3, 1}

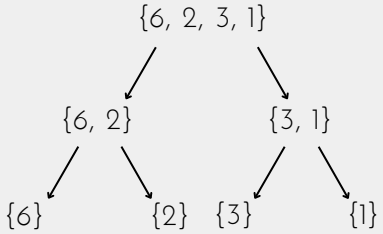
Then we take the next value and compare it our sorted elements
Since **6** > **2** we need to place **2** behind **6**
{2, 6, 3, 1}

We do the same thing with all elements
3 < **6** so it needs to be placed on the left side of **6**
But **3** > **2** so it needs to be placed on the right side of **2**
{2, 3, 6, 1}

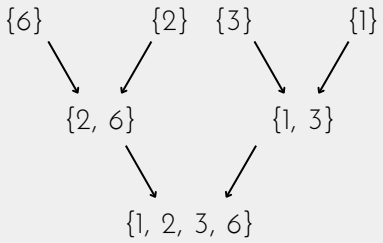
1 < **2** < **3** < **6**, so:
{1, 2, 3, 6}

MERGE SORT

Merge Sort is an efficient and very stable sorting algorithm.
It is based on recursion, division and merging.
We need to divide our array into to equal parts in a recursive manner until we reach single elements.



Then we compare those values and swap the elements if necessary.



QUICK SORT

Quick Sort is an efficient but unstable sorting algorithm.
It is based on choosing a "**pivot**" element.

Then our array is divided into 2 parts:

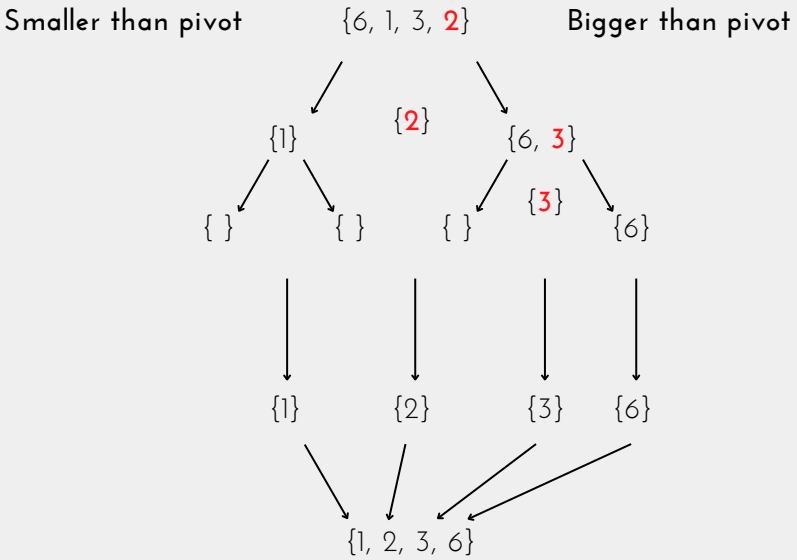
- elements **bigger** than **pivot** (**right** side of **pivot**)
- elements **smaller** than **pivot** (**left** side of **pivot**)

If the value is **bigger** than **pivot** -> we place it on the **right** side
If the value is **smaller** than **pivot** -> we place it on the **left** side

The easiest way to select a **pivot** is just to use the last element of our array.

Algorithm is executed recursively by performing all of those operations until we are left with just a single element.

EXAMPLE:
(**pivot** element will be highlighted in **red color**)
We choose ourlast element as pivot



At last, we need to merge our sorted values into a final, sorted array

COUNTING SORT

Counting Sort doesn't actually compare values. It assumes how big will the range of our numbers be.
It hashes(transforms any given key or a string of characters into another value) the value in a temporary array used for counting .
It later uses indexes of this temporary array, do define values in a sorted manner.

EXAMPLE:
Input: Array[] = **{6, 1, 3, 2, 3}**
Since, we know what theinput is, we can easily say that the **biggest** number is **6** and the **smallest** is **1**
So our range is from **1** to **6**

We then create and array for counting has indexes starting with the smallest number up to the biggest number
And we check how many elements of a specific index(element value) we have, by adding 1 for each iteration to each element
countArray[**1**] = 1
countArray[**2**] = 1
countArray[**3**] = 2
countArray[4] = 0
countArray[5] = 0
countArray[**6**] = 1

Then we add each index starting from 1 to aour sorted array, as many times as the value is on each index
sortedArray[] = {1, 2, 3, 3, 6}, becauese we have:
1 element of value **1**
1 element of value **2**
2 elements of value 3
1 element of value **6**