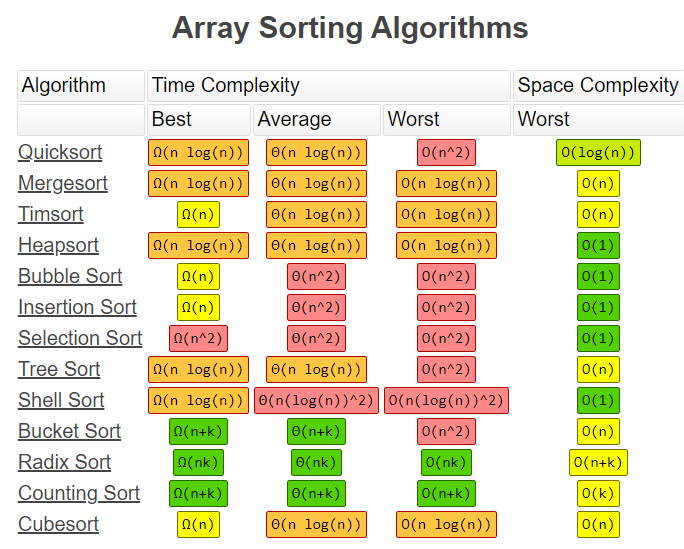
# **SORTING ALGHORITHM**

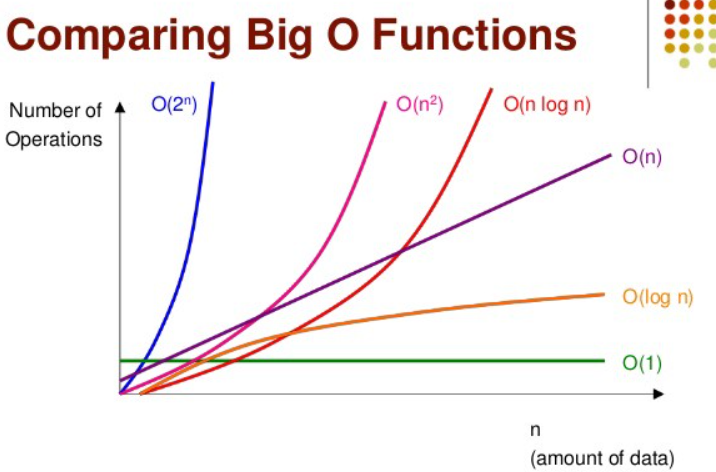
Different types of sorting algorithms

* **Stable and not-stable** – stable algorithm does not change an order of similar elements
* **In-place and not in-place** – in-place does not require an additional space (in-place: quicksort, insertion sort)
* **Comparison and not comparison** – not comparison are bucket, radix, counting

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **BUBBLE SORT** | Ω(n2) | Θ (n2) | O(n2) | O(1) | Yes | Yes | Yes |
| **SELECTION SORT** | Ω(n2) | Θ (n2) | O(n2) | O(1) | Yes | Yes | Yes |
| **INSERTION SORT** | Ω(n) | Θ (n2) | O(n2) | O(1) | Yes | Yes | Yes |
| **SHELL SORT** | Ω(nlog(n)) | Θ (n(log(n)) 2) | O (n(log(n)) 2) | O(1) | Yes | No | Yes |
| **MERGE SORT** | Ω(nlog(n)) | Θ (nlog(n)) | O(nlog(n)) | O(n) | No | Yes | Yes |
| **QUICK SORT** | Ω(nlog(n)) | Θ (nlog(n)) | O(n2) | O(log(n)) | Yes | No | Yes |
| **COUNTING SORT** | Ω(n+k) | Θ(n+k) | O(n+k) | O (k) | No | Yes | No |
| **RADIX SORT** | Ω(n\*k) | Θ(n\*k) | O(n\*dk) |  | No | Yes | No |
| **BUCKET SORT** | Ω(n+k) | Θ(n+k) | O(n2) | O(n) | No | N/A | No |
| **HEAP SORT** |  |  |  |  |  | No |  |

Note: the characteristic [stable] for bucket sort depends on sorting algorithm what you will choose, because bucket sort is a distribution technique



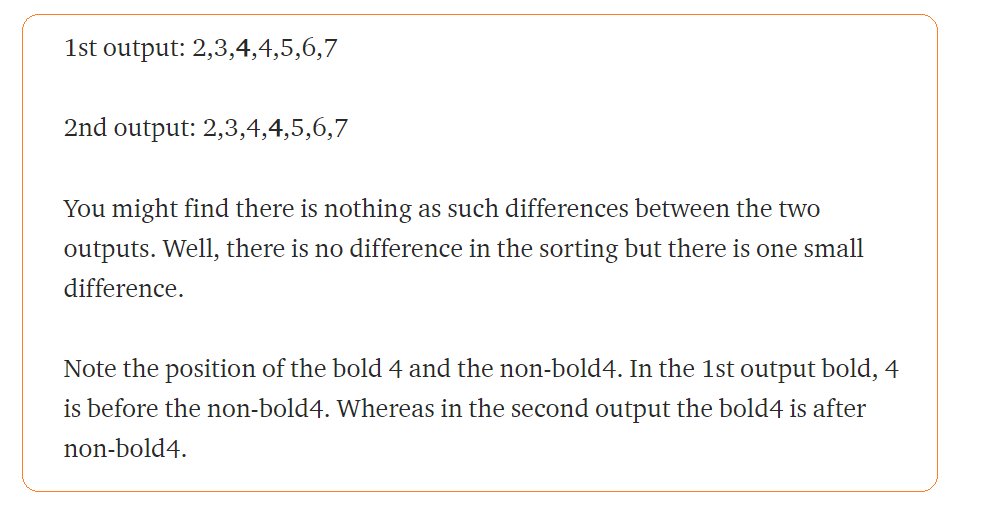


# **STABLE VS UNSTABLE SORTING**

**Stable sorting** - does not change the order of ‘like’ elements in the array

A sorting algorithm is **said to be stable** if it maintains the relative order of numbers/records in the case of tie i.e. if you need to sort **1 1 2 3** then if you don't change order of those first two ones than your algorithm is stable, but if you swap them then it becomes unstable, despite the overall result or sorting order remain same.

(!) **quicksort is unstable but merge sort is a stable sorting algorithm**.



# 

# **NOT COMPARISON SORT**

Radix sort, counting sort, bucket sort

Pros:

Cons:

* Depends on many factors
* A lot of memory especially in counting sort(if there is a big element number)
* Depends on sorting technique which a are stable

# **BUBBLE SORT**

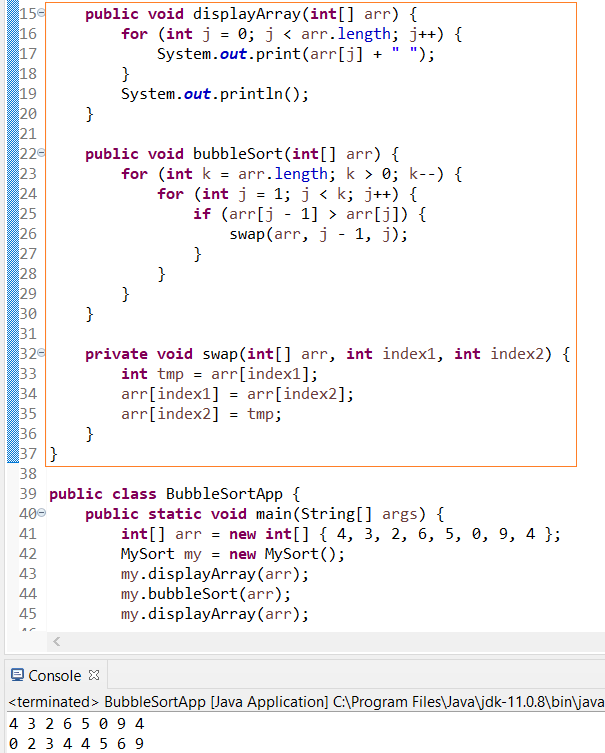
**Bubble sort** – the slowest sorting technique. It compares adjacent elements and move max value to the right

**+:**

* Simple. It is used in educational purposes
* In-place – does not use a lot of extra space

**-:**

* Slow. Best, Average and Worst time complexity is quadratic
* Almost nobody use it



# **SELECTION SORT**

**Selection sort** – it is also slow sorting technique. It find the lowest element in array an move to the left It can be useful in application when an operation of swapping is an expensive and it can minimize a number of swaps

* This algorithm is called selection sort because it repeatedly selects the next-smallest element and swaps it into place

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** |
| **SELECTION SORT** | Ω(n2) | Θ (n2) | O(n2) | O(1) |

**Implementation**:

* Choose the lowest element and compare with the first element. Put on the first place if found element is lowest
* Repeat the same for interval [2..n]
* Repeat until length of array>0
* We start algorithm from position 1 and when found smallest compare with the [0] element

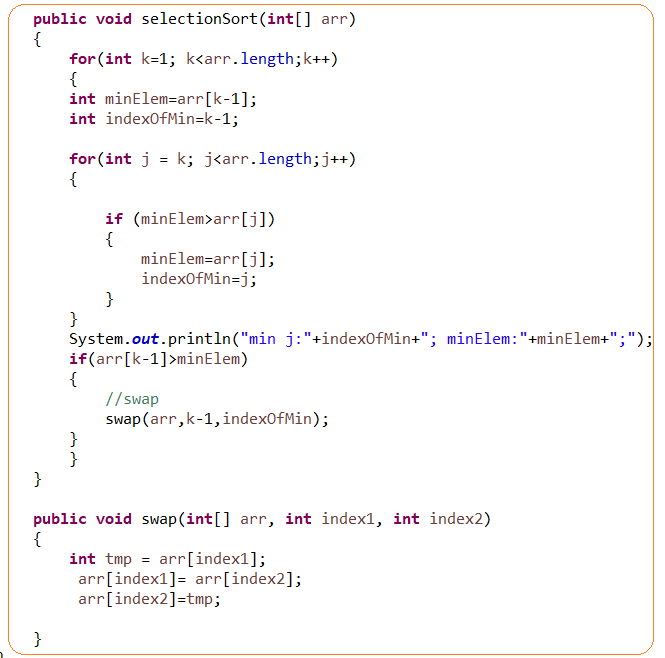
**pros:**

* It can be useful in application when an operation of swapping is **an expensive** and it can minimize a number of swaps

**cons:**

* O(n2) time complexity





# **INSERTION SORT**

**Insertion sort** – it is one of the efficient sorting algorithm for small arrays.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** |
| **INSERTION SORT** | Ω(n) | Θ (n2) | O(n2) | O(1) |

**Implementation**:

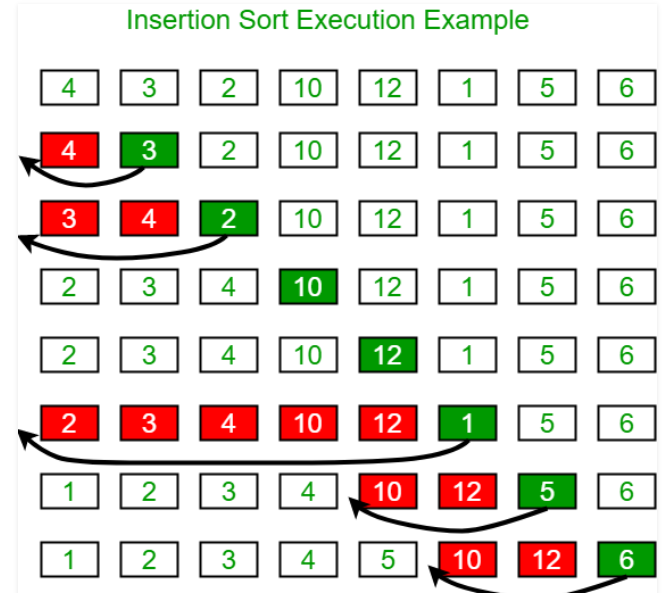
* Choose element [1] and compare with [0] – swap if it is less
* Repeat for [2] and compare with [0],[1]
* Repeat until length of array>0

**pros:**

* Only efficient for small data sets
* The most efficient through slowest techniques like bubble and selection sort
* Can take O(n) time complexity if algorithm already sorted or almost sorted
* O(n2) – is the worst time complexity. It can happen when an array in a reverse order

**cons**:

not efficient for large data



# **SHELL SORT**

**Shell sort** – is an extension of insertion sort. If an array is too large we need , for example, to compare and return element back, back, back … that is an expensive operations. So, we use a special technique to reduce a number of swaps(movements) operations. It means we prepare an array as partially sorted and when we apply an insertion sort method in a final stage we reduce a number of movements

**Implementation**:

* We break an original array and breaks into sub arrays and **sort using insertion algorithm sort on that chunk**. We sort on “the fly” without storing anyway. Breaks is made based on [gap]. Gap – is just element how to divide array
* Repeat that until chunk length is more than 0
* Finally, apply an insertion sort again

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **SHELL SORT** | Ω(nlog(n)) | Θ (n(log(n)) 2) | O (n(log(n)) 2) | O(1) | Yes | No | Yes |

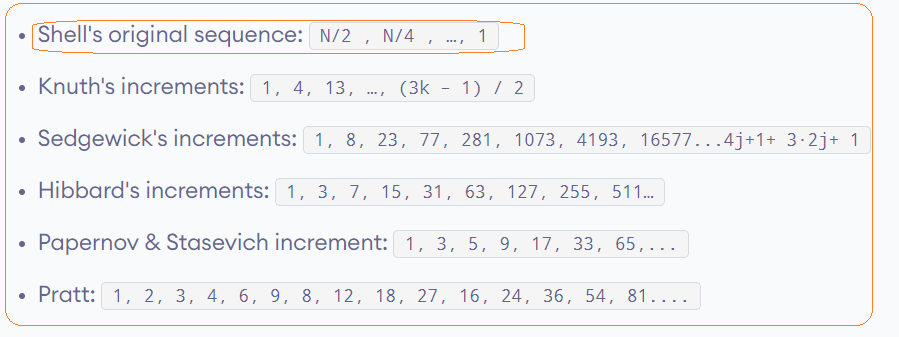
**pros**:

* advanced technique among slowest algorithms. It can be used as enhancement of insertion sort
* worst time complexity can be O (n(log(n)) 2), that is less than O (n 2).

**cons**:

* still slow
* unstable

Gap can be chosen in any way. The original way is a divide n=length of array/2: n/2, (n/2)/2,…

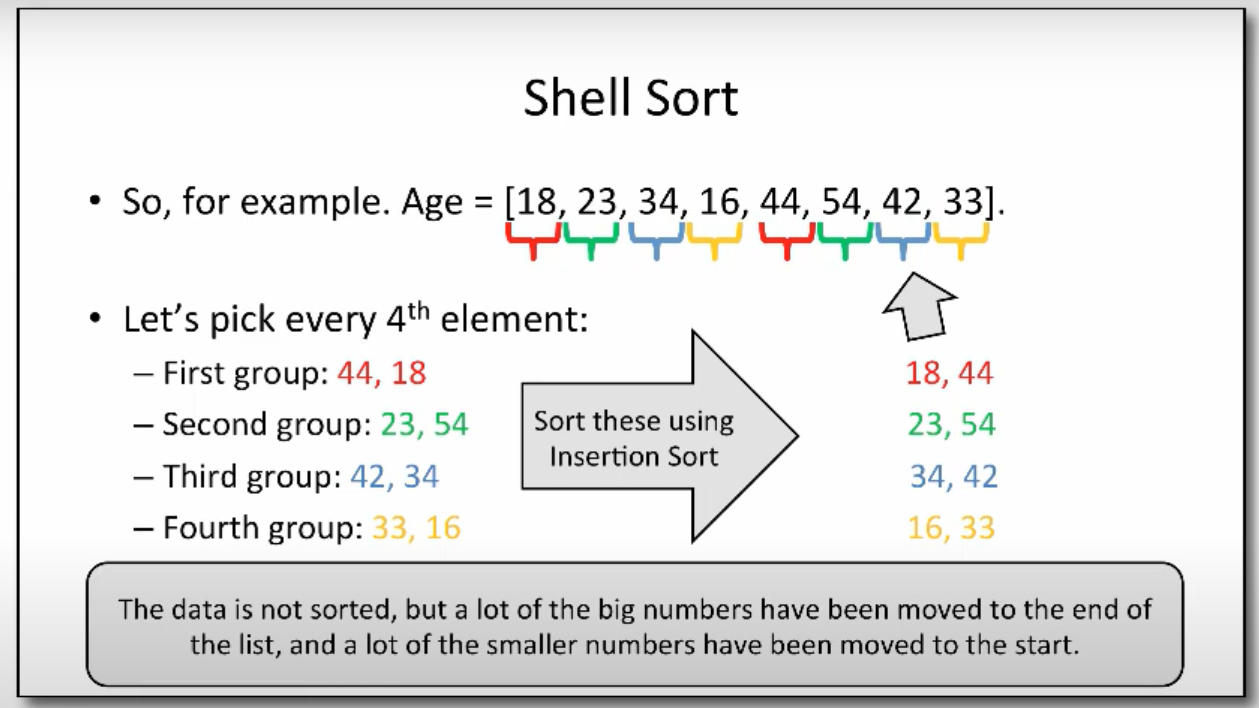


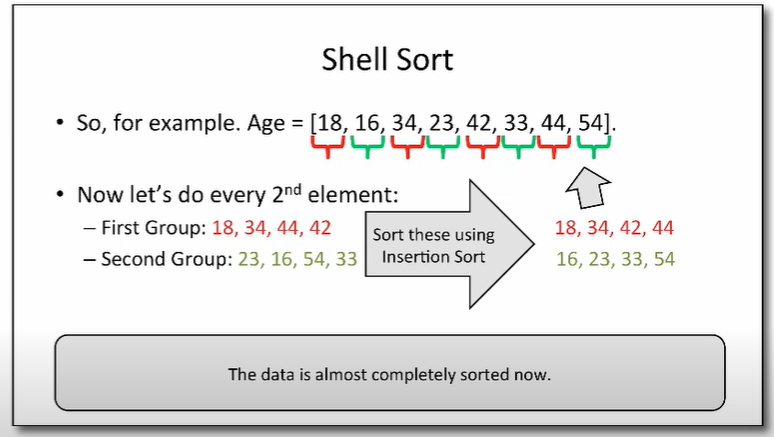
For example, len=11. N=11/2=5; N=5/2=2;

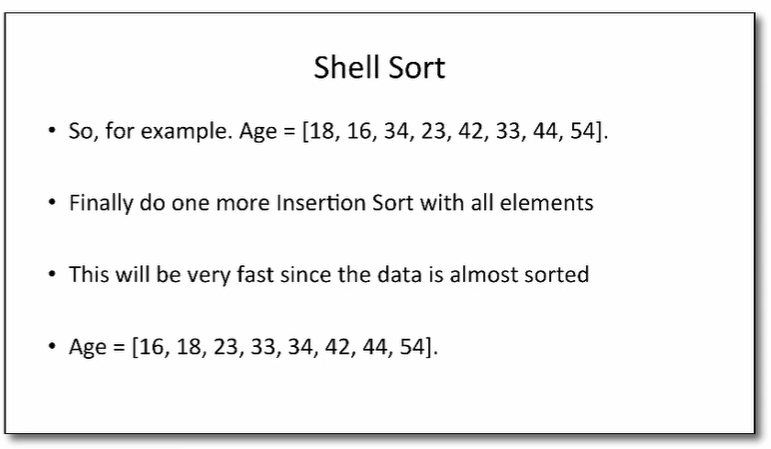




It could help to get to a best scenario and reduce a number of swaps when we apply insertion sort technique









# **MERGE SORT**

Merge Sort – is a divide and conquer algorithm. It has two steps divide into two parts in a middle and [sort and merge ]

Facts:

* Merge sort generally performs less comparisons than quick sort both worst case and on average
* **Merge sorted is more preferred for LinkedList and QuickSort for arrays**

**Implementation**:

1. **Divide** (split): In this step, the input array is divided into 2 halves, the pivot is the midpoint of the array. It recursively repeats until the length is 1. Each divided part is stored in array
2. **Conquer** (sort and merge): In this step, we sort and merge the divided arrays from bottom to top and reach towards our sorted array.

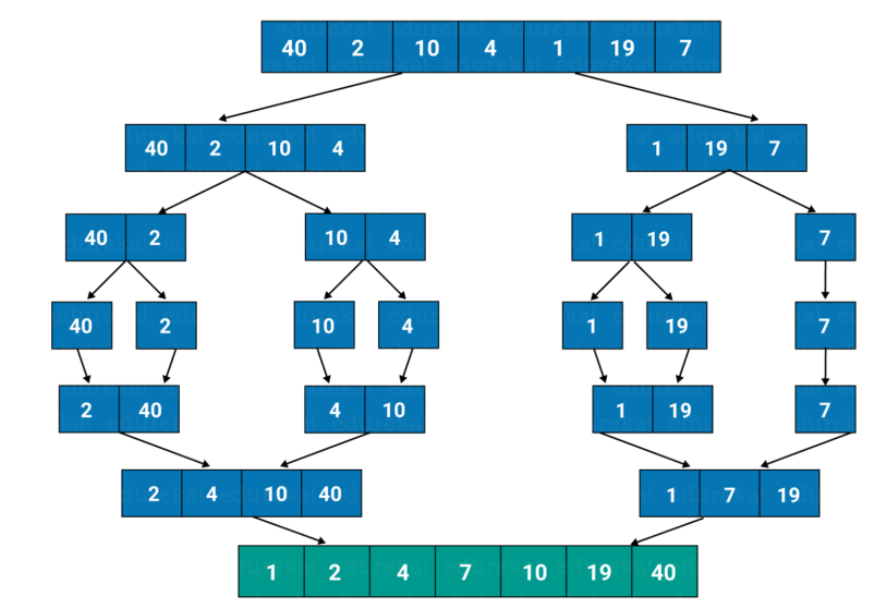
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **MERGE SORT** | Ω(nlog(n)) | Θ (nlog(n)) | O(nlog(n)) | O(n) | No | Yes | Yes |

**pros:**

* **Powerful technique for larger datasets**
* **Merge sorted is more preferred for LinkedList and QuickSort for arrays**
* Best, average and worst time complexity is the same: Ω(nlog(n)), Θ (nlog(n)), O(nlog(n))
* Stable
* It can be applied to files of any size
* Useful when data is a different locations like cache, memory
* merge operation of merge sort can be implemented without extra space for linked lists.

**cons:**

* it requires additional space to store splitted arrays. It can be an issue for large data sets
* not super efficient for small datasets
* it goes a full process even if array sorted or partially sorted



# **QUICKSORT**

QuickSort – is divide and conquer algorithm.

**Implementation**:

* Choose a pivot element and move element less than pivot to the left and greater to the right
* Recursively repeats it

Facts:

* Worst-case time complexity for quicksort is O(n²), although this is an algorithm that rarely falls into its worst-case performance, especially with minor amounts of customization. Typically the Big-O for quicksort is O(n log n). However, a worst case of O(n²) is a knock against it.
* It ironically named because it’s really not even the fastest
* **Merge sorted is more preferred for LinkedList and QuickSort for arrays**

It chooses the [pivot] point and partitioning the collection of data around the pivot

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **QUICK SORT** | Ω(nlog(n)) | Θ (nlog(n)) | O(n2) | O(nlog(n)) | Yes | No | Yes |

**Pros**:

* unlike quicksort and heapsort, and can be easily adapted to operate on linked lists
* In place
* **Merge sorted is more preferred for LinkedList and QuickSort for arrays**

**Cons**:

* O(n2) – is the worst time complexity. It cat happen when a pivot element is chosen badly
* Not stable

**Pivot** – is an arbitrary element.

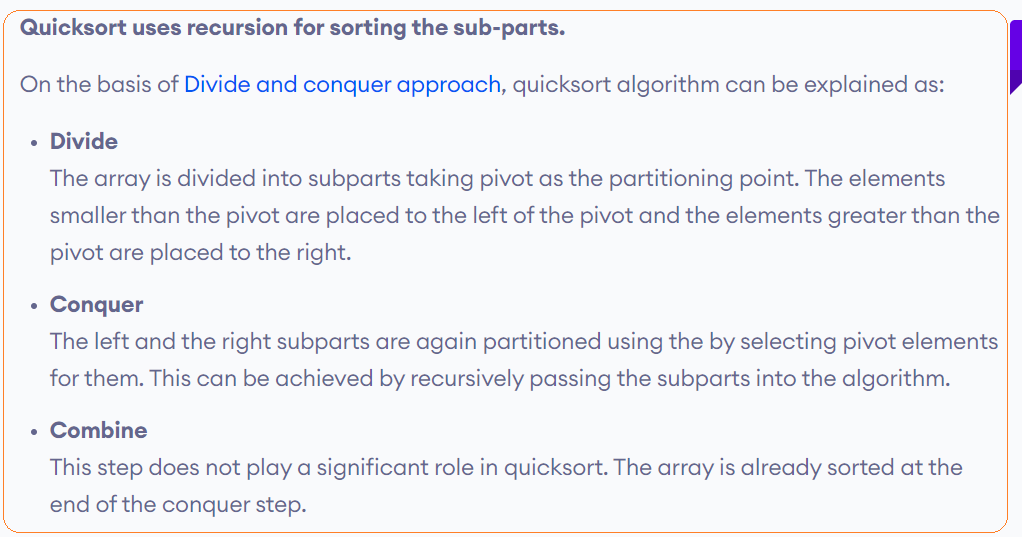
Pivot element can be chosen as

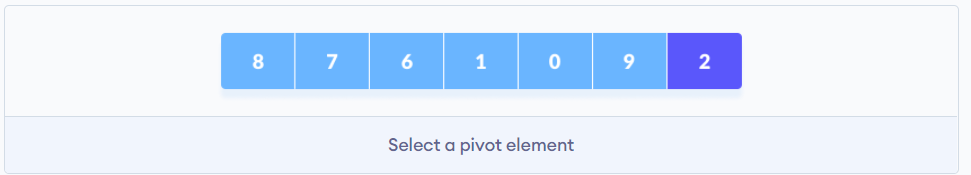
* First element
* Last element
* Medium
* Randomly

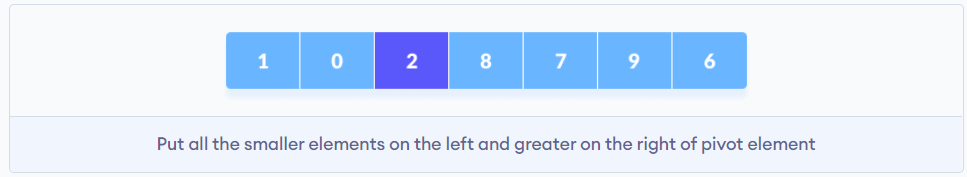
Note: poor chosen element provides the worst time complexity of algorithm as O(n)

There are two worst cases:

* You sort already sorted array and choose as a pivot the first element
* A lot of same elements













# **COUNTING SORT**

**Counting sort** is – an effective sorting algorithm to sort small integers in a linear time

* Not-comparison method

**pros**:

it can be perform even better than QuickSort if number of element are small

**cons**:

* only work for positive integers(including 0). It can not work for other types, because you can not calc frequency of elements
* not effective if there are big numbers, because you need to create an auxiliary array to keep track of all values up to the biggest

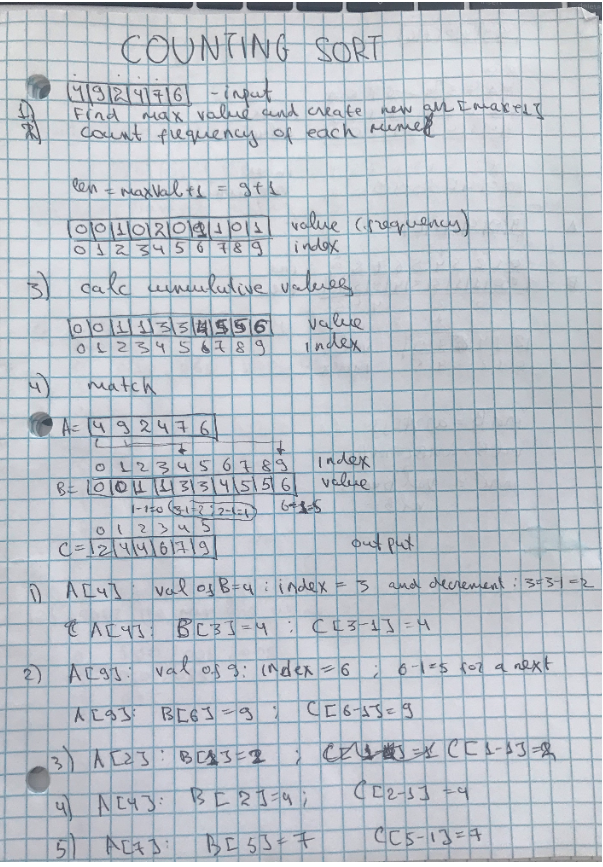
**Time complexity:**

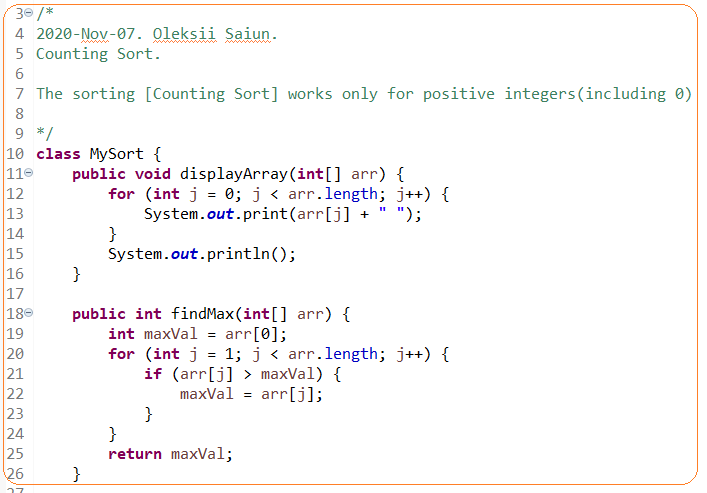
* Ω(n+k)
* Θ(n)
* O(n)

**Space complexity:**

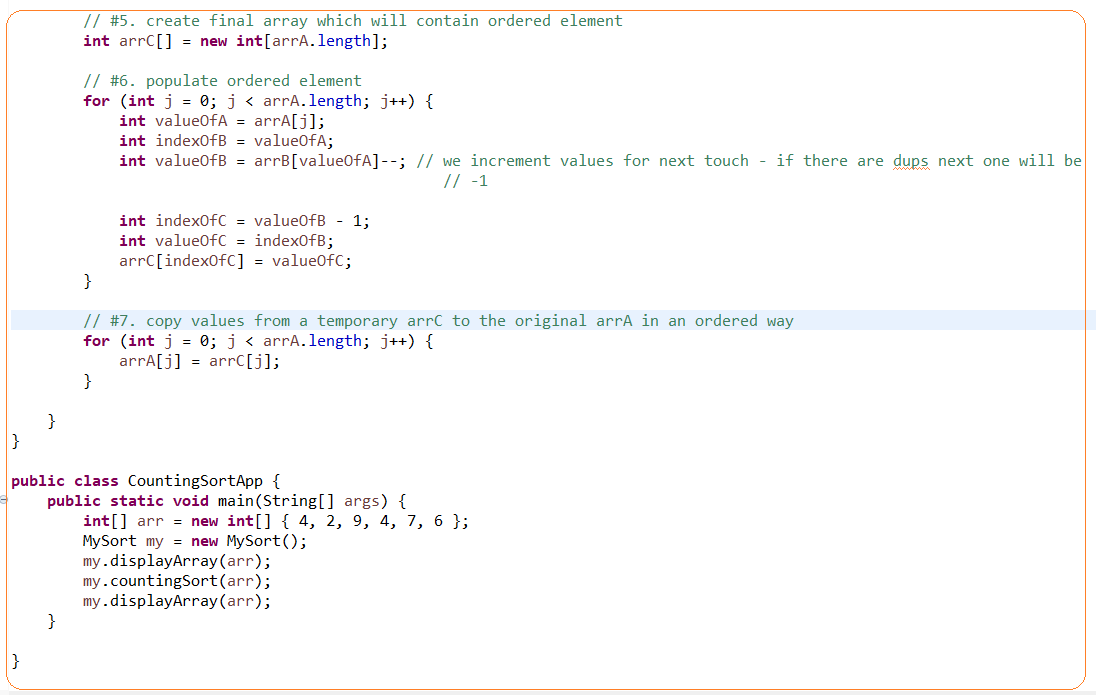
* O(n)

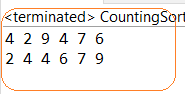
n-number of elements, k – the biggest value











# **RADIX SORT**

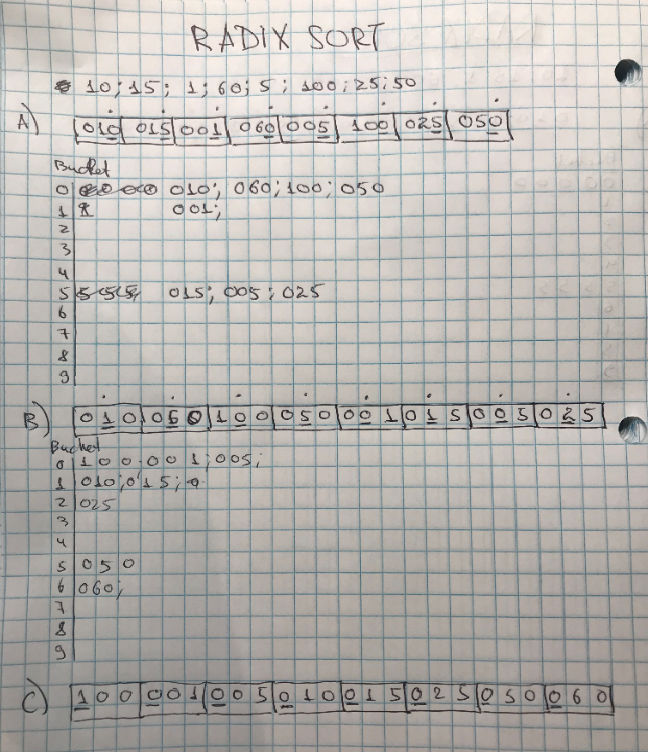
Radix sort – is a sorting algorithm that is used to sort numbers. We sort the numbers from least significant digit to the most significant digit.

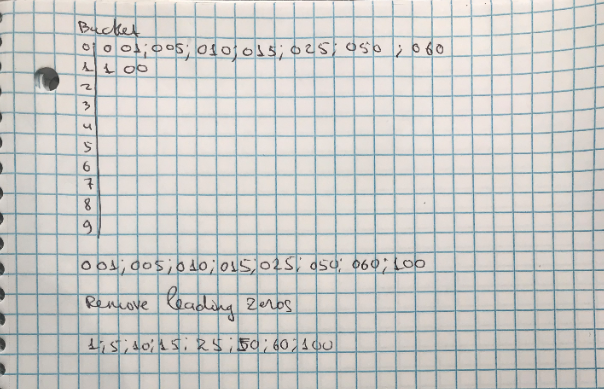
Note: Radix sort came from a real practice how machines sorted mails based on their zip codes

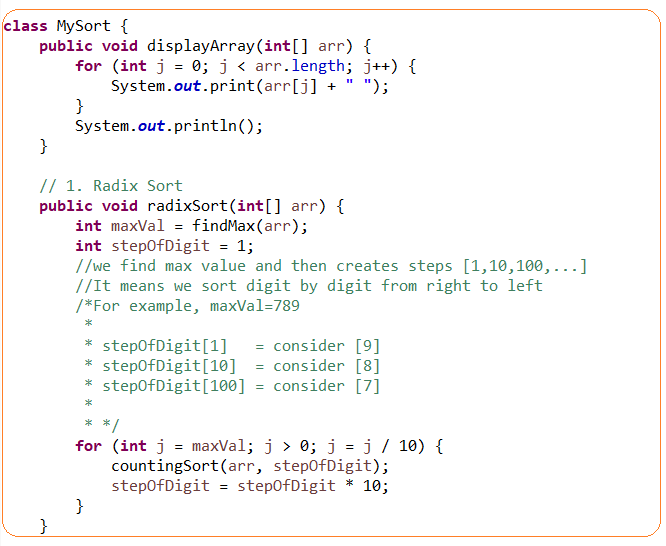
* Not-comparison method
* Radix sort is based on a counting sort. Counting sort is a stable algorithm. We need to use stable algorithm (do not move the same values), otherwise, radix sort will not work

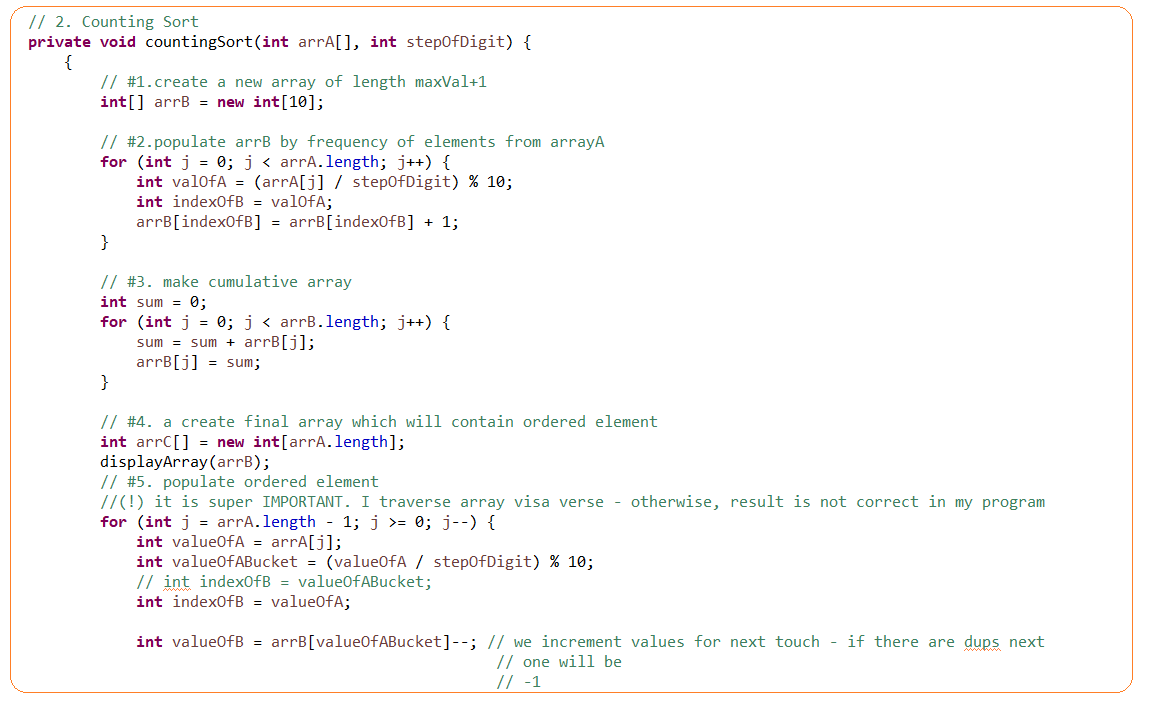
Implementation:

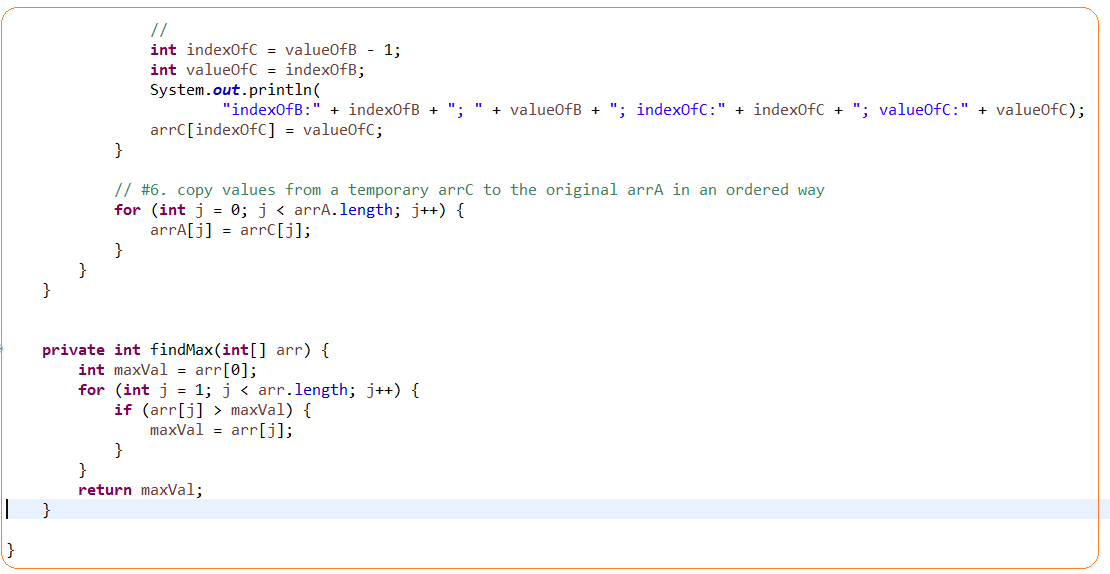
* First of all, we find the number of digits in the biggest number. How many digits are then we will need to perform the number of pass. For example, [9005,100,9000,2] – 4 max number of digits. We will have 4 pass.
* We will have to sort numbers digit by digit starting from the right and put them in a buckets

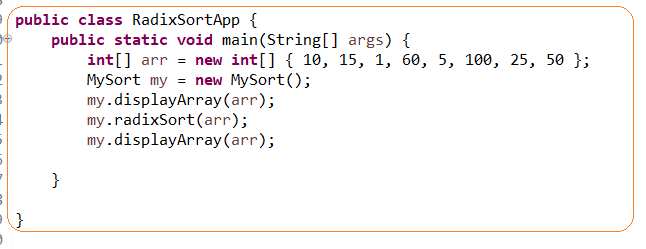


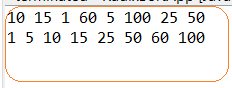












# **BUCKET SORT**

**Bucket sort** – it is not actual sorting technique it is a distribution technique that put elements into buckets in order to avoid a number of comparison and then sorting(any) algorithm apply.

* Can be effectively used when elements evenly distributed and then insertion sort can by applied.
* Radix sort is a special version of bucket sort
* Not-comparison method

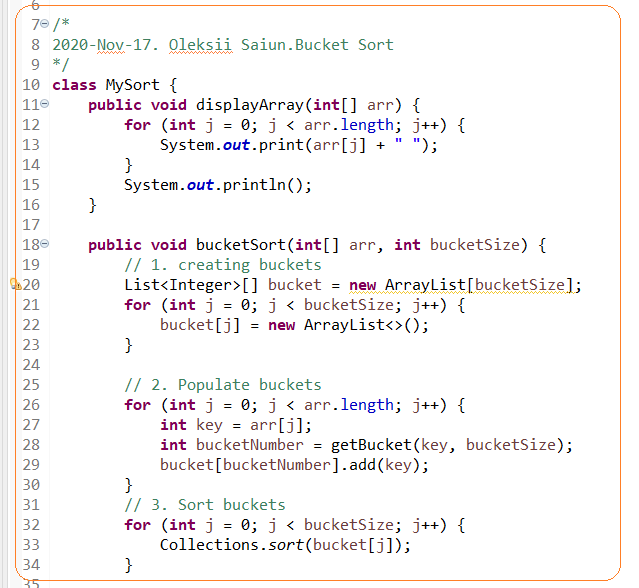
Implementation:

Choose

**Pros**:

* Can be effective if a distribution factor is chosen good
* Can be effective of buckets are sorted in parallel

**Cons**: if you choose a bad distribution factor then you will







# **#1.TASK. MOVE ZEROS TO THE RIGHT END**

