# **SORTING ALGHORITHM**

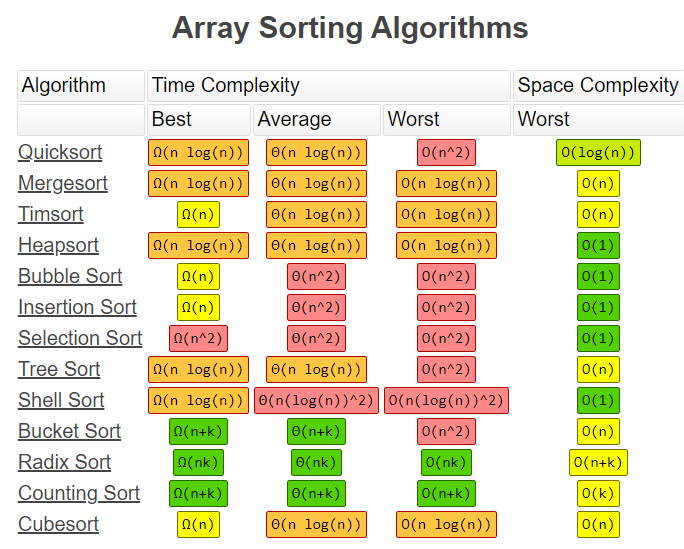
Different types of sorting algorithms

* **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
* **Adaptive and not adaptive**– adaptive algorithm do not sort already sorted element if it finds. Example is a trim sort
* **Efficient and not efficient**- examples of efficient algorithms are MergeSort, QuickSort, HeapSort. Efficiency is decided based on 4 factors: implementation, performance, time and space complexity
* **External sorting** is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive. Thus, external sorting algorithms are external memory algorithms and thus applicable in the external memory model of computation. Merge Sort – is an example

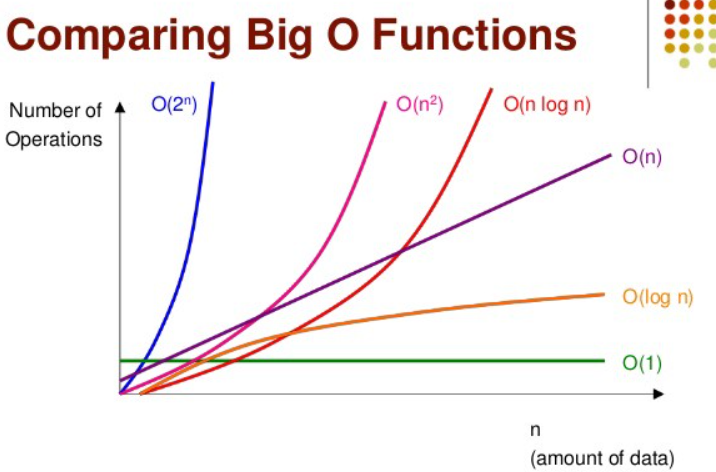
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **BUBBLE SORT** | Ω(n2) | Θ (n2) | O(n2) | O(1) | Yes | Yes | Yes |
| **SELECTION SORT** | Ω(n2) | Θ (n2) | 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\*k) | O (n+k) | No | Yes | No |
| **BUCKET SORT** | Ω(n+k) | Θ(n+k) | O(n2) | O(n) | No | N/A | No |
| **HEAP SORT** | Ω(nlog(n)) | Θ (nlog(n)) | O(nlog(n)) | O(1) | Yes | No | Yes |

Notes:

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



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Algorithm | description | implementation | Pros and Cons |
| **1** | **BUBBLE SORT** | **Bubble sort** – the slowest sorting technique. It compares adjacent elements and move max value to the right | **Implementation**:   * It takes the first element arr[0] and compare with each one if this elements is greater it swaps * It repeats for the next arr[1]… element | **pros:**   * Simple. It is used in educational purposes * In-place – does not use a lot of extra space   **cons:**   * 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 | **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 |
| **3** | **INSERTION SORT** | **Insertion sort** – it is efficient sorting algorithm for small arrays. You can achieve linear time in best case Ω(n) | **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 |
| **4** | **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  It could help to get to a best scenario and reduce a number of swaps when we apply insertion sort technique | **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 | **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 |
| **5** | **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:  1.Merge sort generally performs less comparisons than quick sort both worst case and on average  2.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. | **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 * **Merge sort is used in external sorting** * 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 |
| **6** | **QUICK SORT** | **QuickSort** – is divide and conquer algorithm. Idea is based on choosing a pivot element and shift elements according to this pivot element. It repeats recursively   * 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 | **Implementation**:   * Choose a pivot element and move element less than pivot to the left and greater to the right * Recursively repeats it | **Pros**:   * Good for small arrays * 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 * The worst-case running time for quicksort is O(n2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation * Not stable * Not good it there are a lot of repeating elements * Internal sorting |
| **7** | **COUNTING SORT** | **Counting sort** is – an effective sorting algorithm to sort small integers in a linear time   * Not-comparison method | Implementation:   * Find max value in array and creates an array of length=maxValue+1 * Calc cumulative values for new array * Return output based on matching values from original array and additional for frequency | **pros**:   * it can be perform even better than QuickSort if number of element are small * only good for small numbers * is used for implementation of radix sort   **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 |
| **8** | **RADIX SORT** | **Radix sort** – is a sorting algorithm in a linear time that is based on sorting digit by digit from right to left in buckets. It is useful to sort large integers.  It is based on counting sort and radix is used when counting does not work – there is a big max value in an array  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**:   * We create a bucket (from 0 to 9 range) and sort digit by digit from right to left. For example, [9005,100,9000,2] – 4 max number of digits. We will have 4 pass. * Each sort is based on counting sort * When data is sorted in a bucket we return them in sorted order for that digit | **Pros**:   * It is useful to sort large integers * Sorting can be achieved in linear time * one of the fastest sorting algorithms, but I n particular cases, not general * can be good for large numbers and when values are similar length   **Cons**:   * Radix sort needs to be rewritten if the type of data is changed. * If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting * Only for integers and strings data type * It is not use often especially in libraries, because there a lot of constraints |
| **9** | **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**:   * Create buckets based on some value – like hashvalue or key/arrayLength * Populate bucket and sort – values in buckets must not overlap; any sorting technique can be applied * Merge buckets where there are already sorted values | **Pros**:   * Efficient if there is a big data set and data evenly distributed * 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 get a data skew in a bucket and lower performance |
| **10** | **HEAP SORT** | Heap sort is based on a heap data structure.  It turns out that heap sort is a lot like selection sort in its logic: both algorithms find either the smallest or largest element, “select” it out, and put that item in its correct location in the sorted list.  However, as similar as they are, heap sort is much better than selection sort in one massive way: its performance! Heap sort is basically a super-improved version of selection sort. Yes, it does find the largest element in an unsorted collection and orders it at the back of the list — however, it does all of this work so much faster than selection sort would! | **Implementation**:   * Create and populate a heap * Eliminate element from a root: you swap elements from a root from last element and then heapify new element in a root. It repeats for all remaining elements in array(according to pointer) | **Pros**:   * Very useful if need to find top k elements * An efficient way to build PriorityQueue (there are two options of PriorityQueue – max-priority queue and min-priority queue) * Heap sort is particularly suitable for sorting a huge list of items. * **Job Scheduling** - In Linux OS, heapsort is widely used for job scheduling of processes due to it's O(nlogn) time complexity and O(1) space complexity. * **Graph Algorithms** - It can used in the implementation of priority queue for Djikstra's algorithm, Prim's algorithm and Huffmann encoding as well.   **Cons**:   * Is not stable   Although Heap Sort has O(n log n) time complexity even for the worst case, it doesn't have more applications ( compared to other |

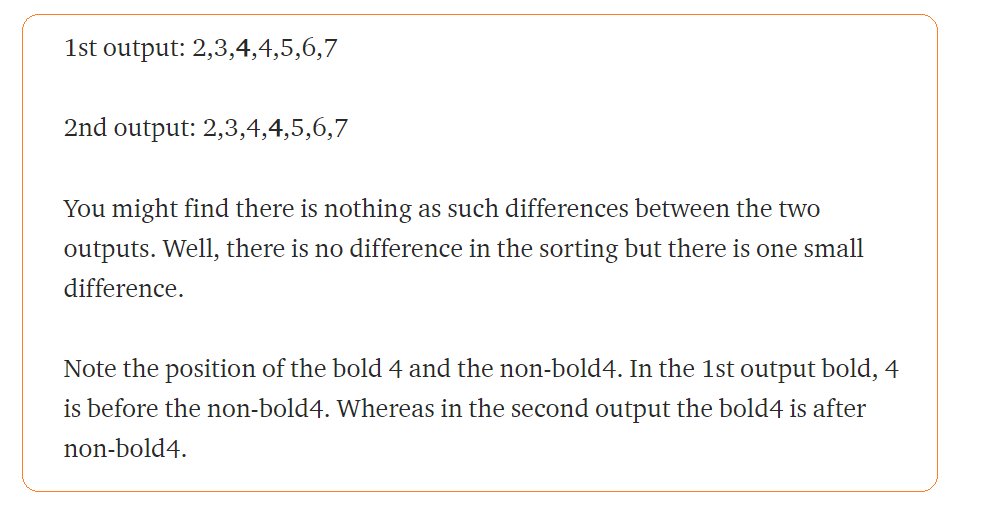


# **STABLE VS UNSTABLE SORTING**

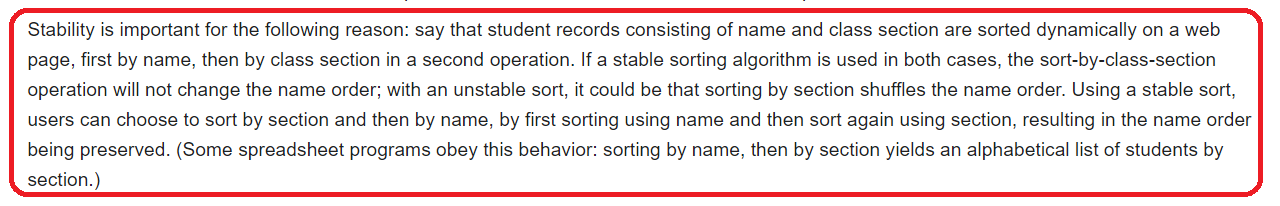
**Stable sorting** - does not change the order of ‘like’ elements in the array

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**.



# 



Note: **if there is a surrogate key then for rows withs the same value an order of SK might be changed after unstable sorting**.

# **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

# **MERGE SORT VS QUCIKS SORT**

|  |  |  |
| --- | --- | --- |
|  | MERGE SORT | QUICK SORT |
|  | Pros:   * Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets * Merge sort is preferred for linked lists * It has a consistent speed on any size of data * 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. * Merge sort is used in external sorting   Cons:  Use additional space | Pros:   * One of the fastest general algorithm * Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets. * Quick sort is preferred for arrays   Cons:   * For large data sets it is better to use merge sort * Internal sorting |
| Divide and conquer | yes | yes |
| In place | no | yes |
| Stable | yes | no |
| Large data sets | better | Not better |
| Worst case complexity | Ω(nlog(n)) | O(n2) – but it can easily avoid if choose good pivot element |
| Difference | **The main difference between quicksort and merge sort is that the quicksort sorts the elements by comparing each element with an element called a pivot while merge sort divides the array into two subarrays again and again until one element is left** |  |

# **BUBBLE SORT**

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

**Implementation**:

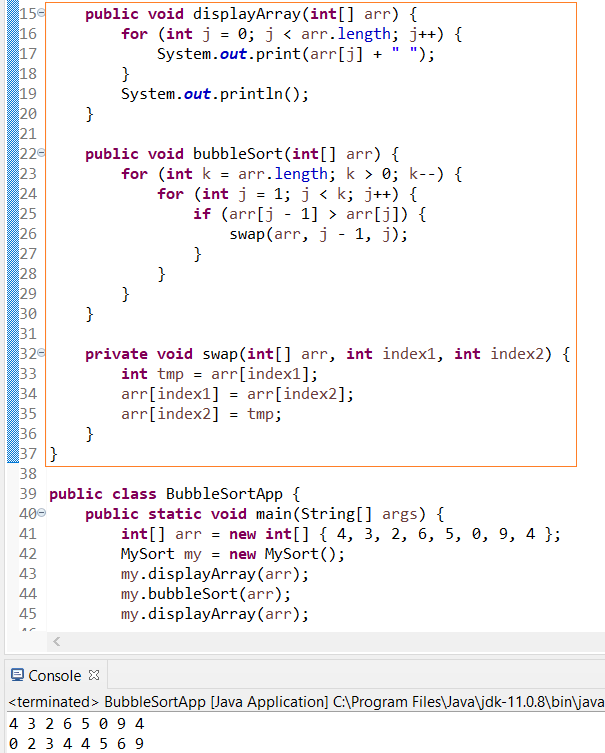
* It takes the first element arr[0] and compare with each one if this elements is greater it swaps
* It repeats for the next arr[1]… element

**pros:**

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

**cons:**

* 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

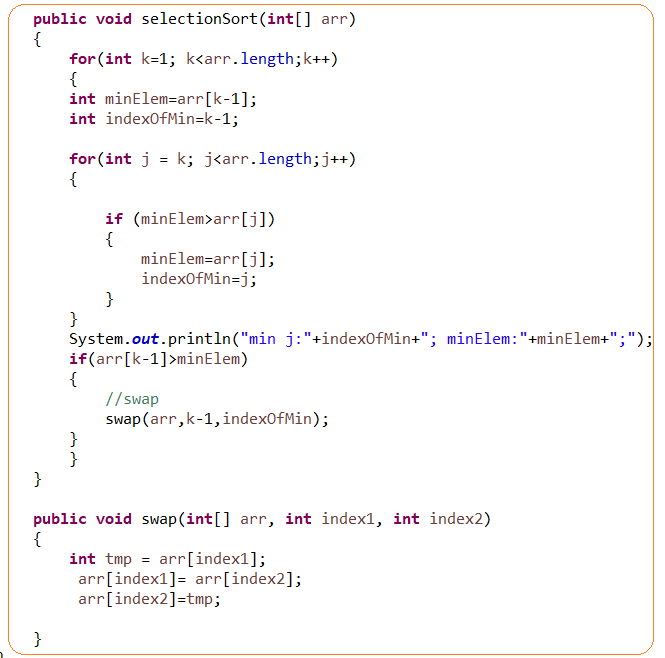
**pros:**

* 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 efficient sorting algorithm for small arrays. You can achieve linear time in best case Ω(n)

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

**Implementation**:

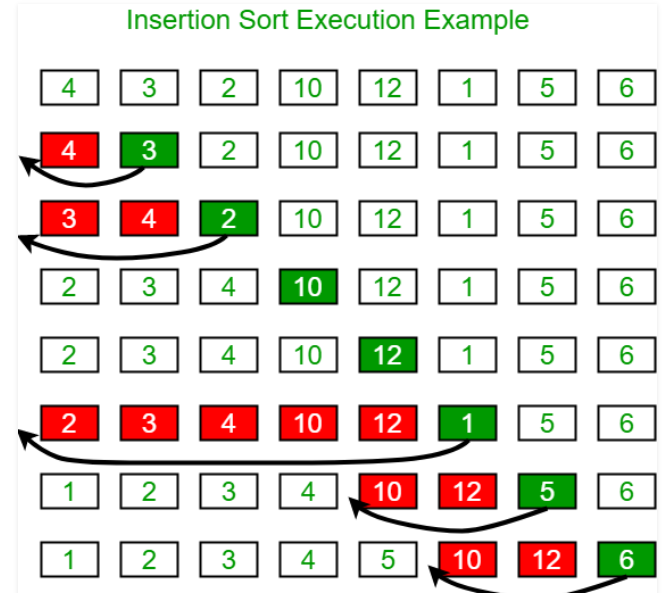
* Choose element [1] and compare with [0] – swap if it is less
* Repeat for [2] and compare with [0],[1]
* Repeat until length of array>0

**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

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

**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 |

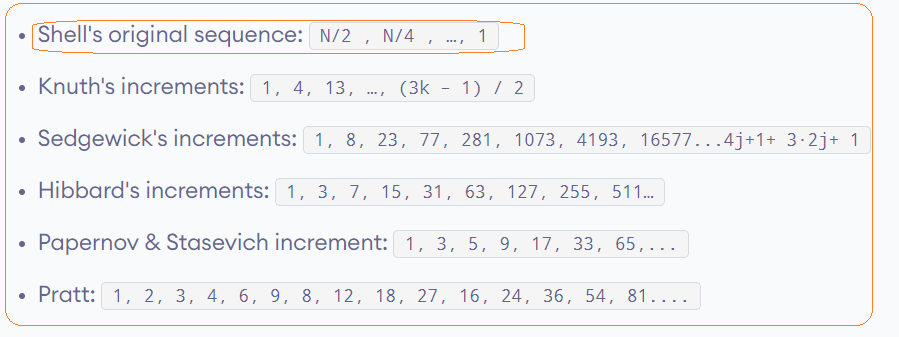
**pros**:

* 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

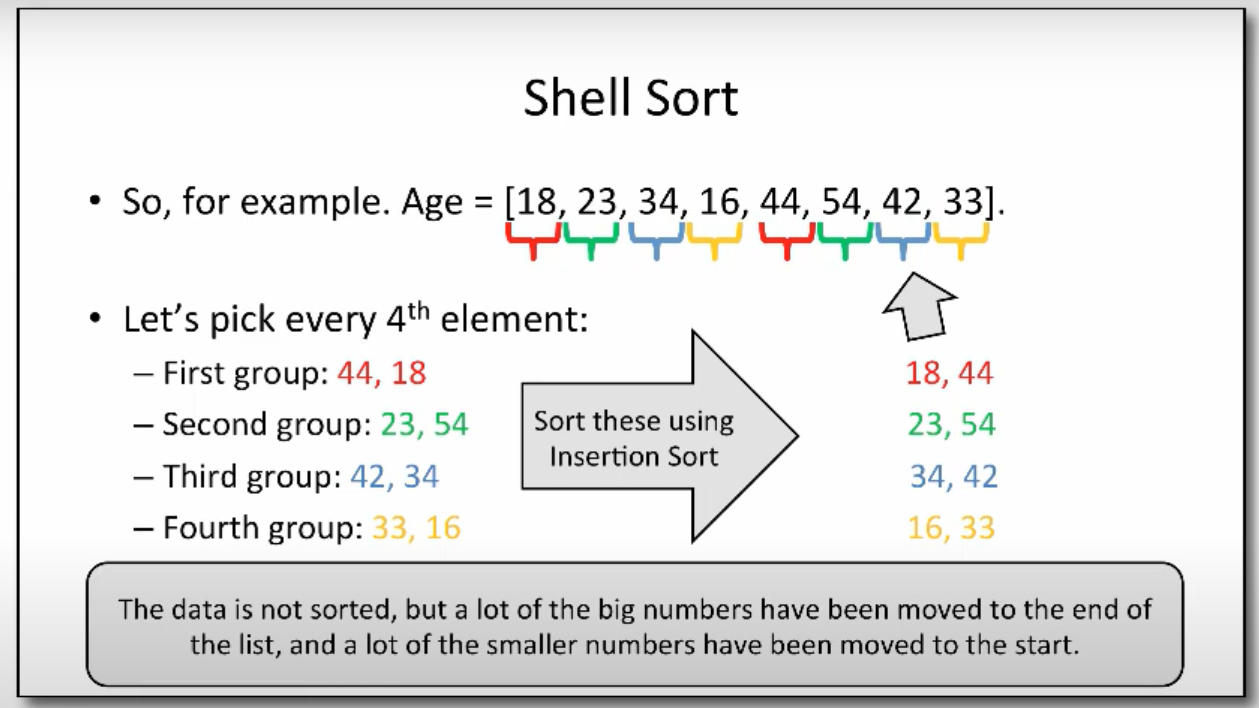
Gap can be chosen in any way. The original way is a divide n=length of array/2: n/2, (n/2)/2,…

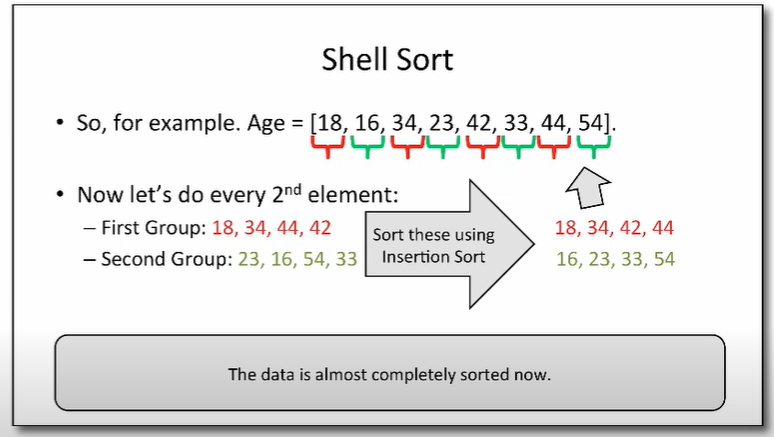


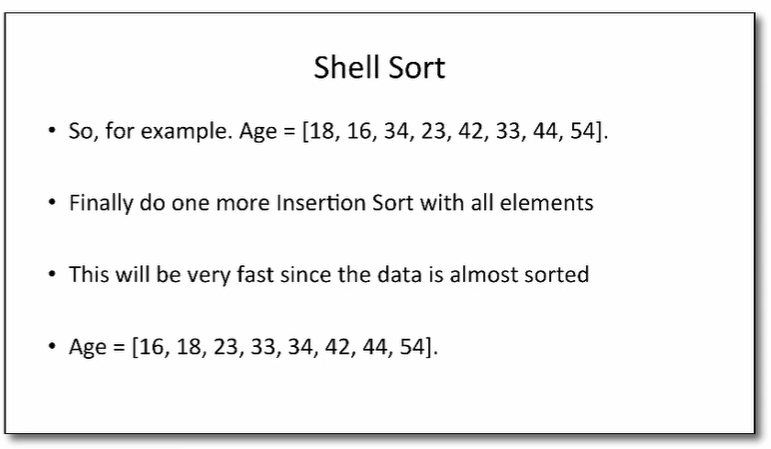
For example, len=11. N=11/2=5; N=5/2=2;













# **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
* **The main difference between quicksort and merge sort is that the quicksort sorts the elements by comparing each element with an element called a pivot while merge sort divides the array into two subarrays again and again until one element is left**
* Merge sort is used in external sorting

**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.

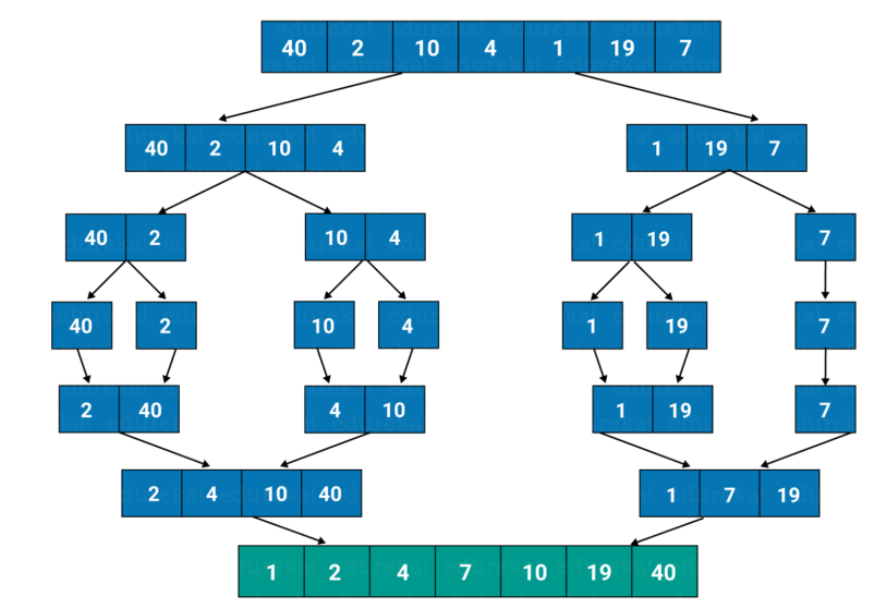
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **MERGE SORT** | Ω(nlog(n)) | Θ (nlog(n)) | O(nlog(n)) | O(n) | No | Yes | Yes |

**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.
* **Merge sort is used in external sorting**

**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. Idea is based on choosing a pivot element and shift elements according to this pivot element. It repeats recursively

**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
* **The main difference between quicksort and merge sort is that the quicksort sorts the elements by comparing each element with an element called a pivot while merge sort divides the array into two subarrays again and again until one element is left**.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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
* The worst-case running time for quicksort is O(n2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation
* Not stable

**Pivot** – is an arbitrary element.

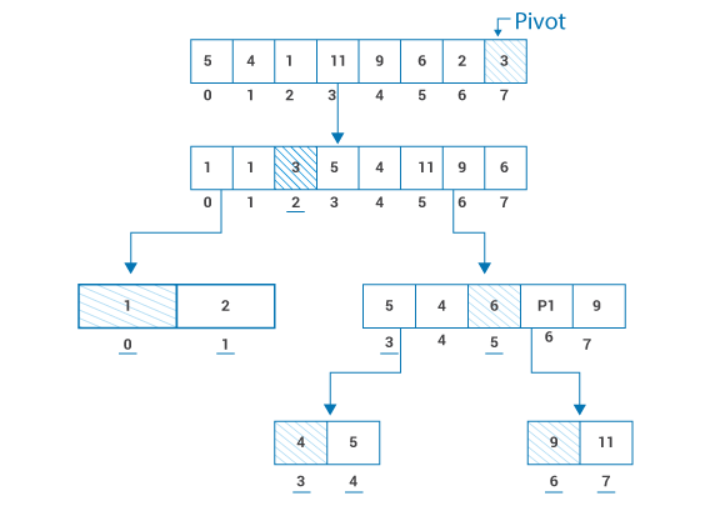
Pivot element can be chosen as

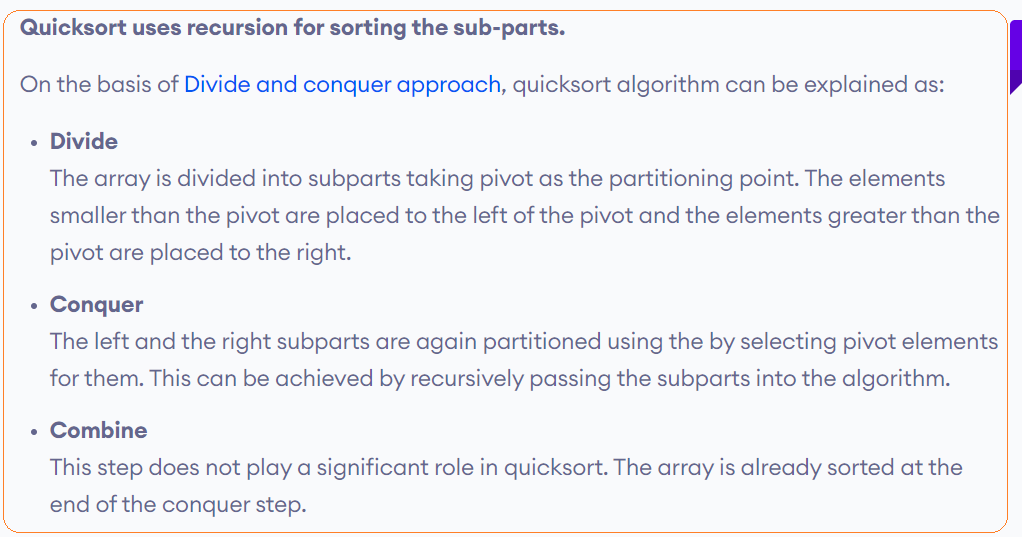
* First element
* Last element
* Medium
* Randomly

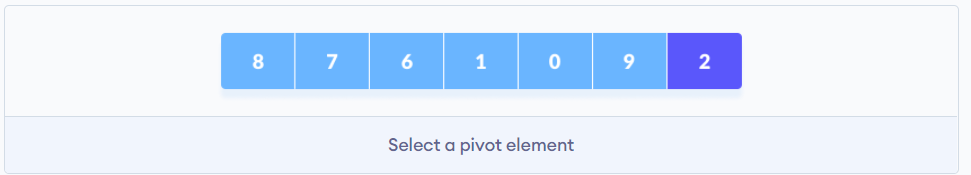
Note: poor chosen element provides the worst time complexity of algorithm as O(n)

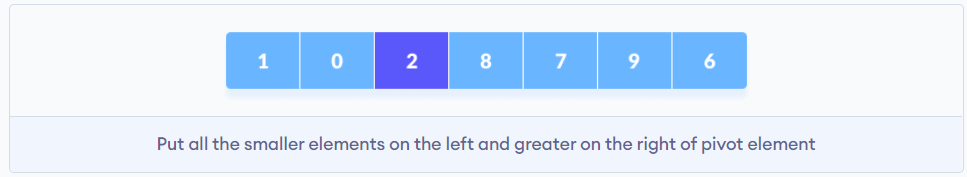
There are two worst cases:

* 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

Implementation:

* Find max value in array and creates an array of length=maxValue+1
* Calc cumulative values for new array
* Return output based on matching values from original array and additional for frequency

facts

* Not-comparison method

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

**pros**:

* it can be perform even better than QuickSort if number of element are small
* only good for small numbers
* is used for implementation of radix sort

**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

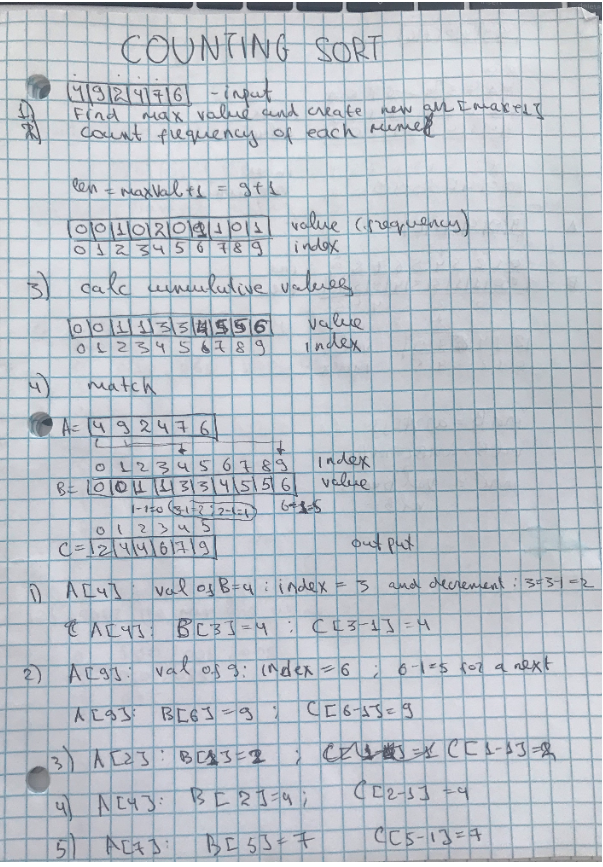
**Time complexity:**

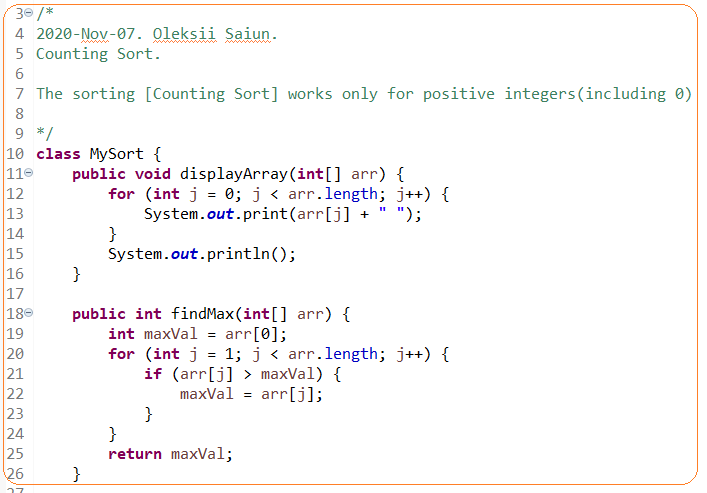
* Ω(n+k)
* Θ(n)
* O(n)

**Space complexity:**

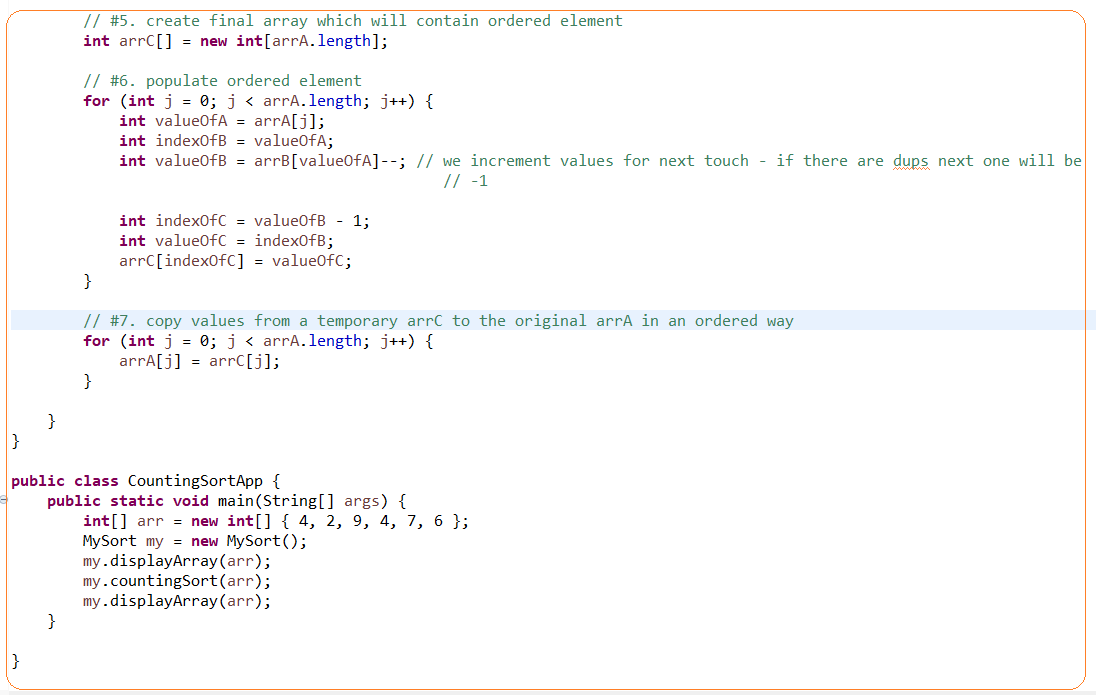
* O(n)

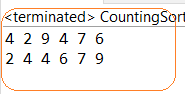
n-number of elements, k – the biggest value











# **RADIX SORT**

**Radix sort** – is a sorting algorithm in a linear time that is based on sorting digit by digit from right to left in buckets. It is useful to sort large integers.

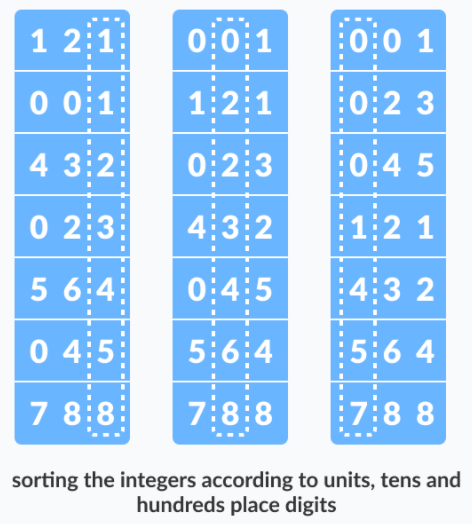
It is based on counting sort and radix is used when counting does not work – there is a big max value in an array

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**:

* We create a bucket (from 0 to 9 range) and sort digit by digit from right to left. For example, [9005,100,9000,2] – 4 max number of digits. We will have 4 pass.
* Each sort is based on counting sort
* When data is sorted in a bucket we return them in sorted order for that digit



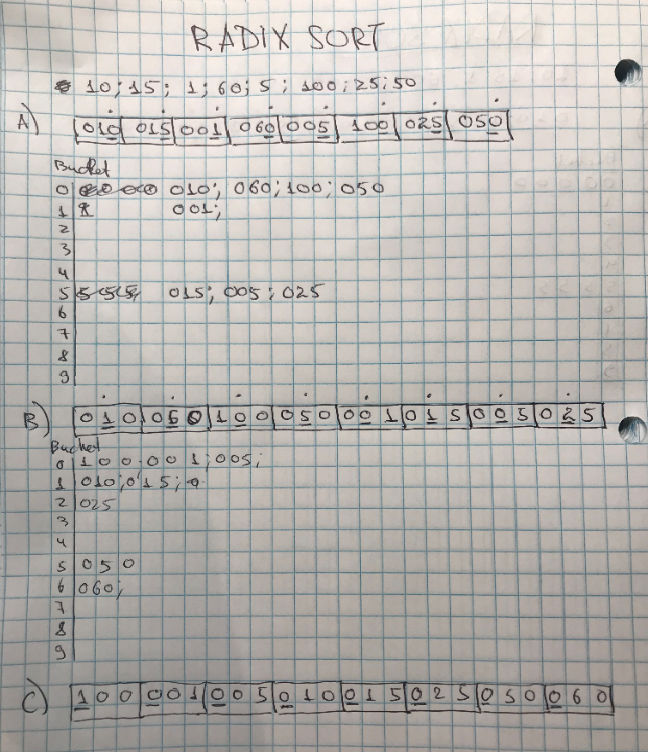
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **RADIX SORT** | Ω(n\*k) | Θ(n\*k) | O(n\*dk) |  | No | Yes | No |

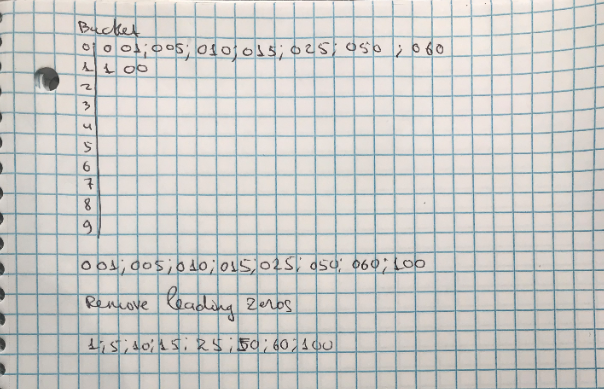
**Pros**:

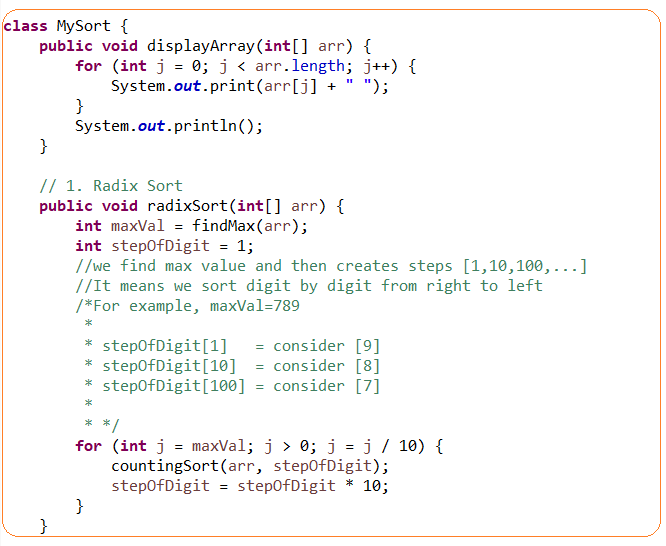
* It is useful to sort large integers
* Sorting can be achieved in linear time
* one of the fastest sorting algorithms, but I n particular cases, not general
* can be good for large numbers and when values are similar length

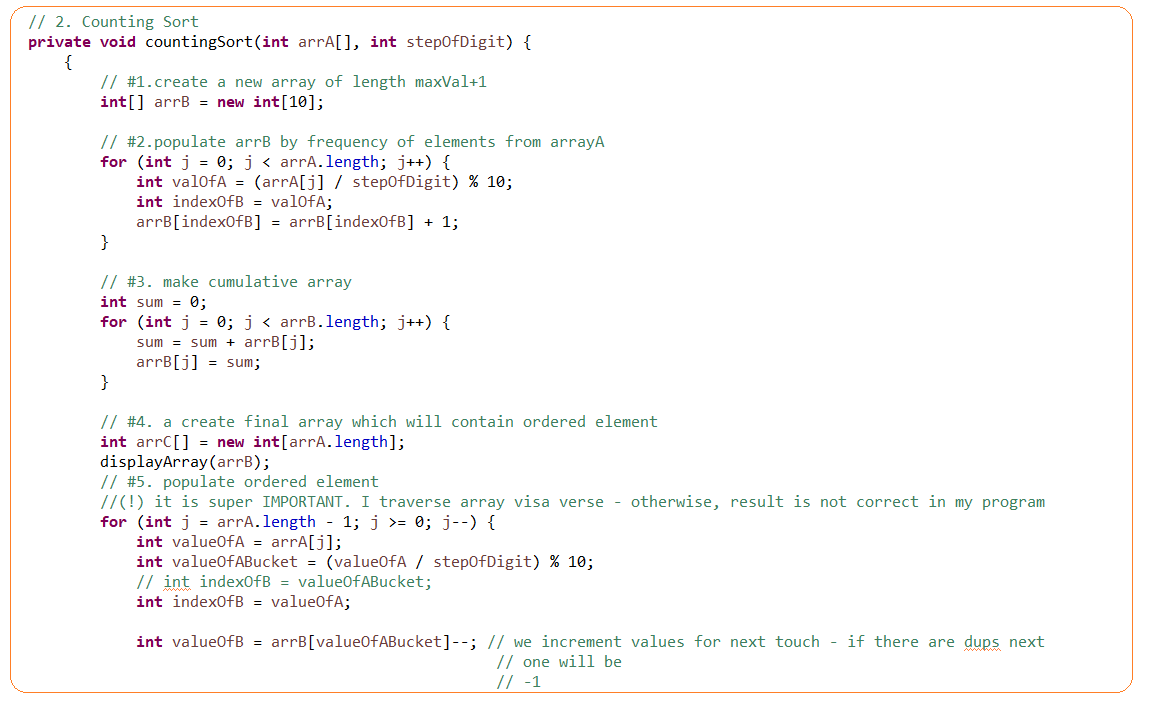
**Cons**:

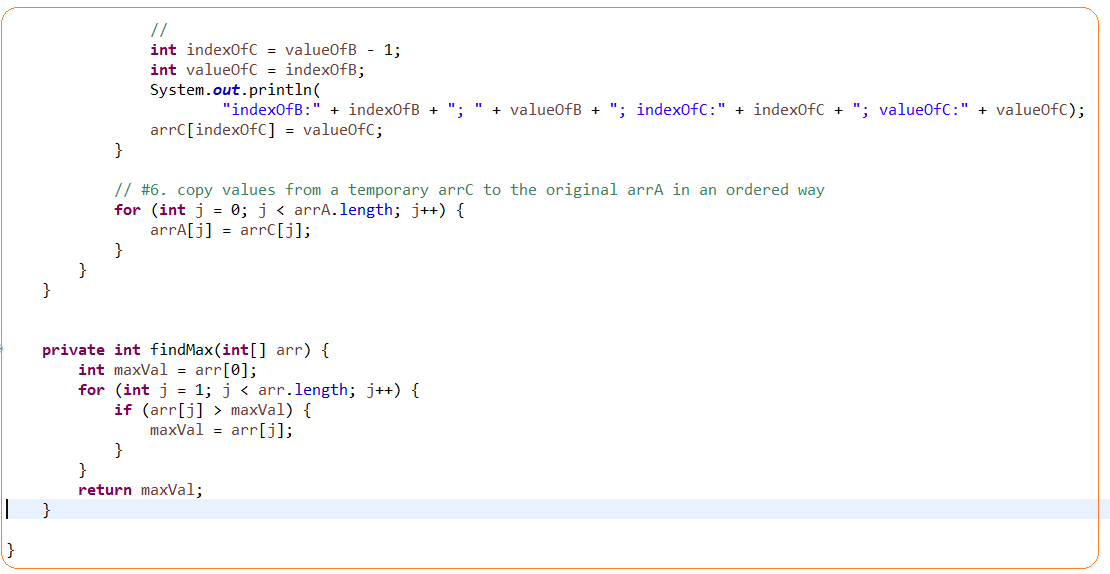
* Radix sort needs to be rewritten if the type of data is changed.
* If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting
* Only for integers and strings data type
* It is not use often especially in libraries, because there a lot of constraints

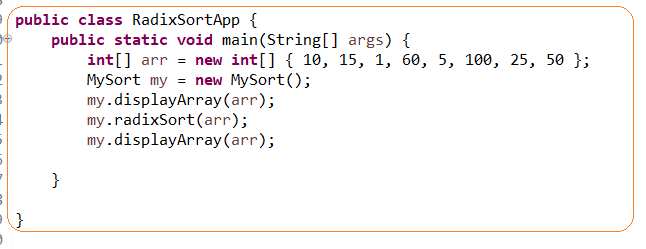


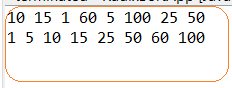












# **BUCKET SORT**

**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

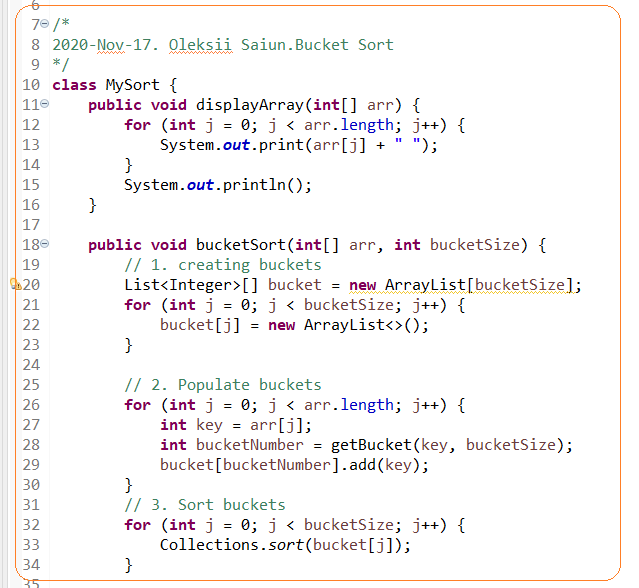
**Implementation**:

* Create buckets based on some value – like hashvalue or key/arrayLength
* Populate bucket and sort – values in buckets must not overlap; any sorting technique can be applied
* Merge buckets where there are already sorted values

**Pros**:

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

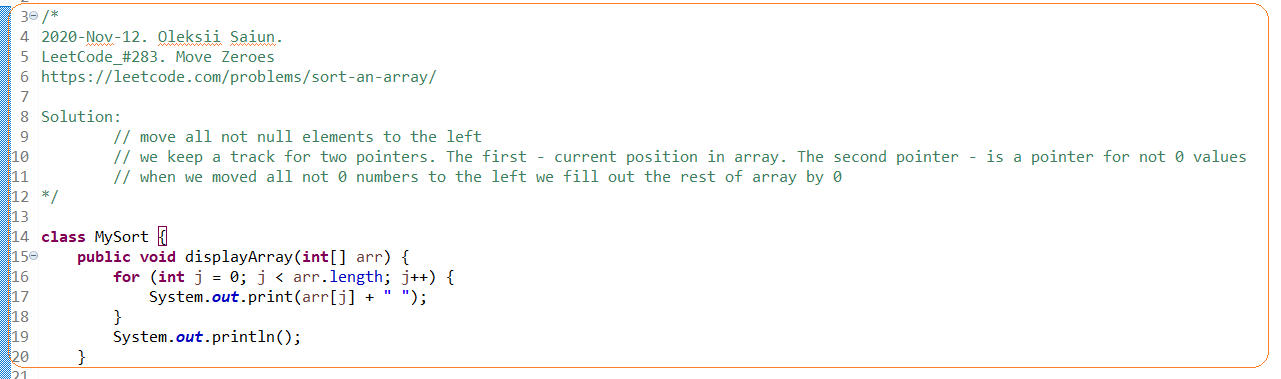
**Cons**: if you choose a bad distribution factor then you will get data skew in a bucket and lower performance

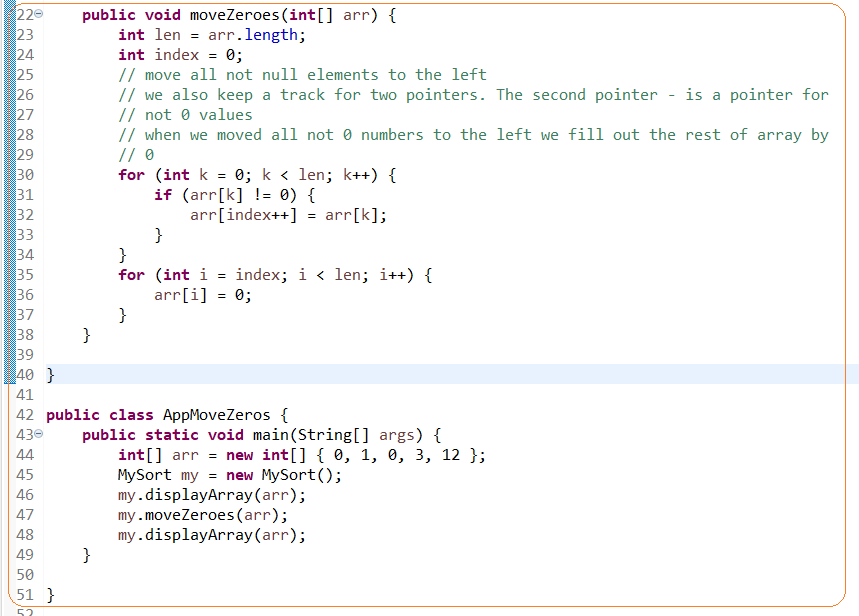






# **#1.TASK. MOVE ZEROS TO THE RIGHT END**







# **HEAP SORT**

Heap sort is based on a heap data structure.

The idea is:

* Create and populate a heap
* Eliminate element from a root: you swap elements from a root from last element and then heapify new elrment in a root. It repeats for all remaining elements in array(according to pointer)

**Fact**:

* It turns out that heap sort is a lot like selection sort in its logic: both algorithms find either the smallest or largest element, “select” it out, and put that item in its correct location in the sorted list.
* However, as similar as they are, heap sort is much better than selection sort in one massive way: its performance! Heap sort is basically a super-improved version of selection sort. Yes, it does find the largest element in an unsorted collection and orders it at the back of the list — however, it does all of this work so much faster than selection sort would!

Sorting:

* Max heap – is used for sorting array in ascending order (after removal elements)
* Min heap – is used for sorting array in descending order (after removal elements)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **BEST** | **AVERAGE** | **WORST** | **SPACE COMPLEXITY** | **IN-PLACE** | **STABLE** | **COMPARISON** |
| **BUCKET SORT** | Ω(n+k) | Θ(n+k) | O(n2) | O(n) | No | N/A | No |

**Pros**:

* Very useful if need to find top k elements
* An efficient way to build PriorityQueue (there are two options of PriorityQueue – max-priority queue and min-priority queue)
* Heap sort is particularly suitable for sorting a huge list of items.
* **Job Scheduling** - In Linux OS, heapsort is widely used for job scheduling of processes due to it's O(nlogn) time complexity and O(1) space complexity.
* **Graph Algorithms** - It can used in the implementation of priority queue for Djikstra's algorithm, Prim's algorithm and Huffmann encoding as well.

**Cons**:

* Is not stable
* Although Heap Sort has O(n log n) time complexity even for the worst case, it doesn't have more applications ( compared to other sorting algorithms like Quick Sort, Merge Sort )

