Pawel Zamorski

Student ID: G00364553

Galway, 19 May 2019

Computational Thinking with Algorithms

Project: Benchmarking Sorting Algorithms

Contents

Introduction

The verb “to sort” has two similar, however distinct meanings. In means to arrange a number of elements in a certain order or to divide them into groups with similar characteristics. Both of those meanings may be used when solving a sorting problem in computer science.

In computer science, the definition of sorting is more precise. “Sorting” chiefly in the strict sense of sorting into order (Knuth, 1968[1]). It refers to a process of organizing a collection of objects in an ordered sequence, in a way that matches some rules. The most-used rules of ordering are numerical and lexicographical order, but other rules may be defined. The sorted collection must satisfy two conditions:

- the elements must be in a non-decreasing order, meaning that every element is greater than or equal to the one before it.

  A collection N is sorted if for all i < j, N[i] =< N[j]

  An empty or singleton collection is always sorted. This may be used a base case in an algorithm. (Wikipedia, Total Order[2])

- the sorted collection is a permutation of an original collection, meaning that the elements are rearranged, but they are the same in both collections

Sorting is used by its own to solve a real-word problems. At first, a sorted collection is much more readable. It may be used to arrange words in a dictionary, a library of books or music, a data in a phone book, folders and files, a list of ordered items in a cart and many other. An ordered collection enables to find an easy and efficient solution for many other problems. When a list of arguments is sorted, it is easier to search an element, to find min and max element, to search a duplicate element, to get a subarray of n elements that fulfil a certain criteria, to merge efficiently two sorted collections. That is why it is very often used as a pre-process for other algorithms.

The invention of sorting techniques dates back before the development of computers. Herman Hollerith implemented a radix sorting method in his punched-card tabulating and sorting machine in the early of 20-th century. The card-walloping machine from 1936 used a merging technique (Knuth, 1968[3])

One of the first computer sorting algorithm, a merge sort, was invented by John von Neumann in 1945 and was used in EDVAC computer. This algorithm is one of the most important algorithm in computer science and is still widely used. There are many other sorting algorithms, like Radix Sort (1954, Seward), Counting Sort (1954, Seward), Quick Sort (1962, Hoare), Heapsort (1964, Williams), Smoothsort (1981, Dijkstra), Introsort (1997, Musser). Although, this field has been widely explored, researches are still carried out and new algorithms or new variations of previous one are developed (e.g. many versions of heapsort). A good example is Timsort, invented by Tim Peters in 2002. It is used in Java for sorting nonprimitive types (Oracle , Arrays[4]) and is the standard sorting algorithm for Python. In 2006 the library sort was published.

Sorting algorithms are divided into two main groups:

- **comparison based sorting** – an algorithm that compares elements with each other in a collection in order to sort them, e.g. Merge sort, Bubble sort, Selection sort, Heap sort, Quick sort. Comparison sorts cannot run faster than O(n log n).
- **non-comparison based sorting** – an algorithm that does not compare elements in a collection in order to sort them, e.g. Radix sort, Count sort, Bucket sort. Non-comparison based sorting algorithm are not bounded by O(n log n).

Each of the sorting algorithms has its own pros and cons. The decision of which one should be used depends on particular situation. The most common algorithm characteristics that should be considered are: time and space complexity, in-place sorting, stability, adaptability. The decision should also depend on assumption or knowledge about input data, especially:

- a size of collection – for a small amount of data it may be better to use a simple algorithm.
- an original sorted state of elements - if the elements are nearly sorted it may be better to use an adaptive algorithm
- keys if they are duplicate or not - if the key are duplicate there may be need for a stable sorting
- a range of values – if number of elements is small but the range between the smallest and greatest input is huge, non-comparison algorithms are inefficient.

Stability (stable sort): a sorting algorithm is stable if the equal elements (with the same key) stay in the in the same order in the sorted collection as they were in an original collection. Stability may be used as a further requirements for sorting algorithm.

Any unstable sorting algorithm may be implemented as stable. It is done by comparing positions of equal elements in an original collection. One way to do this is by temporarily appending an index to the key of each item. The other way is to replace each item in an collection by adding extra variable that stores an index and adding it to the key in a comparator function. However, it always requires extra time and space.

Bubble Sort, Insertion Sort, Merge Sort, Count Sort are stable by nature. Quick Sort, Heap Sort are not stable algorithm. Some sorts such as Radix Sort depend on another sort, with the only requirement that the other sort should be stable. (Geeksforgeeks, stability in sorting algorithms [5])

Adaptability (adaptive sort): the sorting algorithm is adaptive when it is guaranteed to run faster when some items in a original collection are already in order. The adaptive algorithm takes advantage of it and does not re-order already sorted elements.

Internal/external sorting: this term refers to the sorting situation. In the first case, records are stored in the computer's main memory, whereas in the second case the amount of records exceeds the capacity of main memory (Knuth, 1968[6]). Merge sort or quick sort algorithms are good for external sorting. (Wikipedia, External sorting[7])

Inversion: Inversion is a situation when a pair of elements in a collection are out of order. This term is defined among others for permutations:

For a given permutation π, the pair of elements (π_i, π_j) or the pair of places (i, j) is called an inversion if $i < j$ and $\pi(i) > \pi(j)$.

(Wikipedia, Inversion[8])

A collection that is already sorted has no inversions, whereas a collection that is in reverse order has maximum number of inversions, which for a collection of n distinct elements is $\frac{n(n-1)}{2}$

It is worth to mentions, that there number of permutations without repetitions on n elements is *n!*. Hence the probability that there is no inversions in a collection is very low.

The number of inversions is a measures of how much a collection is out of order. This information may be useful for some sorting algorithms, e.g. the running time of Insertion Sort depends on the number of inversions.

Comparator functions – implements logic that governs relative order in a sorted output. The meaning of "less" is defined in comparator function. It enables flexibility to decide what will be used as a key in a sorting process and defines the meaning of "less". It may be implemented for any data types. In Java there is an interface Comparator. It imposes total ordering. It uses a compare method, that takes two arguments, compares them and returns -1, 0 or 1 depending on the result of comparing (Oracle, Comparator[9]). The comparator function is used in sorting process, however it has no influence on this process.

Complexity measures: space and time complexity – a complexity measures are used for analysing a computational algorithm. They measure the amount of resources that is needed for running an algorithm. The **space complexity** describes a total memory space used by an algorithm to perform a task. The space complexity is O(1) if the algorithm does not require extra space to solve a problem. The **time complexity** measures the time needed to run an algorithm. It is described by big O notation. It uses as a metric a number of operations performed by an algorithm and is expressed as a function of the size of an input. The time complexity depends on the state of an input. That is why it is usually divided on the best-case, average-case and a worst-case.

Online/offline – an online algorithm accepts new data during running the procedure. An offline algorithm requires the whole data from the beginning. An example of online sorting algorithm is insertion sort and library sort (Wikipedia, online algorithm[10])

In-place/outplace – a sorting algorithm is considered to be in-place if it uses only a small amount of extra memory for sorting a collection. (Goodrich, 2014[11])

Sorting algorithms

1). Bubble Sort

Bubble sort is a comparison-based algorithm.

Procedure: Bubble sort algorithm compares each pair of adjacent elements starting from the first one in a collection (on the left). The elements in a pair are swapped if the preceding element is greater than the following element. The procedure is repeated on the next pair of elements until it reaches the end of the collection. In that way the greatest element is moved to the right side of the collection. The whole process is repeated iteratively until the collection is sorted. In the next iterations the comparisons are done to the point where element are already sorted.

Implementation: The presented implementation of Bubble sort algorithm uses two loop (outer and inner). The inner loop does not repeats the comparison process on already sorted elements. It uses also a flag that indicates whether the collection is already sorted during the process. If this condition is true the outer loop is broken and the process stops. The condition is true if there is no swap in an inner loop.

Space complexity: O(1)

Time complexity: best case – O(n); average case – O(n^2); worst case – O(n^2)

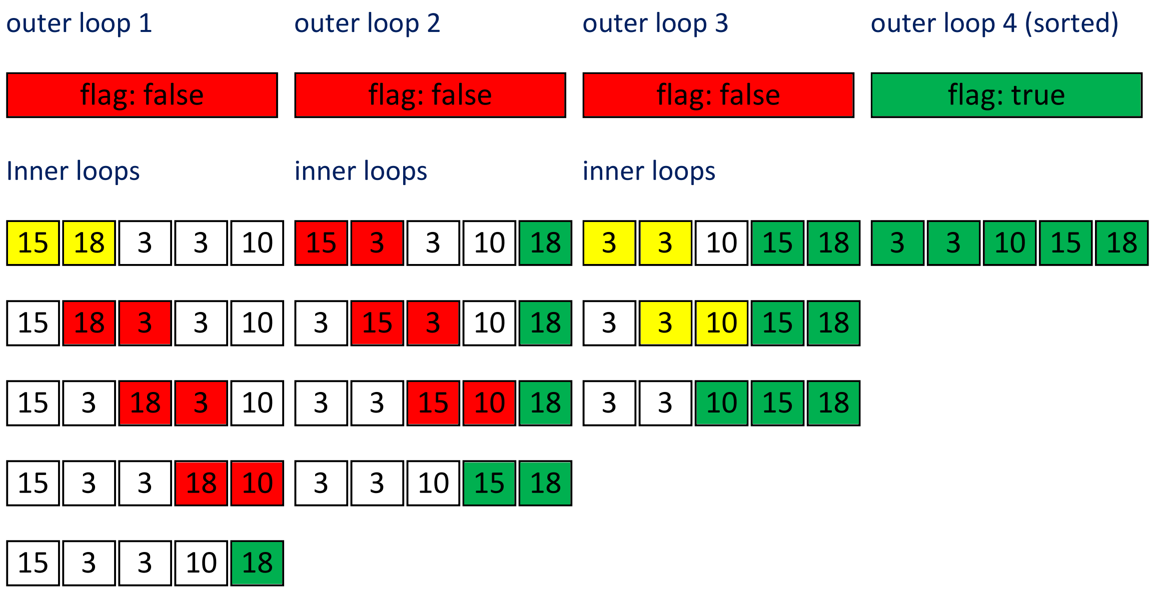
The best case occurs when the original collection was already sorted. It is needed to use a flag in an implementation of algorithm in order to achieve the best case.

Stable: yes

In-place sorting: yes

Internal/external: internal

Diagram:



Explanation to the diagram:

The diagram shows two loops: the outer and the inner. In the first iteration of outer loop the flag is set up to false. The inner loop starts by comparing first pair in the collection: 15 < 18. The condition for swapping is false. The next pair is out of order: 18 > 30. The elements are swapped. The process is repeated until the greatest element in the collection (18) is moved to the right. The inner loop made 4 comparisons for the collections of 5 elements.

Next starts the second iteration of outer loop. The flag is set up to false. The inner loop repeats the process of comparisons. This time 15 is moved to the right of the collection as the next greatest element in the collection. It did not make a comparison with already sorted element. The inner loop made 3 comparisons.

The third iteration of outer loop starts. The flag is set up to false. The inner loop repeats the process of comparisons. It compares equal elements 3 = 3 and does not swap them. The algorithm is stable. There was no swapping done by inner loop in this iteration of outer loop. The flag is set up to true. The forth iteration of outer loop is broken.

Conclusion: Bubble sort is a simple sorting algorithm. It is easy to implement. The average and worst case performance is $O(n^2)$. That is way it is not suitable for sorting a large, unordered data collections. It may be used to sort a small number of elements. It also may be efficient when a collection is already almost sorted, e.g. it takes 2n times when only one element is out of order.

2). Heapsort

Heapsort algorithm was proposed by John Williams in 1964. It is a comparison based sorting algorithm. It uses a **binary heap** data structure to sort elements in a collection. The binary heap was also invented by the same author.

A binary heap is a type of a **binary tree**. The binary tree is a structure consisted of nodes with only one root node and maximum two child nodes per parent node. The binary heap must have two additional properties: a structural property and a relational property.

The structural property means, that the heap tree must have a proper shape. It must be **a complete binary tree**. It means that all levels of the tree, except the lowest one, must be completely filled (every parent except the parents from the penultimate level must have two child). In addition, the nodes in the last level are filled from left to right. (Goodrich, 2014[12])

The relational property relates to the order of elements in a binary heap. It may be:

- non-decreasing order (min-heap) - the element (the key) in a parent node must be less than or equal to the element (the key) in child element. In min-heap the minimal element (the key) is stored at the root.
- non-increasing order (max-heap) – the element (the key) in a parent node must be greater than or equal to the element (the key) in child element. In max-heap the maximal element (the key) is stored at the root. (Wikipedia, Binary heap[13])

It is worth to note, that heap tree allows duplicate elements and the element in a left node does not have to be smaller than the one in the right node. (Medium, learning to love heaps[14])

A complete binary tree may be easily implemented with an array by using the following properties:

- the root node is at index 0
- if the parent element is in index i, the index of left child node is equal to 2*i + 1
- if the parent element is in index i, the index of right child node is equal to 2*i + 2

By using the binary heap it is easy to find the max or min element. After removing this element the heap is not in the correct shape. The root element is missed. To correct it, the last element from the tree is moved to the root. Next the process called **down-heap bubbling** moves this element down of the tree until the tree is in the correct order.

Procedure: Heapsort algorithm uses the above properties and methods. At first step it swaps elements in a collection to build a binary heap structure. Next it swaps the element in index 0, which has the max key with the element in the last index. Then the size of the array is shrink to n-1 element and the process of ordering the elements in a heap is repeated. The down-heap bubbling may be used. Again, the next largest/smallest element is at the root node at index 0. The process is repeated iteratively until all elements are sorted.

There are two ways to build a binary heap structure:

- treat an initial heap as empty and successively insert elements to the heap. It is done by insert operation that takes O(n log n) time in the worst case.

- bottom-up approach, in which a binary heap is construct at first, next elements are inserted to the lowest level of the heap, than the elements are iteratively inserted to the higher levels of the heap altogether with a swap operations if needed. This process takes O(n) time.

There are many implementations of Heapsort. The last one is Katajainen's "ultimate heapsort".

Implementation: The presented implementation of Heapsort algorithm uses bottom-up approach to build a heap. Next it uses down-heap bubbling to rebuild a heap.

Space complexity: O(1)

Time complexity: best case – O(n log n); average case – O(n log n); worst case – O(n log n)

Creating the heap takes O(n log n) or O(n). The down-heap bubbling is O(log n) and is called n times. Swapping the max element to the end of array is O(1). The performance of this algorithm is O(n log n) in best, average and worst case.

Stable: no

In-place sorting: yes

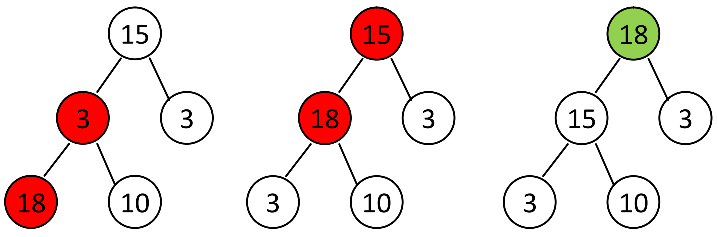
Internal/external: internal

Diagram:

Original array:

15	3	3	18	10

Step 1 – create a max-heap

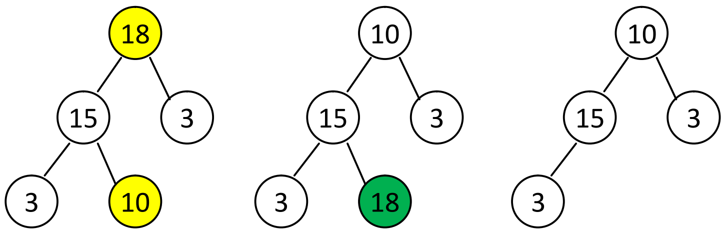


Step 2_1 – move max element (index 0) to the end of array

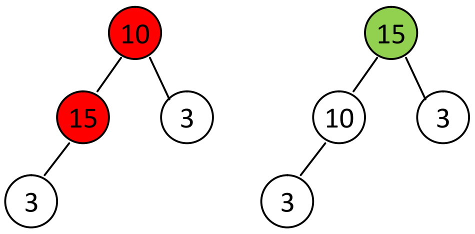
18	15	3	3	10

The last element is sorted. The next binary heap will be shorter.

10	15	3	3	18



Step 3_1 – create a max-heap by moving the element at a root node to its correct position

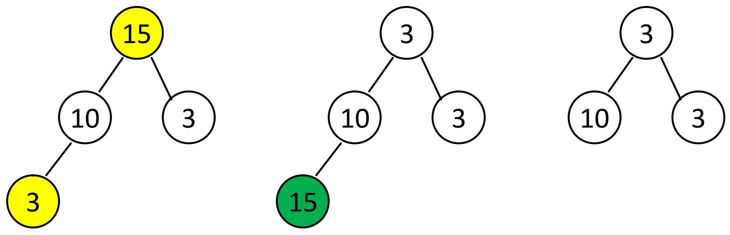


Repeat the process until the collection is sorted:

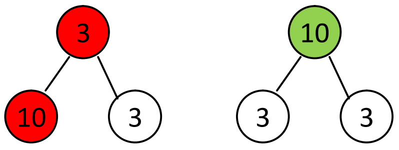
Step 2_2

15	10	3	3	18

3	10	3	15	18



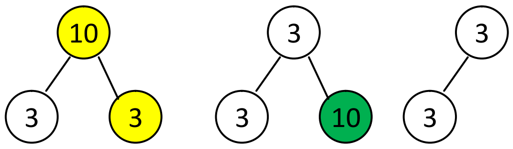
Step 3_2



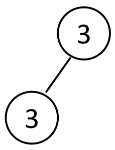
Step 2_3

10	3	3	15	18

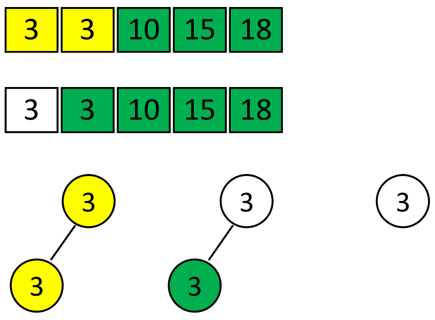
3	3	10	15	18



Step 3_3

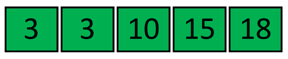


Step 2_4



Step 3_4

Step 2_5



Conclusion: Heapsort uses an efficient way to find the largest element. It does not compares all elements with each other, which take O(n) time. Instead it uses a tree structure. It takes O(log n) to find next maximum element. It is a good algorithm to sort large amount of data. However, the basic implementation is unstable.

3) Counting Sort

Counting Sort algorithm was invented by Harold Seward in 1954.

Counting sort algorithm is an integer algorithm. It means that the keys must be a type of integer. It is an efficient algorithm. It is a non-comparison based sorting algorithm. The non-comparison sorting algorithms are not bound to the O(n log n) running time. However, Counting sort is efficient only if the range of the key values is not significantly greater than the number of elements in a collection. The above are limitations of Counting sort.

Procedure: Counting sort algorithm uses three arrays: an input, an output and an count array. The count array is the size of k, which is the range of key values in the input. The initial values at all indices in the count array is zero (it is the integer array).

The k value may be given explicitly. If it is not known, it must be computed at a first step by looping over the input data to set the range of the key values (min and max value). The count array starts from index 0. It must be correctly initialize. It may be initialize with the size k = max + 1. However it is only efficient if the min value is zero. In other cases, in order to keep it efficient, the count array should be initialize with the size k = max – min + 1. Next the value from the input array must be corrected when operating on the count array. The value from the input array is subtracted by min value. Such implementation will also sort arrays with negative integers.

The algorithm loops over the collection and counts how many elements of each key value there are in the collection. The indices in the count array corresponds to the key values in the input array. For example, if there is no elements of key 3, the value at index 3 of count array will be 0. If there is one element of key 4, the value at index 4 of count array will be 1. If there are two element of key 5, the value at index 5 of count array will be 2, and so on. It will create a histogram of how many times each key appears in the input collection.

Then the count array is modified by using prefix sum computation. It will enable to determine at what index the elements should be placed in the output array. (It determines the range of indices at which each the element should be placed.) This step runs in a loop. The modification is done as follows: the value at each index is replaced by the sum of the value at modified index and the value at the previous index. For the count array C,   C[i] = C[i] + C[i – 1]. The element in index 0 is not modified.

In the next step the output array is filled up with the elements from the input array in the correct order. The algorithm loops over the input array. In order to keep the sorting stable, it should start from the last element. The correct index for each element is checked in the count array and it is equal to the value at the appropriate index of count array – 1. The value in the count array is decremented every time the corresponding element is placed in the output array. (Brilliant, counting sort[15])

The Counting sort is stable, but must be implemented correctly.

Implementation: The presented implementation of Counting sort algorithm uses the procedure described above.

It determines the min and max value of an input array. It initializes a count array with the size k = max – min + 1. It subtracts a value from the input array when operating on the count array: `count[input[i] - min]`. Such implementation will also sort arrays with negative integers.

The algorithm is stable. It starts iteration from the last element when the output array is populated.

Space complexity: O(n + k)

Time complexity: best case – O(n + k); average case – O(n + k); worst case – O(n + k)

The performance of this algorithm is O(n +k) in the best, average and worst case.

Stable: yes

In-place sorting: no

Internal/external: internal

Diagram:

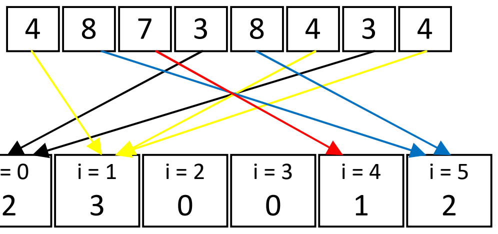
An unsorted collection: {4, 8, 7, 3, 8, 4, 3, 4};

4	8	7	3	8	4	3	4

Step 1: Determine the range of the values in the input array: min = 3, max = 8. The range is 8 – 3 = 5. The size of the count array is 5 + 1 = 6. It is initialize with all values equal to zero.

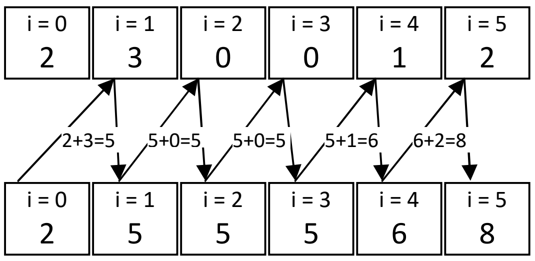
i = 0	i = 1	i = 2	i = 3	i = 4	i = 5
0	0	0	0	0	0

Step 2: Loop over the input array and create a histogram using the count array. Subtract the value from input array by min when dealing with the count array.



The loop starts from the first element in the input array. It is 4. The value after subtraction by min = 3 is 1. The value at index 1 in the count array is incremented by 1. The process is repeated. The final result is shown above.

Step 3: Loop over the count array and modify its values by using prefix sum computation: C[i] = C[i] + C[i – 1]. The value at index 0 stays the same.

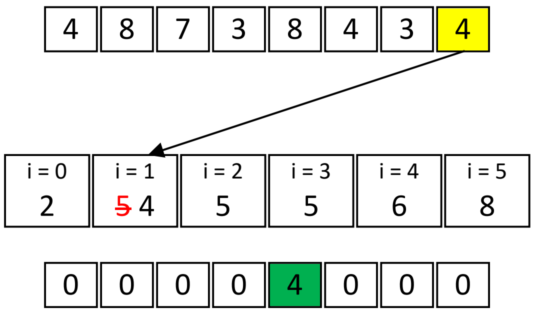


The loop starts from the element in index 1 of the count array. Its value is 3. The value in the previous element is 2. The new value in the index 1 is 3 + 2 = 5. The process is repeated. The final result is shown above.

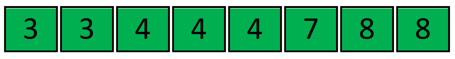
Step 4: Loop over the input array and populate the output array with the elements in the sorted order. The loop starts from the last element of the input array in order to keep the sorting stable. The output array is initialize with the same size as the input array and with all values equal to zero. Subtract the value from input array by min when dealing with the count array.

The below diagram shows the result after the first iteration.

The first value is 4. After subtraction by min = 3 it is 1. This result is used to refer to the index in the count array. The value at the index 1 of the count array is 5. At first this value is subtracted by 1. It gives 4. This result is used to refer to the index in the output array, which is the correct position for the value 4. Finally the value 4 is inserted to the output array at index 4. Next the value at index 1 of the count array is decremented by 1.



The above process is repeated recursively, until the output array is filled with the elements, that are in ordered position.



Step 5: The values from the output array are copied to the input array. It is done iteratively. The values in the input array are sorted.

Conclusion: Counting sort is an efficient non-comparable sorting algorithm. It may run faster than comparable based sorting algorithm. However, it has few limitations. It is an integers sorting algorithm. It is possible to implement this algorithm to floating point numbers and string. Secondly, the running time depends on the range between the maximum and the minimum key value. It is efficient only if this difference is not much greater than the number of item in a collection. For that reason it may be used as a subroutine in a radix sort, that is more efficient for a larger key (Wikipedia, Counting sort[16])

4). Merge Sort

Merge sort algorithm was invented by John von Neumann in 1945. It is an efficient comparison based sorting algorithm.

It is also a divide and conquer algorithm. It may be implemented as a stable sorting algorithm. Usually it is not implemented as an in-place algorithm, however such implementations exist.

Procedure: The merge sort uses the idea that it is much easier to sort a collection containing a small amount of elements than the huge one. Once the small collection is sorted the merging process of two sorted collection is done efficiently.

This approach to solving a problem is called divide and conquer. A divide and conquer algorithm works in three steps:

- divide: divide the input data into two or more subsets. If the input data are small enough, solve the problem and return the solution.
- conquer: use the recursion on subset to solve the problem.
- combine: merge the solution from the subsets to solve the original problem.

The procedure of Merge sort consist of the above three step:

- divide: the base case is when a collection has just one element, as a collection containing one element is sorted. In that situation the collection is returned. If there is more than one element in a collection, it is divided into two sub-collections of about n/2 elements each.
- conquer: recursively sort both sub-collections.
- combine: merge both sorted sub-collections by inserting the elements from sub-collection into the original collection in a sorted order. This step requires comparison procedure (Goodrich, 2014[17])

Implementation: The presented implementation of Merge sort contains all steps described above.

The algorithm is stable, which means that the order of equal elements is the same in the input and output.

Space complexity: O(n)

Time complexity: best case – O(n log n); average case – O(n log n); worst case – O(n log n)

It is O(n log n) time complexity in the best, average and worst case, as it is run recursively and the base case is a collection containing only one element.

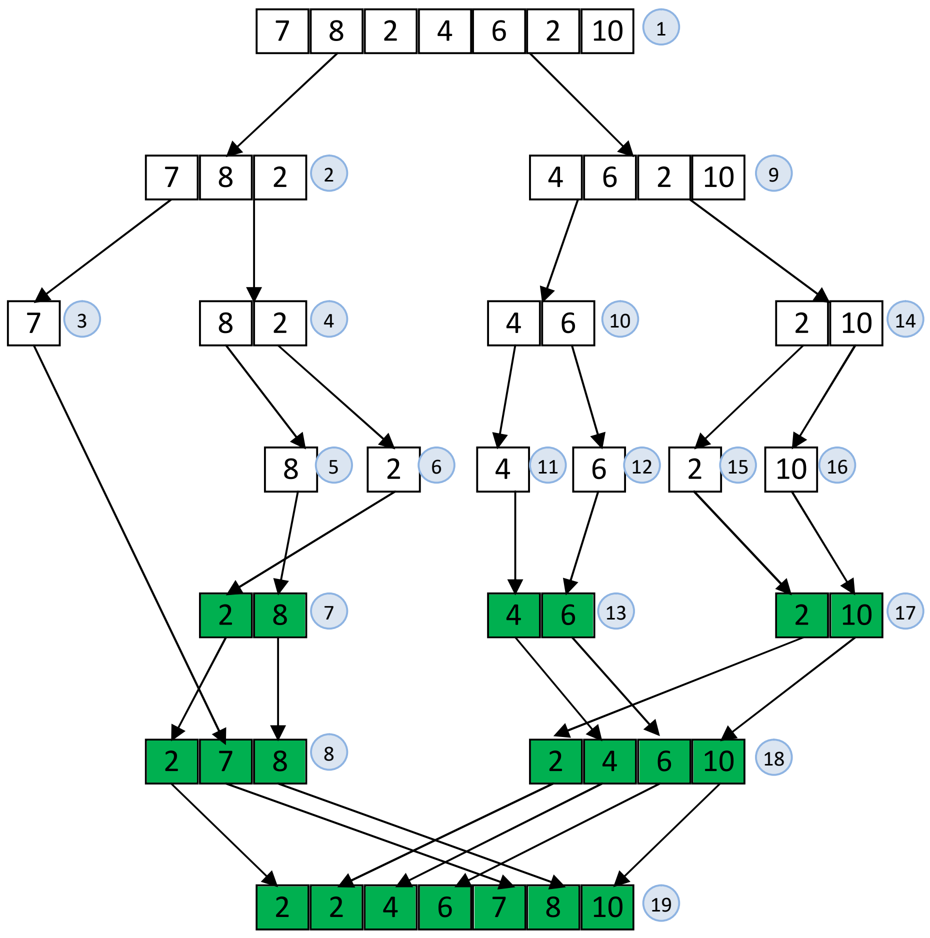
Stable: yes

In-place sorting: no, however it may be implemented as in-place sorting

Internal/external: external

Diagram:

An unsorted collection: {7, 8, 2, 4, 6, 2, 10};



The description to the diagram (The following steps are marked on the diagram with a number in a blue circle):

```
Step no 1: Divide input array: [7, 8, 2, 4, 6, 2, 10]
Left array: [7, 8, 2] Right array: [4, 6, 2, 10]

Step no 2: Divide input array: [7, 8, 2]
Left array: [7] Right array: [8, 2]

Step no 3: The input array: [7] The sort method returns. The array is sorted for
an array of size 1

Step no 4: Divide input array: [8, 2]
Left array: [8] Right array: [2]

Step no 5: The input array: [8] The sort method returns. The array is sorted for
an array of size 1

Step no 6: The input array: [2] The sort method returns. The array is sorted for
an array of size 1

Step no 7: Merging; original input is: [8, 2]
Left array: [8] Right array: [2]
Input after merging: [2, 8]
```

```
Step no 8: Merging; original input is: [7, 8, 2]
Left array: [7] Right array: [2, 8]
Input after merging: [2, 7, 8]

Step no 9: Divide input array: [4, 6, 2, 10]
Left array: [4, 6] Right array: [2, 10]

Step no 10: Divide input array: [4, 6]
Left array: [4] Right array: [6]

Step no 11: The input array: [4] The sort method returns. The array is sorted for
an array of size 1

Step no 12: The input array: [6] The sort method returns. The array is sorted for
an array of size 1

Step no 13: Merging; original input is: [4, 6]
Left array: [4] Right array: [6]
Input after merging: [4, 6]

Step no 14: Divide input array: [2, 10]
Left array: [2] Right array: [10]

Step no 15: The input array: [2] The sort method returns. The array is sorted for
an array of size 1

Step no 16: The input array: [10] The sort method returns. The array is sorted for
an array of size 1

Step no 17: Merging; original input is: [2, 10]
Left array: [2] Right array: [10]
Input after merging: [2, 10]

Step no 18: Merging; original input is: [4, 6, 2, 10]
Left array: [4, 6] Right array: [2, 10]
Input after merging: [2, 4, 6, 10]

Step no 19: Merging; original input is: [7, 8, 2, 4, 6, 2, 10]
Left array: [2, 7, 8] Right array: [2, 4, 6, 10]
Input after merging: [2, 2, 4, 6, 7, 8, 10]
```

The sorted array is: `[2, 2, 4, 6, 7, 8, 10]`

Conclusion: The merge sort uses the idea that it is much easier to sort a collection containing a small amount of elements than the huge one. It is efficient algorithm for a huge collection. However, it does not perform good for a very small input data, because of the first step of dividing.

5) Quick Sort

Quicksort algorithm was invented by Tony Hoare in 1959. It is a comparison based sorting algorithm.

It is also a divide and conquer algorithm, which works recursively and breaks down a problem into a smaller one of the same or similar type.

Procedure: Quicksort uses an element that is called pivot in a sorting process. The pivot is chosen from a collection. The elements that are less than the pivot are moved to the left side of the pivot. The elements that are greater than the pivot are moved to the right side of the pivot. The equal elements may be moved either to the left or to the right side. In that way the collection is partitioned into two smaller unsorted parts. The pivot by itself is already in the right position. The elements in both partitions are also in the correct order in relation to the pivot. It means, that there is no need to compare the left partition elements with the right partition elements (Medium, pivoting to understand quicksort[18])

Next the process is called recursively on both partitions until there is one element in a partition. By definition, if there is only one element in a collection it is considered sorted.

Next the method returns.

The pivot may be chosen in many different ways. It may be always the first element, the last element or the random element in the collection and the sub-collections. The best choice would be an element that gives two equal partitions (a median element).

Quicksort may be implemented as an in-place algorithm. It may be done in the following way. The pivot is keep in the last index. Algorithm uses two pointers: the left-side pointer and the right-side pointer. The left-side pointer starts from the first index, moves up and compares the elements with the pivot. If the element is less than the pivot it leaves the element in place. If it is greater than the pivot, the right-side pointer goes to the action. The right side pointer starts from the penultimate index, moves down and compares the element with the pivot. If it find the smaller element, it swaps that element with the element from left-side pointer. Next the left-side pointer goes to action and the process is repeated. (Goodrich, 2014[19])

Implementation: The presented implementation of Quick sort uses the methods describes above. It is in-place algorithm.

Space complexity: O(n) but it is possible to implement O(1) by using swapping method

Time complexity: best case – O(n log n); average case – O(n log n); worst case – $O(n^2)$

The worst case occurs when the chosen pivot is constantly greater or less than the rest of the elements. In that case no swapping is done which means the collection/sub-collections are not divided into partitions, but the comparison are made. It may happen when the collection is already sorted, nearly sorted or all the elements are equal.

Stable: no, stable variants exist

In-place sorting: no, however it may be implemented as in-place sorting

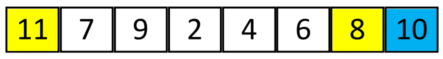
Internal/external: internal

Diagram:

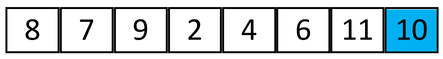
An unsorted collection: {11, 7, 9, 2, 4, 6, 8, 10}

Pivot =10, left pointer: l = 0, right pointer: r = 6.
Scan the elements and swap if necessary (in-place operation). The scanning process uses three loops: one outer and two inner. The outer loop runs until pointers cross each other (the left one is greater than the right one). The inner loops are to check if the elements are on the right side of the pivot. If the left pointer finds the element, that is greater than pivot, the left pointer loop is finished. Otherwise the left pointer is incremented by one. The process for the right pointer is opposite. The elements are swapped if they are on the wrong side and if the left pointer is not greater than the right pointer. After the swapping process the left pointer is incremented and the right pointer is decremented, as the swapped elements are in the correct positions. After the loops finishes the pivot is swapped with the element pointed by the left pointer.

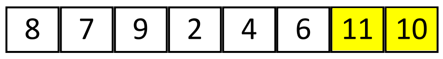
Left pointer stopped on index 0 (the element is greater than the pivot). The right pointer stopped on index 6 (the element is less than the pivot). The elements are swapped, left pointer is incremented (l = 1), right pointer is decremented (r = 5)



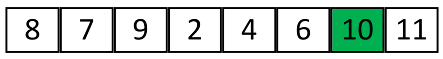
The elements are scanned. The element in index 1 is less than pivot and the left pointer is incremented. The process is continued by using left pointer in a loop, which is incremented. The element at index 5 is less than pivot. The left pointer is incremented (l = 6). Left pointer is greater than the right pointer. The loop is finishes.



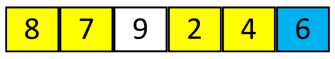
The pivot is swapped with the element at index 6, that is pointed by left pointer (l = 6).



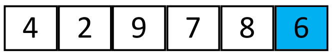
The pivot is in the correct position. There are two partitions on the left and on the right of the pivot.



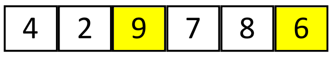
The process is repeated recursively. It starts from the left partition.
Pivot =6, left pointer: l = 0, right pointer: r = 4.
There are two swaps: the element at index 0 is swapped with the element at index 4 and the element in index 1 is swapped with the element in index 3. After these processes the left pointer l = 2, the right pointer r = 2.



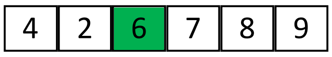
The left pointer checks the element at index 2. It is greater than the pivot. The left pointer stops and is not incremented. The right pointer checks the element at index 2. It is greater than the pivot. The right pointer is decremented r= 1. At that point left pointer is greater than right pointer. The loops stop running.



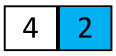
The pivot is swapped with the element at index 2, that is pointed by left pointer (l = 2).



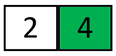
The pivot is in the correct position. There are two partitions on the left and on the right of the pivot.



Pivot =2, left pointer: l = 0, right pointer: r = 0.
The left pointer checks the element at index 0. It is greater than the pivot. The left pointer stops and is not incremented. The right pointer checks the element at index 0. It is greater than the pivot. The right pointer is decremented r = -1. At that point left pointer is greater than right pointer. The loops stop running.

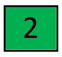


The pivot is swapped with the element at index 0, that is pointed by left pointer (l = 0).

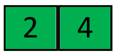


The pivot is in the correct position. There is one partition on the left of the pivot.

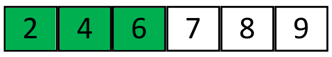
Pivot =2, left pointer: l = 0, right pointer: r = -1.
The array contains only one element. It is sorted by definition. The condition is checked. The left pointer is greater than right pointer. The method returns. The element is at index 0.



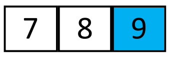
There is no right partition. The method returns.



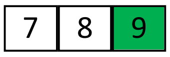
There is partition on the right side of the pivot = 6. The sort method is called recursively.



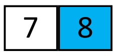
Pivot = 9, left pointer = 3, right pointer = 4.
The elements are scanned using the left pointer. Both elements are less than the pivot. The left pointer is incremented to l = 5. Left pointer is greater than the right pointer. The loop is finishes.



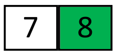
The pivot is swapped with the element at index 5 that is pointed by left pointer (l = 5), which is pivot by itself. The pivot is in the correct position. There is one partition on the left of the pivot.



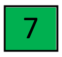
Pivot =8, left pointer: l = 3, right pointer: r = 3.
The left pointer checks the element at index 3. It is less than the pivot. The left pointer is incremented to l = 4. Left pointer is greater than the right pointer. The loops stop running.



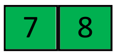
The pivot is swapped with the element at index 4 that is pointed by left pointer (l = 4), which is pivot by itself. The pivot is in the correct position. There is one partition on the left of the pivot.



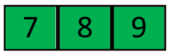
Pivot =7, left pointer: l = 3, right pointer: r = 2.
The array contains only one element. It is sorted by definition. The condition is checked. The left pointer is greater than right pointer. The method returns. The element is at index 3.



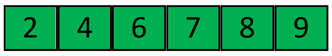
There is no right partition. The method returns.



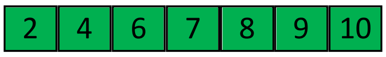
There is no right partition. The method returns.



There is no right partition. The method returns.

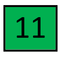


There is no right partition. The method returns.

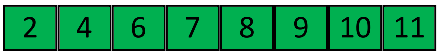


There is one partition on the left of the pivot.

Pivot =11, left pointer: l = 7, right pointer: r = 6.
The array contains only one element. It is sorted by definition. The condition is checked. The left pointer is greater than right pointer. The method returns. The element is at index 7.



There is no partition. The method returns.



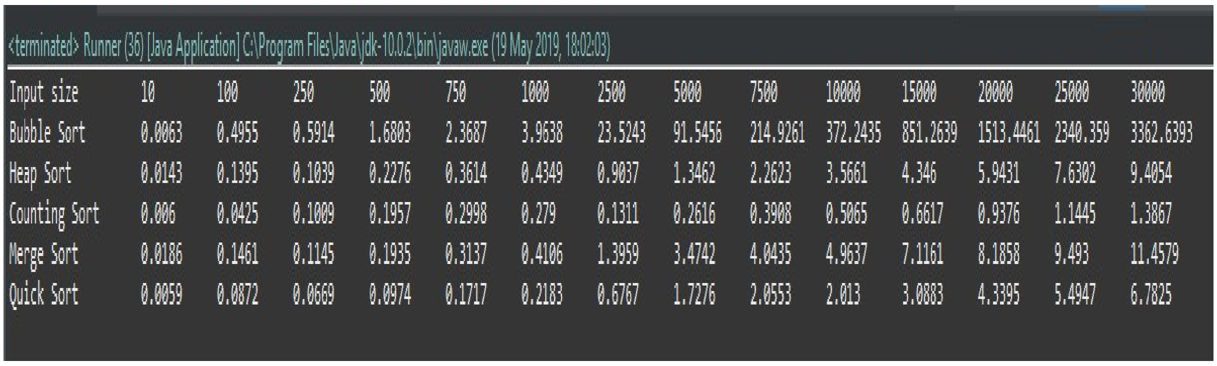
All elements are sorted. Te method returns.

Conclusion: Quicksort algorithm is not a good choice for a sorted or nearly sorted collection. In that case it will run in O(n^2) time. It will not run quickly if the pivot is far from the median. Quicksort is also unstable.

Implementation & Benchmarking

Each sorting algorithm was run 10 times for every input. The average running times are presented below. All values are in milliseconds.

Figure 1 Snapshot from the program output.

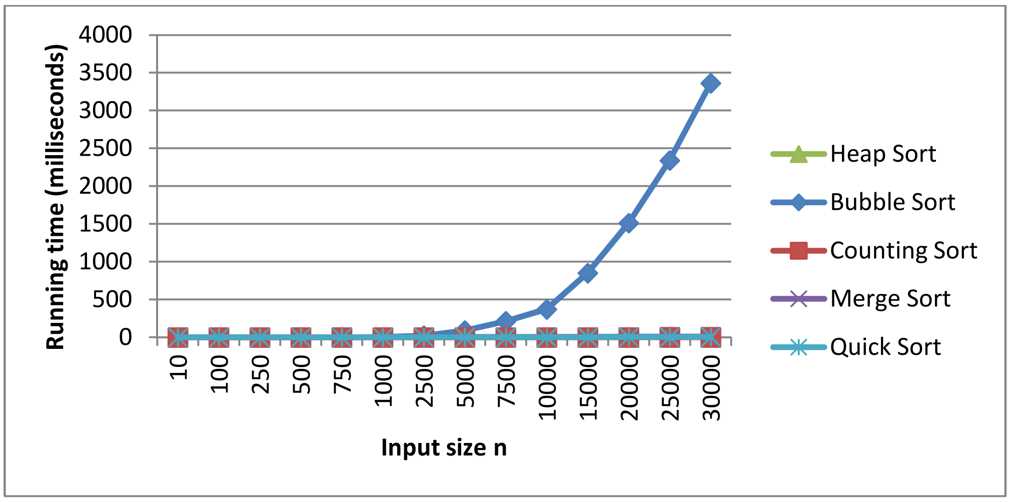


```
<terminated> Runner (36) [Java Application] C:\Program Files\Java\jdk-10.0.2\bin\javaw.exe (19 May 2019, 18:02:03)
Input size     10      100     250     500     750     1000    2500    5000    7500     10000    15000    20000     25000    30000
Bubble Sort    0.0063  0.4955  0.5914  1.6803  2.3687  3.9638  23.5243 91.5456 214.9261 372.2435 851.2639 1513.4461 2340.359 3362.6393
Heap Sort      0.0143  0.1395  0.1039  0.2276  0.3614  0.4349  0.9037  1.3462  2.2623   3.5661   4.346    5.9431    7.6302   9.4054
Counting Sort  0.006   0.0425  0.1009  0.1957  0.2998  0.279   0.1311  0.2616  0.3908   0.5065   0.6617   0.9376    1.1445   1.3867
Merge Sort     0.0186  0.1461  0.1145  0.1935  0.3137  0.4106  1.3959  3.4742  4.0435   4.9637   7.1161   8.1858    9.493    11.4579
Quick Sort     0.0059  0.0872  0.0669  0.0974  0.1717  0.2183  0.6767  1.7276  2.0553   2.013    3.0883   4.3395    5.4947   6.7825
```

Figure 2 The program output gathered in the table

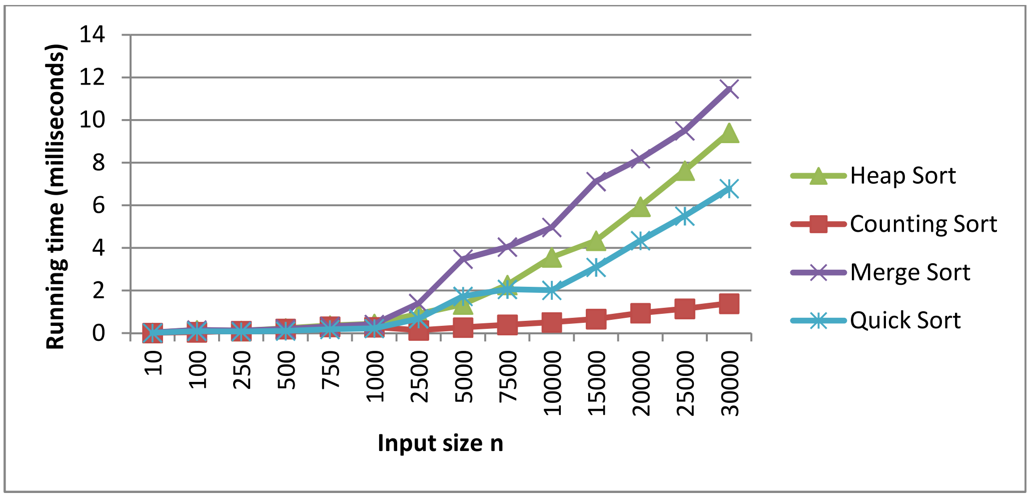
Input size n	Bubble Sort	Heap Sort	Counting Sort	Merge Sort	Quick Sort
10	0.0063	0.0143	0.006	0.0186	0.0059
100	0.4955	0.1395	0.0425	0.1461	0.0872
250	0.5914	0.1039	0.1009	0.1145	0.0669
500	1.6803	0.2276	0.1957	0.1935	0.0974
750	2.3687	0.3614	0.2998	0.3137	0.1717
1000	3.9638	0.4349	0.279	0.4106	0.2183
2500	23.5243	0.9037	0.1311	1.3959	0.6767
5000	91.5456	1.3462	0.2616	3.4742	1.7276
7500	214.9261	2.2623	0.3908	4.0435	2.0553
10000	372.2435	3.5661	0.5065	4.9637	2.013
15000	851.2639	4.346	0.6617	7.1161	3.0883
20000	1513.4461	5.9431	0.9376	8.1858	4.3395
25000	2340.359	7.6302	1.1445	9.493	5.4947
30000	3362.6393	9.4054	1.3867	11.4579	6.7825

Figure 3 Graph with running times of all analysed sorting algorithms.



Between all five analysed sorting algorithms, the Bubble Sort has the worst performance. As it is seen at the table, only for very small input, e.g. array of size 10, Bubble Sort runs faster than some of the other sorting algorithms: Heapsort and Merge Sort and it runs almost as quick as Quick Sort. The probable reason are the extra steps that are done by those three algorithms. Heapsort builds a heap at first step. Merge sort and Quick sort divides the input array.

Figure 4 Graph with running times of four analysed sorting algorithms.



The graph above presents the running time of four sorting algorithms: Heap sort, Merge sort, Quick sort and Counting sort. The first three algorithms have similar running time. They are comparison-based algorithms with the same average case running time n log n. The Quick sort has the worst case n^2. Nonetheless, it has the best performance from all analysed comparison-based sorting algorithms.

The best performance has Counting sort that is the non-comparison-based sorting algorithms. However, its performance is strongly connected with the input characteristics. It this case the input was a type of integers and the range of the key values was not significantly greater than the number of elements in a collection.

Bibliography

[1] Donald Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley 1968
[2] https://en.wikipedia.org/wiki/Total_order
[3] Donald Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley 1968
[4] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html
[5] https://www.geeksforgeeks.org/stability-in-sorting-algorithms/
[6] Donald Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley 1968
[7] https://en.wikipedia.org/wiki/External_sorting
[8] https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)
[9] https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
[10] https://en.wikipedia.org/wiki/Online_algorithm)
[11] Michael Goodrich, Roberto Tamassia, Michael Goldwasser, Data Structures & Algorithms, 2014, page 553
[12] Michael Goodrich, Roberto Tamassia, Michael Goldwasser, Data Structures & Algorithms, 2014, page 370
[13] https://en.wikipedia.org/wiki/Binary_heap
[14] https://medium.com/basecs/learning-to-love-heaps-cef2b273a238

[15] https://brilliant.org/wiki/counting-sort/
[16] https://en.wikipedia.org/wiki/Counting_sort
[17] Michael Goodrich, Roberto Tamassia, Michael Goldwasser, Data Structures & Algorithms, 2014, page 532
[18] https://medium.com/basecs/pivoting-to-understand-quicksort-part-1-75178dfb9313
[19] Michael Goodrich, Roberto Tamassia, Michael Goldwasser, Data Structures & Algorithms, 2014, page 544