Palestine Polytechnic University

College of Information Technology and Computer Engineering

Design and analysis of algorithms

Algorithms Measurement/profiling

Done By:

Maria Abu Sammour,

Haya Alameh,

Morjana Abu Hussein

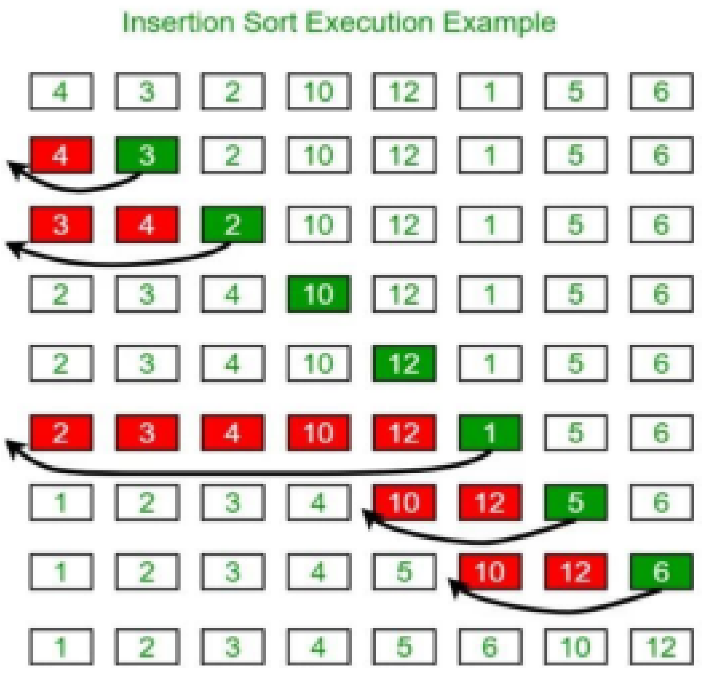
Instructor:

Dr.Faisal Khamaysa

In this project, we implemented 4 sorting algorithms, Insertion Sort, Merge Sort, Quick Sort, and Counting Sort.

Insertion Sort Algorithm

Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.



Insertion Sort work as follows :

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the elements in the array are at their apt position.

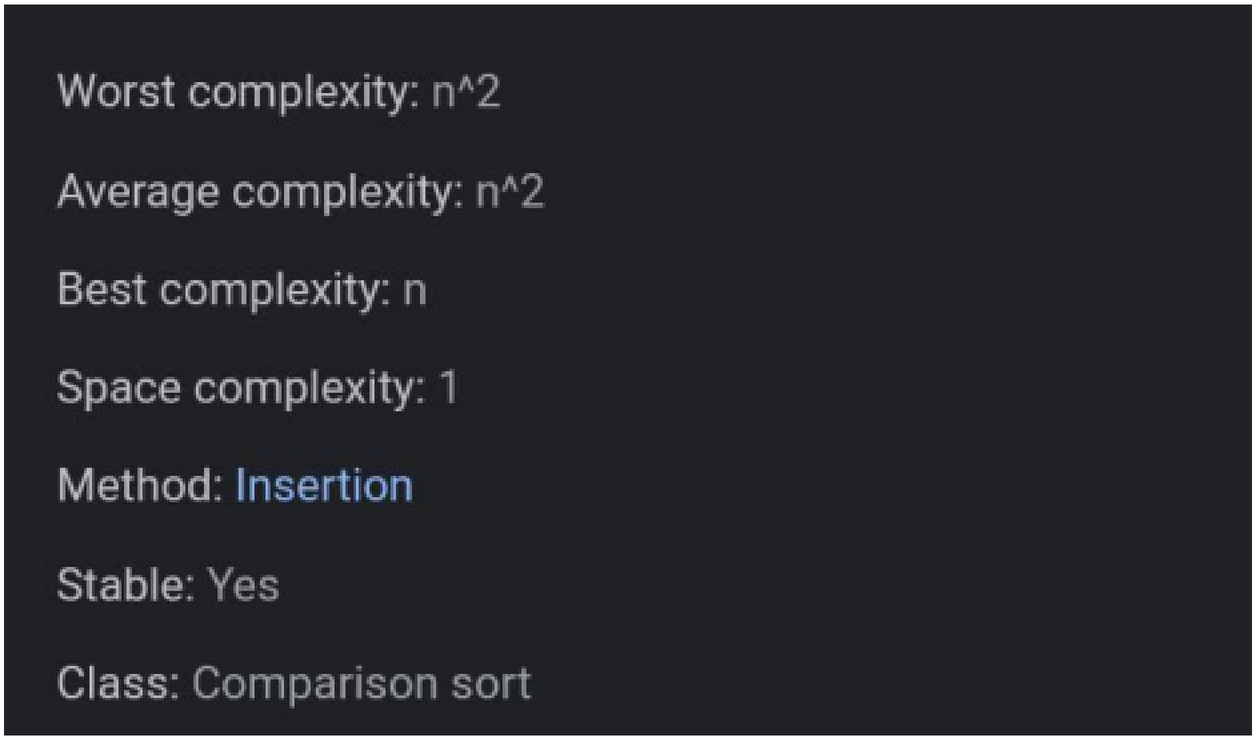
Pseudocode:

```
INSERTION-SORT(A)
 for i = 1 to n
 key ← A [i]
 j ← i – 1
 while j > = 0 and A[j] > key
 A[j+1] ← A[j]
 j ← j – 1
 End while
 A[j+1] ← key
 End for
```

Time Complexity Analysis:

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

Whereas for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements. Thus, making it for n x n, i.e., n2 comparisons. One can also take a look at other sorting algorithms such as Merge sort, Quick Sort, Selection Sort, etc. and understand their complexities.



Worst complexity: n^2

Average complexity: n^2

Best complexity: n

Space complexity: 1

Method: Insertion

Stable: Yes

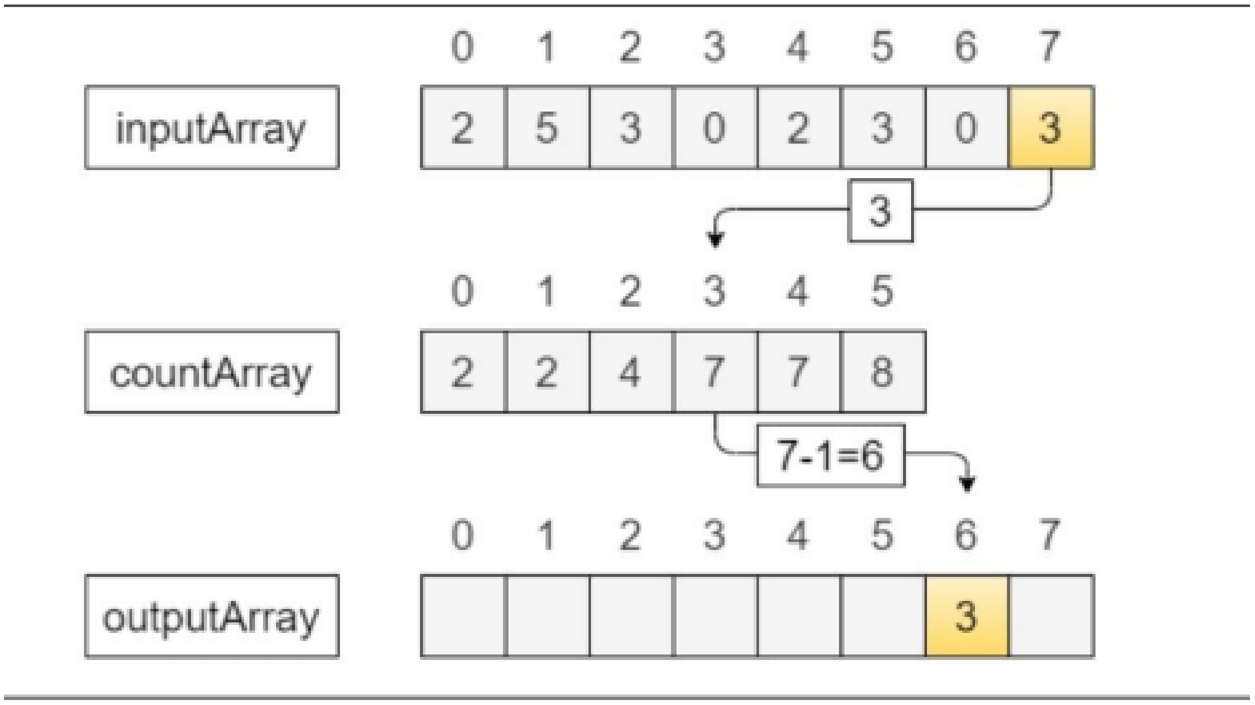
Class: Comparison sort

Counting Sort Algorithm

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then do some arithmetic to calculate the position of each object in the output sequence.
Counting sort makes assumptions about the data, for example, it assumes that values are going to be in the range of 0 to 10 or 10 – 99 etc, Some other assumptions counting sort makes are that input data will be all real numbers.

Like other algorithms this sorting algorithm is not a comparison-based algorithm, it hashes the values in a temporary count array and uses them for sorting.

It uses a temporary array making it a non In Place algorithm.



Counting sort

Points to be noted:

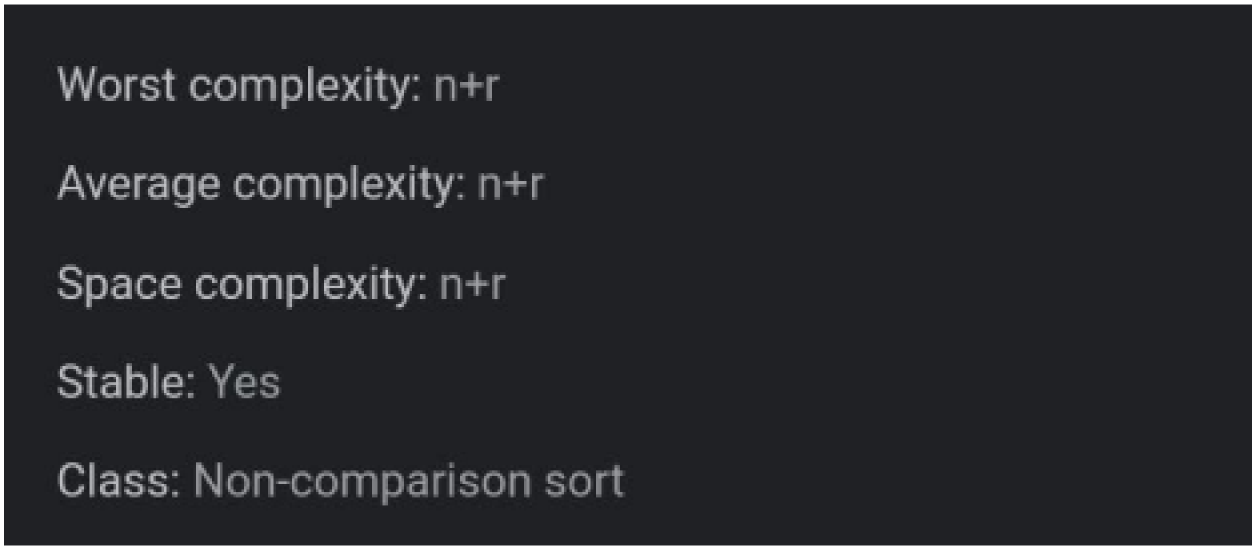
- Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
- It is not a comparison-based sorting. Its running time complexity is O(n) with space proportional to the range of data.
- Counting sort is able to achieve this because we are making assumptions about the data we are sorting.
- It is often used as a subroutine to another sorting algorithm like radix sort. • Counting sort uses partial hashing to count the occurrence of the data object in O(1).
- Counting sort can be extended to work for negative inputs also.
- Counting sort is not a stable algorithm. But it can be made stable with some code changes.

Pseudocode :

```
# variables
# input -- the array of items to be sorted;
# key(x) -- function that returns the key for item x
# k -- a number such that all keys are in the range 0..k-1
# count -- an array of numbers, with indexes 0..k-1, initially all zero
# output -- an array of items, with indexes 0..n-1
# x -- an individual input item, used within the algorithm
# total, oldCount, i -- numbers used within the algorithm
# calculate the histogram of key frequencies:
for x in input:
 count[key(x)] += 1
total = 0 # calculate the starting index for each key:
for i in range(k): # i = 0, 1, ... k-1
 oldCount = count[i]
 count[i] = total
 total += oldCount
# copy to output array, preserving order of inputs with equal
keys: for x in input:
 output[count[key(x)]] = x
 count[key(x)] += 1

return output
```

Time Complexity :



Worst complexity: n+r

Average complexity: n+r

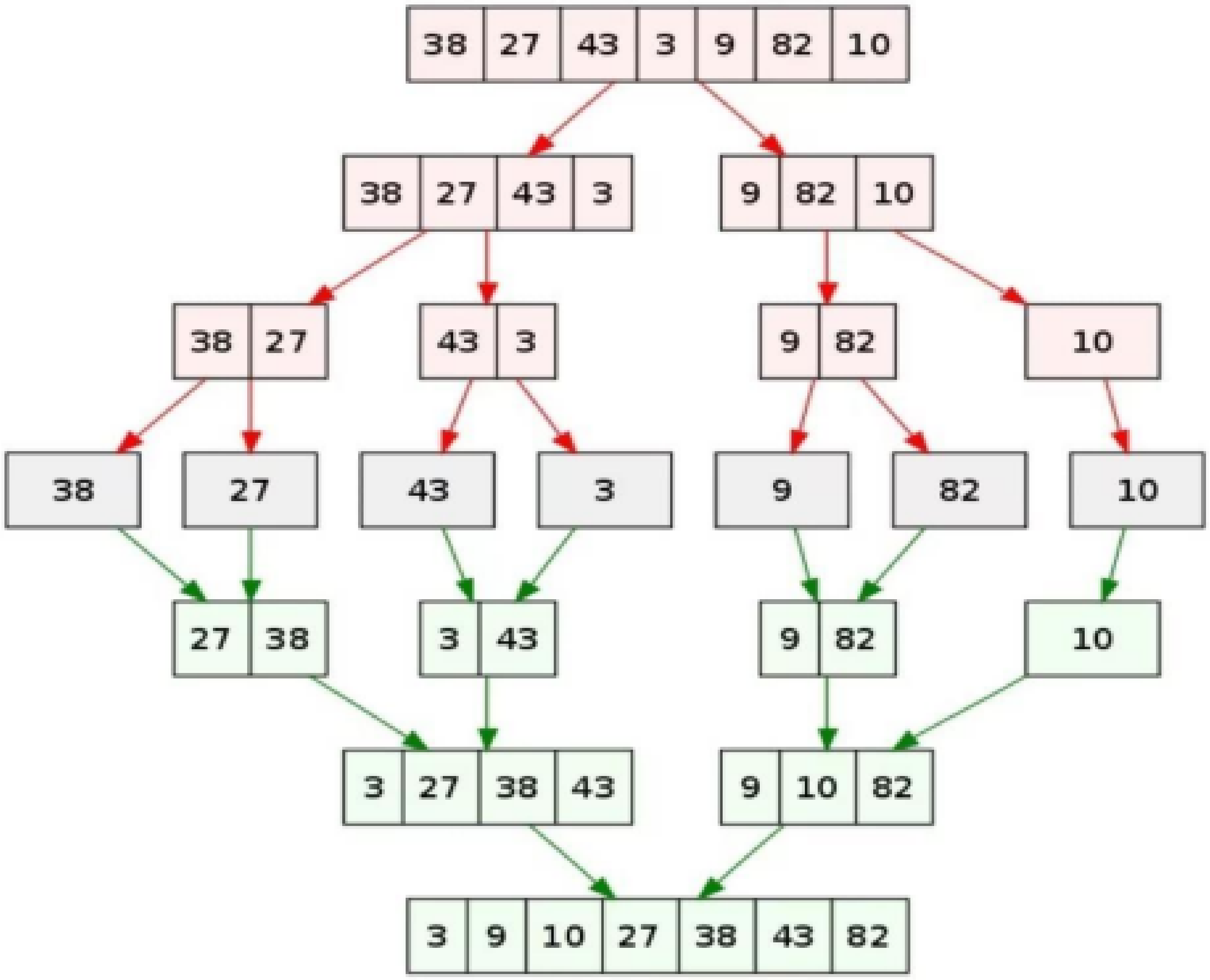
Space complexity: n+r

Stable: Yes

Class: Non-comparison sort

Merge Sort Algorithm

Merge sort is an efficient sorting algorithm that falls under the Divide and Conquer paradigm and produces a stable sort. It operates by dividing a large array into two smaller subarrays and then recursively sorting the subarrays.
In a recursive approach, the problem is broken down into smaller, simple subproblems in a top-down manner until the solution becomes trivial.

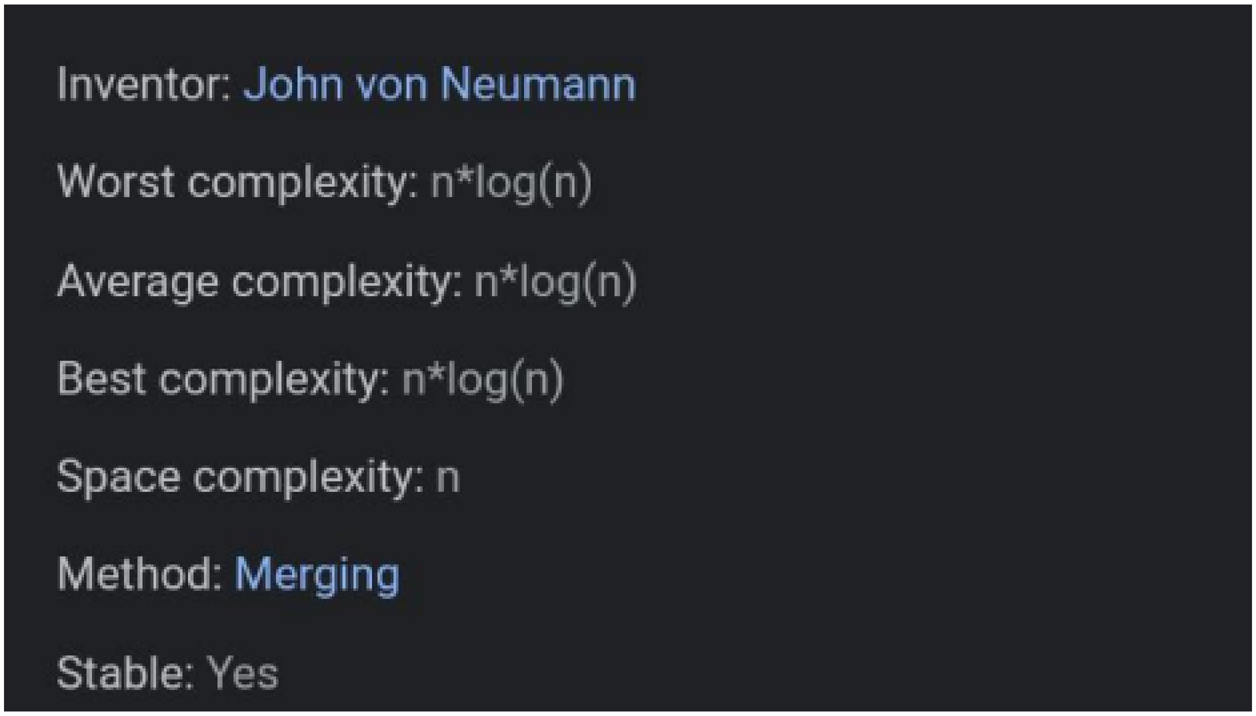


We can also implement merge sort iteratively in a bottom-up manner. We start by sorting all subarrays of elements; then merge results into subarrays of 2 elements, then merge 1. results into subarrays of 4 elements. Likewise, perform successive merges until the array is completely sorted.

The Pseudocode :

```
// Sort elements lo through hi (exclusive) of array A.
algorithm mergesort(A, lo, hi) is
 if lo+1 < hi then // Two or more elements.
 mid = ⌊(lo + hi) / 2⌋
fork mergesort(A, lo, mid)
 mergesort(A, mid, hi)
 join
 merge(A, lo, mid, hi)
```

Time Complexity:



Inventor: John von Neumann

Worst complexity: n*log(n)

Average complexity: n*log(n)

Best complexity: n*log(n)

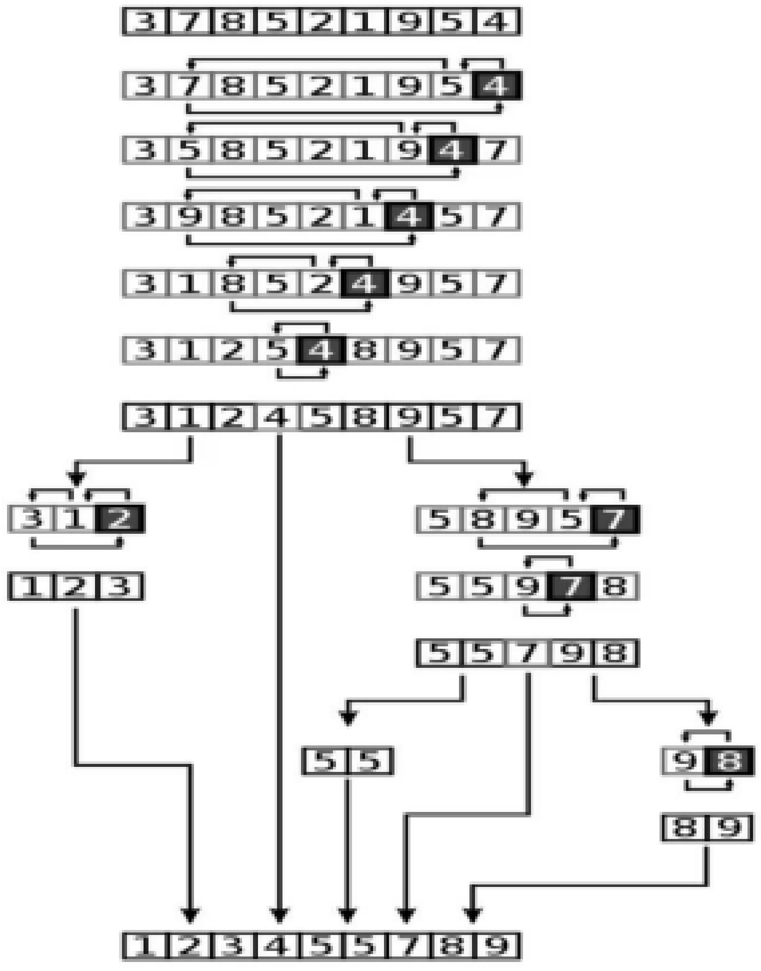
Space complexity: n

Method: Merging

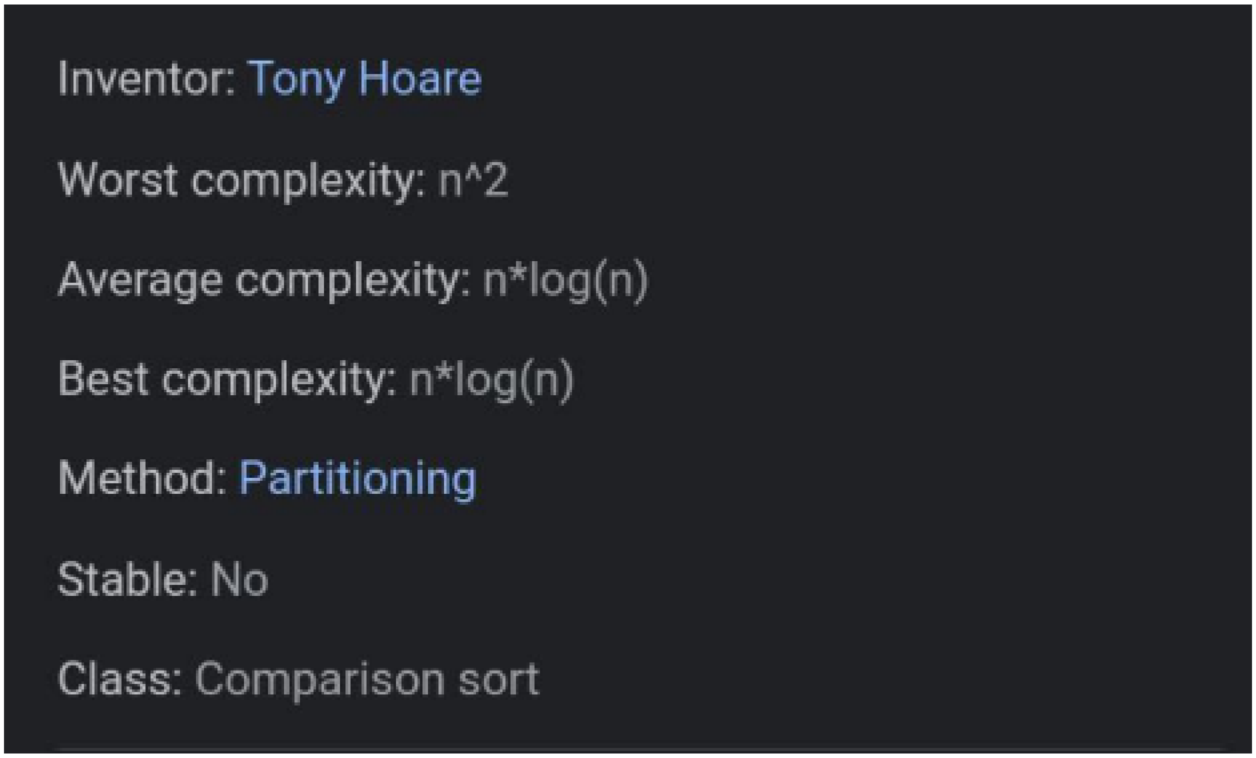
Stable: Yes

Quick Sort Algorithm

Quick sort is basically used to sort any list in a fast and efficient manner. Since the algorithm is inplace, quick sort is used when we have restrictions in space availability too. Please refer to the Application section for further details.Quick sort is basically used to sort any list in a fast and efficient manner. Since the algorithm is inplace, quick sort is used when we have restrictions in space availability too. Please refer to the Application section for further details.



Time Complexity :



Inventor: Tony Hoare

Worst complexity: n^2

Average complexity: n*log(n)

Best complexity: n*log(n)

Method: Partitioning

Stable: No

Class: Comparison sort

Points to be noted:

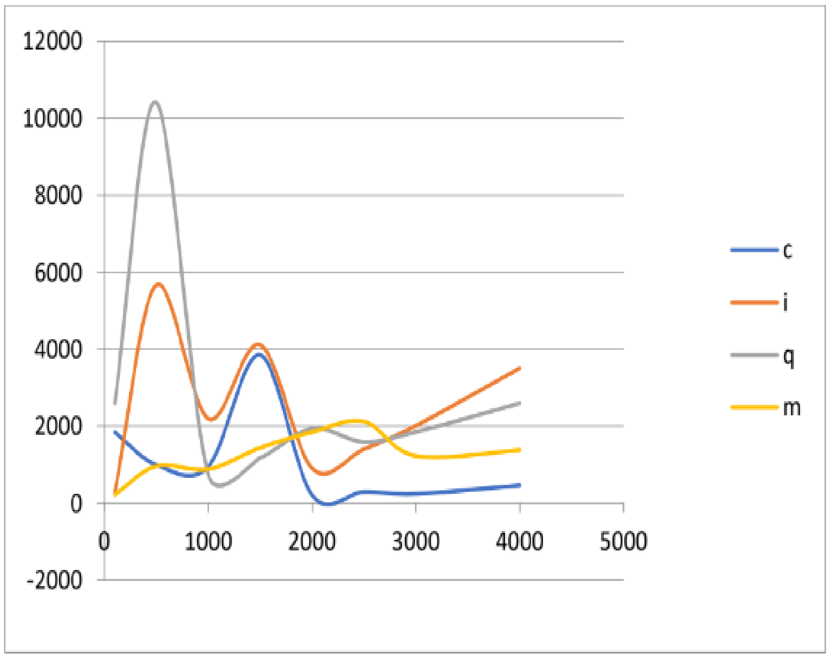
- Merge sort operates well on any type of dataset, whether it is large or small, Quicksort generally is more efficient for small datasets or on those datasets where the elements are more-or-less evenly distributed over the range.
- Speed :Merge sort generally performs fewer comparisons than quicksort both in the worst-case and on average. If performing a comparison is costly, merge sort will have the upper hand in terms of speed.
- Quicksort is generally believed to be faster in common real-life settings. This is mainly due to its lower memory consumption which usually affects time performance as well.

- The main difference between quicksort and merge sort is that 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.

We used Nano Method and converted it to Micro, and these are the results:

size	100	500	1000	1500	2000	2500	3000	4000
counting	1841.4	997.4	950	3852.2	208.2	296.9	245.7	463.1
insertion	291.5	5667.9	2193.8	4116	904.8	1406.7	2014	3508.7
quick	2597.2	10408.9	739.7	1176	1938.6	1590.5	1852.3	2597.5
merge	222.5	964.4	883	1440.1	1848.2	2121.1	1228	1382.6

Then implemented the results in a flow chart as shown below :



As we see here, time average complexity for counting sort is n so it remains constant, while Quick and Merge Sort are similar and the lines overlapped, but after a large number Quicksort needed a long time, and the Insertion Sort takes n^2 time average complexity so it is the biggest one.

At first, the results were uneven(nearly from 500 - 2000), then after a certain number(The point from which it began), the results appeared in a correct order which is Insertion-Merge Quick-counting(from slower to faster).