

**Project Tittle:** RUN-TIME ANALYSIS FOR SORTING ALGORITHMS

**Course Name:** Design and Analysis of Algorithms

**Instructor:** Dr.Faisal Khamayseh

* **Done by:**
* Sara Manasra (201115)
* Yafa Jaradat (201101)
* Mohammad Khamayseh ( 201300)

* **ABSTRACT:**

**Analysis of algorithms complexity is an issue that has always aroused great interest. This is because an algorithm, however ‘smart’ it may seem, it could require a huge execution time. There are analytic techniques of evaluating the complexity of an algorithm, although they usually are difficult to use. Most often, a more convenient solution is to estimate the run-**

**time of the algorithm.**

**Here, a run-time comparative analysis is presented. It consists in evaluating the run-times of three well-known sorting algorithms:**

**(InsertSort, MergeSort, QuickSort and CountingSort**.)

**5 different arrays of different sizes were randomly generated for the tests. Firstly, we find the running time for each algorithm with each input size and then we record these times in a table. Then, Plot the times for each sorting algorithm with all input sizes on the same graph. Finally, discussion regarding the times and/or the graph obtained in comparison with the known theoretical bounds.**

**The empirical results show that the fastest sorting algorithm is CountingSort, followed by QuickSort, followed by MergeSort, then by InsertSort.**

**This observation conforms to the theoretical time complexity.**

## **Introduction:**

**Most often in programming, when an algorithm is developed to solve a particular problem it is not regarded as a complete success. Often it emerges the question of the performance analysis for the created algorithm: complexity and portability on different systems.**

**Complexity is the measure by which an algorithm can be evaluated in comparison with other algorithms that solve the same problem. The aim of the complexity assess is to provide an information about the performances of an algorithm before transpose it in a programming language and without considering the particularities set of the incoming data**.

* **Evaluating the complexity of the algorithm will assume the estimation of:**
* **The time complexity:**

**Number of executions of a certain statement. Usually, this statement is the most significant for the particular problem to solve, referred as the base statement.**

**In sorting problems, the base statement is the comparison between items of the array.**

* **The space complexity:**

**The amount of supplementary memory needed.**

**The time complexity is a theoretical measure, usually it needs complex theories to be computed, therefore, in some cases, it is estimated using the run- complexity. This means the algorithm is implemented in any programming language and the run-time is computed on a particular machine.**

## **Presentation of the sorting algorithms**

* **Insertion sort algorithm:**

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

**Algorithm** :

To sort an array of size n in ascending order:

1. Iterate from arr[1] to arr[n] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped

**Example:**



* **Quick sort algorithm:**

**Quicksort is one of the most efficient sorting algorithms; it has the theoretical mean time complexity of O(n\*log n)**

**wrote his Ph.D. thesis about this sorting algorithm.**

**When implemented properly, Quicksort can be even three times faster in comparison with other sorting algorithm because of its outstanding performance. Precisely because of these performances, it is the most widely used algorithm. Nevertheless, like any type of sorting algorithm, sometimes it is not so efficient. The worst case scenario happens when the array is ordered descending, when it will be O (n2 ). It should be noted that this behavior is rarely met. The algorithm is based on a Divide et Impera method, it successively divides the array in two parts: the items in the left being less than a threshold, whilst the items in the right being greater than it.**

* **Merge sort algorithm** **:**

**It is an efficient, general-purpose,** [**comparison-base**](https://en.wikipedia.org/wiki/Comparison_sort)[**d sorting algorithm.**](https://en.wikipedia.org/wiki/Sorting_algorithm) **Most implementations produce a** [**stable sort,**](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability) **which means that the implementation preserves the input order of****equal elements in the sorted output.**

* [**Counting sort**](http://en.wikipedia.org/wiki/Counting_sort)

It 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 doing some arithmetic to calculate the position of each object in the output sequence.

**Why Quicksort is better than insertion sort?**

6 Answers. Insertion sort is faster for small n because Quick Sort has extra overhead from the recursive function calls. Insertion sort is also more stable than Quick sort and requires less memory.

Is Quicksort slower than insertion sort?

Quicksort is usually faster than sorts that are slower than O(nlogn) (say, Insertion sort with its O(n2) running time), simply because for large n their running times explode.

## **Results**

**In theoretical analysis, using - Line by Line - analysis , the time complexity is as shown below:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | **Insertion** | **Merge** | **Quick** | **Counting** |
| Time complexity | ϴ(n) | ϴ(nlogn) | ϴ(n2) | ϴ(n) |

**In the practical analysis, to verify these values in the average case, we choose a random data, then we sort these data using the four algorithms and calculated the time for each algorithm.**

**We got this table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **size** | **insertionSort** | **mergesort** | **countingsort** | **quicksort** |
| 1000 | 369 | 195 | 20 | 4 |
| 2000 | 2104 | 479 | 61 | 14 |
| 3000 | 4529 | 876 | 108 | 20 |
| 4000 | 7632 | 973 | 143 | 27 |
| 5000 | 9769 | 939 | 142 | 26 |

**In the graph, we took a new scale for insertion algorithm because its values are large compared to the rest of the algorithms, so after doing this we can draw it on the same graph, we took the square root of the algorithm values, then drew the graphic and eventually it became as below:**

## **Conclusions**

* **Insertion Algorithm:**

**the The algorithm obtained the largest time compared to the rest of the algorithms, and larger the size the more time it take , the increase was a quadratic increase, and this thing**

**coincided with theoretical analysis which we know (that the complexity is ϴ(n2)).**

* **Merge Algorithm:**

**Iterative Merge Sort:**

The above function is recursive, so uses [function call stack](http://en.wikipedia.org/wiki/Call_stack) to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution. Unlike [Iterative QuickSort](https://www.geeksforgeeks.org/iterative-quick-sort/), the iterative MergeSort doesn’t require explicit auxiliary stack.

Note: In iterative merge sort, we do bottom up approach ie, start from 2 element sized array (we know that 1 element sized array is already sorted). Also the key point is that since we don’t know how to divide the array exactly as in top down approach, where the 2 element sized array may be of size sequence 2,1,2,2,1…we in bottom up approach assume the array was divided exactly by powers of 2 (n/2,n/4….etc) for an array size of powers of 2, ex: n=2,4,8,16.

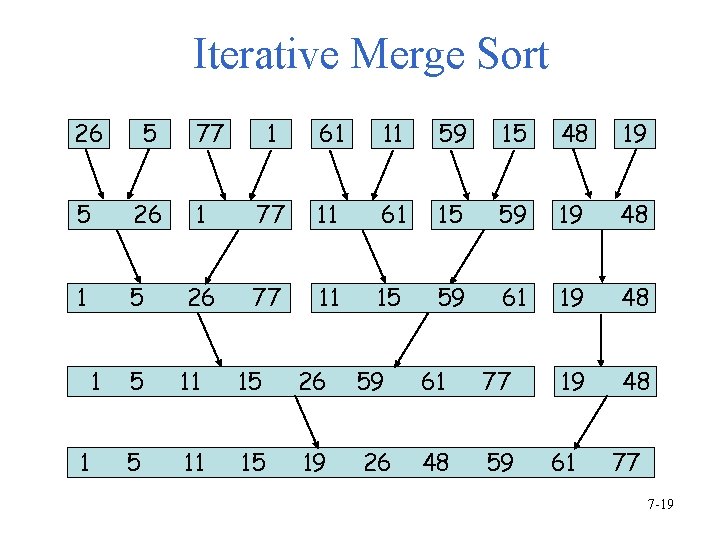
So for other input sizes such as 5, 7, 11 we will have remaining sublist that didn’t go into the power of 2 width at each level as we keep on merging and go upwards, this unmerged sublist which is of size that is not exact power of 2, will remain isolated till the final merge.

To merge this unmerged list at final merge we need to force the mid to be at the start of unmerged list so that it is a candidate for merge.

The unmerged sublist element count that will be isolated until final merge call can be found out using the remainder (n % width). The final merge (when we have uneven lists) can be identified by (width>n/2).

Since width grows by powers of 2 when width == n/2 then it means the input was already of size in powers of 2, else if width < n/2 then we haven’t reached final merge yet, so when width > n/2 we must be having pending unmerged uneven list hence we reset mid only in such case.

The above function can be easily converted to iterative version. Following is iterative Merge Sort.

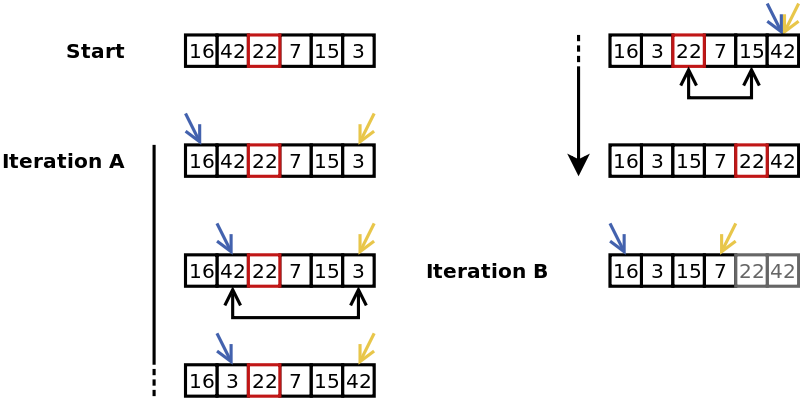


* **Quick Algorithm:**

Like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

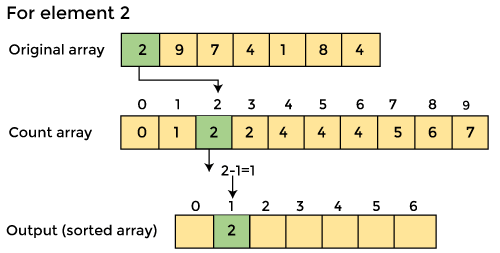
The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



The above-mentioned optimizations for recursive quicksort can also be applied to the iterative version.  
1) Partition process is the same in both recursive and iterative. The same techniques to choose optimal pivot can also be applied to the iterative version.  
2) To reduce the stack size, first push the indexes of smaller half.  
3) Use insertion sort when the size reduces below an experimentally calculated threshold.  
**References:**   
<http://en.wikipedia.org/wiki/Quicksort>  
This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

* **Counting 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 input data will be all real numbers.
* Like other algorithms this sorting algorithm is not a comparison-based algorithm, it hashes the value in a temporary count array and uses them for sorting.
* It uses a temporary array making it a non In Place algorithm.

**the time complexity is ϴ(n).**



Note:

* It is important for counting that data must be integer, not fractional, because these data will be indices for an array using within the execution of algorithm.
* We try to increase the ring, the time of the counting increase as the ring becomes bigger by the same size .So when we say that the relationship is Liner that with constraint which is the ring must be much less than the size.

**References:**

<https://www.geeksforgeeks.org/quick-sort/>

<https://www.geeksforgeeks.org/counting-sort/>

<https://www.geeksforgeeks.org/insertion-sort/>

<https://www.geeksforgeeks.org/iterative-merge-sort/>

Algorithm without tears\ By Dia' AbuZina .