Sorting algorithm analysis

Adrian-Ioan Tuns
FMI-IE-First year

Table of content

1.0 Introduction

This scientific paper has as a goal, to study the difference between sorting algorithms. Sorting is a classic subject in computer science.

For achieving this goal, an data input has undergone both theoretical and practical approaches. The theoretical approach approximates the effect of change on the size of input, to see how the speed of the algorithm changes as the input size increases. The practical approach consists of implementing the certain algorithms in C++, running the programms and checking their execution time.

At the moment of execution, all other applications were closed and the inputs given were in a range from a minimum of 0 elements to a maximum of 1 million elements.

2.0 Description of algorithm

For this experiment:

- The stored data consists of integers only;
- Data is stored in a text file;
- The order of the elements is randomized;
- The compiler used was CodeBlocks with C++ language.

There exists many algorithms for sorting. However, this paper will only make use of insertion sort, quick sort and heap sort.

2.1 Insertion sort

This is an comparison-based sorting algorithm that is easy to implement and requires a constant amount of O(1) of additional memory space. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted.

An element which is to be inserted in this sorted sub-list, has to find it's appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as it's average and worst case complexity are of O(n^2), where n is the number of items.

Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain.

The steps of insertion sort are:

Step 1 – if it's at the first element and it's already sorted, return 1;

Step 2 – pick next element;

Step 3 – compare with all elements in the sorted sub-list;

Step 4 – shift all the elements in the sorted sub-list that are greater than the value to be sorted;

Step 5 – insert the value;

Step 6 – repeat until list is sorted.

Worst case analysis:

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a running time of O(n^2).

2.2 Quick sort

To sort an array of n distinct elements, quick sort takes $O(n \log n)$ time in expectation. Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, called pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(nlogn), where n is the number of items.

It uses **divide and conquer** technique as follows:

Divide: Partition the array into two subarrays such that each element of the first half is less than or equal to pivot, which is, in turn, less than or equal to each element of the second partition.

Conquer: Sort the two subarrays by recursive calls to quick sort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them, as the array is now sorted.

Steps of quick sort pivot algorithm:

Step 1 – choose the highest index value as pivot;

Step 2 – take two variables to point left and right of the list excluding pivot;

Step 3 – left points to the high;

Step 5 – while value at left is less than pivot move right;

Step 6 – while value at right is greater than pivot move left;

Step 7 – if both step 5 and step 6 does not mach swap left and right;

Step 8 – if left is greater or equal to right, the point where they met Is new pivot.

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quick sort as follows:

Step 1 – make the right-most index value pivot;

Step 2 – partition the array using pivot value;

Step 3 – quick sort left partition recusively;

Step 4 – quick sort right partition recursively.

Worst-case analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size n − 1. This may occur if the pivot happens to be the smallest or largest element in the list.

If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. This means that the call tree is a linear chain of n −1 nested calls and the quick sort will have a running time of $O(n^2)$.

2.3 Heap sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort, an algorithm where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

First, what is a Complete Binary Tree must be defined. A Complete Binary Tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2, when index starts from 0.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order. Although heap sort it’s somewhat slower in practice on most machines than a well-implemented quick sort, it has the advantage of a more favorable worst-case O(*n* log *n*) runtime.

3.0 Experimental setup

This experiment was conducted in the following environment:

Operating System: Windows 10 Home

RAM: 16 GB

SSD: 256 GB

Processor: i7-7700HQ CPU quad-core 2.80

Compiler: CodeBlocks

Language: C++

The method used for measuring the time of the sorting algorithms is from <chrono> library. The following fragment of code was used for the measurement:

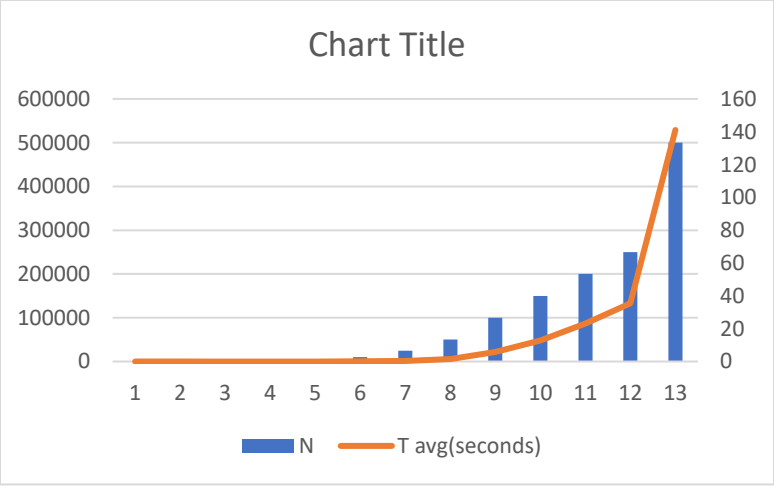
```
high_resolution_clock::time_point t1 = high_resolution_clock::now();

//sortingArray(array);

high_resolution_clock::time_point t2 = high_resolution_clock::now();

auto duration = duration_cast<nanoseconds>( t2 - t1 ).count();
```

The numbers generated were completely random, using Python random library, in PyCharm compiler.

4.0 Experimental results

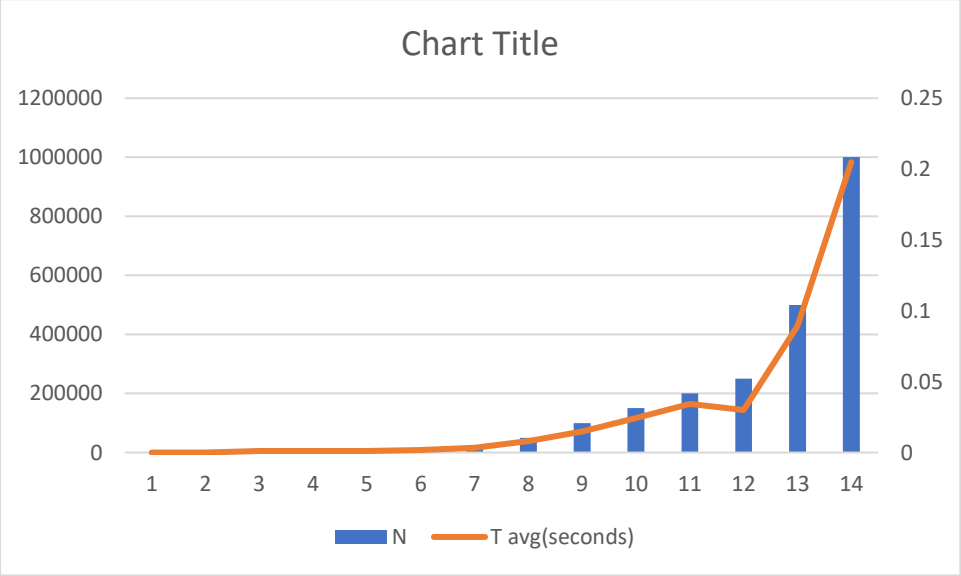
In this section, the results from the experiments were tabulated and charts were then made, with the respective data. The charts show the running-time T avg(seconds) and the input size N.

Insertion sort				
N	T avg(seconds)	T1(s)	T2(s)	T3(s)
0	0	0	0	0
500	0	0	0	0
1000	0.0000992	0.0000986	0.0000997	0.0000993
2500	0.002793467	0.0003988	0.0049906	0.002991
5000	0.015302	0.016933	0.01501	0.013963
10000	0.064499667	0.066822	0.060846	0.065831
25000	0.271939667	0.035931	0.369985	0.409903
50000	1.497652333	1.495993	1.486008	1.510956
100000	5.788853333	5.868278	5.834413	5.663869
150000	12.90380167	12.883548	13.091941	12.735916
200000	23.08006233	23.331528	22.792979	23.11568
250000	35.42784267	35.504962	35.479082	35.299484
500000	141.0506407	141.685831	140.786246	140.679845



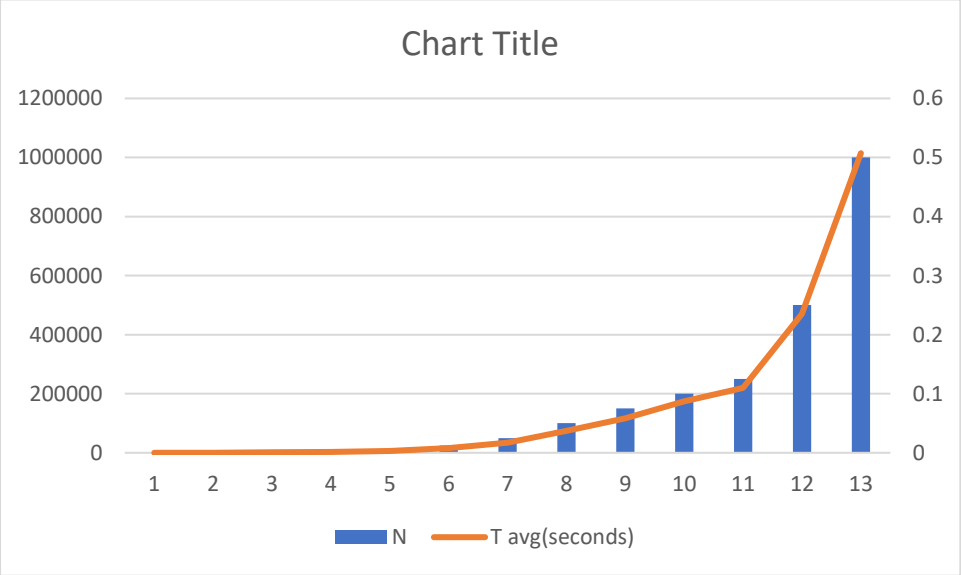
From the chart for insertion sort, we can see that the time required increases drastically after a certain point, as the input size gets bigger.

Quick sort				
N	T avg(seconds)	T1(s)	T2(s)	T3(s)
0	0	0	0	0
500	0	0	0	0
1000	0.001007667	0.001016	0.000996	0.001011
2500	0.000994333	0.000985	0.009004	0.000994
5000	0.001013333	0.000998	0.001044	0.000998
10000	0.001663667	0.001993	0.001004	0.001994
25000	0.003333333	0.004003	0.002991	0.003006
50000	0.008012667	0.00803	0.007974	0.008034
100000	0.014959333	0.014936	0.01496	0.014982
150000	0.024263333	0.024909	0.023945	0.023936
200000	0.034324667	0.034908	0.033912	0.034154
250000	0.030016667	0.041839	0.043735	0.004476
500000	0.089172	0.090757	0.091986	0.084773
1000000	0.204856333	0.227148	0.19948	0.187941



The chart for quick sort shows that our time increases almost linear, even though the size of the input gets to a large scale.

Heap sort				
N	T avg(seconds)	T1(s)	T2(s)	T3(s)
0	0	0	0	0
1000	0	0	0	0
2500	0.000729467	0.0001064	0.001047	0.001035
5000	0.001387667	0.000199	0.002001	0.001963
10000	0.002992667	0.002991	0.002992	0.002995
25000	0.007984	0.007978	0.007994	0.00798
50000	0.016963	0.016955	0.016956	0.016978
100000	0.037242667	0.036877	0.036951	0.0379
150000	0.058525333	0.057859	0.058872	0.058845
200000	0.087220333	0.087791	0.086105	0.087765
250000	0.110111667	0.110426	0.110211	0.109698
500000	0.235308333	0.230438	0.232378	0.243109
1000000	0.507236667	0.501441	0.494676	0.525593



The chart for heap sort shows just the same almost linear inscrease as quick sort, but there is a difference in the time required for sorting, heap sort being slower than quick sort on large inputs.

5.0 Conclusion

From the experiments, it’s conclusively that insertion sort takes too much time to sort a large set of data (ex: 1 million data set), but instead it’s faster when the array it’s small.

Both quick and heap sort are better for big sets of data. Both quick sort and heap sort perform at a similar speed as insertion sort, for small inputs.

From this graphs and experiments, quick sort is a little faster than heap sort and both of them are a lot faster than insertion sort.

References:

https://www.elsevier.com

https://www.wikipedia.org

https://www.khanacademy.org

https://www.geeksforgeeks.org

https://www.tutorialspoint.com/data_structures_algorithms