Algorithmic analysis of practicals

Practical 1 – Multiplication algorithms

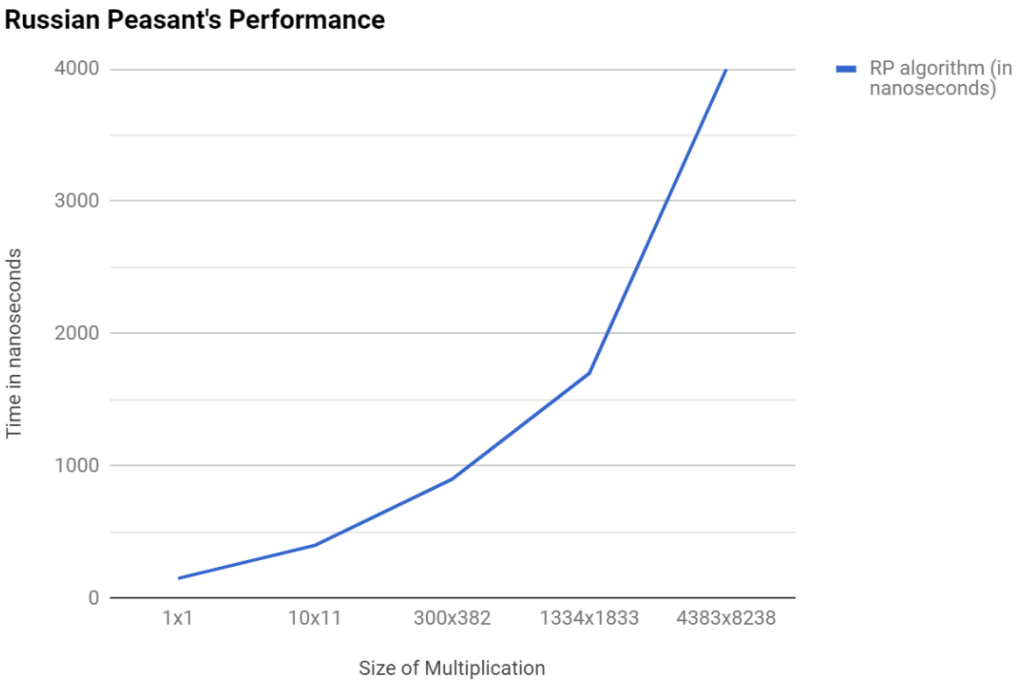
The *Russian Peasant's Algorithm* is an algorithm for multiplication that uses doubling, halving, and addition. This was an algorithm or a tool that was used before computers by people to multiply two numbers. One advantage it possesses over the standard method of multiplication that is taught in many schools (i.e. the Standard Algorithm) is that you do not need to have previously memorized multiplication tables to use the algorithm. In practice, the Russian Peasant's Algorithm was likely calculated with the aid of small stones or beads to represent the units.

Expected complexity:

Worst case time complexity: Θ(logn)

Space complexity: Θ(1)

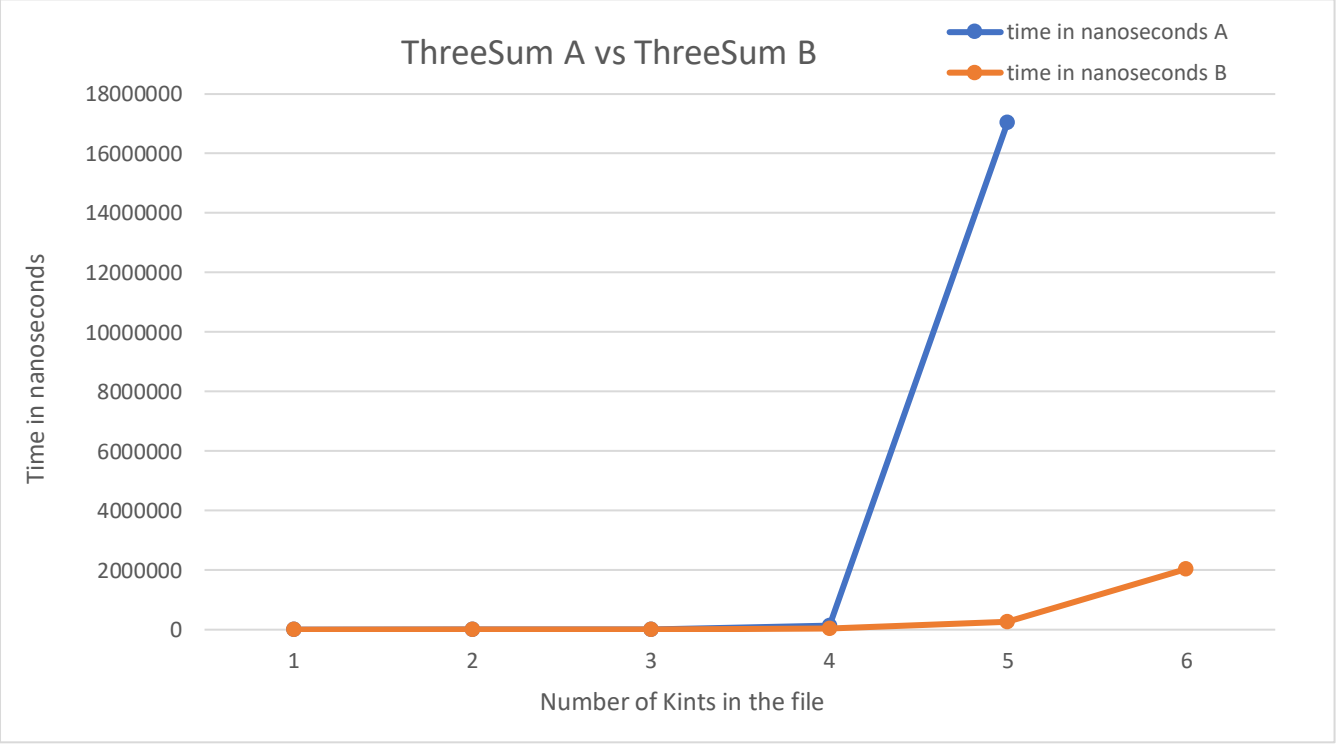
Graphing the results obtained from the execution of my implementation of the algorithm:



Practical 2- Asymptotic Analysis

Comparing two algorithms from different growth classes

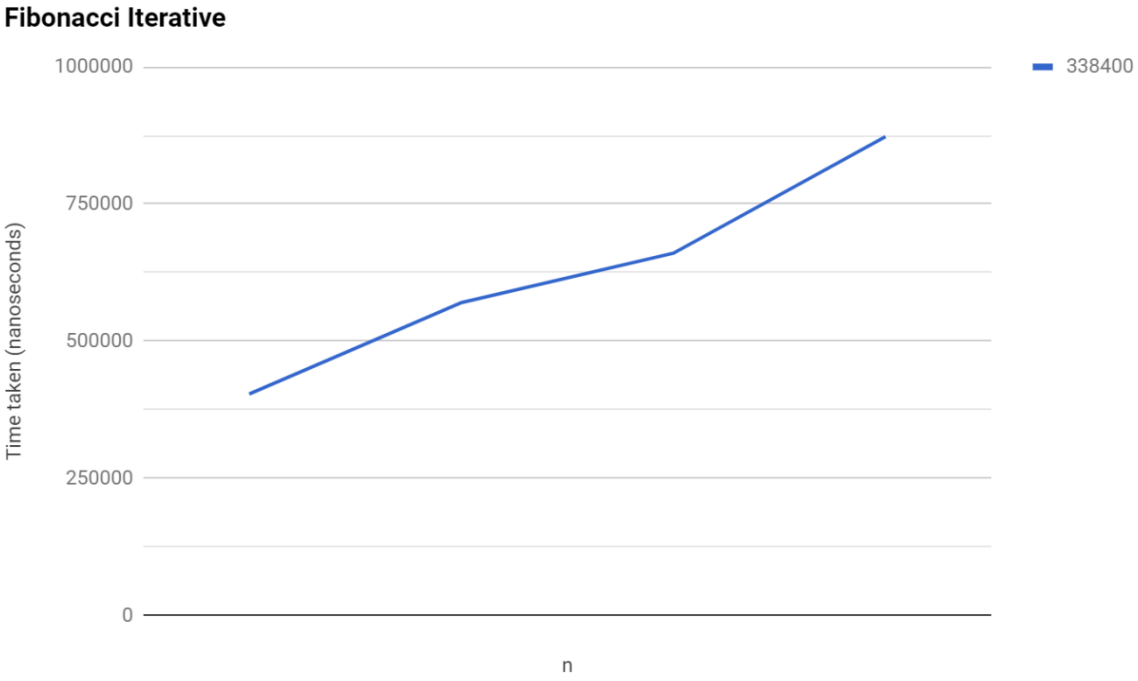
We have two algorithms (ThreeSumA and ThreeSumB) that count the number of triples in a file of N integers that sums to 0 (ignoring integer overflow).

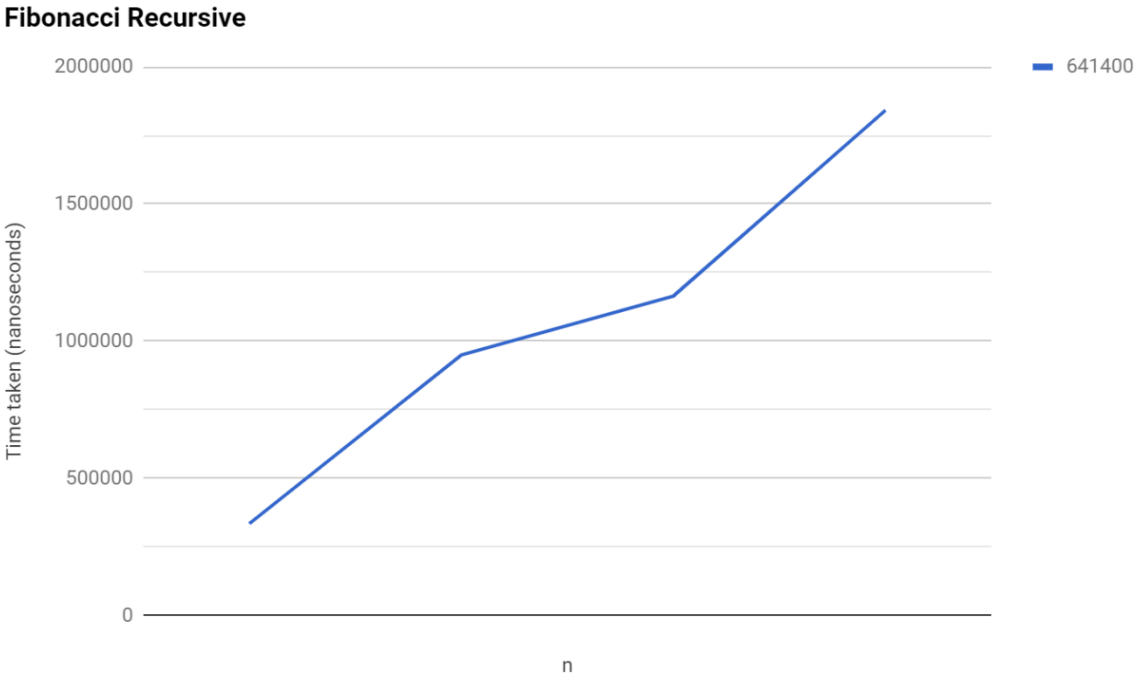


From the results above we can clearly see that the the ThreeSumB is more efficient for the file with larger number of triples in them. By examining the graph we can notice the rise in time taken to count the number of triples in the file. Although there is a time increase in both algorithms, the increase in the threeSumA is a lot more significant than the one in threeSumB. Hence, we can conclude that threeSumB is more efficient to use for larger input.

Practical 3 - Recursion

The Fibonacci Sequence is the series of numbers where the next number is found by adding up the two numbers before it.

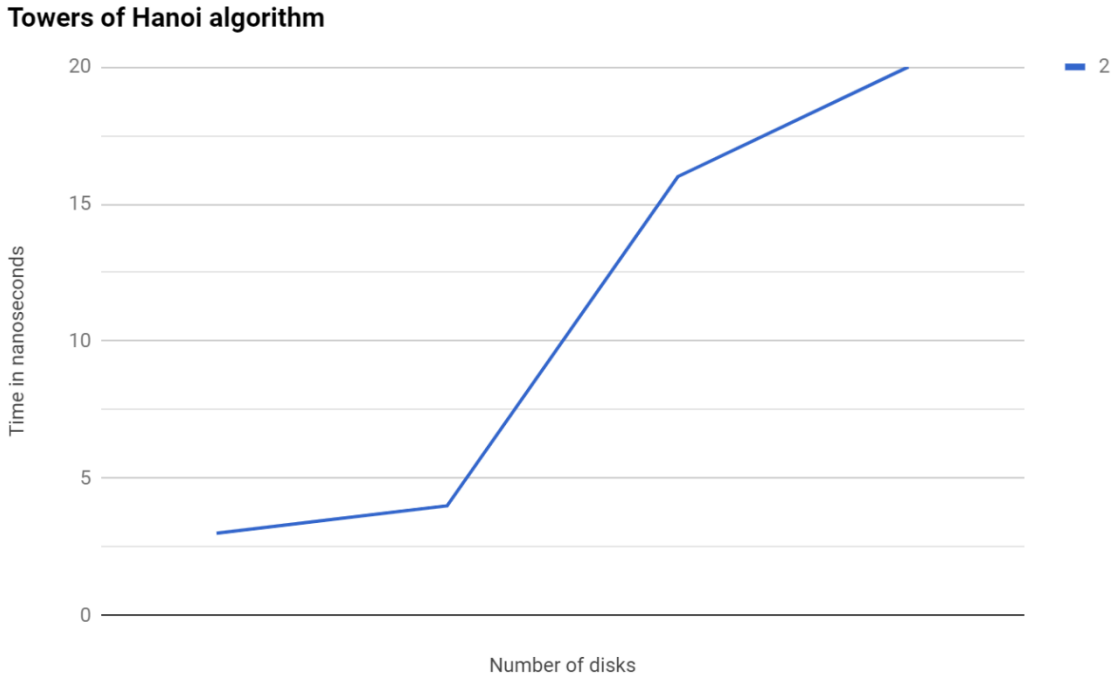




From the results above we can see that overall, the iterative Fibonacci approach is more efficient than the recursive one. The time complexity of the iterative code is linear, as the loop runs from 2 to n, i.e. it runs in O(n) time, while the time taken by recursive Fibonacci is O(2^n) or exponential.

For the iterative approach, as N changes the space/memory used remains the same. Hence it's space complexity is O(1) or constant.

For Fibonacci recursive implementation, the space required is proportional to the maximum depth of the recursion tree.



The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower[1] and sometimes pluralized as Towers) is a mathematical game or puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.

No larger disk may be placed on top of a smaller disk.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is 2n − 1, where n is the number of disks.

(https://en.wikipedia.org/wiki/Tower_of_Hanoi)

We can see from the graph above that the time complexity starts to increase quite rapidly when the number of disks used is 10 or more.

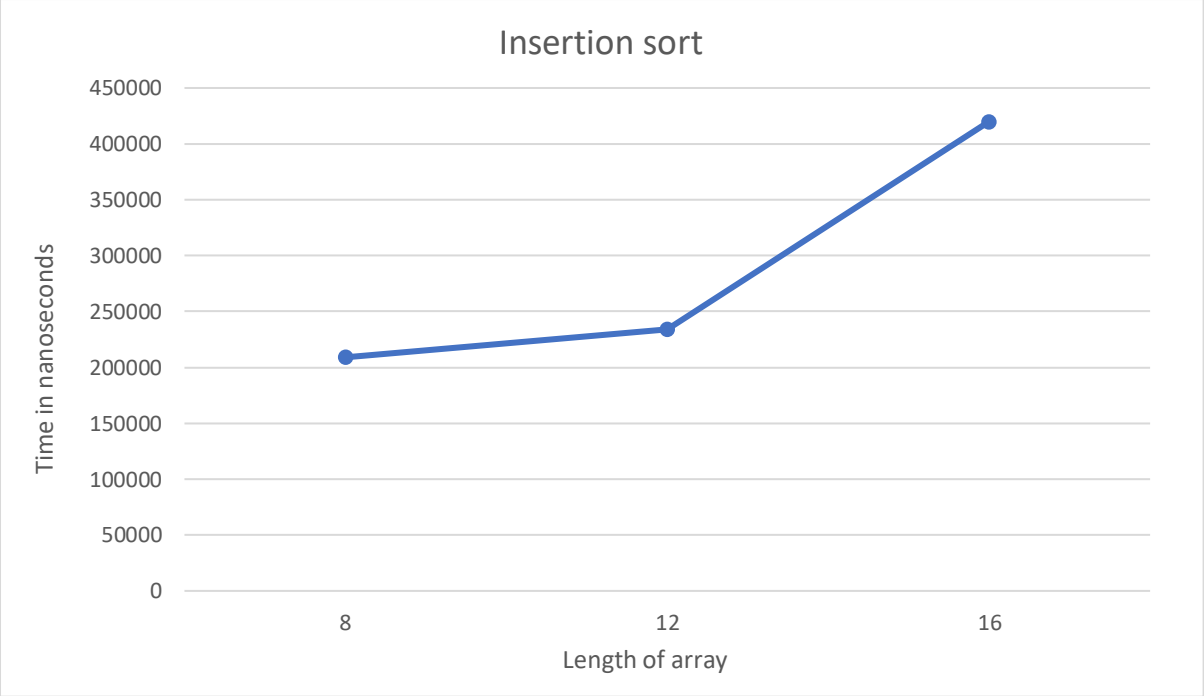
The minimum number of moves required to solve the problem is 2N-1. So overall it has O(2N) exponential running time complexity, where N is the number of disks.

Practical 4 – Sorting 1

Selection sort sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array. On each iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray. In the graph above I have timed my own implementation of selection sort. I'm aware of the fact that the graph should not have looked exactly like this however I'm unable to explain what caused the dip at length 12 to occur. Otherwise the time taken increases as it should.

Time Complexity: O(n2) as there are two nested loops.

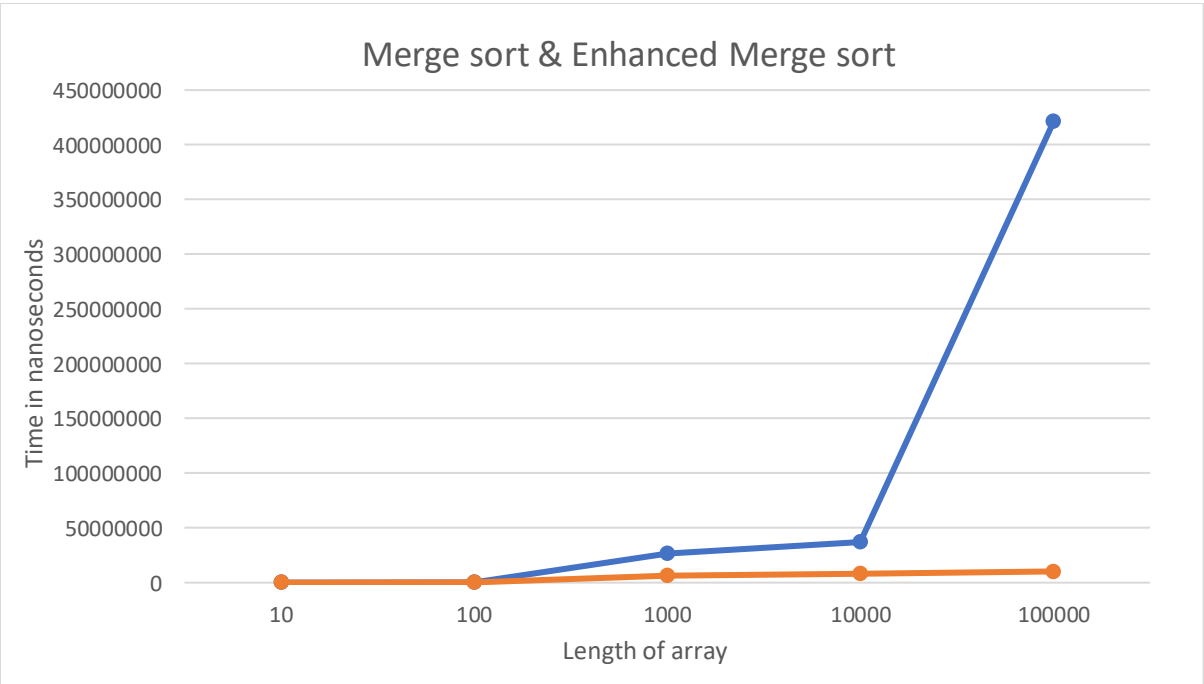
Auxiliary Space: O(1)



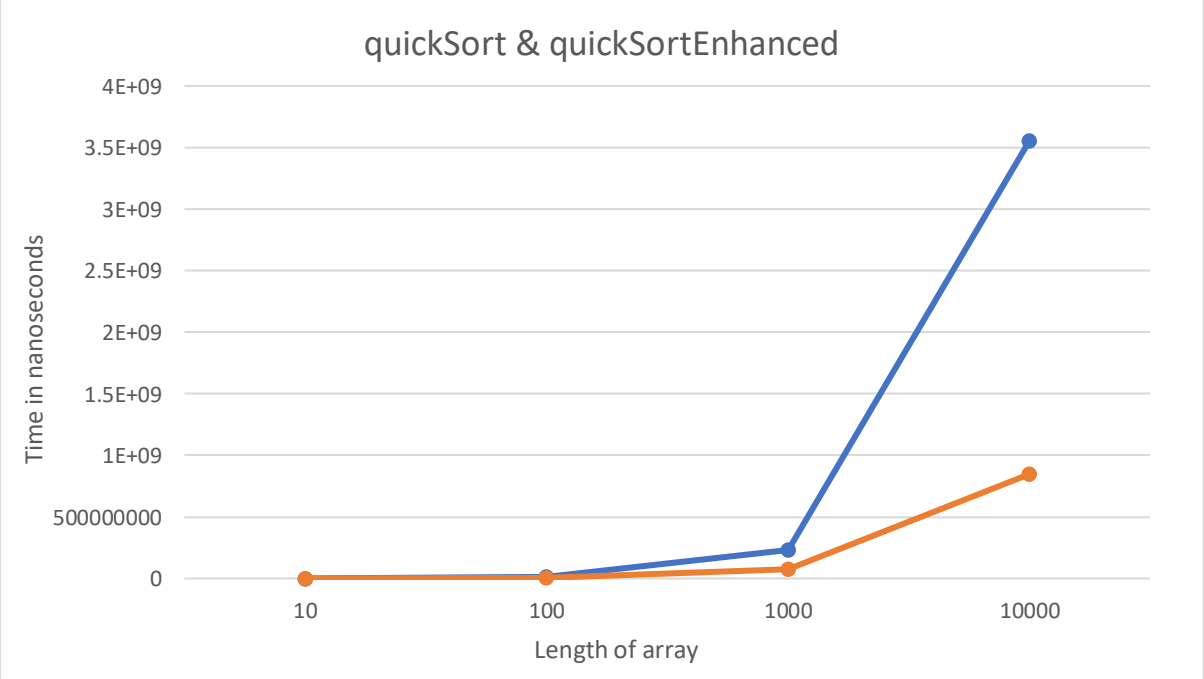
The graph increases as it should with the length of array. I think it encapsulates the O(n) complexity.

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. The time complexity is O(n). It is a good sorting algorithm to use if the input list is already mostly sorted.

Practical 5 – Sorting 2



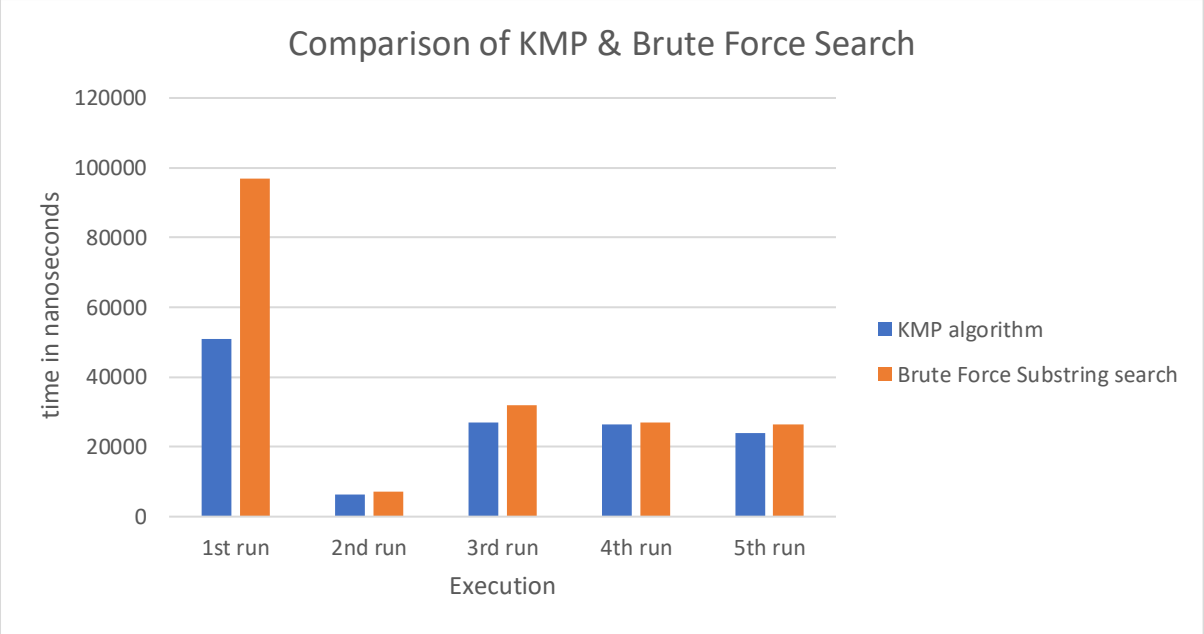
Looking at the graph above enhanced merge Sort made the algorithm 2 or 3 times faster. In order to enhance the algorithm I implement the 3-way merge and used insertion sort for input smaller than a certain cut off point in my case, cuttoff was set be to 8. The overall time complexity of Merge sort is O(nLogn). The space complexity however is O(n). This means that this algorithm takes a lot of space and may slower down operations for larger sets of data as input.



Quicksort is much faster than other O(nlogn) algorithms, due to the fact that its inner loop can be efficiently implemented on most architectures, and in most real-world data, it is possible to make design choices which reduce the probability of requiring quadratic time.

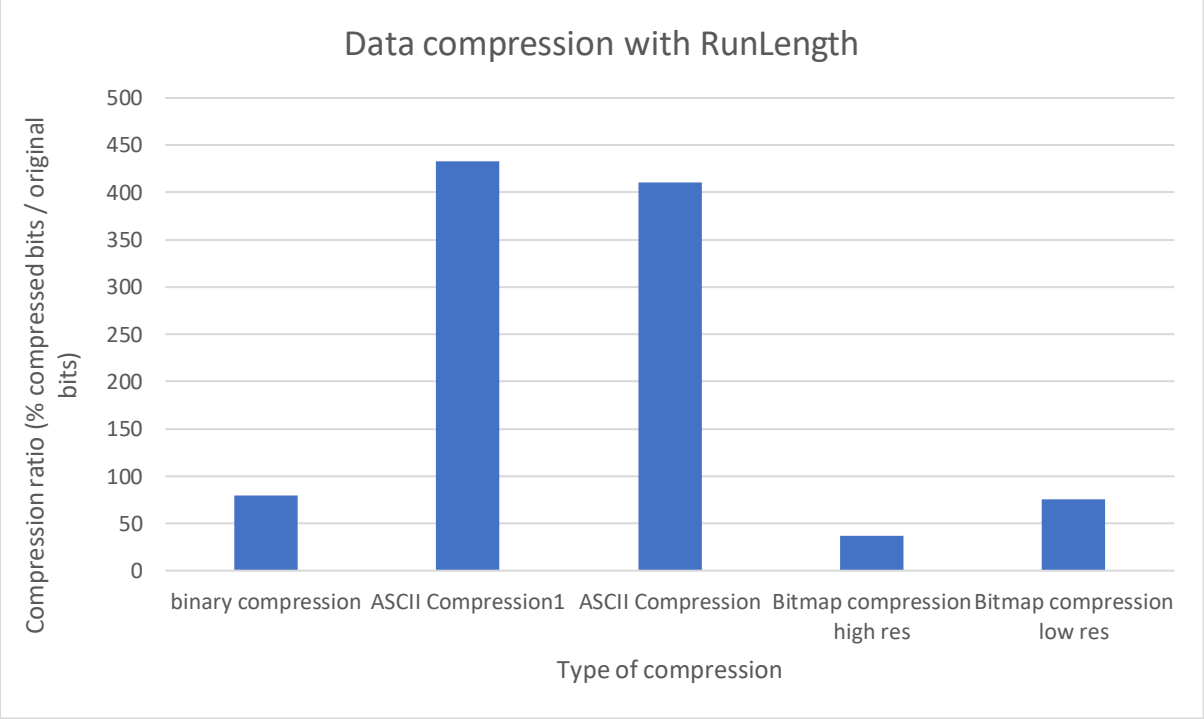
The fact that is has a lower memory requirement is also preferably for performance in a sorting algorithm.

Practical 7 – substring search



KMP is more efficient than the brute force approach since it doesn't have to search through every index of our string like the brute force does.

Practical 9 - Compression



From the graph we can see that the runlength compression is less efficient for files which contain alphanumeric characters. This is because it uses long runs of 1's and 0's for the compression which is why it's better for binary files, as can also be seen in the graph.