Algorithms Report Document

Student number: 21738885

Private Repository Link: https://github.com/alexanderretsis/Algorithms-COMP20290

Module Provided Repository Link:

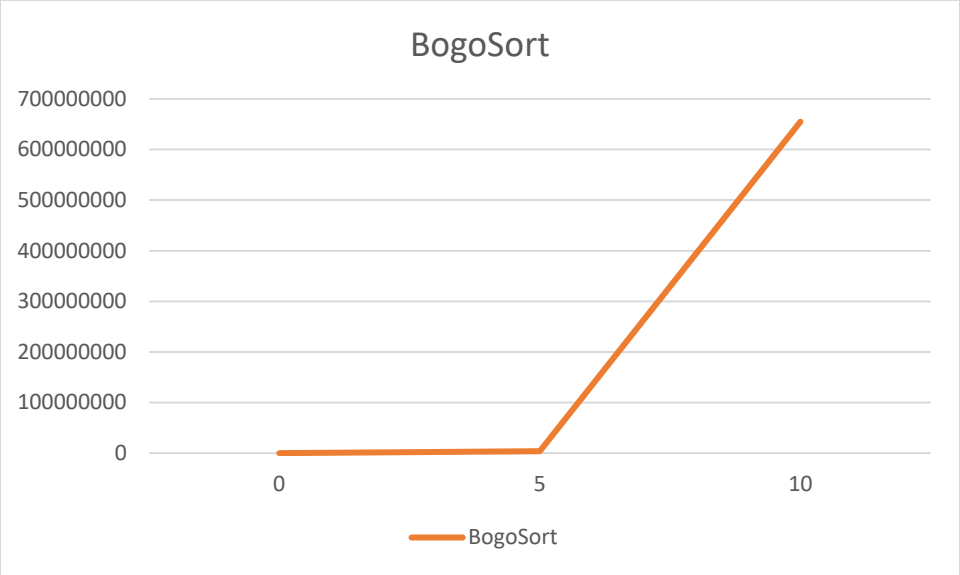
https://github.com/COMP20290-Algorithms/algorithms-repo-alexanderretsis

Sorting Algorithms:

To make sure timing is as accurate as possible **all tests were run at least 15 times**, all time is converted from **nanoseconds** to **milliseconds**.

Bogo Sort:

Number of elements	BogoSort
0	0
5	3.908
10	654.8682



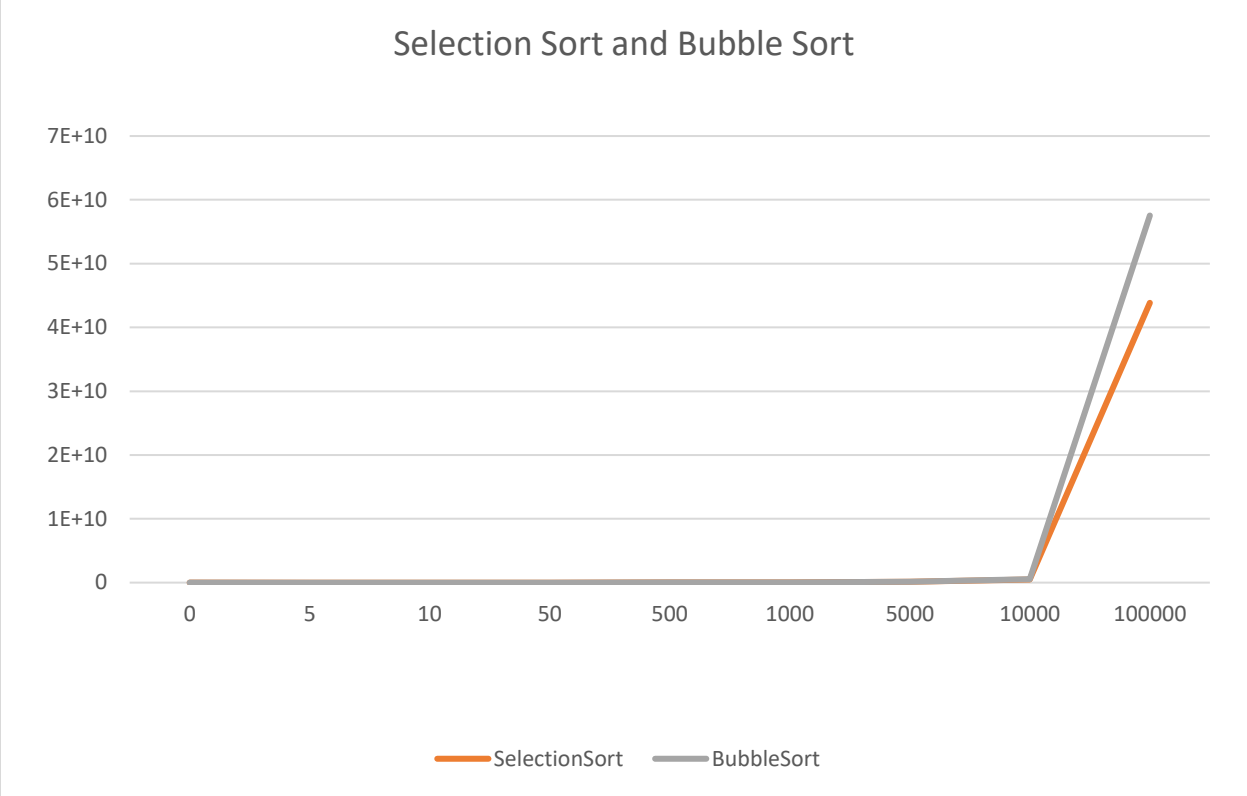
This is about as far as we can get with Bogo Sort without the times becoming ridiculously long, it's a very inefficient algorithm and not very useful at all.

It has a **time complexity of O(n*n!)** as on average the array will have to be sorted n! times before its sorted and we also have to run a check to make sure it actually is sorted and the **auxiliary space is O(1).**

Selection Sort and Bubble Sort:

TIMING

Number of elements	SelectionSort	BubbleSort
0	0	0
5	2.6375	2.5733
10	2.4343	2.1607
50	3.3954	3.2409
500	21.4443	18.8111
1000	31.7083	30.6730
5000	138.19	130.7624
10000	468.8898	512.6005
100000	43855.9907	57541.3315



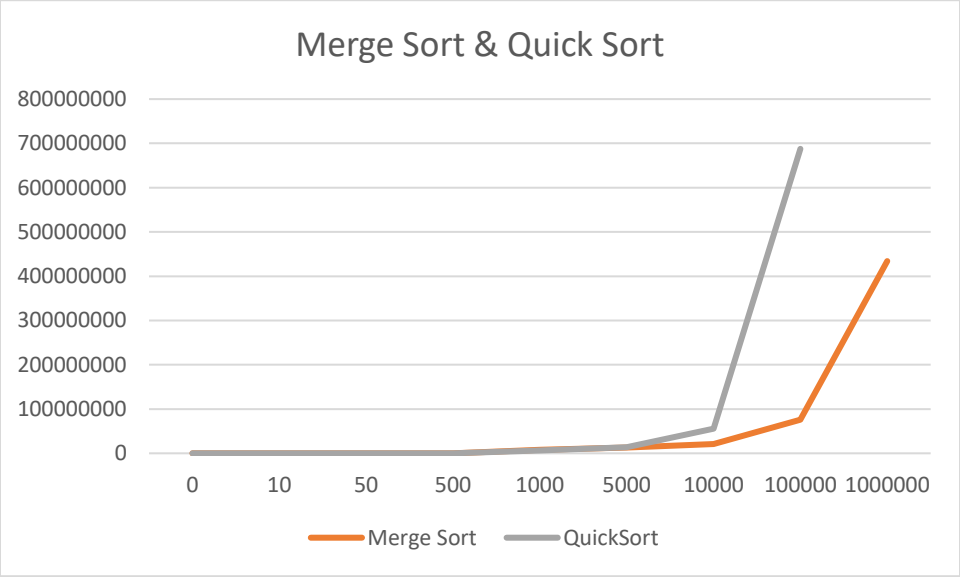
Selection sort, is a great algorithm for checking that all the elements are already sorted, it is also good if we don't have much memory space to work with as it doesn't use much temporary space, however it is not an efficient algorithm and as we can see in the graph is surpassed by **Bubble Sort in most cases.**

It has a **time complexity** of **O(n^2)** because of the 2 nested loops. While **space complexity** of selection sort since it is an in-place algorithm and needs no extra space is **O(1)**

Bubble sort, is good for smaller sets of elements in larger sets it becomes rather inefficient and we have better options to use in those cases it has a **time complexity of O(n^2)** since for each element every other element is also checked and a **space complexity of O(1)** since it requires no extra space.

Merge Sort and Quick Sort

Number of elements	Merge Sort	Quick Sort
0	0	0
10	2.6006	2.6031
50	3.1971	3.0532
500	5.9300	5.4225
1000	8.052	6.8519
5000	13.3556	13.6464
10000	20.7427	55.9435
100000	75.7682	687.6622
1000000	434.0985	**TOO LONG**



Merge sort, is more efficient for bigger sized arrays/lists in comparison to **quick sort**, and especially for linked lists. It has a **time complexity of O(n*log(n))** as the array/list is divided into two halves and then they are merged again in linear time, and a **space complexity of O(n)** as additional storage may be required

Quick Sort, is very efficient for smaller sized data pools and the fastest for information searching, it has a **time complexity of O(n*log(n))** we are splitting our array into 2 partitions of equal size as we would only need one additional partition level, As the array size doubles, so does the partitioning therefore we have log2n partitioning levels for n elements and a **space complexity** of **O(log(n))** as it is an in place recursive algorithm of **depth n.**

Binary String Search:

Binary Search Iterative, the time complexity of this algorithm is **O(log(N))** or **O(1)** in the case of an instant match.

Binary Search Recursive, the only difference with iterative is the **space complexity** in this case being **O(log(N)), time complexity** remains the same **O(log(N))**

Bruteforce & KMP String Search:

Bruteforce, is not very efficient and has many false positives, it can be fast if the pattern is found in the first iteration, the algorithm has a **time complexity of O(m)** in the best case, meaning if the pattern is found in the first m positions of text, and **O(mn) in the worst-case** meaning patter is not found in the first m positions o text and n comparisons have to be made, and if a **pattern is not found at all it has a timing of O(n), the space complexity of this algorithm is O(N).**

KMP, is a very efficient algorithm it holds key information in order to skip some loops and iterations in future, it has worse case efficiency as the preprocessing is always pre-defined and there are no worst cases, the negatives of this algorithm are that it may be inefficient as the alphabet increases and therefore large files with diverse ranges of characters will slow down our algorithm. The **time complexity is O(n)** And **space complexity O(n).**

RLE:

RLE, is a lossless compression algorithm of **time complexity O(n)** And **space complexity O(n),** it is very efficient for files that contain lots of repetitive data but not efficient at all in the opposite case, we also don't know how large the decoded data will be until we decode leading to problems if we are working with limited space.

Huffman.

Huffman, is a lossless compression algorithm of **time complexity O(nlogn)** since each iteration needs O(logn) time to determine the cheapest weight and add the new weight to it, while we have O(n) iterations for each item and a **space complexity of O(n),** it is most efficient for data with repeating and frequently occurring characters, and not at all optimal when we have a huge diversity of characters spread out throughout our data.

Dynamic Programming.

Fibonacci Recursive, the **time complexity** of this algorithm is **O(N)**
Fibonacci Dynamic, using a bottom up approach in this case the time complexity of our algorithm is **O(N)** and **space complexity** is **O(1)**

LCS dynamic, the **time complexity** of this algorithm is **O(xy)** and space complexity is also **O(xy) (**x,y being sOne.length, sTwo.length**).**

KnapSackBrute, the time complexity is **O(2^n)** and **auxiliary space is O(1).**

Knapsack Dynamic, the **time complexity** of this is **O(vW)** since we have a nested loop over **v** and a weight limit **W** and **space complexity** is also **O(vW)** since we are using a 2d array of size vW.