Kier McGuirk
18752609

Documentation

Practical One

The first practical involved the Russian Peasant algorithm which showed an effective way to multiply two large numbers. The individual questions regarding this practical are documented in a PDF in the lab 1 folder.

Table

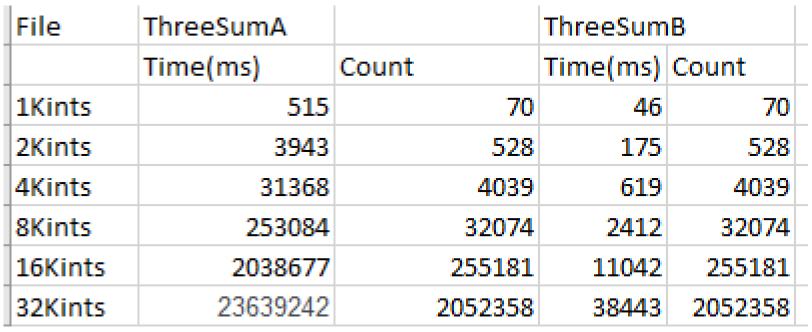
Input	Result	Time
RussianMultiply(10,10)	100	**0ms**
RussianMultiply(100,100)	10000	**0ms**
RussianMultiply(1000,1000)	1000000	**0ms**
RussianMultiply(10000,10000)	100000000	**0ms**
RussianMultiply(100000,100000)	1000000000	**0ms**

After increasing the inputs by an order of magnitude each iteration, it can be seen that the time taken is still infinitesimal. This is probably due to the fact that modern computers can compute such a low complexity algorithm very quickly.

Practical 2

Practical two consisted of questions and algorithms regarding algorithm complexity.

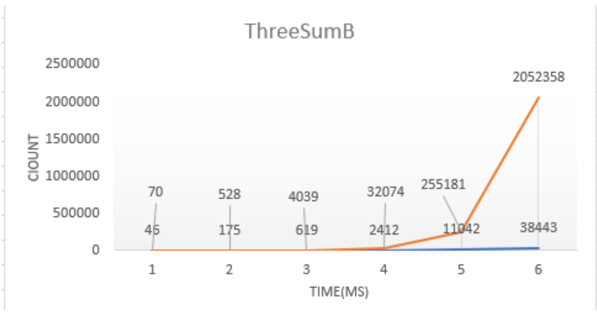
Data

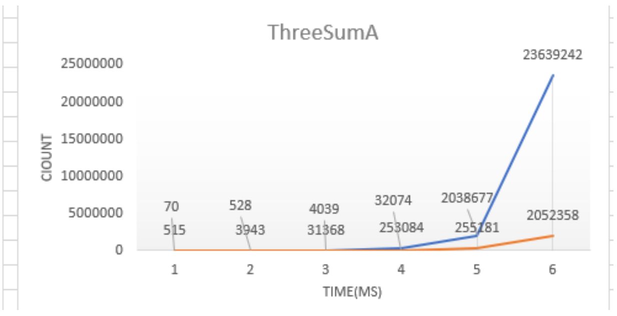


File	ThreeSumA		ThreeSumB	
	Time(ms)	Count	Time(ms)	Count
1Kints	515	70	46	70
2Kints	3943	528	175	528
4Kints	31368	4039	619	4039
8Kints	253084	32074	2412	32074
16Kints	2038677	255181	11042	255181
32Kints	23639242	2052358	38443	2052358

Kier McGuirk
18752609

Graphs





From the graphs, it can be seen that ThreeSumB is vastly more efficient than ThreeSumA. This makes sense, because ThreeSumB has a Binary Search Algorithm meaning that it will perform much more efficiently. The graphs are almost identical, meaning that the algorithms have the same complexity O(n^2).

Practical 3

The third practical saw use of different recursive algorithms, including the well-known 'Towers of Hanoi' problem and the Fibonacci Sequence. The individual recursion-based questions of the practical can be found in a PDF in the lab3 folder. I had created a for loop, to create the Fibonacci sequence from 0-9, using both recursive and iterative functions. We would expect that each function provides the same output, as they are the same algorithm but different renditions of each other.

Table - Fibonacci

Input (n)	fibIterative	fibRecursive
0	1	1
1	1	1
2	2	2
3	3	3
4	5	5
5	8	8
6	13	13
7	21	21
8	34	34
9	55	55

As expected, the two algorithms produce the same results, however, they do not create the "proper" Fibonacci sequence; it skips '0'. However, this is due to how the algorithms were constructed – to begin at 0 for the recursive algorithm n would have to begin with -1, but this cannot be achieved from the provided iterative solution, because (if (n<=1) return 1), which means the value of 0 can never be achieved. This led me to begin at n=0 to provide consistent results.

Kier McGuirk
18752609

Towers of Hanoi

For the Tower's of Hanoi, a solution that solves n disks has been provided in the code repository. It provides a sample of solving n=4 disks.

Practical 4,5,6

I hadn't access to the repo providing the starter code for these practicals, hence I created my own class called "sorting" located under lab4 in the repo. Please disregard the other sorting lab folders, as I have included all the methods n the Sorting class located in lab4.

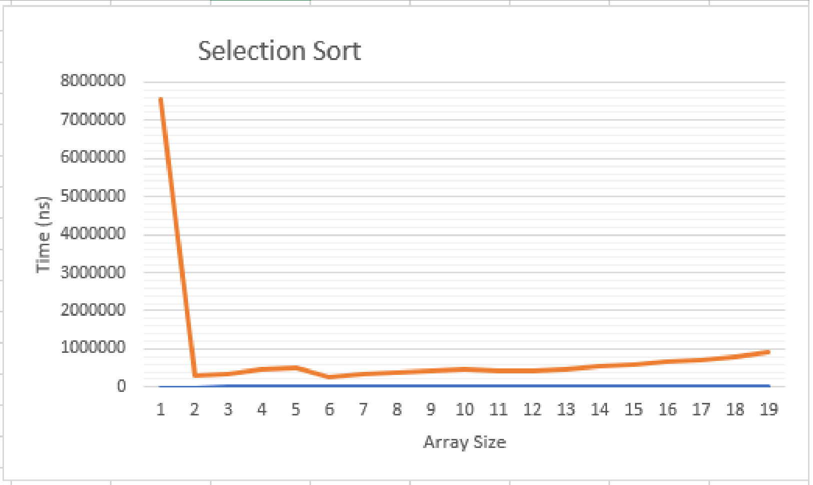
Selection Sort

The selection sort has O(n^2) runtime complexity, which is indicative that its efficiency decreases as the array size gets larger. A graph depicting runtime against array size, should be indicative of this. It is to be expected that there is as the array size increases, the runtime increases.

Results

Runtime against array size was plotted – for each array size, the selection sort would sort a random array 1000 times and note the total time. Here, the first entry is anomalous – it is extraneous how the first entry of array size 1 could take, by far, the longest amount of time to compute. This could be due to environmental factors in the hardware of the computer. We then see that there is a slight fluctuation in time, rather than the predicted gradual increase – this could be due to "luck" that parts of the array could have been randomly sorted. This is the most likely case, considering the predicted trendline is more consistent when the arrays become large (i.e when it is far less likely to be partially sorted).

Num Runs = 1000	
inputSize	Time
1	7541199
2	280470
3	331102
4	443731
5	510301
6	277622
7	320861
8	365225
9	418707
10	456802
11	437465
12	432927
13	477876
14	539876
15	590512
16	678118
17	726478
18	797586
19	926136



Kier McGuirk
18752609

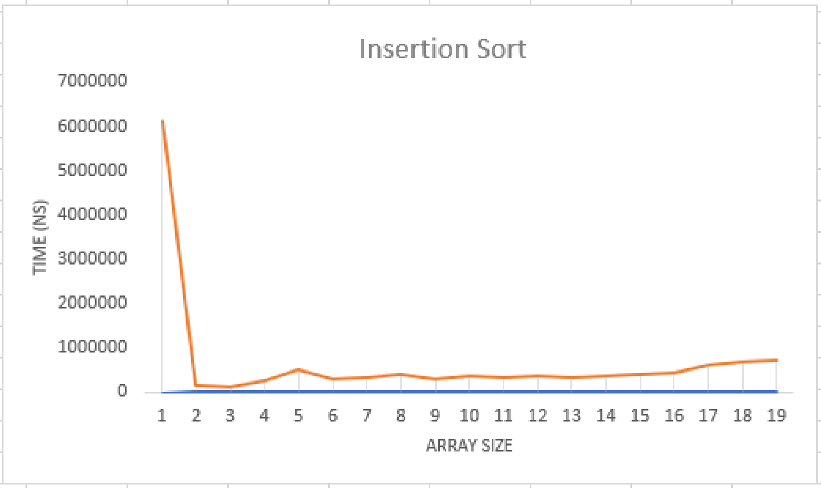
Insertion Sort

The insertion sort also has O(n^2) runtime complexity, which is indicative that its efficiency decreases as the array size gets larger. A graph depicting runtime against array size, should be indicative of this. It is to be expected that there is as the array size increases, the runtime increases. This graph should produce a graph very similar to selection sort.

Results

The first entry is again, anomalous. This is probably due to the aforementioned environmental factors. Moreover, the trendlines show the same early fluctuation and the consistency later on – very similar to the selection sort graph.

Num Runs = 1000	
inputSize	Time
1	6122382
2	151894
3	134830
4	247465
5	504027
6	282166
7	348164
8	394823
9	283301
10	357248
11	331093
12	365247
13	348161
14	370916
15	393671
16	424955
17	626346
18	674121
19	732156



Merge Sort (With selection sort)

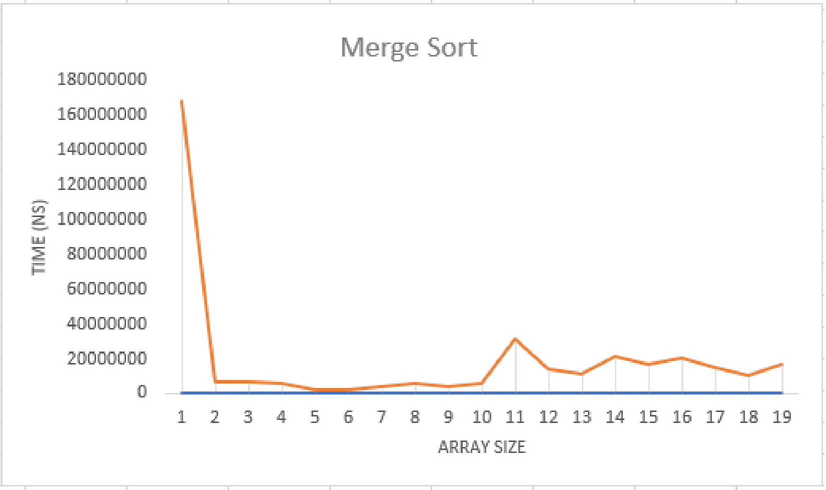
The merge sort has an overall complexity of O(nlogn), meaning that it is much more efficient at sorting larger arrays than the previous sorting algorithms. A graph should depict a much straighter line for this algorithm and a much faster time than the previous algorithms

Results

Astonishingly, the results for this are bizarre. It by far has the longest time out of all of the algorithms. It also creates a very unpredictable trendline. The time does seem to increase as the array size increases, however, these results are still unexplainable. It follows a very similar line as the insertion sort (since it effectively is an insertion sort for the first ten elements) however, after this it becomes bizarre.

Num Runs = 1000

inputSize	Time
1	1.69E+08
2	6790834
3	7021225
4	5826025
5	2301722
6	2325042
7	3738180
8	5655324
9	3964579
10	5878897
11	31139302
12	14234185
13	11042139
14	21230377
15	17230524
16	20142666
17	14718873
18	10018164
19	17268634



Kier McGuirk
18752609

Quick Sort (With selection sort)

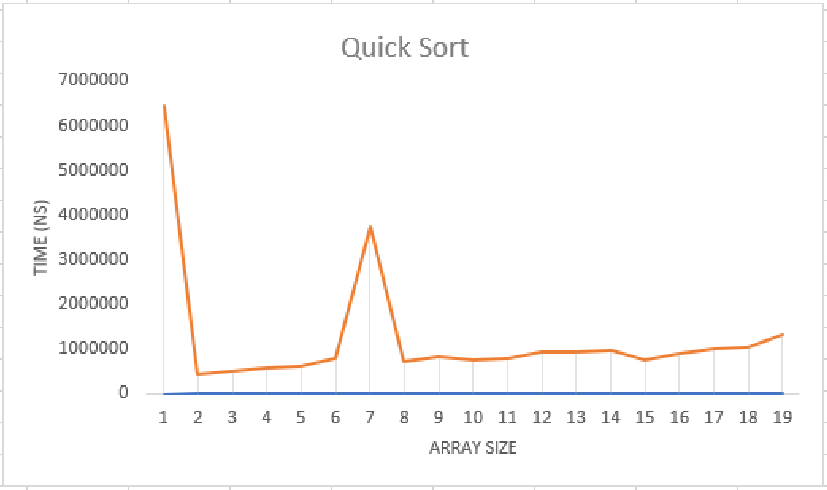
The quick sort also has an overall complexity of O(nlogn), meaning that it is much more efficient at sorting larger arrays than the previous sorting algorithms. A graph should depict a much straighter line for this algorithm and a much faster time than the previous algorithms (insert and selection).

Results

There are two anomalous values: the first and seventh entries. The first entry has been explained previously, however, it seems like the seventh entry must have had probably the worst case runtime for insertion sort. After this it can be seen that the quick sort algorithm is very efficient and the runtime barely increases for the greater array sizes.

Num Runs = 1000

inputSize	Time
1	6464290
2	428946
3	515418
4	590517
5	603591
6	798141
7	3738180
8	729314
9	823749
10	744116
11	787922
12	935862
13	955742
14	958014
15	758888
16	905112
17	997269
18	1059841
19	1332344



Silly Sort

The tests for this were omitted, because I chose to implement the bogo sort, which takes very long to do with even small array size inputs. The working algorithm can still be seen to work in the Sorting class.

Kier McGuirk
18752609

Practical 7

Practical 7 consisted of pattern searching in strings, using a brute force search and a kmp search.

```
String: String txt1 = "ABABDABACDABABCABAB";
String pat1 = "ABABCABAB";
```

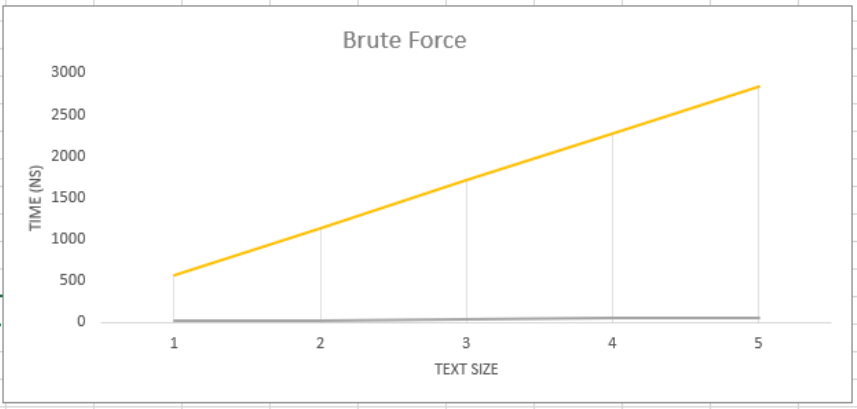
This was the original string and pattern – however, to compare efficiencies I created 4 more strings, which would add the characters ABABDABACD to the previous string, making it ten characters longer and ten characters longer to find the pattern. The time taken to find the pattern for each text 1000 times was recorded and the length of the string was also recorded.

Predictions

I would predict that the KMP search would be able to find the pattern more efficiently because it is a more efficient algorithm – the KMP search has O(n) complexity whereas the brute force search has O(n^2) complexity. We should expect the graphs for each algorithm to be similar due to the simplicity of the test cases. Moreover, we should expect to see that the time should increase proportionally to the increase of the text size (considering that the pattern is found at the end) of each text.

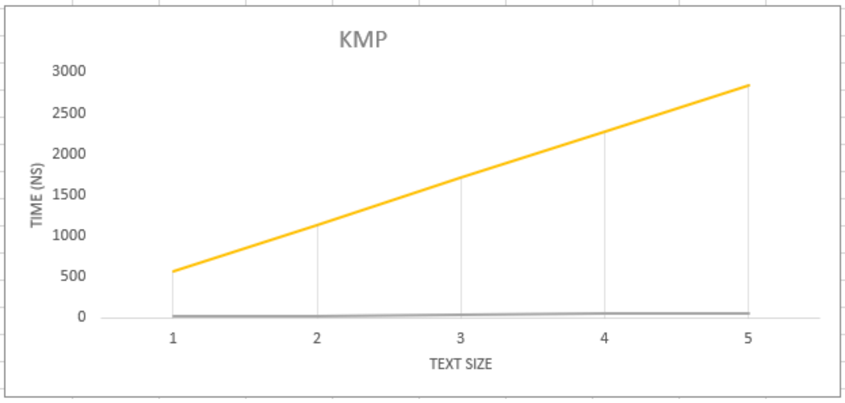
Results

Text Size	Time
19	569
29	1138
39	1707
49	2276
59	2845



Here, we see that the brute force algorithm is plotted Time against Text Size. We see that the graph depicts a trendline where the time is proportional to the text size (as expected.

Text Size	Time
19	1138
29	1707
39	2276
49	3414
59	3983



Likewise, an identical trendline is depicted for the KMP search. However, what is surprising is that the brute force algorithm was actually faster. However, I believe this to be due to the fact that the pattern is at the end of each text.

Practical 8

This practical consists of Tries and the code and hand-drawn trie can be found in the repo under lab8.

Practical 9

This practical consists of run length encoding and the answers and code can be found under Lab 9 in the repo