Nicholas Ang
CS302

Homework 2 Documentation

Bubble Sort Class Implementation:

I implemented this class with seven public methods and three private members. The public methods are the default constructor, destructor, readFileToArray, writeArrayToFile, sortB, getArr, and getSize. The private members are size, maxSize, and the array pointer. The default constructor dynamically allocates memory for the array of maxSize and sets the amount of values in the array to zero. The destructor deletes the dynamically allocated array and sets its pointer to nullptr. The readFileToArray method allows me to get data from a file and place it inside the object. It reads the values line by line and inserts it into its corresponding place in the array. The writeArrayToFile method outputs the values of the array one by one to a designated file. For example, sorting an unsorted 1000.txt file would write the values to the 1000_sortedQ.txt file. The getter getArr method returns the pointer of the array. This allows me to manipulate and sort the array inside the object. The getter getSize returns the amount of values that are in the array. The sortB method traverses through the array, comparing the values of an element with the value next to it. If it is greater than the value next to it, it swaps places with it.

Quick Sort Class Implementation:

I implemented this class with eight public methods and three private members. The public methods are the default constructor, destructor, readFileToArray, writeArrayToFile, sortQ, partition, getArr, and getSize. Most of the methods in the quickSort class are the same or similar to those in the bubbleSort class. The big difference is in the sorting methods which for the quickSort class are sortQ and partition. The sortQ method sets a value as the pivot by calling the partition method and then recursively calls itself to partition and sort the left and right sides of the pivot. The partition method selects the last value in the array as the pivot. It then loops and checks whether the current element it is looking at is less than the pivot. If it is, then it checks if the pivotIndex is the same as the current index. When it is not the same, it swaps the pivotIndex and the current index value and increments the pivotIndex value. When it is the same, it just increments the pivotIndex. After looping, it swaps the pivotIndex and the last element and returns the pivotIndex.

Main CPP file:

The main function is a menu that has 4 options. The first option bubble sorts and quick sorts a file with a thousand random values. This is looped ten times and outputs the average time, swaps, and comparisons. The second option bubble sorts and quick sorts a file with ten thousand random values. This is looped ten times and outputs the average time, swaps, and comparisons. The third option does the same thing as the first and second options but uses a file with a hundred thousand random values. This is also looped ten times and outputs the average time, swaps, and comparisons. The values of the final iteration of the loop are outputted to the corresponding sorted text file. I did this to be able to check whether my algorithms and code were sorting the values properly. I also added a function to create text files with random values from 0 to 1000000. The values are written to a file for as many values that I want to sort.

I compiled the sortMain executable with cmake. In the folder with the CMakeLists.txt file, I entered cmake ./ into the terminal. Then I entered make all to create all the executables. The randomly generated values and files are created when options 1, 2, or 3 are selected for its corresponding amount.

Output:
Option 1 - Sort 1000:
```
      BUBBLE SORT 1000
      Avg Time Taken: 0.00886213
      Avg # of Comparisons: 499500
      Avg # of Swaps: 255818

      QUICK SORT 1000
      Avg Time Taken: 0.000226223
      Avg # of Comparisons: 11895
      Avg # of Swaps: 5250

      BUBBLE SORT 1000 - Sorted
      Avg Time Taken: 0.00167457
      Avg # of Comparisons: 499500
      Avg # of Swaps: 0

      QUICK SORT 1000 - Sorted
      Avg Time Taken: 0.00195759
      Avg # of Comparisons: 499500
      Avg # of Swaps: 999
```

Option 2 - Sort 10000:
```
      BUBBLE SORT 10000
      Avg Time Taken: 0.766858
      Avg # of Comparisons: 49995000
      Avg # of Swaps: 24966181

      QUICK SORT 10000
      Avg Time Taken: 0.00266029
      Avg # of Comparisons: 156401
      Avg # of Swaps: 75785

      BUBBLE SORT 10000 - Sorted
      Avg Time Taken: 0.148925
```

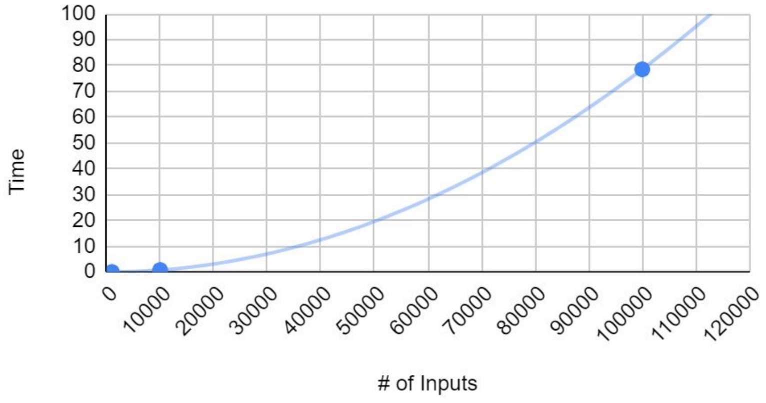
```
Avg # of Comparisons: 49995000
Avg # of Swaps: 0

QUICK SORT 10000 - Sorted
Avg Time Taken: 0.141467
Avg # of Comparisons: 49765133
Avg # of Swaps: 9950
```

Option 3 - Sort 100000

```
BUBBLE SORT 100000
Avg Time Taken: 78.519
Avg # of Comparisons: 4999950000
Avg # of Swaps: 2501177405

QUICK SORT 100000
Avg Time Taken: 0.0339013
Avg # of Comparisons: 2040812
Avg # of Swaps: 988614

BUBBLE SORT 100000 - Sorted
Avg Time Taken: 15.0268
Avg # of Comparisons: 4999950000
Avg # of Swaps: 0

QUICK SORT 100000 - Sorted
Avg Time Taken: 13.5372
Avg # of Comparisons: 4756797956
Avg # of Swaps: 95293
```

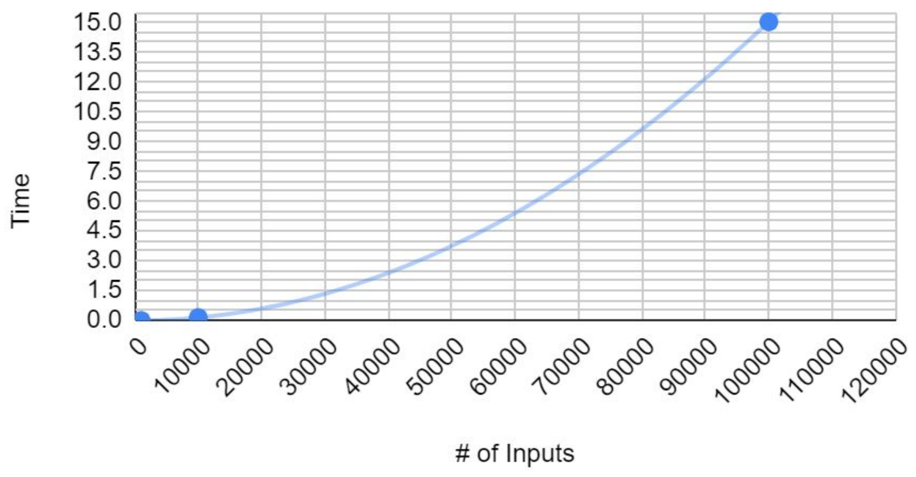
Bubble Sort Unsorted	
# of Inputs	Time
1000	0.00886213
10000	0.766858
100000	78.519

Bubble Sort Unsorted

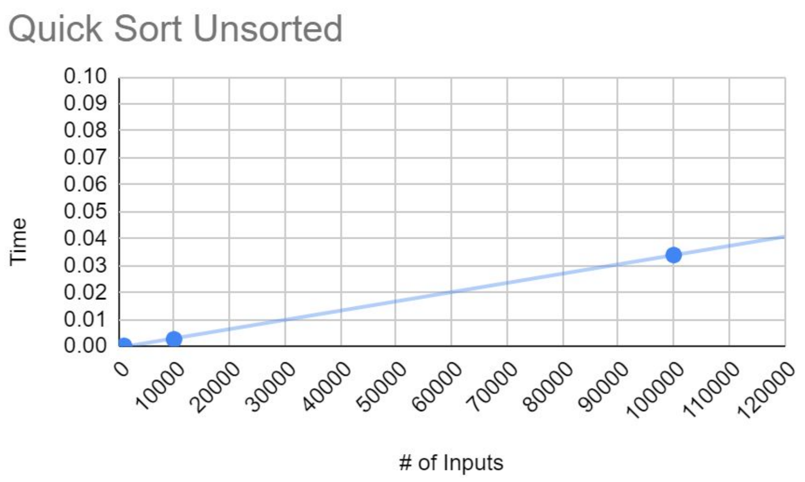


Bubble Sort Sorted	
# of Inputs	Time
1000	0.00167457
10000	0.148925
100000	15.0268

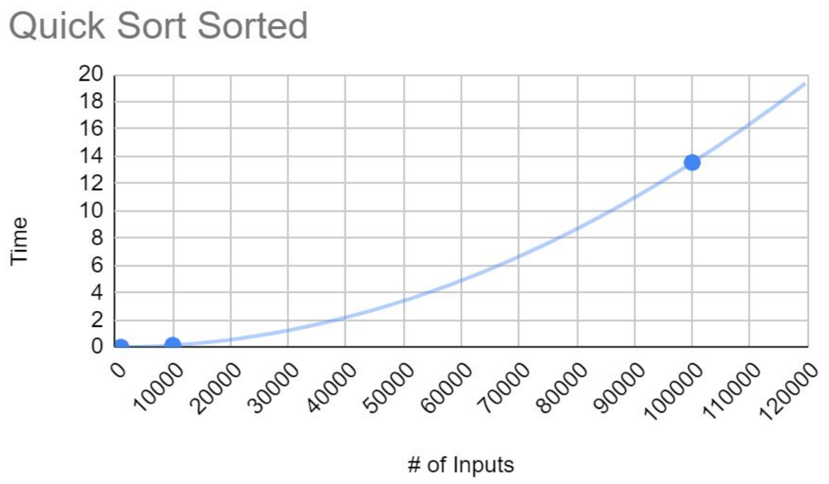
Bubble Sort Sorted



Quick Sort Unsorted	
# of Inputs	Time
1000	0.000226223
10000	0.00266029
100000	0.0339013



Quick Sort Sorted	
# of Inputs	Time
1000	0.00195759
10000	0.141467
100000	13.5372



The average case of bubble sort is O(n^2). The worst case of bubble sort is O(n^2).
The average case of quick sort is O(nlog(n)) while the worst case of quick sort is O(n^2).

Since bubble sort is O(n^2), the amount of time taken to sort should be 100 times more as the amount of inputs increases by 10 times. The graph of the unsorted bubble sort shows a trendline that is polynomial. For 1000 values, the time taken is 0.00886213 and for 10000 values, the time taken is 0.766858. The time taken for 10000 values is approximately 100 times the time taken for 1000 values as the 10000 values time is two decimal places to the left. Similarly, this is seen when the bubble sort is used on an already sorted file. The graph for an already sorted bubble sort is polynomial and fits the time complexity of a graph of n^2. However, sorting an already sorted file is significantly faster than sorting an unsorted file for bubble sort as there is no need to swap in an already sorted file.

Quick sort is significantly faster than bubble sorting. Its average case is O(nlog(n)) while its worst case is O(n^2). The graph of the unsorted quick sort shows a trendline that is fairly linear. As the number of inputs increases by 10 times, the time taken to sort also increases by 10 times. This can be seen as for 1000 values, the time is 0.000226223 and for 10000 values, the time is 0.00266029. The time for 10000 values can be seen as close to 10 times more than the time for 1000 values because the decimal place moves one place to the left. The graph of a nlog(n) initially dips before being a linear line so this trendline matches with the time complexity of O(nlog(n)). The time greatly increases in its worst case which is when it is sorting an already sorted array. However, it does not take as long as a bubble sort. It has an O(n^2) time complexity and this is reflected in the graph. It has a polynomial trendline and as the number of inputs increases by ten times, the time increases by approximately 100 times. For 10000 values, the time taken is 0.141467 and for 100000 values, the time taken is 13.5372. The time taken for 100000 values is approximately 100 times the time taken for 10000 values as the 1000 values time is two decimal places to the left. This fits with the graph of n^2.

The design of computationally efficient solutions is important because it affects time and resources. If a solution is not very efficient, this leads to a large waste of time and resources that could be used in other projects. As people say, “Time is money”. The inefficient solution would take longer and cost more due to higher resource usage. Bubble sorting for example is very slow and costly compared to quick sorting. Where it takes nearly 80 seconds to sort 100000 values, it takes 0.03 seconds with quick sort. In the bubble sort, it swaps billions of times which is very costly and time consuming but with quick sort, it swaps less than a million times. Bubble sort results in a sorted array but it is much slower than using quick sort. Creating more efficient algorithms and solutions is aligned with a sustainable future as it will use less resources and social efforts to produce similar results.