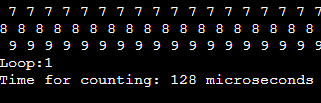
**Part 1: Comparison of Sorting Algorithms**

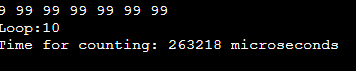
**1. Theoretical Question**

Assume the input elements are integers in the range [0, n-1]. For each sorting algorithm, determine the best, average, and worst-case inputs theoretically. (DO NOT refer to experimental results yet) Include a sentence or two of justification for each algorithm.

**Algorithms:**

1. **Insertion**
   1. The **best case** for insertion sort occurs when the entire array is already sorted. The loop would only run once and see that everything is sorted so the time complexity would be O(n). The **average case** for insertion sort occurs when the array is in no particular order. The time complexity is O(n^2) because on average n^2 inversions are expected. The **worst case** for insertion sort occurs when the array is sorted in the reverse order. There would n - 1 comparisons and swaps made which would lead to time complexity of O(n^2).
2. **Selection** 
   1. **Best case** for selection sort is when inputs are already sorted. The swap for each comparison is avoided so time complexity is O(n^2). **Worst case** is when the array is in reverse sorted array. Every element will have to be compared with another element, making it a lengthy process. Time complexity is O(n^2). **Average case** israndom input. The number of comparisons will be about the same for each case resulting in a time complexity of O(n^2).
3. **Bubble** 
   1. The **best case** for bubble sort occurs when the entire array is already sorted. The time complexity is O(n) because n - 1 comparisons would be made. The **average case** occurs when elements are in a random order. The time complexity is O(n^2) because on average there would be (n/2) swaps and n passes. The **worst case** occurs when an array is reverse sorted. The time complexity is O(n^2) because n number of comparisons and passes are made.
4. **Merge sort**
   1. Merge sort divides the array into two equal halves, calls itself for the two halves, and then merges back together to create a sorted array. This results in the best, average, and worst cases being O(nlogn). Merge sort uses the divide and conquer strategy to sort an array. The **best case** occurs when inputs are already sorted O(nlogn). **Worst case** occurs when the left and right array have alternating elements resulting in the maximum number of comparisons. When elements are required to be sorted in reverse order O(nlogn). **Average case** occurs when the array elements are not in proper ascending or descending order O(nlogn).
5. **Quicksort**
6. Quicksort is another sorting algorithm like merge sort that uses the divide and conquer strategy. Unlike merge sort, quick sort uses a pivot element to partition the elements which results in different time complexities. **Best case** occurs whenthe partition process picks the middle element or near the middle as pivot O(nLogn). **Worst case** occurs when the partition process always picks the greatest or smallest element as the pivot O(n^2). **Average case** occurswhen array elements are not in an increasing or decreasing sequence O(nLogn).
7. **Heapsort**
   1. Heapsort’s theoretical time complexity is O(nlogn) in the best, average, and worst cases. The **best case** input is when the array is sorted in descending order because the array is already a max-heap and there is no need to swap elements and recursively max-heapify to build the heap. The **worst case** input is when the array is sorted in ascending order because the array must be completely reversed to build the heap. The **average case** input is when the array elements are random and not in any particular order. In every input case, heapsort still has to swap the first and last elements and to max-heapify after removing each element.
8. **Counting Sort**
   1. **Best Case** for counting sort is whenvalues are similar in the same range. Keeping track of the occurrences along with finding the index of each value results in linear time, so time complexity is O(n). **Worst case** is when the data is skewed. Skewed data contains outliers in the dataset which makes the range very large, resulting in a time complexity of O(k). **Average case** is when the number and range of elements are equally dominant. The range of inserted elements are not greater than the values that need to be sorted, so time complexity is O(n+k).
9. **Radix Sort**
10. **Best case:** *If all values in the array are single digit values(0-9), radix sort can sort all numbers in the first loop. Below is an example of an array of size 1000 with values between 0-9, as compared to an array with values between 0-99:*  **O(n\*d)**

****

****

1. **Worst case:** *If all values in the array are equal in number of digit values with an outlier in the array, it will cause excess looping. For example, given an array of 1,000 values with 999 values equal to 42 and one value being 1,000,000, this will cause excess looping because of how the auxiliary function works-* **O(n^2)**
2. **Average case:***Average time for a radix sort array with no particular pattern-* **O(d\*(n+b))**

**2. Data generation and experimental setup**

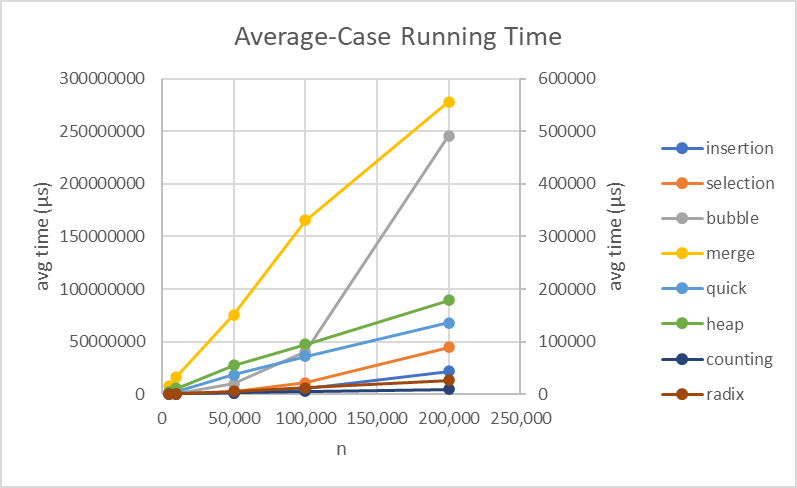
* **What kind of machine did you use?**
* **What timing mechanism?**
* **How many times did you repeat each experiment?**
* **What times are reported?**
* **How did you select the inputs?**
* **Did you use the same inputs for all sorting algorithms?**

The experiment was performed on a laptop with an Intel Core i7-8550U CPU using the Code::Blocks IDE. The experiments were timed in microseconds using the C++ chrono library steady\_clock. Each experiment was repeated 5 times and then the average of those times was reported. The input sizes were 5,000, 10,000, 50,000, 100,000 and 200,000 for all the sorting algorithms. The smallest input size was chosen so that the algorithms with the lowest time complexities run it in greater than 1 microsecond. The largest input size was chosen so that the highest time complexities can run it and finish in a reasonable amount of time.

**3. Which of the five sorts seems to perform the best (consider the best version of Quicksort)?**

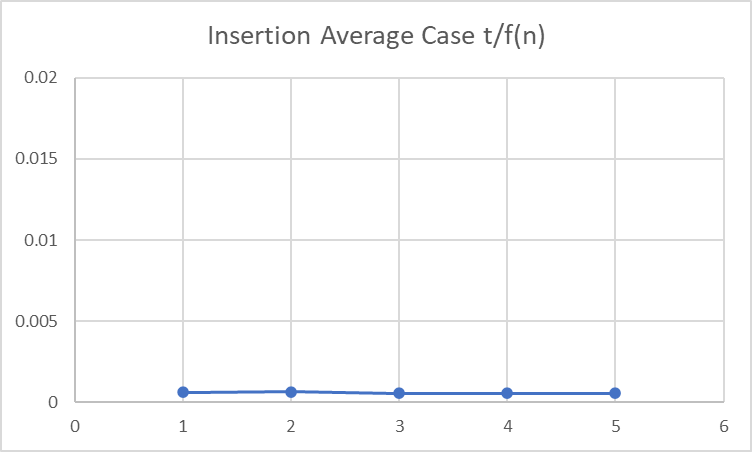
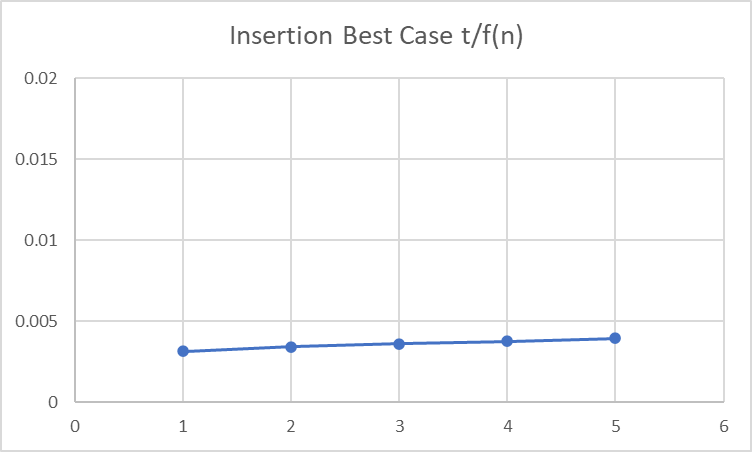
Because there is a large difference in the times between some of the algorithms, the algorithms with O(n^2) time complexity were graphed on the primary vertical axis and the rest were graphed on the secondary vertical axis.

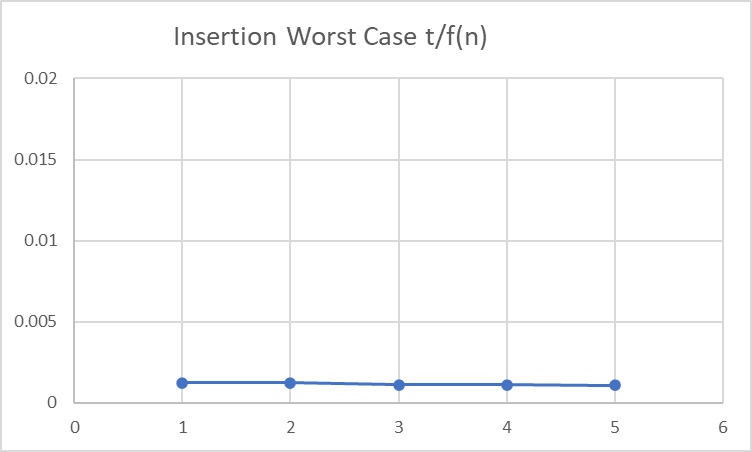
In our experiment, counting sort seems to perform the best.

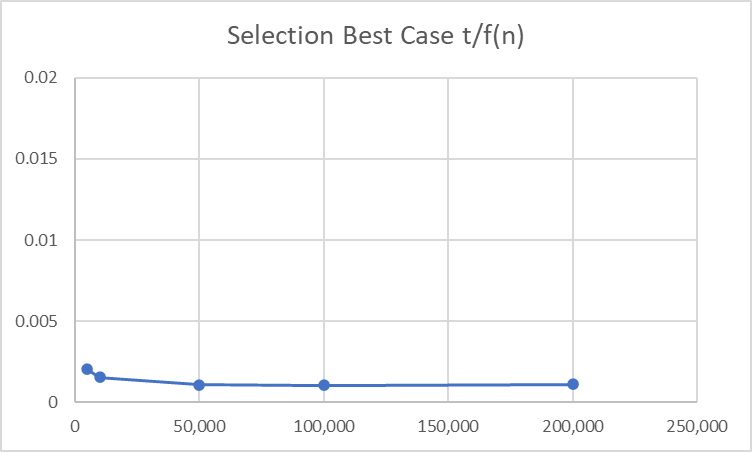
****

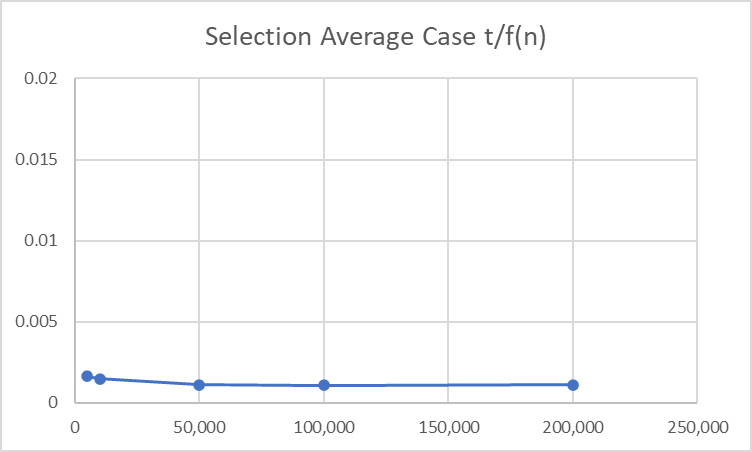
**4. To what extent do the best, average and worst-case analyses (from class/textbook) of each sort agree with the experimental results?**

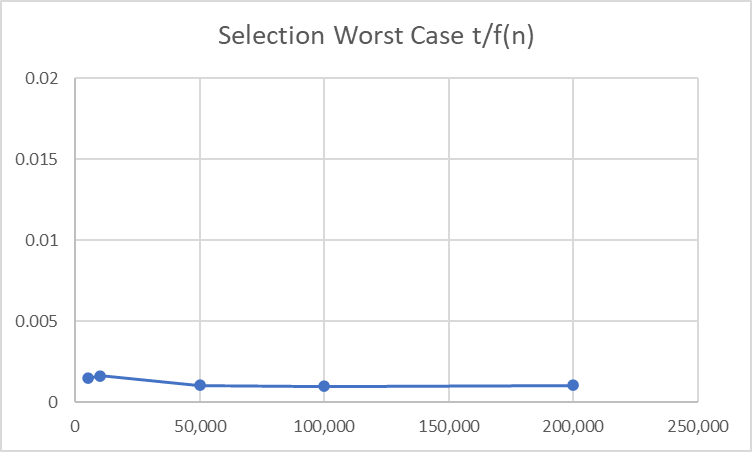
To determine whether the predicted times of O(f(n)) agree with the experimental results, we divided each of the average times by f(n) to see if the graph forms a horizontal line. For almost every algorithm, the experimental results were fairly close to the best, average and worst case analyses. The exception to this was bubble sort whose best case did not match with its theoretical time complexity. The reason the time complexity did not match may be because the algorithm was implemented with a nested loop, going through both loops may have caused the experimental runs to take longer than what was theorized.

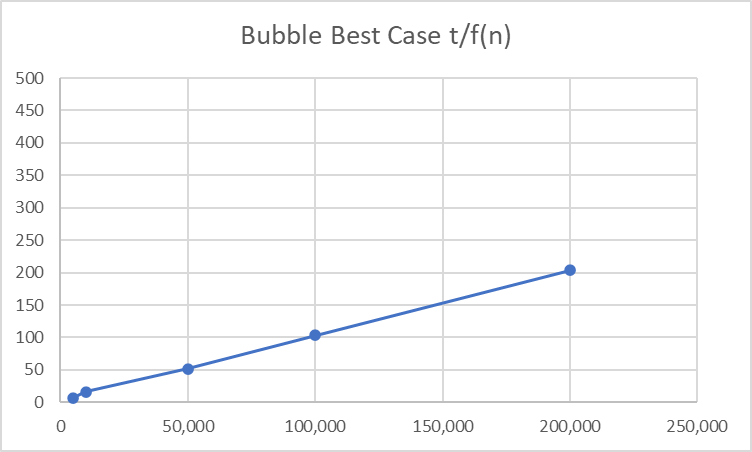
****

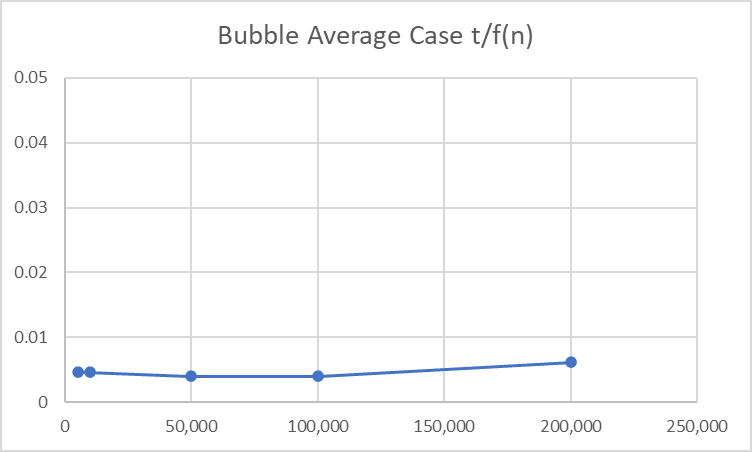
****

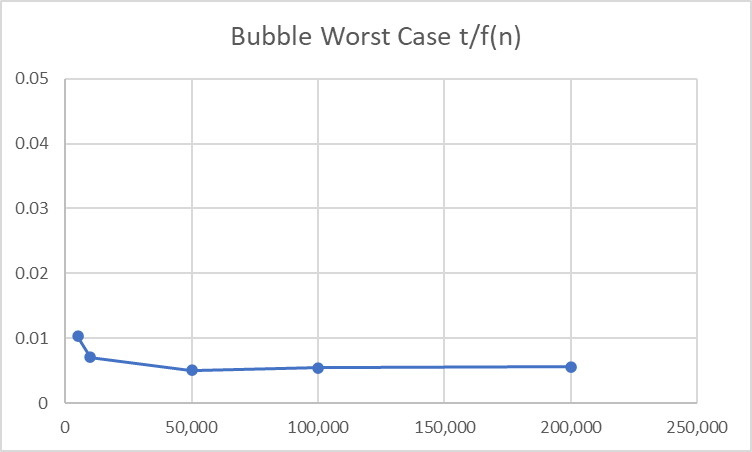
****

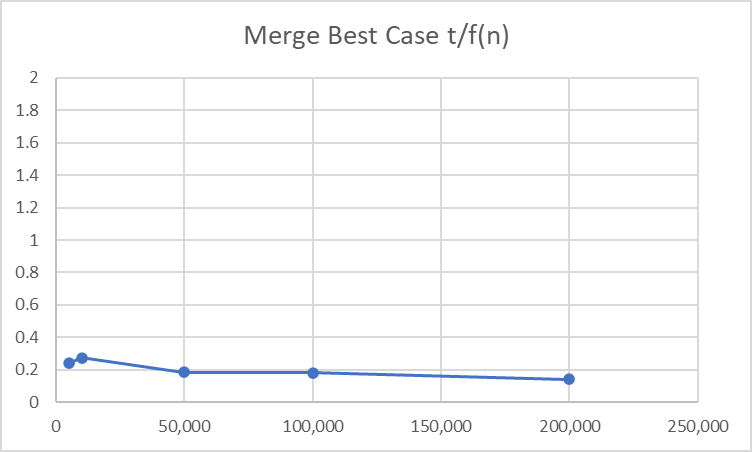
****

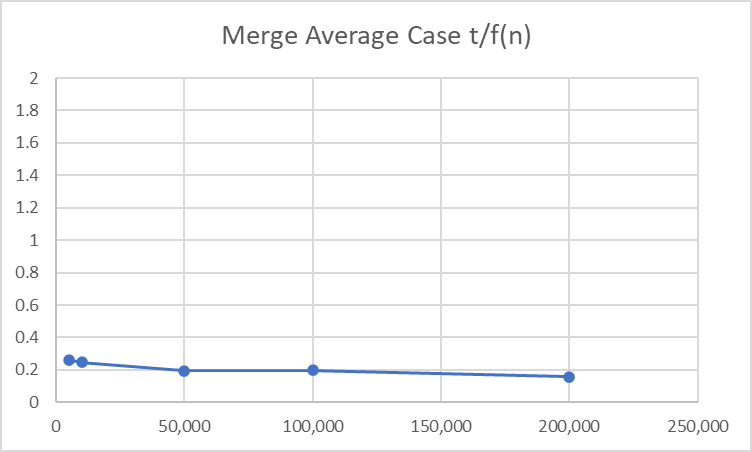
****

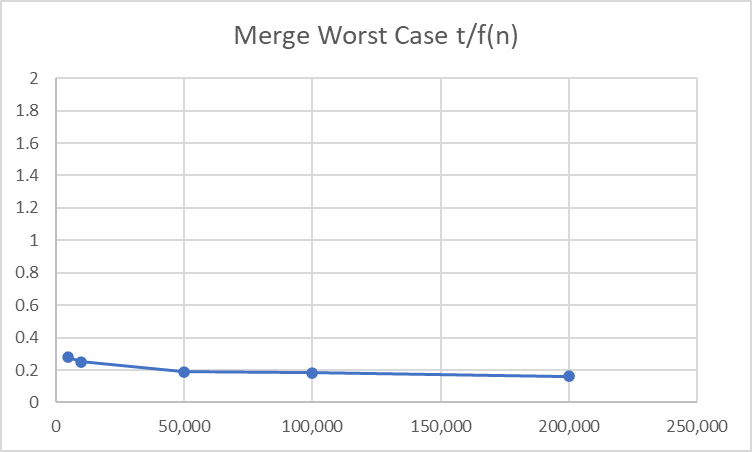
****

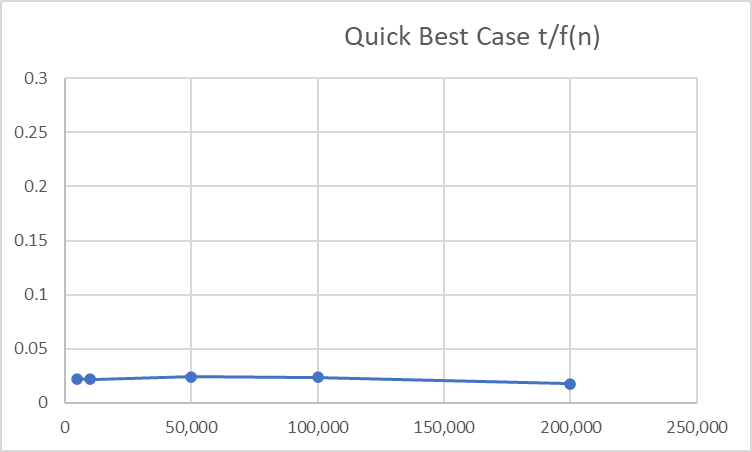
****

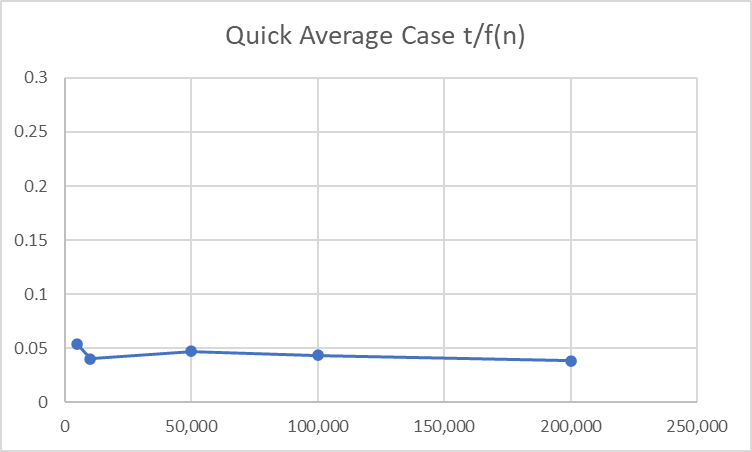
****

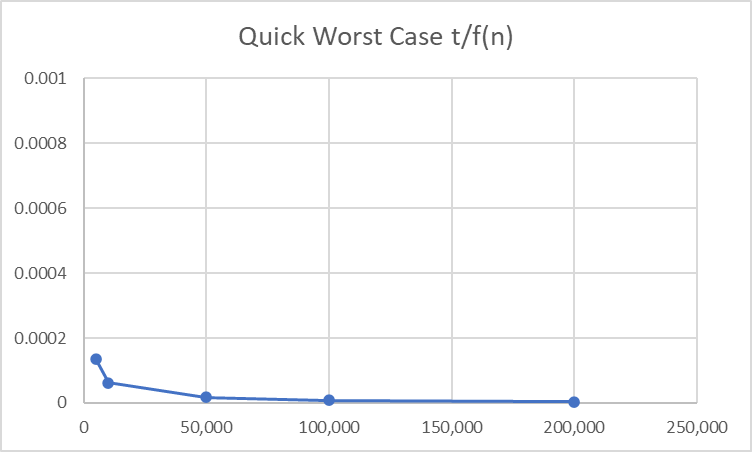
****

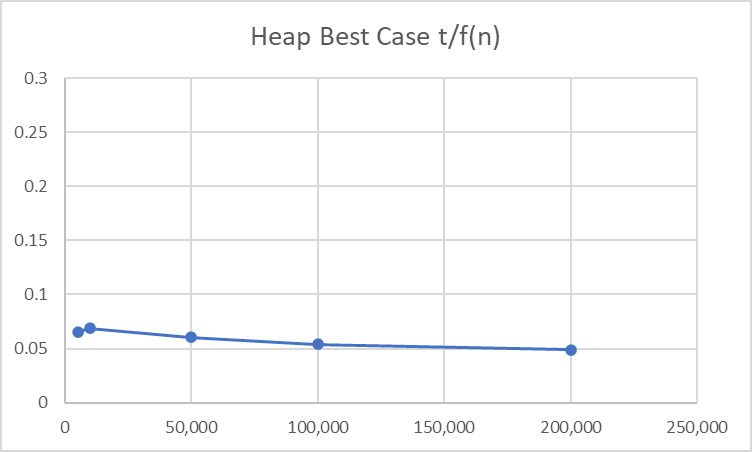
****

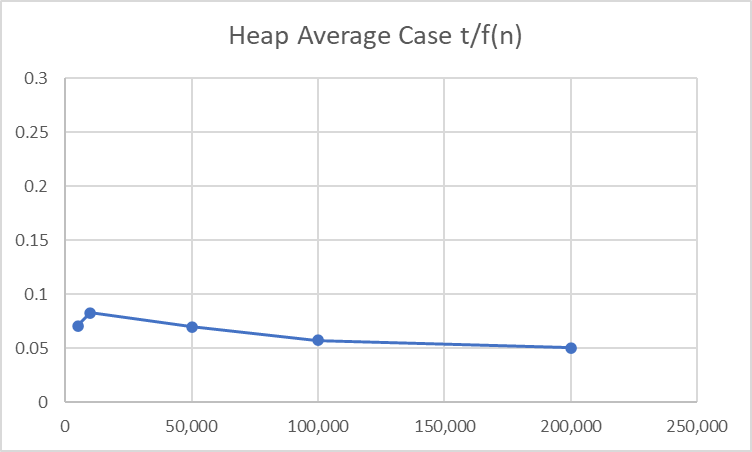
****

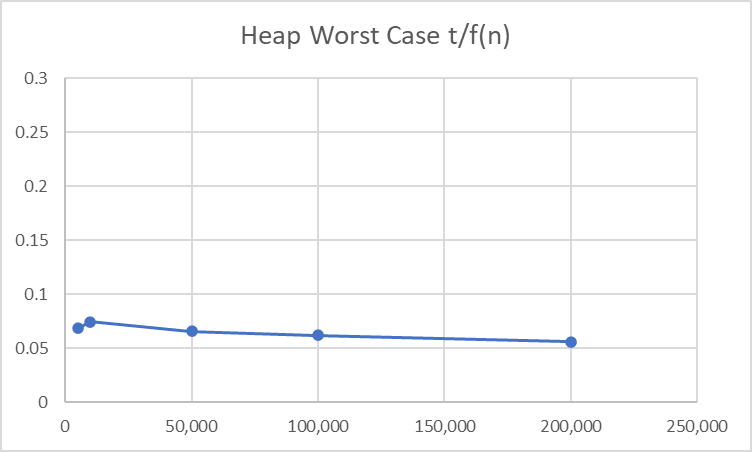
****

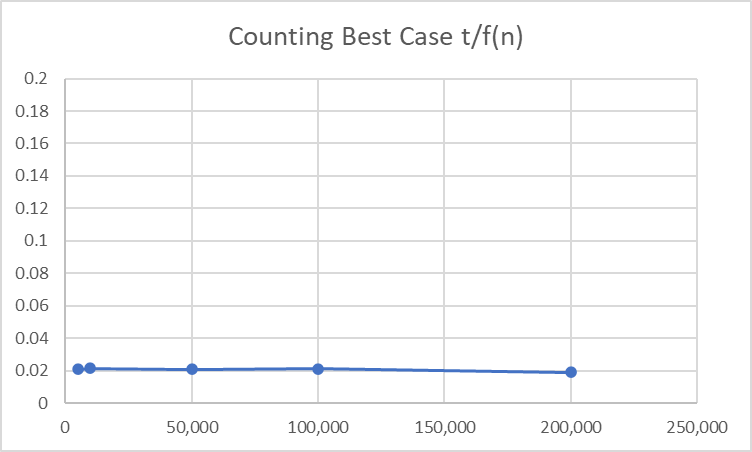
****

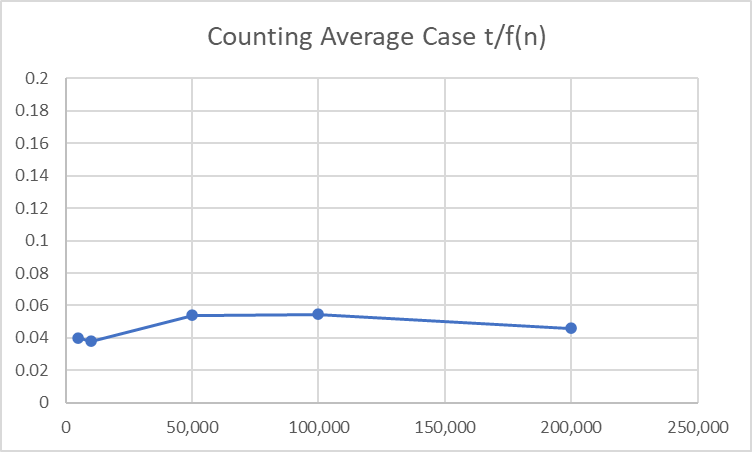
****

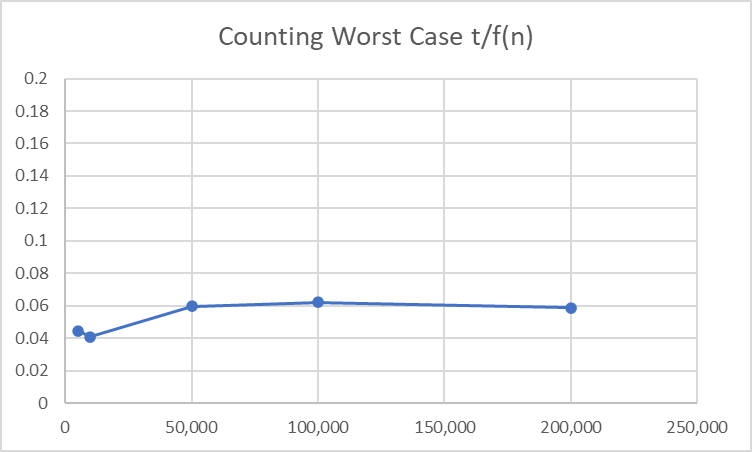
****

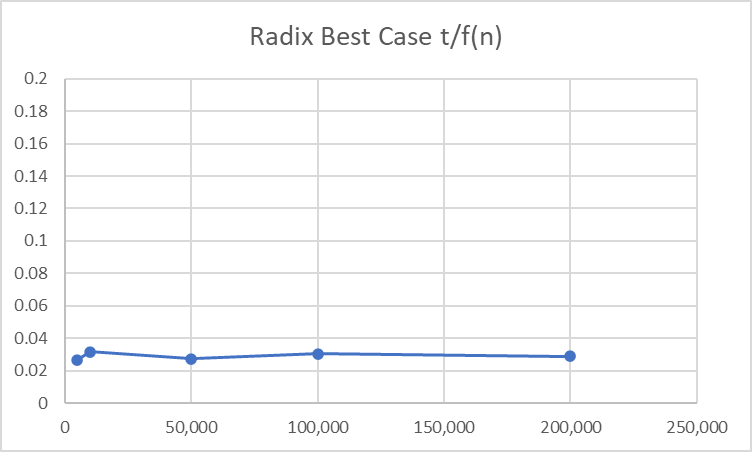
****

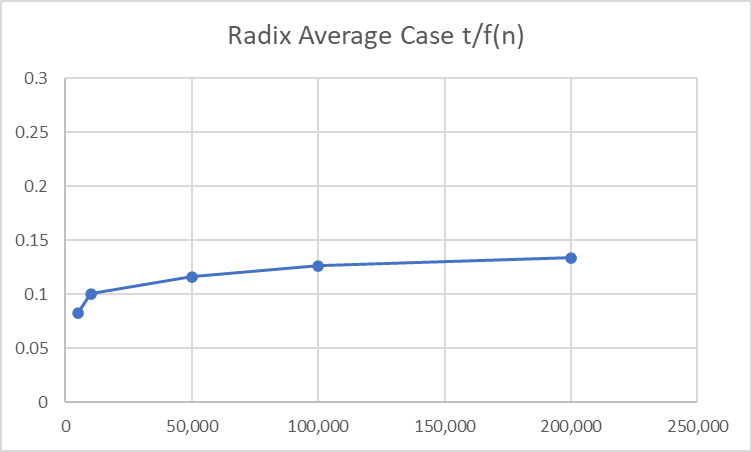
****

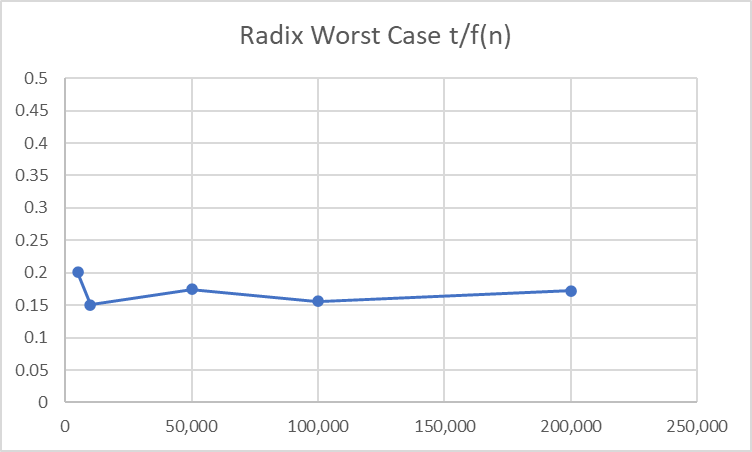
****

****

****

****

****

****

**5. For the comparison sorts, is the number of comparisons really a good predictor of the execution time? In other words, is a comparison a good choice of basic operation for analyzing these algorithms?**

The number of comparisons is a good predictor of the execution time. This is because the execution increases or decreases based on the run time of the sorting algorithm. The number of comparisons compared to our execution time in our charts shows that the bigger the number of comparisons/elements the same or greater the execution time.

**Part 2: Problem Solving and Analysis**

**1)**

BruteForceSum(S, x)

for *i* = 0 to *S.length*

for *j* = i + 1 to *S.length*

if *S*[*i*] + *S*[*j*] = *x*

return true

return false

The brute-force method checks all possible pairs of numbers. The inner loop executes *S.length* times for each element of *S*, so that every element is checked against every other element to see if it equals the sum. It has a time complexity of O(n^2) because the outer loop runs *S.length* times and the inner loop runs *S.length*-1 times.

**2)**

This efficient algorithm has a time complexity of O(nlogn) because it utilizes merge sort as part of the algorithm. It has a better time complexity than O(n^2). The while loop has a time complexity of O(n), so adding O(nlogn) with O(n) results in O(nlogn). In this algorithm, two pointers are implemented. The first (low) element and the last (high) element are pointed at. Two values are added together and checked. If the total is greater than value, the right pointer is then shifted to the left. If the total is less than value, the left pointer is shifted to the right. The two values that equal the total will be returned once the values are checked.

eff(array, value, size)

MergeSort(array)

low <- 0

high <- size - 1

while (low < high)

total <- array[low] + array[high]

If total == value

return 1

else if total > value

high--

else

low++

return 0