MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING

CSE 246
Analysis of Algorithms

Homework 2

Hale ŞAHİN -150116841

STEP 1:

For the step 1, I generated inputs of sized 10,100 and 1000. For each input group there is sorted, reverse ordered and randomly come inputs. For the best case of each sorting algorithm required ordered input array. For the worst case they are required reverse ordered. That is why I select these inputs. My time metrics will be seconds. My CPU is so fast, I can't get reasonable results with clock() function. I used real time. My computer runs 2601 MHz speed. That is mean it processes 2601,000,000 process in one second. So, I will calculate theoretical results according to this information.

1 sec 2601,000,000 process

X

X sec 10 or 100 or 1000 or 100000 process etc.

X = 10/2601000000 for input size 10 and if T(n)=n

STEP 2:

I implemented algorithms in C programming language with separate classes. I gave them the inputs I have decided on step 1 and get the output of time in seconds which was the comparison metric I have decided on step 1. Then I run the algorithms and get the experimental results for each input.

STEP 3:

1. Insertion Sort Algorithm

Insertion sorting algorithm gives minimum time complexity O(n) with the best case, and maximum time complexity with the O(n^2) with the worst case. Best case is input come ordered already. Worst case input is reverse ordered.

Best Case Example List:

0-6-11-22-25-27-28-31-39-45-57-63-78-89-92-100

Worst Case Example List:

100-92-89-78-63-57-45-39-31-28-27-25-22-11-6-0

Average Case Example List:

25-11-57-63-100-92-6-45-78-89-22-0-31-27-39-45-28

❖ THEORETICAL RESULTS

In best case: T(n) = n

- n=10, T(n) = 10/2601,000,000 = 3,844e^{-9} sec
- n=100, T(n) = 100/2601,000,000 = 3,844e^{-8} sec
- n=1000, T(n)= 1000/2601,000,000 = 3,844e^{-7} sec

In worst case: T(n) = n^2

- n=10, T(n) = 100/2601,000,000 = 3,844e^{-8} sec
- n=100, T(n) = 10000/2601,000,000 = 3,844e^{-6} sec
- n=1000, T(n) = 1000000/2601,000,000 = 3,844e^{-4} sec

In average case: T(n) = n^2

- n=10, T(n)= 100/2601,000,000 = 3,844e^{-8} sec
- n=100, T(n) = 10000/2601,000,000 = 3,844e^{-6} sec
- n=1000, T(n) = 1000000/2601,000,000 = 3,844e^{-4} sec

❖ EXPERIMENTAL RESULTS

In Best Case: T(n)=n

- n=10, T(n)= 3,948e^{-9} sec
- n=100, T(n)= 1,5790574e^{-7} sec
- n=1000, T(n)= 4,7369394e^{-7} sec

In Worst Case: T(n)=n^2

- n=10, T(n)= 3,939494e^{-8} sec
- n=100, T(n)= 2,40798108e^{-6} sec
- n=1000, T(n)= 5,6438520551e^{-4} sec

In Average Case: T(n)= n^2

- n=10, T(n)= 3,951136e^{-8} sec
- n=100, T(n)= 1,34219881e^{-6} sec
- n=1000, T(n)= 1,1495349463e^{-4} sec

❖ CONCLUSION

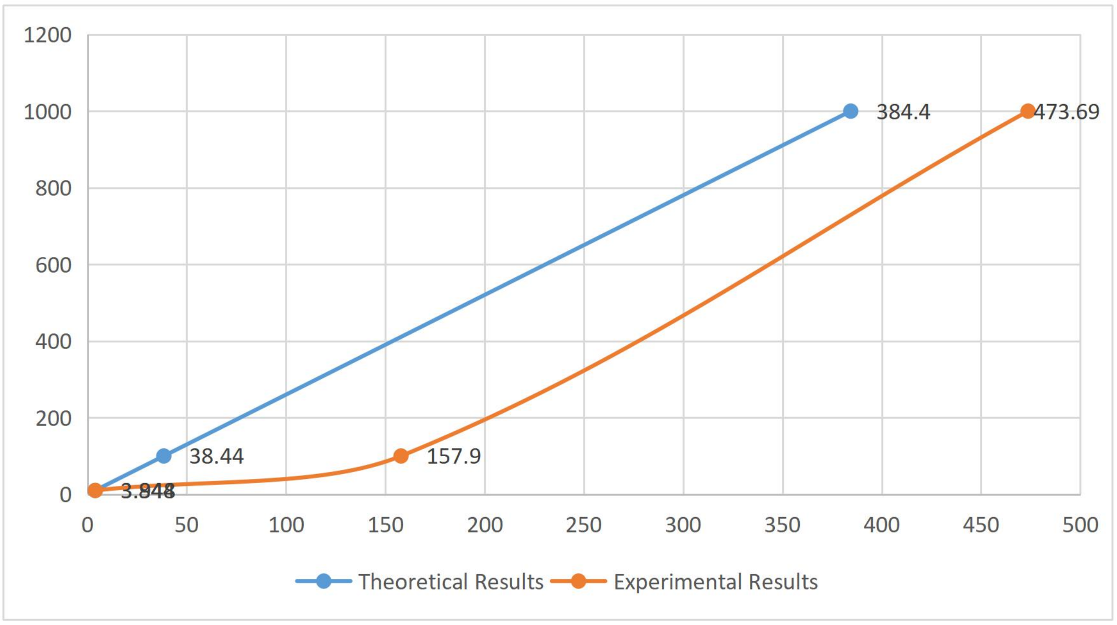

Figure 1. Best Case Results for Insertion Sort in nanoseconds

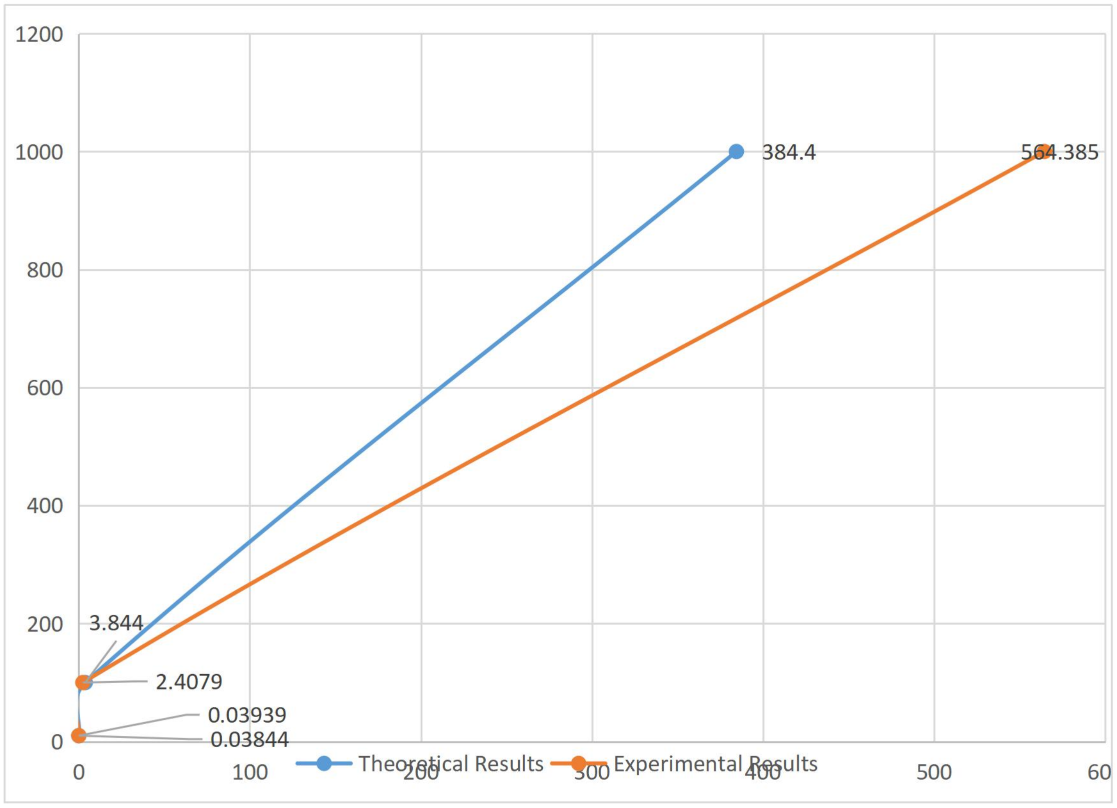

Figure 2.Worst Case Results for Insertion Sort in microseconds

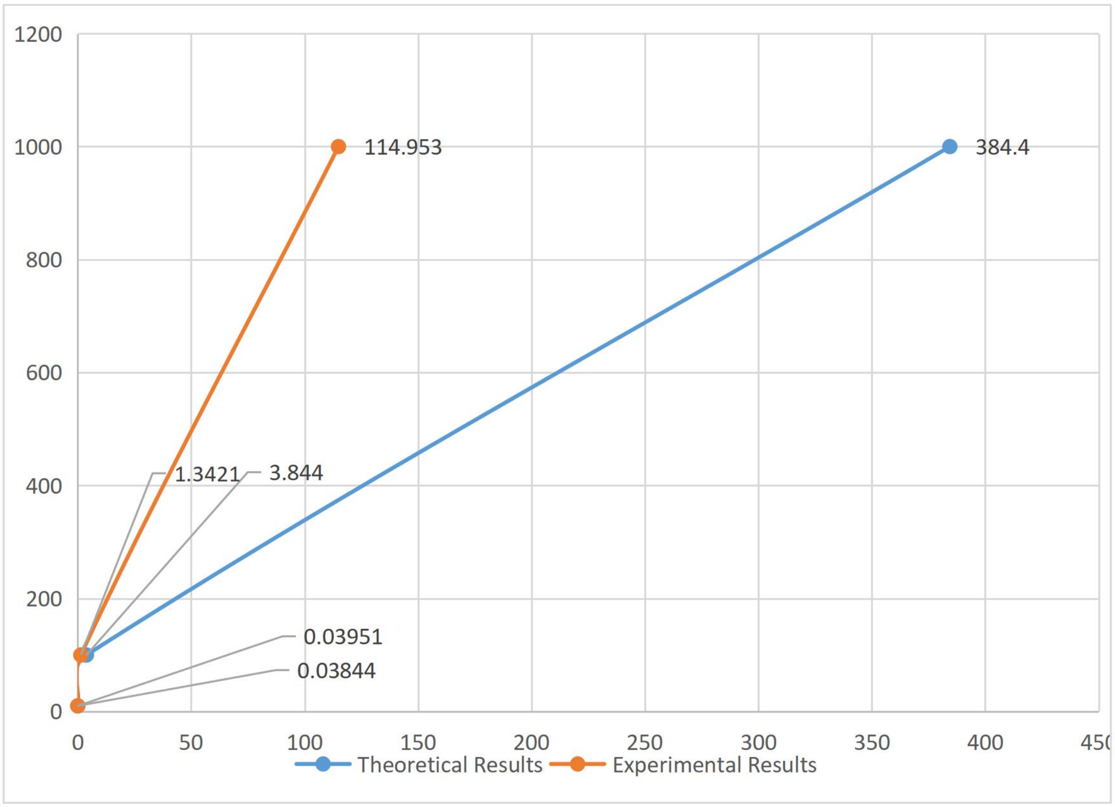

Figure 3.Average Case Results for Insertion Sort in microseconds

Results have shown that, when input size is smaller, theoretical results nearly same with the experimental results. When input size getting larger difference get increased with it.

2.Merge Sort Algorithm

Merge Sort algorithm time complexity of is exactly ϴ(nlogn) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.But it uses extra memory.

Best case Example List:

1-3-7-9-13-16-21-23-25-29-30-33-39-48-65-100

Worst case Example List:

1-25-13-29-7-30-21-65-3-29-16-48-9-33-23-100

Average case Example List:

23-100-39-3-1-25-9-48-21-65-33-7-13-29-30-16

STEP 3:

❖ **THEORETICAL RESULTS**

In best case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= 1.277e^{-8} sec
- n=100, T(n) = 664.38/2601,000,000= 2.554e^{-7} sec
- n=1000, T(n) = 9965.78/2601,000,000= 3.831e^{-6} sec

In worst case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= 1.277e^{-8} sec
- n=100, T(n) = 664.38/2601,000,000= 2.554e^{-7} sec
- n=1000, T(n) = 9965.78/2601,000,000= 3.831e^{-6} sec

In average case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= 1.277e^{-8} sec
- n=100, T(n) = 664.38/2601,000,000= 2.554e^{-7} sec
- n=1000, T(n) = 9965.78/2601,000,000= 3.831e^{-6} sec

❖ **EXPERIMENTAL RESULTS**

In best case: $T(n) = n\log_2 n$

- n=10, T(n) = 1.579e^{-8} sec
- n=100, T(n) = 1.184e^{-7} sec
- n=1000, T(n) = 1.638e^{-6} sec

In worst case: $T(n) = n\log_2 n$

- n=10, T(n) = 1.578e^{-8} sec
- n=100, T(n) = 1.223e^{-7} sec
- n=1000, T(n) = 1.358e^{-6} sec

In average case: $T(n) = n\log_2 n$

- n=10, T(n) = 1.579e^{-8} sec
- n=100, T(n) = 1.381e^{-7} sec
- n=1000, T(n) = 1.460e^{-6} sec

❖ CONCLUSION

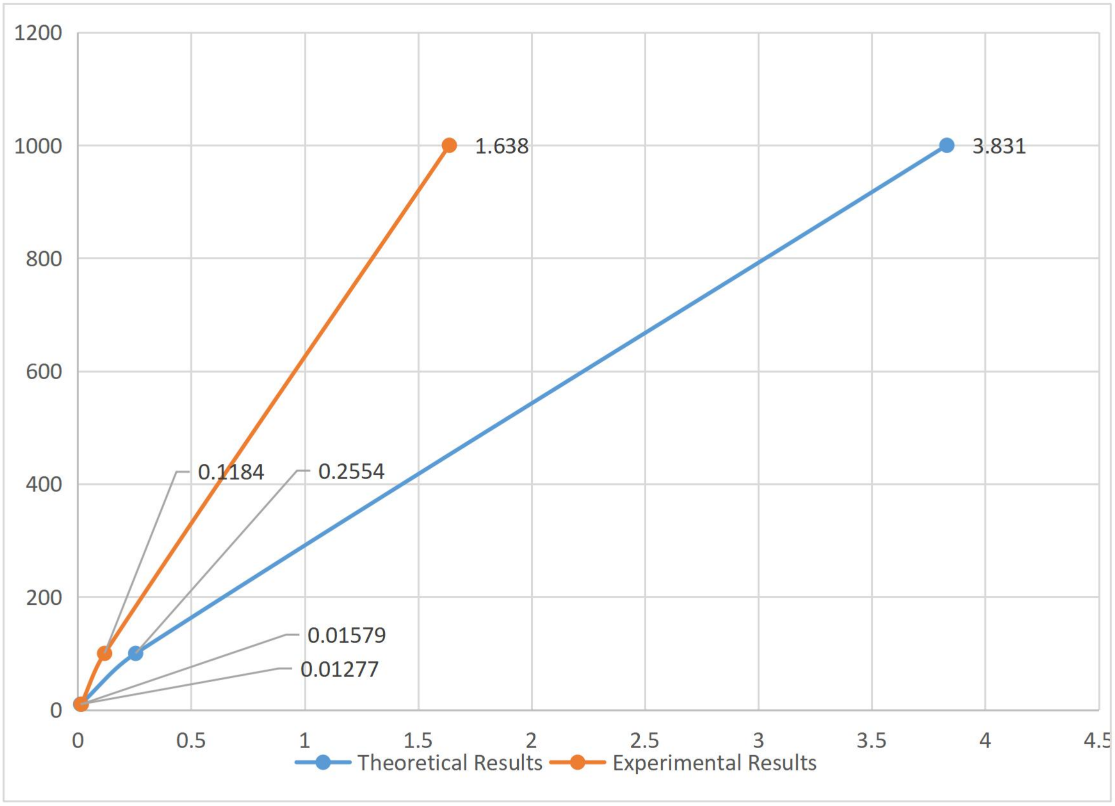

Figure1. Result comparison of merge sort in microseconds for the best case

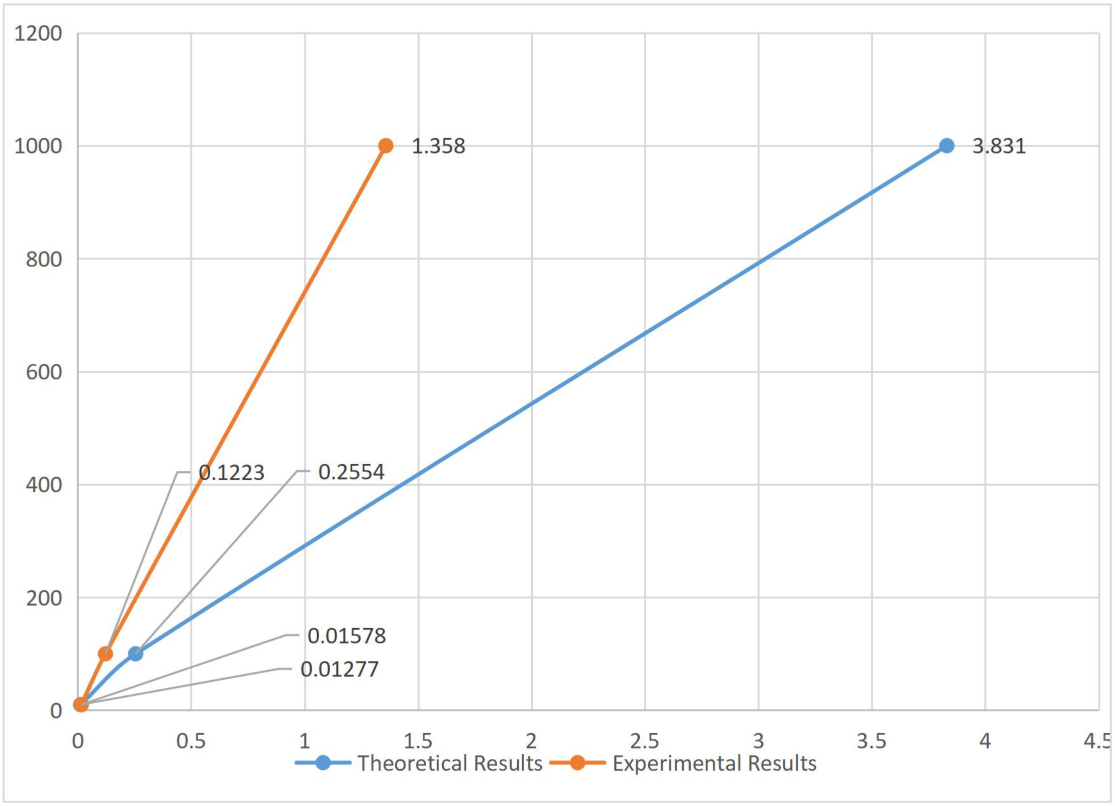

Figure2. Result comparison of merge sort in microseconds for the worst case

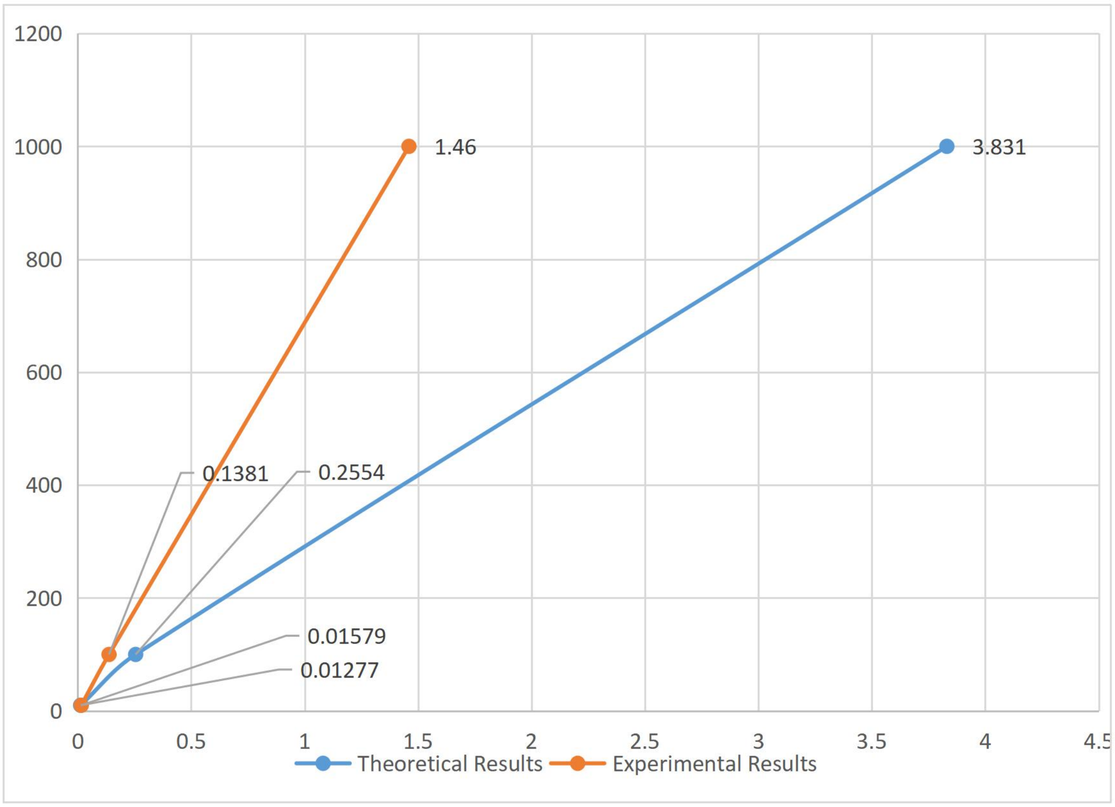

Figure3. Result comparison of merge sort in microseconds for the average case

When I examined the results, experimental and theoretical results are exactly same for the small input size but for the larger sized inputs difference increase for a reasonable amount of time. When I look results of merge sort compared with the insertion sort, there is definitely a big difference where merge sort is much better than insertion sort with all cases.

3.Quick Sort Algorithm

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

Always pick first element as pivot.

Always pick last element as pivot (implemented below)

Pick a random element as pivot.

Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. So I wrote two different quicksort function one with the pivot selecting middle element and one with the pivot selecting last element for the worst case. Best case time complexity is $\Theta(n\log n)$.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. The worst case would occur when the array is already sorted in increasing order and pivot is the last element which is greatest element and also reverse order input with the pivot is the last element which is the smallest element. Worst case time complexity is $\Theta(n^2)$.

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Average time complexity of quick sort is O(nlogn).

STEP 3:

❖ THEORETICAL RESULTS

In best case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= $1.277e^{-8}$ sec
- n=100, T(n) = 664.38/2601,000,000= $2.554e^{-7}$ sec
- n=1000, T(n) = 9965.78/2601,000,000= $3.831e^{-6}$ sec

In worst case: $T(n) = n^2$

- n=10, T(n) = 100/2601,000,000 = $3{,}844e^{-8}$ sec
- n=100, T(n) = 10000/2601,000,000 = $3{,}844e^{-6}$ sec

- n=1000, T(n) = 1000000/2601,000,000 = 3,844e^{-4} sec

In average case: T(n) = $n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= 1.277e^{-8} sec
- n=100, T(n) = 664.38/2601,000,000= 2.554e^{-7} sec
- n=1000, T(n) = 9965.78/2601,000,000= 3.831e^{-6} sec

❖ EXPERIMENTAL RESULTS

In best case: T(n) = $n\log_2 n$

- n=10, T(n) = 1.184e^{-8} sec
- n=100, T(n) = 9.475e^{-8} sec
- n=1000, T(n) = 2.684e^{-6} sec

In worst case: T(n) = n^2

- n=10, T(n) = 1.184e^{-8} sec
- n=100, T(n) = 3.67e^{-7} sec
- n=1000, T(n) = 2.797e^{-5} sec

In average case: T(n) = $n\log_2 n$

- n=10, T(n) = 1.185e^{-8} sec
- n=100, T(n) = 1.658e^{-7} sec
- n=1000, T(n) = 2.301e^{-6} sec

❖ CONCLUSION

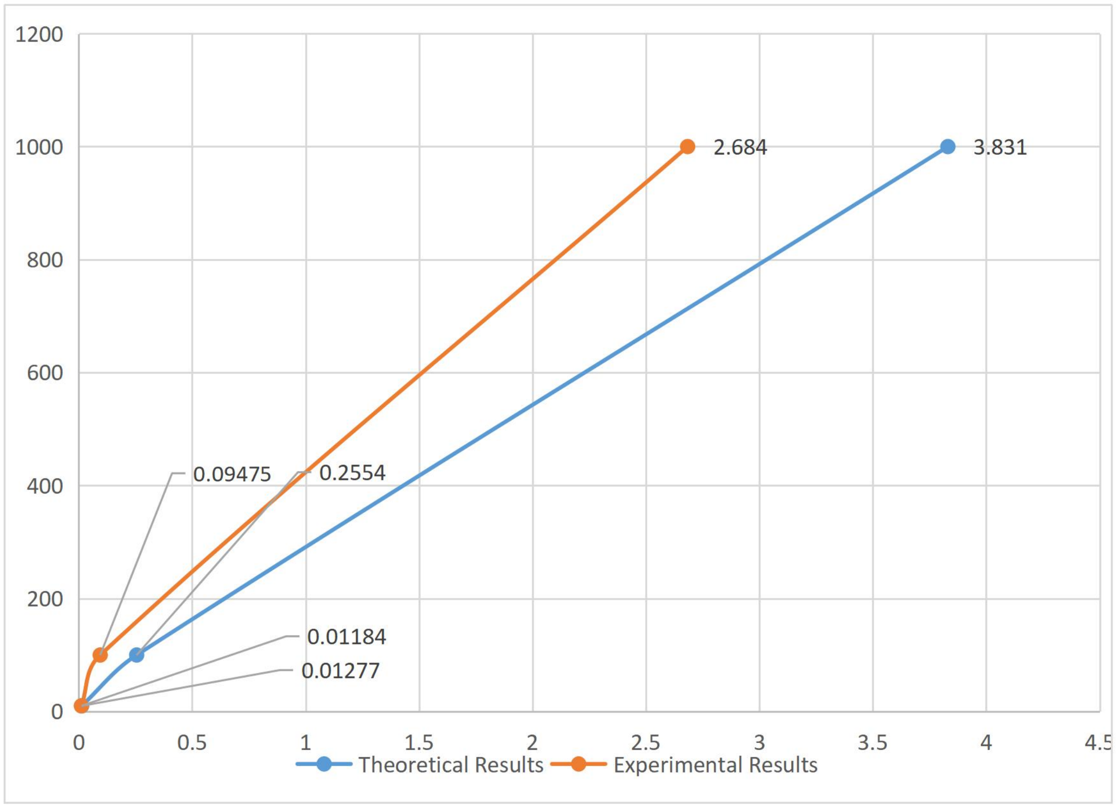

Figure1.Best case comparison for quick sort in microseconds

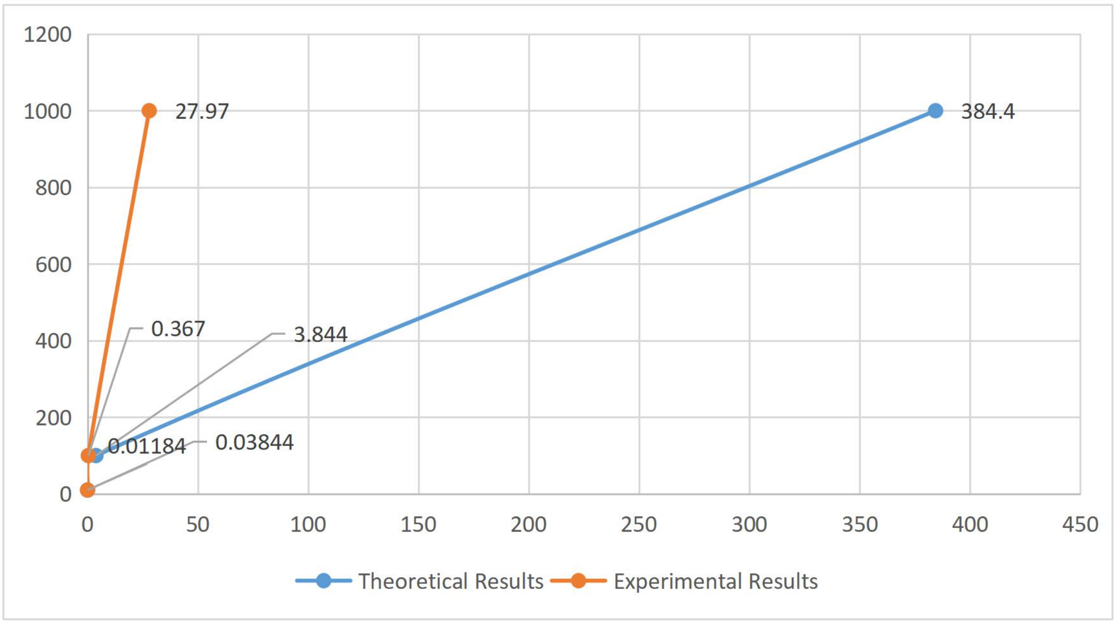

Figure2. Worst case comparison for quick sort in microseconds

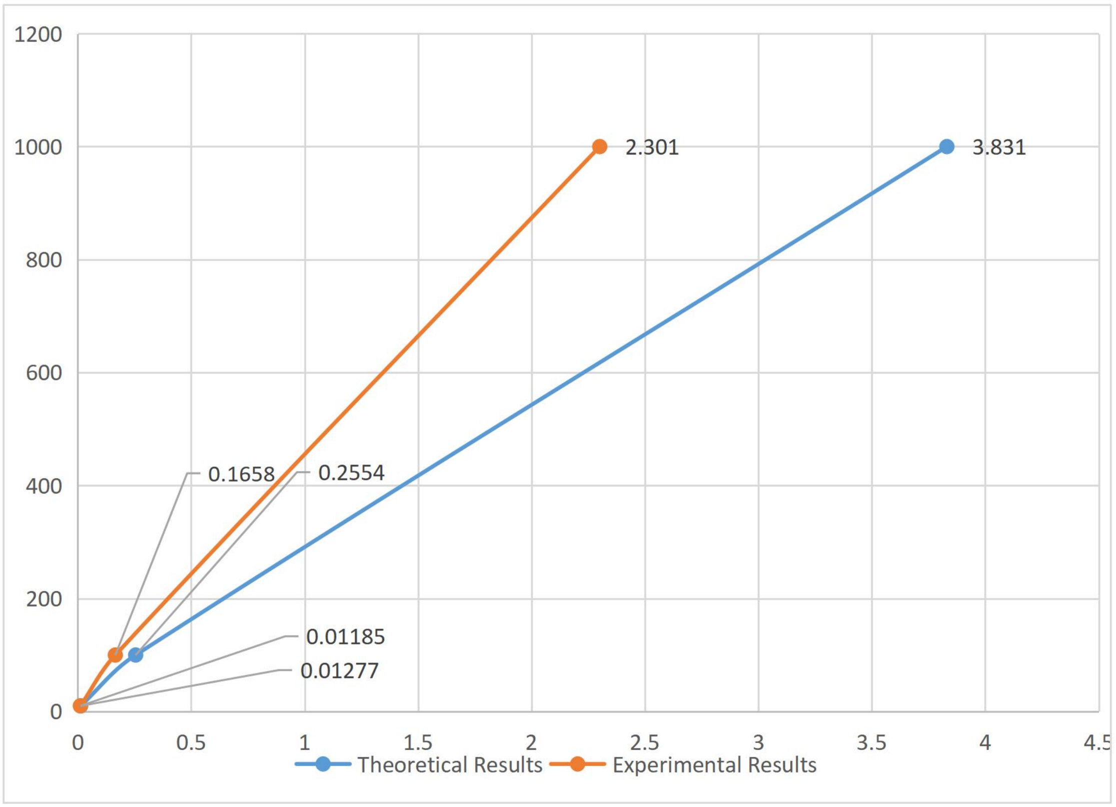

Figure3.Average case comparison for quick sort in microseconds

Although the worst case time complexity of QuickSort is O(n^2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

4.Heap Sort Algorithm

Heap Sort Algorithm for sorting in increasing order starts building a max heap from the input data. At this point, the largest item is stored at the root of the heap. Then it replaced by the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.Then repeating these steps while size of heap is greater than 1.

Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn). Heap sort algorithm that I implemented didn't work for the input which contains reverse ordered numbers. I designed algorithm to build a maxheap.

STEP 3:

❖ THEORETICAL RESULTS

In best case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= $1.277e^{-8}$ sec
- n=100, T(n) = 664.38/2601,000,000= $2.554e^{-7}$ sec
- n=1000, T(n) = 9965.78/2601,000,000= $3.831e^{-6}$ sec

In worst case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= $1.277e^{-8}$ sec
- n=100, T(n) = 664.38/2601,000,000= $2.554e^{-7}$ sec
- n=1000, T(n) = 9965.78/2601,000,000= $3.831e^{-6}$ sec

In average case: $T(n) = n\log_2 n$

- n=10, T(n) = 33.21/2601,000,000= $1.277e^{-8}$ sec
- n=100, T(n) = 664.38/2601,000,000= $2.554e^{-7}$ sec
- n=1000, T(n) = 9965.78/2601,000,000= $3.831e^{-6}$ sec

❖ EXPERIMENTAL RESULTS

In best case: $T(n) = n\log_2 n$

- n=10, T(n) = $1.184e^{-8}$ sec
- n=100, T(n) = $1.381e^{-7}$ sec
- n=1000, T(n) = $3.568e^{-6}$ sec

In average case: $T(n) = n\log_2 n$

- n=10, T(n) = $1.579e^{-8}$ sec
- n=100, T(n) = $1.500e^{-7}$ sec
- n=1000, T(n) = $2.096e^{-6}$ sec

❖ CONCLUSION

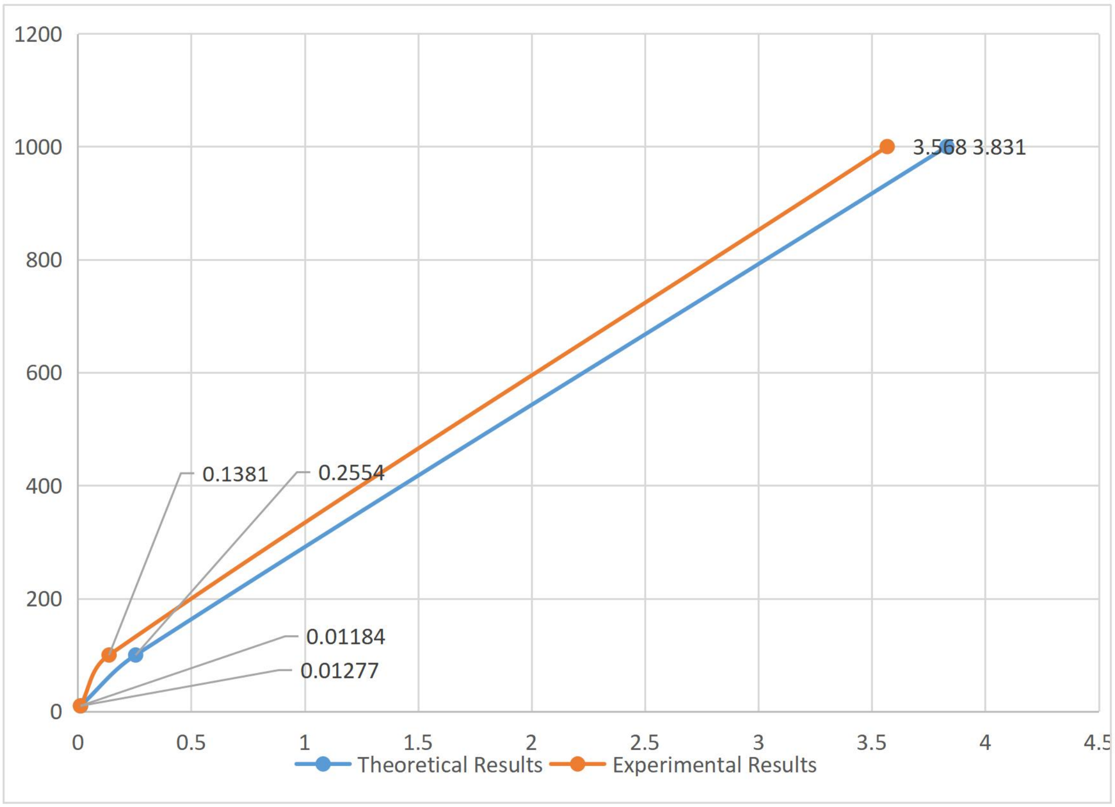

Figure1.Comparing results of heap sort for best case in microseconds

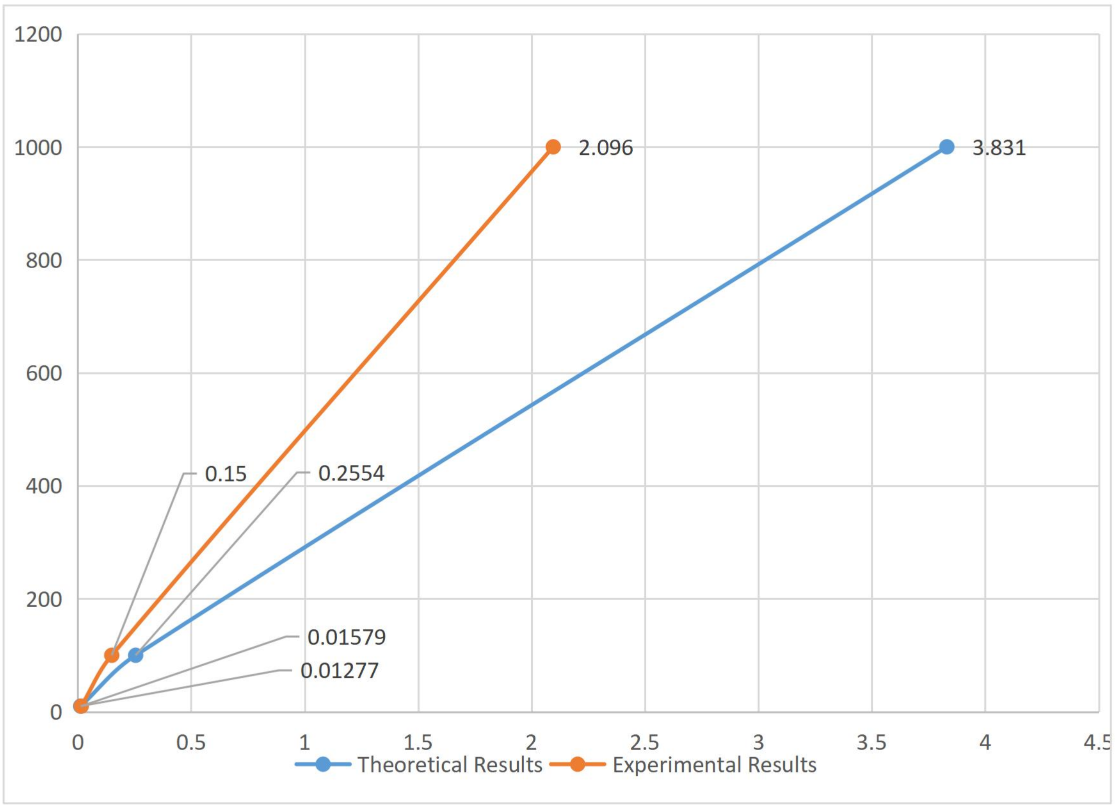

Figure2.Comparing results of heap sort for average case in microseconds

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. Results have shown that heap sort gives really nearly same sensitive for the experimental and theoretical results. Results are very high because they are in range of microseconds, so that means heap sort is really slow algorithm when I compare this results with other results I get from the other 4 algorithms.

5.Counting Sort Algorithm

Counting sort is a sorting technique based on keys between a specific range. It

works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence. O(n+k) where n is the number of elements in input array and k is the range of input. In my inputs range of input and size of inputs are same.

STEP 3:

❖ THEORETICAL RESULTS

In best case: T(n) = n+k

- n=10,k=10, T(n) = 20/2601,000,000= 7.689e^{-9} sec
- n=100, k=100,T(n) = 200/2601,000,000= 7.689e^{-8} sec
- n=1000,k=1000 T(n) = 2000/2601,000,000= 7.689e^{-7} sec

In worst case: T(n) = n+k

- n=10,k=10, T(n) = 20/2601,000,000= 7.689e^{-9} sec
- n=100, k=100,T(n) = 200/2601,000,000= 7.689e^{-8} sec
- n=1000,k=1000 T(n) = 2000/2601,000,000= 7.689e^{-7} sec

In average case: T(n) = n+k

- n=10,k=10, T(n) = 20/2601,000,000= 7.689e^{-9} sec
- n=100, k=100,T(n) = 200/2601,000,000= 7.689e^{-8} sec
- n=1000,k=1000 T(n) = 2000/2601,000,000= 7.689e^{-7} sec

❖ EXPERIMENTAL RESULTS

In best case: $T(n) = n\log_2 n$

- n=10, T(n) = $7.899e^{-9}$ sec
- n=100, T(n) = $4.737e^{-8}$ sec
- n=1000, T(n) = $8.684e^{-7}$ sec

In worst case: $T(n) = n\log_2 n$

- n=10, T(n) = $1.579e^{-8}$ sec
- n=100, T(n) = $4.342e^{-8}$ sec
- n=1000, T(n) = $7.322e^{-6}$ sec

In average case: $T(n) = n\log_2 n$

- n=10, T(n) = $2.368e^{-8}$ sec
- n=100, T(n) = $3.55e^{-8}$ sec
- n=1000, T(n) = $3.868e^{-7}$ sec

❖ CONCLUSION

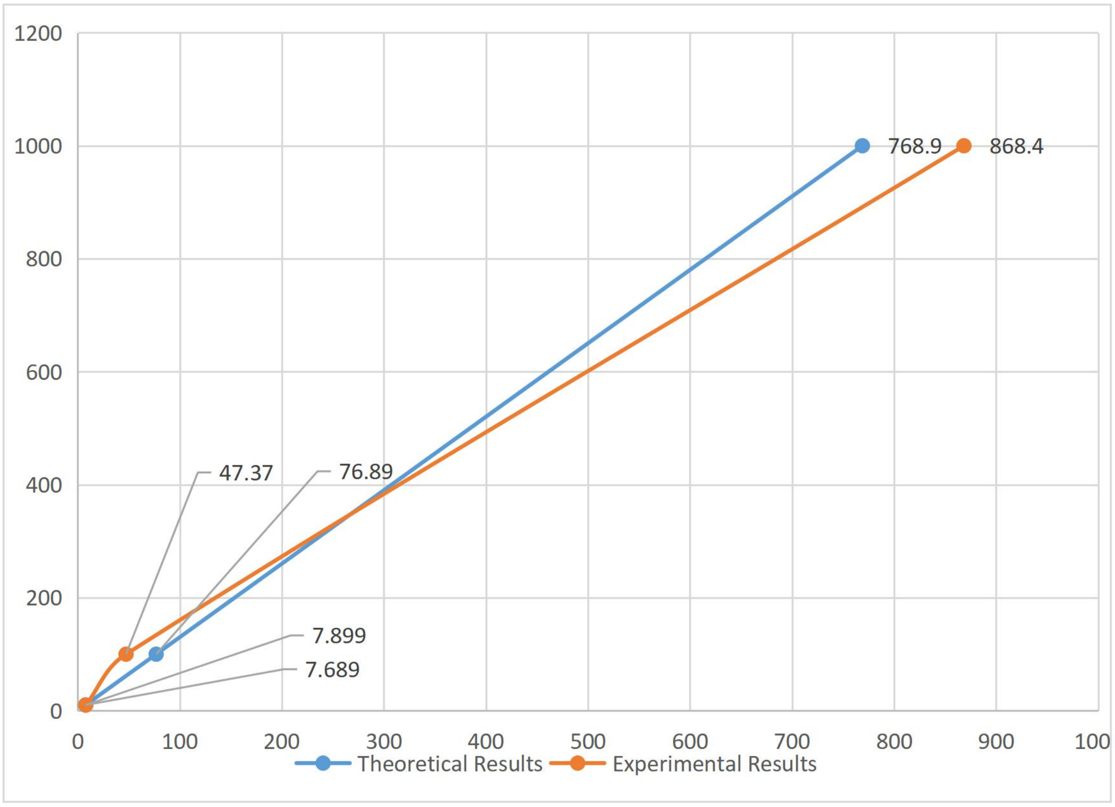

Figure1. Best case comparison for Counting sort in nanoseconds

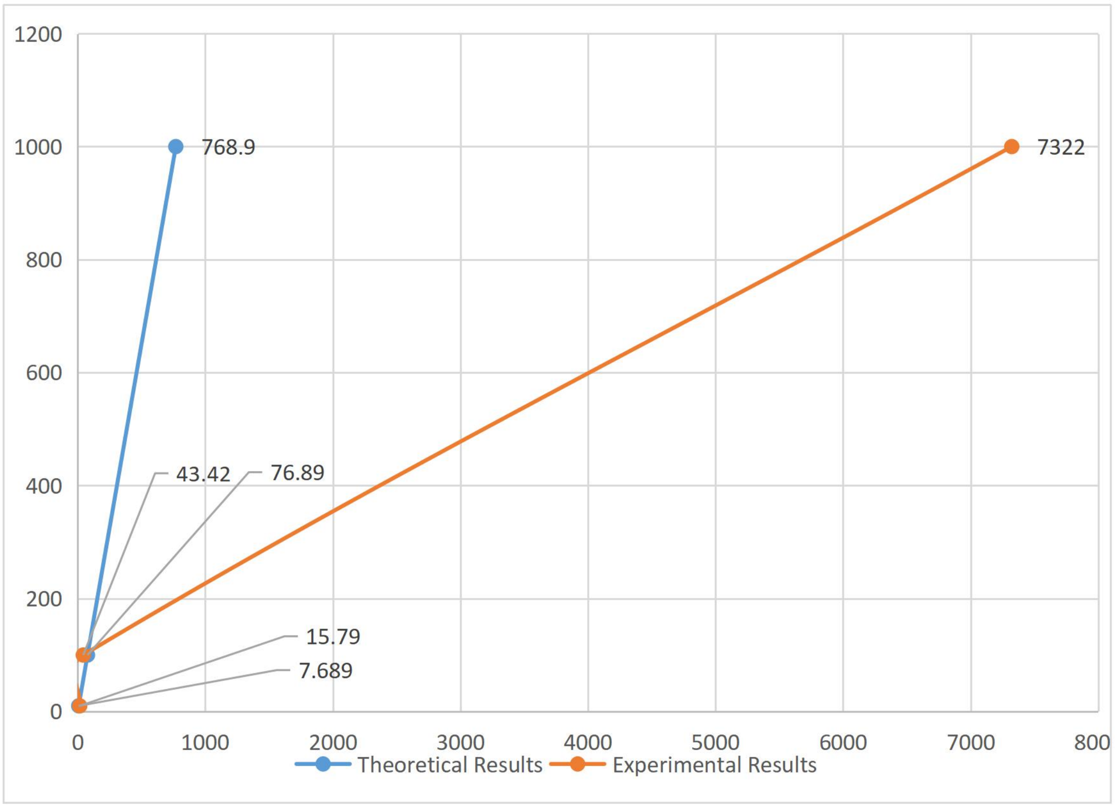

Figure2. Worst case comparison for Counting sort in nanoseconds

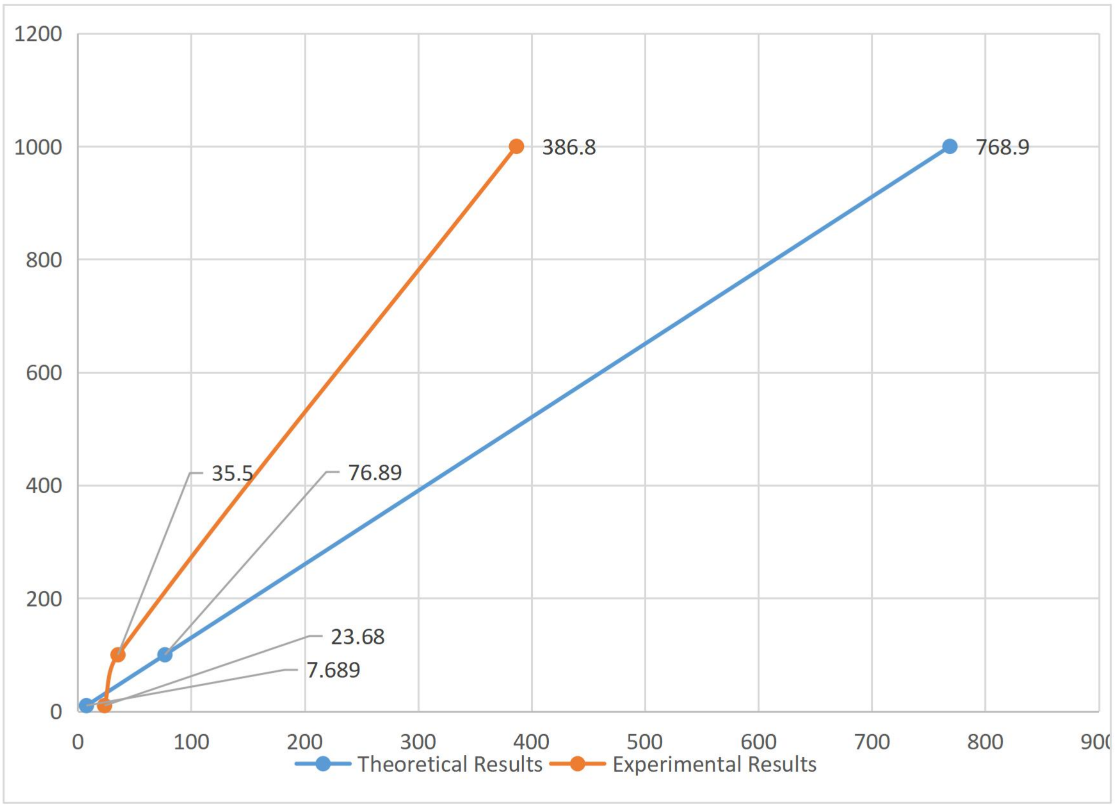

Figure3. Average case comparison for Counting sort in nanoseconds

Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K. Results off experiments very similar for sorted, unsorted and reverse sorted cases. But they are a bit different then theoretical result this is because theoretical results shows maximum time complexity it is in big Oh notation so it says it can be less than this number and it suits for experimental results. Results are very low so counting sort can be fastest algorithm.