Marmara University
Department of Computer Engineering
CSE246
Homework 2

Subject : Analysis of Algorithms
Programming Language : Java
Student : Ahmet KAYA

Experiment : Insertion , Quick and Merge Sort Performance Compare

1. Sorting Algorithms
2. Inputs
3. Results
4. Comparing Performance
5. Comparing Emprical with Theorical Results
6. Comment

1.Sorting algorithms

1.Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Bentley shows a three-line C version, and a five-line optimized version[1]:116
- Efficient for (quite) small data sets
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(nk)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

When people manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort.[2]

1.1 Best Case for Insertion Sort

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

1.2 Worst Case for Insertion Sort

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n^2)).

1.3 Average Case for Insertion Sort

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a

certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

2. Quicksort

The task is to sort an array (or list) elements using the *quicksort* algorithm. The elements must have a strict weak order and the index of the array can be of any discrete type. For languages where this is not possible, sort an array of integers.

Quicksort, also known as *partition-exchange sort*, uses these steps.

1. Choose any element of the array to be the pivot.
2. Divide all other elements (except the pivot) into two partitions.
   - All elements less than the pivot must be in the first partition.
   - All elements greater than the pivot must be in the second partition.
3. Use recursion to sort both partitions.
4. Join the first sorted partition, the pivot, and the second sorted partition.

The best pivot creates partitions of equal length (or lengths differing by 1). The worst pivot creates an empty partition (for example, if the pivot is the first or last element of a sorted array). The runtime of Quicksort ranges from $O(n \log n)$ with the best pivots, to $O(n^2)$ with the worst pivots, where n is the number of elements in the array.

3. Merge Sort

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p .. r]$:

1. **Divide Step**

   If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. **Conquer Step**

Conquer by recursively sorting the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. **Combine Step**

Combine the elements back in $A[p .. r]$ by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

2.INPUTS

In this experiment there are several inputs with different sizes and different sort types. There are two different criterion to compare sorting algorithms, one of them is to check same sorting algorithm with different sizes and sort types of inputs. The other one is to check each sorting algorithm with same sizes and sort types of inputs.

The first comparison gives us the information about what a sorting algorithm do when input size or sort type change and the information about the execution time, total basic operation number and comparison number. The second comparison gives us the information about what the sorting algorithms which we want to compare do with same inputs, which is the best algorithm according to execution time, base operations and used memory.

To understand the quick, merge and insertion sorts about their execution time and total base operation number. For this experiment there are six different size of inputs chosen and for each sizes of inputs there are four different sort type chosen.

The table shown below, shows the input sizes and types to compare sort algorithms.

TABLE 2.1

Sort Type	Input Size						
Random Sorted	5000	10000	25000	50000	100000	250000	500000
Best Sorted	5000	10000	25000	50000	100000	250000	500000
Worst Sorted	5000	10000	25000	50000	100000	250000	500000
Almost Sorted	5000	10000	25000	50000	100000	250000	500000

Random Sorted

Random variables located in each index of input array.

Best Sorted

Increasing order variables located in increasing index of input array.

Worst Sorted

Decreasing order variables located in increasing index of input array.

Almost Sorted

As a best sorted input but in a few random indexes of input there are random variables.

Input Size

Different input sizes for different execution time.

Different Inputs for a Sorting Algorithm

The inputs are chosen to see the difference between the execution time of a sorting algorithm. There can be an experiment with an array type T1, array size S1 and sorting algorithm A1. If we calculate the execution time of sorting algorithm A1 for this variables, we can find the execution time TM1 and base operation number B1.

T1 + S1 + A1 ⟹ TM1 + B1

T1 + S2 + A1 ⟹ TM2 + B2

T? + S? + A1 ⟹ TM? + B?

After this experiment

- If we just change the input size, there can be a new execution time and base operation number for the sorting algorithm S1
- If we just change the sort type, there can be a new execution time and base operation number for the sorting algorithm S1

According to this opinion, if we change the input size or sort type for a sorting algorithm. We can understand the change of execution time and base operation number of the choosen sorting algorithm.

Same Inputs for Different Sorting Algorithms

Use of an input with a size S1 and sort type T1, to compare different sorting algorithms (A1,A2,A3….) can show us the execution times and base operation numbers of sorting algorithms with this input to compare the algorithms.

T1 + S1 + A1 ⟹ TM1 + B1

T1 + S1 + A2 ⟹ TM2 + B2

T1 + S1 + A? ⟹ TM? + B?

According to this opinion, if we use the same input for all sorting algorithms. We can compare the outputs of the algorithm (Execution time, Base op. Num.) to each other to see which algorithm is best.

3. RESULTS

The experiment evaluated 10 times and the results of each sorting algorithms are recorded. The average results for all situations are calculated.

3.1 Insertion Sort

Insertion sort evaluated for all input size and type (TABLE 2.1) and the outputs are recorded .

a) The plot shown below shows the time (millisecond) with different input sizes for **Insertion Sort**

3.2 Quick Sort – Pivot Median of Three Element

Quick sort evaluated for all input size and type (TABLE 2.1) and the outputs are recorded .

a) The plot shown below shows the time (millisecond) with input sizes for Quick Sort (Median of Three).

3.3 Quick Sort – Pivot First Element

Quick sort evaluated for all input size and type (TABLE 2.1) and the outputs are recorded .

a) The plot shown below shows the time(millisecond) with different input sizes for Quick Sort (First Element).

3.4 Merge Sort

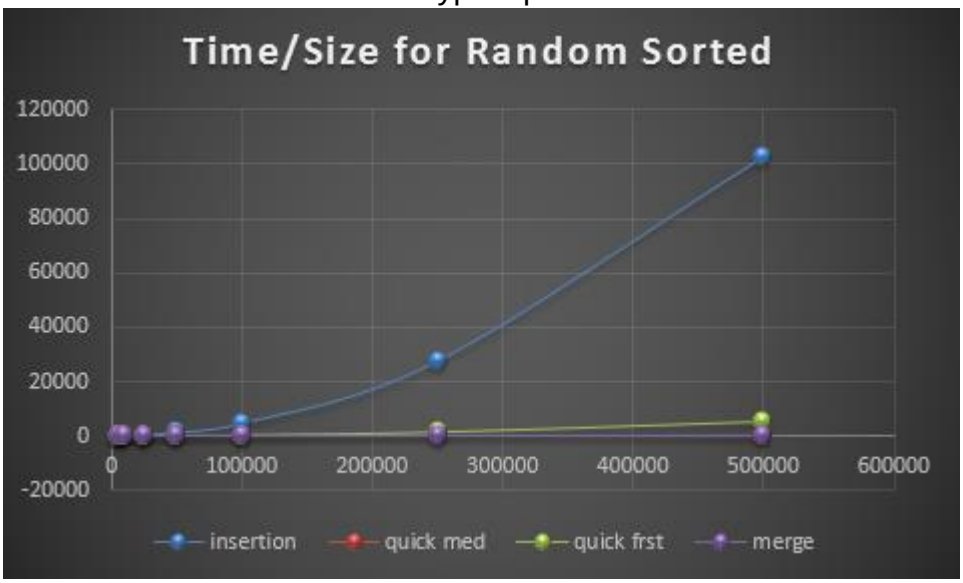
Merge sort evaluated for all input size and type (TABLE 2.1) and the outputs are recorded .

a) The plot shown below shows the time(millisecond) with different input sizes for Merge Sort

3.5 Random Input Type for All Sorting Algorithms

Random inputs with different sizes (TABLE 2.1) are created and same inputs are sorted with all sorting algorithms. The plots below shows the performances of the sorting algorithms for random inputs.

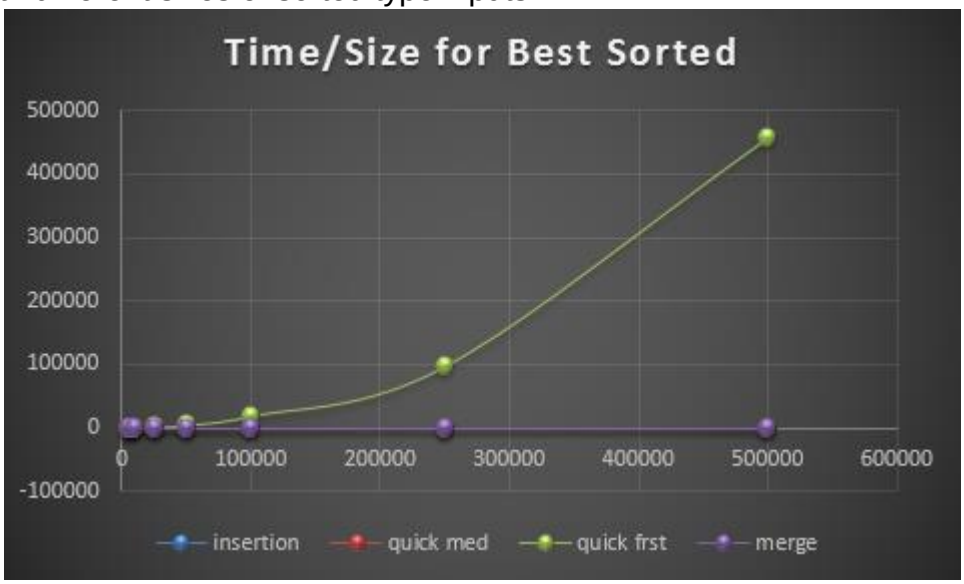
a) The plot shown below shows the time(millisecond) of each sorting algorithms with different sizes of random type inputs.



3.6 Sorted Input Type for All Sorting Algorithms

Sorted inputs with different sizes (TABLE 2.1) are created and same inputs are sorted with all sorting algorithms. The plots below shows the performances of the sorting algorithms for random inputs.

a) The plot shown below shows the time(millisecond) of each sorting algorithms with different sizes of sorted type inputs.



3.7 Reverse Sorted Type for All Sorting Algorithms

Reverse sorted inputs with different sizes (TABLE 2.1) are created and same inputs are sorted with all sorting algorithms. The plots below shows the performances of the sorting algorithms for reverse sorted inputs.

a)

3.8 Almost Sorted Type for All Sorting Algorithms

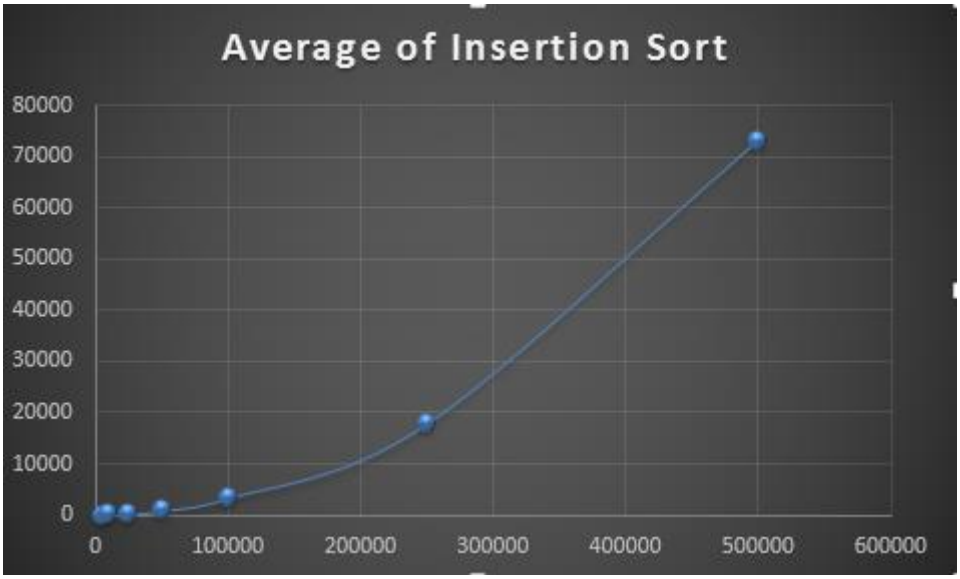
Almost sorted inputs with different sizes (TABLE 2.1) are created and same inputs are sorted with all sorting algorithms. The plots below shows the performances of the sorting algorithms for almost sorted inputs.

3.9 Average Time Situations for All Sorting Algorithms

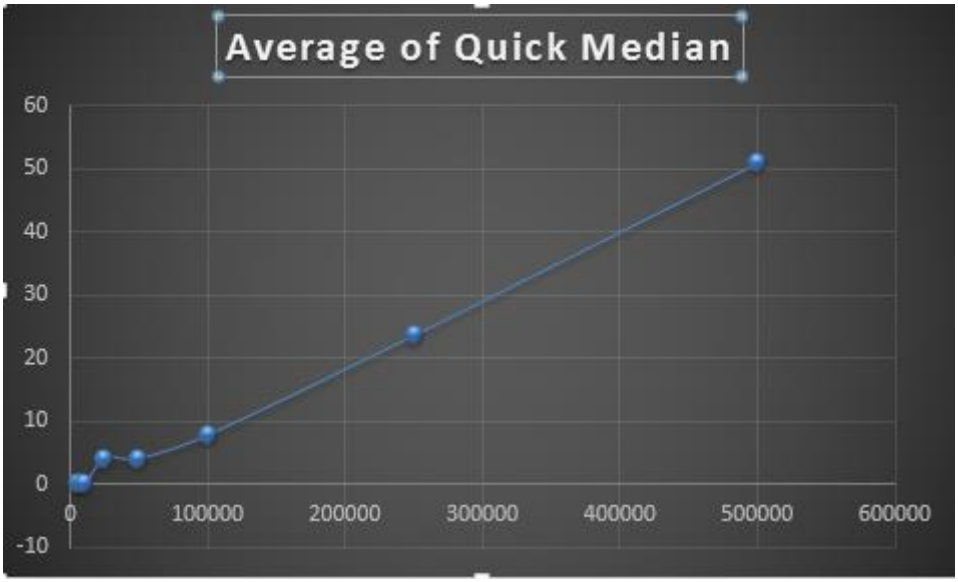
Average situations are calculated with random,sorted,reverse sorted and almost sorted types arrays with different sizes. There are 7 different sizes (TABLO 2.1). The averages of the situations are used to find the average of sorting algorithms.

The tables shown below show the average situations of all sorting algoritms

a) Average of Insertion Sort

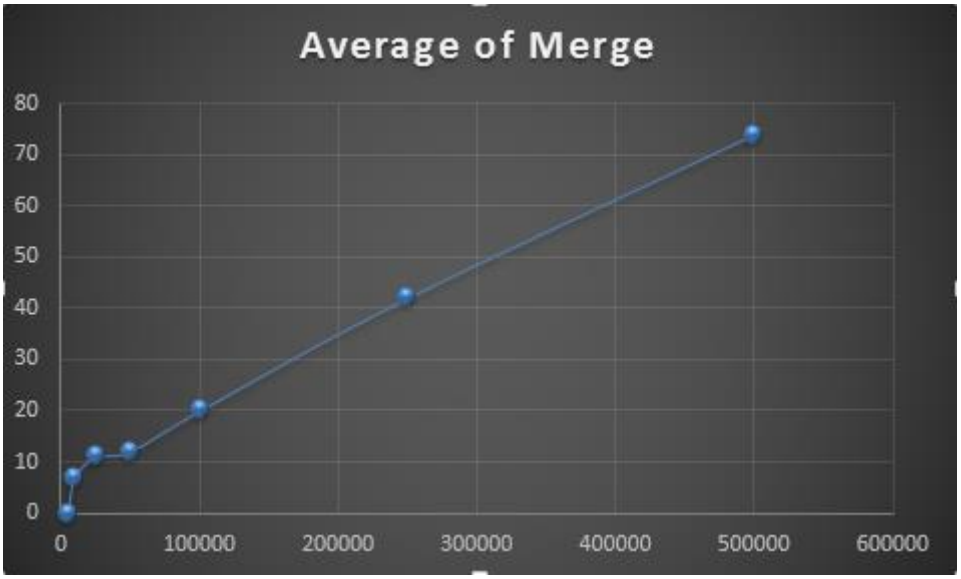


b) Average of Quick with three of median Sort



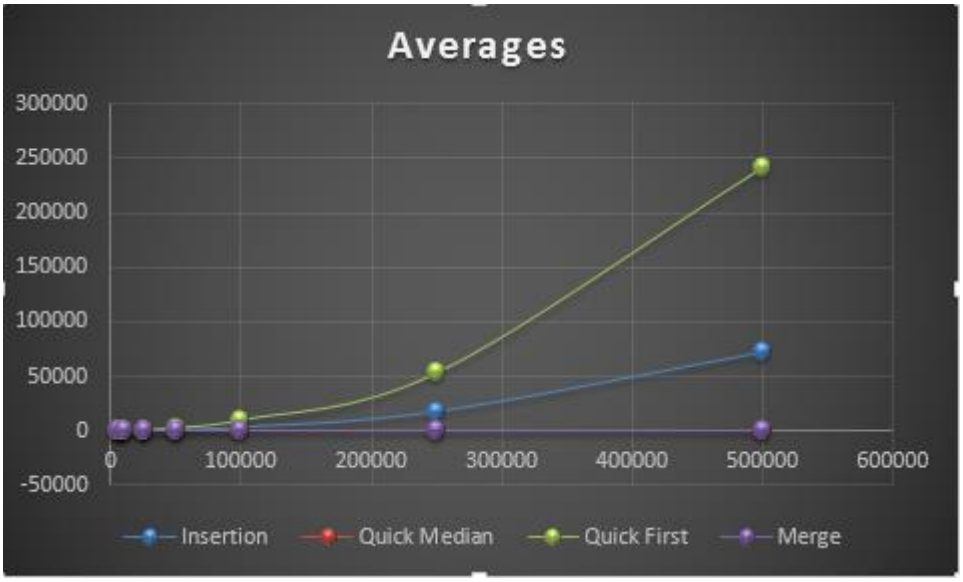
c) Average of Quick with pivot first element Sort

d) Average of Merge Sort



e) Averages of All Algorithms

The plot shown below shows the averages of all sorting algorithms.



4.COMPARING PERFORMANCE

For random type of arrays with different sizes, there are some different situations. If there is an array with small sizes. All the sort algorithms are useful but if there is a larger array, the insertion sort is very bad. The merge and quick sorts are useful for random type and larger size of arrays (Plot 3.5.a).

For already sorted arrays with different sizes, the situation is changed. Because in insertion sort the sorting process is going one by one. There is no need to do anything because of the arrays type is already sorted. So the insertion sort is already good. Merge and Quick sort with median of three are also useful for already sorted arrays. But the Quick sort with pivot is first element is not useful for sorted arrays. Becuase the algorithm choose the pivot as first element and for this pivot it checks all the elements of the array in decreasing index order and finds the right place for pivot. But pivot is already in right place so it makes more comparisons for all pivots. (Plot 3.6.a).

For reverse sorted arrays with different sizes, the Merge and Quick with median of three are useful. The Insertion sort is not useful than merge and quick with median but it is useful than Quick with pivot first element (Plot 3.7.a).

For almost sorted arrays with different sizes, the situation is similar to already sorted arrays situation. The algorithms with best performances are Merge and Quick with median of three. The insertion is also useful for arrays with size of smaller. But the Quick sort with pivot is first element have the worst performance (Plot 3.8.a). The thing that makes the performance worst may be the process while searching the right place for pivot. The algorithm tries to find the rigth place and while doing this, it must do more comparisons because the pivot is already in right place because of the almost sorted array type. Same situation with already sorted arrays.

For average situations of sorting algorithms, The Merge and Quick with Median of Three have best performance. The Quick with Pivot is First have good performance in small sizes of inputs. Getting larger of array size, the performance of the quick sort(pivot is first) goes down. The Insertion Sort have the worst performance .

5.COMPARİNG EMPRICAL WITH THEORICAL RESULTS

The theorical part

	Best	Average	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))
Insertion Sort	O(n)	O(n^2)	O(n^2)

5.1 Insertion Sort Compare

In theory the insertion sort has O(n) for best case. The plot below has a linear growth rate for basic operation number.

As the theoretical results, the empirical results have same situation.

In theory the insertion sort has O(n^2) for worst case. We can check the cases with base operation number for insertion sort because of the one by one process.

The empirical results are shown on plot below.

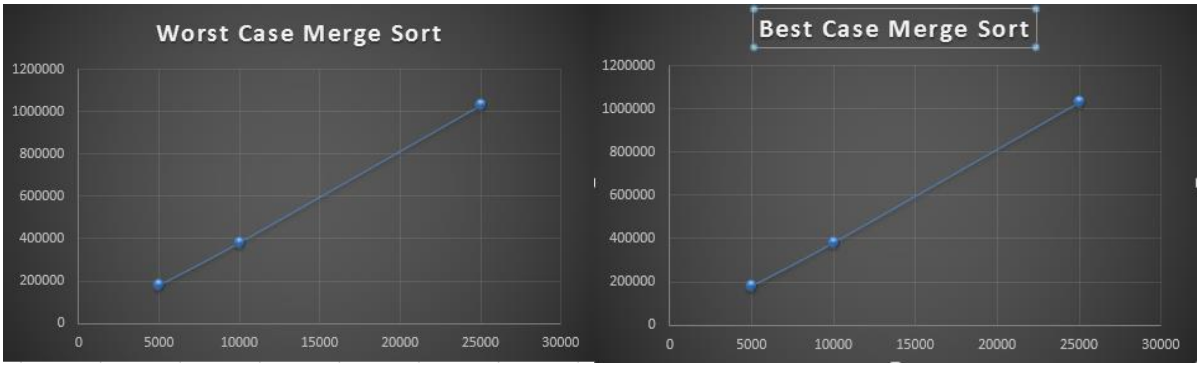
There is a similarity with empirical and theoretical results.

The average case is shown in plot (3.9.a). It has same growth rate with the worst case situation. It is similar to theoratical results.

5.2 Merge Sort

Merge sort have same growth rate O(n log (n)) in all cases in theory. If we check the plot (Plot 3.4.a) we can see the results of the best and worst cases of merge sort. The growth rate of the best and worst cases are same and it is similar to theoretical results and if we check the base operation numbers of worst and best cases, we can see the similar growth rate for each situation.

Merge Sort Base Operation Plots (Best - Worst)



For average case of merge sort. We can check the plot (3.9.d). The growth rate of the average case is almost same with best and worst case and it is similar to theoratical results.

5.3 Quick Sort Median of Three

The best and worst cases for this sorting algorithm. It is similar with theoratical results. We can check it on plot(3.2.a).

5.4 Quick Sort Pivot is First Element

The best and worst cases for this sorting algorithm. According to the plot (3.3.a) the best and worst cases have same growth rate and it is similar to theoratical results. For the average case we can check the plot (3.9.c). It has a growth rate same with theoratical results.

6.COMMENT

All the algorithms have different performances for different situations. The insertion sort is really good for really small samples or sorted input situations. But if input size grows, the performance of the insertion sort is going to be worst. If we have a very small or sorted arrays. We can use the insertion sort.

The quick sort algorithm also have good performance for more situations. The method to choose pivot is really important. Because the pivot makes decision on comparison number. For a sorted array the pivot choosen as first element, makes the process so long. But for same array the pivot choosen as median of three, makes the process very efficient. For random inputs each of the methods have good performance.

The merge sort is very useful for all situations. In all cases it uses the same process (divide and conquer). So for same sizes of arrays, it makes same base operations.

In quick sort for each method to choose pivot. The sorted array is not best case.

As a result, there is no rule to choose a sorting algorithm for an input. Because everything can change with your input.