ALGORITHM ANALYSIS

HOMEWORK 2 REPORT

OSMAN MANTICI

Sorting Algorithms

Step 1)

Deciding on Reasonable Inputs, Generating Reasonable Inputs, Deciding on Reasonable Metrics.

This experiment asked to comparing performances of different sorting methods while finding median value of an array. I decide to choose nine different input file. These are categorized in terms of input sizes which are 100, 1000 and 5000. Because insertion sort, merge sort, max heap sort and quick sort has differently bigOh or theta bound. In addition these three different input file separate in to three different types which are increased order, decreased order and randomly selected order. Because of algorithms behavior, most of them show best performance when input is already sorted and shows worst performance on the contrary side that input sorted reversely.

My experiment metrics will be millisecond based. My computer processer has 1.80GHz so it equal to 1,800,000,000 Hz and it means processer process 1,800,000,000 processes in a second.

Step 2)

Coding and Running

All algorithms implemented on java environment. I use "System.nanoTime()" in order to measure execution time of the relevant sorting code.

Step 3)

Providing Some Plots or Tables to Illustrate Performance of the Algorithms.

a.) Insertion Sort

Insertion sort algorithm has average $O(n^2)$ complexity. But it gives us O(n) on best case while on the other hand $O(n^2)$ on worst case.

Best case of this algorithm, input state in to already sorted. Like 0, 1, 5, 9, 15, 23…

Worst case of this algorithm, input state in to reversely sorted. It's like …23, 15, 9, 5, 1, 0

Average case of this algorithm, input like 1, 15, 5, 0, 23, 9…

- Theoretical Measurements

Best case: t(n)=n

N=100, t(n)=100/1,800,000,000 = $5.555e^{-8}$sec

N=1000, t(n)=1000/1,800,000,000 =$5.555e^{-7}$sec

N=5000, t(n)=5000/1,800,000,000 =$5.555e^{-6}$sec

Worst case: $t(n)=n^2$

N=100, t(n)=10,000/1,800,000,000 = $5.555e^{-6}$sec

N=1000, t(n)=1,000,000/1,800,000,000 =$5.555e^{-4}$sec

N=5000, t(n)=25,000,000/1,800,000,000 =$1.388e^{-3}$sec

Average case: $t(n)= n^2$

N=100, t(n)=10,000/1,800,000,000 = $5.555e^{-6}$sec

N=1000, t(n)=1,000,000/1,800,000,000 =$5.555e^{-4}$sec

N=5000, t(n)=25,000,000/1,800,000,000 =$1.388e^{-3}$sec

- Experimental measurements

Best case: t(n)=n

N=100, t(n)= 0.0025569sec

N=1000, t(n)= 0.0035344sec

N=5000, t(n)= 0.0083671sec

Worst case: $t(n)=n^2$

N=100, t(n)= 0.003155301sec

N=1000, t(n)= 0.0254524sec

N=5000, t(n)= 0.025833899sec

Average case: $t(n)= n^2$

N=100, t(n)= 0.004260119sec

N=1000, t(n)= 0.0119867sec

N=5000, t(n)= 0.0192793sec

b.) Merge Sort

Merge sort algorithm is an divide and conquer algorithm which has all cases O(nlogn) complexity. It divides list two parts until there is left one element and after merge them all with sorting until obtain main list. By the way it uses extra memory such as O(n) total with O(n) auxiliary.

Best case of this algorithm, input state in to already sorted. Like 0, 1, 5, 9, 15, 23…

Worst case of this algorithm, input state in to reversely sorted. It's like …23, 15, 9, 5, 1, 0

Average case of this algorithm, input like 1, 15, 5, 0, 23, 9…

- Theoretical Measurements

Best case: t(n)=nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)= 18495/1,800,000,000 =$2.78e^{-6}$sec

Worst case: t(n)=nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)= 18495/1,800,000,000 =$2.78e^{-6}$sec

Average case: t(n)= nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)=18495/1,800,000,000 =$2.78e^{-6}$sec

- Experimental measurement

Best case: t(n)= nlogn

N=100, t(n)= 0.002037099sec

N=1000, t(n)= 0.0038123sec

N=5000, t(n)= 0.0052829sec

Worst case: t(n)= nlogn

N=100, t(n)= 0.0021264sec

N=1000, t(n)= 0.006181sec

N=5000, t(n)= 0.0063445sec

Average case: t(n)= nlogn

N=100, t(n)= 0.003855899sec

N=1000, t(n)= 0.005223301sec

N=5000, t(n)= 0.0056131sec

c.) Max Removal Max Heap Sort

Max Removal Max Heap Sort is a sorting algorithm is that constructs a max heap while taking input and root of that heap has always max element. After fully constructed heap, max element of heap is removed and heap decreased 1 by 1. Furthermore heapify has O(logn) complexity of all cases. Creating heap process has O(n) and over all heap sort complexity is O(nlogn). Also it uses O(n) space complexity. It is sort of a improved selection sort.

Best case of this algorithm, input state in to already sorted. Like 0, 1, 5, 9, 15, 23…

Worst case of this algorithm, input state in to reversely sorted. It's like …23, 15, 9, 5, 1, 0

Average case of this algorithm, input like 1, 15, 5, 0, 23, 9…

- Theoretical Measurement

Best case: t(n)=nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)= 18495/1,800,000,000 =$2.78e^{-6}$sec

Worst case: t(n)=nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =1.6e^{-6}sec

N=5000, t(n)= 18495/1,800,000,000 =2.78e^{-6}sec

Average case: t(n)= nlogn

N=100, t(n)=200/1,800,000,000 = 1.1e^{-7}sec

N=1000, t(n)=3000/1,800,000,000 =1.6e^{-6}sec

N=5000, t(n)=18495/1,800,000,000 =2.78e^{-6}sec

- Experimental measurement

Best case: t(n)= nlogn

N=100, t(n)= 0.0020018sec

N=1000, t(n)= 0.0027671sec

N=5000, t(n)= 0.0045035sec

Worst case: t(n)= nlogn

N=100, t(n)= 0.0021725sec

N=1000, t(n)= 0.005207sec

N=5000, t(n)= 0.0070831sec

Average case: t(n)= nlogn

N=100, t(n)= 0.002004sec

N=1000, t(n)= 0.003570401sec

N=5000, t(n)= 0.0052927sec

d.) Median-of-Three Quick Sort

Quick sort walk through with a pivot element whole sorting process. And median-of-three version of quick sort has always pivot as a median value of first, last and medium keys of an array. There is more than one pivot always unless the array size bigger than 2. Because it is a divide and conquer algorithm too. The pivot selection done by partition function. Elements smaller than pivot placed on left of pivot and elements bigger than pivot placed on right side of pivot. Best case depends on the pivot element. If it is always selected medium

element, this complexity closes to O(n) or O(nlogn) overall. Worst case and average case has same O(nlogn) complexity.

- Theoretical Measurement

Best case: t(n)=nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)= 18495/1,800,000,000 =$2.78e^{-6}$sec

Worst case: t(n)= nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)=18495/1,800,000,000 =$2.78e^{-6}$sec

Average case: t(n)= nlogn

N=100, t(n)=200/1,800,000,000 = $1.1e^{-7}$sec

N=1000, t(n)=3000/1,800,000,000 =$1.6e^{-6}$sec

N=5000, t(n)=18495/1,800,000,000 =$2.78e^{-6}$sec

- Experimental measurement

Best case: t(n)= nlogn

N=100, t(n)= 0.002699451sec

N=1000, t(n)= I could not get a result, execution did not stop

N=5000, t(n)= I could not get a result, execution did not stop

Worst case: t(n)= nlogn

N=100, t(n)= 0.00286367sec

N=1000, t(n)= I could not get a result, execution did not stop

N=5000, t(n)= I could not get a result, execution did not stop

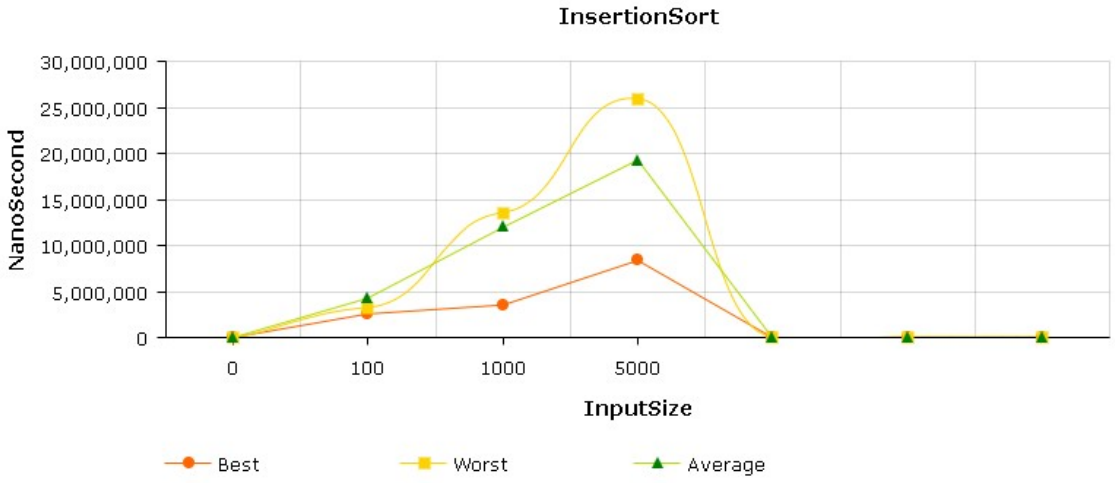
Average case: t(n)= nlogn

N=100, t(n)= 0.005011658sec

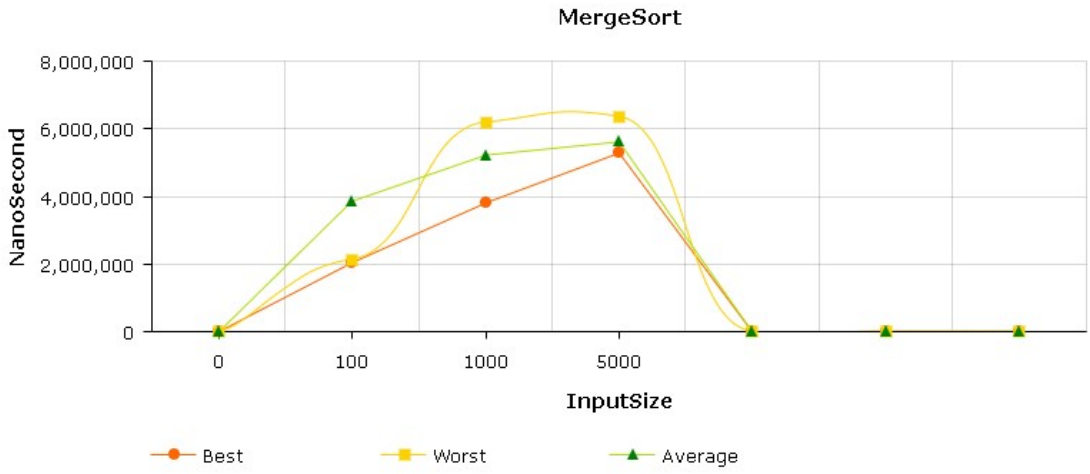
N=1000, t(n)= I could not get a result, execution did not stop

N=5000, t(n)= I could not get a result, execution did not stop

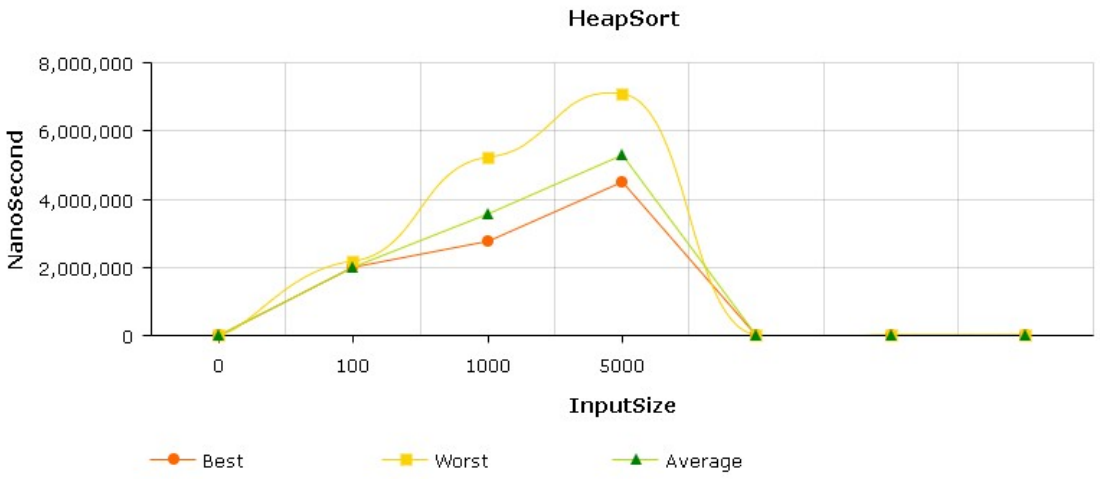
Conclusion



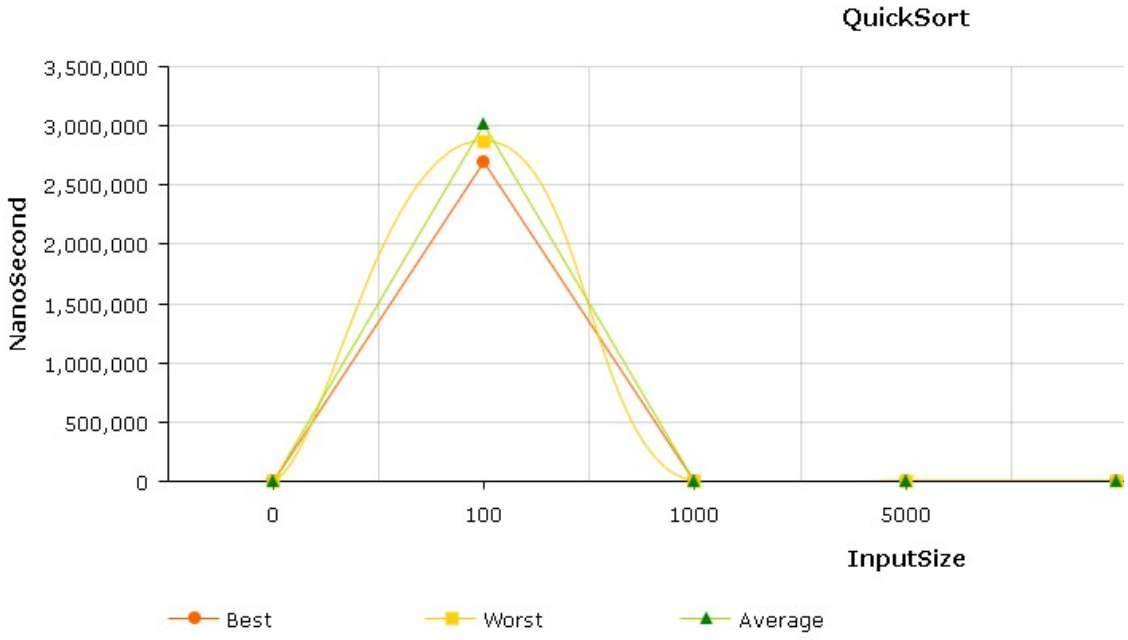
Insertion sort has good results on best cases of different inputs but at worst cases when input size increase the execution time reaches extremely huge values.



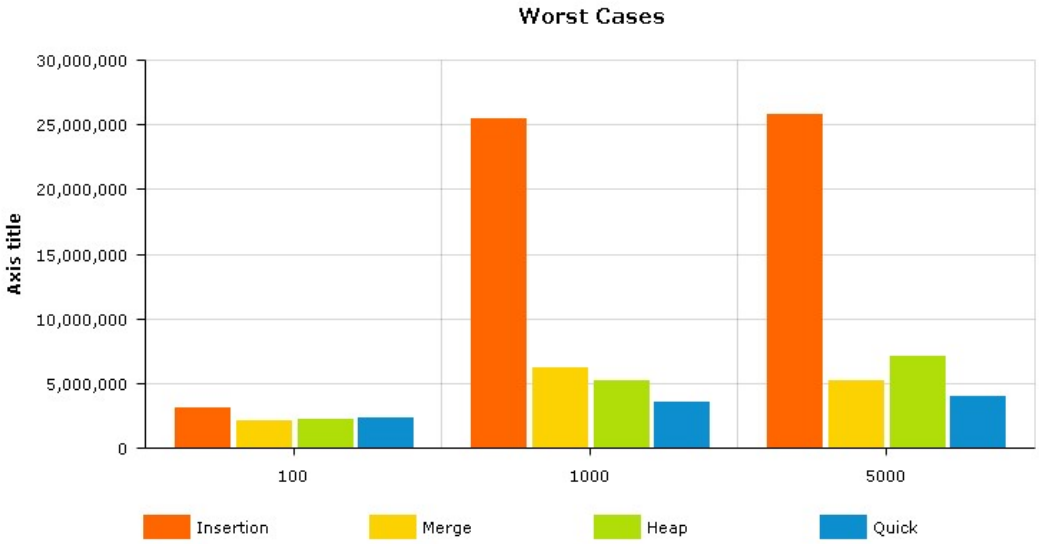
As I can see, merge sort has good accuracy with low size inputs. It seems worst and best cases not extremely different. Although while size increasing execution time also increasing but the difference of them not significantly large.



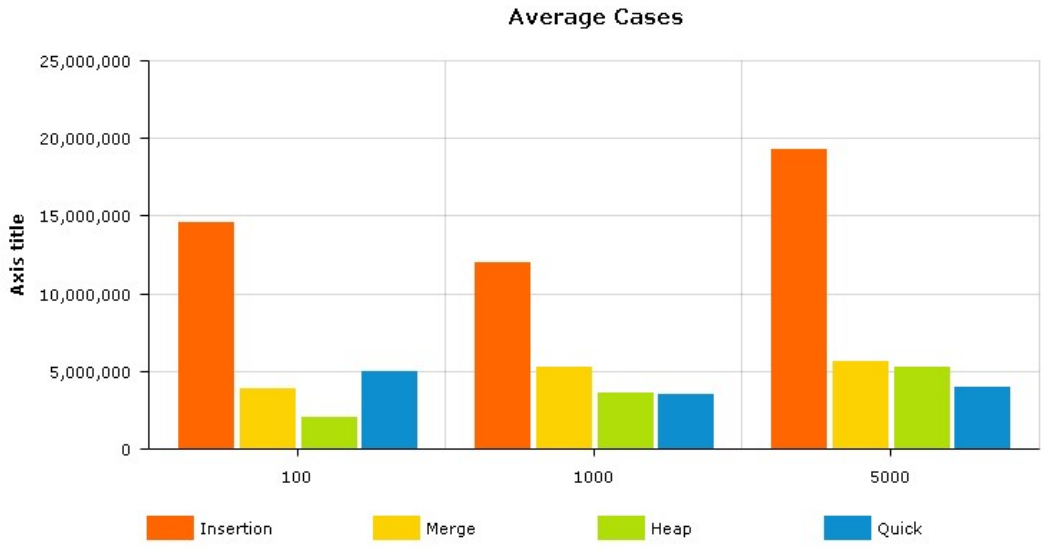
Heap sort also has a good execution time. At the small sizes performance do not changes slightly. As a fact, large number inputs also have very efficient execution time. It seems worst case time is so large but actually is a good outcome.



I could not get any output for large inputs for quick sort but if I interpret an outcome quick sort with median of three algorithm is gives us good outputs rather than merge and insertion sort definitely and I don't think there is a big different between heap sort and quick sort. Although whether the difference is big or not I think quick sort is better algorithm for sorting. Because its heapify function has O(logn).

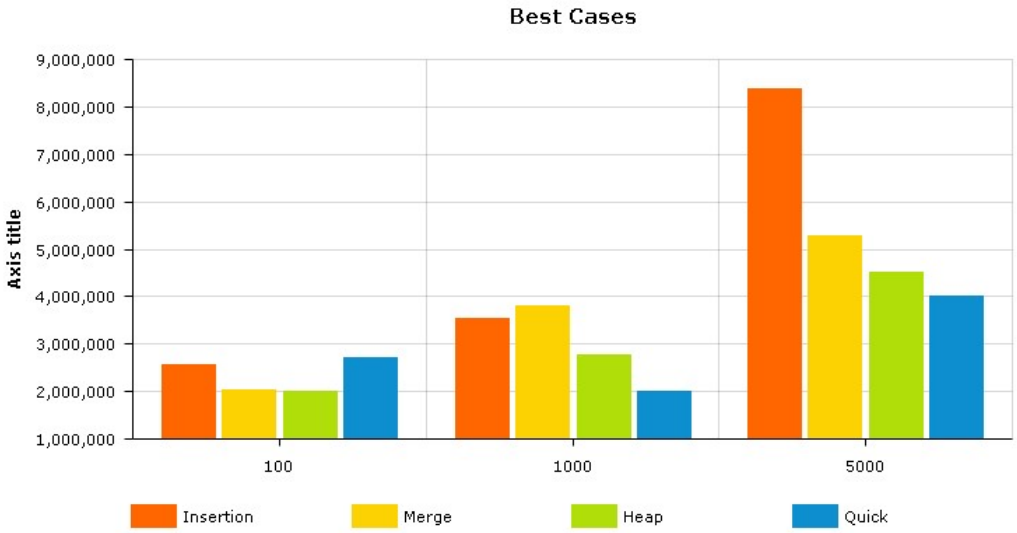


As we can see insertion sort requires very very large time if input size increases even if input size n have not reach yet n^2. The best algorithm on worst cases is quick sort. If I sort the sorting algorithms the line is like insertion, merge, heap and quick sort worst to best. But we can see that the input size effects the time too much.



On average cases, we conclude that again insertion sort is the slowest algorithm all input sizes again. Heap sort has a good output than the others in small size. But the important thing is as the input increasing, quick sort and merge have a better accuracy on

values. Heap sort also has good outputs but heap sort more much uses memory heap than merge sort.



If we look the best cases, we easily see that insertion sort is a good algorithm at a point of input size. Because we see that before the execution time increases rapidly. Another conclusion that the heap sort is also a good algorithm at a point of input size because as I mentioned before it is a kind of improved selection sort and it not completely different from insertion sort. But more important conclusion that merge sort execution time is not good enough this time. Because it requires unnecessarily divide operation. List is already sort but it does not control this situation.