DAA - Sorting Analysis Report

By - **Mohd Javed Ali**
IIT2018501

1. Insertion Sort

Brief of Algorithm : It is based on comparison of elements, which divides array into 2 parts Sorted and unsorted .For sorting in ascending order array , first half of array is considered sorted , whereas other part considered as unsorted. Element of first half are sequentially compared with outer loop element. If Outer loop element is smaller than it is swapped with larger element present in first half.

Pseudo Code :

```
int n     //Size of array
Int A[n]  //Array
for i in 1 to  n
   for j in 0 to i-1
      if(A[i]<A[j])
        swap(A[i],A[j])
```

Complexity Analysis : Time Complexity

Let time taken by each statement C_0 ,C_1 , C_2 , C_3, C_4 and inner loop Command takes C_5, C_6 , C_7 , C_8 time respectively. Then outer loop runs for n times whereas inner loop runs i times respectively.

$$T(n) = C_0 + C_1*n + C_2*n + C_3*n + C_4*n + \sum^{i=n-1}_{i=2} (C_5*i + C_6*i + C_7*i + C_8*i)$$

$$= C_0' + C_1'*n + \sum^{i=n-1}_{i=2}(C_i)$$

$$= a + b*n + c*n^2$$

$$T(n) = \Theta(n^2)$$

1. Best Case : $\Omega(n)$

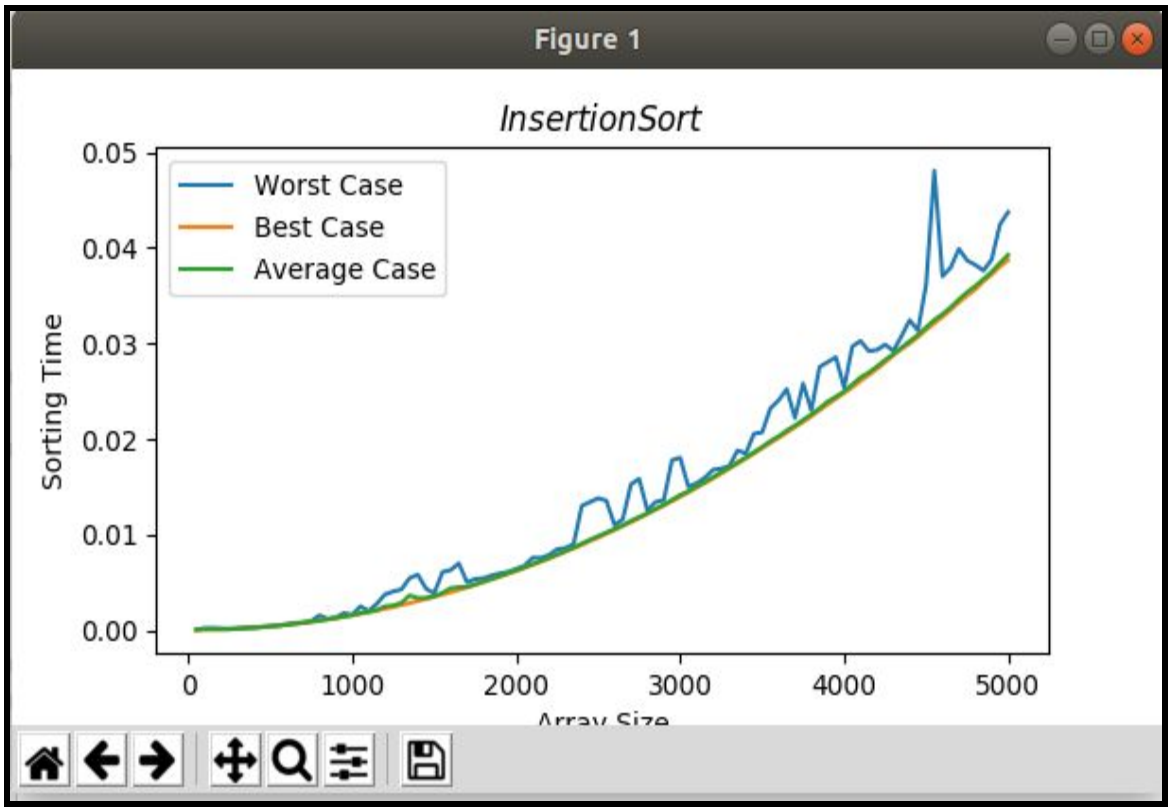
2. Average Case : $\Theta(n^2)$

3. Worst Case : $O(n^2)$

Space Complexity - O(1)

Comments : Average Case Time Complexity follows graph of y=x^2 pattern and for sorted part it contains y=x portion also

Graph:



2. Quick Sort

Brief of Algorithm : It is based on divide and conquer algorithm where repeatedly division of and their merge in sorted order takes place. Pivot is selected which could be any random array element or last or first element of array.
Then elements less than pivot are stored in first half and element larger than pivot are stored in second half and pivot is inserted at its correct position in sorted array.

Pseudo Code :

For Partition - **partition(A,low,up)**

```
pivot = up
ptr = low-1
for j low to high          //Element less and greater than pivot
  if(A[j] < pivot)         //are seperated
   swap(A[j],A[i++])
swap(A[ptr+1], A[u])
return ptr+1
```

Quick Sort Function: **quick_sort(int A[], int l, int u)**

```
if(l<u)
  pivot = partition(A,l,u);
  quick_sort(A,l,pivot-1);
  quick_sort(A,pivot+1,u);
```

Complexity Analysis : Time Complexity

Using Master Method to solve it -

$T(n) = \{ \ \Theta(1)$ if n=1
$\{ \ \Theta(n) + T(a) + T(n-a-1) + T(1)$ if n>1

For Best Case

$T(n) = \{ \ C_1$ or $\Theta(1)$ if n=1
$\{ T(n/2) + T(n/2) + \Theta(n)$ if n>1

Complexity - $\Theta(n\log n)$ //Calculated in next part

For Worst Case :

$T(n) = \{ \Theta(1) \text{ if } n=1$
$\{ \Theta(n) + T(1) + T(n-1) \text{ if } n>1$

For each level :

(cn)
$\Theta(1)$ c(n-1)
$\Theta(1)$ c(n-2)
$\Theta(1)$ c(n-3)
…….

$T(n) = cn + c(n-1) + c(n-2) + \ldots\ldots + n*\Theta(1)$

$= \sum_{i=0}^{i=n-1} c(n-i) + n*\Theta(1)$

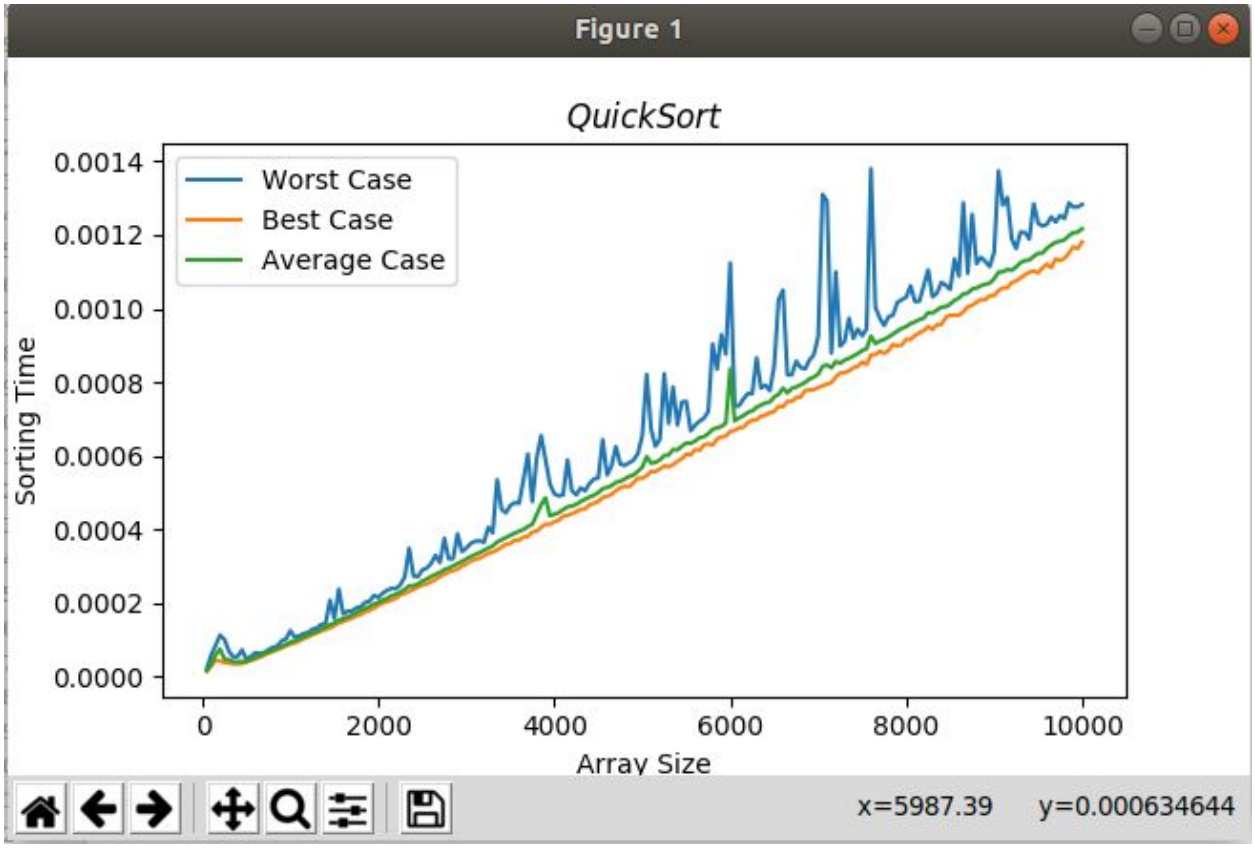
$=\Theta(n) + \Theta(n^2)$
$= \Theta(n^2)$

1. Best Case : $\Omega(n\log(n))$
2. Average Case :$\Theta(n\log(n))$
3. Worst Case : O(n^2)

Space Complexity - O(1)

Comments : Average Case Time Complexity follows graph of y=x*lnx pattern whereas for Big n it is tending to follow $y=x^2$ pattern dotted by blue lines. For smaller value of n, it is x*lnx and x^2 function are comparable whereas for large n it shows great deviation.

Graph : Attached Below



3. Merge Sort

Brief of Algorithm : It is based on divide and conquer algorithm in which array is divided and merged back to sorted array. Recursion is used array is divided till size becomes 1 , then merger of 2 sorted array takes place such that overall it forms sorted array

Pseudo Code : int n //Number of elements
int A[n] // Array

Merge Function : **merge(A[] , low, up)**

```
if(l<u)
   mid = (low+up)/2
   merge(A,low,mid)
   merge(A,mid+1,up)
```

patch(A,low,up,mid)

Function for merging 2 sorted arrays:

```
patch(A[],  low , up ,mid)
   B[] - low to mid
   C[] - mid+1 to up
   i=0 and j=0
   while(i<B.size() && j<C.size())
   {
       if(B[i]<C[j])
          A[low++] = B[i++]
      else
          A[low++] = C[j++]
   }
   while(i<B.size())
         A[low++] = B[i++]
   while(j<C.size())
         A[low++] = C[j++]
```

Complexity Analysis : Time Complexity

$T(n) = \{ C1 \text{ or } \Theta(1) \text{ if } n=1$
$\{ T(n/2) + T(n/2) + \Theta(n) \text{ if } n>1$

Let time complexity of each statement be c , then merge sort is based On divide and conquer algorithm,which repeatedly divide array in 2 half

		Sum
From top level -	cn	cn
	cn/2 , cn/2	cn
	cn/4 , cn/4 , cn/4 , cn/4	cn
	….	...
	……..	cn

At depth d, complexity of each problem will be $(cn)/(2^d)$
Number of sub-problems are 2^d

Total time taken at depth d - $(cn)/2^d * 2^d = 1$
$T(n) = \sum^{i=\log_2 n - 1}_{i=0} cn + \Theta(n)$

$= cnlog_2n + \Theta(n)$

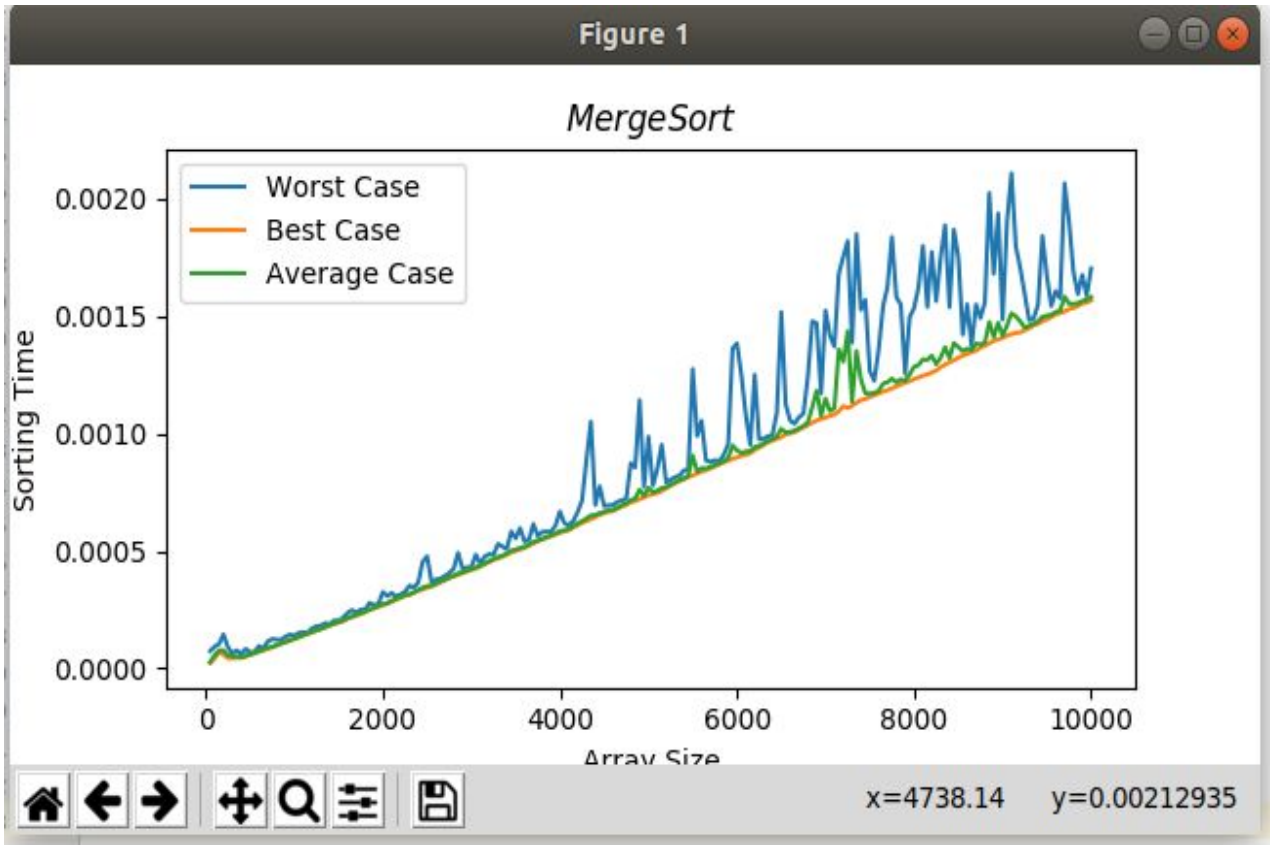
$= \Theta(nlog_2n)$

Therefore , $T(n) = \begin{cases} \Theta(1) & \text{if } n=0 \\ 2*T(n/2) + \Theta(n) & \text{if } n>0 \end{cases}$

1. Best Case : $\Omega(nlogn)$
2. Average Case : $\Theta(nlogn)$
3. Worst Case : O(nlogn)

Space Complexity - O(n)

Comments : Average Case Time Complexity follows graph of y = x*logx pattern . For small values of n all cases tends to follow same trend whereas as n increases worst case time complexity tends to deviate from best and average case complexity.



4 . Heap Sort

Brief of Algorithm : It is based on binary heap data structure . A binary heap is complete binary tree in which items are stored such that value present in parent node is greater than both of its child nodes i.e greater than left and right child nodes , such after heapifying it top element will be maximum . Array is used for representing heap where for ith node , parent is present At (i/2)th index , left child is present at 2*i index and right child is present At 2*i + 1 index.

Pseudo Code :

Nodes **from index n/2+1 to n are leaf nodes**

For heapify : **heapify(A,i,n) //i is current index**

```
low=2*i
up=2*i+1
max = i  //max is index to be swapped
if(low<=n and A[low]>A[max])
      max=low
if(up<=n and A[up]>A[max])
       max=up
if(i!=max)
{
        swap(A[i],A[max])
        heapify(A,max,n)
}
```

For buildheap: **buildheap(A,n)**

```
for i from n/2 to 0
    heapify(A,i,n)
```

Complexity Analysis : Time Complexity

This contains 2 parts buildheap and heapify , buildheap takes $\Theta(n)$ take Whereas heapify takes $\Theta(\log n)$ time

A heap of size n has at most $(n/2^{h+1})$ nodes with height h.
For Building Heap :

$$T(n) = \sum_{h=0}^{h=\log_2 n} [(n/2^{h+1}]*\Theta(h)$$

$$= \Theta(n * \sum_{h=0}^{h=\log_2 n} [(n/2^h])$$

$$= \Theta(n * \sum_{h=0}^{\infty} [(n/2^h])$$

Using Property of GP : $\sum_{h=0}^{\infty} nx^n = x/(1-x)^2$

$$T(n) = \Theta(n * (½) / (1 - (½)^2)$$

$$= \Theta(2*n) = \Theta(n) \text{ for building heap}$$

Whereas heapify function has complexity of Θ(logn)

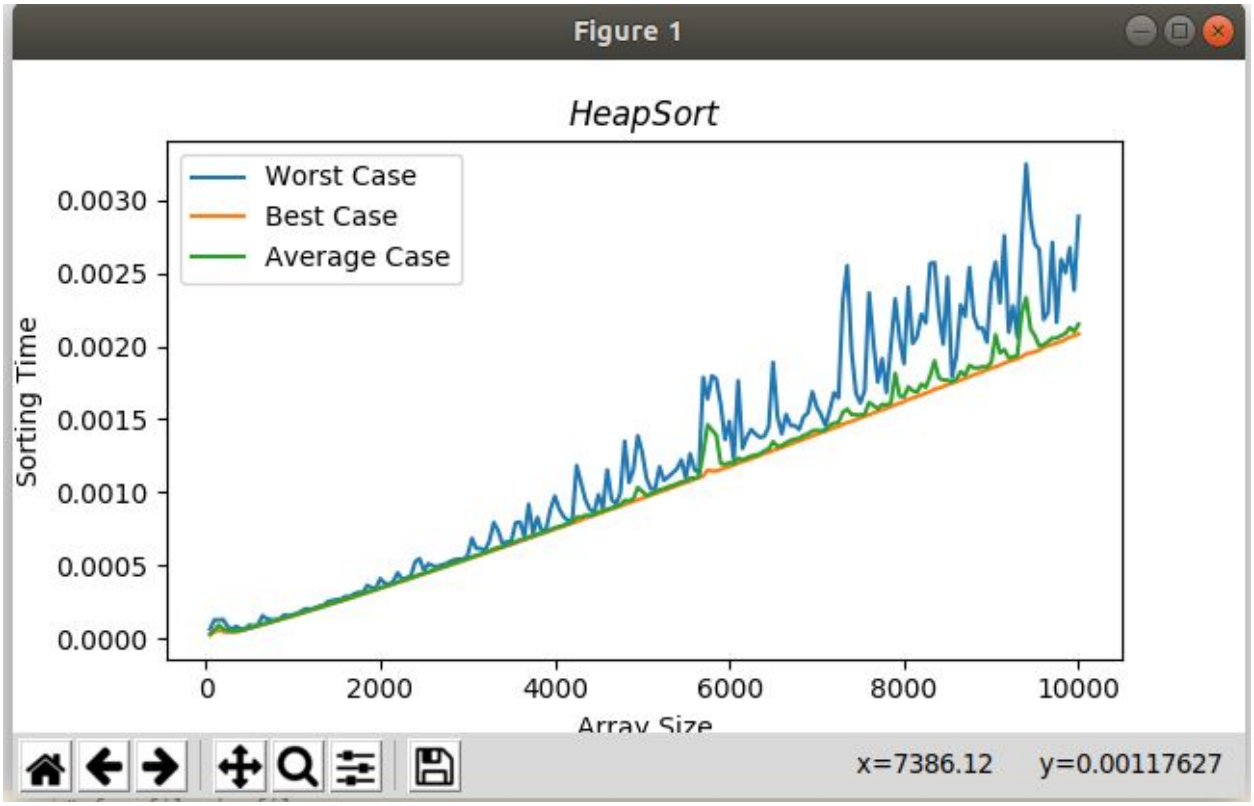
So , overall complexity of heapsort is Θ(nlogn)

1. Best Case : Ω(nlogn)
2. Average Case : Θ(nlogn)
3. Worst Case : O(nlogn)

Space Complexity - O(1)

Comments : Average Case Time Complexity follows graph of y = x*logx pattern . For small values of n all cases tends to follow same trend whereas as n increases worst case time complexity tends to deviate from best and average case complexity.

Graph: Attached Below



5. Hash Table Sort

Brief of Algorithm : This sorting algorithm is based on hashing of current array element with index of another array i.e B[A[i]] but this it has limitation it can be upto 10^5 elements because this is the maximum size of array.
It marks position of current array element in second array corresponding to their indexes then traversing second array from index 0 if value at Present index is greater than 0 , then we print it. It will print sorted array.

Pseudo Code :

```
int n     //Size of array
int A[n]  //Array
Int B[100005]
memset(B, 0 ,sizeof(B))
for i from 1 to n
        B[A[i]]++
max    //Stores maximum element of array A
for j from 1 to max
```

```
if(B[i]>0)
  print(i)
```

Complexity Analysis : Time Complexity

It has only single loop which has bound upto maximum element of Array , i.e max_element.

1. Best Case : Ω(max_element + n)

2. Average Case : Θ(max_element + n)

3. Worst Case : O(max_element + n)

Space Complexity - O(max_element)

Comments : Average, Best and Worst case follows nearly same trend for particular n.
Graph: