**TIME COMPLEXITY OF AN ALGORITHM**

* Time complexity of an algorithm is generally defined as the amount of time taken for it to run.
* There are two ways in which an algorithm’s time complexity can be determined first one being empirical or experimental analysis and second one being asymptotic or theoretical analysis.
* Generally an algorithm has three different cases best case, average case, worst case where the inputs differ and the run time depends on the kind of input we give.
* In the given assignment we will study about the different cases for different sorting algorithms like Insertion sort, Selection sort, Merge sort, Bubble sort.
* In the best case the input we give is a sorted array and we analyse the time complexity, in worst case the input is a reverse sorted array and in average case a random array is generated and time taken by different algorithms is analysed.
* In experimental analysis we plot a graph between the execution time of the algorithm T (N) vs input size N for different sort algorithms and determine which algorithm functions well for different cases.
* Asymptotic notations to determine an algorithm’s time complexity are of five types

O − Big Oh

Ω − Big omega

θ − Big theta

o − Little Oh

ω − Little omega

Below we analyse the best, worst, average cases of different sorting algorithms that includes selection sort, insertion sort, merge sort, bubble sort.

**Bubble sort**

**Pseudo code**

begin Bubble Sort(list)

for all elements of list

if list[g] > list[g+1]

swap(list[g], list[g+1])

end if

end for

return list

end Bubble sort

* In experimental analysis I plotted a graph for time taken by the bubble sort to run for different cases which are average case, best case, worst case. For the average case the input size is from 100-4000 with an interval of 50 and for the best and worst cases the size was from 0-5000 with an interval of 100.
* When we look into the asymptotic notation for analysing the bubble sort

time complexity for average case is O(n2).In average case as mentioned above we generate a random array and then the sort is performed.

* The basic operation in the bubble sort iterates in arithmetic progression with a sequence n-1+n-2+n-3…..+3+2+1 hence the time complexity is O(n2)
* Best case time complexity for bubble sort is O(n).In this case the array is already a sorted one therefore the number of comparisons is N hence complexity is O(n).
* Where as in worst case the time complexity of this sort is O(n2).The inner loop runs for n-1 times for the first iteration and n-2 times for 2nd iteration ….1.When we sum up all the iterations it can be denoted by n\*(n-1)/2 .

**Growth order function**

* For the average case the time complexity in the case of bubble sort is O(n2)

Complexity when n is doubled ie; O (2N) is 4n2

Complexity of O(KN) is K2n2 , by dividing the above expressions the growth order function of bubble sort in average case is O(KN)/ O(2N)=k2

Hence the bubble sort grows at the rate of K2 in this case.

* For best case the order of growth function is defined as O(KN) /O(2N)=K therefore the bubble sort grows at the rate of K in best case.
* In worst case the bubble sort grows at the rate of K2.

**Selection Sort**

**Pseudo Code**

SELECTION-SORT(A)  
 for j ← 1 to n-1  
 minimum← j  
 for i ← j + 1 ton  
 if A[ i ] < A[ minimum ]  
 minimum ← i  
 Exchange A[ j ] ↔ A[ minimum ]

* In experimental analysis I plotted a graph for time taken by the selection sort to run for different cases which are average case, best case, worst case. For the average case the input size is from 100-4000 with an interval of 50 and for the best and worst cases the size was from 0-5000 with an interval of 100.
* In selection sort the time complexity for all the cases is the same ie; O(n2)
* In all the cases the basic operation iterates in arithmetic progression with the sequence of n-1+n-2+n-3…..+3+2+1 hence the time complexity is O(n2)

**Growth Order Function**

* For average case the time complexity of selection sort is O(n2)

Complexity when N is doubled is 4 n2

Complexity of O(KN) is K2n2,after simplifying the growth order function in the average case O(KN)/ O(2N)=k2

Therefore the selection sort grows at the rate of K2 in the average case

* When we consider the best and worst cases for this sort the growth order function is similar to that of average case and it grows at the rate of K2.

**Insertion sort**

**Pseudo Code**

insert(b,k) i←k

x ← b[k]

while x < b[i − 1]

x ← b[i]

b [i ] ← b[i − 1] i ←i −1

b [i]←x return

insertion-sort (b,n)

m ← select-min(b,1,n) swap(b,1,m)

for i from 2 upto n

insert (b,i,n) return

* In empirical analysis I plotted a graph for time taken by the bubble sort to run for different cases which are average case, best case, worst case. For the average case the input size is from 100-4000 with an interval of 50 and for the best and worst cases the size was from 0-5000with an interval of 100.**.**
* For average case the time complexity of insertion sort is O(n2).
* In the average case the list is partially sorted but the all elements have to be compared with each other the inner for loop iterates in arithmetic progression with the sequence 1+2…n-1 and therefore the complexity is O(n2).
* In best case the list is fully sorted already therefore the time complexity is O(n),the inner for loop is executed with a sequence of 1+2….+n-1 which leads to time complexity being O(n).
* In worst case the time complexity for insertion sort is O(n2),the loop statements run in arithmetic progression and summation of all the iterations is n\*(n-1)/2.

**Growth Order Function**

* In the average case complexity is O(n2)when the input size is doubled the complexity for O(2n) is 4n2 therefore the complexity for O(KN)is K2n2 hence the insertion sort grows at the rate of K2.
* In the worst case complexity is O(n2)when the input size is doubled the complexity for O(2n) is 4n2 therefore the complexity for O(KN)is k2n2 hence the sort grows at the rate of K2.
* In the best case complexity is O(n)when the input size is doubled the complexity is 4n therefore the complexity for O(KN)is Kn hence in the best case the insertion sort grows at the rate of K.

**Merge sort**

**Psuedo Code**

Merge Sort(array A, int d, int f)

{

if (d < f)

{

e= (d + f)/2

Merge Sort (A, d, e)

Merge Sort (A, e+1, f)

Merge (A, d, e, f)

}

**}**

* In empirical analysis I plotted a graph for time taken by the merge sort to run for different cases which are average case, best case, worst case. For the average case the input size is from 100-4000 with an interval of 50 and for the best and worst cases the size was from 0-5000with an interval of 100.
* For average case the time complexity the is O(n log n) this algorithm follows a divide and conquer approach the input halved everytime and therefore the time to run the algorithm is O(n log n)
* For best and worst cases the time complexity is the same ie; O (n log n).

**Growth Order Function**

* The time complexity of merge sort in average case is O(n log n)

Complexity for O(2N) is O(2n log(2n))

Complexity for O(KN)is O(2n log(Kn))

by dividing the above two expressions, order of growth function O(KN)/ O(N)= K[log K +1],hence this sort grows at the rate of K[Log k+1] in the average case.

* In the merge sort, for both best and worst cases the order of growth function is similar to that of the average case.

**Comparison of sorts**

|  |  |  |  |
| --- | --- | --- | --- |
| Sort | Best Case | Worst Case | Average Case |
| Insertion Sort | O(n) | O(n2) | O(n2) |
| Selection Sort | O(n2) | O(n2) | O(n2) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| Bubble Sort | O(n) | O(n2) | O(n2) |

* When we consider small data sets insertion and selection sorts are more efficient when compared to merge sort
* But, when we consider large data sets merge is very efficient compared to the other sorts.
* Overall, merge sort is the most efficient algorithm as it follows a divide and conquer approach therefore its time complexity is less than the other sort algorithms.
* Though bubble sort is really simple, its impractical for most of the problems as the comparisons made are very high.