Design and Analysis of Algorithms (CS206)

Assignment - 2

U19CS012

1. Given the following algorithms, answer the questions.

• Insertion sort: Sorting Problem

Input: A Sequence of n numbers, **a1,a2,…,an**

Output: A Permutation (Reordering) (**a1',a2',…,an'**) of Input Sequence such that **a1'≤ a2'≤ … ≤ an'**

1.1. (T) Analyze the time complexity of above algorithms using the RAM model

• Lets Analyze Insertion Sort

• The time taken to sort depends on the fact that we are sorting how many numbers

• Also, the time to sort may change depending upon whether the *array is almost sorted* (can you see if the array was sorted we had very little job).

• So, we need to define the meaning of the **input size** and **running time**.

In Sorting Problem,

Input Size = Number of Integers we are Sorting
Running Time = Proportional to the Number of Operations Performed

U19CS012

A) Running Time of Insertion Sort [RAM Model Analysis]

STEPS	COST	TIMES
for $j = 1$ to $n-1$	c_1	n
key = A[j]	c_2	$n-1$
$i = j-1$	c_3	$n-1$
while $i > 0$ and A[i] > key	c_4	$\sum_{j=1}^{n-1}(t_j)$
A[i+1] = A[i]	c_5	$\sum_{j=1}^{n-1}(t_j - 1)$
$i = i-1$	c_6	$\sum_{j=1}^{n-1}(t_j - 1)$
A[i+1] = key	c_7	$n-1$

In RAM Model, the total time is the sum of that for each statement.

$$\boxed{T(n) = c_1(n) + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1}(t_j) + c_5\sum_{j=1}^{n-1}(t_j-1) + c_6\sum_{j=1}^{n-1}(t_j-1) + c_7(n-1)}$$

(A) BEST CASE: If the array is already sorted (while loop see's in only 1 check that A[i] < key so while loop terminates. Thus $\boxed{t_j = 1}$ and

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1}(1) + c_5\sum_{j=1}^{n-1}(1-1) + c_6\sum_{j=1}^{n-1}(1-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$$= \boxed{O(n)} = \text{Linear function of } (n) = a(n) + b, \ (a, b \in \mathbb{R}^{+}) \text{ constants}$$

(B) WORST CASE: If the array is reverse sorted array:
- While loop requires comparision with A[j-1] to A[0], i.e $\boxed{t_j = j}$

$$T(n) = c_1(n) + (c_2+c_3)(n-1) + c_4\sum_{j=1}^{n-1}(j) + (c_5+c_6)\sum_{j=1}^{n-1}(j-1) + c_7(n-1)$$

$$= \left(\frac{c_4+c_5+c_6}{2}\right)n^2 + \left(c_1+c_2+c_3-\frac{c_4}{2}-\frac{3c_5}{2}-\frac{3c_6}{2}\right)n + (c_5+c_6-c_2-c_3-c_7)$$

$$= an^2 + bn + c, \quad a, b, c \rightarrow \text{constanst} = \text{quadratic function of } n$$

U19CS012

ⓒ AVERAGE CASE : All the input of particular size are equally probable

If the element of Array [0 to j-1] are randomly chosen We can assume that half of the elements are greater then A[j] while half are less. ie $t_j = (j/2)$

$$T(n) = c_1(n) + (c_2+c_3)(n-1) + c_4\sum_{j=1}^{n-1}\left(\frac{j}{2}\right) + (c_5+c_6)\sum_{j=1}^{n-1}\left(\frac{j}{2}-1\right) + c_7(n-1)$$

$(c_5+c_6)(n-1)$

$$= (c_1+c_2+c_3+c_7)n + (c_2+c_3-c_7) + c_4\frac{n(n-1)}{(4)} + (c_5+c_6)\frac{n(n-1)}{4} - \wedge$$

$$= \left(c_1+c_2+c_3+c_7 + -\frac{c_4}{4} - \frac{c_5}{4} - \frac{c_6}{4} + c_5 + c_6\right)n + (c_2+c_3-c_7-c_5-c_6)$$

$$+ \left(\frac{c_4}{4}+\frac{c_5}{4}+\frac{c_6}{4}\right)n^2$$

$= \underline{an^2+bn+c}$, $a, b, c \in$ constant

= quadratic type

$= \underline{O(n^2)}$ ≈ <u>worst case</u> run time of algorithm

In some cases, average case may <u>tilt towards Best Case</u>.

1.2. (L) Implement the above algorithms using the programming language of your choice.

```
• Insertion-sort(A)

for j=1 to (length(A)-1)
key = A[j]
// Insert A[j] into the sorted sequnce A[0...j-1]
i=j-1
while i>0 and A[i]>key
     A[i+1]=A[i]
     i=i-1
//Since A[i]<=key, so we place key on the right side of A[i]
A[i+1]=key
```

1.3. (L) Provide the details of Hardware/Software you used to implement algorithms and to measure the time.

Hardware Details of My Laptop:

PARAMETER	LAPTOP CONFIGURATION
Operating System	Microsoft Windows **10**.0.19042
Processor	Intel(R) Core(TM) i5-10210U [Core **i5 10th Gen**]
CPU	**1.60GHz**, 2112 Mhz, **4** Core(s), 8 Logical Processor(s)
System Type	x64-based PC [**64 Bit**]
RAM	**8.00** GB
Hard Drive/SSD	512 GB **SSD**

Software Used:

PARAMETER	LAPTOP CONFIGURATION
Code Editor	**Visual Studio** Code [Version 1.52]
Compiler	gcc (MinGW.org **GCC-8.2.0-5**) 8.2.0
Time	Measured using **chrono** Library in C++
Programming Language Used	**C++**

1.4. (L) Submit the code (complete programs).

```cpp
// HEADERS AND NAMESPACE
#include <bits/stdc++.h>
// INSTEAD OF ALL THESE
#include <iostream>
// For Creating File
#include <fstream>
#include <vector>
// For set - precision
#include <iomanip>
// For Time Calculation
#include <chrono>
// For File Name and Output File Name
#include <string>

using namespace std;
using namespace std::chrono;

// COMMONLY USED TYPES
typedef long long ll;
typedef vector<ll> vll;

// Basic Algorithm Implementation of Insertion Sort
void insertion_sort(vll &arr)
{
    ll sz = arr.size(), key, i, j;

    for (j = 1; j < sz; j++)
    {
        key = arr[j];
        // Insert arr[j] into sorted sequence A[0...j-1]
        i = j - 1;
        while (i >= 0 && arr[i] > key)
        {
            arr[i + 1] = arr[i];
            i = i - 1;
        }
        // Since A[i]<=key, so we place key on right side of arr[i]
        arr[i + 1] = key;
    }

    return;
}

int main()
{
    // For Read & Write from "Input File" and  Return Output to "Output" File
    freopen("output.txt", "a+", stdout);
```

```cpp
// EDIT THIS FILE NUMBER , LIMIT and Number of Times File Runs
int file_no = 1;
int limit = 5;
int each_file_runs = 2;

for (; file_no <= limit; file_no++)
{
    string inp_file = "File";
    string num = to_string(file_no);
    string ext = ".txt";
    inp_file += num;
    inp_file += ext;

    ifstream File;
    File.open(inp_file);

    vector<ll> arr;

    ll number, idx = 0;
    while (!File.eof())
    {
        File >> number;
        arr.push_back(number);
    }

    ll Best_Duration = 0, Worst_Duration = 0, Average_Duration = 0;
    auto start = high_resolution_clock::now();
    auto end = high_resolution_clock::now();
    auto time_taken = duration_cast<nanoseconds>(end - start);
    for (int f = 0; f < each_file_runs; f++)
    {
        // ------------------------AVERAGE CASE [O(n^2)]----------------------------

        start = high_resolution_clock::now();
        // Function Here
        insertion_sort(arr);
        // Function Ends here
        end = high_resolution_clock::now();
        time_taken = duration_cast<nanoseconds>(end - start);
        Average_Duration += time_taken.count();

        // ------------------------BEST CASE [0(n)]----------------------------
        // The Array is Already Sorted from Average Case, So it Becomes out Best Case
        // sort(arr.begin(), arr.end());
        start = high_resolution_clock::now();
        // Function Here
        insertion_sort(arr);
        // Function Ends here
        end = high_resolution_clock::now();
```

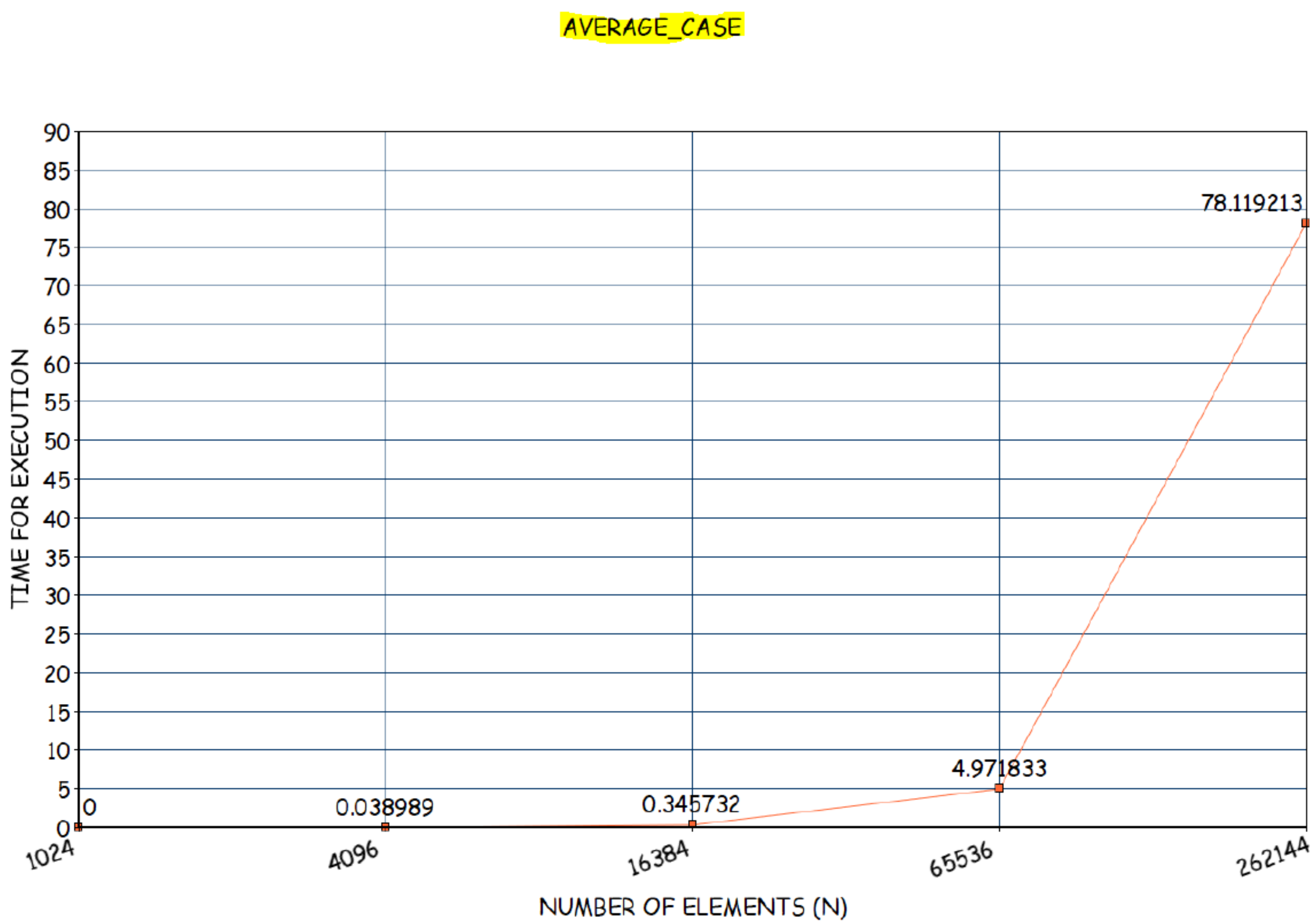
```
            time_taken = duration_cast<nanoseconds>(end - start);
            Best_Duration += time_taken.count();

            // ------------------------WORST CASE [0(n^2)]----------------------------
            // This will Reverse the Sorted Array, Therfore we will Get the Worst Case

            reverse(arr.begin(), arr.end());
            // sort(arr.begin(), arr.end(), greater<ll>());
            start = high_resolution_clock::now();
            // Function Here
            insertion_sort(arr);
            // Function Ends here
            end = high_resolution_clock::now();
            time_taken = duration_cast<nanoseconds>(end - start);
            Worst_Duration += time_taken.count();
        }

        cout << "---------------------------------------------------------" << endl;
        cout << inp_file << endl;
        cout << "AVERAGE CASE : ";
        double avg = (double)Average_Duration / (double)each_file_runs;
        avg *= 1e-9;
        cout << fixed << avg << setprecision(9);
        cout << " seconds" << endl;
        cout << "BEST CASE    : ";
        double best = (double)Best_Duration / (double)each_file_runs;
        best *= 1e-9;
        cout << fixed << best << setprecision(9);
        cout << " seconds" << endl;
        cout << "WORST CASE   : ";
        double worst = (double)Worst_Duration / (double)each_file_runs;
        worst *= 1e-9;
        cout << fixed << worst << setprecision(9);
        cout << " seconds" << endl;
    }

    return 0;
}
```

1.5. (L) Measure the average-case time (considering current data of ten files) of insertion sort for all ten files. Plot a graph.

INSERTION SORT ALGORITHM

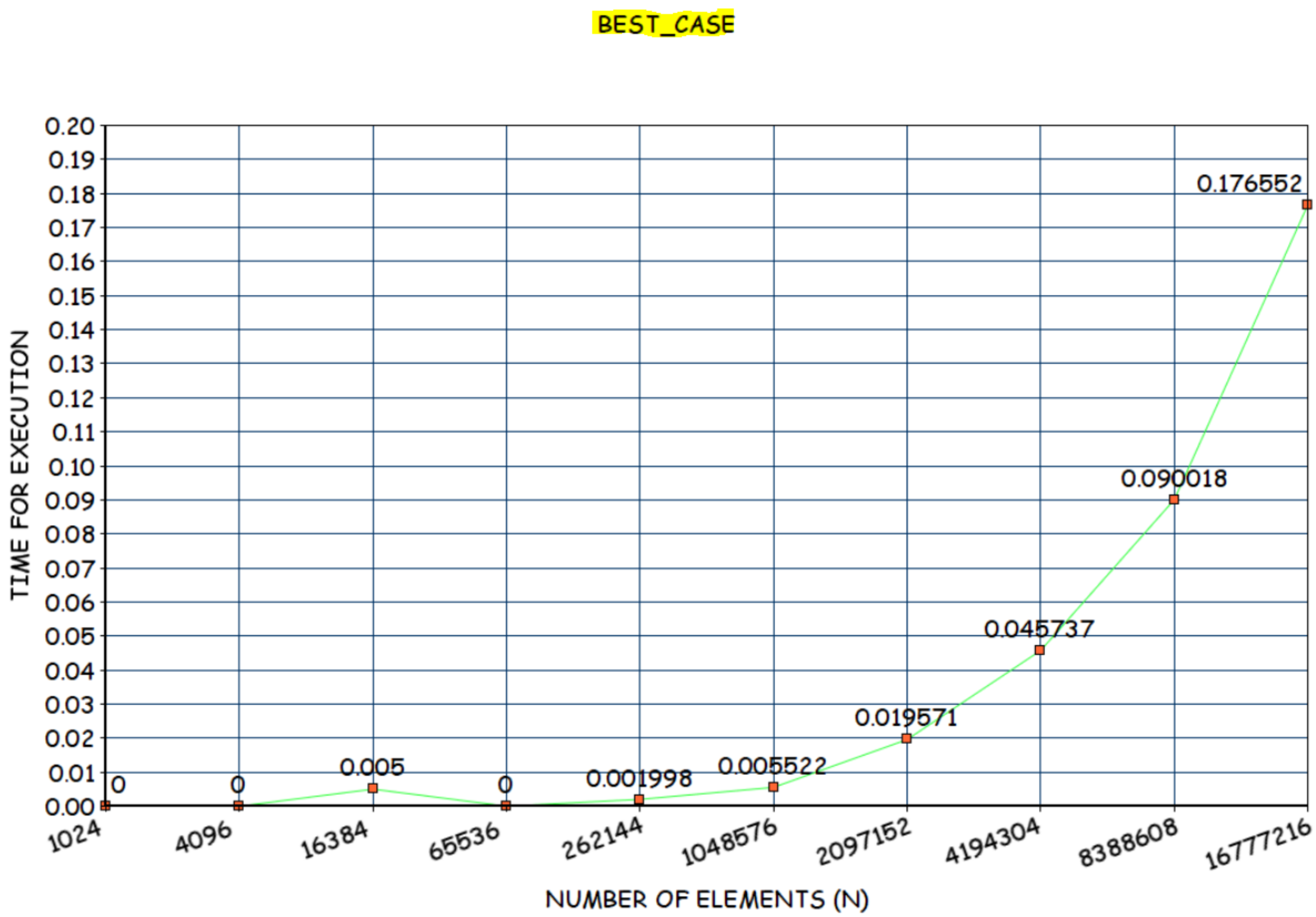
FILE	Number of Elements	AVERAGE CASE [in sec]
1	1024 = 2^10	0.000000000
2	4096 = 2^12	0.038989000
3	16384 = 2^14	0.345732500
4	65536 = 2^16	4.971833000
5	262144 = 2^18	78.119213000

After File 5 Onwards, It would take a Minimum of 2 hrs for Each File Execution. So Avoided Executing for Rest of the Files.



1.6. (L) Measure the best-case time of insertion sort for all ten files. Plot a graph

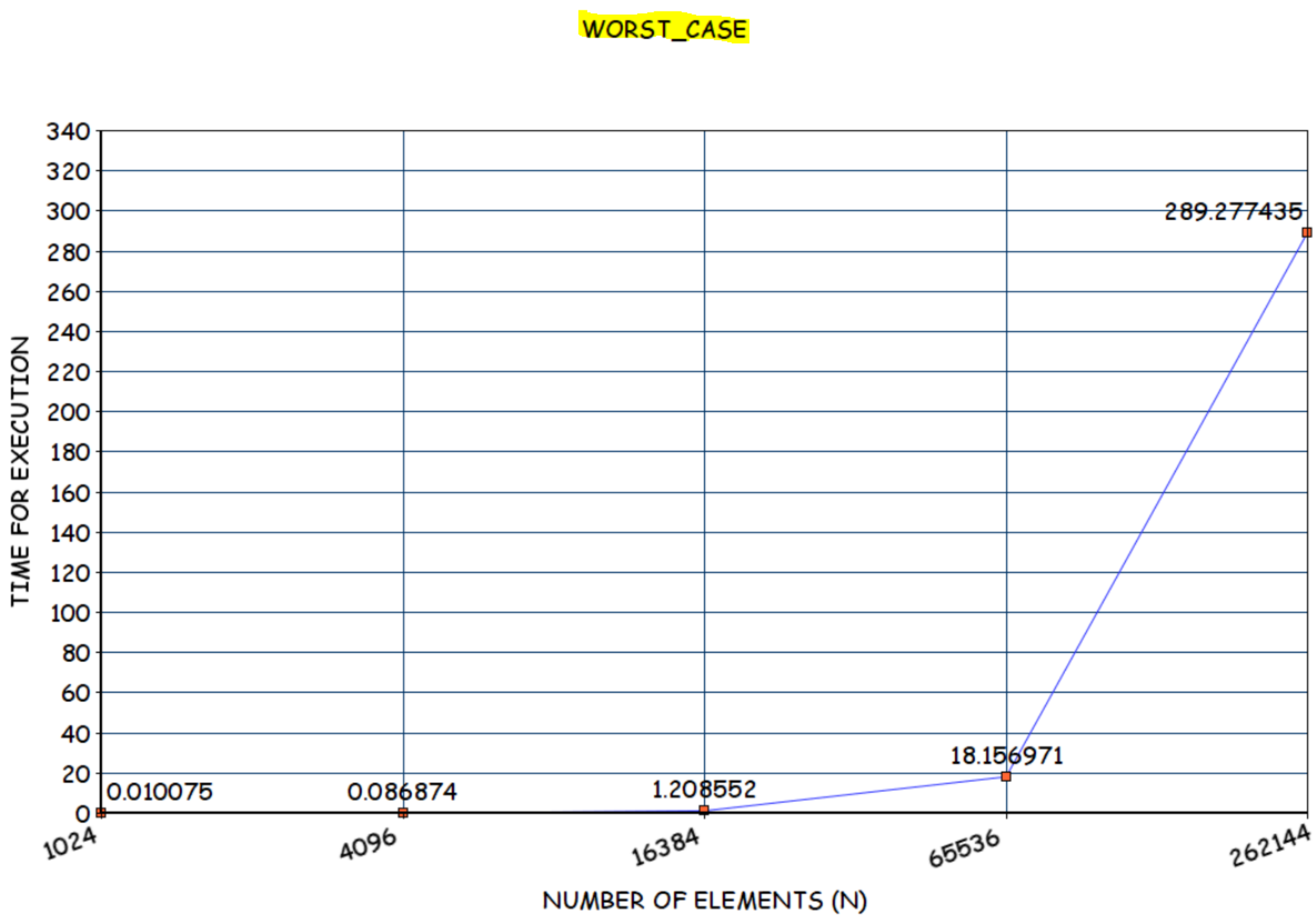
FILE	Number of Elements	BEST CASE [in sec]
1	1024 = 2^10	0.000000000
2	4096 = 2^12	0.000000000
3	16384 = 2^14	0.005000000
4	65536 = 2^16	0.000000000
5	262144 = 2^18	0.001998500
6	1048576 = 2^20	0.005522500
7	2097152 = 2^21	0.019571000
8	4194304 = 2^22	0.045737000
9	8388608 = 2^23	0.090017500
10	16777216 = 2^24	0.176551500



1.7. (L) Measure the worst-case time of insertion sort for all ten files. Plot a graph.

FILE	Number of Elements	WORST CASE [in sec]
1	1024 = 2^10	0.010075500
2	4096 = 2^12	0.086874000
3	16384 = 2^14	1.208551500
4	65536 = 2^16	18.156971000
5	262144 = 2^18	289.277434500

After File 5 Onwards, It would take a <u>Minimum of 3 hrs</u> for Each File Execution. So Avoided Executing for Rest of the Files.



1.8. (T) Assume that you don't know the time complexity of above algorithms.

1.8.1. Can you predict the same based on your implementation?

Definitely Yes.

Since 1 sec takes 10^8 Operations [Approximation]

X sec takes '?' Operations

So From Time Taken we can get the Number of Operations it performs.

Eg:

No of Operations [in File 10 Best Case] = 0.176551500 * (10^8) = 17655150

= [Approximately Equal to 16777216 = 2^24 = N]

= O(N)

Therefore, Time Complexity for **Best Case** [Prediction] = **O(N)**

1.8.2. Do they match with theoretical time complexity? **Yes**~~/No.~~

1.8.3. If yes, then write the time complexity of algorithm. If no, then write the difference.

Time Complexity of Insertion Sort

BEST CASE = If the Array is Already Sorted = O(N)

Running Time is Linear Function of N

WORST CASE = If the Array is Reverse Sorted = O(N^2)

Running Time is Quadratic Function of N

AVERAGE CASE = O(N^2) [Approximately]

Instead of Input of Particular Type [Sorted or Reverse Sorted]

, All the Inputs of Given Sizes are Equally Probable

If First Half , We can assume that half the elements are greater than A[j] while half are less.

On the average, thus tj=j/2. [In RAM Model]
Plugging this value into T(n) [RAM Model Equation] still leaves it Quadratic.

Thus, in this case Average case is Equivalent to Worst Case Time Complexity.

Remark : Since the Input is Random, Average Case may Tilt Towards Best Case as well.

BEST CASE [THEORATICAL CALCULATION]

FILE	NUMBER OF ELEMENTS	NO OF OPERATIONS [CASE] = O(N)	APPROX TIME TAKEN [OP/10^8]
FILE 1	1024 = 2^10	1024	0.00001024
FILE 2	4096 = 2^12	4096	0.00004096
FILE 3	16384 = 2^14	16384	0.00016384
FILE 4	65536 = 2^16	65536	0.00065536
FILE 5	262144 = 2^18	262144	0.00262144
FILE 6	1048576 = 2^20	1048576	0.01048576
FILE 7	2097152 = 2^21	2097152	0.02097152
FILE 8	4194304 = 2^22	4194304	0.04194304
FILE 9	8388608 = 2^23	8388608	0.08388608
FILE 10	16777216 = 2^24	16777216	0.16777216

WORST/AVERAGE CASE [THEORATICAL CALCULATION]

FILE	NUMBER OF ELEMENTS	NO OF OPERATIONS [CASE] = O(N^2)	APPROX TIME TAKEN [OP/10^8]
FILE 1	1024 = 2^10	2^20	0.0104 seconds = 0.01 sec
FILE 2	4096 = 2^12	2^24	0.167 seconds = 0.16 sec
FILE 3	16384 = 2^14	2^28	2.684 seconds = 2.6 sec
FILE 4	65536 = 2^16	2^32	43 seconds = 43 sec
FILE 5	262144 = 2^18	2^36	687 seconds = 11 mins
FILE 6	1048576 = 2^20	2^40	10995 seconds = 3 hrs 3 mins
FILE 7	2097152 = 2^21	2^42	43980 seconds = 12 hrs 13 mins
FILE 8	4194304 = 2^22	2^44	175922 seconds = 2 days 52 hrs 2 mins
FILE 9	8388608 = 2^23	2^46	703687 seconds = 8 days 3 hrs 28 mins
FILE 10	16777216 = 2^24	2^48	2814750 seconds = 32 days 13 hrs 52 mins

CONCLUSION:

1.) Efficient for sorting small numbers

2.) In place sort: Takes an array A[0..n-1] (sequence of n elements) and arranges them in place, so that it is sorted.

3.) Maintains relative order of the input data in case of two equal values (stable) & Algorithm is Also Adaptive.

SUBMITTED BY:

U19CS012

BHAGYA VINOD RANA