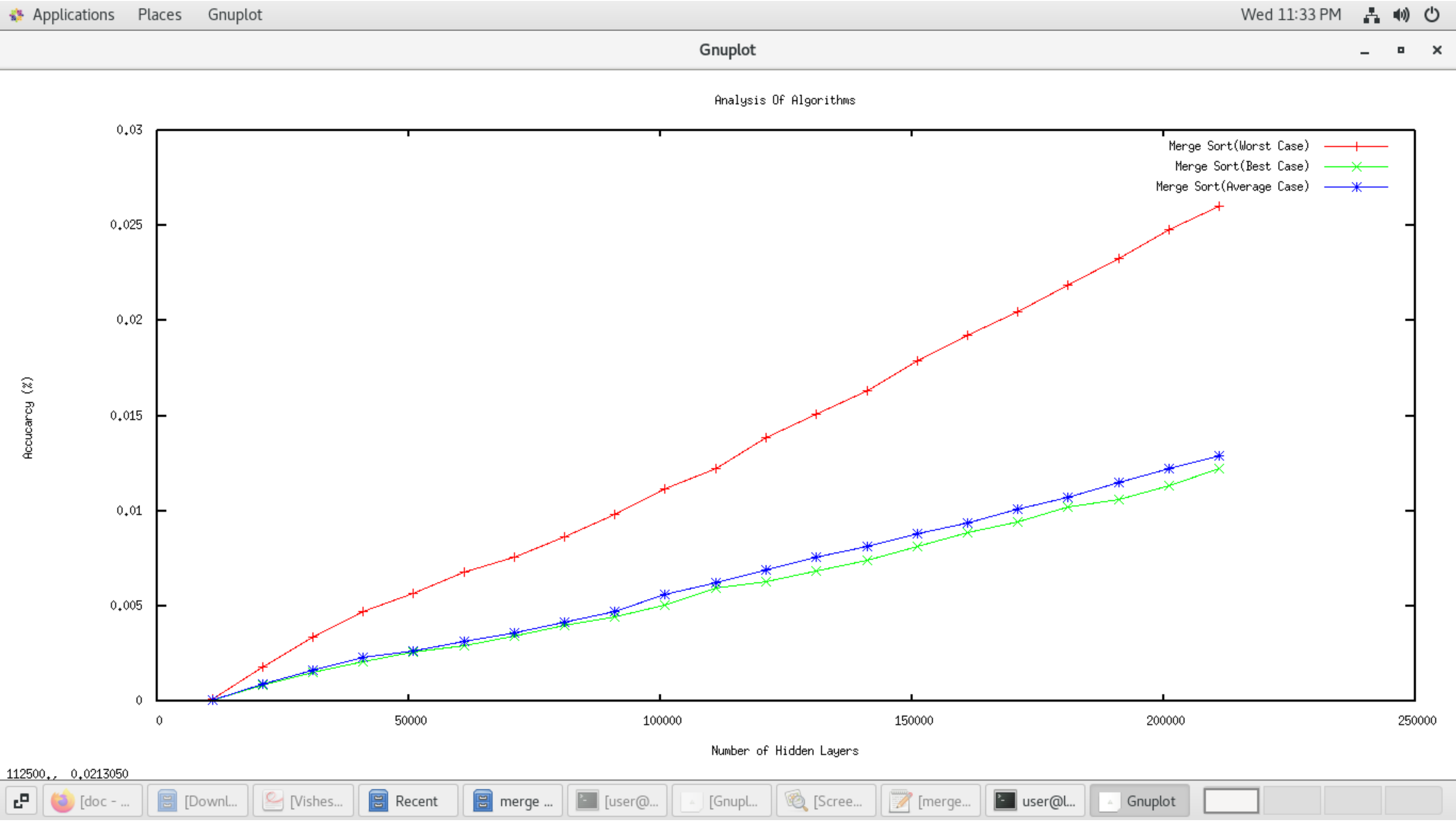
NAME:- Sourav Paul
GROUP:- D2
REG. NO:- 20214056
BRANCH:- CSE DEPT.

ASSIGNMENT - 02

1). Write a C Program to analyse the time complexity of Merge Sort Algorithm. Also plot its graph for all cases.

```
Ans:-  #include<stdio.h>
#include<stdlib.h>
#include<time.h>

// utility functions
void reverse(long int arr[],  long int n)
{
   long int i, temp;
  for(i = 0; i < n / 2; i++)
  {
    temp = arr[i];
    arr[i] = arr[n - 1 - i];
    arr[n - 1 - i] = temp;
  }
}

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(long int arr[], long int l, long int m, long int r)
{
    long int i, j, k;
    long int n1 = m - l + 1;
    long int n2 = r - m;

    // create temp arrays
     long int Left[n1], Right[n2];

    // Copy data to temp arrays Left[] and Right[]
    for (i = 0; i < n1; i++)
        Left[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        Right[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (Left[i] <= Right[j]) {
            arr[k] = Left[i];
            i++;
        }
        else {
            arr[k] = Right[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of Left[], if there are any
    while (i < n1) {
        arr[k] = Left[i];
        i++;
        k++;
    }

    // Copy the remaining elements of Right[], if there   are any
    while (j < n2) {
        arr[k] = Right[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */

// function for merge sort algorithm
void mergeSort( long int arr[], long int l, long int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
         long int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}



int main()
{
        FILE *fp = fopen("mergedata.txt", "w"); // write only
         // test for files not existing.
         if (fp == NULL)
          {
            printf("Error! Could not open file\n");
            exit(-1);
          }


        long int n = 1000;
        int temp = 0;

        double mergeAvg[21], mergeBest[21], mergeWorst[21]; //
array to store time duration of insertion sort algorithm
        printf("Array_Size Merge_Worst Merge_Best
Merge_Average\n");

        while(temp++ < 21)
        {
                long int num[n];
                 long int i;
                for(i = 0; i < n; i++)
                 {
                   long int number = rand() % n + 1;
                    num[i] = number;
                 }
                 clock_t start, end;
                  // using clock_t to store time  clock_t start, end;
                  start = clock();
                  mergeSort(num,0,n-1);
                  end = clock();
                  // WORST CASE FOR INSERTION SORT
                  mergeWorst[temp] = (double)(end - start) /
CLOCKS_PER_SEC;


                // BEST CASE FOR INSERTION SORT
                 start = clock();
  }

                  mergeSort(num,0,n-1);
                  end = clock();

                  mergeBest[temp] = (double)(end - start) /
CLOCKS_PER_SEC;

                // AVERAGE CASE FOR INSERTION SORT
                reverse(num,n);

                 start = clock();
                  mergeSort(num,0,n-1);
                  end = clock();

                  mergeAvg[temp] = (double)(end - start) /
CLOCKS_PER_SEC;



                  // type conversion to long int
                   // for plotting graph with integer values
         printf("%li %lf %lf %lf\n",
n,mergeWorst[temp],mergeBest[temp],mergeAvg[temp]);

         n += 10000;
    // write to file
         fprintf(fp,"%li %lf %lf %lf\n",
n,mergeWorst[temp],mergeBest[temp],mergeAvg[temp]);

}
        fclose(fp);
        return 0;
}
```

```
/*OUTPUT

Array_Size Merge_Worst Merge_Best Merge_Average
11000 0.000130 0.000063 0.000073
21000 0.001821 0.000852 0.000926
31000 0.003375 0.001537 0.001651
41000 0.004699 0.002099 0.002304
51000 0.005691 0.002622 0.002645
61000 0.006777 0.002942 0.003163
71000 0.007589 0.003423 0.003624
81000 0.008648 0.003977 0.004184
91000 0.009841 0.004469 0.004731
101000 0.011191 0.005082 0.005605
111000 0.012240 0.005947 0.006256
121000 0.013833 0.006293 0.006897
131000 0.015111 0.006852 0.007609
141000 0.016345 0.007410 0.008125
151000 0.017863 0.008145 0.008819
161000 0.019215 0.008879 0.009375
171000 0.020471 0.009444 0.010074
181000 0.021845 0.010237 0.010739
191000 0.023287 0.010617 0.011485
201000 0.024797 0.011326 0.012228
211000 0.025994 0.012211 0.012877

*/
```

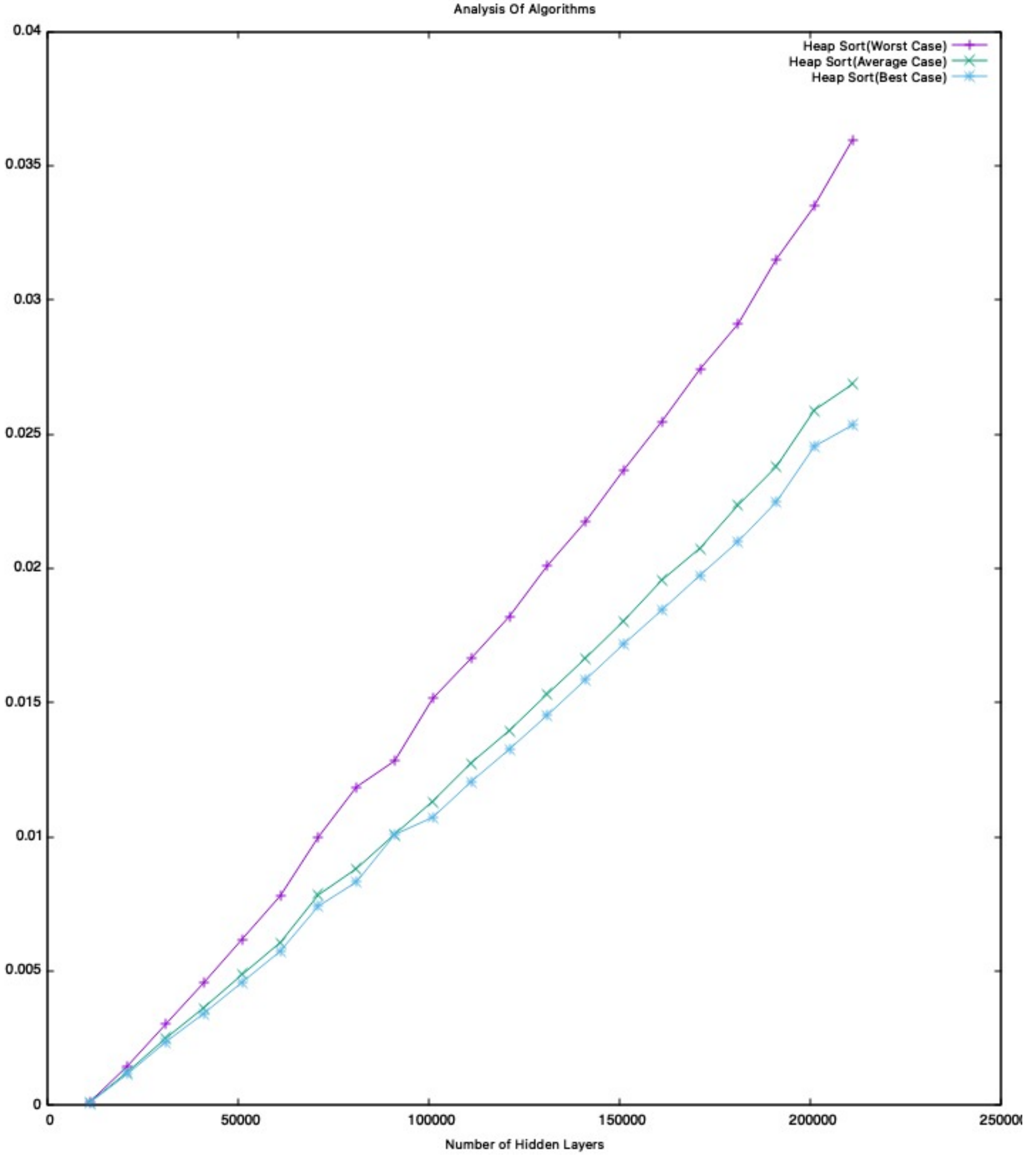


2). Write a C Program to analyse the time complexity of Heap Sort Algorithm. Also plot its graph for all cases.

Ans:-
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

// utility functions
void swap(long int* a, long int* b)
{

    long int temp = *a;

    *a = *b;
    *b = temp;
}

void reverse(long int arr[],  long int n)
{
   long int i, temp;
  for(i = 0; i < n / 2; i++)
  {
    temp = arr[i];
    arr[i] = arr[n - 1 - i];
    arr[n - 1 - i] = temp;
  }
}

// To heapify a subtree rooted with node i
// which is an index in arr[].
// n is size of heap
void heapify(long int arr[], long int N, long int i)
{
    // Find largest among root, left child and right child

    // Initialize largest as root
    long int largest = i;

    // left = 2*i + 1
    long int left = 2 * i + 1;

    // right = 2*i + 2
    long int right = 2 * i + 2;

    // If left child is larger than root
    if (left < N && arr[left] > arr[largest])

        largest = left;
    // If right child is larger than largest
    // so far
    if (right < N && arr[right] > arr[largest])

        largest = right;

    // Swap and continue heapifying if root is not largest
    // If largest is not root
    if (largest != i) {

        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(long int arr[], long int N)
{
 long int i;
    // Build max heap
    for (i = N / 2 - 1; i >= 0; i--)

        heapify(arr, N, i);

    // Heap sort
    for (int i = N - 1; i >= 0; i--) {

        swap(&arr[0], &arr[i]);

        // Heapify root element to get highest element at
        // root again
        heapify(arr, i, 0);
    }
```

```
}

int main()
{
        FILE *fp = fopen("heapdata.txt", "w"); // write only
         // test for files not existing.
         if (fp == NULL)
           {
             printf("Error! Could not open file\n");
             exit(-1);
           }


        long int n = 1000;
        int temp = 0;

        double heapAvg[21], heapBest[21], heapWorst[21]; // array
to store time duration of insertion sort algorithm
        printf("Array_Size Heap_Worst Heap_Average
Heap_Best\n");

        while(temp++ < 21)
        {
                long int num[n];
                 long int i;
                for(i = 0; i < n; i++)
                 {
                   long int number = rand() % n + 1;
                    num[i] = number;
                 }
                 clock_t start, end;
                  // using clock_t to store time  clock_t start, end;
                  start = clock();
                  heapSort(num,n);
                  end = clock();
                  // WORST CASE FOR INSERTION SORT
                  heapWorst[temp] = (double)(end - start) /
CLOCKS_PER_SEC;


                // AVERAGE CASE FOR INSERTION SORT
                 start = clock();
                  heapSort(num,n);
                  end = clock();

                  heapAvg[temp] = (double)(end - start) /
CLOCKS_PER_SEC;

                // BEST CASE FOR INSERTION SORT
                reverse(num,n);

                 start = clock();
                  heapSort(num,n);
                  end = clock();

                  heapBest[temp] = (double)(end - start) /
CLOCKS_PER_SEC;



                  // type conversion to long int
                   // for plotting graph with integer values
         printf("%li %lf %lf %lf\n",
n,heapWorst[temp],heapAvg[temp],heapWorst[temp]);

         n += 10000;
     // write to file
         fprintf(fp,"%li %lf %lf %lf\n",
n,heapWorst[temp],heapAvg[temp],heapWorst[temp]);

}
        fclose(fp);
```

```
        return 0;
}

/*OUTPUT
Array_Size Heap_Worst Heap_Average Heap_Best
11000 0.000102 0.000093 0.000086
21000 0.001465 0.001237 0.001167
31000 0.003016 0.002492 0.002355
41000 0.004563 0.003633 0.003418
51000 0.006184 0.004863 0.004580
61000 0.007798 0.006058 0.005735
71000 0.009992 0.007834 0.007415
81000 0.011859 0.008813 0.008318
91000 0.012828 0.010092 0.010082
101000 0.015155 0.011324 0.010720
111000 0.016643 0.012746 0.012048
121000 0.018198 0.013953 0.013258
131000 0.020098 0.015317 0.014529
141000 0.021737 0.016647 0.015850
151000 0.023660 0.018034 0.017172
161000 0.025469 0.019554 0.018437
171000 0.027425 0.020756 0.019745
181000 0.029106 0.022357 0.020998
191000 0.031517 0.023789 0.022469
201000 0.033491 0.025896 0.024547
211000 0.035940 0.026862 0.025337
```


Analysis Of Algorithms
Heap Sort(Worst Case)
Heap Sort(Average Case)
Heap Sort(Best Case)
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
50000
100000
150000
200000
25000
Number of Hidden Layers