Text

Description automatically generated



FACULTY OF ENGINEERING AND TECHNOLOGY BACHELOR OF TECHNOLOGY

Design and Analysis of Algorithm

(DAA) (203105318)

V SEMESTER

Computer Science & Engineering Department

**CERTIFICATE**

*This is to certify that*

*Mr.* **Varia Dhruv Prafulbhai** *with Enrolment No.* **210303105821** *has successfully completed his laboratory experiments in the subject (with Code)* **Design and Analysis of Algorithm (203105318)** *from the department of* **Computer Science and Engineering** *during the academic year* ***2022-2023.***



**Date of Submission …..…………… Staff In charge …..……………**

**Head of Department …..……………**

**TABLE OF CONTENT**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Sr.**  **No** | **Experiment Title** | **Page No** | | **Date of Start** | **Date of Completing** | **Sign** | **Marks**  **(Out of 10)** |
| **From** | **To** |
| 1 | Implementation and Time analysis of Bubble, Selection and Insertion sorting algorithms for best case, average case & worst case. |  |  |  |  |  |  |
| 2 | Implementation and Time analysis of Max-Heap sort algorithm. |  |  |  |  |  |  |
| 3 | Implementation and Time analysis of Merge Sort algorithms for best case, Average case &Worst-case using Divide and Conquer |  |  |  |  |  |  |
| 4 | Implementation and Time analysis of Quick-Sort algorithms for best case, Average case &Worst-case using Divide and Conquer. |  |  |  |  |  |  |
| 5 | Write a program to solve fractional knapsack problem. |  |  |  |  |  |  |
| 6 | Implementation and Time analysis of Kruskal’s Minimum spanning Tree algorithms. |  |  |  |  |  |  |
| 7 | Implementation and Time analysis of Prim’s Minimum spanning Tree algorithms |  |  |  |  |  |  |
| 8 | Write a program to solve 0-1 knapsack problem. |  |  |  |  |  |  |
| 9 | Implementation and Time analysis of Depth First Search (DFS) Graph Traversal and Breadth First Traversal |  |  |  |  |  |  |

**Practical : 1**

*Implementation and Time analysis of Bubble, Selection and Insertion sorting algorithms for best case, average case & worst case.*

**Code 1 : With Random Numbers**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Swap utility

void swap(long int\* a, long int\* b)

{

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

// Bubble sort

void bubbleSort(long int a[], long int n)

{

for (long int i = 0; i < n - 1; i++) {

for (long int j = 0; j < n - 1 - i; j++) {

if (a[j] > a[j + 1]) {

swap(&a[j], &a[j + 1]);

}

}

}

}

// Insertion sort

void insertionSort(long int arr[], long int n)

{

long int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

// Move elements of arr[0..i-1], that are

// greater than key, to one position ahead

// of their current position

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

// Selection sort

void selectionSort(long int arr[], long int n)

{

long int i, j, midx;

for (i = 0; i < n - 1; i++) {

// Find the minimum element in unsorted array

midx = i;

for (j = i + 1; j < n; j++)

if (arr[j] < arr[j])

midx = j;

// Swap the found minimum element

// with the first element

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

}

}

int main()

{

long int n = 10000;

int it = 0;

// Arrays to store time duration

// of sorting algorithms

double tim1[10], tim2[10], tim3[10];

printf("A\_size, Bubble, Insertion, Selection\n");

// Performs 10 iterations

while (it++ < 10) {

long int a[n], b[n], c[n];

// generating n random numbers

// storing them in arrays a, b, c

for (int i = 0; i < n; i++) {

long int no = rand() % n + 1;

a[i] = no;

b[i] = no;

c[i] = no;

}

// using clock\_t to store time

clock\_t start, end;

// Bubble sort

start = clock();

bubbleSort(a, n);

end = clock();

tim1[it] = ((double)(end - start))

// Insertion sort

start = clock();

insertionSort(b, n);

end = clock();

tim2[it] = ((double)(end - start));

// Selection sort

start = clock();

selectionSort(c, n);

end = clock();

tim3[it] = ((double)(end - start));

// type conversion to long int

// for plotting graph with integer values

printf("%li, %li, %li, %li\n",

n,

(long int)tim1[it],

(long int)tim2[it],

(long int)tim3[it]);

FILE \*fp=NULL;

fp=fopen("complexity.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim1[it]);

fp=fopen("complexity1.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim2[it]);

fp=fopen("complexity2.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim3[it]);

// increases the size of array by 10000

n += 10000;

}

// Code for auto run and load data into GNU Plot for graph….

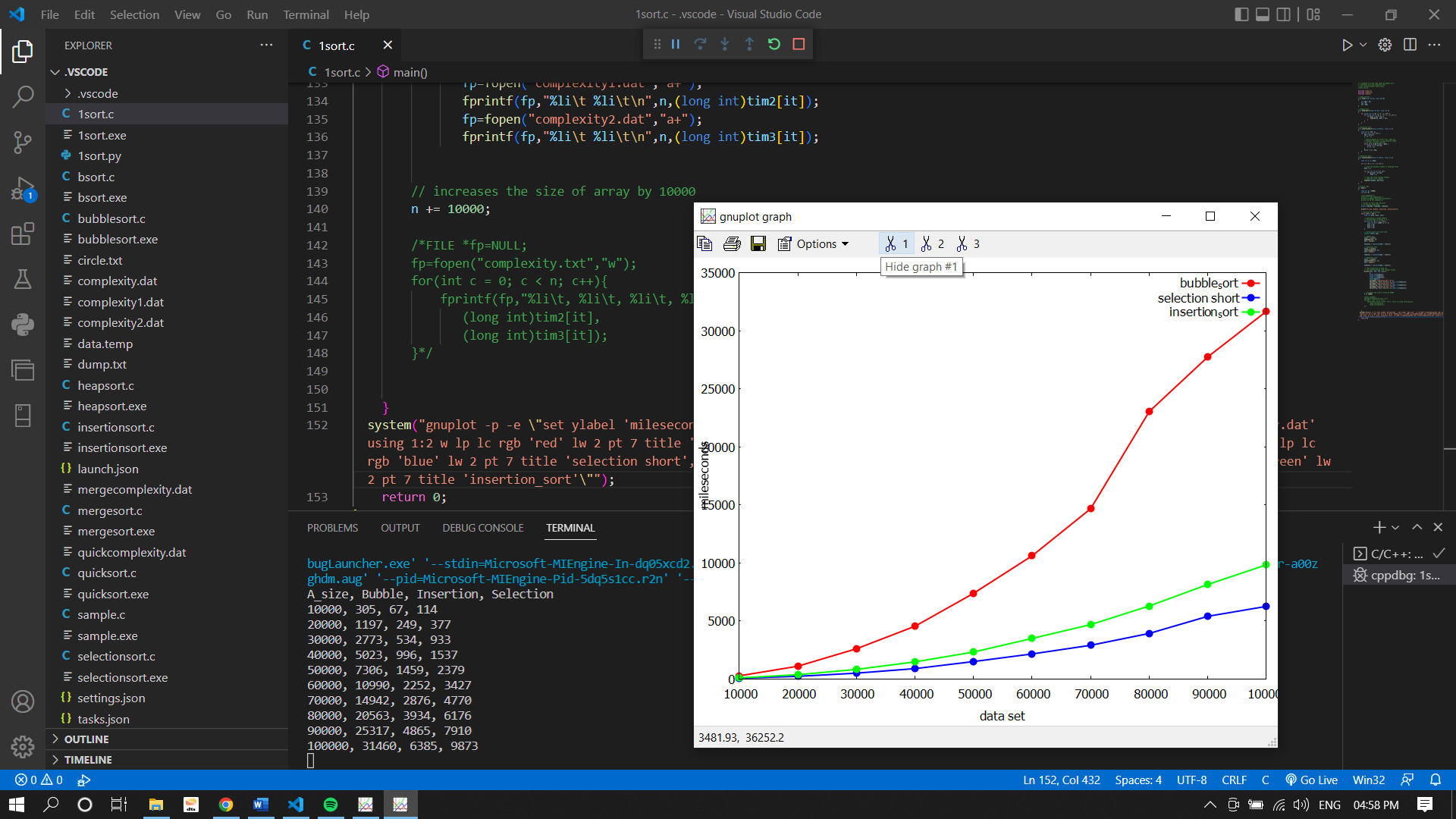
system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/complexity.dat' using 1:2 w lp lc rgb 'red' lw 2 pt 7 title 'bubble\_sort','D:/SEM 5 programming/DAA Lab/.vscode/complexity1.dat' using 1:2 w lp lc rgb 'blue' lw 2 pt 7 title 'selection short','D:/SEM 5 programming/DAA Lab/.vscode/complexity2.dat' using 1:2 w lp lc rgb 'green' lw 2 pt 7 title 'insertion\_sort'\"");

return 0;

}

**Output :**

**Avg Case:**

****

**Code 2 : With User input array**

#include<stdio.h>

#include<stdlib.h>

void display(int a[],int n);

void bubble\_sort(int a[],int n);

void selection\_sort(int a[],int n);

void insertion\_sort(int a[],int n);

void display(int arr[],int n)

{

for(int i=0;i<n;i++)

{

printf(" %d ",arr[i]);

}

}

void bubble\_sort(int arr[],int n)

{

int i,j,temp;

for(i=0;i<n;i++)

{

for(j=0;j<n-i-1;j++)

{

if(arr[j]>arr[j+1])

{

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

}

}

printf("After Bubble sort Elements are : ");

display(arr,n);

}

void selection\_sort(int arr[],int n)

{

int i,j,temp;

for(i=0;i<n-1;i++)

{

for(j=i+1;j<n;j++)

{

if(arr[i]>arr[j])

{

temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

}

printf("After Selection sort Elements are : ");

display(arr,n);

}

void insertion\_sort(int arr[],int n)

{

int i,j,min;

for(i=1;i<n;i++)

{

min=arr[i];

j=i-1;

while(min<arr[j] && j>=0)

{

arr[j+1]=arr[j];

j=j-1;

}

arr[j+1]=min;

}

printf("After Insertion sort Elements are : ");

display(arr,n);

}

int main()

{

int n,choice,i;

char ch[20];

printf("Enter no. of elements u want to sort : ");

scanf("%d",&n);

int arr[n];

printf("Enter %d Element : ",n);

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("\nPlease select any option Given Below for Sorting : \n");

while(1)

{

printf("\n1. Bubble Sort\n2. Selection Sort\n3. Insertion Sort\n4. Display Array.\n5. Exit the Program.@\_dv99\_,Enroll:821\n");

printf("\nEnter your Choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

bubble\_sort(arr,n);

break;

case 2:

selection\_sort(arr,n);

break;

case 3:

insertion\_sort(arr,n);

break;

case 4:

display(arr,n);

break;

case 5:

return 0;

default:

printf("\nPlease Select only 1-5 option ----\n");

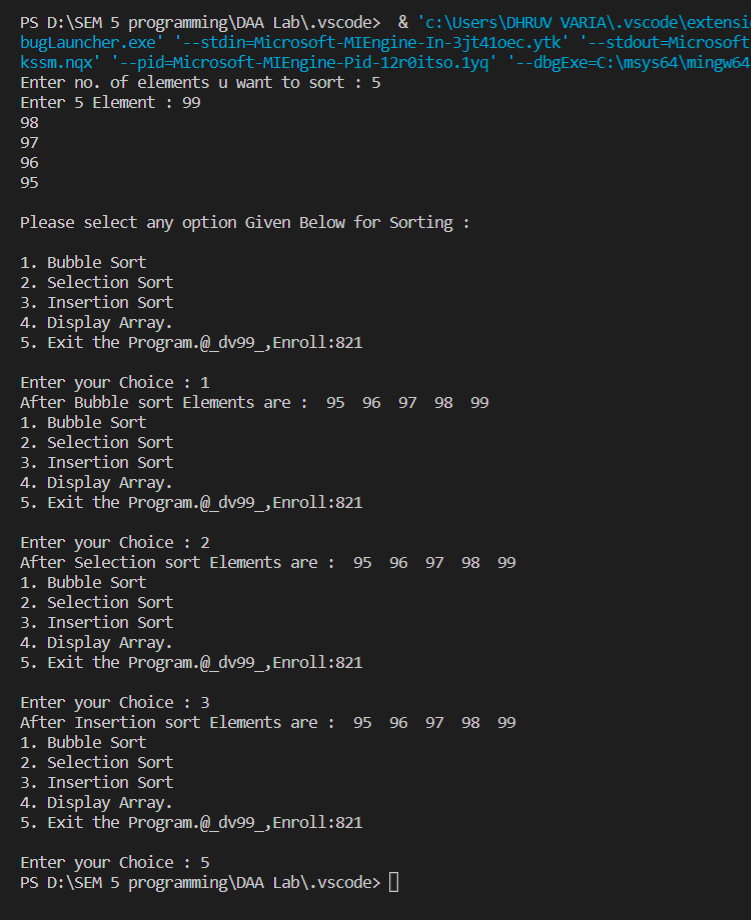
}

}

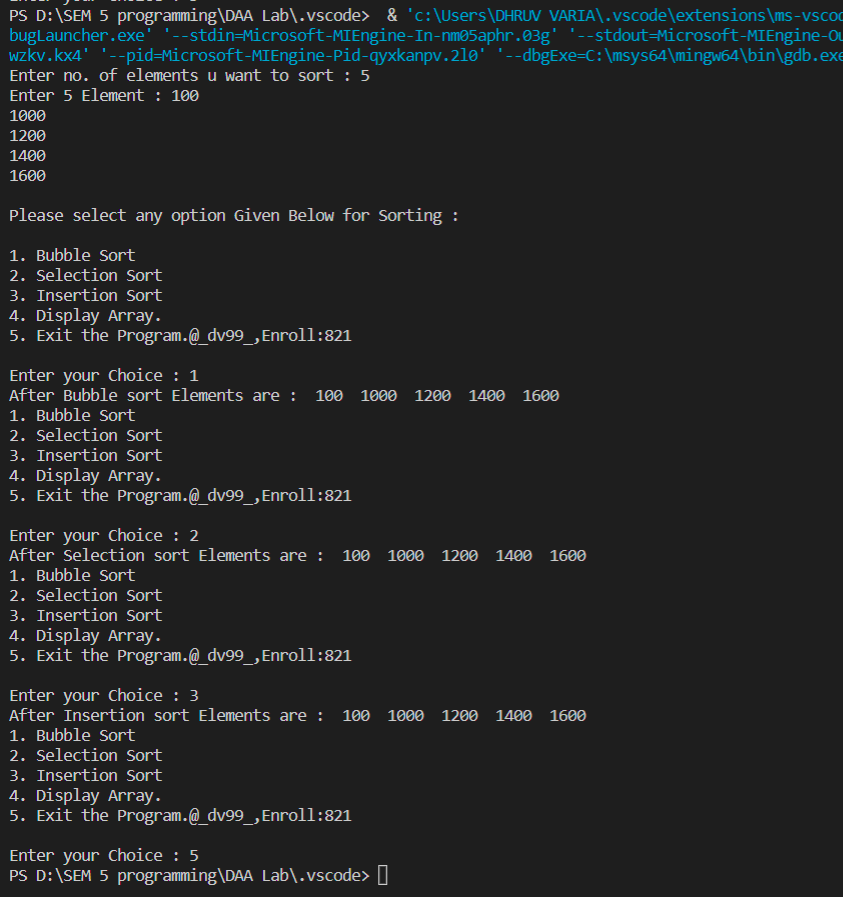
return 0;

}

**Output :** **Best Case**

****

**Worst Case**



**Practical : 2**

*Implementation and Time analysis of Max-Heap sort algorithm.*

**Code 1 : With Random Numbers**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

/\* function to heapify a subtree. Here 'i' is the

index of root node in array a[], and 'n' is the size of heap. \*/

void heapify(int a[], int n, int i)

{

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left child

int right = 2 \* i + 2; // right child

// If left child is larger than root

if (left < n && a[left] > a[largest])

largest = left;

// If right child is larger than root

if (right < n && a[right] > a[largest])

largest = right;

// If root is not largest

if (largest != i) {

// swap a[i] with a[largest]

int temp = a[i];

a[i] = a[largest];

a[largest] = temp;

heapify(a, n, largest);

}

}

void heapSort(int a[], int n)

{

for (int i = n / 2 - 1; i >= 0; i--)

heapify(a, n, i);

// One by one extract an element from heap

for (int i = n - 1; i >= 0; i--) {

/\* Move current root element to end\*/

// swap a[0] with a[i]

int temp = a[0];

a[0] = a[i];

a[i] = temp;

heapify(a, i, 0);

}

}

void printArr(int arr[], int n)

{

for (int i = 0; i < n; ++i)

{

printf("%d", arr[i]);

printf(" ");

}

}

int main()

{

// using clock\_t to store time

clock\_t start, end;

start = clock();

int h=15;

int it = 0;

double tim1[10];

int a[h];

//int a[] = {7,6,5,4,3,2,1};

for (int i = 0; i < h; i++)

{

long int no = rand() % h + 1;

a[i] = no;

}

int n = sizeof(a) / sizeof(a[0]);

printf("Before sorting array elements are - \n");

printArr(a, n);

heapSort(a, n);

printf("\nAfter sorting array elements are - \n");

printArr(a, n);

end = clock();

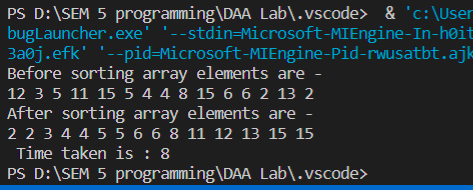
tim1[it] = ((double)(end - start));

printf("\n Time taken is : %li\n",(long int)tim1[it]);

return 0;

}

**Output :**



**Complexity :**

Time Complexity of this operation is O(Log n) because we insert the value at the end of the tree and traverse up to remove violated property of min/max heap.

1. Heapify()

In the heapify() function, we walk through the tree from top to bottom. The height of a binary tree (the root not being counted) of size n is log2 n at most.

The complexity for the heapify() function is accordingly O(log n).

1. Heapsort()

To initially build the heap, the heapify() method is called for each parent node – backward, starting with the last node and ending at the tree root.

A heap of size n has n/2 (rounded down) parent nodes:

Since the complexity of the heapify() method is O(log n) as shown above, the complexity for the heapsort() method is, therefore, maximum\* O(n log n).

The time complexity of Heapsort is : O(n log n)

**Practical : 3**

*Implementation and Time analysis of Merge Sort algorithms for Best case, Average case &Worst-case using Divide and Conquer.*

**Code 1 : With Random Numbers**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[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 (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r) {

// Same as (l+r)/2, but avoids overflow for

// large l and h

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);

}

}

void printArray(int A[], int size)

{

int i;

for (i = 0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

}

int main()

{

long int n = 10000;

int it = 0;

double tim1[10];

while (it++ < 10) {

long int arr[n];

for (int i = 0; i < n; i++)

{

long int no = rand() % n + 1;

arr[i] = no;

}

clock\_t start, end;

int arr\_size = sizeof(arr) / sizeof(arr[0]);

start = clock();

mergeSort(arr, 0, arr\_size - 1);

end = clock();

tim1[it] = ((double)(end - start));

// type conversion to long int

// for plotting graph with integer values

printf("%li %li \n", n,(long int)tim1[it]);

//For Saving in file. #Dhruv

FILE \*fp=NULL;

fp=fopen("mergecomplexity.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim1[it]);

n += 10000;

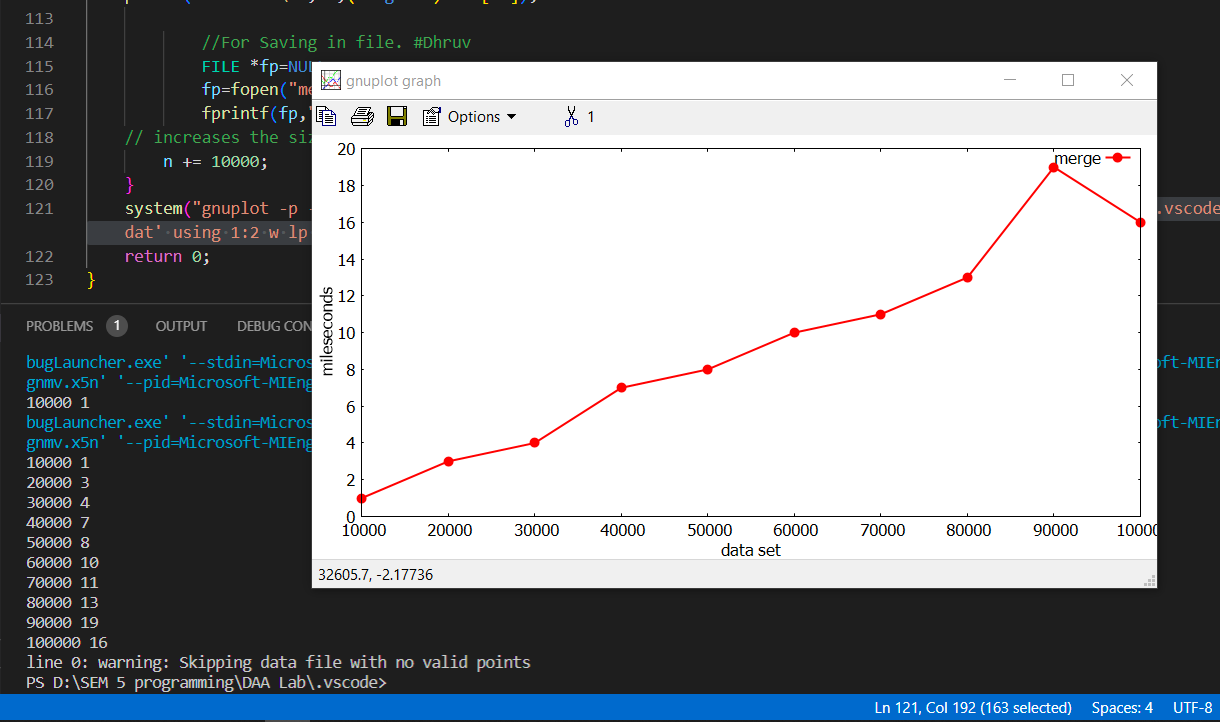
}

system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/mergecomplexity.dat' using 1:2 w lp lc rgb 'red' lw 2 pt 7 title 'merge'\"");

return 0;

}

**Output :**



**Code 2 : With User input array**

**/\* Functions are same as Code 1 \*/**

int main()

{

int i,n;

int it = 0;

printf("Enter no. of elements u want to sort : ");

scanf("%d",&n);

int arr[n];

printf("Enter %d Element : ",n);

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("\n Your array is : ");

for(int i=0;i<n;i++)

{

printf(", %d ",arr[i]);

}

int arr\_size = sizeof(arr) / sizeof(arr[0]);

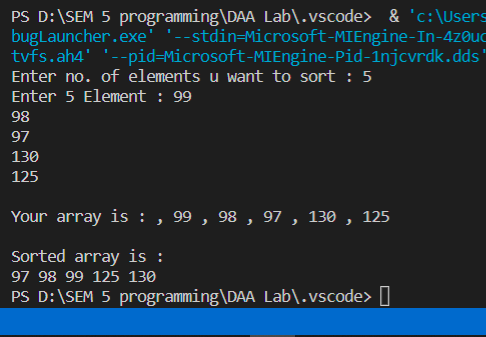
mergeSort(arr, 0, arr\_size - 1);

printf("\n\n Sorted array is : \n");

printArray(arr, arr\_size);

}

**Output :**

****

**Complexity :**

Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation. T(n) = 2T(n/2) + θ(n)

The solution of the above recurrence is O(nLogn). The list of size N is divided into a max of Logn parts, and the merging of all sublists into a single list takes O(N) time, the worst-case run time of this algorithm is O(nLogn)

Best Case Time Complexity: O(n\*log n)

Worst Case Time Complexity: O(n\*log n)

Average Time Complexity: O(n\*log n)

The time complexity of MergeSort is O(n\*Log n) in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

**Practical : 4**

*Implementation and Time analysis of Quick Sort algorithms for Best case, Average case & Worst-case using Divide and Conquer.*

**Code 1 : With Random Numbers**

#include<stdio.h>

#include <stdlib.h>

#include <time.h>

void quicksort(int arr[],int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(arr[i]<=arr[pivot]&&i<last)

i++;

while(arr[j]>arr[pivot])

j--;

if(i<j){

temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

temp=arr[pivot];

arr[pivot]=arr[j];

arr[j]=temp;

quicksort(arr,first,j-1);

quicksort(arr,j+1,last);

}

}

int main()

{

long int n = 10000;

int it = 0;

double tim1[10];

while (it++ < 10) {

long int arr[n];

// generating n random numbers

// storing them in arrays a, b, c

for (int i = 0; i < n; i++)

{

long int no = rand() % n + 1;

arr[i] = no;

}

clock\_t start, end;

int arr\_size = sizeof(arr) / sizeof(arr[0]);

start = clock();

quicksort(arr,0,arr\_size-1);

end = clock();

tim1[it] = ((double)(end - start));

// type conversion to long int

// for plotting graph with integer values

printf("%li %li \n", n,(long int)tim1[it]);

//For Saving in file. #Dhruv

FILE \*fp=NULL;

fp=fopen("quickcomplexity.dat","a+");

fprintf(fp,"%li\t %li\t\n",n,(long int)tim1[it]);

n += 10000;

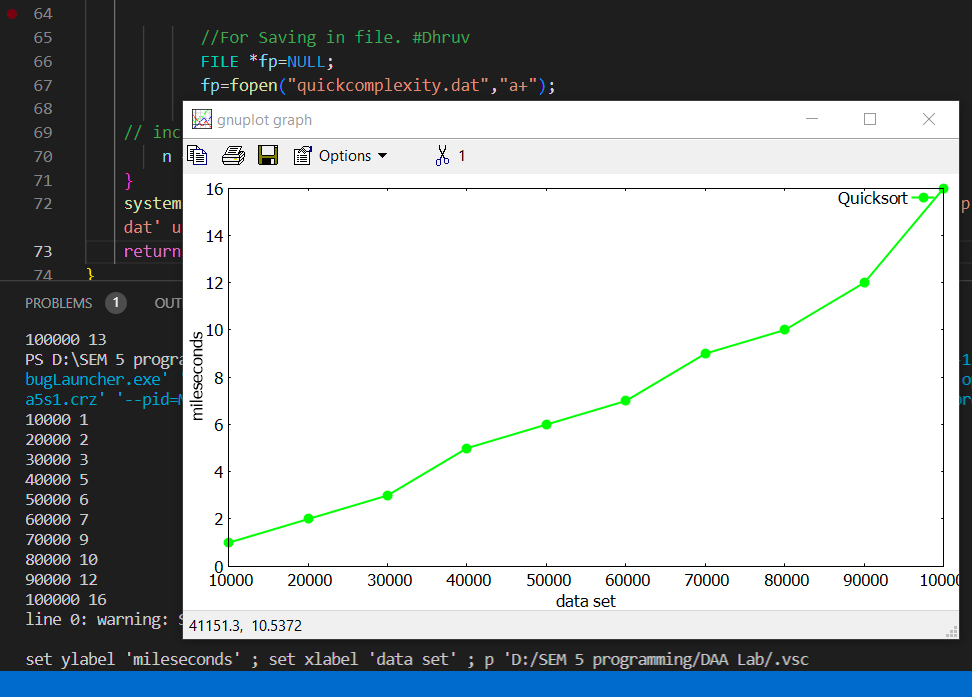
}

system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/quickcomplexity.dat' using 1:2 w lp lc rgb 'green' lw 2 pt 7 title 'Quicksort'\"");

return 0;

}

**Output :**

****

**Code 2 : With User input array**

**/\* Functions are same as Code 1 \*/**

int main()

{

int i,n;

int it = 0;

double tim1[10];

printf("Enter no. of elements u want to sort : ");

scanf("%d",&n);

int arr[n];

printf("Enter %d Element : ",n);

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("\nYour array is : ");

for(int i=0;i<n;i++)

{

printf(", %d ",arr[i]);

}

int arr\_size = sizeof(arr) / sizeof(arr[0]);

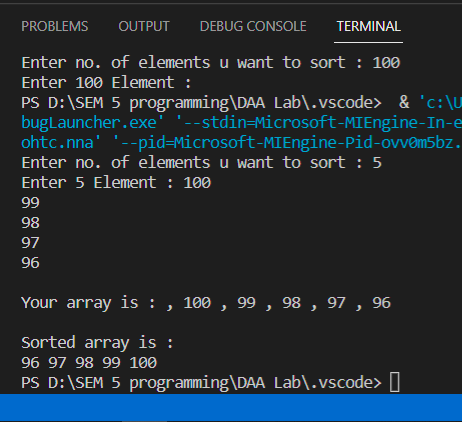
quicksort(arr,0,arr\_size-1);

printf("\n\nSorted array is : \n");

printArray(arr, arr\_size);

}

**Output :**



**Complexity :**

* **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n\*logn)**.
* **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n2)**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

**Practical : 5**

*Write a program to solve fractional knapsack problem.*

#include <stdio.h>

int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W

int knapsack(int W, int wt[], int val[], int n)

{

int i, w;

int K[n+1][W+1];

// Build table K[][] in bottom up manner

for (i = 0; i <= n; i++)

{

for (w = 0; w <= W; w++)

{

if (i==0 || w==0)

K[i][w] = 0;

else if (wt[i-1] <= w)

K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

else

K[i][w] = K[i-1][w];

}

}

return K[n][W];

}

int main()

{

int val[] = {20,30,66,40,60};

int wt[] = {10,20,30,40,50};

int W = 100;

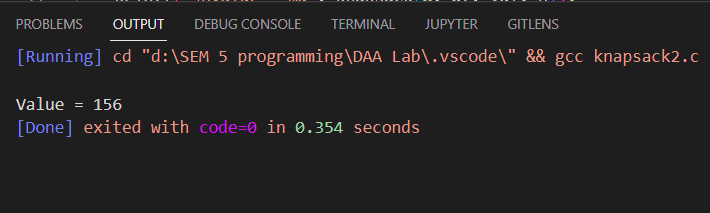
int n = sizeof(val)/sizeof(val[0]);

printf("\nValue = %d", knapsack(W, wt, val, n));

return 0;

}

**Output :**



**Practical : 6**

*Implementation and Time analysis of Krushkal’s Minimum spanning Tree algorithms*

#include <iostream>

#include <algorithm>

#include<bits/stdc++.h>

#include <chrono>

#include <ctime>

using namespace std;

using namespace std::chrono;

class Edge

{

public:

int source;

int dest;

int weight;

};

bool compare(Edge e1, Edge e2)

{

return e1.weight<e2.weight; //increasing order sort

}

int findParent(int v, int \*parent)

{

if( parent[v]==v)

{

return v;

}

else

{

return findParent(parent[v],parent);

}

}

void kruskals(Edge \*input, int n, int E)

{

//Sort the input array in ascending order based on weights.

//using inbuilt sort function. passing starting % end of array.

//we use our comparision for sort because array type is edge class, it will not compare by default e.g int, float values

sort(input, input+E, compare);

Edge \*output=new Edge[n-1]; //Output array for mst

int count=0;

int i=0;

int \*parent = new int[n]; //parent array for checking loops in mst

for (int i=0; i<n; i++)

{

parent[i]=i;

}

while (count!=n-1)

{

Edge currentEdge = input[i];

//check if we can add the current edge in mst or not

int sourceParent = findParent(currentEdge.source, parent);

int destParent = findParent(currentEdge.dest, parent);

if (sourceParent!=destParent)

{

output[count]=currentEdge;

count++;

parent[sourceParent]=destParent; //update parent to child in parrent array

}

i++;

}

cout<<endl<<"Output-"<<endl;

for(int i=0; i <n-1; i++)

{

if(output[i].source<output[i].dest)

{

cout<<output[i].source<<" "<<output[i].dest<<" "<<output[i].weight<<endl;

}

else

{

cout<<output[i].dest<<" "<<output[i].source<<" "<<output[i].weight<<endl;

}

}

}

int main()

{

int n, E,s,d,w; //n=total no vertices //E=total no edges //input array & its type is edge. no int,double...

cout<<"Enter number of vertices and edges:- "<<endl;

cin>>n>>E;

Edge \*input=new Edge[E];

cout<<"Source - Destination - Weight"<<endl;

for(int i=0; i<E; i++)

{

cin>>s>>d>>w;

input[i].source= s;

input[i].dest= d;

input[i].weight= w;

}

//Using Chrono library for high resolution clock to get time complexity

std::chrono::time\_point<std::chrono::system\_clock> start, end;

start = std::chrono::system\_clock::now();

kruskals(input, n, E);

end = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end - start;

std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);

std::cout << "finished computation at " << std::ctime(&end\_time)<< "elapsed time: " << elapsed\_seconds.count() << "s\n";

return 0;

}

/\* GRAPH:-

{2}

0---------------1

| - -|

| -{3} {2} - |

{4}| - - |{1}

| - - |

| - - |

| - {6} -|

2---------------3

| - -|

| -{10} {11}- |

{9}| - - |{1}

| - - |

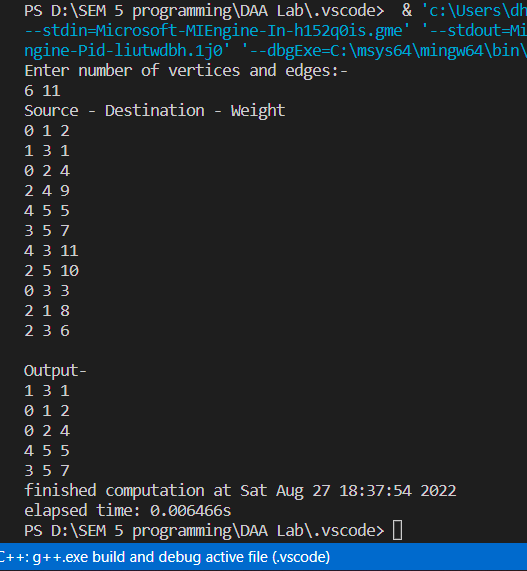
| - - |

| - {6} -|

4---------------5

\*/

**Output :**

****

**Practical : 7**

*Implementation and Time analysis of Prim’s Minimum spanning Tree algorithms*

#include<iostream>

#include <chrono>

#include <ctime>

using namespace std;

using namespace std::chrono;

// Number of vertices in the graph

const int V=6;

// Function to find the vertex with minimum key value

int min\_Key(int key[], bool visited[])

{

    int min = 999, min\_index;  // 999 represents an Infinite value

    for (int v = 0; v < V; v++) {

        if (visited[v] == false && key[v] < min) {

            // vertex should not be visited

            min = key[v];

            min\_index = v;

        }

    }

    return min\_index;

}

// Function to print the final MST stored in parent[]

int print\_MST(int parent[], int cost[V][V])

{

    int minCost=0;

    cout<<"Edge \tWeight\n";

    for (int i = 1; i< V; i++) {

        cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";

        minCost+=cost[i][parent[i]];

    }

    cout<<"Total cost is"<<minCost;

}

// Function to find the MST using adjacency cost matrix representation

void find\_MST(int cost[V][V])

{

    int parent[V], key[V];

    bool visited[V];

    // Initialize all the arrays

    for (int i = 0; i< V; i++) {

        key[i] = 999;    // 99 represents an Infinite value

        visited[i] = false;

        parent[i]=-1;

    }

    key[0] = 0;  // Include first vertex in MST by setting its key vaue to 0.

    parent[0] = -1; // First node is always root of MST

    // The MST will have maximum V-1 vertices

    for (int x = 0; x < V - 1; x++)

    {

        // Finding the minimum key vertex from the

        //set of vertices not yet included in MST

        int u = min\_Key(key, visited);

        visited[u] = true;  // Add the minimum key vertex to the MST

        // Update key and parent arrays

        for (int v = 0; v < V; v++)

        {

            // cost[u][v] is non zero only for adjacent vertices of u

            // visited[v] is false for vertices not yet included in MST

            // key[] gets updated only if cost[u][v] is smaller than key[v]

            if (cost[u][v]!=0 && visited[v] == false && cost[u][v] < key[v])

            {

                parent[v] = u;

                key[v] = cost[u][v];

            }

        }

    }

    // print the final MST

    print\_MST(parent, cost);

}

// main function

int main()

{

    int cost[V][V];

    cout<<"Enter the vertices for a graph with 6 vetices"<<endl;

    for (int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        {

            cin>>cost[i][j];

        }

    }

    std::chrono::time\_point<std::chrono::system\_clock> start, end;

    start = std::chrono::system\_clock::now();

    find\_MST(cost);

    end = std::chrono::system\_clock::now();

    std::chrono::duration<double> elapsed\_seconds = end - start;

    std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);

    std::cout << "finished computation at " << std::ctime(&end\_time)<< "elapsed time: " << elapsed\_seconds.count() << "s\n";

    return 0;

}

/\*  GRAPH:-

               {6}          {3}

          b-------------c----------d

         .|            .           |

     {4}. |           .            |

       .  |          .             |

      .   |{3}     .  {1}          |{2}

     a    |      .                 |

      .   |    .                   |

   {2} .  |  .                     |

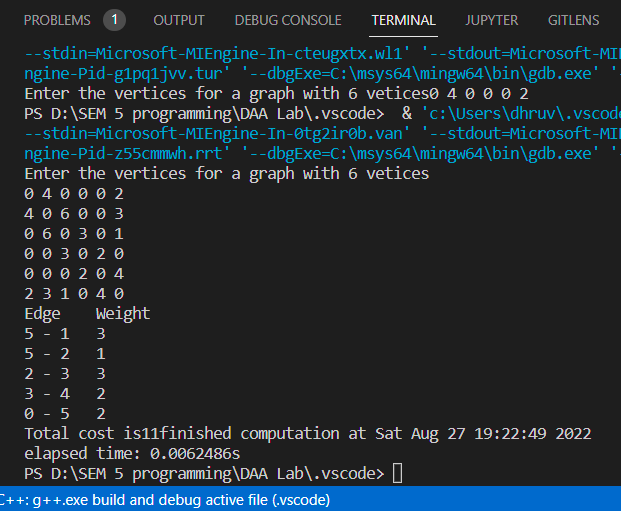
        . |.                       |

          f----------------------- e

                     {4}

\*/

**Output :**

**

**Practical : 8**

*Write a program to solve 0-1 knapsack problem.*

**CODE :-**

#include <bits/stdc++.h>

using namespace std;

void knapsack(int n,float weight[],float profit[],float capacity)

{

    float x[20],tp =0; //tp--->Total Profit

    int i,j,u;

    u = capacity;

    for(i =0 ; i<n ; i++)

    {

        x[i] = 0.0;

    }

    for(i =0 ; i<n ; i++)

    {

        if(weight[i]> u)

        {

            break;

        }

        else

        {

            x[i] = 1.0; // object is accepted

            tp = tp + profit[i]; // adding profit

            u = u - weight[i]; // remob\ving weigth in capiicity

        }

    }

    cout<<"\nThe Weight Vectoe is :- ";

    for(i =0 ; i<n ; i++){

        cout<<weight[i]<<"\t";

    }

    if(i<n)

        x[i] = u / weight[i];

        if (x[i] == 1)

            tp = tp +(x[i] \* profit[i]); //get int capecity of boj in 0.---

    cout<<"\nThe Result Vectoe is :- ";

    for(i =0 ; i<n ; i++){

        if(x[i] == 1)

            cout<<x[i]<<"\t";

        else

            cout<<"0"<<"\t";

    }

cout<<"\nMaxinum Profit is :- "<<tp;

}

int main()

{

    float weight[20], profit[20] , capacity;

    int num ,i ,j ,w = 0 ,p = 0;

    float ratio[20],temp;

    cout<<"\nEnter the number of objects :-";

    cin>>num;

    cout<<"\nEnter the weight and profit of each object :-\n";

    cout<<"Weight"<<"\t"<<"Profit"<<endl;

    for( i =0;i<num;i++)

    {

        cout<<i+1<<"-\n";

        cin>>weight[i];cout<<"\t";cin>>profit[i];

    }

    cout<<"\nTotal amouth of weigth :- ";

    for(i =0 ; i<num ; i++){

        w = w + weight[i];

    }

    cout<<w;

    cout<<"\nTotal amouth of profit :- ";

    for(i =0 ; i<num ; i++){

        p = p + profit[i];

    }

    cout<<p;

    cout<<"\nEnter the capacity of kanpsack:- ";

    cin>>capacity;

    for( i =0;i<num;i++)

    {

        ratio[i] = profit[i]/weight[i];

    }

    for( i =0 ;i<num;i++)

    {

        for(j =1+i;j<num; j++)

        {

            if(ratio[i]<ratio[j])

            {

                temp = ratio[j];

                ratio[j] = ratio[i];

                ratio[i] = temp;

                temp = weight[j];

                weight[j] = weight[i];

                weight[i] = temp;

                temp = profit[j];

                profit[j] = profit[i];

                profit[i] = temp;

            }

        }

    }

    cout<<"\nThe V / W  ratios is :- ";

    for(i =0 ; i<num ; i++){

        cout<<ratio[i]<<"\t";

    }

    knapsack(num,weight,profit,capacity);

    return 0;

}

/\*

5

1-

20

        50

2-

10

        20

3-

20

        90

4-

40

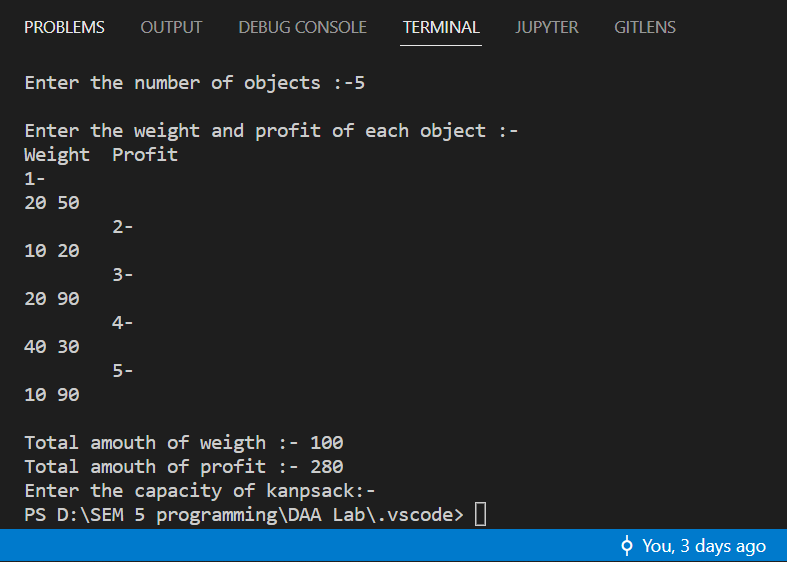
        30

5-

10

        90\*/

**Output :-**

****

**Practical : 9**

Implementation & Time Analysis of Depth First Search (DFS) Graph Traversal and Breadth First Traversal (BFS) Graph Traversal.

**CODE 1 :- BFS TRAVERSAL**

#include<bits/stdc++.h>

#include <algorithm>

#include<bits/stdc++.h>

#include <chrono>

#include <ctime>

#include <fstream>

using namespace std;

using namespace std::chrono;

const int N = 1e5+2;

bool vis[N];

vector<int> adj[N];

int main()

{

    for(int i =0; i<N; i++)

        vis[i] = 0;

    int n,m;

    cout<<"Number of nodes: ";

    cin>>n;

    cout<<"\n Number of edges: ";

    cin>>m;

    //Edge input;

    int x,y;

    for (int i = 0; i < m; i++)

    {

        cin >> x  >> y;

        adj[x].push\_back(y);

        adj[y].push\_back(x);

    }

    std::chrono::time\_point<std::chrono::system\_clock> start, end;

    start = std::chrono::system\_clock::now()

    queue<int> q;

    q.push(1);

    vis[1] = true;

    while(!q.empty())

    {

        int node = q.front();

        q.pop();

        cout<< node << " ";

        vector<int> :: iterator it;

        for(it= adj[node].begin() ; it != adj[node].end();it++)

        {

            if(!vis[\*it])

            {

                vis[\*it] = 1;

                q.push(\*it);

            }

        }

    }

    end = std::chrono::system\_clock::now();

    std::chrono::duration<double> elapsed\_seconds = end - start;

    std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);

    std::cout << "finished computation at " << std::ctime(&end\_time)<< "elapsed time: " << elapsed\_seconds.count() << "s\n";

    //File handling

    // Create and open a text file

    string filename("bfscomplexity.txt");

    ofstream file\_out;

    file\_out.open(filename, std::ios\_base::app);

    file\_out <<elapsed\_seconds.count() << endl;

    cout << "Data stored Successfully !" << endl;

    return 0;

}

//  system("gnuplot -p -e \"set ylabel 'mileseconds' ; set xlabel 'data set' ; p 'D:/SEM 5 programming/DAA Lab/.vscode/bfscomplexity.txt' using 1:2 w lp lc rgb 'red' lw 2 pt 7 title 'BFS','D:/SEM 5 programming/DAA Lab/.vscode/dfscomplexity.txt' using 1:2 w lp lc rgb 'blue' lw 2 pt 7 title 'DFS',/"");

/\*

7

        1

       / \

      2   3

    /|\ \  \

   / | \  \ \

  /  |  \   \\

 4   5   6    7

 \*/

7

1 2

1 3

2 4

2 5

2 6

2 7

3 7

\*/

/\*

**CODE 1 :- DFS TRAVERSAL**

#include<bits/stdc++.h>

#include <algorithm>

#include<bits/stdc++.h>

#include <chrono>

#include <ctime>

#include <fstream>

using namespace std;

using namespace std::chrono;

const int N = 1e5+2;

bool vis[N];

vector<int> adj[N];

void dfs(int node)

{

    //preorder

    vis[node] = 1;

    cout<< node << " ";

    // inorder

    vector<int> :: iterator it;

    for(it = adj[node].begin(); it != adj[node].end(); it++)

    {

        if(vis[\*it]);

        else

        {

            dfs(\*it);

        }

    }

    //  For postorder remove from above

    // cout<< node << " ";

}

int main()

{

    int n,m;

    cout<<"Number of nodes: ";

    cin>>n;

    cout<<"\n Number of edges: ";

    cin>>m;

    for(int i =0 ; i<=n ; i++)

        vis[i] = false;

    int x,y;

    for (int i = 0; i < m; i++)

    {

        cin >> x  >> y;

        adj[x].push\_back(y);

        adj[y].push\_back(x);

    }

    std::chrono::time\_point<std::chrono::system\_clock> start, end;

    start = std::chrono::system\_clock::now();

    dfs(1);

    end = std::chrono::system\_clock::now();

    std::chrono::duration<double> elapsed\_seconds = end - start;

    std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);

    std::cout << "finished computation at " << std::ctime(&end\_time)<< "elapsed time: " << elapsed\_seconds.count() << "s\n";

    //File handling

    // Create and open a text file

    string filename("dfscomplexity.txt");

    ofstream file\_out;

    file\_out.open(filename, std::ios\_base::app);

    file\_out <<elapsed\_seconds.count() << endl;

    cout << "Data stored Successfully !" << endl;

    return 0;

}

/\*

7

7

1 2

1 3

2 4

2 5

2 6

2 7

3 7

\*/

/\*

        1

       / \

      2   3

    /|\ \  \

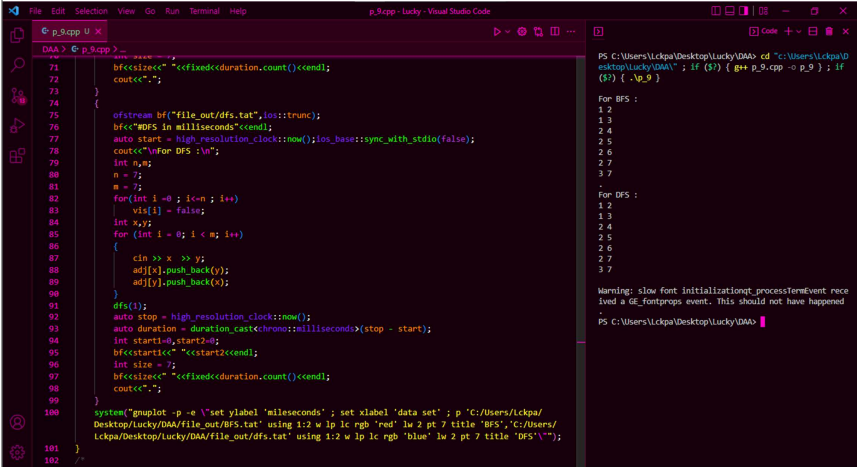
   / | \  \ \

  /  |  \   \\

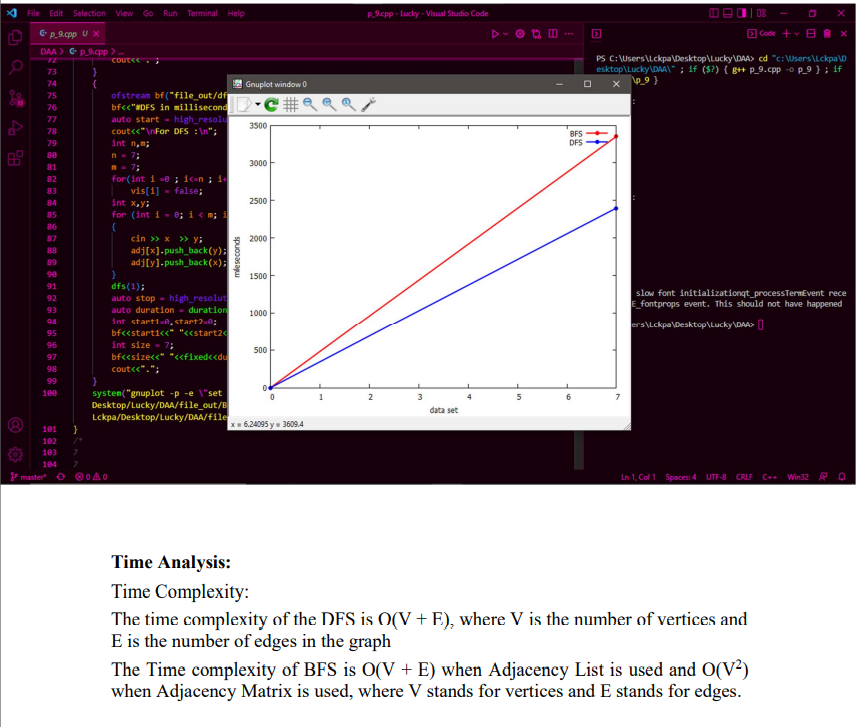
 4   5   6    7

\*/

**Output :-**

****

****

****