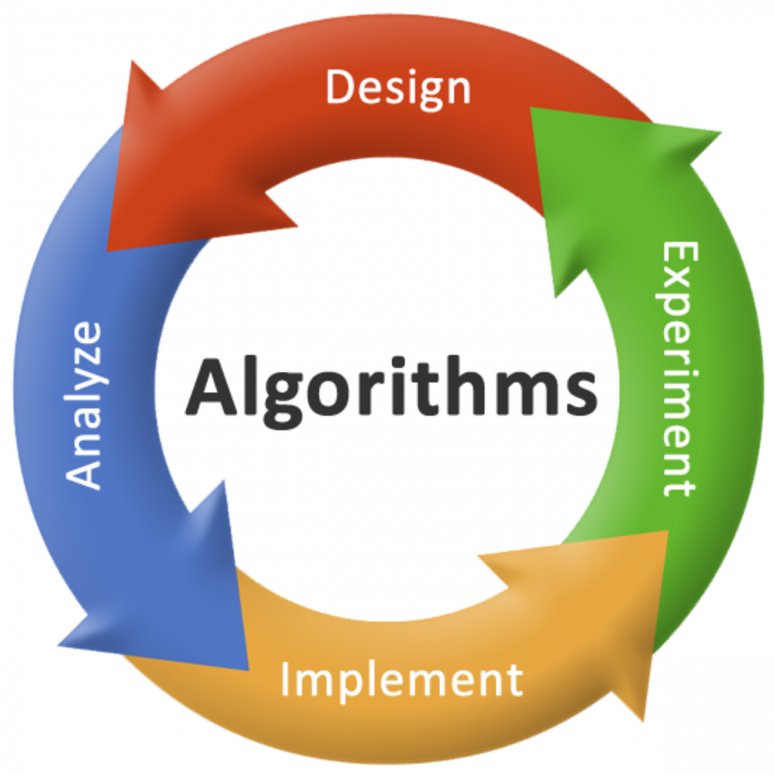
**Design and Analysis of Algorithms**

**Practical File**



***NAME : KHUSHI JAIN***

***ROLL NO : 20078570037***

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Practical** | **Page No.** |
| **1** | Implement Insertion Sort (The program should report the number of comparisons) | 3 |
| **2** | Implement Merge Sort (The program should report the number of comparisons) | 8 |
| **3** | Implement Heap Sort(The program should report the number of comparisons) | 12 |
| **4** | Implement Randomized Quick sort (The program should report the number of comparisons) | 19 |
| **5** | Implement Radix Sort | 23 |
| **6** | Implement Bucket Sort | 25 |
| **7** | Implement Randomized Select | 28 |
| **8** | Implement Breadth-First Search in a graph | 30 |
| **9** | Implement Depth-First Search in a graph | 34 |
| **10** | Write a program to determine the minimum spanning tree of a graph using both Prims and Kruskals algorithm | 36 |
| **11** | Write a program to solve the weighted interval scheduling problem | 44 |
| **12** | Write a program to solve the 0-1 knapsack problem | 48 |

**PRACTICAL 1 : Implement Insertion Sort (The program should report the number of comparisons)**

#include <bits/stdc++.h>

using namespace std;

//function to sort an array and return the number of comparisons

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

{

int comparison = 0;

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

{

int current = arr[i];

int j = i - 1;

while (arr[j] > current && j >= 0)

{

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

j--;

comparison++;

}

if (j >= 0)

{

comparison++;

}

arr[j + 1] = current;

}

return comparison;

}

//function to generate random array and create .csv file

void generate\_random\_array(int size, int check)

{

fstream fout;

int comp;

int arr[size];

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

{

arr[i] = (rand() % 100);

}

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

if (check == 0) //average case

{

comp = insertion\_sort(arr, size);

fout.open("average\_case.csv", ios::out | ios::app);

fout << size << "," << comp << endl;

}

else if (check == 1) //best\_case

{

sort(arr, arr + n);

comp = insertion\_sort(arr, size);

fout.open("best\_case.csv", ios::out | ios::app);

fout << size << "," << comp << endl;

}

else if (check == 2) //worst\_case

{

sort(arr, arr + n, greater<int>());

comp = insertion\_sort(arr, size);

fout.open("worst\_case.csv", ios::out | ios::app);

fout << size << "," << comp << endl;

}

}

//average case

void average\_case(int start, int end)

{

int interval = (end - start) / 60;

int count = 0;

for (int i = start; i < end && count < 60; i += interval)

{

generate\_random\_array(i, 0);

count++;

}

}

//best\_case

void best\_case(int start, int end)

{

int interval = (end - start) / 20;

int count = 0;

for (int i = start; i < end && count < 20; i += interval)

{

generate\_random\_array(i, 1);

count++;

}

}

//worst\_case

void worst\_case(int start, int end)

{

int interval = (end - start) / 20;

int count = 0;

for (int i = start; i < end && count < 20; i += interval)

{

generate\_random\_array(i, 2);

count++;

}

}

//main function

int main()

{

int min\_value = 30;

int max\_value = 1000;

average\_case(min\_value, max\_value);

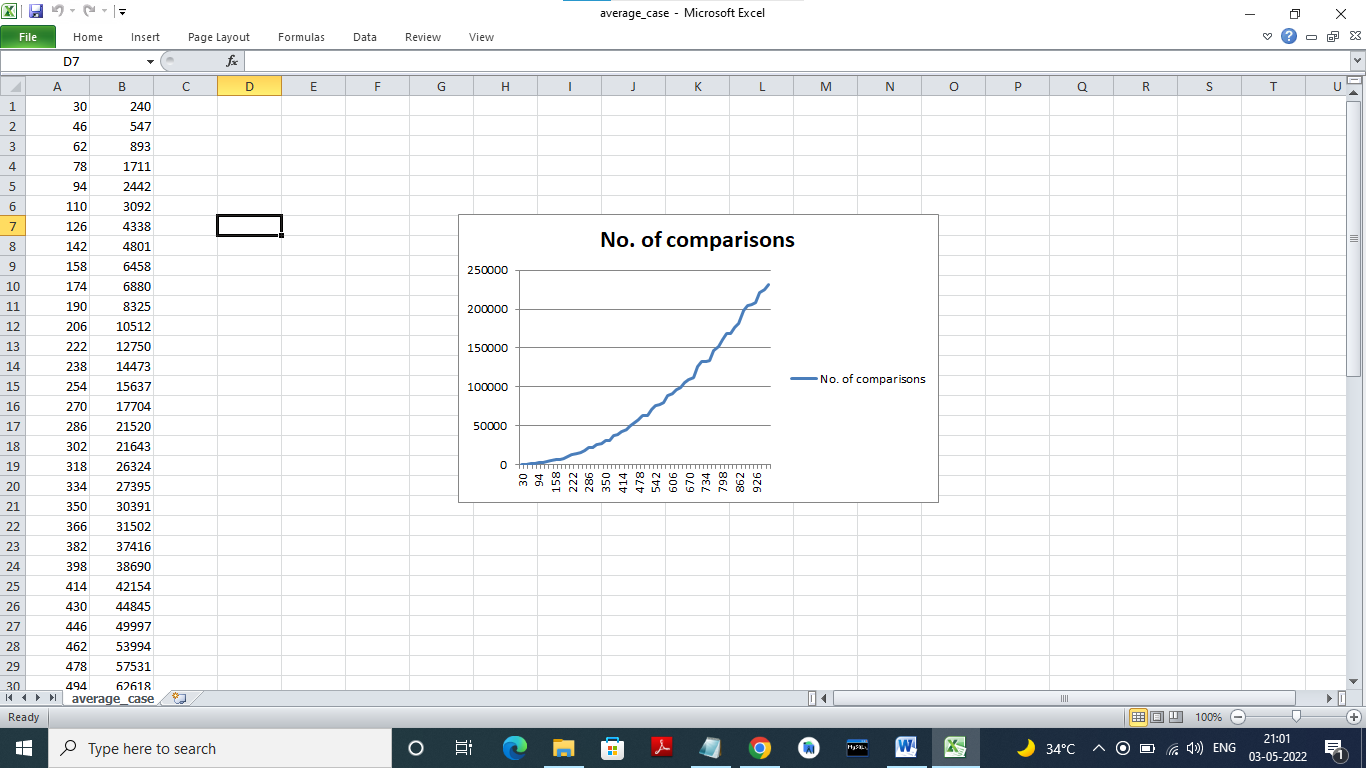
best\_case(min\_value, max\_value);

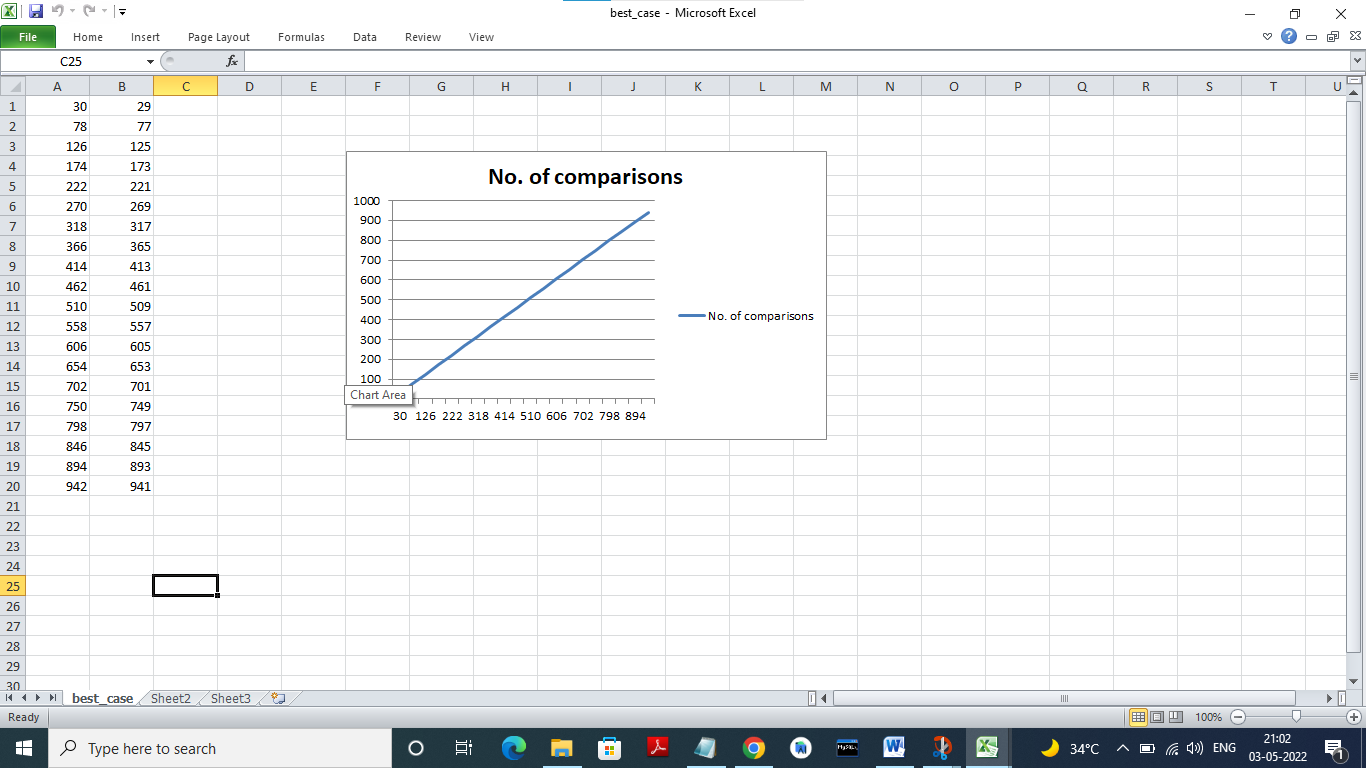
worst\_case(min\_value, max\_value);

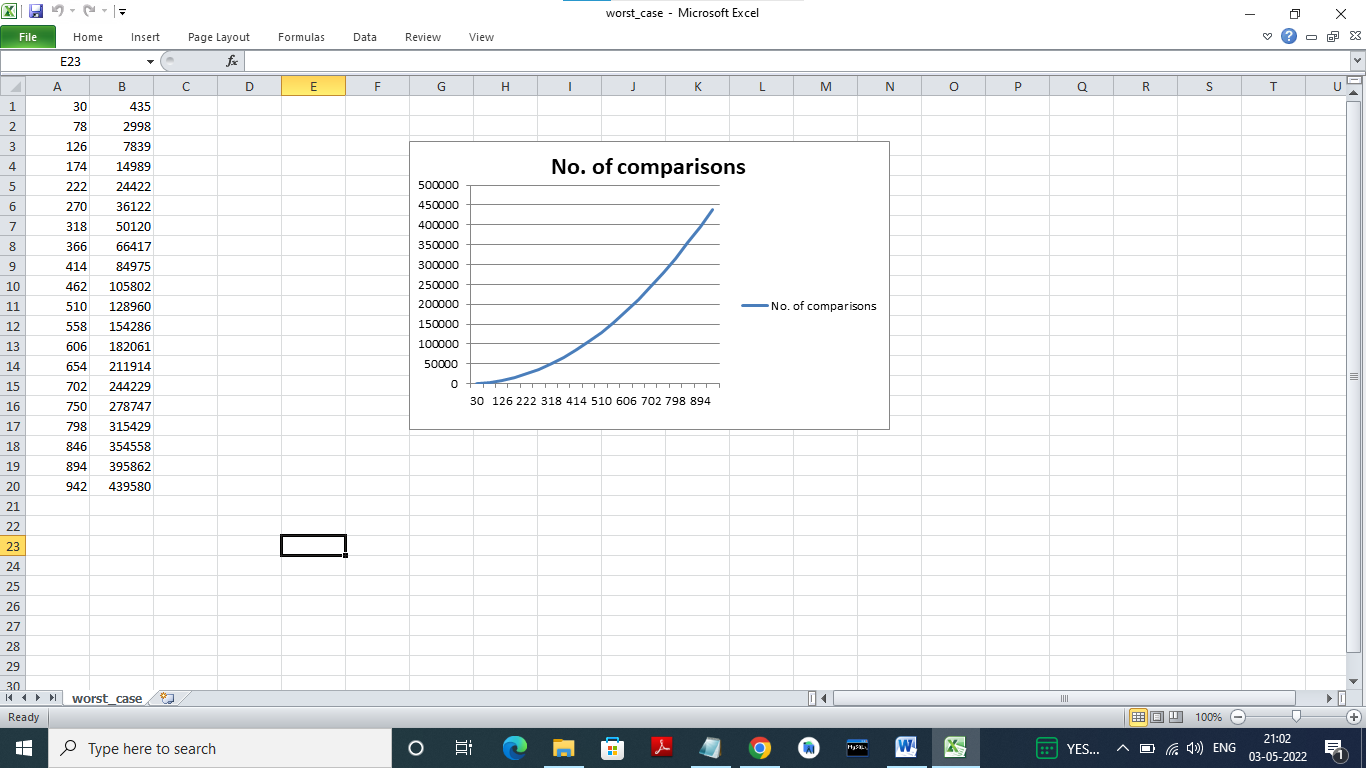
return 0;

}

**OUTPUT**







**PRACTICAL 2: Implement Merge Sort (The program should report the number of comparisons)**

#include <bits/stdc++.h>

using namespace std;

int comp = 0;

int n = 0;

void printarray( int size,int comp){

fstream fout;

fout.open("merge.csv", ios::out | ios::app);

fout << size << "," << comp << endl;

}

void merge(int arr[], int leftFirst, int leftLast, int rightFirst, int rightLast, int size){

int temparr[size];

int index = leftFirst;

int saveFirst = leftFirst;

while((leftFirst <= leftLast) && ( rightFirst <= rightLast)){//compare and select smallest from two subarrays

if(arr[leftFirst] < arr[rightFirst]){

temparr[index] = arr[leftFirst];

leftFirst++;

}

else

{

temparr[index] = arr[rightFirst];

rightFirst++;

}

index++;

comp++;

}

while(leftFirst <= leftLast){

temparr[index] = arr[leftFirst];

leftFirst++;

index++;

}

while(rightFirst <= rightLast){

temparr[index] = arr[rightFirst];

rightFirst++;

index++;

}

for(index = saveFirst; index <= rightLast; index++)

arr[index] = temparr[index];

}

void mergesort(int a[], int start, int end, int size){

if(start < end){

int mid = (start+end)/2;

int n=mid+1;

mergesort(a,start, mid,size);

mergesort(a,n,end,size);

merge(a, start,mid, n, end, size);

}

}

void randomNum(int size)

{

int arr[size];

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

{

arr[i] = (rand() % 100);

}

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

int start = 0;

int end = n-1;

mergesort(arr, start, end, size);

}

int main(){

int min = 30;

int max = 1000;

int interval = (max - min) / 100;

int count = 0;

for (int i = min; i < max && count < 100; i += interval)

{

randomNum(i );

printarray(i,comp);

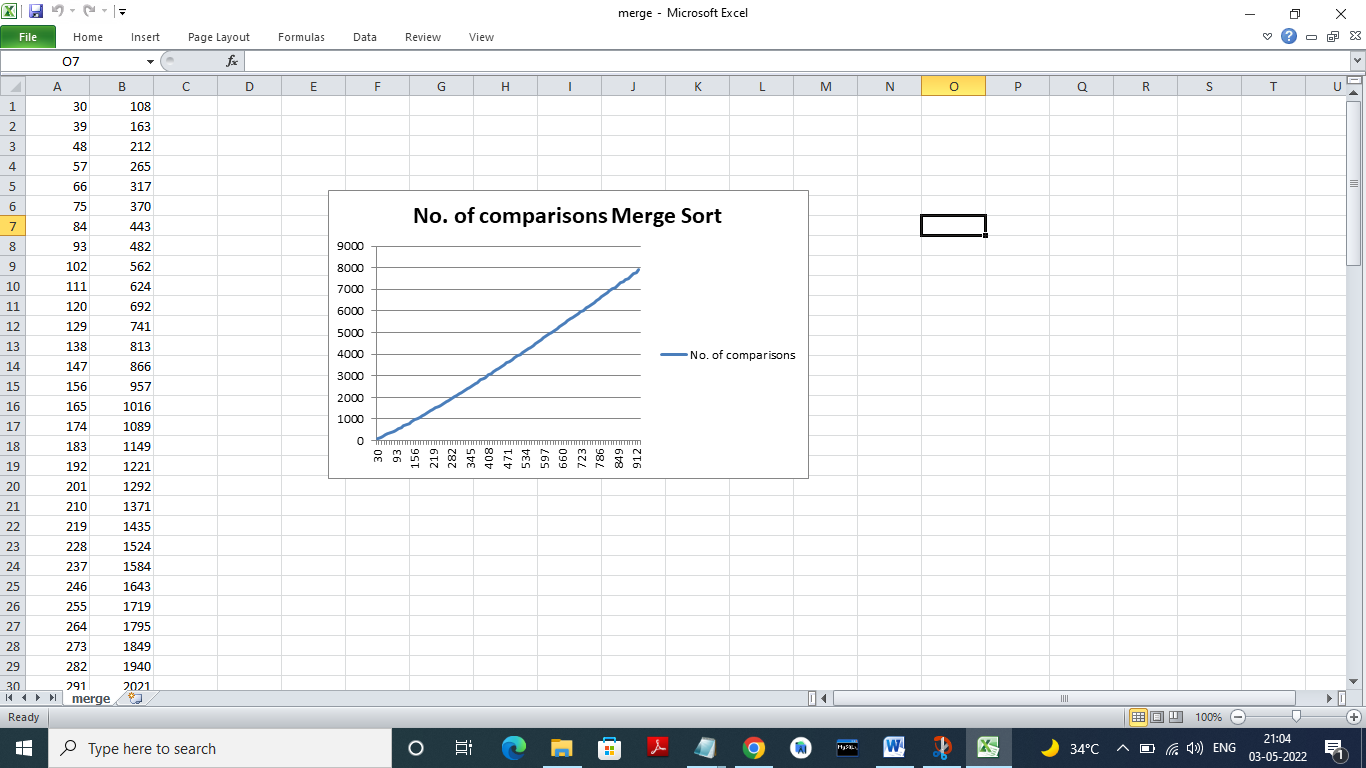
count++;

comp=0;

}

}

**OUTPUT**



**PRACTICAL 3 : Implement Heap Sort(The program should report the number of comparisons)**

#include<iostream>

#include <algorithm>

#include<fstream>

using namespace std;

//Heapify function

void heapify(int arr[],int n,int i,int &count)

{

int largest=i;

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

int r=(i\*2)+2; //right child

if(l<n&&(count+=1)&&(arr[l]>arr[largest])) // If left child is larger than root

{

largest=l;

}

if(r<n&&(count+=1)&&(arr[r]>arr[largest])) // If right child is larger than largest so far

{

largest=r;

}

if(largest != i ) // If largest is not root

{

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

heapify(arr,n,largest,count); // Recursively heapify the affected sub-tree

}

}

//heap sort func

int Heap\_Sort(int arr[],int n,int &comp)

{

// For building heap

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

{

heapify(arr,n,i,comp);

}

// for deletion from heap

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

{

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

heapify(arr,i,0,comp);

}

return comp;

}

// For generating Random array

void generate\_random\_array(int arr[], int size)

{

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

{

arr[i] = (rand() % 100);

}

}

// Average Case

void heap\_avg(int num\_cases = 100, int min\_size = 30, int max\_size = 1000)

{

int interval = (max\_size - min\_size) / num\_cases;

int comp = 0;

for (int size = min\_size, i = 0; i < num\_cases; i++, size += interval)

{

int \*Arr = new int[size];

generate\_random\_array(Arr, size);

fstream fout1;

Heap\_Sort(Arr,size,comp);

fout1.open("HeapAverage.csv", ios::out | ios::app);

fout1 << size << "," << comp << endl;

comp = 0;

}

}

//Best Case

void heap\_best(int num\_cases = 20, int min\_size = 30, int max\_size = 1000)

{

int interval = (max\_size - min\_size) / num\_cases;

int comp = 0;

for (int size = min\_size, i = 0; i < num\_cases; i++, size += interval)

{

int \*Arr = new int[size];

generate\_random\_array(Arr, size);

sort(Arr, Arr + size, greater<int>());

fstream fout1;

Heap\_Sort(Arr,size,comp);

fout1.open("HeapBest.csv", ios::out | ios::app);

fout1 << size << "," << comp << endl;

comp = 0;

}

}

//Worst Case

void heap\_Worst(int num\_cases = 20, int min\_size = 30, int max\_size = 1000)

{

int interval = (max\_size - min\_size) / num\_cases;

int comp = 0;

for (int size = min\_size, i = 0; i < num\_cases; i++, size += interval)

{

int \*Arr = new int[size];

generate\_random\_array(Arr, size);

sort(Arr, Arr + size);

fstream fout1;

Heap\_Sort(Arr,size,comp);

fout1.open("HeapWorst.csv", ios::out | ios::app);

fout1 << size << "," << comp << endl;

comp = 0;

}

}

int main()

{

heap\_avg();

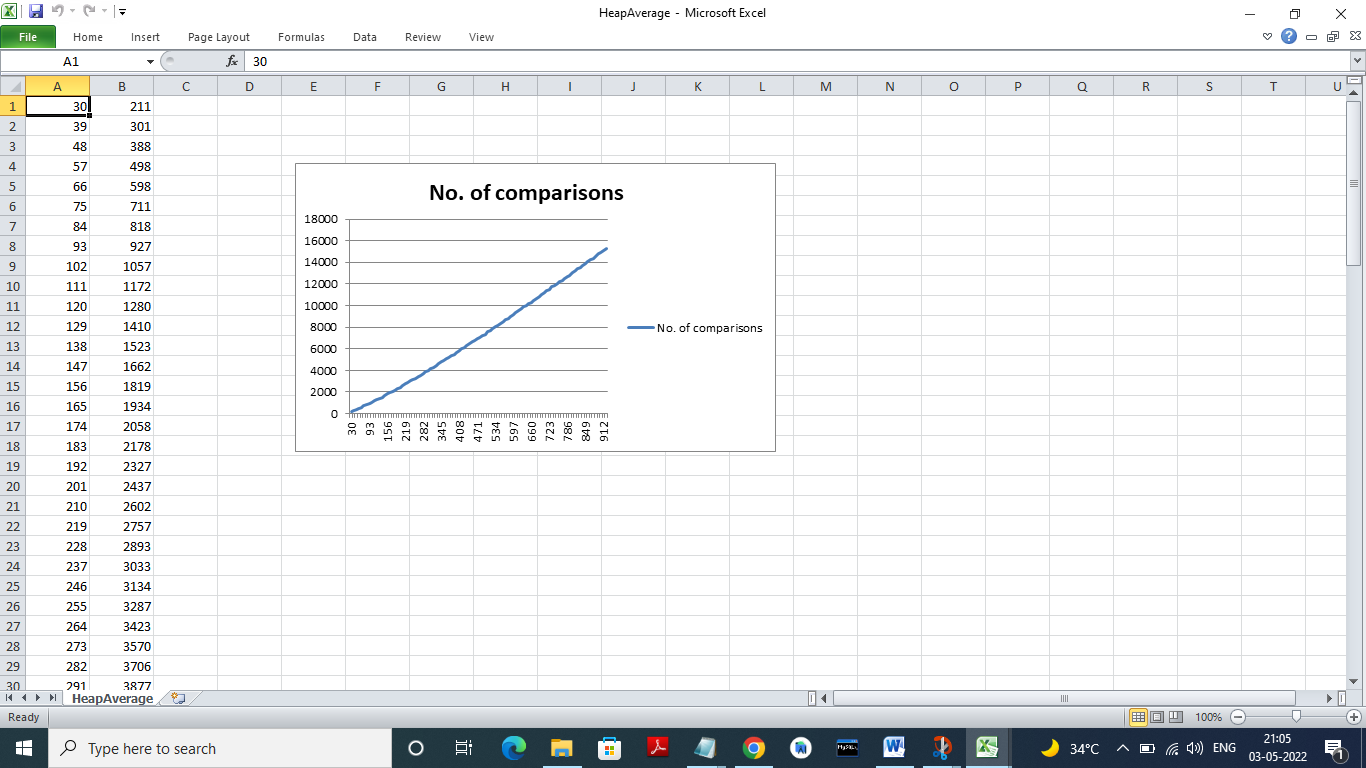
heap\_best();

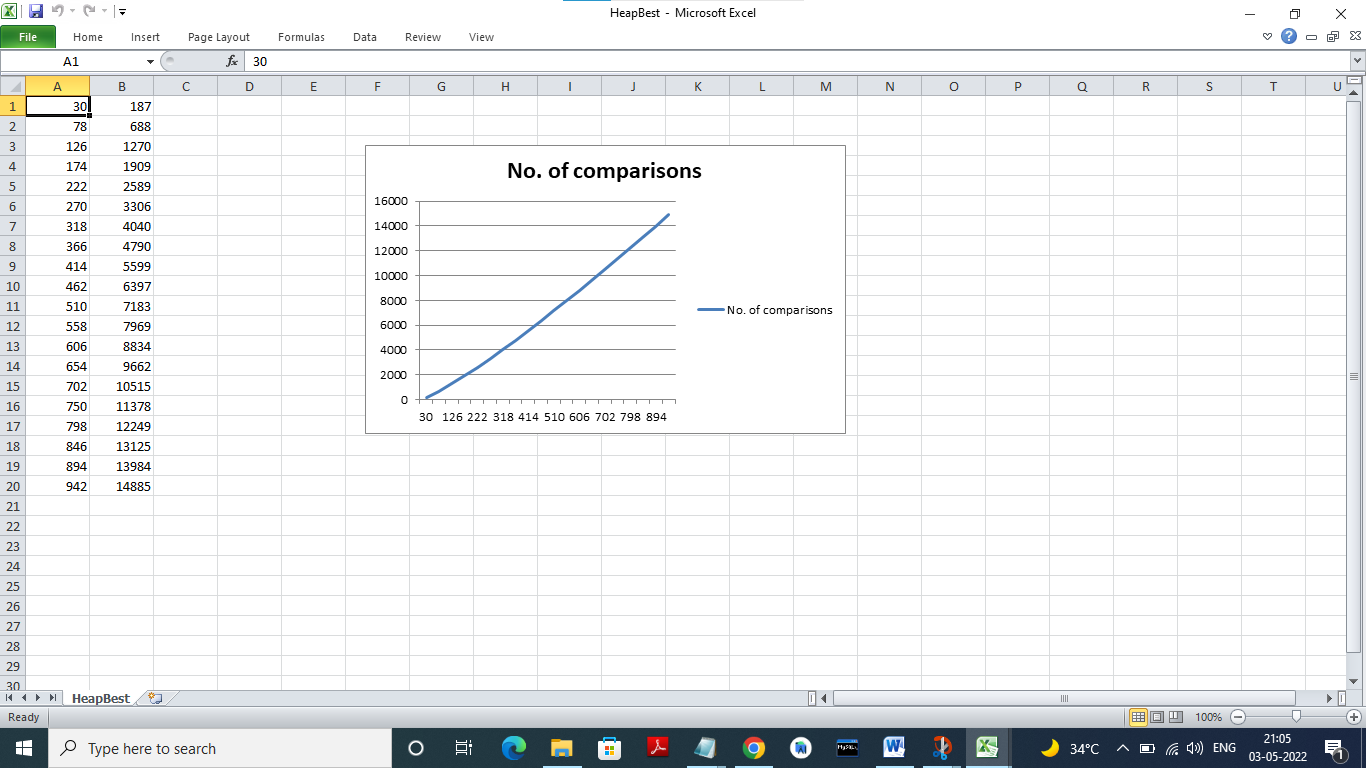
heap\_Worst();

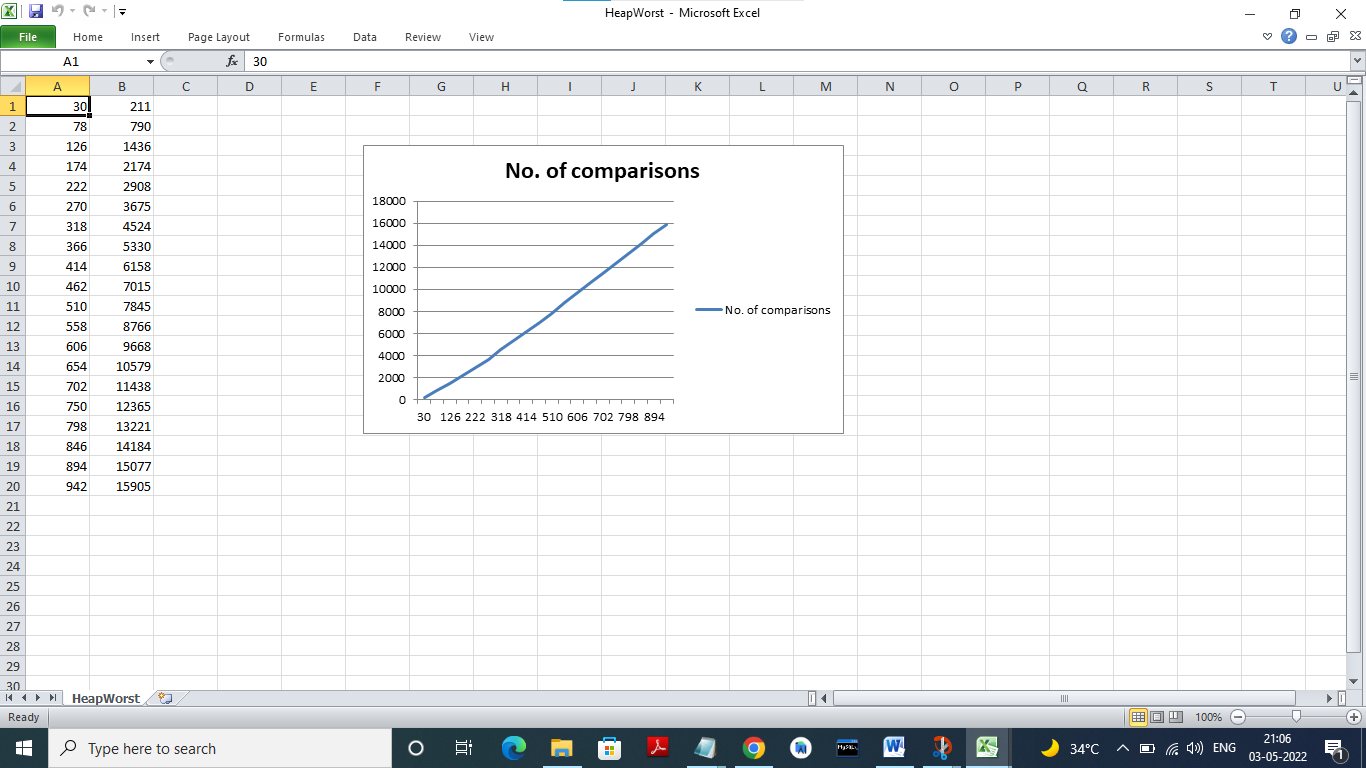
return 0;

}

**OUTPUT**







**PRACTICAL 4 : Implement Randomized Quick sort (The program should report the number of comparisons)**

#include <iostream>

#include <fstream>

using namespace std;

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

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int arr[], int low, int high, int &numOfcomp)

{

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

if (numOfcomp++, arr[j] < pivot)

{

i++;

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

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

int random\_partition(int arr[], int low, int high, int &numOfcomp)

{

int random\_index = (rand() % (high + 1));

if (arr[random\_index] != arr[high])

swap(&arr[random\_index], &arr[high]);

return partition(arr, low, high, numOfcomp);

}

void random\_quickSort(int arr[], int low, int high, int &numOfcomp)

{

if (low < high)

{

int pi = partition(arr, low, high, numOfcomp);

random\_quickSort(arr, low, pi - 1, numOfcomp);

random\_quickSort(arr, pi + 1, high, numOfcomp);

}

}

void generate\_random\_array(int arr[], int size)

{

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

{

arr[i] = (rand() % 1000);

}

}

void average\_case(int cases, int min, int max)

{

ofstream average\_case\_file("average\_quick\_Random.csv");

int interval = (((max - min) / cases) / 10.0) \* 10;

int count = 0;

for (int i = min; count < cases; i += interval, count++)

{

int \*random\_array = new int[i];

generate\_random\_array(random\_array, i);

int numOfcomp = 0;

random\_quickSort(random\_array, 0, i - 1, numOfcomp);

average\_case\_file << i << "," << numOfcomp << "\n";

delete[] random\_array;

}

average\_case\_file.close();

}

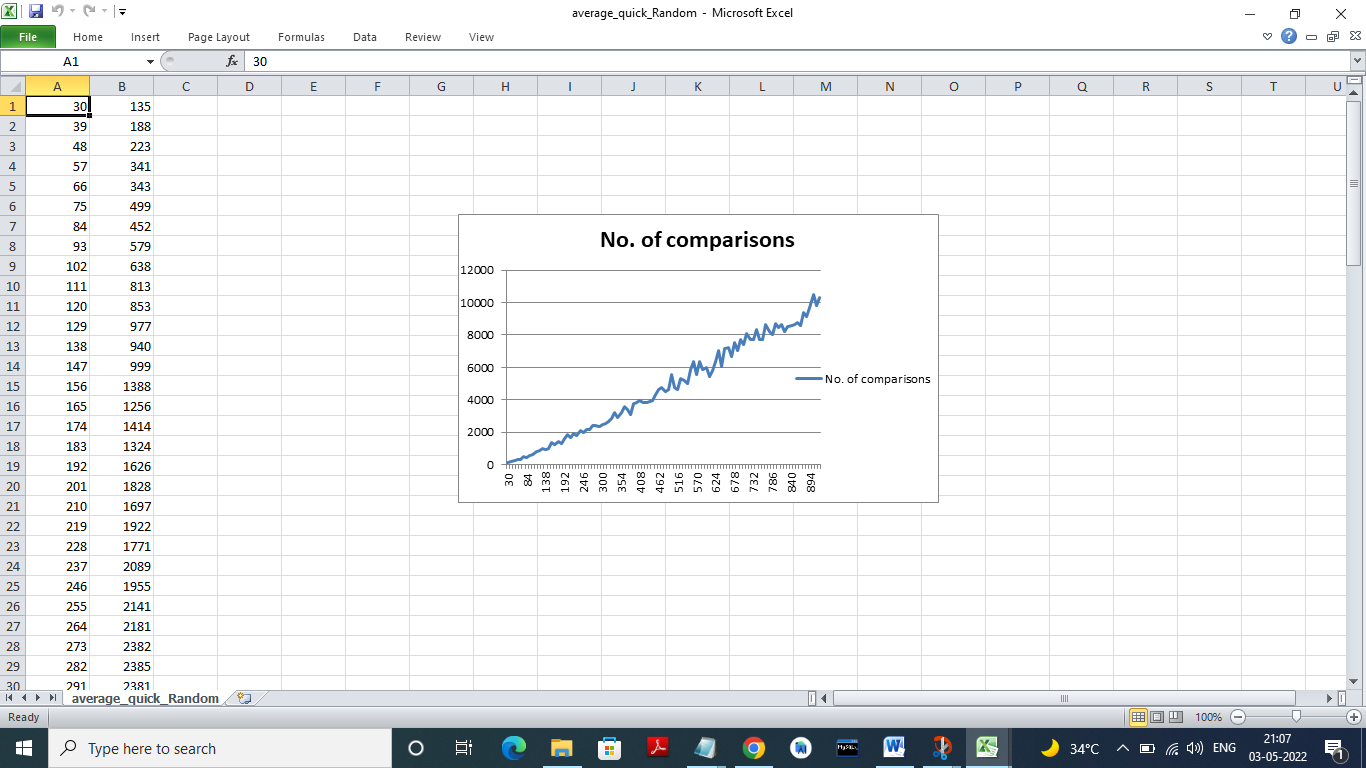
int main()

{

average\_case(100, 30, 1000);

}

**OUTPUT**



**PRACTICAL 5 : Implement Radix Sort**

#include <iostream>

using namespace std;

int getMax(int arr[], int n)

{

int mx = arr[0];

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

if (arr[i] > mx)

mx = arr[i];

return mx;

}

// A function to do counting sort of arr[] according to

// the digit represented by exp.

void countSort(int arr[], int n, int exp)

{

int output[n];

int i, count[10] = { 0 };

// Store count of occurrences in count[]

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

count[(arr[i] / exp) % 10]++;

// Change count[i] so that count[i] now contains actual

// position of this digit in output[]

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

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

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

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

arr[i] = output[i];

}

void radixsort(int arr[], int n)

{

int m = getMax(arr, n);

for (int exp = 1; m / exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

void print(int arr[], int n)

{

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

cout << arr[i] << " ";

}

int main()

{

int arr[] = { 76, 34, 11, 88, 90, 3, 1 };

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

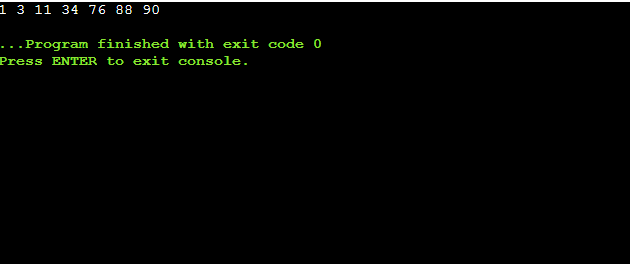
radixsort(arr, n);

print(arr, n);

return 0;

}

**OUTPUT**



**PRACTICAL 6 : Implement Bucket Sort**

#include <iostream>

using namespace std;

int getMax(int a[], int n) // function to get maximum element from the given array

{

int max = a[0];

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

if (a[i] > max)

max = a[i];

return max;

}

void bucket(int a[], int n) // function to implement bucket sort

{

int max = getMax(a, n); //max is the maximum element of array

int bucket[max], i;

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

{

bucket[i] = 0;

}

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

{

bucket[a[i]]++;

}

for (int i = 0, j = 0; i <= max; i++)

{

while (bucket[i] > 0)

{

a[j++] = i;

bucket[i]--;

}

}

}

void printArr(int a[], int n) // function to print array elements

{

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

cout<<a[i]<<" ";

}

int main()

{

int a[] = {35, 56, 11, 9, 22, 1};

int n = sizeof(a) / sizeof(a[0]); // n is the size of array

cout<<"Before sorting array elements are - ";

printArr(a, n);

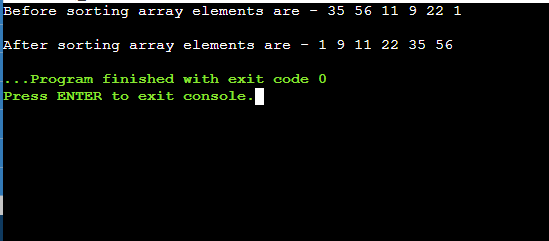
bucket(a, n);

cout<<"\n\nAfter sorting array elements are - ";

printArr(a, n);

}

**OUTPUT**



**PRACTICAL 7 : Implement Randomized Select**

#include <iostream>

using namespace std;

int random\_partition(int \*arr, int start, int end)

{

int pivotIdx = start + rand() % (end - start + 1);

int pivot = arr[pivotIdx];

swap(arr[pivotIdx], arr[end]);

pivotIdx = end;

int i = start - 1;

for (int j = start; j <= end - 1; j++)

{

if (arr[j] <= pivot)

{

i = i + 1;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[pivotIdx]);

return i + 1;

}

int random\_selection(int \*arr, int start, int end, int k)

{

int i = 0;

if (start == end)

return arr[start];

if (k == 0)

return -1;

if (start < end)

{

int mid = random\_partition(arr, start, end);

i = mid - start + 1;

if (i == k)

return arr[mid];

else if (k < i)

return random\_selection(arr, start, mid - 1, k);

else

return random\_selection(arr, mid + 1, end, k - i);

}

}

int main()

{

int A[] = {9, 5, 7, 1, 10, 2, 3};

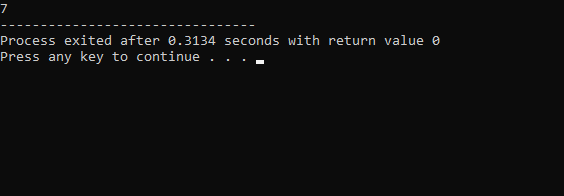
int arr = random\_selection(A, 0, 6, 5);

cout << arr;

return 0;

}

**OUTPUT**



**PRACTICAL 8 : Implement Breadth-First Search in a graph**

// Breadth First Search

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver code

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

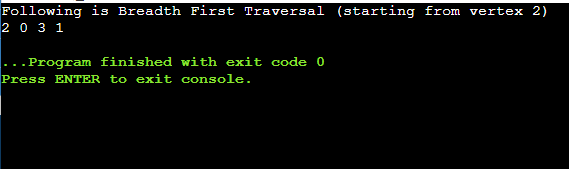
<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

**OUTPUT**



**PRACTICAL 9 : Implement Depth-First Search in a graph**

// Depth First Search

#include <bits/stdc++.h>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFS(int v)

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

// Driver code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

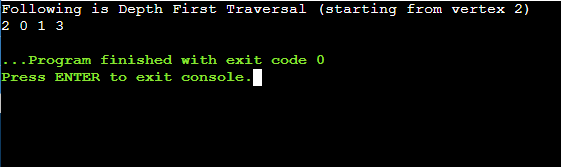
" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

**OUTPUT**



**PRACTICAL 10 : Write a program to determine the minimum spanning tree of a graph using both Prims and Kruskals algorithm**

**PRIMS ALGORITHM**

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with minimum key value, from the set of vertices not yet included in MST

int minKey(int key[], bool mstSet[])

{

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

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

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the constructed MST stored in parent[]

void printMST(int parent[], int graph[V][V])

{

cout<<"Edge \tWeight\n";

for (int i = 1; i < V; i++)

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

}

// Function to construct and print MST for a graph represented using adjacency matrix representation

void primMST(int graph[V][V])

{

int parent[V]; // Array to store constructed MST

int key[V]; // Key values used to pick minimum weight edge in cut

bool mstSet[V]; // To represent set of vertices included in MST

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX, mstSet[i] = false;

// Always include 1st vertex in MST. Make key 0 so that this vertex is picked as first vertex.

key[0] = 0;

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

// The MST will have V vertices

for (int count = 0; count < V - 1; count++)

{

// Pick the minimum key vertex from the set of vertices not yet included in MST

int u = minKey(key, mstSet);

mstSet[u] = true; // Add the picked vertex to the MST Set

// Update key value and parent index of the adjacent vertices of the picked vertex.

// Consider only those vertices which are not yet included in MST

for (int v = 0; v < V; v++)

// graph[u][v] is non zero only for adjacent vertices of m mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

// print the constructed MST

printMST(parent, graph);

}

// Driver code

int main()

{

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

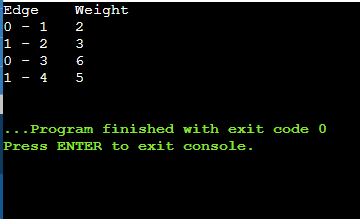
// Print the solution

primMST(graph);

return 0;

}

**OUTPUT**



**KRUSKALS ALGORITHM**

#include <bits/stdc++.h>

using namespace std;

class DSU {

int\* parent;

int\* rank;

public:

DSU(int n)

{

parent = new int[n];

rank = new int[n];

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

parent[i] = -1;

rank[i] = 1;

}

}

// Find function

int find(int i)

{

if (parent[i] == -1)

return i;

return parent[i] = find(parent[i]);

}

// union function

void unite(int x, int y)

{

int s1 = find(x);

int s2 = find(y);

if (s1 != s2) {

if (rank[s1] < rank[s2]) {

parent[s1] = s2;

rank[s2] += rank[s1];

}

else {

parent[s2] = s1;

rank[s1] += rank[s2];

}

}

}

};

class Graph {

vector<vector<int> > edgelist;

int V;

public:

Graph(int V) { this->V = V; }

void addEdge(int x, int y, int w)

{

edgelist.push\_back({ w, x, y });

}

void kruskals\_mst()

{

// 1. Sort all edges

sort(edgelist.begin(), edgelist.end());

// Initialize the DSU

DSU s(V);

int ans = 0;

cout << "Following are the edges in the "

"constructed MST"

<< endl;

for (auto edge : edgelist) {

int w = edge[0];

int x = edge[1];

int y = edge[2];

// take that edge in MST if it does form a cycle

if (s.find(x) != s.find(y)) {

s.unite(x, y);

ans += w;

cout << x << " -- " << y << " == " << w

<< endl;

}

}

cout << "Minimum Cost Spanning Tree: " << ans;

}

};

int main()

{

Graph g(4);

g.addEdge(0, 1, 10);

g.addEdge(1, 3, 15);

g.addEdge(2, 3, 4);

g.addEdge(2, 0, 6);

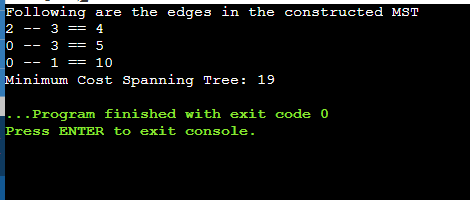
g.addEdge(0, 3, 5);

g.kruskals\_mst();

return 0;

}

**OUTPUT**



**PRACTICAL 11 : Write a program to solve the weighted interval scheduling problem**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Data structure to store a Job

struct Job

{

int start, finish, weight;

};

void findMaxWeightJobs(vector<Job> jobs)

{

// sort the jobs according to increasing order of their start time

// sort(jobs.begin(), jobs.end(),

// [](Job &x, Job &y)

// {

// return x.start < y.start;

// });

// output:The jobs involved in the maximum weight are (1, 4, 30) (4, 5, 60) (5, 9, 50)

// sort the jobs according to increasing order of their finish time

sort(jobs.begin(), jobs.end(),

[](Job &x, Job &y)

{

return x.finish < y.finish;

});

// output:The jobs involved in the maximum weight are (1, 4, 30) (4, 5, 60) (5, 9, 50)

// get the number of jobs

int n = jobs.size();

// base case

if (n == 0)

{

return;

}

// `tasks[i]` stores the index of non-conflicting jobs involved in the

// maximum weight, which ends with the i'th job

vector<int> tasks[n];

// `maxweight[i]` stores the total weight of jobs in `tasks[i]`

int maxWeight[n];

// consider every job

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

{

// initialize current weight to 0

maxWeight[i] = 0;

// consider each `j` less than `i`

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

{

// update i'th job if the j'th job is non-conflicting and leading to the

// maximum weight

if (jobs[j].finish <= jobs[i].start && maxWeight[i] < maxWeight[j])

{

tasks[i] = tasks[j];

maxWeight[i] = maxWeight[j];

}

}

// end current task with i'th job

tasks[i].push\_back(i);

maxWeight[i] += jobs[i].weight;

}

// find an index with the maximum weight

int index = 0;

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

{

if (maxWeight[i] > maxWeight[index])

{

index = i;

}

}

cout << "The jobs involved in the maximum weight are ";

for (int i : tasks[index])

{

cout << "(" << jobs[i].start << ", " << jobs[i].finish << ", "

<< jobs[i].weight << ") ";

}

}

// Main Function(Driver Code)

int main()

{

vector<Job> jobs{

{0, 6, 60},

{5, 9, 50},

{1, 4, 30},

{4, 5, 60},

{5, 7, 30},

{3, 5, 10},

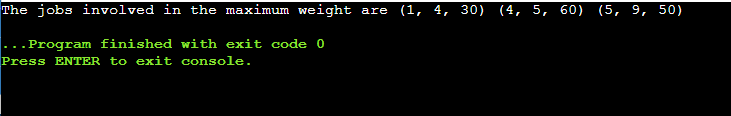
{7, 8, 10}};

findMaxWeightJobs(jobs);

return 0;

}

**OUTPUT**



**PRACTICAL 12 : Write a program to solve the 0-1 knapsack problem**

**DYNAMIC PROGRAMMING**

// Program to solve the 0-1 knapsack problem using dynamic programming.

#include <iostream>

using namespace std;

int max(int x, int y) {

return (x > y) ? x : y;

}

int knapSack(int W, int w[], int v[], int n) {

int i, wt;

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

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

for (wt = 0; wt <= W; wt++) {

if (i == 0 || wt == 0)

K[i][wt] = 0;

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

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

else

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

}

}

return K[n][W];

}

int main() {

int n, W;

cout << "Enter the number of items in a Knapsack : ";

cin >> n;

cout<<endl;

int v[n], w[n];

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

cout << "Enter profit value and weight for item " << i << ": ";

cin >> v[i];

cin >> w[i];

}

cout << "\nEnter the capacity of knapsack : ";

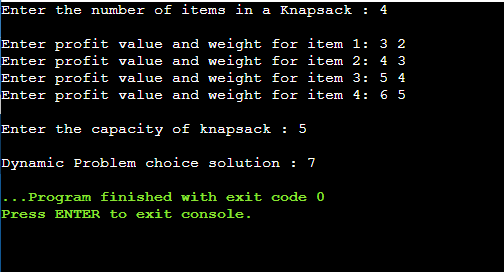
cin >> W;

cout << "\nDynamic Problem choice solution : "<< knapSack(W, w, v, n);

return 0;

}

**OUTPUT**



**GREEDY**

#include <iostream>

using namespace std;

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

{

double total\_Profit = 0;

int c = capacity;

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

{

if (weight[i] > c)

break;

else

{

total\_Profit += profit[i];

c -= weight[i];

}

}

cout << "\nMaximum profit : " << total\_Profit;

}

// Main function(Driver Code)

int main()

{

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

cout << "\nEnter the no. of objects : ";

int num;

cin >> num;

cout << "\nEnter the weights and profits of each object (With space) : \n";

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

{

cin >> weight[i] >> profit[i];

}

cout << "\nEnter the capacity of knapsack : ";

cin >> capacity;

knapsack(num, weight, profit, capacity);

return 0;

}

**OUTPUT**

