Lab Manual

for

Subject Title: **Design and Analysis of Algorithms Lab**

Subject Code: **PCCS-623**

Prepared by:

Dr. Preetpal Kaur Buttar

Assistant Professor (CSE)

Department of Computer Science & Engineering
Sant Longowal Institute of Engineering & Technology,
Longowal

Syllabus

Title of the course : **Design and Analysis of Algorithms Lab**

Subject Code : **PCCS-623**

Weekly load : **4 Hrs** **LTP 0-0-4**

Credit **2**

Course Outcomes: At the end of the course, the students will be able to:

CO1	learn how to analyze a problem and design a solution for the problem
CO2	acquire hands-on skills to implement various advanced algorithm design such as divide-and-conquer, greedy algorithms
CO3	implement quick sort, merge sort algorithm, BFS and DFS algorithms
CO4	implement dynamic programming algorithm for the 0/1 knapsack problem
CO5	utilize data structures and/or algorithmic design techniques for developing efficient computer algorithm for solving real-world problems

CO/PO Mapping: (Strong(3)/Medium(2)/Weak(1) indicates strength of correlation):														
Cos	Program Outcomes (PO's)/ Program Specific Outcomes (PSO's)													
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	3	3	3	2	2	0	0	0	0	0	0	2	1	2
CO2	3	3	3	2	2	0	0	0	0	0	0	0	2	2
CO3	3	3	2	2	0	2	2	0	0	0	0	0	3	3
CO4	0	0	1	1	0	1	0	2	1	2	0	1	2	3
CO5	3	3	3	2	2	0	0	0	0	0	0	0	2	2

LIST OF PRACTICALS

1. Use divide and conquer method to recursively implement:
   a) Binary search
   b) Linear search
2. Sort a given set of elements using the insertion sort method.
3. Sort a given set of elements using the selection sort method.
4. Sort a given set of elements using the quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
5. Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of

elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

6. Sort a given set of elements using the heap sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
7. From a given vertex in an unweighted connected graph, find shortest paths to other vertices using BFS.
8. Given an undirected graph, use BFS to check if there is a cycle in the graph.
9. Use BFS to find all connected components in a graph and finding all vertices within a connected component.
10. Given a directed graph, use DFS to check if there is a cycle in the graph.
11. Check whether a given graph is connected or not using DFS method.
12. Use DFS to find all strongly connected components in a directed graph.
13. Obtain the topological ordering of vertices in a given directed graph.
14. For a given set of elements, construct an AVL Tree and also display balance factor for each node.
15. Find the median of the two sorted arrays of different sizes using divide and conquer approach.
16. Count the number of inversions in an array with the help of merge sort.
17. Find out the closest pair of points in the array using divide and conquer approach.
18. Given a value V, if we want to make change for V Rs., and we have infinite supply of each of the denominations in Indian currency, i.e., we have infinite supply of {1, 2, 5, 10, 20, 50, 100, 200, 2000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change? Implement the greedy algorithm to find minimum number of coins.
19. Find the minimum cost spanning tree of a given undirected graph using:
    a) Kruskal's algorithm
    b) Prim's algorithm
20. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.
21. Implement the longest ascending subsequence problem using dynamic programming.

22. Implement matrix chain multiplication problem using dynamic programming.
23. Implement 0/1 Knapsack problem using dynamic programming.
24. Write a program to find the shortest path using Bellman-Ford algorithm.
25. Implement branch and bound scheme to find the optimal solution for:
    a) Job assignment problem
    b) Traveling salesperson problem.

Table of Contents

Program 1

Aim: Use divide and conquer method to recursively implement:

a) **Binary search**

b) **Linear search**

Introduction:

In binary search algorithm, we use the method of divide-and-conquer. The prerequisite for this algorithm is that the array must be sorted. In linear search algorithm, we compare targeted element with each element of the array. If the element is found, then its position is displayed.

Approach:

In binary search algorithm, we use two variables start (s) and end (e). In each iteration, we calculate the mid index and compare the mid element with the required element to be searched. If matched, we return the mid index, otherwise, we discard one half based on some condition.

Linear search works as follows:

1. We keep on comparing each element with the element to search until it is found or the list ends.
2. We use an index variable which starts from the last index and keeps on decreasing and comparing the elements. If found it immediately return that index otherwise if index became negative it means we have exhausted the array and not found the element.

Code:

(a) Binary search

```
#include <bits/stdc++.h>
using namespace std;
int find_index_Binery_Search(int arr[], int target, int l,int h)
{
    if(l>h) return -1;
    int mid=l+(h-l)/2;
    if(arr[mid]==target) return mid;
    if(arr[mid]<target){//check in right half
        return find_index_Binery_Search(arr,target,mid+1,h);
    }
    //check in left half
    return find_index_Binery_Search(arr,target,l,mid-1);
```

```
}
int main()
{
    int n, target;
    cout << "Enter the Size: ";
    cin >> n;
    cout << endl;
    cout << "Enter the Target: ";
    cin >> target;
    cout << endl;
    int arr[n];
    cout << "Enter the Array Elements in Sorted Form: ";
    for (int i = 0; i < n; i++) cin >> arr[i];
    cout << "The Index of the Number we are find is: " <<
find_index_Binery_Search(arr, target, 0,n-1) << endl;
    return 0;
}
```

Output:

```
Enter the size: 10
Enter the Target: 3
Enter the Array Elements in Sorted Form: 1 2 3 4 5 6 7 8 9 10
The Index of the Number we are find is:  = 2
```

Complexity analysis:

Time complexity: O(log(n))

Space complexity: It consumes recursion call stack space as **O(log(n))**.

(b) Linear search

```
#include <bits/stdc++.h>
using namespace std;
int findindex(int ind, int arr[], int target, int n)
{
    if (ind == n) return -1;
    if (arr[ind] == target) return ind;
    return findindex(ind + 1, arr, target, n);
}
int main()
{
    int n, target;
    cout << "Enter the Size: ";
    cin >> n;
```

```
    cout << endl;
    cout << "Enter the Target: ";
    cin >> target;
    cout << endl;
    int arr[n];
    cout << "Enter the Array: ";
    for (int i = 0; i < n; i++) cin >> arr[i];
    cout << "The Index of the Number we are find is: " <<
findindex(0, arr, target, n) << endl;
    return 0;
}
```

Output:

```
Enter the size: 10
Enter the Target: 3
Enter the Array Elements: 1 2 3 4 5 6 7 8 9 10
The Index of the Number we are find is:  = 2
```

Complexity analysis:

Time complexity: O(N)

Space complexity: it consumes Recursion call stack space as O(N).

Program 2

Aim: Sort a given set of elements using the insertion sort method.

Introduction:

In insertion sort, we are trying to insert the element to its correct position into the array. In this method of sorting the array gets sorted from left to right side.

Approach:

1. We use an outer loop which runs for n-1 times.
2. For each iteration of the outer loop, we run an inner loop which will insert the $\mathbf{i}^{th}$ element to its correct position in the left side sorted array.

Code:

```
#include <bits/stdc++.h>
using namespace std;
void insertionSort(vector<int> &arr)
{
    int n = arr.size();
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while (j > 0 && arr[j] < arr[j - 1])
        {
            swap(arr[j], arr[j - 1]);
            j--;
        }
    }
}
void rec(vector<int> &arr, int ind)
{
    if (ind == arr.size())return;
    int j = ind;
    while (j > 0 && arr[j] < arr[j - 1])
    {
        swap(arr[j], arr[j - 1]);
        j--;
    }
    rec(arr, ind + 1);
}
int main()
```

```
{
    vector<int> arr;
    int n;
    cout << "Enter the size of array : ";
    cin >> n;
    cout << "Enter " << n << " elements : ";
    for (int i = 0; i < n; i++)
    {
        int a;
        cin >> a;
        arr.push_back(a);
    }
    cout << "Sorted array : ";
    // insertionSort(arr);
    rec(arr, 0);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Output:

```
Enter the size of array : 8
Enter 8 elements :  6 5 7 0 1 2 3 4
Sorted array : 0 1 2 3 4 5 6 7
```

Complexity analysis:

Time complexity:

a) Best case – O(N)
b) Worst case – $O(N^2)$

Space complexity: O(1) since we are not using any extra memory space.

Program 3

Aim: Sort a given set of elements using the selection sort method.

Introduction:

In selection sort, we are trying to select the current element index and search for any index whose element is lower than the element at current if so exit then swap the element.

Approach:

1. We use an outer loop which runs for **n-2** times.
2. For each iteration of the outer loop, we run an inner loop which will search for the index whose value is lesser than the current index.
3. If the element is lesser than the value of current index, then we swap it.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int min_ind(int a[], int l, int h)
{
    int ind = l;
    for (int i = l + 1; i <= h; i++)
    {
        if (a[ind] > a[i]) ind = i;
    }
    return ind;
}
int main()
{
    int n;
    cout << "Enter the Size of the Array: ";
    cin >> n;
    cout << endl;
    int a[n];
    cout << "Enter the Array: ";
    for (int i = 0; i < n; i++) cin >> a[i];
    for (int i = 0; i < n - 1; i++)
    {
        int min ind = min_ind(a, i + 1, n - 1);
        if (a[minind] < a[i]) swap(a[minind], a[i]);
    }
```

```
    cout<<"Sorted Array: ";
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}
```

Output:

```
Enter the size of the Array : 8
Enter the Array elements :  6 5 7 0 1 2 3 4
Sorted array : 0 1 2 3 4 5 6 7
```

Complexity analysis:

Time complexity: $O(N^2)$

Space complexity: $O(N)$

Program 4

Aim: Sort a given set of elements using the quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Introduction:

In quick sort, we are trying to find the pivot element before which all the elements are lesser than and after that all the elements are greater.

Approach:

Quick sort works on the principle of **divide-and-conquer**, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

1. **Choose a pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursive call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. **Base case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;
int partition(vector<int> &arr, int l, int h)
{
    int x = arr[h];
    int i = l - 1;
    for (int j = l; j < h; j++)
    {
        if (arr[j] <= x)
        {
            i = i + 1;
```

```
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[h]);
    return i + 1;
}
void quick_sort(vector<int> &arr, int l, int h)
{
    if (l < h)
    {
        int q = partition(arr, l, h);
        quick_sort(arr, l, q - 1);
        quick_sort(arr, q + 1, h);
    }
}
int main()
{
    int n;
    cout << "Enter the size of the Array: ";
    cin >> n;
    cout << endl;
    vector<int> arr(n);
    clock_t start, end;
    start = clock();
    cout << "Enter the Array Elements: ";
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % n;
    }
    cout << endl;
    quick_sort(arr, 0, n - 1);
    end = clock();
    cout << "Sorted Array: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Time Taken " << double(end - start) << " microsecond"
<< endl;
}
```

Output:

```
Enter the size of the Array : 8
Enter the Array elements :  6 5 7 0 1 2 3 4
Sorted Array : 0 1 2 3 4 5 6 7
Time Taken : 0.0 microsecond
```

Complexity analysis:

Time complexity:

a) **Average case: O(N logN)** when both partition have random elements.

b) **Worst case**: **O(N^2)**, for example, in case where the elements in one part of the array are already sorted. Then, one part will have n-1 elements and the other one will be empty after call to the partition function.

Space complexity: O(1)

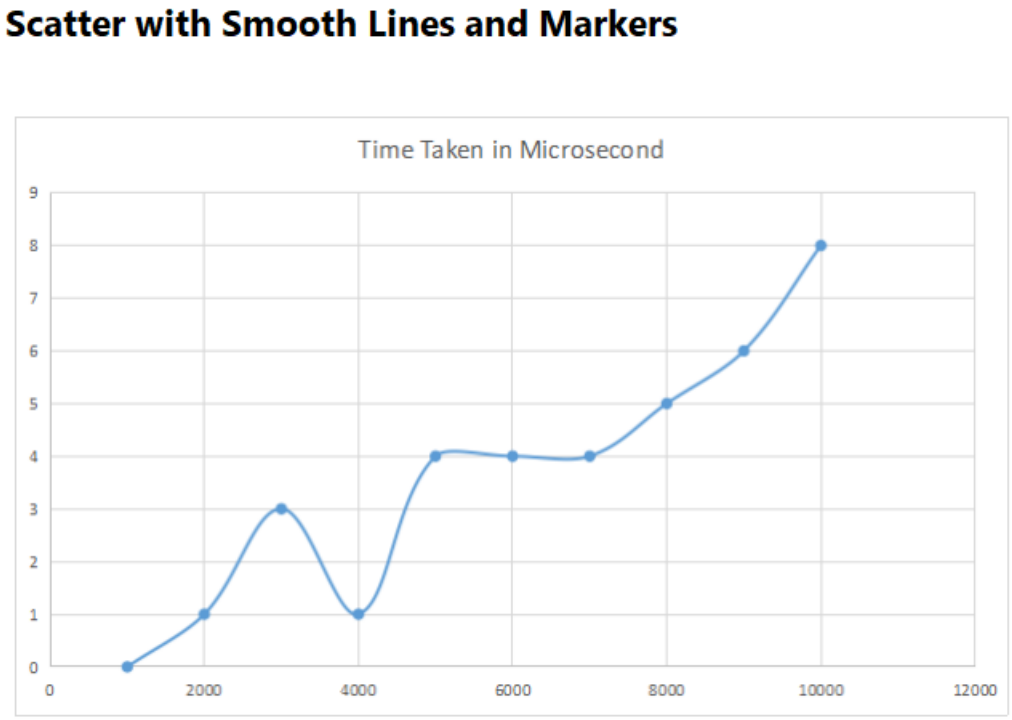

Fig 4.1: The x- axis shows n, the number of elements in the list to be sorted and the y-axis shows the time in microseconds. It plots a graph of the time taken by the quick sort algorithm for different values of n.

Program 5

Aim: Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Introduction:

In merge sort, we divide the array into two halves and then merge the two halves while sorting.

Approach:

Merge sort follows the **divide-and-conquer** approach to sort a given array of elements.

Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Code:

```
#include <bits/stdc++.h>
using namespace std;
void merge_array(int v[], int p, int mid, int r)
{
    int b[r];
    int k = p;
    int i = p;
    int j = mid + 1;

    while (i <= mid && j <= r)
    {
        if (v[i] <= v[j])
        {
            b[k] = v[i];
            k++;i++;
        }
```

```
        else if (v[i] > v[j])
        {
            b[k] = v[j];
            j++;k++;
        }
    }
    if (i <= mid)
    {
        while (i <= mid)
        {
            b[k] = v[i];
            i++; k++;
        }
    }
    else if (j <= r)
    {
        while (j <= r)
        {
            b[k] = v[j];
            j++; k++;
        }
    }
    for (int l = p; l <= r; l++)
    {
        v[l] = b[l];
    }
}
void merge_sort(int v[], int p, int r)
{
    if (p < r)
    {
        int mid = (p + r) / 2;
        merge_sort(v, p, mid);
        merge_sort(v, mid + 1, r);
        merge_array(v, p, mid, r);
    }
}
int main()
{
    int n;
    cout << "Enter the size of the array:-> ";
    cin >> n;
    int arr[n] = {0};
```

```
    clock_t start, end;
    start = clock();

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    merge_sort(arr, 0, n - 1);
    end = clock();
    cout << "Sorted Array Elements: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Time taken " << (end - start) << endl;
    return 0;
}
```

Output:

```
Enter the size of the array:-> 10
Sorted Array Elements: 0 24 34 41 58 62 64 67 69 78
Time taken 0
```

Complexity analysis:

Time complexity: O(N logN)
Space complexity: O(1)

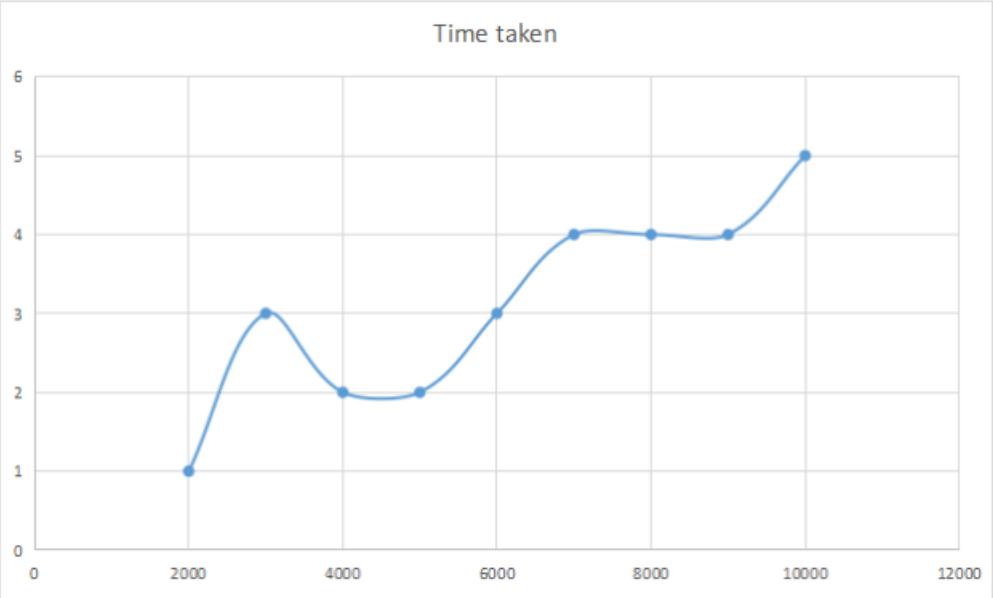

Fig 5.1: The x- axis shows n, the number of elements in the list to be sorted and the y-axis shows the time in microseconds. It plots a graph of the time taken by the merge sort algorithm for different values of n.

Program 6

Aim: Sort a given set of elements using the heap sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Introduction:

Heap sort is a comparison-based sorting technique based on binary heap. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In heap sort, we use binary heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

Approach:

First convert the array into a max heap using heapify. In this way, the array elements are re-arranged to follow heap properties. Then one by one, delete the root node of the max heap and replace it with the last node and heapify. Repeat this process while size of heap is greater than 1.

- Rearrange array elements so that they form a max heap.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.
    - Heapify the remaining elements of the heap.
- Finally, we get sorted array.

Code:

```
#include <bits/stdc++.h>
using namespace std;
void heapify(vector<int> &v, int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && v[largest] <= v[l])
```

```
    {
        largest = l;
    }
    if (r < n && v[r] >= v[largest])
    {
        largest = r;
    }
    if (largest != i)
    {
        swap(v[i], v[largest]);
        heapify(v, n, largest);
    }
}

void Build_Max_Heap(vector<int> &v)
{
    int n = v.size();
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        heapify(v, n, i);
    }
}
vector<int> Heap_sort(vector<int> &v)
{
    Build_Max_Heap(v);
    vector<int> ans;
    for (int i = v.size() - 1; i >= 0; i--)
    {
        ans.push_back(v[0]);
        swap(v[0], v[i]);
        heapify(v, i, 0);
    }
    return ans;
}
int max_element(vector<int> &v)
{
    int n = v.size();
    if (n == 0)
        return -1;
    int mx = v[0];
    return mx;
}

void heap_increase(vector<int> &v, int i, int key)
{
    if (key < v[i])
    {
        cout << "Error" << endl;
```

```
        return;
    }
    v[i] = key;
    while (i >= 0 && v[ceil(float(i)/2.0)-1] < v[i])
    {
        swap(v[i], v[ceil(float(i)/2.0)-1]);
        i = ceil(float(i)/2.0)-1;
    }
}
void insert_element(vector<int> &v, int key)
{
    v.push_back(key);
    heapify(v, v.size(), 0);
}
int main()
{
    int n;
    cout << "Enter the Size of the Array: ";
    cin >> n;
    vector<int> v1(n, 0);
    clock_t start1, end1; // for heap_sort
    start1 = clock();
    for (int i = 0; i < n; i++)
    {
        v1[i] = rand() % 100 +1;
    }
    cout << endl
         << "Entered Elements of the Array: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cout << v1[i] << " ";
    }
    cout << endl;
    vector<int> sortedv1 = Heap_sort(v1);
    end1 = clock();
    cout << "Sorted Array: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cout << sortedv1[i] << " ";
    }
    cout << endl;
    cout << "Time for Heap Sort: " << end1 - start1 << endl<< endl;
    // for maxelement
    clock_t start2, end2;
    start2 = clock();
    int maxelement = max_element(sortedv1);
    end2 = clock();
    cout << "Max element: " << maxelement << endl;
```

```
    cout << "Time for Max Element : " << end2 - start2 << endl
         << endl;
    // insert Element
    clock_t start3, end3;
    int element;
    cout << "Enter the value you want to insert ";
    cin >> element;
    cout << "Array before the insert " << endl;
    for (int i = 0; i < sortedv1.size(); i++)
    {
        cout << sortedv1[i] << " ";
    }
    cout << endl;
    start3 = clock();
    insert_element(sortedv1, element);
    vector<int> v2=Heap_sort(sortedv1);
    end3 = clock();
    cout << "Array after the insert " << endl;
    for (int i = 0; i < v2.size(); i++)
    {
        cout << v2[i] << " ";
    }
    cout << endl;
    cout << "Time for insert element " << end3 - start3 << endl<<endl;
    // for Value change
    clock_t start4, end4;
    int index;
    cout << "Enter the Index where you want to change the value: ";
    cin >> index;
    int key;
    cout << "Enter the value: ";
    cin >> key;
    cout << "Array before the change at the ith index " << endl;
    for (int i = 0; i < v2.size(); i++)
    {
        cout << v2[i] << " ";
    }
    cout << endl;
    start4 = clock();
    heap_increase(v2, index, key);
    end4 = clock();
    cout << "Elements of the array after the change at the ith index " << endl;
    for (int i = 0; i < v2.size(); i++)
    {
        cout << v2[i] << " ";
    }
    cout << endl;
```

```
    cout << "Time for the changing the element at " << index << " th index "
<< end4 - start4 << endl;
    return 0;
}
```

Output:

```
Enter the Size of the Array: 10

Entered Elements of the Array:
42 68 35 1 70 25 79 59 63 65
Sorted Array:
79 70 68 65 63 59 42 35 25 1
Time for Heap Sort: 6

Max element: 79
Time for Max Element : 0

Enter the value you want to insert 90
Array before the insert
79 70 68 65 63 59 42 35 25 1
Array after the insert
90 79 70 68 65 63 59 42 35 25 1
Time for insert element 1

Enter the Index where you want to change the value: 7
Enter the value: 100
Array before the change at the ith index
90 79 70 68 65 63 59 42 35 25 1
Elements of the array after the change at the ith index
100 90 70 79 65 63 59 68 35 25 1
Time for the changing the element at 7 th index 0
```

Complexity analysis:

	Insert	Search	Find_min	Del_min	Increase_key
Time Complexity	O(LogN)	O(N)	O(1)	O(LogN)	O(LogN)
Space Complexity	O(N)	O(N)	O(N)	O(N)	O(N)

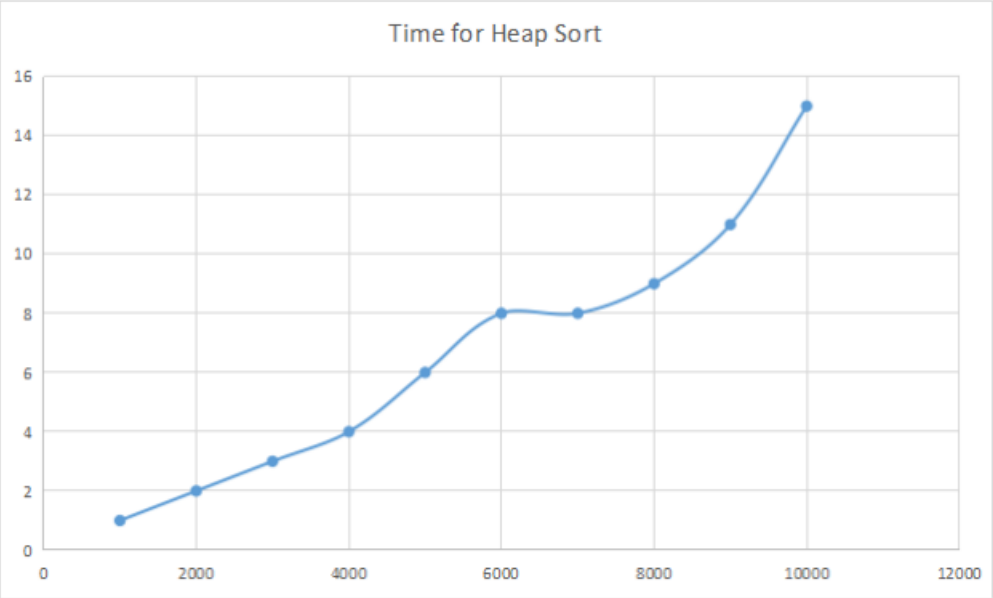

Fig 6.1: The x- axis shows n, the number of elements in the list to be sorted and the y-axis shows the time in microseconds. It plots a graph of the time taken by the heap sort algorithm for different values of n.

Program 7

Aim: From a given vertex in an unweighted connected graph, find the shortest paths to other vertices using BFS.

Introduction:

As the aim suggests, here we will have one vertex named source vertex from which we will find the shortest paths to all other nodes using BFS.
BFS is one of the methods of graph traversal where we first explore all the neighbours, then neighbours of neighbours, and so on.
BFS traversal is done by using queue data structure.

Approach:

The idea is to use a modified version of BFS in which we keep storing the parent of a given vertex while doing the breadth-first search. We first initialize an array **`dist[0, 1, …., v-1]`** such that **`dist[i]`** stores the distance of vertex **`i`** from the source vertex and array **`par[0, 1, …., v-1]`** such that **`par[i]`** represents the parent of the vertex **`i`** in the breadth-first search starting from the source.

Now we get the length of the path from source to any other vertex from array **`dist[]`**, and for printing the path from source to any vertex we can use array **`par[]`**.

Code:

```
#include <bits/stdc++.h>
using namespace std;
vector<pair<int, int>> BFS(vector<int> adj[], int src, vector<int>
&vis,vector<int>&parent)
{
    vector<pair<int, int>> ans;
    // queue of the pair node and level from the source node
    queue<pair<int, int>> q;
    q.push({src, 0});
    vis[src] = 1;
    while (!q.empty())
    {
        int node = q.front().first;
        int level = q.front().second;
        ans.push_back({node, level});
        q.pop();
        for (auto it : adj[node])
```

```
        {
            if (vis[it] == 0)
            {
                vis[it] = 1;
                parent[it]=node;
                q.push({it, level + 1});
            }
        }
    }
    return ans;
}
int main()
{
    int n, e;
    cout << "Enter the number of nodes ";
    cin >> n;
    cout << "Enter the number of edges ";
    cin >> e;
    cout << "Enter the nodes :" << endl;

    vector<vector<int>> v(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> v[i][j];
        }
    }
    vector<int> adj[n + 1];
    for (int i = 0; i < e; i++)
    {
        adj[v[i][0]].push_back(v[i][1]);
        adj[v[i][1]].push_back(v[i][0]);
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
```

```
    }
    // visited array
    vector<int> vis(n + 1, 0);
    vector<int> parent(n+1,-1);
    // BFS CALL
    vector<pair<int, int>> ans = BFS(adj, 1, vis,parent);
    vector<int>path[n+1];
    for(int i=0;i<ans.size();i++){
        int node=ans[i].first;
       vector<int>cur;
       while(node!=-1){
          cur.push_back(node);
          node=parent[node];
       }
       path[ans[i].first]=cur;
    }
    for(int i=0;i<ans.size();i++){
        cout<<"Node: "<<ans[i].first<<"  Dis: "<<ans[i].second<<"
"<<"Path: ";
         for(auto it:path[ans[i].first]){
            cout<<it<<"->";
         }cout<<"-1"<<endl;
    }
    return 0;
}
```

Output:

```
Enter the number of nodes 10
Enter the number of edges 13
Enter the nodes :
1 2
1 3
1 4
2 3
4 5
4 8
5 6
5 7
6 7
6 9
6 8
8 9
```

```
9 10
The Adjacency List:
1: 2 3 4
2: 1 3
3: 1 2
4: 1 5 8
5: 4 6 7
6: 5 7 9 8
7: 5 6
8: 4 6 9
9: 6 8 10
10: 9
Node: 1  Dis: 0 Path: 1->-1
Node: 2  Dis: 1 Path: 2->1->-1
Node: 3  Dis: 1 Path: 3->1->-1
Node: 4  Dis: 1 Path: 4->1->-1
Node: 5  Dis: 2 Path: 5->4->1->-1
Node: 8  Dis: 2 Path: 8->4->1->-1
Node: 6  Dis: 3 Path: 6->5->4->1->-1
Node: 7  Dis: 3 Path: 7->5->4->1->-1
Node: 9  Dis: 3 Path: 9->8->4->1->-1
Node: 10  Dis: 4 Path: 10->9->8->4->1->-1
```

Complexity analysis:

Time complexity: O(N+E)

Space complexity: O(N)

Program 8

Aim: Given an undirected graph, use BFS to check if there is a cycle in the graph.

Introduction:

In an undirected graph, we can detect a cycle using BFS traversal. A cycle in an undirected graph means that we can reach the same vertex after covering some vertices. The basic idea is to keep track of visited nodes during BFS traversal. If a node is encountered more than once during BFS, it indicates the presence of a cycle in the graph.

Example:

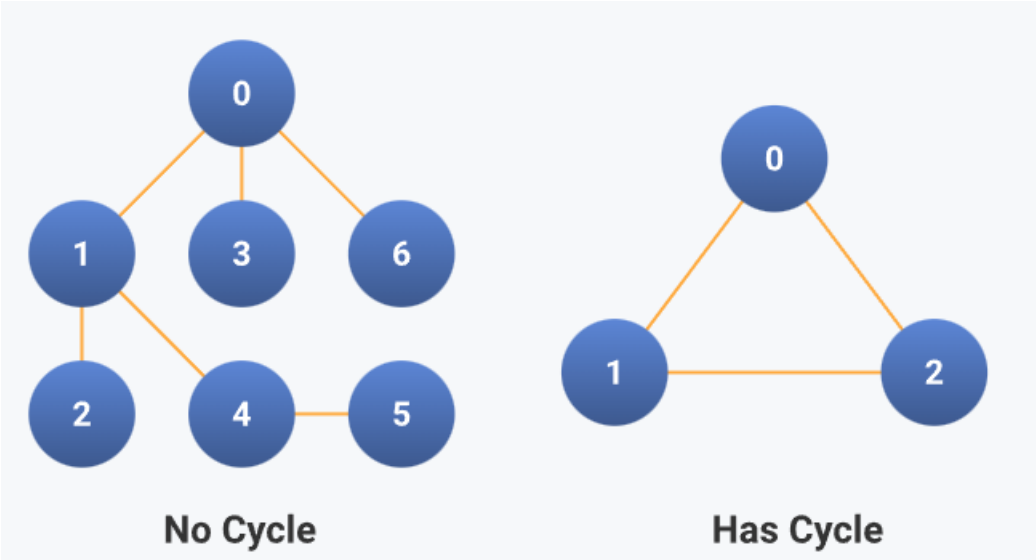

Fig. 8.1: Examples of graphs without a cycle (on left) and with a cycle (on right)

A cycle in graph theory is a path that originates at one vertex and terminates at the same vertex. In Fig. 8.1, we can see there is no cycle in the left graph, but there is a cycle in the right graph.

Approach:

1. Start BFS traversal from each unvisited node in the graph.
2. While traversing, mark each visited node.
3. If a node is encountered that is already marked as visited, it implies the presence of a cycle.
4. Continue BFS traversal until all nodes are visited or a cycle is detected.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

bool Cycle_Check(int src, vector<int> &vis, vector<int> &par,
vector<int> adj[])
{
    queue<int> q;
    vis[src] = 1;

    q.push(src);
    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        for (auto it : adj[node])
        {
            if (vis[it] == 0)
            {
                vis[it] = 1;
                par[it] = node;

                q.push(it);
            }
            else if (par[node] != it)
            {
                return true;
            }
        }
    }
    return false;
}
int main()
{
    int n, e;
    cout << "Enter the number of nodes ";
    cin >> n;
    cout << "Enter the number of edges ";
    cin >> e;
    cout << "Enter the nodes :" << endl;

    vector<vector<int>> v(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
```

```
        for (int j = 0; j < 2; j++)
        {
            cin >> v[i][j];
        }
    }

    vector<int> adj[n + 1];
    for (int i = 0; i < e; i++)
    {
        adj[v[i][0]].push_back(v[i][1]);
        adj[v[i][1]].push_back(v[i][0]);
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
    }
    cout << endl;
    vector<int> vis(n + 1, 0);
    vector<int> par(n + 1, -1);
    bool flag=false;
    for (int i = 1; i <= n; i++)
    {
        if (vis[i] == 0)
        {
            if(Cycle_Check(i,vis,par,adj)){
                flag=true;
                break;
            }
        }
    }
    if(flag){
        cout<<"There is cycle in the graph. "<<endl;
    }
    else {
        cout<<"There is no cycle in the graph. "<<endl;
    }
    return 0;
```

```
}
```

Output:

```
Enter the number of nodes 11
Enter the number of edges 10
Enter the nodes :
1 2
1 3
2 4
3 4
5 6
7 8
7 10
8 9
9 11
10 11
The Adjacency List:
1: 2 3
2: 1 4
3: 1 4
4: 2 3
5: 6
6: 5
7: 8 10
8: 7 9
9: 8 11
10: 7 11
11: 9 10

There is cycle in the graph.

Enter the number of nodes 5
Enter the number of edges 4
Enter the nodes :
1 2
2 3
3 4
3 5
The Adjacency List:
1: 2
2: 1 3
3: 2 4 5
```

```
4: 3
5: 3

There is no cycle in the graph.
```

Complexity analysis:

Time complexity: O(N+E)

Space complexity:O(N+E) for the `parent` and `visited` vectors and for the queue.

Program 9

Aim: Use BFS to find all connected components in a graph and finding all vertices within a connected component.

Introduction:

In graph theory, a component of an undirected graph is a connected subgraph that is not part of any larger connected sub-graph. The idea is to perform either BFS or DFS starting from every unvisited vertex, and we get all connected components.

Approach:

1. **Initialize Data Structures**:
    - Create a `visited` set (or array) to keep track of nodes that have already been visited.
    - Create an empty list called `components` to store each connected component found in the graph.
2. **Loop Over Each Node**:
    - For each node in the graph, check if it has been visited. If it has, skip it. If it hasn't, this node starts a new connected component.
3. **BFS Traversal**:
    - Start a BFS from this unvisited node:
        - Initialize a queue and add the starting node.
    - Create an empty list, `component`, to store the nodes of the current connected component.
    - While the queue is not empty:
    - Dequeue a node, mark it as visited, and add it to the `component` list.
        - For each neighbor of the current node:
            - If the neighbor has not been visited, add it to the queue and mark it as visited.
4. **Store the Component**:
    - Once the BFS completes for a starting node, add the `component` list to the `components` list.
5. **Repeat Until All Nodes are Visited**:
    - Continue the process for each unvisited node until all nodes have been visited.
6. **Return the Result**:
    - After all nodes have been processed, the `components` list will contain each connected component in the graph as a separate list.

Code:

```
#include <bits/stdc++.h>
using namespace std;

vector<int> connected_comp(int src, vector<int> &vis, vector<int>
adj[], vector<int> &cur)
{
    queue<int> q;
    vis[src] = 1;
    cur.push_back(src);
    q.push(src);
    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        for (auto it : adj[node])
        {
            if (vis[it] == 0)
            {
                vis[it] = 1;
                ;
                cur.push_back(it);
                q.push(it);
            }
        }
    }
    return cur;
}
int main()
{
    int n, e;
    cout << "Enter the number of nodes ";
    cin >> n;
    cout << "Enter the number of edges ";
    cin >> e;
    cout << "Enter the nodes :" << endl;

    vector<vector<int>> v(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
        for (int j = 0; j < 2; j++)
        {
```

```
            cin >> v[i][j];
        }
    }

    vector<int> adj[n + 1];
    for (int i = 0; i < e; i++)
    {
        adj[v[i][0]].push_back(v[i][1]);
        adj[v[i][1]].push_back(v[i][0]);
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
    }
    cout << endl;
    vector<int> vis(n + 1, 0);
    vector<vector<int>> ans;
    for (int i = 1; i <= n; i++)
    {
        vector<int> cur;
        if (vis[i] == 0)
        {
            connected_comp(i, vis, adj, cur);
        }
        ans.push_back(cur);
        cur.clear();
    }
    cout << "Components " << endl;
    for (auto it : ans)
    {
        for (auto i : it)
        {
            cout << i << " ";
        }cout<<endl;
    }
    return 0;
}
```

Output:

```
Enter the number of nodes 11
Enter the number of edges 10
Enter the nodes :
1 2
1 3
2 4
3 4
5 6
7 8
7 10
8 9
9 11
10 11
The Adjacency List:
1: 2 3
2: 1 4
3: 1 4
4: 2 3
5: 6
6: 5
7: 8 10
8: 7 9
9: 8 11
10: 7 11
11: 9 10

Components
1 2 3 4
5 6
7 8 10 9 11
```

Complexity analysis:

Time complexity: O(N)

Space complexity: It consumes recursion call stack space as O(N).

Program 10

Aim: Given a directed graph, use DFS to check if there is a cycle in the graph.

Introduction:

To find cycle in a directed graph we can use the (DFS) technique. It is based on the idea that there is a cycle in a graph only if there is a back edge [i.e., a node points to one of its ancestors in a DFS tree] present in the graph.

To detect a back edge, we need to keep track of the visited nodes that are in the current recursion stack [i.e., the current path that we are visiting]. Note that all ancestors of a node are present in recursion call stack during DFS. So, if there is an edge to an ancestor in DFS, then this is a back edge.

Approach:

1. **Initialize Data Structures**:
   - Maintain a `visited` set to track visited nodes.
   - Use two dictionaries: `entry` and `exit`, to store the entry and exit times of each node during DFS traversal.
   - Maintain a counter, `time`, to record the entry and exit times.
2. **Define DFS Function with Numbering**:
   - Define a recursive function `dfs(node)` that:
     - Marks the current node as visited.
     - Records the `entry` time for the node and increments `time`.
     - For each neighbor of the current node:
       - If the neighbor has not been visited, recursively call `dfs(neighbor)`.
         - If this recursive call detects a cycle (returns `True`), propagate `True` to indicate a cycle was found.
       - If the neighbor is already visited:
         - Check if the entry and exit times indicate a back edge:
           - If `neighbor` has an entry time earlier than `node`'s entry time and does not have an exit time yet (indicating it's still in the recursion stack), a back edge exists, which confirms a cycle.
     - After exploring all neighbors, record the `exit` time for the node and increment `time`.
3. **Loop Over Each Node**:
   - For each unvisited node in the graph, call `dfs(node)`.

- If any DFS call detects a cycle, return `True` immediately.

4. **Return the Result**:
   - If no cycle is found after checking all nodes, return `False`.

Code:

```
#include <bits/stdc++.h>
using namespace std;

void dfs(int node, vector<int> adj[], vector<int> &pre, vector<int>
&post, vector<int> &vis, int &cnt)
{
    vis[node] = 1;
    for (auto it : adj[node])
    {
        if (vis[it] == 0)
        {
            pre[it] = cnt;
            cnt++;
            dfs(it, adj, pre, post, vis, cnt);
        }
    }
    post[node] = cnt;
    cnt++;
    return;
}
int main()
{
    int n;
    cout << "Enter the Number of vertices: ";
    cin >> n;
    int e;
    cout << "Enter the Number of Edges: ";
    cin >> e;
    vector<vector<int>> edges(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> edges[i][j];
        }
    }
```

```
    vector<int> adj[n + 1];
    for (int i = 0; i < e; i++)
    {
        adj[edges[i][0]].push_back(edges[i][1]);
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
    }
    cout << endl;

    vector<int> pre(n + 1, 0);
    vector<int> post(n + 1, 0);
    vector<int> vis(n + 1, 0);
    int cnt = 0;
    for (int i = 1; i <= n; i++)
    {
        if (vis[i] == 0)
        {
            pre[i] = cnt;
            cnt++;
            dfs(i, adj, pre, post, vis, cnt);
        }
    }
    cout << "Pre and Post of each node: " << endl;
    cout << "Node: "
         << "Pre "
         << "Post" << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << "      " << pre[i] << "     " << post[i] << endl;
    }
    // check back edge using (pre[j],post[j]) contain
(pre[i],post[i])
    int flag = 0;
    for (int i = 0; i < e; i++)
    {
```

```
        int u = edges[i][0];
        int v = edges[i][1];
        if (pre[v] < pre[u] && post[u] < post[v])
        {
            cout << "The first occurrence of back edges between
nodes " << u << " " << v << endl;
            flag = 1;
            break;
        }
    }
    if (flag)
    {
        cout << "Yes, there is Cycle in the graph " << endl;
    }
    else
        cout << "No, there is no Cycle in the graph " << endl;
}
```

Output:

```
Enter the Number of vertices: 5
Enter the Number of Edges: 5
1 2
2 3
3 4
4 5
5 2
The Adjacency List:
1: 2
2: 3
3: 4
4: 5
5: 2

Pre and Post of each node:
Node: Pre Post
1     0    9
2     1    8
3     2    7
4     3    6
5     4    5
The first occurrence of back edges between nodes 5 2
Yes, there is Cycle in the graph
```

```
Enter the Number of vertices: 5
Enter the Number of Edges: 4
1 2
2 3
3 4
4 5
The Adjacency List:
1: 2
2: 3
3: 4
4: 5
5:

Pre and Post of each node:
Node: Pre Post
1     0    9
2     1    8
3     2    7
4     3    6
5     4    5
No, there is no Cycle in the graph
```

Complexity analysis:

Time complexity: O(N+E)

Space complexity: O(N+E) for the `parent` and `visited` vectors, a call stack of O(N) and O(2*E) for storing the edges.

Program 11

Aim: Check whether a given graph is connected or not using DFS method.

Introduction:

In graph theory, a component of an undirected graph is a connected subgraph that is not part of any larger connected sub-graph. The basic idea of DFS is this: it methodically explores every edge. We start over from different vertices as necessary. As soon as we discover a vertex, DFS starts exploring from it.

Approach:

1. **Choose a Starting Node**:
   - Select any node in the graph as the starting point for DFS.
2. **Initialize Data Structures**:
   - Create a visited set (or array) to keep track of nodes that have been visited.
3. **Perform DFS Traversal**:
   - Define a recursive DFS function that:
     - Marks the current node as visited.
     - For each neighbor of the current node, if it hasn't been visited, recursively call the DFS function on that neighbor.
4. **Check if All Nodes Are Visited**:
   - After the DFS completes, check if the size of the visited set is equal to the total number of nodes in the graph.
     - If all nodes are visited, the graph is connected.
     - If not, the graph is disconnected.

Code:

```
#include <bits/stdc++.h>
using namespace std;

void connected_comp(int src, vector<int> &vis, vector<int> adj[],
vector<int> &cur)
{
    queue<int> q;
    vis[src] = 1;
    cur.push_back(src);
    for(auto it:adj[src]){
        if(vis[it]==0){
            connected_comp(it,vis,adj,cur);
        }
```

```
    }

}
int main()
{
    int n, e;
    cout << "Enter the number of nodes ";
    cin >> n;
    cout << "Enter the number of edges ";
    cin >> e;
    cout << "Enter the nodes :" << endl;

    vector<vector<int>> v(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> v[i][j];
        }
    }

    vector<int> adj[n + 1];
    for (int i = 0; i < e; i++)
    {
        adj[v[i][0]].push_back(v[i][1]);
        adj[v[i][1]].push_back(v[i][0]);
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
    }
    cout << endl;
    vector<int> vis(n + 1, 0);
    vector<vector<int>> ans;
    for (int i = 1; i <= n; i++)
    {
        vector<int> cur;
        if (vis[i] == 0)
```

```
        {
            connected_comp(i, vis, adj, cur);
        }
        ans.push_back(cur);

    }
    cout << "Components " << endl;
    for (auto it : ans)
    {
        for (auto i : it)
        {
            cout << i << " ";
        }cout<<endl;
    }
    return 0;
}
```

Output:

```
Enter the number of nodes 11
Enter the number of edges 10
Enter the nodes :
1 2
1 3
2 4
3 4
5 6
7 8
7 10
8 9
9 11
10 11
The Adjacency List:
1: 2 3
2: 1 4
3: 1 4
4: 2 3
5: 6
6: 5
7: 8 10
8: 7 9
9: 8 11
10: 7 11
11: 9 10
```

```
Components
1 2 3 4
5 6
7 8  9 11 10
```

Complexity analysis:

Time complexity: O(N+E)

Space complexity: It consumes recursion call stack space as O(N) and O(N) for storing the solution in the form of a 2D vector.

Program 12

Aim: Use DFS to find all strongly connected components in a directed graph.

Introduction:

In a directed graph, a strongly connected component (SCC) is a subset of vertices where every vertex in the subset is reachable from every other vertex in the same subset by traversing the directed edges. Finding the SCCs of a graph can provide important insights into the structure and connectivity of the graph, with applications in various fields such as social network analysis, web crawling, and network routing.

Example:

The graph shown in Fig. 12.1 has two strongly connected components {1,2,3,4} and {5,6,7} since there is path from each vertex to every other vertex in the same strongly connected component.

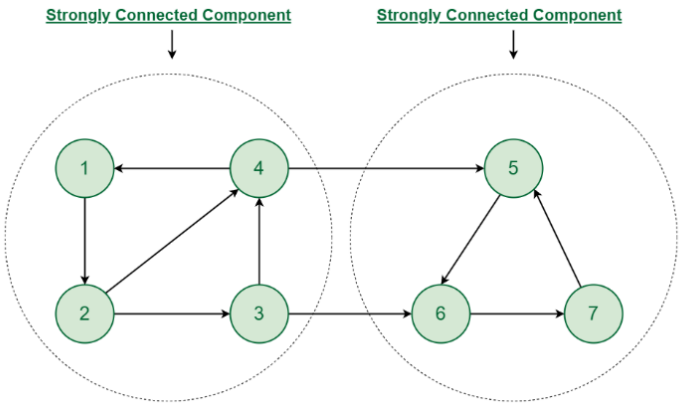

Fig. 12.1: Two strongly connected components in a directed graph

Approach:

1. **Initialize Data Structures**:
   - Create an empty stack `finish_stack` to store nodes in the order of their exit times (to perform the second DFS pass in this order).
   - Create a `visited` set to keep track of visited nodes during DFS traversals.
2. **First DFS Pass (Original Graph)**:
   - Perform DFS on each unvisited node to determine the **finish times** for all nodes.
   - For each node visited in DFS, record its exit time by pushing it onto the `finish_stack` when the DFS finishes for that node.
3. **Transpose the Graph**:
   - Create a transposed version of the graph by reversing the direction of all edges.

- This helps in identifying SCCs, as any SCC in the original graph will remain connected in the transposed graph but with reversed edges.

4. **Second DFS Pass (Transposed Graph)**:
   - Reset the `visited` set to start a new DFS pass.
   - Perform DFS on each node in the order defined by the `finish_stack` (i.e., in decreasing order of exit times from the first DFS pass).
   - For each DFS call, collect all nodes visited in this call into an SCC component and add this component to the list of SCCs.
   - Each DFS traversal in this pass will reveal a strongly connected component.
5. **Return the SCCs**:
   - After the second DFS pass, all SCCs will be found.

Code:

```
#include <bits/stdc++.h>
using namespace std;

void dfs(int node, vector<int> adj[], vector<int> &pre, vector<int>
&post, vector<int> &vis, int &cnt)
{
    vis[node] = 1;
    for (auto it : adj[node])
    {
        if (vis[it] == 0)
        {
            pre[it] = cnt;
            cnt++;
            dfs(it, adj, pre, post, vis, cnt);
        }
    }
    post[node] = cnt;
    cnt++;
    return;
}
void dfscc(int
node,vector<int>revadj[],vector<int>&cur,vector<int>&vis){
    vis[node]=1;
    cur.push_back(node);
    for(auto it:revadj[node]){
        if(vis[it]==0){
            dfscc(it,revadj,cur,vis);
        }
    }
```

```
}
int main()
{
    int n;
    cout << "Enter the Number of vertices: ";
    cin >> n;
    int e;
    cout << "Enter the Number of Edges: ";
    cin >> e;
    vector<vector<int>> edges(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> edges[i][j];
        }
    }

    vector<int> adj[n];
    vector<int>revadj[n];
    for (int i = 0; i < e; i++)
    {
        adj[edges[i][0]].push_back(edges[i][1]);
        revadj[edges[i][1]].push_back(edges[i][0]);
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
    }
    cout << endl;

    vector<int> pre(n, 0);
    vector<int> post(n, 0);
    vector<int> vis(n, 0);
    int cnt = 0;
    for (int i = 0; i < n; i++)
    {
        if (vis[i] == 0)
```

```
        {
            pre[i] = cnt;
            cnt++;
            dfs(i, adj, pre, post, vis, cnt);
        }
    }
    cout << "Pre and Post of each node: " << endl;
    cout << "Node: "
         << "Pre "
         << "Post" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << i << "      " << pre[i] << "     " << post[i] << endl;
    }
    vector<pair<int, int> >vpost;
    for(int i=0;i<n;i++){
        vpost.push_back({post[i],i});
    }
    sort(vpost.begin(),vpost.end(),greater<pair<int,int>>());

    cout << "The Rev Adjacency List: " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << i << ": ";
        for (auto it : revadj[i])
        {
            cout << it << " ";
        }
        cout << endl;
    }
    cout << endl;
    vector<int>revvis(n,0);
    vector<vector<int>>ans;

    for(auto it:vpost){
        int node=it.second;

        vector<int>scc;
        if(revvis[node]==0){
           dfscc(node,revadj,scc,revvis);
        }

        ans.push_back(scc);
    }
```

```
   cout<<"The SCC Components are: "<<endl;
    for(auto it:ans){
        for(auto node:it){
            cout<<node<<" ";
        }cout<<endl;
    }
    return 0;
}
```

Output:

```
Enter the Number of vertices: 8
Enter the Number of Edges: 9
0 1
1 2
2 3
3 0
2 4
4 5
5 6
6 4
6 7
The Adjacency List:
0: 1
1: 2
2: 3 4
3: 0
4: 5
5: 6
6: 4 7
7:
Pre and Post of each node:
Node: Pre Post
0     0    15
1     1    14
2     2    13
3     3    4
4     5    12
5     6    11
6     7    10
7     8    9
The Rev Adjacency List:
0: 3
1: 0
2: 1
```

```
3: 2
4: 2 6
5: 4
6: 5
7: 6
The SCC Components are:
0 3 2 1
4 6 5
7
```

Complexity analysis:

Time complexity: O(N+E)

Space complexity: It consumes recursion call stack space as O(N).

Program 13

Aim: Obtain the topological ordering of vertices in a given directed graph.

Introduction:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological sorting for a graph is not possible if the graph is not a DAG.
Input parameters:

- `int a[MAX][MAX]` - adjacency matrix of the input graph
- `int n` - number of vertices in the graph

Approach:

1. **Initialize Data Structures**:
    - Calculate the **indegree** (the number of incoming edges) of each node. Use a dictionary or list `indegree` to store these values.
    - Create an empty queue `queue` to store nodes with an indegree of 0 (nodes with no dependencies).
    - Create an empty list `topo_order` to store the topological ordering of nodes.
2. **Enqueue Nodes with Indegree 0**:
    - Traverse all nodes in the graph. For each node with an indegree of 0, add it to the queue. These nodes can be placed first in the topological ordering.
3. **Process Nodes in Queue**:
    - While the queue is not empty:
        - Dequeue a node from the front of the queue.
        - Add this node to `topo_order`.
        - For each neighbor of this node:
            - Decrease the indegree of the neighbor by 1 (as we've removed an incoming edge by processing the current node).
            - If the indegree of the neighbor becomes 0, enqueue it (it has no remaining dependencies and can be added to the topological order).
4. **Check for Cycles**:
    - After processing all nodes, check if the size of `topo_order` is equal to the number of nodes in the graph.

- If they are equal, the graph is a DAG, and **`topo_order`** contains the topological ordering.
- If not, the graph has at least one cycle, making a topological sort impossible.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cout << "Enter the Number of vertices: ";
    cin >> n;
    int e;
    cout << "Enter the Number of Edges: ";
    cin >> e;
    vector<vector<int>> edges(e, vector<int>(2, 0));
    for (int i = 0; i < e; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cin >> edges[i][j];
        }
    }

    vector<int> adj[n+1];
    vector<int>indegre(n+1,0);
    for (auto it:edges)
    {
        adj[it[0]].push_back(it[1]);
        indegre[it[1]]++;
    }
    cout << "The Adjacency List: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (auto it : adj[i])
        {
            cout << it << " ";
        }
        cout << endl;
```

```
    }

    queue<int>q;
    for(int i=1;i<n;i++){
        if(indegre[i]==0)
        {
            q.push(i);
        }
    }
    cout<<"The Topological Ordering: "<<endl;
    while(!q.empty()){
        int node=q.front();
        q.pop();
        cout<<node<<" ";
        for(auto it:adj[node]){
            indegre[it]--;
            if(indegre[it]==0){
                q.push(it);
            }
        }
    }
    return 0;
}
```

Output:

```
Enter the Number of vertices: 7
Enter the Number of Edges: 8
1 2
1 5
2 3
2 4
3 7
4 7
4 6
5 6
The Adjacency List:
1: 2 5
2: 3 4
3: 7
4: 7 6
5: 6
6:
```

```
7:
The Topological Ordering:
1 2 5 3 4 7 6
```

Complexity analysis:

Time complexity: O(N+E)

Space complexity: It takes space as O(N) for storing the indegrees and O(N) for the queue.

Program 14

Aim: For a given set of elements, construct an AVL Tree and also display balance factor for each node.

Introduction:

An AVL (Adelson-Velsky and Landis) Tree is a type of self-balancing binary search tree. In an AVL Tree, the difference between the heights of left and right subtrees (known as the balance factor) for any node is at most |1|. If the balance factor exceeds 1 (i.e., if it is -2 or +2), the tree needs rebalancing using rotations.

Balance Factor for a node:

Balance Factor=Height of Left Subtree−Height of Right Subtree

Approach:

1. Insert Elements into the AVL Tree
    1. Insert each element into the binary search tree while maintaining the BST properties.
    2. After each insertion, compute the balance factor for each node starting from the inserted node up to the root.
    3. If the balance factor of any node is not within the range [−1,1][-1, 1][−1,1], perform rotations to rebalance the tree.
2. Rotations in AVL Tree: To rebalance the tree, use the following rotations based on the node's balance factor:
    1. Right Rotation (Single Rotation): Used when there's a left-heavy subtree.
    2. Left Rotation (Single Rotation): Used when there's a right-heavy subtree.
    3. Left-Right Rotation (Double Rotation): Used when there's a left-right imbalance.
    4. Right-Left Rotation (Double Rotation): Used when there's a right-left imbalance.
3. Display the AVL Tree and Balance Factor for Each Node: After constructing the AVL Tree, you can represent each node with its value and balance factor.

Code:

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Node
{
public:
    int val;
    Node *left;
    Node *right;
    int height;
    Node(int nodeval)
    {
        val = nodeval;
        left = NULL;
        right = NULL;
        height = 1;
    }
};

class AVL
{

public:
    int height(Node *N)
    {
        if (N == NULL)
            return 0;
        return N->height;
    }

    Node *rightRotate(Node *c)
    {
        Node *Y = c->left;
        Node *YR = Y->right;

        // Perform rotation
        Y->right = c;
        c->left = YR;

        // Update height
        c->height = max(height(c->left), height(c->right)) + 1;
        Y->height = max(height(Y->left), height(Y->right)) + 1;

        return Y;
    }
```

```
    Node *leftRotate(Node *c)
    {
        Node *Y = c->right;
        Node *YL = Y->left;

        // Perform rotation
        Y->left = c;
        c->right = YL;

        // Update height
        c->height = max(height(c->left), height(c->right)) + 1;
        Y->height = max(height(Y->left), height(Y->right)) + 1;

        return Y;
    }

    int BalanceFactor(Node *N)
    {
        if (N == NULL)
            return 0;
        return height(N->left) - height(N->right);
    }

    Node *insert(Node *node, int val)
    {

        if (node == NULL)
            return new Node(val);

        if (val < node->val)
            node->left = insert(node->left, val);

        else if (val > node->val)
            node->right = insert(node->right, val);

        else
            return node;

        node->height = 1 + max(height(node->left), height(node-
>right));

        int balance = BalanceFactor(node);
```

```
        // Left Left Case
        if (balance > 1 && val < node->left->val)
            return rightRotate(node);

        // Right Right Case
        if (balance < -1 && val > node->right->val)
            return leftRotate(node);

        // Left Right Case
        if (balance > 1 && val > node->left->val)
        {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && val < node->right->val)
        {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        return node;
    }

    Node *successor(Node *node)
    {
        Node *current = node;

        while (current->left != NULL)
            current = current->left;

        return current;
    }

    Node *deleteNode(Node *root, int val)
    {

        if (root == NULL)
            return root;

        if (val < root->val)
```

```
            root->left = deleteNode(root->left, val);

        else if (val > root->val)
            root->right = deleteNode(root->right, val);

        else
        {
            // only one child or no child
            if ((root->left == NULL) || (root->right == NULL))
            {
                Node *temp = root->left ? root->left : root->right;
                // No child case
                if (temp == NULL)
                {
                    temp = root;
                    root = NULL;
                }
                else // One child case
                    *root = *temp;
                free(temp);
            }
            else
            {
                Node *temp = successor(root->right);

                root->val = temp->val;

                root->right = deleteNode(root->right, temp->val);
            }
        }

        if (root == NULL)
            return root;

        root->height = 1 + max(height(root->left), height(root-
>right));

        int balance = BalanceFactor(root);

        // Left Left Case
        if (balance > 1 && BalanceFactor(root->left) >= 0)
            return rightRotate(root);
```

```
        // Left Right Case
        if (balance > 1 && BalanceFactor(root->left) < 0)
        {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && BalanceFactor(root->right) <= 0)
            return leftRotate(root);

        // Right Left Case
        if (balance < -1 && BalanceFactor(root->right) > 0)
        {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }

    void Postorder(Node *root)
    {
        if (root != NULL)
        {
            Postorder(root->left);
            Postorder(root->right);
            cout << root->val << " ";
        }
    }
};

int main()
{
    AVL obj;
    Node *root = NULL;
    int n;
    cout << "Enter the Number of nodes ";
    cin >> n;
    cout << endl;
    cout << "Insert the nodes: " << endl;
    int nodeval;
    for (int i = 0; i < n; i++)
```

```
    {
        cin >> nodeval;
        root = obj.insert(root, nodeval);
    }

    cout << "Postorder traversal of the constructed AVL tree is" <<
endl;
    obj.Postorder(root);
    cout << endl;

    cout << "Do you Want to delete any Node: 1/0" << endl;

    bool flag;
    cin >> flag;

    if (flag)
    {
        int nodeval;
        cout << "Enter the node value: ";
        cin >> nodeval;

        obj.deleteNode(root, nodeval);
        cout << "Your Postorder Traversal After deleting the node: "
<< nodeval << endl;
        obj.Postorder(root);
    }

    else
        cout << "Thank You" << endl;

    return 0;
}
```

Output:

```
Enter the Number of nodes 9

Insert the nodes:
8
9
2
0
4
3
```

```
7
5
10
Postorder traversal of the constructed AVL tree is
0 3 2 5 7 10 9 8 4
Do you Want to delete any Node: 1/0
1
Enter the node value: 0
Your Postorder Traversal After deleting the node: 0
3 2 5 7 10 9 8 4
```

Complexity analysis:

Time complexity: O(NlogN) for insertion and **O(LogN)** for deletion.
Space complexity: O(1)

Program 15

Aim: Find the median of the two sorted arrays of different sizes using divide and conquer approach.

Introduction:

To find the median of two sorted arrays of different sizes using a divide and conquer approach, the most efficient method involves using binary search on the smaller of the two arrays.

Approach:

The given arrays are sorted, so merge the sorted arrays in an efficient way and keep the count of elements processed so far. So, when we reach half of the total, print the median. There are two cases:

- **Case 1:** m+n is odd, the median is at $(m+n)/2^{th}$ index in the array obtained after merging both the arrays.
- **Case 2:** m+n is even, the median will be the average of elements at index ((m+n)/2 – 1) and (m+n)/2 in the array obtained after merging both the arrays.

Code:

```
#include <bits/stdc++.h>
using namespace std;
double median(vector<int> &a)
{
    int n = a.size();
    if (n % 2)
    {
        return double(a[n / 2]);
    }
    return (1.0 * a[n / 2] + 1.0 * a[n / 2 - 1]) / 2.0;
}
double median_two_sorted_array(vector<int> &a, vector<int> &b)
{
    if (a.size() == 2 and b.size() == 2)
    {
        int num1 = max(a[0], b[0]);
        int num2 = min(a[1], b[1]);
        return (1.0 * num1 + 1.0 * num2) / 2.0;
    }
```

```
    double m1 = median(a);
    double m2 = median(b);
    if (m1 == m2) return m1;
    else if (m1 > m2)
    {
        a.erase(a.begin() + (a.size() / 2) + 1, a.end());
        int st = b.size() % 2 ? b.size() / 2 : b.size() / 2 - 1;
        b.erase(b.begin(), b.begin() + st);
    }
    else
    {
        int st = a.size() % 2 ? a.size() / 2 : a.size() / 2 - 1;
        a.erase(a.begin(), a.begin() + st);
        b.erase(b.begin() + (b.size() / 2) + 1, b.end());
    }
    return median_two_sorted_array(a, b);
}
int main()
{
    vector<int> a = {1, 12, 20, 40, 50};
    vector<int> b = {5, 16, 19, 25, 60};
   cout << "The median of two sorted array is: " <<
median_two_sorted_array(a, b);
}
```

Output:

```
The median of two sorted array is: 19.5
```

Complexity analysis:

Time complexity: O(LogN)

Space complexity: It consumes some recursion call stack space.

Program 16

Aim: Count the number of inversions in an array with the help of merge sort.

Introduction:

Two array elements **`arr[i]`** and **`arr[j]`** form an inversion if **`arr[i] > arr[j]`** and **`i < j`**. Inversion Count for an array indicates that how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is maximum.

Approach:

We can use merge sort to count the inversions in an array.

- First, we divide the array into two halves: left half and right half.
- Next, we recursively count the inversions in both halves.
- While merging the two halves back together, we also count how many elements from the left half array are greater than elements from the right half array, as these represent cross inversions (i.e., element from the left half of the array is greater than an element from the right half during the merging process in the merge sort algorithm).
- Finally, we sum the inversions from the left half, right half, and the cross inversions to get the total number of inversions in the array.

This approach efficiently counts inversions while sorting the array.

Code:

```
#include <bits/stdc++.h>
using namespace std;

pair<vector<int>, int> merge(vector<int> &v1)
{
    if (v1.size() == 1)
    {
        return {v1, 0};
    }
    int size = v1.size();
    vector<int> cur;
    vector<int> a(v1.begin(), v1.begin() + size / 2);
    vector<int> b(v1.begin() + size / 2, v1.end());
    pair<vector<int>, int> p1 = merge(a);
    pair<vector<int>, int> p2 = merge(b);
    int n = a.size(), m = b.size(), inv = p1.second + p2.second;
```

```
    int i = 0, j = 0;
    while (i < n && j < m)
    {
        if (a[i] <= b[j])
        {
            cur.push_back(a[i]);
            i++;
        }
        else
        {
            cur.push_back(b[j]);
            j++;
            inv += (n - i);
        }
    }
    while (i < n)
    {
        cur.push_back(a[i++]);
    }
    while (j < n)
    {
        cur.push_back(b[j++]);
    }

    return {cur, inv};
}
int main()
{
    vector<int> a = {5, 8, 3, 2, 7, 9};
    pair<vector<int>, int> ans = merge(a);
    cout << ans.second << endl;
    return 0;
}
```

Output:

```
6
```

Complexity analysis:

Time complexity: O(NLogN)

Space complexity: It consumes recursion call stack space O(N) and O(N) to create a copy of the input array.

Program 17

Aim: Find out the closest pair of points in the array using divide and conquer approach.

Introduction:

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. Here we looking for the smallest distance between two points. This distance is called the Euclidean distance, which we calculate using formula:

$$||xy|| = \sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision.

We can calculate the smallest distance in O(nLogn) time using divide-and-conquer strategy.

Approach:

Input: An array of `n` points `P[]`

Output: The smallest distance between two points in the given array.

As a pre-processing step, the input array is sorted according to `x` coordinates.

1. Find the middle point in the sorted array, we can take `P[n/2]` as middle point. Divide the given array in two halves: the first subarray from `P[0]` to `P[n/2]`, the second subarray from `P[n/2+1]` to `P[n-1]`.
   Recursively, find the smallest distances in both subarrays. Let the distances be `dl` and `dr`. Find the minimum of `dl` and `dr`. Let `d=min(dl,dr)`.
2. Now, we have an upper bound `d` of minimum distance. So, we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Build an array `strip[]` of the points whose `x` coordinate less than `d`.
3. Sort the array `strip[]` according to `y` coordinates. This takes O(nLogn). It can be optimized to O(n) by recursively sorting and merging.
   6) Find the smallest distance in `strip[]`. This is tricky. From the first look, it seems to be a $O(n^2)$ step, but it is actually O(n).
4. Finally return the minimum of `d` and distance calculated in step 6.

Code:

```
#include <bits/stdc++.h>
using namespace std;
```

```
bool comparex(pair<int, int> a, pair<int, int> b)
{
    return a.first < b.first;
}
bool comparey(pair<int, int> a, pair<int, int> b)
{
    return a.second < b.second;
}
float dist(pair<int, int> p1, pair<int, int> p2)
{
    int x = p1.first - p2.first;
    int y = p1.second - p2.second;
    return sqrt(x * x + y * y);
}
float stripClosest(vector<pair<int, int>> strip, int size, float d)
{
    float mini = d;
    sort(strip.begin(), strip.end(), comparey);
    for (int i = 0; i < size; i++)
    {
        for (int j = i + 1; j < size && (strip[j].second -
strip[i].second < mini); j++)
        {
            if (dist(strip[i], strip[j]) < mini)mini = dist(strip[i],
strip[j]);
        }
    }
    return mini;
}
float bruteforce(vector<pair<int, int>> &v)
{
    float mini = INT_MAX;
    for (int i = 0; i < v.size(); i++)
    {
        for (int j = i + 1; j < v.size(); j++)
        {
            float dis = dist(v[i], v[j]);
            if (dis < mini)mini = dis;
        }
    }
    return mini;
}
float closest(vector<pair<int, int>> &v)
```

```
{
    if (v.size() <= 3) return bruteforce(v);
    int mid = v.size() / 2;
    pair<int, int> midpoint = v[mid];
    vector<pair<int, int>> left(v.begin(), v.begin() + mid);
    vector<pair<int, int>> right(v.begin() + mid, v.end());
    float dl = closest(left);
    float dr = closest(right);
    float d = min(dl, dr);
    vector<pair<int, int>> strip(v.size());
    int j = 0;
    for (int i = 0; i < v.size(); i++)
    {
        if (abs(v[i].first - midpoint.first) < d)
        {
            strip[j] = v[i];
            j++;
        }
    }
    return min(d, stripClosest(strip, j, d));
}
int main()
{
    vector<pair<int, int>> points{{1, 2}, {3, 2}, {4, 2}, {5, 2}, {6,
2}};
    int n = points.size();
    sort(points.begin(), points.end(), comparex);
    cout << "Smallest Dist: " << closest(points) << endl;
}
```

Output:

```
Smallest Dist: 1
```

Complexity analysis:

Time complexity: $O(N(\log N)^2)$

Space complexity: It consumes recursion call stack space as $O(\log N)$.

Program 18

Aim: Given a value V, if we want to make change for V Rs., and we have infinite supply of each of the denominations in Indian currency, i.e., we have infinite supply of {1, 2, 5, 10, 20, 50, 100, 200, 2000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change? Implement the greedy algorithm to find minimum number of coins.

Introduction:

The intuition would be to take coins with greater value first. This can reduce the total number of coins needed. Start from the largest possible denomination and keep adding denominations while the remaining value is greater than 0.

Approach:

- Sort the array of coins in decreasing order.
- Initialize **`ans`** vector as empty.
- Find the largest denomination that is smaller than remaining amount and while it is smaller than the remaining amount:
- Add found denomination to **`ans`**. Subtract value of found denomination from amount.
- If amount becomes 0, then print **`ans`**.

Code:

```
#include <bits/stdc++.h>
using namespace std;
void minimum_coin(vector<int> &denomination, int targetvalue)
{
    int ans = 0, n = denomination.size();
    for (int i = n - 1; i >= 0; i--)
    {
        if (targetvalue >= denomination[i])
        {
            ans = targetvalue / denomination[i];
            targetvalue %= denomination[i];
            cout << denomination[i] << " "<<ans<<endl;
        }
        if (targetvalue == 0)break;
    }
}
int main()
```

```
{
    vector<int> denomination{1, 2, 5, 10, 20, 50, 100, 200, 2000};
    int targetvalue;
    cout << "Enter the Target Value: ";
    cin >> targetvalue;
    cout << "Coins and their numbers" << endl;
    minimum_coin(denomination, targetvalue);
    return 0;
}
```

Output:

```
Enter the Target Value:255
Coins and their numbers
200 1
50 1
5 1
```

Complexity analysis:

Time complexity: O(NLogN)
Space complexity:O(1)

Program 19

Aim: Find the minimum cost spanning tree of a given undirected graph using:

a) **Kruskal's algorithm**
b) **Prim's algorithm**

Introduction:

A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree.

In **Kruskal's algorithm**, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus, we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence, this is a greedy algorithm.

The **Prim's algorithm** starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Approach:

Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.

The working of Prim's algorithm can be described by using the following steps:

1. Determine an arbitrary vertex as the starting vertex of the MST.
2. Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3. Find edges connecting any tree vertex with the fringe vertices.
4. Find the minimum among these edges.
5. Add the chosen edge to the MST if it does not form any cycle.
6. Return the MST and exit.

Code:

(a) Kruskal's algorithm

```
#include <bits/stdc++.h>
using namespace std;
struct edges
{
    int dist, u, v;
};
class Compare
{
public:
    bool operator()(edges a, edges b)
    {
        return a.dist > b.dist;
    }
};
int main()
{
    int n, e;
    cout << "Enter the Nodes and edges : ";
    cin >> n >> e;
    cout << "Enter the edges and weights : " << endl;
    int u, v, wt;
    vector<pair<int, int>> adj[n + 1];
    priority_queue<edges, vector<edges>, Compare> pq;
    for (int i = 0; i < e; i++)
    {
        cin >> u >> v >> wt;
        pq.push({wt, u, v});
        adj[u].push_back({v, wt});
        adj[v].push_back({u, wt});
    }

    vector<int> components(n + 1);
    for (int i = 1; i <= n; i++)
    {
        components[i] = i;
    }
    int minimumCost = 0;
    int cnt = 0;
    while (cnt < n - 1)
    {
        auto it = pq.top();
        int u1 = it.u;
        int v1 = it.v;
        pq.pop();
        if (components[u1] != components[v1])
        {
            cnt++;
```

```
            minimumCost += it.dist;
            for (int i = 1; i <= n; i++)
            {
                if (components[i] == components[v1])
                {
                    components[i] = components[u1];
                }
            }
        }
    }
    cout << endl;

    cout << "The minimum cost: " << minimumCost << endl;
}
```

(b) Prim's algorithm

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n, e;
    cout << "Enter the number of node ";cin >> n;
    cout << "Enter the number of edges ";cin >> e;
    int u, v, wt;
    vector<pair<int, int>> adj[n + 1];
    for (int i = 0; i < e; i++)
    {
        cin >> u >> v >> wt;
        adj[u].push_back({v, wt});
        adj[v].push_back({u, wt});
    }
    vector<int> parent(n + 1, -1);
    vector<int> dist(n + 1, INT_MAX);
    vector<int> MSTvis(n + 1, 0);
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    dist[1] = 0;
    pq.push({0, 1});
    int minimumcost = 0;
    for (int count = 1; count <= n - 1; count++)
    {
        int node = pq.top().second; // minimum weight element
        pq.pop();
```

```
        MSTvis[node] = true;
        for (auto it : adj[node])
        {
            int nextnode = it.first;
            int weight = it.second;
            if (MSTvis[nextnode] == false && weight < dist[nextnode])
            {
                parent[nextnode] = node;
                dist[nextnode] = weight;
                pq.push({dist[nextnode], nextnode});
            }
        }
    }
    for (int i = 1; i <= n; i++)
    {
        minimumcost += dist[i];
    }
    cout << "The minimum cost for MST: " << minimumcost << endl;
    return 0;
}
```

Output:

(a) Kruskal's algorithm

```
Enter the number of node 6
Enter the number of edges 9
1 2 7
1 3 8
2 3 3
2 5 6
3 5 4
3 4 3
4 6 2
4 5 2
5 6 5
The minimum cost for MST: 17
```

(b) Prim's algorithm

```
Enter the number of node 6
Enter the number of edges 9
1 2 7
1 3 8
```

```
2 3 3
2 5 6
3 5 4
3 4 3
4 6 2
4 5 2
5 6 5
The minimum cost for MST: 17
```

Complexity analysis:

(a) Kruskal's algorithm

Time complexity: $O(N^2)$

Space complexity: O(V + E) for the `parent` and `visited` vectors and for the queue.

(b) Prim's algorithm

Time complexity: O(E Log V + V Log V)

Space complexity: O(V + E) for the `parent` and `visited` vectors and for the queue.

Program 20

Aim: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

Introduction:

The idea is to generate a shortest path tree with a given source as a root. Maintain an adjacency matrix with two sets,

- one set contains vertices included in the shortest-path tree,
- the other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Approach:

1. **Initialization**:
    - Create a set `visited` to track vertices whose shortest path from `source` has been found.
    - Create an array `dist[]` to store the shortest distance from `source` to every vertex. Initialize all distances as infinity (∞), except for the source vertex `source` which is set to 0.
    - Create an array `prev[]` to store the previous vertex on the shortest path from `source` for reconstructing paths later.
    - Use a **min-priority queue** (implemented using a min-heap) to efficiently extract the vertex with the smallest distance.
2. **Iterate**:
    - While the priority queue is not empty:
        - Extract the vertex `u` with the smallest distance value from the queue.
        - Mark `u` as visited.
        - For each unvisited neighbor `v` of `u`:
            - If the path from `source` to `v` through `u` is shorter than the current known distance to `v` (i.e., `dist[u]+weight(u,v)<dist[v]`), update `dist[v]` and set `prev[v]` to `u`.
            - Add or update `v` in the priority queue with its new distance.
3. **Termination**:
    - Once all vertices have been processed, the `dist[]` array contains the shortest distances from the source to all other vertices.

- The **`prev[]`** array can be used to reconstruct the shortest path to any vertex.

Code:

```
#include <bits/stdc++.h>
using namespace std;
void minimum_dis(int src, vector<int> &dist, vector<int> &parent,
vector<pair<int, int>> adj[])
{
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    pq.push({0, src});
    dist[src] = 0;
    while (!pq.empty())
    {
        int dis = pq.top().first;
        int node = pq.top().second;
        pq.pop();
        for (auto it : adj[node])
        {
            int curwt = it.second;
            int curnode = it.first;
            if (dist[curnode] > dis + curwt)
            {
                dist[curnode] = dis + curwt;
                pq.push({dist[curnode], curnode});
                parent[curnode] = node;
            }
        }
    }
}
int main()
{
    int n,e;
    cout << "Enter the number of nodes: "; cin >> n;
    cout << "Enter the number of edges: ";cin >> e;
    vector<pair<int, int>> adj[n + 1];
    cout << "Enter the edges " << endl;
    int u, v, wt;
    for (int i = 0; i < e; i++)
    {
        cin >> u >> v >> wt;
```

```
        adj[u].push_back({v, wt});
        adj[v].push_back({u, wt});
    }
    vector<int> parent(n + 1, -1);
    vector<int> dist(n + 1, INT_MAX);
    minimum_dis(1, dist, parent, adj);
    for (int i = 1; i <= n; i++)
    {
        cout << "Path from " << 1 << " to node " << i << " is:";
        int node = i;
        while (parent[node] > -1)
        {
            cout << parent[node] << "->";
            node = parent[node];
        }
        cout << " Distance " << dist[i] << endl;
    }
}
```

Output:

```
Enter the number of nodes: 7
Enter the number of edges: 8
Enter the edges
0 1 2
0 2 6
1 3 5
2 3 8
3 5 15
3 4 10
4 6 2
5 6 6
Path from 0 to node 0 is: Distance 0
Path from 0 to node 1 is:0-> Distance 2
Path from 0 to node 2 is:0-> Distance 6
Path from 0 to node 3 is:1->0-> Distance 7
Path from 0 to node 4 is:3->1->0-> Distance 17
Path from 0 to node 5 is:3->1->0-> Distance 22
Path from 0 to node 6 is:4->3->1->0-> Distance 19
```

Complexity analysis:

Time complexity: O(E Log V)

Space complexity: O(V) for the **`parent`** and **`visited`** vectors and for the queue.

Program 21

Aim: Implement the longest ascending subsequence problem using dynamic programming.

Introduction:

Given an array A of n integers, find the length of the longest subsequence that is strictly increasing. A subsequence is a sequence that can be derived by deleting some or no elements from the original array without changing the order of the remaining elements.

Approach:

The key idea is to use a dynamic programming (DP) array where `dp[i]` represents the length of the longest increasing subsequence that ends at index `i`.

Steps:

1. Initialize the DP array:
    - Let `dp[i]` be the length of the longest ascending subsequence ending at index `i`.
    - Initialize `dp` to all 1's because the shortest increasing subsequence ending at any element is the element itself.
    - Thus, `dp[i]=1` for all `i`.
2. Fill the DP array:
    - For each element `A[i]`, check all previous elements `A[j]` where `j<i`.
    - If `A[j]<A[i]`, then `A[i]` can extend the increasing subsequence ending at `j`.
    - Update `dp[i]` as: `dp[i]=max(dp[i],dp[j]+1)`
3. Result:
    - The length of the longest increasing subsequence is the maximum value in the `dp` array.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    cout << "Enter the elements: ";
```

```
    vector<int> arr(n);
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));

    for (int i = n - 1; i >= 0; i--)
    {
        for (int prev = i - 1; prev >= -1; prev--)
        {
            int skip = dp[i + 1][prev + 1];
            int take = 0;
            if (prev == -1 || arr[i] > arr[prev])
            {
                take = 1 + dp[i + 1][i + 1];
            }
            dp[i][prev + 1] = max(take, skip);
        }
    }
    cout << "The the longest increasing Sub sequence length:  " <<
dp[0][0] << endl;
    return 0;
}
```

Output:

```
Enter the number of elements: 10
Enter the elements: 1 3 2 5 4 6 8 7 9 10
The the longest increasing Sub sequence length:  7
```

Complexity analysis:

Time complexity: $O(N^2)$

Space complexity: $O(N^2)$

Program 22

Aim: Implement matrix chain multiplication problem using dynamic programming.

Introduction: Given a sequence of matrices A_1, A_2, …, A_n, where matrix A_i has dimensions $p_{i-1} \times p_i$, find the most efficient way to multiply these matrices together. The objective is to minimize the number of scalar multiplications required. The matrices can only be multiplied in the order given, but we can change the way they are parenthesized.

Example:

Suppose you have matrices with dimensions:
A_1: 10×30, A_2: 30×5, A_3: 5×60
You need to decide where to place the parentheses to minimize the multiplication cost.

Approach:

We'll use a dynamic programming approach to solve this problem optimally by using a table to store the minimum multiplication costs for multiplying subsets of matrices.

Key Idea

- Define a 2D array `dp` where `dp[i][j]` represents the minimum cost of multiplying matrices from A_i to A_j.
- The goal is to calculate `dp[1][n]`, which represents the minimum cost of multiplying the entire sequence of matrices.
- The recursive formula to find the minimum cost is:

$$dp[i][j] = \min_{i \le k < j} (dp[i][k] + dp[k+1][j] + p_{i-1} \times p_k \times p_j$$

  - Here, $p_{i-1} \times p_k \times p_j$ is the cost of multiplying the two resulting matrices after dividing at k.

Dynamic Programming Table Construction

1. Initialization:
   - If there's only one matrix, the cost is zero, so `dp[i][i] = 0` for all i.
2. Filling the table:
   - Iterate over increasing chain lengths.
   - For each length, compute the cost for every possible starting index.
   - For each possible split point k, calculate the cost and update the table.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int mcm(int i, int j, vector<int> &v,vector<vector<int>>&dp)
{
    if (i == j)
    {
        return 0;
    }
    if(dp[i][j]!=-1)return dp[i][j];
    int ans = INT_MAX;
    for (int k = i; k < j; k++)
    {
        int temp = v[i - 1] * v[k] * v[j] + mcm(i, k, v,dp) + mcm(k
+ 1, j, v,dp);
        ans = min(ans, temp);
    }
    return dp[i][j]=ans;
}
int main()
{

    int n;
    cout << "Enter the number of Elements: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the numbers: ";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    vector<vector<int> >dp(n,vector<int>(n,-1));
    cout << "The minimum cost of the Matrix Chain Multiplication: "
<< mcm(1, n - 1, arr,dp) << endl;
}
```

Output:

```
Enter the number of Elements: 6
Enter the numbers: 1 4 12 30 20 5
The minimum cost of the Matrix Chain Multiplication: 1108
```

Complexity analysis:

Time complexity: $O(N^3)$
Space complexity: $O(N^2)$

Program 23

Aim: Implement 0/1 Knapsack problem using dynamic programming.

Introduction:

Given:

- A list of items, each with a **weight** and a **value**.
- A knapsack with a maximum **capacity** *W*.

Objective:

- Select items to include in the knapsack such that:
  - The **total weight** of the selected items does not exceed *W*.
  - The **total value** is maximized.

Note: Each item can either be taken completely (1) or not taken at all (0), hence the "0/1" in the problem name.

Example:

Suppose we have the following items:

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	8

And a knapsack with a capacity of *W*=5. The goal is to select items such that the total weight does not exceed 5 and the value is maximized.

Approach:

We will solve this problem using dynamic programming by creating a table to store the maximum value we can achieve with a given capacity.

Key Idea

- Define a 2D array `dp` where:
  - `dp[i][w]` represents the maximum value that can be obtained with the first i items and a knapsack capacity of w.
- **Base Case**:
  - If there are no items or the capacity is zero, the maximum value is zero:

$$dp[0][w] = 0 \text{ for all } w$$
$$dp[i][0] = 0 \text{ for all } i$$

- **Recurrence Relation**:
  - For each item iii, we have two choices:
    1. **Exclude the item**: The value remains the same as without this item: `dp[i-1][w]`.
    2. **Include the item**: Add its value to the best value achievable with the remaining capacity: `dp[i-1][w - weight[i]] + value[i]`.
  - Thus, the recurrence relation is:

    $$dp[i][w] = \max\left(dp[i-1][w], dp[i-1][w - weight[i]] + value[i]\right)$$
  - If the item's weight is greater than the current capacity w, we cannot include it:

    $$dp[i][w] = dp[i-1][w]$$

Code:

```
#include <bits/stdc++.h>
using namespace std;
int maximum_profit(int i, int W, vector<pair<int, int>> &wp,
vector<vector<int>> &dp)
{
    if (i == 0)
    {
        if (W >= wp[0].first)
        {
            W -= wp[0].first;
            return wp[0].second;
        }
        else
            return 0;
    }
    if (dp[i][W] != -1)
        return dp[i][W];
    int nottake = 0 + maximum_profit(i - 1, W, wp, dp);
    int take = 0;
    if (W >= wp[i].first)
    {
        W -= wp[i].first;
        take = wp[i].second + maximum_profit(i - 1, W, wp, dp);
    }
    return dp[i][W] = max(take, nottake);
}
int main()
```

```
{
    int n;
    cout << "Enter the size: ";
    cin >> n;
    cout << "Enter the pairs of weight and profit: " << endl;
    vector<pair<int, int>> wp;
    int w, p;
    for (int i = 0; i < n; i++)
    {
        cin >> w >> p;
        wp.push_back({w, p});
    }
    int target;
    cout << "Enter the target ";
    cin >> target;
    vector<vector<int>> dp(n + 1, vector<int>(target + 1, -1));
    cout << "Maximum Profit: " << maximum_profit(n - 1, target, wp,
dp) << endl;

    return 0;
}
```

Output:

```
Enter the pairs of weight and profit:
1 1
2 2
5 5
7 7
Enter the target 7
Maximum Profit: 9
```

Complexity analysis:

Time complexity: O(N*capacity)
Space complexity: O(N*capacity)

Program 24

Aim: Write a program to find the shortest path using Bellman-Ford algorithm.

Introduction:

The Bellman-Ford algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph. It is particularly useful when the graph contains edges with negative weights. Unlike Dijkstra's algorithm, Bellman-Ford can handle negative weight edges, and it can also detect negative weight cycles.

Approach:

Given a weighted, directed graph with V vertices and E edges, and a source vertex S, find the shortest path from S to every other vertex. If a negative weight cycle exists, the algorithm will indicate that. The following are the steps to implement the Bellman-Ford algorithm:

1. Initialize distances:
    - Set the distance to the source vertex S as 0.
    - Set the distance to all other vertices as infinity.
2. Relax edges:
    - Repeat $V-1$ times (where V is the number of vertices):
        - For each edge (u, v) with weight w, update the distance to vertex v as:
          if distance[u]+w<distance[v], then update distance[v]=distance[u]+w
3. Check for negative weight cycles:
    - Perform one more iteration over all edges. If any distance can still be updated, it means there is a negative weight cycle.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n, e;
    cout << "Enter the number of Nodes: ";
    cin >> n;
    cout << "Enter the number of Edges: ";
    cin >> e;
```

```
    vector<vector<int>> edges;
    int u, v, wt;
    cout << "Enter the edges: " << endl;
    for (int i = 0; i < e; i++)
    {
        cin >> u >> v >> wt;
        edges.push_back({u, v, wt});
    }
    vector<int> dis(n, INT_MAX);
       dis[0] = 0;
    for (int i = 1; i <= n - 1; i++)
    {
        for (int j = 0; j < e; j++)
        {
            int u1 = edges[j][0];
            int v1 = edges[j][1];
            int wt1 = edges[j][2];
            if (dis[u1] != INT_MAX && dis[u1] + wt1 < dis[v1])
            {
                dis[v1] = dis[u1] + wt1;
            }
        }
    }
    // for checking negative cycle
    for (int i = 0; i < e; i++)
    {
        int u1 = edges[i][0];
        int v1 = edges[i][1];
        int wt1 = edges[i][2];
        if (dis[u1] != INT_MAX && dis[v1] > dis[u1] + wt1)
        {
            cout << "Graph contains negative Cycle" << endl;
            break;
        }
    }
    cout << "The distance from 0: " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << "Node : " << i << " distance " << dis[i] << endl;
    }
    cout << endl;
    return 0;
}
```

Output:

```
Enter the number of Nodes: 5
Enter the number of Edges: 7
Enter the edges:
0 3 3
0 4 12
2 3 -2
1 2 2
3 1 5
3 2 3
4 2 7
The distance from 0:
Node : 0 distance 0
Node : 1 distance 8
Node : 2 distance 6
Node : 3 distance 3
Node : 4 distance 12
```

Complexity analysis:

Time complexity: O(V×E) where V is the number of vertices and E is the number of edges. The algorithm iterates $V-1$ times over all the edges, making it slower than Dijkstra's algorithm for dense graphs.

Space complexity: O(V) for storing the distances.

Program 25

Aim: Implement branch and bound scheme to find the optimal solution for:

(a) Job assignment problem

(b) Traveling salesperson problem

Introduction:

The **Job Assignment Problem** is a classic combinatorial optimization problem where the goal is to assign n jobs to n persons such that the total cost is minimized, given that each person can perform only one job and each job can be assigned to only one person.

The **Traveling Salesperson Problem (TSP)** is a classic combinatorial optimization problem. The goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.

Approach:

(a) Job assignment problem

Problem Statement: Given a cost matrix of size $n \times n$, where $cost[i][j]$ represents the cost of assigning job j to person i, we need to find an assignment that minimizes the total cost.

Key Concepts

1. State Space Tree:
   - Each node in the tree represents a partial assignment of jobs to persons.
   - The root node represents no jobs assigned.
   - Each level represents assigning a job to one person.
2. Cost Function:
   - We define the cost of a partial solution using the current assignment and a lower bound for the remaining unassigned jobs.
   - This helps in pruning branches that cannot lead to an optimal solution.
3. Bounding Function:
   - The lower bound for a node is calculated using the sum of the minimum costs of the remaining jobs for each unassigned person.
   - If the lower bound of a node is greater than the current best solution, that branch is pruned.

(b) Traveling salesperson problem

Problem Statement

Given a set of n cities and a distance matrix where $cost[i][j]$ represents the cost of traveling from city i to city j, the objective is to find a tour that visits each city exactly once, minimizes the total travel cost, and returns to the starting city. For TSP,

Branch-and-Bound helps reduce the number of possible routes to explore by using bounds to prune non-optimal branches.

Key Concepts

1. **State Space Tree**:
   - Each node in the tree represents a partial solution.
   - The root node represents starting from a given city.
   - Each level represents visiting another city.
2. **Cost Function**:
   - We define the cost of a partial solution as the current cost of traveling along the route taken so far.
3. **Bounding Function**:
   - The **lower bound** for a node is calculated as the current cost of the partial route plus an estimate (using the minimum cost edges) for completing the rest of the tour.
   - If the lower bound of a node is greater than the current best solution, that branch is pruned.

Code:

(a) Job assignment problem

```
#include <bits/stdc++.h>
using namespace std;

const int N = 4; // Number of jobs (adjust as needed)
int cost[N][N] = {
    {9, 2, 7, 8},
    {6, 4, 3, 7},
    {5, 8, 1, 8},
    {7, 6, 9, 4}
};

// Structure to store information of a node in the search tree
struct Node {
    vector<bool> assigned;  // Keeps track of which jobs are
assigned
    int person;             // Current level in the tree (person
index)
    int cost;               // Cost of reaching this node
    int lower_bound;        // Lower bound of the path cost
};

// Function to calculate the lower bound of a node
int calculateLowerBound(Node &node) {
    int bound = node.cost;

    // Calculate the sum of the minimum cost for each unassigned job
```

```
    for (int i = node.person + 1; i < N; i++) {
        int min_cost = INT_MAX;
        for (int j = 0; j < N; j++) {
            if (!node.assigned[j]) {
                min_cost = min(min_cost, cost[i][j]);
            }
        }
        bound += min_cost;
    }
    return bound;
}

// Comparison function for priority queue (min-heap)
struct Compare {
    bool operator()(const Node &a, const Node &b) {
        return a.lower_bound > b.lower_bound;
    }
};

// Branch and Bound algorithm to find the minimum assignment cost
int branchAndBound() {
    // Initialize priority queue (min-heap)
    priority_queue<Node, vector<Node>, Compare> pq;

    // Start from the root node (level -1, no jobs assigned)
    Node root;
    root.person = -1;
    root.cost = 0;
    root.assigned = vector<bool>(N, false);
    root.lower_bound = calculateLowerBound(root);

    // Add root to the priority queue
    pq.push(root);

    int min_cost = INT_MAX;

    // Process nodes until the priority queue is empty
    while (!pq.empty()) {
        Node current = pq.top();
        pq.pop();

        // If we have reached the last person, update the minimum
cost
        if (current.person == N - 1) {
            min_cost = min(min_cost, current.cost);
            continue;
        }
```

```
        // Generate children for the current node
        int next_person = current.person + 1;
        for (int j = 0; j < N; j++) {
            // If job j is not yet assigned, assign it to
next_person
            if (!current.assigned[j]) {
                Node child = current;
                child.person = next_person;
                child.cost += cost[next_person][j];
                child.assigned[j] = true;

                // Calculate the lower bound for this child node
                child.lower_bound = calculateLowerBound(child);

                // If the lower bound is better than the current
minimum cost, push it to the queue
                if (child.lower_bound < min_cost) {
                    pq.push(child);
                }
            }
        }
    }

    return min_cost;
}

int main() {
    cout << "Minimum assignment cost: " << branchAndBound() << endl;
    return 0;
}
```

(b) Traveling salesperson problem

```
#include <bits/stdc++.h>
using namespace std;

const int N = 4; // Number of cities (adjust as needed)
const int INF = INT_MAX;

// Cost matrix representing the distance between cities
int cost[N][N] = {
    {INF, 10, 15, 20},
    {10, INF, 35, 25},
    {15, 35, INF, 30},
    {20, 25, 30, INF}
};

// Structure to represent a node in the state space tree
struct Node {
```

```
    vector<int> path;       // Stores the current path
    int reducedCostMatrix[N][N]; // Reduced cost matrix for bounding
    int cost;               // Cost of reaching this node
    int level;              // Level of the node (number of cities
visited)
    int currentCity;        // The current city in the path
};

// Function to reduce a matrix and calculate its cost
int reduceMatrix(int matrix[N][N], int reducedMatrix[N][N]) {
    int cost = 0;

    // Row reduction
    for (int i = 0; i < N; i++) {
        int rowMin = INF;
        for (int j = 0; j < N; j++) {
            rowMin = min(rowMin, matrix[i][j]);
        }
        if (rowMin != INF && rowMin != 0) {
            cost += rowMin;
            for (int j = 0; j < N; j++) {
                matrix[i][j] -= rowMin;
            }
        }
    }

    // Column reduction
    for (int j = 0; j < N; j++) {
        int colMin = INF;
        for (int i = 0; i < N; i++) {
            colMin = min(colMin, matrix[i][j]);
        }
        if (colMin != INF && colMin != 0) {
            cost += colMin;
            for (int i = 0; i < N; i++) {
                matrix[i][j] -= colMin;
            }
        }
    }

    // Copy the reduced matrix
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            reducedMatrix[i][j] = matrix[i][j];
        }
    }

    return cost;
```

```
}

// Comparison function for the priority queue (min-heap)
struct Compare {
    bool operator()(const Node &a, const Node &b) {
        return a.cost > b.cost;
    }
};

// Branch and Bound algorithm for TSP
int branchAndBoundTSP() {
    priority_queue<Node, vector<Node>, Compare> pq;
    Node root;
    root.path.push_back(0);  // Starting from city 0
    root.cost = reduceMatrix(cost, root.reducedCostMatrix);
    root.level = 0;
    root.currentCity = 0;

    pq.push(root);
    int minCost = INF;

    // Process nodes in the priority queue
    while (!pq.empty()) {
        Node current = pq.top();
        pq.pop();

        // If we have visited all cities, complete the tour
        if (current.level == N - 1) {
            current.cost += cost[current.currentCity][0];
            minCost = min(minCost, current.cost);
            continue;
        }

        // Generate children nodes
        for (int i = 0; i < N; i++) {
            if (find(current.path.begin(), current.path.end(), i) ==
current.path.end()) {
                Node child;
                child.path = current.path;
                child.path.push_back(i);
                child.level = current.level + 1;
                child.currentCity = i;

                // Copy the parent's reduced cost matrix
                int childMatrix[N][N];
                memcpy(childMatrix, current.reducedCostMatrix,
sizeof(childMatrix));
```

```
                // Update the cost and reduce the matrix
                for (int j = 0; j < N; j++) {
                    childMatrix[current.currentCity][j] = INF;
                    childMatrix[j][i] = INF;
                }
                childMatrix[i][0] = INF;

                child.cost = current.cost +
cost[current.currentCity][i] + reduceMatrix(childMatrix,
child.reducedCostMatrix);

                if (child.cost < minCost) {
                    pq.push(child);
                }
            }
        }
    }

    return minCost;
}

int main() {
    int minCost = branchAndBoundTSP();
    cout << "Minimum cost to visit all cities: " << minCost << endl;
    return 0;
}
```

Output:

(a) Job assignment problem

```
Minimum assignment cost: 13
```

(b) Traveling salesperson problem

```
Minimum cost to visit all cities: 81
```

Complexity analysis:

(a) Job assignment problem

Time Complexity: The worst-case time complexity is O($N!$), but the Branch-and-Bound method significantly reduces the number of nodes explored by pruning suboptimal branches.

Space Complexity: O(N^2) due to the storage of nodes in the priority queue.

(b)Traveling salesperson problem

Time Complexity: In the worst case, it is O($N!$), but the Branch-and-Bound technique reduces the number of nodes explored by pruning suboptimal branches.

Space Complexity: O(N^2) due to the storage of matrices in nodes.