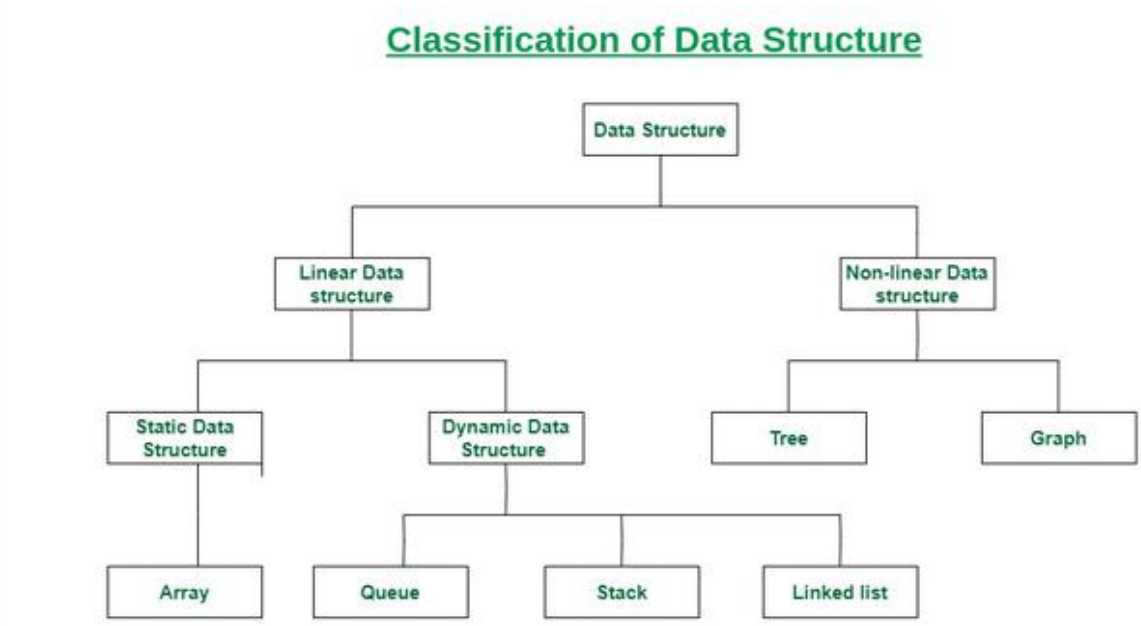
Data Structures and ALGORITHMS

Data Structures:

"**A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.**"

Like Arrays, LinkedList, Stack, Queue, Tree, Graph, etc.



Linear DS:

Data Structures in which data elements are arranged sequentially or linearly. Like values are arranged in a linear fashion.

Static DS:

Static Data structures have fixed memory sizes like arrays, etc.

Dynamic DS:

Dynamic Data Structures the size of memory is not fixed like list, etc.

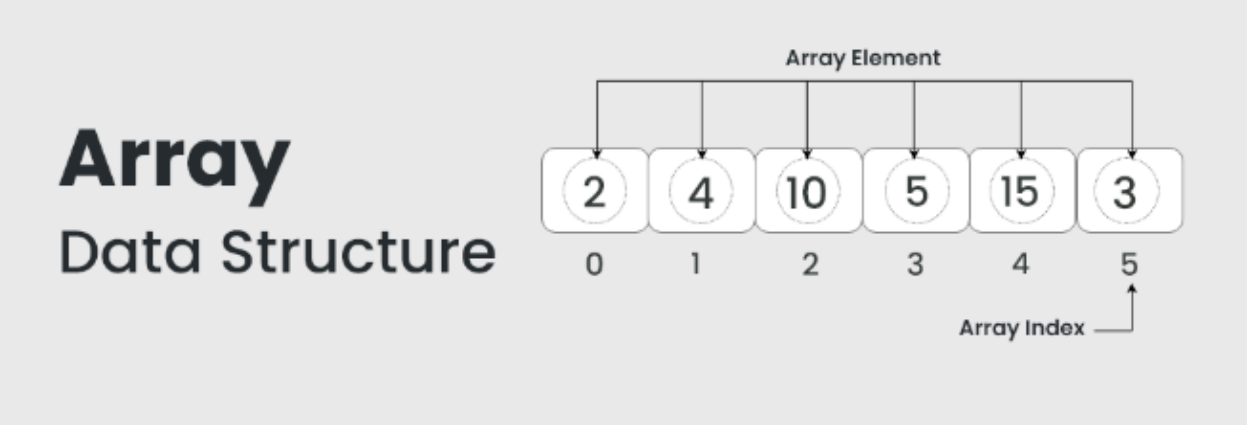
Nonlinear DS:

Data Structures in which data elements are not arranged sequentially or linearly. Like values are not arranged in a linear fashion. Like hash table, tree, etc.

"The common operations applied on various Data structures include traversing, searching, deletion, insertion, updating, etc."

ARRAYS

An array is a fundamental data structure and is a type of linear data structure that stores a fixed-size collection of elements of the same data type in a contiguous memory location. Each element in the array is identified by its index or position, starting from 0.



Basic terminologies of array

Array Index: In an array, elements are identified by their indexes. Array index starts from 0.

Array element: Elements are items stored in an array and can be accessed by their index.

Array Length: The length of an array is determined by the number of elements it can contain.

Why are Array Data Structures needed?

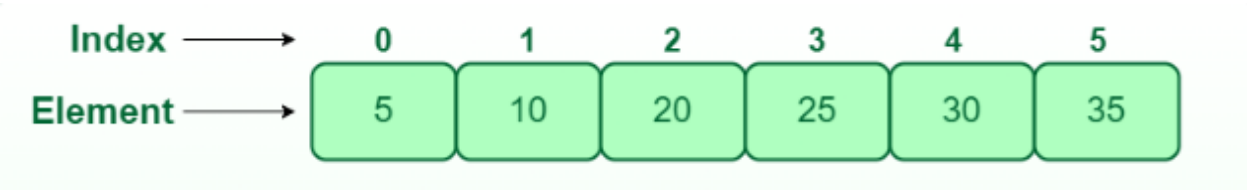
Assume there is a class of five students and if we have to keep records of their marks in examination then, we can do this by declaring five variables individual and keeping track of records but what if the number of students becomes very large, It would be challenging to manipulate and maintain the data.

What it means is that we can use normal variables (v1, v2, v3, ..) when we have a small number of objects. But if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an **array** is to represent many instances in one variable.

Types of Array:

- **1-D array:**

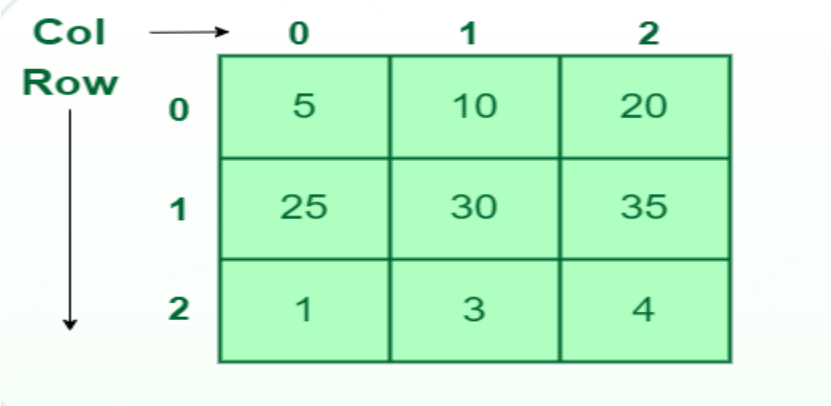
You can imagine a 1d array as a row, where elements are stored one after another.



- **Multidimensional Array:**

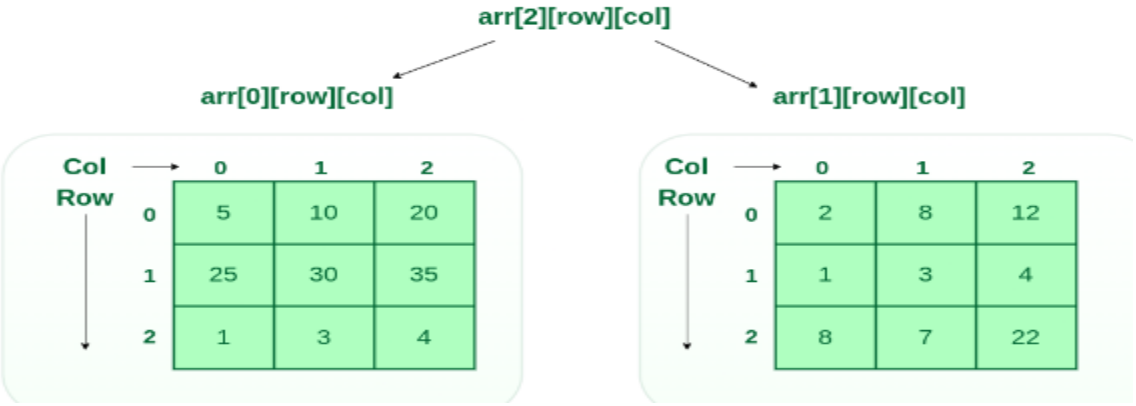
2-D array:

2D array can be considered as an array of arrays or as a matrix consisting of rows and columns.



3-D array:

A 3-D array contains three dimensions, so it can be considered an array of two-dimensional arrays.



Array operations:

Traversal: Traverse through the elements of an array.

Insertion: Inserting a new element in an array.

Deletion: Deleting element from the array.

Updating: Updating the value of a particular index.

Searching: Search for an element in the array.

Sorting: Maintaining the order of elements in the array.

Advantages:

1. Arrays store multiple data of similar types with the same name.
2. Array data structures are used to implement other data structures like linked lists, stacks, queues, trees, graphs, etc.

Disadvantages:

1. As arrays have a fixed size, once the memory is allocated to them, it cannot be increased or decreased, making it impossible to store extra data if required.
2. If the array is allocated with a size larger than required, it may lead to memory wastage.
3. Inserting or deleting elements from an array can be inefficient. If an element is inserted or removed, other elements might need to be shifted, resulting in a high time complexity say O(n).
4. If you need to resize an array (to add more elements, for example), it often involves creating a new array, copying elements over, and then deleting the old array, which can be computationally expensive.

Theoretical questions on arrays:

https://www.geeksforgeeks.org/introduction-to-arrays-data-structure-and-algorithm-tutorials/

More about arrays:

https://www.geeksforgeeks.org/array-data-structure/

add description ->>> https://leetcode.com/problems/number-of-arithmetic-triplets/

Q: Find second largest number in the array

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vec = {0, 4, 1, 3, 5};
    int maxi1 = vec[0];
    int maxi2 = -1;
    for (int i = 1; i < vec.size(); i++)
    {
        if (maxi1 < vec[i])
        {
            maxi2 = maxi1;
            maxi1 = vec[i];
        }
        else if (maxi1 > vec[i] && maxi2 < vec[i])
        {
            maxi2 = vec[i];
        }
    }
    cout << maxi2;
}
```

Time complexity: O(N) space complexity: O(1)

Q: Remove duplicates from sorted array.

```cpp
    int removeDuplicates(vector<int>& nums) {
        int count=1;
        int i=0;
        for(int j=1;j<nums.size();j++){
            if(nums[i]!=nums[j]){
                nums[i+1]=nums[j];
                count++;
                i++;
            }
```

```
        }
        return count;
    }
}
```

Time complexity: O(N) space complexity: O(1)

Q: Left rotate an array by D places.

Approach 1:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vec = {1, 2, 3, 4, 5};
    int ro = 2;
    int size = ro % vec.size();
    int temp[size];
    for (int i = 0; i < size; i++)
    {
        temp[i] = vec[i];
    }
    for (int i = size; i < vec.size(); i++)
    {
        vec[i - size] = vec[i];
    }
    for (int i = vec.size() - size; i < vec.size(); i++)
    {
        vec[i] = temp[i - (vec.size() - size)];
    }

    for (int i = 0; i < vec.size(); i++)
    {
        cout << vec[i] << endl;
```

```
    }
}
```

Time complexity: O(N+size) space complexity: O(size)

Approach 2:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> vec = {1, 2, 3, 4, 5};
    int ro = 3;
    int size = ro % vec.size();
    reverse(vec.begin(), vec.begin() + size);
    reverse(vec.begin() + size, vec.end());
    reverse(vec.begin(), vec.end());
    for (auto i : vec)
    {
        cout << i << " ";
    }
}
}
```

Time complexity: O(2N) space complexity: O(1)

Q: Move zeros to end.

```
#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> vec = {1, 0, 2, 3, 0, 0, 1, 2};
    int j = -1;
    for (int idx = 0; idx < vec.size(); idx++)
    {
        if (vec[idx] == 0)
        {
            j = idx;
            break;
        }
    }

    for (int i = j + 1; i < vec.size(); i++)
    {
        if (vec[i] != 0)
        {
            swap(vec[i], vec[j]);
            j++;
        }
    }
    for (int i = 0; i < vec.size(); i++)
    {
        cout << vec[i] << " ";
    }
}
```

Time complexity: O(N) space complexity: O(1)

Q: Union of 2 sorted array.

```cpp
#include <iostream>
#include <vector>

using namespace std;
```

```cpp
int main()
{
    vector<int> vec1 = {1, 1, 2, 2, 3, 4};
    vector<int> vec2 = {1, 2, 3, 3, 4, 5, 6};
    vector<int> unionArr;
    int i = 0;
    int j = 0;
    while (i < vec1.size() && j < vec2.size())
    {
        if (vec1[i] >= vec2[j])
        {
            if (unionArr.size() == 0 || unionArr.back() != vec2[j])
            {
                unionArr.push_back(vec2[j]);
            }

            j++;
        }
        else if (vec2[j] >= vec1[i])
        {
            if (unionArr.size() == 0 || unionArr.back() != vec1[i])
            {
                unionArr.push_back(vec1[i]);
            }

            i++;
        }
    }

    while (i < vec1.size())
    {
        if (unionArr.back() != vec1[i])
        {
            unionArr.push_back(vec1[i]);
        }
        i++;
    }
    while (j < vec2.size())
    {
        if (unionArr.back() != vec2[j])
        {
            unionArr.push_back(vec2[j]);
        }
        j++;
    }
```

```
    for (int idx = 0; idx < unionArr.size(); idx++)
    {
        cout << unionArr[idx] << " ";
    }
}
```

Time complexity: O(N1+N2)

space complexity: O(N1+N2)

Q: Intersection of 2 sorted array.

```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
    vector<int> vec1 = {1, 1, 2, 2, 3, 4};
    vector<int> vec2 = {1, 2, 3, 3, 4, 5, 6};
    vector<int> IntersecArr;
    int i = 0;
    int j = 0;
    while (i < vec1.size() && j < vec2.size())
    {
        if (vec1[i] > vec2[j])
        {
            j++;
        }
        else if (vec1[i] < vec2[j])
        {
            i++;
        }
        else
        {
            IntersecArr.push_back(vec1[i]);
```

```
            i++;
            j++;
        }
    }

    for (int idx = 0; idx < IntersecArr.size(); idx++)
    {
        cout << IntersecArr[idx] << " ";
    }
}
```

Time complexity: O(N1+N2) space complexity: O(n)

Q: Find the missing number.

```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
    vector<int> vec = {1, 2, 4, 5};
    int n = 5;
    int actualsum = (n * (n + 1)) / 2;
    int itrsum = 0;
    for (int i = 0; i < vec.size(); i++)
    {
        itrsum += vec[i];
    }
    cout << actualsum - itrsum;
}
```

Time complexity: O(N) space complexity: O(1)

Q: Longest Subarray with sum k and print subarray as well.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vec = {1, 1, 1, 2, 3, 2, 1};
    int left = 0;
    int right = 0;
    int sum = vec[0];
    int maxLen = 0;
    int maxLeft = 0;
    int maxRight = 0;
    int k = 3;

    while (right < vec.size())
    {
        while (left <= right && sum > k)
        {
            sum -= vec[left];
            left++;
        }

        if (sum == k && maxLen < right - left + 1)
        {
            maxLen = right - left + 1;
            maxLeft = left;
            maxRight = right;
        }

        right++;
        if (right < vec.size())
            sum += vec[right];
    }

    cout << "Max Length: " << maxLen << endl;
    cout << "Subarray with Sum equal to " << k << ": ";
```

```
    for (int i = maxLeft; i <= maxRight; i++)
    {
        cout << vec[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Time complexity: O(N) space complexity: O(1)

Q: 2sum

Approach 1:

```
vector<int> twoSum(vector<int> &nums, int target)
{
    unordered_map<int, int> numMap;
    vector<int> result;

    for (int i = 0; i < nums.size(); ++i)
    {
        int complement = target - nums[i];
        if (numMap.find(complement) != numMap.end())
        {
            result.push_back(numMap[complement]);
            result.push_back(i);
            return result;
        }
        numMap[nums[i]] = i;
    }

    return result;
}
```

Time complexity: O(N) space complexity: O(1)

Approach 2:

```cpp
vector<int> twoSum(vector<int> &nums, int target)
{
    vector<int> result;

    // Sorting the input array
    sort(nums.begin(), nums.end());

    // Two pointers approach
    int left = 0;
    int right = nums.size() - 1;
    while (left < right)
    {
        int sum = nums[left] + nums[right];
        if (sum == target)
        {
            result.push_back(left);
            result.push_back(right);
            return result;
        }
        else if (sum < target)
        {
            left++;
        }
        else
        {
            right--;
        }
    }

    return result;
}
```

Time complexity: O(N* Log N) space complexity: O(1)

Q: Sort array of 0, 1 ,2 (Dutch national flag algo)

```cpp
void sortColors(vector<int> &nums)
{
    int low = 0;                // Pointer for 0
    int mid = 0;                // Pointer for 1
    int high = nums.size() - 1; // Pointer for 2

    while (mid <= high)
    {
        if (nums[mid] == 0)
        {
            swap(nums[low], nums[mid]);
            low++;
            mid++;
        }
        else if (nums[mid] == 1)
        {
            mid++;
        }
        else
        { // nums[mid] == 2
            swap(nums[mid], nums[high]);
            high--;
        }
    }
}
```

Time complexity: O(N) space complexity: O(1)

Q: Majority Element (Moore’s voting algo)

```cpp
int main()
{
    vector<int> vec = {1, 1, 1, 2, 3, 2, 1};
    int el = vec[0];
    int cnt = 1;
    for (int i = 0; i < vec.size(); i++)
    {
        if (cnt == 0)
        {
            el = vec[i];
```

```
            cnt = 1;
        }
        else if (vec[i] == el)
        {
            cnt++;
        }
        else
        {
            cnt--;
        }
    }
    int cnt1 = 0;
    for (int i = 0; i < vec.size(); i++)
    {
        if (vec[i] == el)
        {
            cnt1++;
        }
    }
    if (cnt1 > vec.size() / 2)
    {
        cout << el;
    }

    cout << -1;
}
```

Time complexity: O(N) space complexity: O(1)

Q: Maximum Subarray sum also print subarray as well (Kadane algo)

```
#include <iostream>
#include <vector>
using namespace std;
```

```cpp
int main()
{
    vector<int> vec = {1, 1, -3, 2, 1};
    int maxi = -99999999;
    int sum = 0;
    int start = 0, end = 0; // Variables to store the indices of the subarray
with maximum sum

    for (int i = 0; i < vec.size(); i++)
    {
        sum += vec[i];
        if (sum > maxi)
        {
            maxi = sum;
            end = i; // Update the end index of the subarray
        }
        if (sum < 0)
        {
            sum = 0;
            start = i + 1; // Reset start index when sum becomes negative
        }
    }

    cout << "Maximum sum: " << maxi << endl;
    cout << "Subarray with maximum sum: ";
    for (int i = start; i <= end; i++)
    {
        cout << vec[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Time complexity: O(N) space complexity: O(1)

Q: Rearrange array element by sign

```cpp
int main()
{
    vector<int> vec = {3, 1, -2, -5, 2, -4};
    vector<int> ans(vec.size(), 0);
    int poidx = 0;
    int negidx = 1;
    for (int i = 0; i < vec.size(); i++)
    {
        if (vec[i] < 0)
        {
            ans[negidx] = vec[i];
            negidx += 2;
        }
        else
        {
            ans[poidx] = vec[i];
            poidx += 2;
        }
    }
    for (auto i : ans)
    {
        cout << i << " ";
    }
}
```

Time complexity: O(N) space complexity: O(N)

Q: Leaders in an array

```cpp
int main()
{
    vector<int> vec = {10, 22, 12, 3, 0, 6};
    vector<int> ans;
    ans.push_back(vec[vec.size() - 1]);
    int maxi = vec[vec.size() - 1];
    for (int i = vec.size() - 2; i >= 0; i--)
    {
        if (vec[i] > maxi)
        {
            ans.push_back(vec[i]);
```

```
        }
        maxi = max(maxi, vec[i]);
    }
    for (auto i : ans)
    {
        cout << i << endl;
    }
}
```

Time complexity: O(N) space complexity: O(N)

Q: Longest consecutive sequence

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

int longestSuccessiveElements(vector<int> &a)
{
    int n = a.size();
    if (n == 0)
        return 0;

    int longest = 1;
    unordered_set<int> st;
    // put all the array elements into set:
    for (int i = 0; i < n; i++)
    {
        st.insert(a[i]);
    }

    // Find the longest sequence:
    for (auto it : st)
    {
        // if 'it' is a starting number:
        if (st.find(it - 1) == st.end())
```

```
        {
            // find consecutive numbers:
            int cnt = 1;
            int x = it;
            while (st.find(x + 1) != st.end())
            {
                x = x + 1;
                cnt = cnt + 1;
            }
            longest = max(longest, cnt);
        }
    }
    return longest;
}

int main()
{
    vector<int> a = {100, 200, 1, 2, 3, 4};
    int ans = longestSuccessiveElements(a);
    cout << "The longest consecutive sequence is " << ans << "\n";
    return 0;
}
```

Time complexity: O(N) space complexity: O(N)

Q: Next Permutation

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

vector<int> nextGreaterPermutation(vector<int> &A)
{
    int n = A.size(); // size of the array.

    // Step 1: Find the break point:
    int ind = -1; // break point
```

```cpp
    for (int i = n - 2; i >= 0; i--)
    {
        if (A[i] < A[i + 1])
        {
            // index i is the break point
            ind = i;
            break;
        }
    }

    // If break point does not exist:
    if (ind == -1)
    {
        // reverse the whole array:
        reverse(A.begin(), A.end());
        return A;
    }

    // Step 2: Find the next greater element
    //         and swap it with arr[ind]:

    for (int i = n - 1; i > ind; i--)
    {
        if (A[i] > A[ind])
        {
            swap(A[i], A[ind]);
            break;
        }
    }

    // Step 3: reverse the right half:
    reverse(A.begin() + ind + 1, A.end());

    return A;
}

int main()
{
    vector<int> A = {2, 1, 5, 4, 3, 0, 0};
    vector<int> ans = nextGreaterPermutation(A);

    cout << "The next permutation is: [";
    for (auto it : ans)
    {
        cout << it << " ";
```

```
    }
    cout << "]n";
    return 0;
}
```

Time complexity: O(N) space complexity: O(1)

Q: Set matrix zero

Approach 1:

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

void markRow(vector<vector<int>> &matrix, int n, int m, int i)
{
    // set all non-zero elements as -1 in the row i:
    for (int j = 0; j < m; j++)
    {
        if (matrix[i][j] != 0)
        {
            matrix[i][j] = -1;
        }
    }
}

void markCol(vector<vector<int>> &matrix, int n, int m, int j)
{
    // set all non-zero elements as -1 in the col j:
    for (int i = 0; i < n; i++)
    {
        if (matrix[i][j] != 0)
        {
            matrix[i][j] = -1;
        }
    }
}

vector<vector<int>> zeroMatrix(vector<vector<int>> &matrix, int n, int m)
```

```
{

    // Set -1 for rows and cols
    // that contains 0. Don't mark any 0 as -1:

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (matrix[i][j] == 0)
            {
                markRow(matrix, n, m, i);
                markCol(matrix, n, m, j);
            }
        }
    }

    // Finally, mark all -1 as 0:
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (matrix[i][j] == -1)
            {
                matrix[i][j] = 0;
            }
        }
    }

    return matrix;
}

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {1, 0, 1}, {1, 1, 1}};
    int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>> ans = zeroMatrix(matrix, n, m);

    cout << "The Final matrix is: n";
    for (auto it : ans)
    {
        for (auto ele : it)
        {
            cout << ele << " ";
```

```
        }
        cout << "n";
    }
    return 0;
}
```

Time complexity: O(O((N*M)*(N + M)) + O(N*M)) space complexity: O(1)

Approach 2:

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

vector<vector<int>> zeroMatrix(vector<vector<int>> &matrix, int n, int m)
{

    int row[n] = {0}; // row array
    int col[m] = {0}; // col array

    // Traverse the matrix:
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (matrix[i][j] == 0)
            {
                // mark ith index of row wih 1:
                row[i] = 1;

                // mark jth index of col wih 1:
                col[j] = 1;
            }
        }
    }

    // Finally, mark all (i, j) as 0
    // if row[i] or col[j] is marked with 1.
    for (int i = 0; i < n; i++)
    {
```

```
        for (int j = 0; j < m; j++)
        {
            if (row[i] || col[j])
            {
                matrix[i][j] = 0;
            }
        }
    }

    return matrix;
}

int main()
{
    vector<vector<int>> matrix = {{1, 1, 1}, {1, 0, 1}, {1, 1, 1}};
    int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>> ans = zeroMatrix(matrix, n, m);

    cout << "The Final matrix is: n";
    for (auto it : ans)
    {
        for (auto ele : it)
        {
            cout << ele << " ";
        }
        cout << "n";
    }
    return 0;
}
```

Time complexity: O(2* (N*M)) space complexity: O(N+M)

Q: Rotate matrix by 90 degree.

Approach 1:

```cpp
#include <bits/stdc++.h>

using namespace std;
vector<vector<int>> rotate(vector<vector<int>> &matrix)
{
    int n = matrix.size();
    vector<vector<int>> rotated(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            rotated[j][n - i - 1] = matrix[i][j];
        }
    }
    return rotated;
}

int main()
{
    vector<vector<int>> arr;
    arr = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    vector<vector<int>> rotated = rotate(arr);
    cout << "Rotated Image" << endl;
    for (int i = 0; i < rotated.size(); i++)
    {
        for (int j = 0; j < rotated[0].size(); j++)
        {
            cout << rotated[i][j] << " ";
        }
        cout << "n";
    }
}
```

Time complexity: O(N*M) space complexity: O(N*M)

Approach 2:

```cpp
#include <bits/stdc++.h>
```

```cpp
using namespace std;
void rotate(vector<vector<int>> &matrix)
{
    int n = matrix.size();
    // transposing the matrix
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
    // reversing each row of the matrix
    for (int i = 0; i < n; i++)
    {
        reverse(matrix[i].begin(), matrix[i].end());
    }
}

int main()
{
    vector<vector<int>> arr;
    arr = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    rotate(arr);
    cout << "Rotated Image" << endl;
    for (int i = 0; i < arr.size(); i++)
    {
        for (int j = 0; j < arr[0].size(); j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << "n";
    }
}
```

Time complexity: O(N*M) space complexity: O(1)

Q: Spiral traversal of matrix.

```cpp
#include <bits/stdc++.h>

using namespace std;

vector<int> printSpiral(vector<vector<int>> mat)
{

    // Define ans array to store the result.
    vector<int> ans;

    int n = mat.size();    // no. of nows
    int m = mat[0].size(); // no. of columns

    // Initialize the pointers reqd for traversal.
    int top = 0, left = 0, bottom = n - 1, right = m - 1;

    // Loop until all elements are not traversed.
    while (top <= bottom && left <= right)
    {

        // For moving left to right
        for (int i = left; i <= right; i++)
            ans.push_back(mat[top][i]);

        top++;

        // For moving top to bottom.
        for (int i = top; i <= bottom; i++)
            ans.push_back(mat[i][right]);

        right--;

        // For moving right to left.
        if (top <= bottom)
        {
            for (int i = right; i >= left; i--)
                ans.push_back(mat[bottom][i]);

            bottom--;
        }

        // For moving bottom to top.
        if (left <= right)
        {
            for (int i = bottom; i >= top; i--)
```

```
                ans.push_back(mat[i][left]);

            left++;
        }
    }
    return ans;
}

int main()
{

    // Matrix initialization.
    vector<vector<int>> mat{{1, 2, 3, 4},
                            {5, 6, 7, 8},
                            {9, 10, 11, 12},
                            {13, 14, 15, 16}};

    vector<int> ans = printSpiral(mat);

    for (int i = 0; i < ans.size(); i++)
    {

        cout << ans[i] << " ";
    }

    cout << endl;

    return 0;
}
```

Time complexity: O(N*M) space complexity: O(N*M)

Q: Pascal Triangle

1) Print a number at a particular row and column.

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

int nCr(int n, int r)
```

```
{
    long long res = 1;

    // calculating nCr:
    for (int i = 0; i < r; i++)
    {
        res = res * (n - i);
        res = res / (i + 1);
    }
    return res;
}

int pascalTriangle(int r, int c)
{
    int element = nCr(r - 1, c - 1);
    return element;
}

int main()
{
    int r = 5; // row number
    int c = 3; // col number
    int element = pascalTriangle(r, c);
    cout << "The element at position (r,c) is: "
         << element << "n";
    return 0;
}
```

Time complexity: O(r) space complexity: O(1)

2) Print the particular row

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

void pascalTriangle(int n)
{
    long long ans = 1;
    cout << ans << " "; // printing 1st element

    // Printing the rest of the part:
    for (int i = 1; i < n; i++)
```

```
    {
        ans = ans * (n - i);
        ans = ans / i;
        cout << ans << " ";
    }
    cout << endl;
}

int main()
{
    int n = 5;
    pascalTriangle(n);
    return 0;
}
```

Time complexity: O(N) space complexity: O(1)

3) Print pascal triangle.

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

vector<int> generateRow(int row)
{
    long long ans = 1;
    vector<int> ansRow;
    ansRow.push_back(1); // inserting the 1st element

    // calculate the rest of the elements:
    for (int col = 1; col < row; col++)
    {
        ans = ans * (row - col);
        ans = ans / col;
        ansRow.push_back(ans);
    }
    return ansRow;
}

vector<vector<int>> pascalTriangle(int n)
```

```
{
    vector<vector<int>> ans;

    // store the entire pascal's triangle:
    for (int row = 1; row <= n; row++)
    {
        ans.push_back(generateRow(row));
    }
    return ans;
}

int main()
{
    int n = 5;
    vector<vector<int>> ans = pascalTriangle(n);
    for (auto it : ans)
    {
        for (auto ele : it)
        {
            cout << ele << " ";
        }
        cout << "n";
    }
    return 0;
}
```

Time complexity: O(N*N) space complexity: O(1)

Searching algorithms:

- **Linear Search/ Sequential Search:**

  Sequential search is a simple algorithm that starts at the beginning of the array searches for the target value one element at a time and continues looking at the

target value in whole array. If the target value is found, the algorithm returns the index of the element.

Time complexity: O(n)-> where n is the size of the array

```
int linearSearch(int arr[], int target, int size)

{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == target)
        {
            return i;  // element found at index i
        }
    }
    return -1; // element not found
}
```

- **Binary Search:**

Binary search is a more efficient algorithm that works by repeatedly dividing the array in half and searching the half that is most likely to contain the target value. The algorithm starts by comparing the target value to the middle element of the array. If the target value is equal to the middle element, the algorithm returns the index of the element. If the target value is less than the middle element, the algorithm searches the lower half of the array. If the target value is greater than the middle element, the algorithm searches the upper half of the array. The algorithm continues dividing the array in half until the target value is found or until the array is empty.

Note: This algorithm is only applicable if the array is sorted.

Time complexity: O(log n)-> where n is the size of an array.

```
int binarySearch(int arr[], int l, int r, int x)

{
    while (l <= r)
    {
        int m = (l + r) / 2;
```

```
        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was not present
    return -1;
}
```

Q: First and last occurrence in an array

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

int firstOccurrence(vector<int> &arr, int n, int k) {
    int low = 0, high = n - 1;
    int first = -1;

    while (low <= high) {
        int mid = (low + high) / 2;
        // maybe an answer
        if (arr[mid] == k) {
            first = mid;
            //look for smaller index on the left
            high = mid - 1;
        }
        else if (arr[mid] < k) {
            low = mid + 1; // look on the right
        }
```

```
        else {
            high = mid - 1; // look on the left
        }
    }
    return first;
}

int lastOccurrence(vector<int> &arr, int n, int k) {
    int low = 0, high = n - 1;
    int last = -1;

    while (low <= high) {
        int mid = (low + high) / 2;
        // maybe an answer
        if (arr[mid] == k) {
            last = mid;
            //look for larger index on the right
            low = mid + 1;
        }
        else if (arr[mid] < k) {
            low = mid + 1; // look on the right
        }
        else {
            high = mid - 1; // look on the left
        }
    }
    return last;
}


pair<int, int> firstAndLastPosition(vector<int>& arr, int n, int k) {
    int first = firstOccurrence(arr, n, k);
    if (first == -1) return { -1, -1};
    int last = lastOccurrence(arr, n, k);
    return {first, last};
}


int main()
{
    vector<int> arr =  {2, 4, 6, 8, 8, 8, 11, 13};
    int n = 8, k = 8;
    pair<int, int> ans = firstAndLastPosition(arr, n, k);
    cout << "The first and last positions are: "
         << ans.first << " " << ans.second << "\n";
```

```
    return 0;
}
```

Time complexity: O(2* Log n) space complexity: O(1)

Q: Search an element in rotated sorted array 1.

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

int search(vector<int>& arr, int n, int k) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;

        //if mid points the target
        if (arr[mid] == k) return mid;

        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k <= arr[mid]) {
                //element exists:
                high = mid - 1;
            }
            else {
                //element does not exist:
                low = mid + 1;
            }
        }
        else { //if right part is sorted:
            if (arr[mid] <= k && k <= arr[high]) {
                //element exists:
                low = mid + 1;
            }
            else {
                //element does not exist:
                high = mid - 1;
            }
```

```
        }
    }
    return -1;
}

int main()
{
    vector<int> arr = {7, 8, 9, 1, 2, 3, 4, 5, 6};
    int n = 9, k = 1;
    int ans = search(arr, n, k);
    if (ans == -1)
        cout << "Target is not present.\n";
    else
        cout << "The index is: " << ans << "\n";
    return 0;
}
```

Time complexity: O(Log n)

space complexity: O(1)

Q: Search an element in rotated sorted array !!.

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

bool searchInARotatedSortedArrayII(vector<int>&arr, int k) {
    int n = arr.size(); // size of the array.
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;

        //if mid points the target
        if (arr[mid] == k) return true;

        //Edge case:
        if (arr[low] == arr[mid] && arr[mid] == arr[high]) {
            low = low + 1;
            high = high - 1;
            continue;
        }
```

```
        //if left part is sorted:
        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k <= arr[mid]) {
                //element exists:
                high = mid - 1;
            }
            else {
                //element does not exist:
                low = mid + 1;
            }
        }
        else { //if right part is sorted:
            if (arr[mid] <= k && k <= arr[high]) {
                //element exists:
                low = mid + 1;
            }
            else {
                //element does not exist:
                high = mid - 1;
            }
        }
    }
    return false;
}

int main()
{
    vector<int> arr = {7, 8, 1, 2, 3, 3, 3, 4, 5, 6};
    int k = 3;
    bool ans = searchInARotatedSortedArrayII(arr, k);
    if (!ans)
        cout << "Target is not present.\n";
    else
        cout << "Target is present in the array.\n";
    return 0;
}
```

Time complexity: O(Log n) space complexity: O(1)

Sorting algorithms:

- **Selection Sort:**

  Given an array of numbers: [5, 2, 4, 1, 3], let's perform the selection sort step by step:
  Step 1: Find the smallest number in the array (1) and swap it with the element at the beginning of the array:
  Array after step: [1, 2, 4, 5, 3]
  Step 2: Find the smallest number in the remaining unsorted portion of the array (2) and swap it with the second element:
  Array after step: [1, 2, 4, 5, 3]
  Step 3: Find the smallest number in the remaining unsorted portion of the array (3) and swap it with the third element:
  Array after step: [1, 2, 3, 5, 4]
  Step 4: Find the smallest number in the remaining unsorted portion of the array (4) and swap it with the fourth element: Array after step: [1, 2, 3, 4, 5] ->> it is now sorted.

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

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

        // Find the index of the minimum element in the unsorted portion
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }

        // Swap the minimum element with the current element
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

Time complexity: O(n^2) in all cases including best, average, and worst. Because in case if array is already sorted the algorithm will still iterate through the array to find the minimum value.

- **Bubble Sort:**

Given an array of numbers: [5, 2, 4, 1, 3], let's perform the bubble sort step by step:
Pass 1: Compare adjacent elements and swap if necessary. Repeat until the largest element "bubbles up" to the end:

Comparing 5 and 2: Swap (array after swap: [2, 5, 4, 1, 3])
Comparing 5 and 4: Swap (array after swap: [2, 4, 5, 1, 3])
Comparing 5 and 1: Swap (array after swap: [2, 4, 1, 5, 3])
Comparing 5 and 3: Swap (array after swap: [2, 4, 1, 3, 5])
Pass 1 complete: Largest element "bubbled up" to the end.

Pass 2: Repeat the process for the remaining unsorted portion (excluding the last element which is already in its correct position):

Comparing 2 and 4: No swap (array remains the same)
Comparing 4 and 1: Swap (array after swap: [2, 1, 4, 3, 5])
Comparing 4 and 3: Swap (array after swap: [2, 1, 3, 4, 5])
Pass 2 complete.

Pass 3: Continue the process for the remaining unsorted portion:
Comparing 2 and 1: Swap (array after swap: [1, 2, 3, 4, 5])
Pass 3 complete.

Pass 4: Final pass (since there are only two elements left to compare):
Comparing 1 and 2: No swap (array remains the same).
Pass 4 is complete.
The array is now sorted! The final sorted array is [1, 2, 3, 4, 5].

```
void bubbleSort(int arr[], int n)

{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
```

```
            }
        }
    }
}
```

Time complexity: O(n^2) in average and best cases. But in case of best cases time complexity will be O(n) because in the best case, the array will be sorted and in 2nd loop, only comparisons will be made and code inside the loop(swapping) will not be done.

- **Insertion Sort:**

  Given an array of numbers: [5, 2, 4, 1, 3], let's perform the insertion sort step by step:

  Initial Array: [5, 2, 4, 1, 3]
  Pass 1: The first element, 5, is already considered sorted.
  Array after pass: [5, 2, 4, 1, 3]

  Pass 2: Consider the second element, 2. Compare it with the previous element and shift larger elements to the right until the correct position is found:
  Shift 5 to the right.
  Insert 2 in the first position.
  Array after pass: [2, 5, 4, 1, 3]

  Pass 3: Consider the third element, 4. Compare it with the previous element and shift larger elements to the right until the correct position is found:
  Insert 4 between 2 and 5.
  Array after pass: [2, 4, 5, 1, 3]

  Pass 4: Consider the fourth element, 1. Compare it with the previous element and shift larger elements to the right until the correct position is found:
  Shift 5 and 4 to the right.
  Shift 2 to the right.
  Insert 1 in the first position.
  Array after pass: [1, 2, 4, 5, 3]

  Pass 5: Consider the fifth element, 3. Compare it with the previous element and shift larger elements to the right until the correct position is found:
  Shift 5 to the right.
  Shift 4 to the right.
  Insert 3 between 2 and 4.

Array after pass: [1, 2, 3, 4, 5]
The array is now sorted! The final sorted array is [1, 2, 3, 4, 5].

```
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Time complexity: O(n^2) in average and best cases. But in case of best cases time complexity will be O(n) because in the best case, the array will be sorted and the code inside the inner loop will not run the condition hardly will become true.

Subarrays:

The contiguous part of an array is called a subarray. Like in an example of array [1,2,3,4] the subarrays are [1,2,3], [2,3,4], [3,4] while [1,3,4] is not subarray.

Note: *The total number of subarrays in an array is n*(n+1)/2*

Subsequence:

A subsequence of an array is a sequence of elements obtained by deleting some elements from the original array without changing the order of the remaining elements. Like in array [1,2,3,4] the possible subsequences are [1,2], [1,2,3], [1,3,4] etc.

Note: The total number of subsequences in an array is 2^n

"Every subarray is a subsequence, but every subsequence is not a subarray"

Techniques:

➢ Sliding Window Technique

The sliding window technique is a widely used algorithmic approach for efficiently processing arrays or sequences in a way that reduces time complexity. It's particularly useful when you need to maintain a fixed-width "window" over a sequence of elements and optimize a specific metric within that window.

Problems:

- Maximum Sum Subarray of Fixed Length (or Size K)
- Minimum Size Subarray Sum
- Longest Substring Without Repeating Characters
- Substring with Distinct Elements

Q: Find the maximum sum of a subarray of a fixed length k

```
int maxSubarraySum(vector<int> &nums, int k)

{
    int n = nums.size();

    // Calculate the sum of the first 'k' elements as the initial maximum sum.
    int maxSum = 0;
    for (int i = 0; i < k; ++i)
    {
        maxSum += nums[i];
    }

    // Initialize the current sum with the initial maximum sum.
    int currentSum = maxSum;

    // Slide the window and update the maximum sum as needed.
    for (int i = k; i < n; ++i)
    {
      currentSum = currentSum - nums[i - k] + nums[i]; // Update the current sum.
      maxSum = std::max(maxSum, currentSum); // Update the maximum sum.
    }
```

```
    return maxSum;
}
```

Time complexity: O(n)

"The above approach uses fixed-sized sliding window."

The window can be dynamically sized known as a dynamically sized sliding window.

Q: Find the shortest subarray with the sum that is greater than or equal to x.

```
int dynamic_sliding_window(vector<int> &nums, int x)
{
    int n = nums.size();
    int minL = 9999999;
    int start = 0;
    int end = 0;
    int currSum = 0;
    while (end < n)
    {
        currSum = currSum + nums[end];
        end++;
        while (start < end && currSum >= x)
        {
            currSum = currSum - nums[start];
            start++;
            minL = min(minL, end - start + 1);
        }
    }
    return minL;
}
```

Time complexity: O(n). The time complexity is not O(N^2) because the inner while loop, which uses the start pointer, does not iterate through the entire array for each iteration of the outer while loop.

Q: Find the subarray with the given sum.

```cpp
pair<int, int> findSubarrayWithSum(std::vector<int> &nums, int targetSum)

{
    int left = 0;
    int currentSum = 0;

    for (int right = 0; right < nums.size(); ++right)
    {
        currentSum += nums[right];

        while (currentSum > targetSum)
        {
            currentSum -= nums[left];
            ++left;
        }

        if (currentSum == targetSum)
        {
            // A subarray with the target sum is found.
            return std::make_pair(left, right);
        }
    }

    // If no subarray with the target sum is found, return {-1, -1}.
    return std::make_pair(-1, -1);
}
```

Time complexity: O(n). The time complexity is not O(N^2) because the inner while loop, which uses the start pointer, does not iterate through the entire array for each iteration of the outer while loop.

Q: Find the maximum sum of the subarray.

----- (*Kadane's Algorithm*) ----

```cpp
int maxSubarraySum(std::vector<int> &nums)

{
```

```
    int n = nums.size();
    int maxSum = INT_MIN; // Initialize with a very small value.
    int currentSum = 0;

    for (int i = 0; i < n; ++i)
    {
        currentSum += nums[i];

        // Update maxSum if the currentSum becomes greater.
        if (currentSum > maxSum)
        {
            maxSum = currentSum;
        }


       // If currentSum becomes negative, reset it to zero.
        if (currentSum < 0)
        {
            currentSum = 0;
        }
    }

    return maxSum;
}
```

Time complexity: O(n)

Q: Pair Sum problem check if 2 elements exist in an array whose sum is equal to k.

```
bool hasPairWithSum(std::vector<int> &nums, int k)

{
    // Sort the array in non-decreasing order.
    sort(nums.begin(), nums.end());

    int left = 0;
    int right = nums.size() - 1;

    while (left < right)
    {
```

```
        int currentSum = nums[left] + nums[right];

        if (currentSum == k)
        {
            // Pair with the sum k is found.
            return true;
        }
        else if (currentSum < k)
        {
            // Move the left pointer to increase the sum.
            left++;
        }
        else
        {
            // Move the right pointer to decrease the sum.
            right--;
        }
    }

    // No pair with the sum k is found.
    return false;
}
```

*Time complexity: O(n*logn)*

Q: Longest Even odd subarray with threshold.

```
int longestAlternatingSubarray(vector<int>& nums, int threshold) {

         int i=0;
         int count=0;
         bool flag=false;
         int maxL=0;
         while(i<nums.size()){
             if(!flag){
                if(nums[i]%2==0 && nums[i]<=threshold){
                    count=1;
                    flag=true;
                    i++;
                }else{
                    i++;
```

```
                    flag=false;
                }
            }
            else if(flag){
                if(nums[i]%2!=nums[i-1]%2 && nums[i]<=threshold){
                    count++;
                    i++;
                }
                else{
                    flag=false;
                    count=0;
                }
            }
            maxL=max(maxL,count);
        }
        return maxL;

    }
```

Time complexity: O(n)

Q: Count complete subarrays in an array. Or

Number of substrings containing all 3 characters (a,b,c) Or problems like these

```
int countCompleteSubarrays(vector<int> &nums)
{

    int n = nums.size();
    map<int, int> mp;
    for (int i = 0; i < n; i++)
    {
        mp[nums[i]]++;
    }
    map<int, int> mp1;
    int l = 0;
    int ans = 0;
    for (int r = 0; r < n; r++)
    {
        mp1[nums[r]]++;
        while (l <= r && mp1.size() == mp.size())
        {
            mp1[nums[l]]--;
```

```
            if (mp1[nums[l]] == 0)
            {
                mp1.erase(nums[l]);
            }
            l++;
        }
        ans += l;
    }
    return ans;
}
```

Time complexity: O(n)

➢ Prefix Sum Approach:

The prefix sum technique, also known as cumulative sum or cumulative prefix sum, is a fundamental concept in computer science and is particularly useful in various algorithms and data structures. It involves precomputing cumulative sums of elements in an array to efficiently answer range sum queries. The main purpose of computing prefix sums is to efficiently calculate the sum of elements within a range [left, right] of the original array. Consider array is nums = [1, 2, 3, 4, 5]. Final prefix Sum array: [0, 1, 3, 6, 10, 15]. It is used in various problems, Efficient Range Sum Queries, etc. These problems use O(n) time complexity and O(n) space complexity.

Q: Range sum queries.

```
class RangeSum
{
private:
    vector<int> prefixSum;

public:
    RangeSum(vector<int> &nums)
    {
        int n = nums.size();
        prefixSum.resize(n + 1, 0);

        // Compute prefix sum
        for (int i = 0; i < n; i++)
        {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }
```

```
    }

    int sumRange(int left, int right)
    {
        return prefixSum[right + 1] - prefixSum[left];
    }
};
```

Time complexity: O(n)

Space complexity: O(n)

Q: Count the number of nice subarrays.

```
int numberOfSubarrays(vector<int> &nums, int k)
{
    int count = 0;
    int sum = 0;
    for (int i = 0; i < nums.size(); i++)
    {
        if (nums[i] % 2 == 1)
            nums[i] = 1;
        else
            nums[i] = 0;
    }
    unordered_map<int, int> mpp;
    mpp[0] = 1;
    for (int i = 0; i < nums.size(); i++)
    {
        sum += nums[i];
        mpp[sum]++;
        count += mpp[sum - k];
    }
    return count;
}
```

Time complexity: O(n)

Space complexity: O(n)

Matrix Problems:

Q: Spiral order traversal of matrix

```
const int n = 4, m = 5; // n-> row  m->column
int arr[n][m];
int row_start = 0, col_start = 0, col_end = m - 1, row_end = n - 1;
while (row_start <= row_end && col_start <= col_end)
{
    // for row start
    for (int col = col_start; col <= col_end; col++)
    {
        cout << arr[row_start][col];
    }
    row_start++;

    // for column end
    for (int row = row_start;row<=row_end;row++){
        cout<<arr[row][col_end];
    }
    col_end--;

    // for row end
    for (int col = col_end; col >= col_start;col--){
        cout << arr[row_end][col];
    }
    row_end--;
```

```
        // for column start
        for(int row=row_end; row >= row_start;row--){
            cout << arr[row][col_start];
        }
        col_start++;
    }
```

Time complexity: O(n^2)

Q: Search a number in a matrix where each row is sorted and each column is sorted in ascending order.

```
    const int n = 4, m = 5; // n-> row  m->column

    int arr[n][m];
    int r = 0, c = m - 1;
    int target = 5;
    while (r < n && c >= 0)
    {
        if (arr[r][c] == target)
            return true;
        if (arr[r][c] > target)
            c--;
        else
            r++;
    }
```

Time complexity: O(n+m)

Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.

Examples are the tower of Hanoi, in-order/preorder/post-order traversals, etc.

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise an infinite loop will occur.

How are recursive functions stored in memory?

Recursion uses more memory because the recursive function adds to the stack with each recursive call and keeps the values there until the call is finished. The

recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

What is the difference between direct and indirect recursion?

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly.

Q: Calculate the sum till n using recursion.

```
int sum(int n)
{
    if (n == 1)
        return 1;
    return n + sum(n - 1);
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Find the nth Fibonacci using recursion.

```
int fib(int n)
{
    if (n == 0 || n == 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

Time complexity: O(2^n) *Space complexity: O(n)*

Q: Check array is sorted or not.

```
bool sorted(int arr[], int n)
{
```

```
    if (n == 1)
        return true;

    bool restArray = sorted(arr + 1, n - 1);
    return (arr[0] < arr[1] && restArray);
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Print numbers till n both in ascending and descending order.

```
void dec(int n)
{
    if (n == 0)
        return;
    cout << n << endl;
    dec(n - 1);
}
```

```
void inc(int n)
{
    if (n == 0)
        return;
    inc(n - 1);
    cout << n << endl;
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Find first and last occurrence of a number in an array.

```
int firstocc(int arr[], int n, int i, int num)
{
    if (i == n)
```

```
        return -1;
    if (arr[i] == num)
        return i;
    return firstocc(arr, n, i + 1, num);
}
```

```
int lastocc(int arr[], int n, int i, int num)
{
    if (i == -1)
        return -1;
    if (arr[i] == num)
        return i;
    return lastocc(arr, n, i - 1, num);
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Reverse a string.

```
void reversing(string s)
{
    if (s.length() == 0)
        return;
    reversing(s.substr(1));
    cout << s[0];
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: In a string replace pi with 3.14.

```
void replacewithPi(string s)
{
    if (s.length() == 0)
        return;
```

```
    if (s[0] == 'p' && s[1] == 'i')
    {
        cout << 3.14;
        replacewithPi(s.substr(2));
    }
    else
    {
        cout << s[0];
        replacewithPi(s.substr(1));
    }
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Reverse an array.

```
void reverse(int arr[], int l, int r)
{
    if (l >= r)
        return;
    swap(arr[l], arr[r]);
    reverse(arr, l + 1, r - 1);
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Check string is palindrome or not

```
bool isPalindrome(string s, int l, int r)
{
    if (l >= r)
        return true;
    if (s[l] != s[r])
    {
        return false;
    }
    return isPalindrome (s, l + 1, r - 1);
```

```
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Printing subsequences of an array.

```
void subsequences(int idx, int arr[], vector<int> &ds, int n)
{
    if (idx == n)
    {
        for (auto i : ds)
        {
            cout << i << " ";
        }
        if(ds.size()==0)
            cout << "{}";
        cout << endl;

        return;

    }

    ds.push_back(arr[idx]);
    subsequences(idx + 1, arr, ds, n);
    ds.pop_back();
    subsequences(idx + 1, arr, ds, n);
}
```

*Time complexity: O(2^n*n)* *Space complexity: O(n)*

Q: Printing subsequences of an array with sum k.

```
void subsequences(int i, vector<int> &ds, int s, int n, int sum, int arr[])
{
    if (i == n)
    {
        if (s == sum)
        {
            for (auto i : ds)
```

```
            {
                cout << i << " ";
            }
            cout << endl;
        }
        return;
    }
    ds.push_back(arr[i]);
    s += arr[i];
    subsequences(i + 1, ds, s, n, sum, arr);
    ds.pop_back();
    s -= arr[i];
    subsequences(i + 1, ds, s, n, sum, arr);
}
```

*Time complexity: O(2^n*n)* *Space complexity: O(n)*

Q: Print the number of subsequences whose sum equal to k.

```
int subsequences(int i, int s, int n, int sum, int arr[])
{
    if (i == n)
    {
        if (s == sum)
        {
            return 1;
        }
        return 0;
    }
    s += arr[i];
    int l = subsequences(i + 1, s, n, sum, arr);
    s -= arr[i];
    int r = subsequences(i + 1, s, n, sum, arr);
    return l + r;
}
```

Time complexity: O(2^n) *Space complexity: O(n)*

Q: Tower of Hanoi

```
void toh(int n, char s, char d, char h)
{
    if (n == 0)
        return;

    toh(n - 1, s, h, d);
    cout << "Move from " << s << " to " << d << endl;
    toh(n - 1, h, d, s);
}
```

Time complexity: O(2^n) *Space complexity: O(n)*

Q: Remove Duplicates

```
string remoDup(string s)
{
    if (s.length() == 0)
        return "";
    char a = s[0];
    string ans = remoDup(s.substr(1));

    if (a == ans[0])
    {
        return ans;
    }
    return (a + ans);
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: In a string move all x to end.

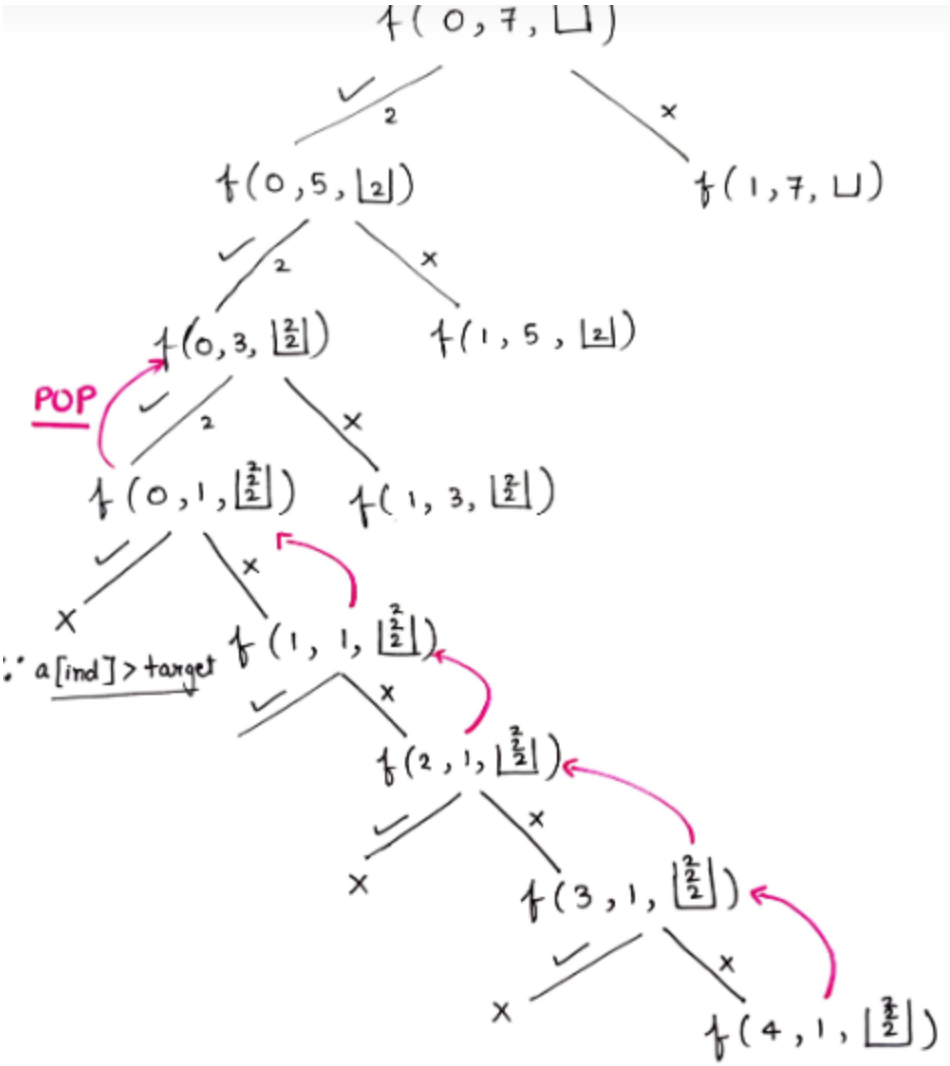
```
string remoDup(string s)
{
    if (s.length() == 0)
        return "";
    char a = s[0];
    string ans = remoDup(s.substr(1));

    if (a == 'x')
    {
```

```
        return ans + a;
    }
    return (a + ans);
}
```

Time complexity: O(n) *Space complexity: O(n)*

Q: Combination sum

https://leetcode.com/problems/combination-sum/

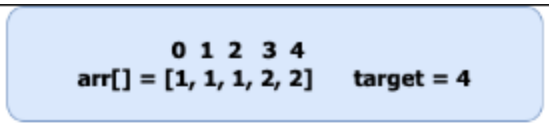


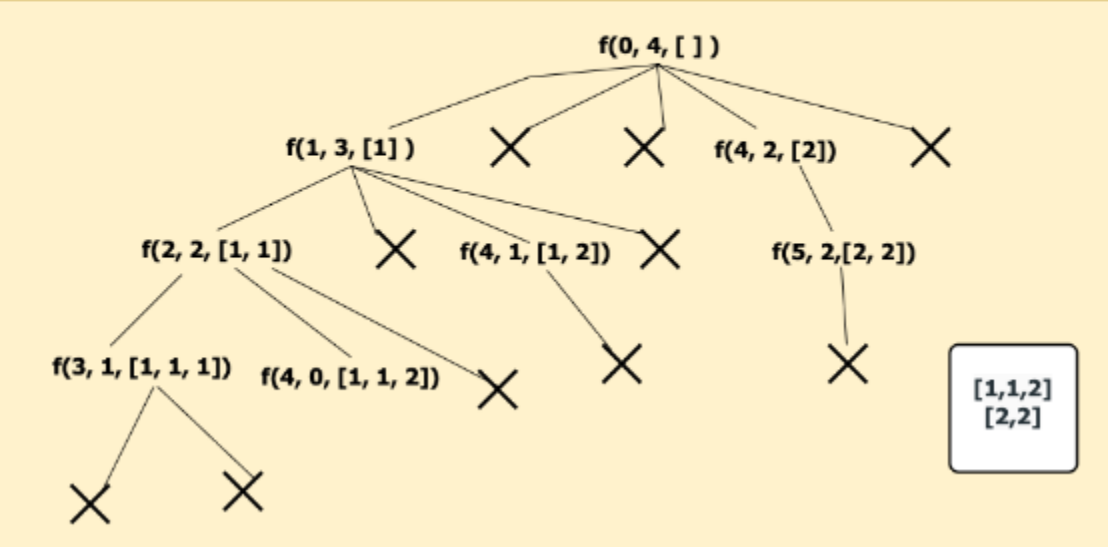

```
void findCombination(int ind, int target, vector<int> &arr, vector<vector<int>>
&ans, vector<int> &ds)
{
    if (ind == arr.size())
    {
        if (target == 0)
        {
            ans.push_back(ds);
        }
        return;
    }
    // pick up the element
    if (arr[ind] <= target)
    {
        ds.push_back(arr[ind]);
        findCombination(ind, target - arr[ind], arr, ans, ds);
```

```
        ds.pop_back();
    }

    findCombination(ind + 1, target, arr, ans, ds);
}
vector<vector<int>> combinationSum(vector<int> &candidates, int target)
{
    vector<vector<int>> ans;
    vector<int> ds;
    findCombination(0, target, candidates, ans, ds);
    return ans;
}
```

Q: Combination sum 2

https://leetcode.com/problems/combination-sum-ii/



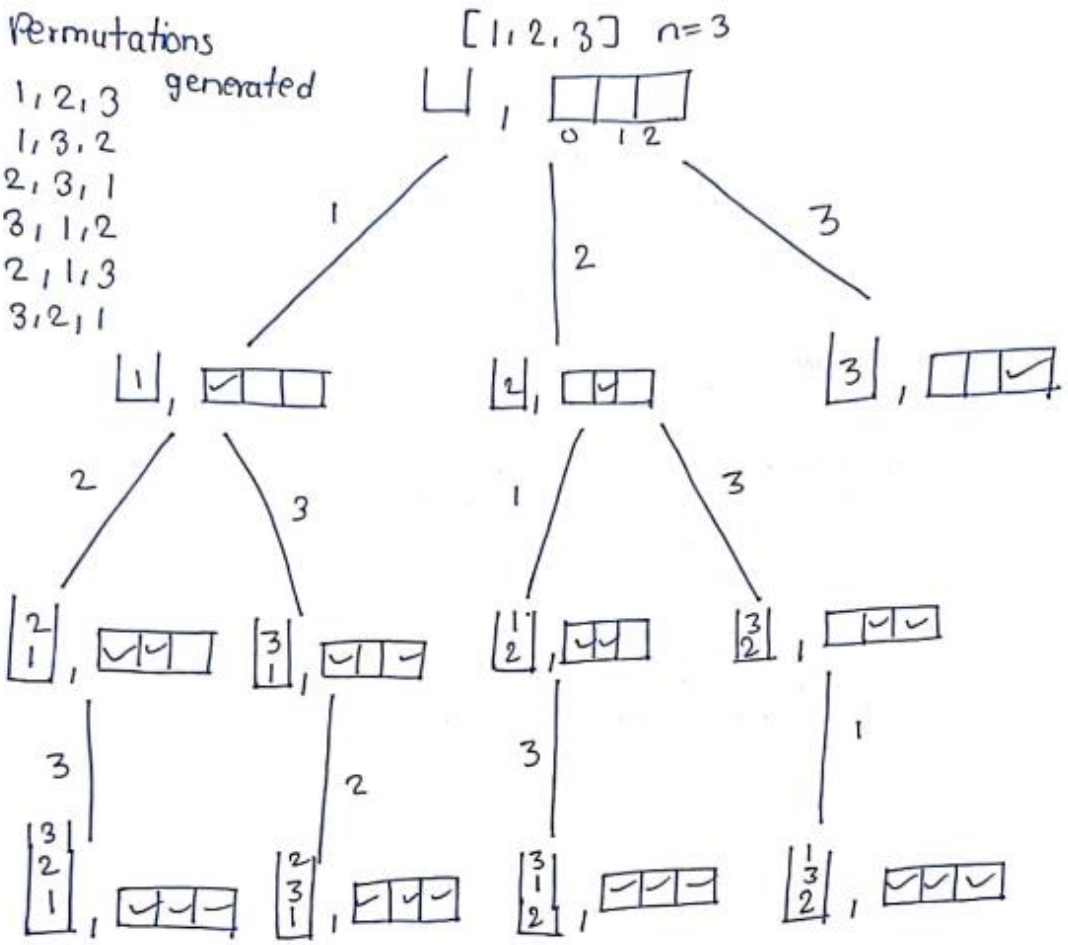



```
void findCombination(int ind, int target, vector<int> &arr, vector<vector<int>>
&ans, vector<int> &ds)
{
    if (target == 0)
    {
        ans.push_back(ds);
        return;
```

```
    }
    for (int i = ind; i < arr.size(); i++)
    {
        if (i > ind && arr[i] == arr[i - 1])
            continue;
        if (arr[i] > target)
            break;
        ds.push_back(arr[i]);
        findCombination(i + 1, target - arr[i], arr, ans, ds);
        ds.pop_back();
    }
}

vector<vector<int>> combinationSum2(vector<int> &candidates, int target)
{
    sort(candidates.begin(), candidates.end());
    vector<vector<int>> ans;
    vector<int> ds;
    findCombination(0, target, candidates, ans, ds);
    return ans;
}
```

Q: Print permutations of array/string.

Approach 1

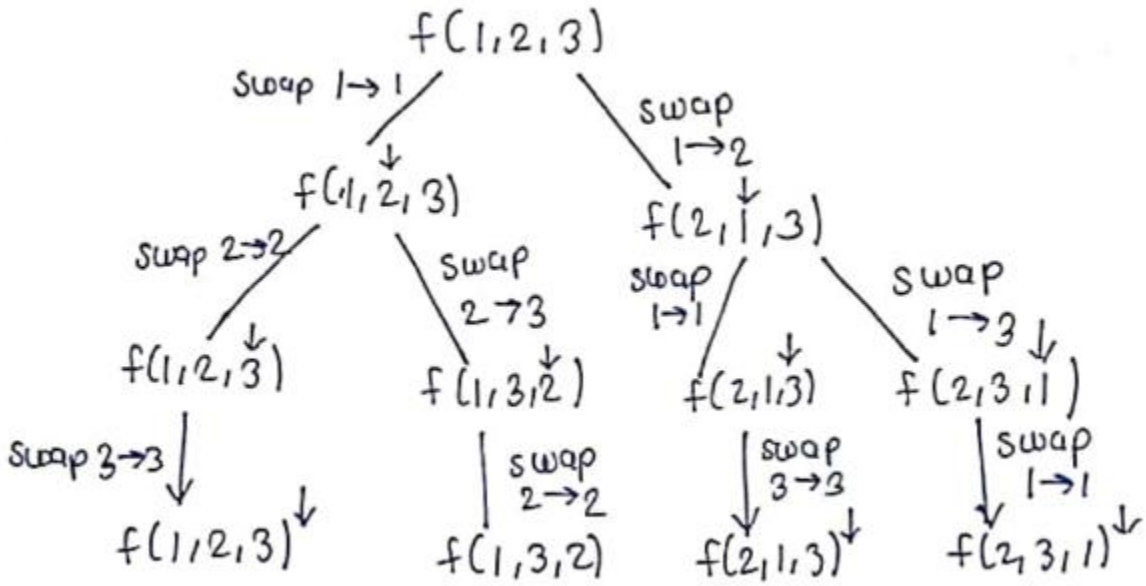



```
void recurPermute(vector<int> &ds, vector<int> &nums, vector<vector<int>> &ans,
int freq[])
{
    if (ds.size() == nums.size())
    {
        ans.push_back(ds);
        return;
    }
    for (int i = 0; i < nums.size(); i++)
    {
        if (!freq[i])
        {
            ds.push_back(nums[i]);
            freq[i] = 1;
            recurPermute(ds, nums, ans, freq);
            freq[i] = 0;
            ds.pop_back();
        }
```

```
    }
}

vector<vector<int>> permute(vector<int> &nums)
{
    vector<vector<int>> ans;
    vector<int> ds;
    int freq[nums.size()];
    for (int i = 0; i < nums.size(); i++)
        freq[i] = 0;
    recurPermute(ds, nums, ans, freq);
    return ans;
}
```

*Time complexity: O(n!*n)* *Space complexity: O(n)+O(n)*

Approach 2:




```
void recurPermute(int index, vector<int> &nums, vector<vector<int>> &ans)
{
    if (index == nums.size())
```

```
    {
        ans.push_back(nums);
        return;
    }
    for (int i = index; i < nums.size(); i++)
    {
        swap(nums[index], nums[i]);
        recurPermute(index + 1, nums, ans);
        swap(nums[index], nums[i]);
    }
}

vector<vector<int>> permute(vector<int> &nums)
{
    vector<vector<int>> ans;
    recurPermute(0, nums, ans);
    return ans;
}
```

*Time complexity: O(n!*n)* *Space complexity: O(n)*

Q: Subset sum 1

```
void subsetSumsHelper(int ind, vector<int> &arr, int n, vector<int> &ans, int
sum)
{
    if (ind == n)
    {
        ans.push_back(sum);
        return;
    }
    // element is picked
    subsetSumsHelper(ind + 1, arr, n, ans, sum + arr[ind]);
    // element is not picked
    subsetSumsHelper(ind + 1, arr, n, ans, sum);
}
vector<int> subsetSums(vector<int> arr, int n)
{
    vector<int> ans;
    subsetSumsHelper(0, arr, n, ans, 0);
    sort(ans.begin(), ans.end());
    return ans;
}
```

*Time complexity: O(2^n*log(2^n))* *Space complexity: O(2^n)*

Q: Subset sum 2

```
void findSubsets(int ind, vector<int> &nums, vector<int> &ds, vector<vector<int>>
&ans)
{
    ans.push_back(ds);
    for (int i = ind; i < nums.size(); i++)
    {
        if (i != ind && nums[i] == nums[i - 1])
            continue;
        ds.push_back(nums[i]);
        findSubsets(i + 1, nums, ds, ans);
        ds.pop_back();
    }
}

vector<vector<int>> subsetsWithDup(vector<int> &nums)
{
    vector<vector<int>> ans;
    vector<int> ds;
    sort(nums.begin(), nums.end());
    findSubsets(0, nums, ds, ans);
    return ans;
}
```

Backtracking

Backtracking is a problem-solving technique used in programming. It involves trying out different possible solutions to a problem and then "backtracking" to find the best or correct solution.

Backtracking is often used for problems where you have to make a series of choices, and if you make a wrong choice at some point, you need to backtrack and try a different approach. It's commonly used in tasks like finding paths in a maze, solving puzzles, or optimizing solutions.

Q: N Queen Problem

```
bool isSafe1(int row, int col, vector<string> board, int n)
{
    // check upper element
    int duprow = row;
    int dupcol = col;

    while (row >= 0 && col >= 0)
    {
        if (board[row][col] == 'Q')
            return false;
        row--;
        col--;
    }

    col = dupcol;
    row = duprow;
    while (col >= 0)
    {
        if (board[row][col] == 'Q')
            return false;
        col--;
    }

    row = duprow;
    col = dupcol;
```

```
    while (row < n && col >= 0)
    {
        if (board[row][col] == 'Q')
            return false;
        row++;
        col--;
    }
    return true;
}

void solve(int col, vector<string> &board, vector<vector<string>> &ans, int n)
{
    if (col == n)
    {
        ans.push_back(board);
        return;
    }
    for (int row = 0; row < n; row++)
    {
        if (isSafe1(row, col, board, n))
        {
            board[row][col] = 'Q';
            solve(col + 1, board, ans, n);
            board[row][col] = '.';
        }
    }
}

vector<vector<string>> solveNQueens(int n)
{
    vector<vector<string>> ans;
    vector<string> board(n);
    string s(n, '.');
    for (int i = 0; i < n; i++)
    {
        board[i] = s;
    }
    solve(0, board, ans, n);
    return ans;
}
```

Q: Sudoko Solver

```
bool isValid(vector<vector<char>> &board, int row, int col, char c)
{
    for (int i = 0; i < 9; i++)
    {
        if (board[i][col] == c)
            return false;

        if (board[row][i] == c)
            return false;

        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
            return false;
    }
    return true;
}

bool solveSudoku(vector<vector<char>> &board)
{
    for (int i = 0; i < board.size(); i++)
    {
        for (int j = 0; j < board[0].size(); j++)
        {
            if (board[i][j] == '.')
            {
                for (char c = '1'; c <= '9'; c++)
                {
                    if (isValid(board, i, j, c))
                    {
                        board[i][j] = c;

                        if (solveSudoku(board))
                            return true;
                        else
                            board[i][j] = '.';
                    }
                }

                return false;
            }
        }
```

```
    }
    return true;
}
```

Q: Palindrome Partitioning

```
bool isPalindrome(string s, int start, int end)
{
    while (start <= end)
    {
        if (s[start++] != s[end--])
            return false;
    }
    return true;
}
void partitionHelper(int index, string s, vector<string> &path,
                     vector<vector<string>> &res)
{
    if (index == s.size())
    {
        res.push_back(path);
        return;
    }
    for (int i = index; i < s.size(); ++i)
    {
        if (isPalindrome(s, index, i))
        {
            path.push_back(s.substr(index, i - index + 1));
            partitionHelper(i + 1, s, path, res);
            path.pop_back();
        }
    }
}

vector<vector<string>> partition(string s)
{
    vector<vector<string>> res;
    vector<string> path;
    partitionHelper(0, s, path, res);
    return res;
}
```

Q: Rat in Maza

1)

```
void findPathHelper(int i, int j, vector<vector<int>> &a, int n, vector<string>
&ans, string move,
                    vector<vector<int>> &vis)
{
    if (i == n - 1 && j == n - 1)
    {
        ans.push_back(move);
        return;
    }

    // downward
    if (i + 1 < n && !vis[i + 1][j] && a[i + 1][j] == 1)
    {
        vis[i][j] = 1;
        findPathHelper(i + 1, j, a, n, ans, move + 'D', vis);
        vis[i][j] = 0;
    }

    // left
    if (j - 1 >= 0 && !vis[i][j - 1] && a[i][j - 1] == 1)
    {
        vis[i][j] = 1;
        findPathHelper(i, j - 1, a, n, ans, move + 'L', vis);
        vis[i][j] = 0;
    }

    // right
    if (j + 1 < n && !vis[i][j + 1] && a[i][j + 1] == 1)
    {
        vis[i][j] = 1;
        findPathHelper(i, j + 1, a, n, ans, move + 'R', vis);
        vis[i][j] = 0;
    }

    // upward
    if (i - 1 >= 0 && !vis[i - 1][j] && a[i - 1][j] == 1)
    {
        vis[i][j] = 1;
        findPathHelper(i - 1, j, a, n, ans, move + 'U', vis);
        vis[i][j] = 0;
    }
```

```
}

vector<string> findPath(vector<vector<int>> &m, int n)
{
    vector<string> ans;
    vector<vector<int>> vis(n, vector<int>(n, 0));

    if (m[0][0] == 1)
        findPathHelper(0, 0, m, n, ans, "", vis);
    return ans;
}
```

2)

```
bool isSafe(int **arr, int x, int y, int n) {
    if (x < n && y < n && arr[x][y] == 1) {
        return true;
    }
    return false;
}

bool ratinMaze(int** arr, int x, int y, int n, int** solArr) {
    if ((x == (n - 1)) && (y == (n - 1))) {
        solArr[x][y] = 1;
        return true;
    }
    if (isSafe(arr, x, y, n)) {
        solArr[x][y] = 1;
        if (ratinMaze(arr, x + 1, y, n, solArr)) {
            return true;
        }
        if (ratinMaze(arr, x, y + 1, n, solArr)) {
            return true;
        }
        solArr[x][y] = 0; //backtracking
        return false;
    }
    return false;
}
```

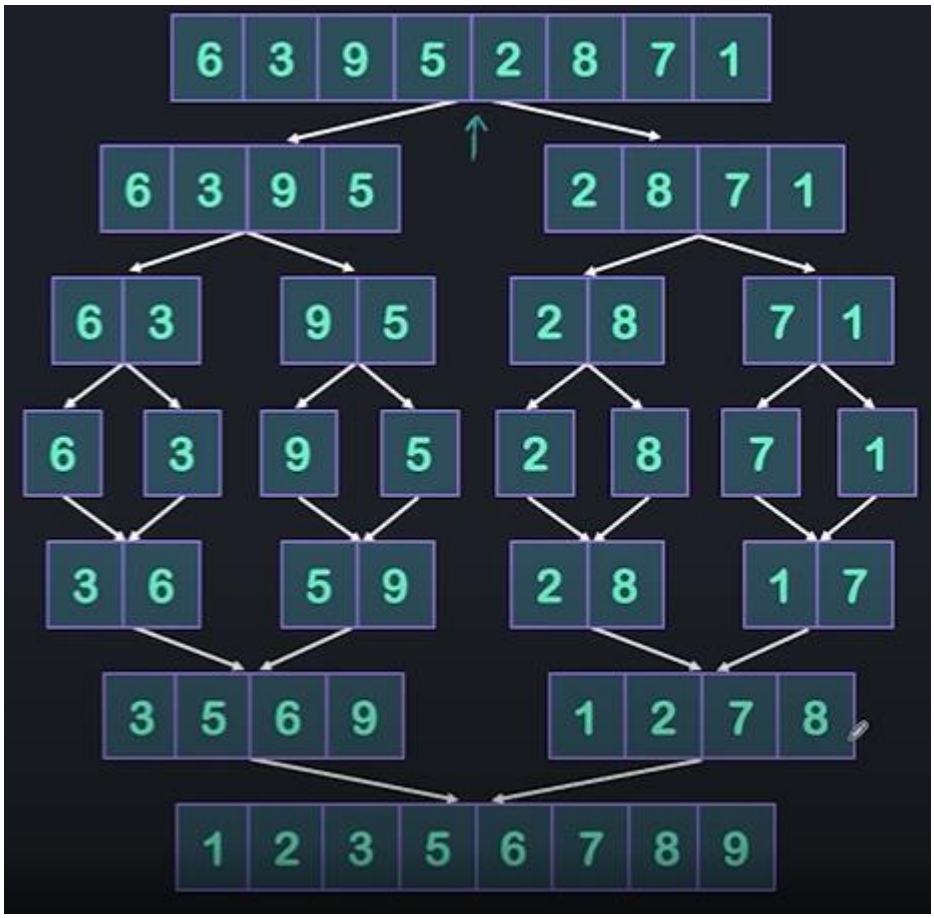
Q: kth permutation sequence

```
string getPermutation(int n, int k)
{
    int fact = 1;
    vector<int> numbers;
    for (int i = 1; i < n; i++)
    {
        fact = fact * i;
        numbers.push_back(i);
    }
    numbers.push_back(n);
    string ans = "";
    k = k - 1;
    while (true)
    {
        ans = ans + to_string(numbers[k / fact]);
        numbers.erase(numbers.begin() + k / fact);
        if (numbers.size() == 0)
        {
            break;
        }

        k = k % fact;
        fact = fact / numbers.size();
    }
    return ans;
}
```

Advanced Sorting Algorithm

➢ *Merge Sort:*

Merge Sort is a highly efficient and stable sorting algorithm that follows the "**Divide and Conquer**" strategy. It begins by dividing the unsorted list into smaller sub lists, each containing one element. This process is recursively applied until each sub list consists of only one element. Then, the algorithm starts merging these sub lists back together in a sorted manner. It compares the elements from each sub list and arranges them in the correct order. Due to its efficient use of recursion and stable sorting property, Merge Sort is widely used for sorting large datasets. However, it's important to note that Merge Sort requires additional memory for the temporary arrays used in the merging step.



```
void merge(int arr[], int l, int mid, int r) {
    int n1 = mid - l + 1;
    int n2 = r - mid;
    int a[n1];
    int b[n2]; //temp arrays
    for (int i = 0; i < n1; i++) {
        a[i] = arr[l + i];
    }
    for (int i = 0; i < n2; i++) {
        b[i] = arr[mid + 1 + i];
    }
    int i = 0;
    int j = 0;
    int k = l;
    while (i < n1 && j < n2) {
        if (a[i] < b[j]) {
            arr[k] = a[i];
            k++; i++;
        }
        else {
            arr[k] = b[j];
            k++; j++;
        }
    }
    while (i < n1) {
        arr[k] = a[i];
        k++; i++;
    }
    while (j < n2) {
        arr[k] = b[j];
        k++; j++;
    }
}
```

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

    if (l < r) {
        int mid = (l + r) / 2;
        mergeSort(arr, l, mid);
        mergeSort(arr, mid + 1, r);
        merge(arr, l, mid, r);
    }
}
```

Time complexity: O(nlogn) *Space complexity: O(n)*

Best case: O(nlogn) *Average case: O(nlogn)* *Worst case: O(nlogn)*

➢ ***Quick Sort:***

Quick Sort is a highly efficient sorting algorithm that follows the divide-and-conquer paradigm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. This process continues until the base case is reached, which is an array of one or zero elements. Quick Sort is known for its speed and is widely used in practice. However, it's important to note that its performance can degrade to O(n^2) in the worst case if the pivot selection is poorly done, but on average, it runs in O(n log n) time, making it a popular choice for sorting large datasets.

```
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // Set pivot to the last element
    int i = (low - 1);

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

```
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Time complexity: O(nlogn) *Space complexity: O(n)*

Best case: O(nlogn) *Average case: O(nlogn)* *Worst case: O(n^2)*

➢ ***Count Sort:***

Counting Sort is a simple yet highly efficient sorting algorithm that operates on a limited range of elements. It begins by identifying the maximum value within the input array to establish the range. Subsequently, a count array is created to keep track of the frequency of each element. The algorithm then iterates through the original array, incrementing the corresponding count for each element. This step provides a tally of how many times each element occurs in the array. Following this, Counting Sort reconstructs the sorted array by iterating through the count array. For each element, it adds the respective number of occurrences to the sorted array. This process ensures that the elements are arranged in ascending order.

```
void countSort(int arr[], int n) {
    int k = arr[0];
    for (int i = 0; i < n; i++) {
        k = max(k, arr[i]);
    }
    int count[k] = {0};
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }
    for (int i = 1; i <= k; i++) {
        count[i] += count[i - 1];
    }
    int output[n];
    for (int i = n - 1; i >= 0; i--) {
        output[--count[arr[i]]] = arr[i];
    }
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}
```

Time complexity: O(n) *Space complexity: O(n)*

➢ ***DNF sort:***

DNF (Dutch National Flag) Sort is an efficient sorting algorithm designed to sort an array with three distinct elements. It was named after the Dutch national flag's three colors: red, white, and blue, which symbolize the three distinct elements in the array. This algorithm efficiently partitions the array into three sections: one for elements less than a given value, one for elements equal to it, and one for elements greater than it.

```
void dnfSort(int arr[], int n) {
    int low = 0;
    int mid = 0;
    int high = n - 1;
    while (mid <= high) {
        if (arr[mid] == 0) {
            swap(arr[low], arr[mid]);
            low++; mid++;
        }
        else if (arr[mid] == 1) {
            mid++;
        }
        else {
            swap(arr[mid], arr[high]);
            high--;
        }
    }
}
```

Time complexity: O(n) *Space complexity: O(1)*

➢ ***Wave Sort:***

Wave Sort is a relatively straightforward yet effective sorting algorithm that arranges an array into a "wave-like" pattern. In a wave, each element is greater than or equal to its adjacent elements. The algorithm works by iterating through the array and swapping adjacent elements in pairs to create this wave pattern.

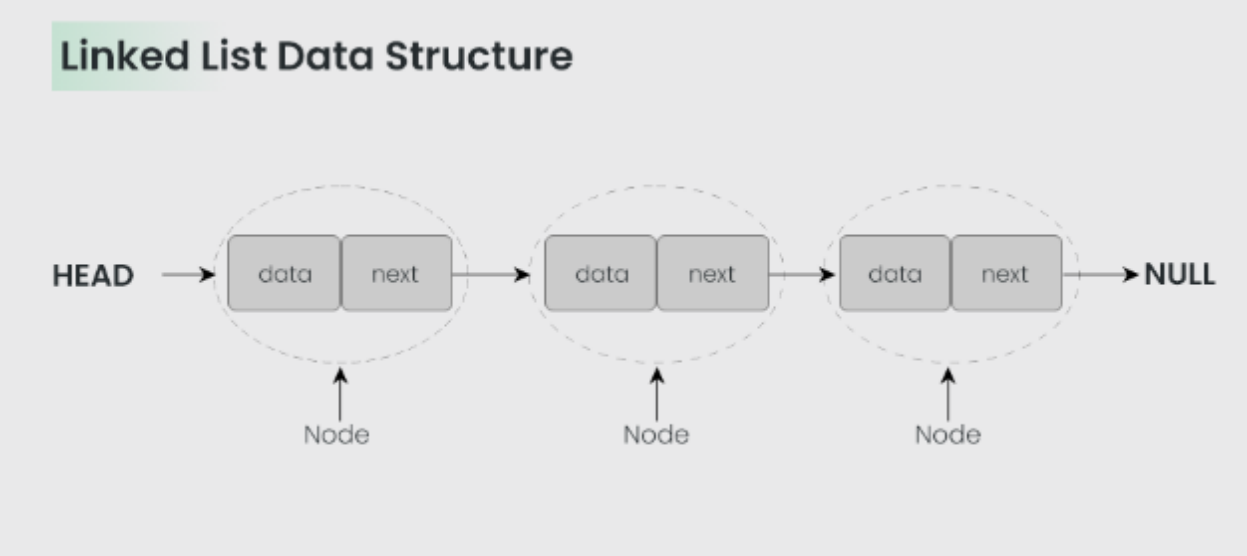
```
arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= .....
```

```
void wavesort(int arr[], int n) {
    for (int i = 1; i < n; i += 2) {
        if (arr[i] > arr[i - 1]) {
            swap(arr[i], arr[i - 1]);
        }
        if (arr[i] > arr[i + 1] && i <= n - 2) {
            swap(arr[i], arr[i + 1]);
        }
    }
}
```

Time complexity: O(n) *Space complexity: O(1)*

LinkedList

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Why linked list data structure needed?

Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needs to be updated.

Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.

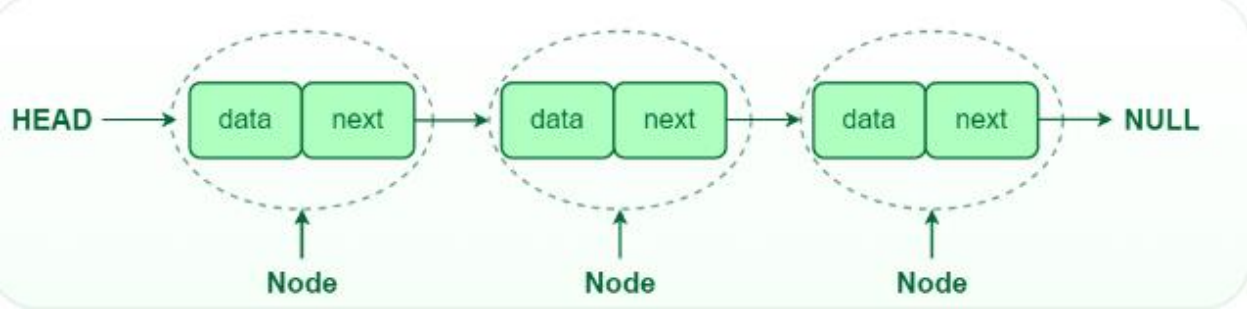
Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Types of LinkedList:

- Singly LinkedList
- Doubly LinkedList
- Circular LinkedList

Singly LinkedList:

In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.

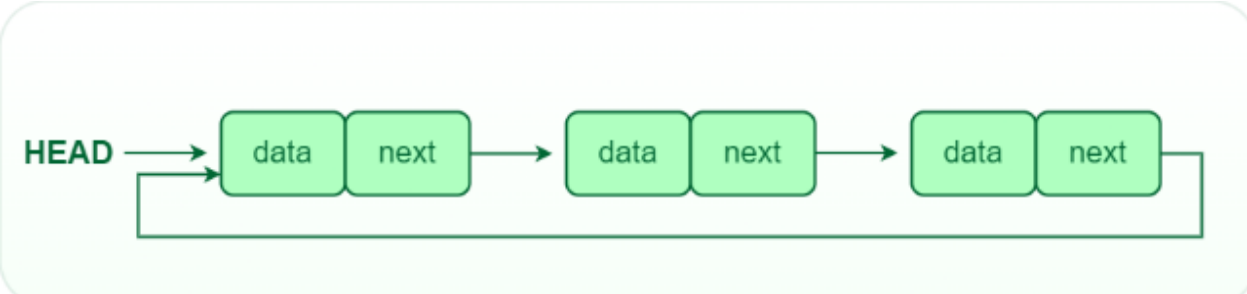


Doubly LinkedList:

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.

Circular LinkedList:

In a circular linked list, the last node points back to the head node, creating a circular structure. It can be either singly or doubly linked.



Advantages of LinkedList:

- Dynamic Size: Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
- Insertion and Deletion: Adding or removing elements from a linked list is efficient, especially for large lists.
- Flexibility: Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of LinkedList:

- Random Access: Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
- Extra Memory: Linked lists require additional memory for storing the pointers, compared to arrays.

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Basic operations:

- Insertion
- Deletion
- Searching

Singly LinkedList:

```
struct Node
{
    int data;
    Node *next;
    Node(int val)
    {
        data = val;
        next = NULL;
    }
};
void insertAtTail(Node *&root, int val)
{
    Node *n = new Node(val);
    if (root == NULL)
    {
        root = n;
        return;
    }
    Node *temp = root;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = n;
}
void insertAtHead(Node *&root, int val)
{
    Node *n = new Node(val);
    n->next = root;
    root = n;
}
void insertAtPos(Node *root, int val, int pos)
```

```
{
    Node *n = new Node(val);
    Node *temp = root;
    int idx = 1;
    while (temp->next != NULL && idx != pos - 1)
    {
        temp = temp->next;
        idx++;
    }
    n->next = temp->next;
    temp->next = n;
}
void deleteAtTail(Node *root)
{
    Node *temp = root;
    while (temp->next->next != NULL)
    {
        temp = temp->next;
    }
    Node *del = temp->next;
    temp->next = NULL;
    delete del;
}
void deleteAtHead(Node *&root)
{
    Node *del = root;
    root = root->next;
    delete del;
}
void deleteAtPos(Node *root, int pos)
{
    Node *temp = root;
    int idx = 1;
    while (temp->next != NULL && idx != pos - 1)
    {
        temp = temp->next;
        idx++;
    }
    Node *del = temp->next;
    temp->next = del->next;
    delete del;
}
void display(Node *root)
{
    Node *temp = root;
```

```
    if (temp == NULL)
    {
        cout << "NULL";
        return;
    }
    while (temp != NULL)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
}
int size(Node *root)
{
    Node *temp = root;
    int a = 0;
    while (temp != NULL)
    {
        a++;
        temp = temp->next;
    }
    return a;
}
void search(Node *root, int key)
{
    Node *temp = root;
    while (temp != NULL)
    {
        if (temp->data == key)
        {
            cout << "Element found!" << endl;
            return;
        }
        temp = temp->next;
    }
    cout << "Element not found!" << endl;
}
```

Reversing a LinkedList:

Iterative Approach:

```
void reverse(Node *&root)
```

```
{
    Node *prev = NULL;
    Node *curr = root;
    Node *next;
    while (curr != NULL)
    {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    root = prev;
}
```

Recursive Approach:

```
Node *Recursivereverse(Node *&root)
{
    if (root == NULL || root->next == NULL)
        return root;

    Node *node1 = Recursivereverse(root->next);
    root->next->next = root;
    root->next = NULL;
    return node1;
}
```

Detection and removal of the cycle in a LinkedList:

```
void makeCycle(Node *root, int k)
{
    Node *temp = root;
    int c = 1;
    Node *cycle;
    while (temp->next != NULL)
    {
        if (c == k)
        {
```

```
            cycle = temp;
        }
        temp = temp->next;
        c++;
    }
    temp->next = cycle;
}
```

Floyd's Algorithm for detection of cycle in a list:

```
bool detectCycle(Node *root)
{
    Node *fast = root;
    Node *slow = root;
    while (fast != NULL && fast->next != NULL)
    {
        fast = fast->next->next;
        slow = slow->next;
        if (fast == slow)
        {
            return true;
        }
    }
    return false;
}
```

Floyd's Algorithm for Removing cycle in a list:

```
void removeCycle(Node *root)
{
    Node *fast = root;
    Node *slow = root;
    while (fast != NULL && fast->next != NULL)
    {
        fast = fast->next->next;
        slow = slow->next;
        if (fast == slow)
        {
            fast = root;
            break;
        }
    }
```

```
    while (slow->next != fast->next)
    {
        slow = slow->next;
        fast = fast->next;
    }
    slow->next = NULL;
}
```

Reverse K nodes in LinkedList:

```
node* reversek(node* &head, int k) {
    node* prevptr = NULL;
    node* currptr = head;
    node* nextptr;
    int count = 0;
    while (currptr != NULL && count < k) {
        nextptr = currptr->next;
        currptr->next = prevptr;
        prevptr = currptr;
        currptr = nextptr;
        count++;
    }

    if (nextptr != NULL) {
        head->next = reversek(nextptr, k);
    }
    return prevptr;
}
```

Point of intersection of 2 Linked Lists:

```
void intersectLists(Node *root, Node *root1, int pos)
{
    Node *tail = root1;
    while (tail->next != NULL)
    {
        tail = tail->next;
    }
    Node *temp = root;
    while (pos != 1)
    {
        temp = temp->next;
        pos--;
    }
    tail->next = temp;
}

void pointOfIntersection(Node *root, Node *root1)
{
    int l1 = size(root);
    int l2 = size(root1);
    int diff = l1 - l2;
    Node *temp;
    Node *temp1;
    if (diff > 0)
    {
        temp = root;
        temp1 = root1;
    }
    else
    {

        temp = root1;
        temp1 = root;
        diff = abs(diff);
    }
    while (temp != NULL && diff != 0)
    {
        temp = temp->next;
        diff--;
    }
    while (temp != NULL && temp1 != NULL)
```

```
    {
        if (temp == temp1)
        {
            cout << temp->data;
            return;
        }
        temp = temp->next;
        temp1 = temp1->next;
    }
}
```

Merge 2 sorted Linked Lists:

```
Node *Merge2SortedLists(Node *root, Node *root1)
{
    Node *dummyNode;
    Node *rNode = dummyNode;
    Node *temp = root;
    Node *temp1 = root1;
    while (temp != NULL && temp1 != NULL)
    {
        if (temp->data > temp1->data)
        {
            dummyNode->next = temp1;
            dummyNode = dummyNode->next;
            temp1 = temp1->next;
        }
        else
        {
            dummyNode->next = temp;
            dummyNode = dummyNode->next;
            temp = temp->next;
        }
    }
    while (temp != NULL)
    {
        dummyNode->next = temp;
        dummyNode = dummyNode->next;
        temp = temp->next;
    }
    while (temp1 != NULL)
```

```
    {
        dummyNode->next = temp1;
        dummyNode = dummyNode->next;
        temp1 = temp1->next;
    }
    return rNode->next;
}
```

Put even position nodes after odd position nodes:

```
void evenAfterOdd(node* &head) {
    node* odd = head;
    node* even = head->next;
    node* evenStart = even;
    while (odd->next != NULL && even->next != NULL) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }
    odd->next = evenStart;
    if (odd->next != NULL) {
        even->next = NULL;
    }
}
```

Sort Linked list of 0's, 1's, 2's:

Doubly LinkedList:

```
void insertAtTail(Node *&root, int val)
{
    Node *n = new Node(val);
    if (root == NULL)
    {
        root = n;
        return;
    }
    Node *temp = root;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = n;
    n->prev = temp;
}
void insertAtHead(Node *&root, int val)
{
    Node *n = new Node(val);
    n->next = root;
    root->prev = n;
    root = n;
}
void insertAtPos(Node *root, int val, int pos)
{
    Node *n = new Node(val);
    int idx = 1;
    Node *temp = root;
    while (temp->next != NULL && idx != pos)
    {
        temp = temp->next;
        idx++;
    }
    n->prev = temp->prev;
    temp->prev->next = n;
    n->next = temp;
    temp->prev = n;
}
void deleteAtTail(Node *root)
{
```

```
    Node *temp = root;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    Node *del = temp;
    temp->prev->next = NULL;
    delete del;
}
void deleteAtHead(Node *&root)
{
    Node *del = root;
    root->next->prev = NULL;
    root = del->next;
    delete del;
}
void deleteAtPos(Node *root, int pos)
{
    Node *temp = root;
    int idx = 1;
    while (idx != pos)
    {
        temp = temp->next;
        idx++;
    }
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
}
bool search(Node *root, int key)
{
    Node *temp = root;
    while (temp != NULL)
    {
        if (temp->data == key)
            return true;
        temp = temp->next;
    }
    return false;
}

void reverse(Node *&root)
{
    Node *prev = NULL;
    Node *curr = root;
    Node *nex;
```

```
    while (curr != NULL)
    {
        nex = curr->next;
        curr->next = prev;
        curr->prev = nex;
        prev = curr;
        curr = nex;
    }
    root = prev;
}

void size(Node *root)
{
    Node *temp = root;
    int a = 0;
    while (temp != NULL)
    {
        a++;
        temp = temp->next;
    }
    cout << a << endl;
}
void display(Node *root)
{
    Node *temp = root;
    while (temp != NULL)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
}
```

Circular LinkedList:

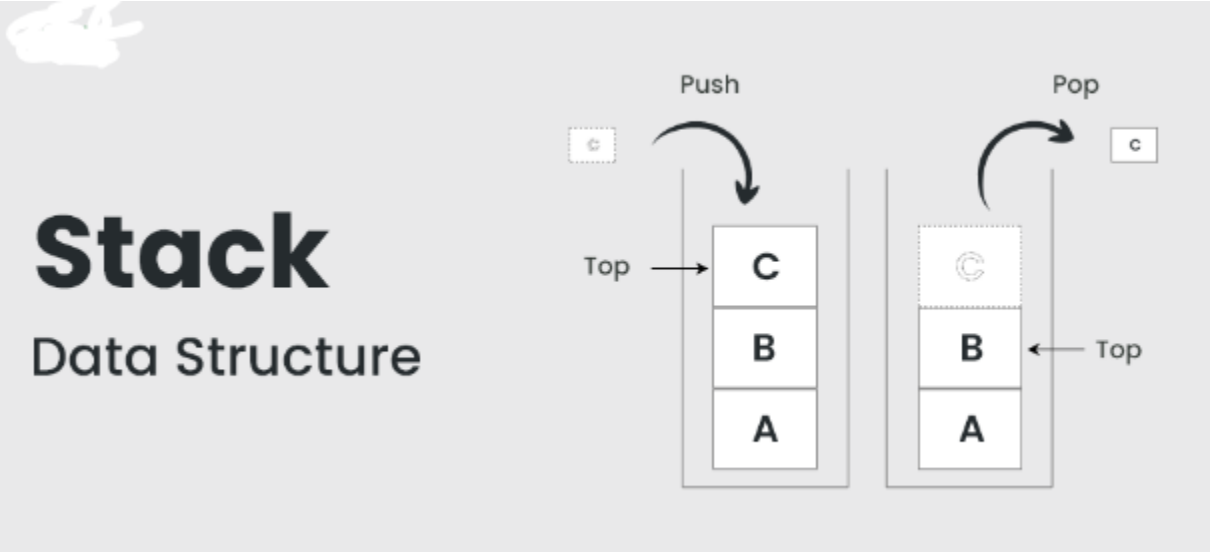
```
void insertAtTail(Node *&head, int val)
{
    Node *n = new Node(val);
    Node *temp = head;
    if (temp == NULL)
    {
        head = n;
```

```
        return;
    }
    while (temp->next != NULL)
    {
        if (temp->next->next == head)
        {
            temp->next->next = n;
            n->next = head;
            return;
        }
        temp = temp->next;
    }
    temp->next = n;
    n->next = head;
}
void insertAtHead(Node *&head, int val)
{
    Node *n = new Node(val);
    Node *temp = head;
    n->next = head;
    while (temp->next->next != head)
    {
        temp = temp->next;
    }
    temp->next->next = n;
    head = n;
}
void insertAtPos(Node *&head, int val, int pos)
{
    Node *n = new Node(val);
    Node *temp = head;
    Node *temp1;
    int count = 0;
    while (temp->next != head && count != pos)
    {
        temp1 = temp;
        temp = temp->next;
        count++;
    }
    temp1->next = n;
    n->next = temp;
}
void deleteAtTail(Node *head)
{
    Node *temp = head;
```

```
    while (temp->next->next != head)
    {
        temp = temp->next;
    }
    Node *todel = temp->next;
    temp->next = head;
    delete todel;
}
void deleteAtHead(Node *&head)
{
    Node *temp = head;
    Node *todel = temp;
    Node *temp1 = temp->next;
    while (temp->next != head)
    {
        temp = temp->next;
    }
    temp->next = temp1;
    head = temp1;
    delete todel;
}
void deleteAtPos(Node *head, int pos)
{
    Node *temp = head;
    Node *temp1;
    int count = 0;
    while (temp->next != head && count != pos)
    {
        temp1 = temp;
        temp = temp->next;
        count++;
    }
    Node *todel = temp;
    temp1->next = temp->next;
    delete todel;
}
```

Stack

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



Basic operation on Stack:

push() to insert an element into the stack

pop() to remove an element from the stack

top() Returns the top element of the stack.

isEmpty() returns true if the stack is empty else false.

size() returns the size of the stack.

Types of Stack:

- ***Fixed Size Stack:*** As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs. This type of stack is implemented using an array data structure.
- ***Dynamic Size Stack:*** A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Applications of Stack:

Function Calls and Recursion:

Stacks are used in programming languages to manage function calls. When a function is called, its parameters and return address are pushed onto the call stack. When the function completes, it pops these values off the stack to resume execution.

Expression Evaluation:

Stacks are used to evaluate arithmetic expressions, including infix, postfix, and prefix notations. They help maintain the correct order of operations.

Balanced Parentheses and Brackets:

Stacks are employed to check for balanced parentheses, brackets, and braces in expressions. This is crucial in compilers and interpreters for ensuring syntactic correctness.

Undo Operations in Text Editors:

Stacks can be used to implement the "Undo" functionality in text editors. Each action (like typing a character) is pushed onto a stack, allowing for easy reversal of operations.

Backtracking Algorithms:

Algorithms like Depth-First Search (DFS) in graph traversal and recursive algorithms often use stacks to keep track of the state of the program.

Browser History:

Stacks can be used to implement forward and backward navigation in web browsers. Each visited page is pushed onto a stack, allowing the user to go back and forth.

Array implementation of Stack:

```
class Stack
{
public:
    int size = 0;
    int *arr = new int[5];
    void push(int val)
    {
```

```
        if (size != 5)
            arr[size++] = val;
        else
            cout << "Stack overflow: " << endl;
    }
    void pop()
    {
        if (size != 0)
            --size;
        else
            cout << "Stack underflow: " << endl;
    }
    int top()
    {
        if (size != 0)
            return arr[size - 1];
        else
            cout << "No element in stack" << endl;
    }
    int peek(int idx)
    {
        if (idx <= size)
            return arr[idx];

        cout << "Not found!" << endl;
        return -1;
    }
    int getSize()
    {
        return size;
    }
    bool isEmpty()
    {
        return size == 0;
    }
    bool isFull()
    {
        return size == 5;
    }
    bool search(int val)
    {
        for (int i = 0; i < size; i++)
        {
            if (arr[i] == val)
            {
```

```
                return true;
            }
            else
                return false;
        }
    }
}
```

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of array implementation:

- It is not dynamic i.e., it doesn't grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
- The total size of the stack must be defined beforehand.

LinkedList implementation of Stack:

```
struct Node
{
    int data;
    Node *next;
    Node(int val)
    {
        data = val;
        next = NULL;
    }
};
void push(Node *&head, int val)
{
    Node *n = new Node(val);
```

```
    Node *temp = head;
    if (temp == NULL)
    {
        head = n;
        return;
    }

    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = n;
}
void pop(Node *&head)
{
    Node *temp = head;
    if (temp == NULL)
    {
        cout << "No element to pop" << endl;
        return;
    }
    if (temp->next == NULL)
    {
        head = NULL;
        return;
    }
    while (temp->next->next != NULL)
    {
        temp = temp->next;
    }
    Node *todel = temp->next;
    temp->next = NULL;
    delete todel;
}
void Top(Node *head)
{
    Node *temp = head;
    if (temp == NULL)
    {
        cout << "No element to top" << endl;
        return;
    }
    while (temp->next != NULL)
    {
        temp = temp->next;
```

```
    }
    cout << temp->data << endl;
}
bool isEmpty(Node *head)
{
    return head == NULL;
}
```

Advantages of Linked List implementation:

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- It is used in many virtual machines like JVM.

Disadvantages of Linked List implementation:

- Requires extra memory due to the involvement of pointers.
- Random accessing is not possible in stack.

Prefix Evaluation:

```
int evaluatePrefixExpression(string expr)
{
    stack<int> operands;

    for (int i = expr.size() - 1; i >= 0; i--)
    {
        if (isdigit(expr[i]))
        {
            operands.push(expr[i] - '0'); // Convert character to integer
        }
        else
        {
            int operand1 = operands.top();
            operands.pop();
            int operand2 = operands.top();
            operands.pop();
```

```
            switch (expr[i])
            {
            case '+':
                operands.push(operand1 + operand2);
                break;
            case '-':
                operands.push(operand1 - operand2);
                break;
            case '*':
                operands.push(operand1 * operand2);
                break;
            case '/':
                operands.push(operand1 / operand2);
                break;
            }
        }
    }

    return operands.top();
}
```

Postfix Evaluation:

```
int evaluatePostfixExpression(string expr)
{
    stack<int> operands;

    for (char c : expr)
    {
        if (isdigit(c))
        {
            operands.push(c - '0'); // Convert character to integer
        }
        else
        {
            int operand2 = operands.top();
            operands.pop();
            int operand1 = operands.top();
            operands.pop();
```

```
            switch (c)
            {
            case '+':
                operands.push(operand1 + operand2);
                break;
            case '-':
                operands.push(operand1 - operand2);
                break;
            case '*':
                operands.push(operand1 * operand2);
                break;
            case '/':
                operands.push(operand1 / operand2);
                break;
            }
        }
    }

    return operands.top();
}
```

Infix to Postfix:

```
int precedence(char op)
{
    if (op == '^')
        return 3;
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}

string infixToPostfix(string infix)
{
    stack<char> operators;
    string postfix = "";
```

```
    for (char c : infix)
    {
        if (isalnum(c))
        {
            postfix += c; // If it's an operand, add to the output
        }
        else if (c == '(')
        {
            operators.push(c); // Push opening bracket onto the stack
        }
        else if (c == ')')
        {
            while (!operators.empty() && operators.top() != '(')
            {
                postfix += operators.top(); // Pop operators until '(' is
encountered
                operators.pop();
            }
            operators.pop(); // Pop '('
        }
        else
        {
            while (!operators.empty() && precedence(c) <=
precedence(operators.top()))
            {
                postfix += operators.top(); // Pop operators with higher or equal
precedence
                operators.pop();
            }
            operators.push(c); // Push the current operator
        }
    }

    while (!operators.empty())
    {
        postfix += operators.top(); // Pop any remaining operators
        operators.pop();
    }

    return postfix;
}
```

Infix to Prefix:

```
int precedence(char op)
{
    if(op==’^’)
        return 3;
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}

string infixToPrefix(string infix)
{
    stack<char> operators;
    string postfix = "";

    // Reverse the infix expression
    for (int i = 0; i < infix.length(); i++)
    {
        if (infix[i] == '(')
            infix[i] = ')';
        else if (infix[i] == ')')
            infix[i] = '(';
    }
    reverse(infix.begin(), infix.end());

    for (char c : infix)
    {
        if (isalnum(c))
        {
            postfix += c; // If it's an operand, add to the output
        }
        else if (c == '(')
        {
            operators.push(c); // Push opening bracket onto the stack
        }
        else if (c == ')')
        {
            while (!operators.empty() && operators.top() != '(')
            {
```

```
                postfix += operators.top(); // Pop operators until '(' is
encountered
                operators.pop();
            }
            operators.pop(); // Pop '('
        }
        else
        {
            while (!operators.empty() && precedence(c) <=
precedence(operators.top()))
            {
                postfix += operators.top(); // Pop operators with higher or equal
precedence
                operators.pop();
            }
            operators.push(c); // Push the current operator
        }
    }

    while (!operators.empty())
    {
        postfix += operators.top(); // Pop any remaining operators
        operators.pop();
    }

    // Reverse the postfix expression to get prefix
    reverse(postfix.begin(), postfix.end());

    return postfix;
}
```

Prefix to Infix:

```
bool isOperator(char x)
{
    switch (x)
    {
    case '+':
    case '-':
    case '/':
```

```
    case '*':
    case '^':
    case '%':
        return true;
    }
    return false;
}

// Convert prefix to Infix expression
string preToInfix(string pre_exp)
{
    stack<string> s;

    // length of expression
    int length = pre_exp.size();

    // reading from right to left
    for (int i = length - 1; i >= 0; i--)
    {

        // check if symbol is operator
        if (isOperator(pre_exp[i]))
        {

            // pop two operands from stack
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();

            // concat the operands and operator
            string temp = "(" + op1 + pre_exp[i] + op2 + ")";

            // Push string temp back to stack
            s.push(temp);
        }

        // if symbol is an operand
        else
        {

            // push the operand to the stack
            s.push(string(1, pre_exp[i]));
        }
    }
```

```
    // Stack now contains the Infix expression
    return s.top();
}
```

Postfix to Infix:

```
bool isOperator(char x)
{
    switch (x)
    {
    case '+':
    case '-':
    case '/':
    case '*':
    case '^':
    case '%':
        return true;
    }
    return false;
}

// Convert postfix to Infix expression
string postToInfix(string post_exp)
{
    stack<string> s;

    // length of expression
    int length = post_exp.size();

    // reading from left to right
    for (int i = 0; i < length; i++)
    {
        // check if symbol is operator
        if (isOperator(post_exp[i]))
        {
            // pop two operands from stack
            string op2 = s.top();
            s.pop();
```

```
            string op1 = s.top();
            s.pop();

            // concat the operands and operator
            string temp = "(" + op2 + post_exp[i] + op1 + ")";

            // Push string temp back to stack
            s.push(temp);
        }
        else
        {
            // if symbol is an operand
            // push the operand to the stack
            s.push(string(1, post_exp[i]));
        }
    }

    // Stack now contains the Infix expression
    return s.top();
}
```

Prefix to Postfix:

```
bool isOperator(char x)
{
    switch (x)
    {
    case '+':
    case '-':
    case '/':
    case '*':
        return true;
    }
    return false;
}

// Convert prefix to Postfix expression
string preToPost(string pre_exp)
```

```
{

    stack<string> s;
    // Length of expression
    int length = pre_exp.size();

    // reading from right to left
    for (int i = length - 1; i >= 0; i--)
    {
        // check if symbol is operator
        if (isOperator(pre_exp[i]))
        {
            // pop two operands from stack
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();

            // concat the operands and operator
            string temp = op1 + op2 + pre_exp[i];

            // Push string temp back to stack
            s.push(temp);
        }

        // if symbol is an operand
        else
        {

            // push the operand to the stack
            s.push(string(1, pre_exp[i]));
        }
    }

    // stack contains only the Postfix expression
    return s.top();
}
```

Postfix to Prefix:

```
bool isOperator(char x)
{
    switch (x)
    {
    case '+':
    case '-':
    case '/':
    case '*':
        return true;
    }
    return false;
}

// Convert postfix to Prefix expression
string postToPre(string post_exp)
{
    stack<string> s;

    // length of expression
    int length = post_exp.size();

    // reading from left to right
    for (int i = 0; i < length; i++)
    {

        // check if symbol is operator
        if (isOperator(post_exp[i]))
        {

            // pop two operands from stack
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();

            // concat the operands and operator
            string temp = post_exp[i] + op2 + op1;

            // Push string temp back to stack
            s.push(temp);
        }

        // if symbol is an operand
        else
        {
```

```
            // push the operand to the stack
            s.push(string(1, post_exp[i]));
        }
    }

    string ans = "";
    while (!s.empty())
    {
        ans += s.top();
        s.pop();
    }
    return ans;
}
```

Reverse a stack using Recursion:

```
void insertAtBottom(stack<int>& s, int item) {
    if (s.empty()) {
        s.push(item);
        return;
    }
    int temp = s.top();
    s.pop();
    insertAtBottom(s, item)
    s.push(temp);
}

void reverseStackRecursively(stack<int>& s) {
    if (!s.empty()) {
        int temp = s.top();
        s.pop();
        reverseStackRecursively(s);
        insertAtBottom(s, temp);
    }
}
```

Balanced Parenthesis:

```
bool areParenthesesBalanced(string expr)
{
    stack<char> s;
    char x;

    for (char i : expr)
    {
        if (i == '(' || i == '{' || i == '[')
        {
            s.push(i);
            continue;
        }

        if (s.empty())
            return false;

        switch (i)
        {
        case ')':
            x = s.top();
            s.pop();
            if (x == '{' || x == '[')
                return false;
            break;

        case '}':
            x = s.top();
            s.pop();
            if (x == '(' || x == '[')
                return false;
            break;

        case ']':
            x = s.top();
            s.pop();
            if (x == '(' || x == '{')
                return false;
            break;
        }
    }

    return (s.empty());
}
```

Monotonic Stack:

A monotonic stack is a stack whose elements are monotonically increasing or decreasing. It contains all qualities that a typical stack has and its elements are all monotonic decreasing or increasing.

Types of Monotonic Stack:

There are 2 types of monotonic stacks:

- Monotonic Increasing Stack
- Monotonic Decreasing Stack

Monotonic Increasing Stack:

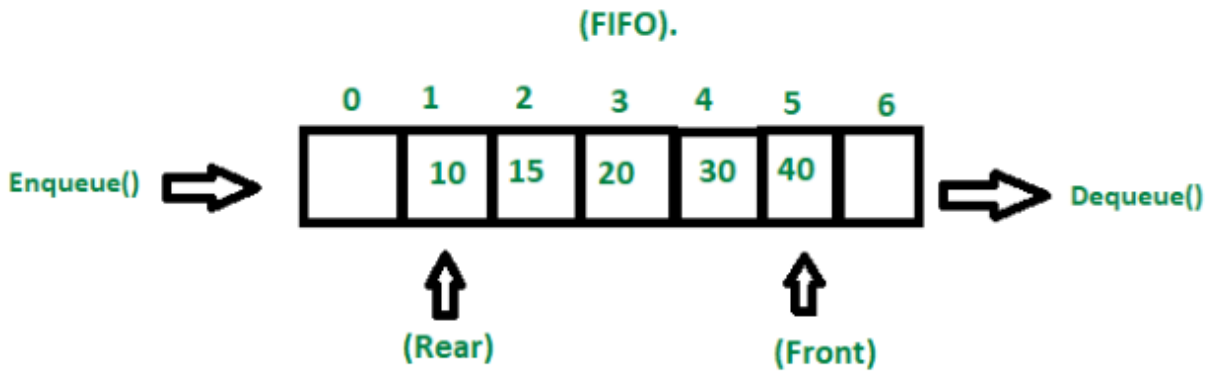
It is a stack in which the elements are in increasing order from the bottom to the top of the stack.

Monotonic Decreasing Stack:

A stack is monotonically decreasing if It's elements are in decreasing order from the bottom to the top of the stack.

Queue

A queue is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order. We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.



Basic Operations for Queue in Data Structure:

Some of the basic operations for Queue in Data Structure are:

Enqueue() – Adds (or stores) an element to the end of the queue..

Dequeue() – Removal of elements from the queue.

Peek() or front() - Acquires the data element available at the front node of the queue without deleting it.

rear() – This operation returns the element at the rear end without removing it.

isFull() – Validates if the queue is full.

isNull() – Checks if the queue is empty.

Types of Queue:

There are different types of queues:

Simple Queue: *Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.*

Circular Queue*: In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.*

Priority Queue*: This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority. The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values. The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.*

Dequeue: *Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end. Because of this property it may not obey the First In First Out property.*

Array Implementation of Queue:

```
class queue
{
private:
    int front;
    int back;
    int arr[100];

public:
    queue()
    {
        front = -1;
        back = -1;
    }
    void enqueue(int val)
    {
        if (back == 100)
        {
            cout << "Queue overflow!" << endl;
            return;
        }

        arr[++back] = val;
        if (front == -1)
            ++front;
    }
    void dequeue()
    {
        if (front == -1 || front > back)
        {
            cout << "No element to dequeue" << endl;
            return;
```

```
        }
        front++;
    }
    int Front()
    {
        if (front == -1 || front > back)
        {
            cout << "No element to show!" << endl;
            return 0;
        }
        return arr[front];
    }
    int size()
    {
        return back;
    }
    bool empty()
    {
        return (front > back) || front == -1;
    }
}
```

Advantages of Array Implementation:

- Easy to implement.
- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.

Disadvantages of Array Implementation:

- Static Data Structure, fixed size.
- If the queue has a large number of enqueue and dequeue operations, at some point (in case of linear increment of front and rear indexes) we may not be able to insert elements in the queue even if the queue is empty (this problem is avoided by using circular queue).
- Maximum size of a queue must be defined prior.

LinkedList Implementation of Queue:

```
struct Node
{
    int data;
    Node *next;
    Node(int val)
    {
        data = val;
        next = NULL;
    }
};
class queue
{
private:
    Node *head;
    int count;

public:
    queue()
    {
        head = NULL;
        count = 0;
    }
    void enqueue(int val)
    {
        Node *n = new Node(val);
        if (head == NULL)
        {
            head = n;
            count++;
            return;
        }
        Node *temp = head;
        while (temp->next != NULL)
        {
            temp = temp->next;
        }
        temp->next = n;
        count++;
    }
    void dequeue()
    {
```

```
        if (head == NULL)
        {
            cout << "No element to dequeue!" << endl;
            return;
        }
        Node *temp = head;
        Node *todel = temp;
        head = temp->next;
        delete todel;
        count--;
    }
    int Front()
    {
        if (head == NULL)
        {
            cout << "No element to show!" << endl;
            return -1;
        }
        return head->data;
    }
    bool empty()
    {
        return head == NULL;
    }
    int size()
    {
        return count;
    }
}
```

Advantages of LinkedList:

- Linked lists can dynamically grow and shrink in size during program execution, unlike arrays, which have a fixed size.
- Inserting or deleting elements in a linked list can be done in constant time (O(1)) if you have a reference to the node you want to modify. This makes linked lists efficient for dynamic operations.

Disadvantages of LinkedList:

- Accessing an element in a linked list is slower compared to arrays because you have to traverse from the head to the desired element. This results in O(n) time complexity for access.
- Each node in a linked list contains not only data but also a reference (or pointer) to the next node. This consumes additional memory for these references.

Queue using Stack:

```
class Queue
{
private:
    stack<int> s1;
    stack<int> s2;

public:
    void push(int val)
    {
        s1.push(val);
    }
    void pop()
    {
        if (s2.empty() && s1.empty())
        {
            cout << "No element to pop!" << endl;
            return;
        }
        if (s2.empty() && !s1.empty())
        {
            while (!s1.empty())
            {
                s2.push(s1.top());
                s1.pop();
```

```
            }
        }

        s2.pop();
    }
    int top()
    {
        if (s2.empty() && s1.empty())
        {
            cout << "No element in queue!" << endl;
            return 0;
        }
        if (s2.empty() && !s1.empty())
        {
            while (!s1.empty())
            {
                s2.push(s1.top());
                s1.pop();
            }
        }
        return s2.top();
    }
    bool empty()
    {
        if (s2.empty() && s1.empty())
            return true;
        else if (!s1.empty() && s2.empty())
            return false;
        else if (s1.empty() && !s2.empty())
            return false;
    }
};
```

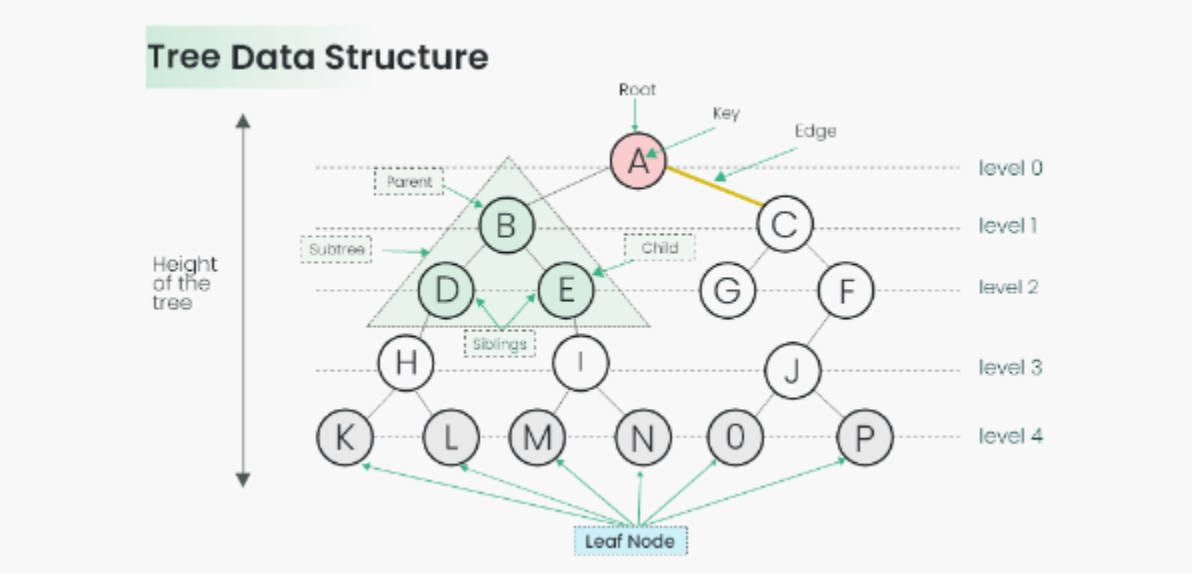
Stack using Queue:

```
class Stack
{
private:
    queue<int> q1;
```

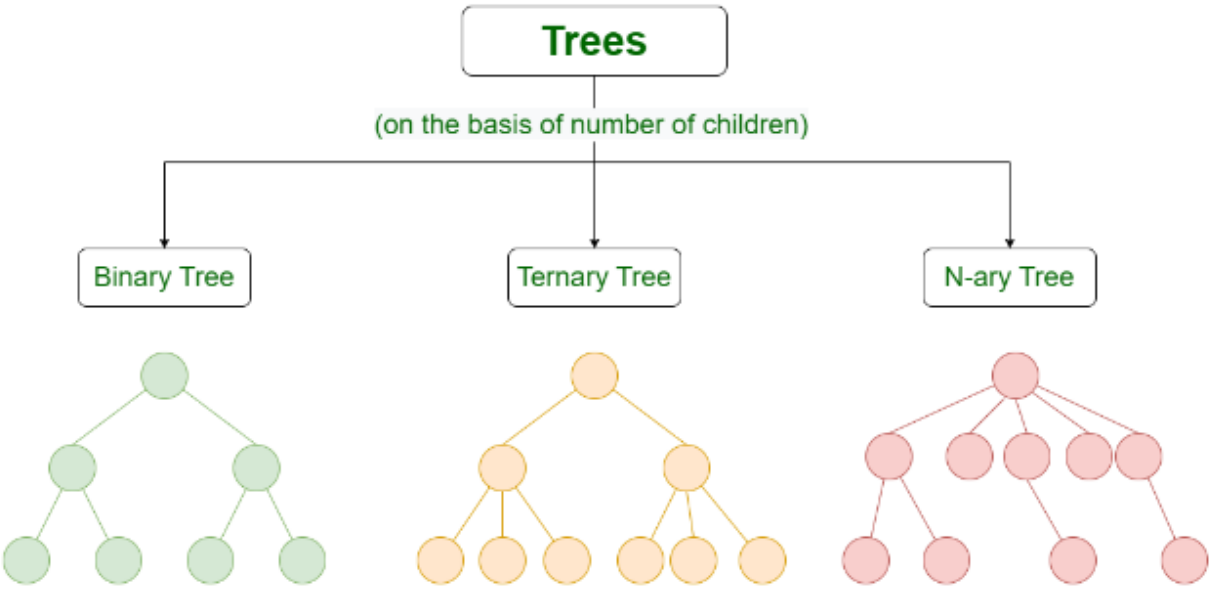
```
    queue<int> q2;

public:
    void push(int val)
    {
        q2.push(val);
        while (!q1.empty())
        {
            q2.push(q1.front());
            q1.pop();
        }
        q1 = q2;
        while (!q2.empty())
        {
            q2.pop();
        }
    }
    int Top()
    {
        if (q1.empty())
        {
            cout << "No element!" << endl;
            return 0;
        }
        return q1.front();
    }
    void pop()
    {
        if (q1.empty())
        {
            cout << "No element to pop!" << endl;
            return;
        }
        q1.pop();
    }
    bool empty()
    {
        return q1.empty();
    }
};
```

Tree

Tree Data Structure is a hierarchical data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes.



Types of Trees in Data Structure based on the number of children:



Binary Tree:

A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Ternary Tree:

A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.

N-ary Tree:

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

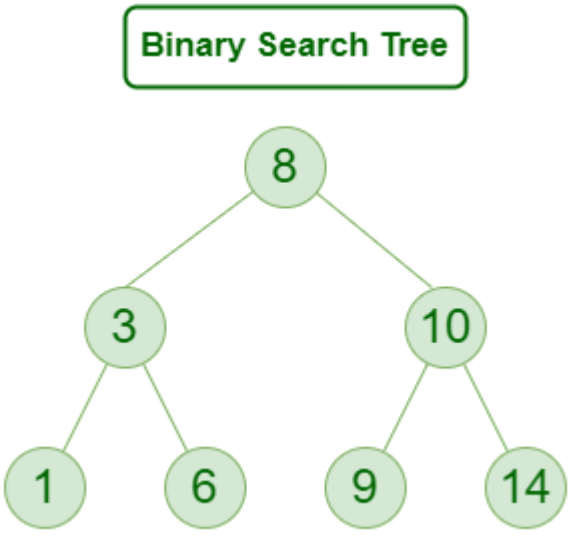
Binary Search Tree:

A binary Search Tree is a node-based binary tree data structure that has the following properties:

The left subtree of a node contains only nodes with keys lesser than the node’s key.

The right subtree of a node contains only nodes with keys greater than the node’s key.

The left and right subtree each must also be a binary search tree.



Applications:

- Databases use tree data structure for indexing.
- Tree data structure is used in file directory management.
- DNS uses tree data structure.
- Trees are used in several games like moves in chess.
- Decision-based algorithms in machine learning uses tree algorithms.

Advantages:

Efficient searching: Trees are particularly efficient for searching and retrieving data. The time complexity of searching in a tree is typically O(log n), which means that it is very fast even for very large data sets.

Flexible size: Trees can grow or shrink dynamically depending on the number of nodes that are added or removed. This makes them particularly useful for applications where the data size may change over time.

Easy to traverse: Traversing a tree is a simple operation, and it can be done in several different ways depending on the requirements of the application. This makes it easy to retrieve and process data from a tree structure.

Disadvantages:

Memory overhead: Trees can require a significant amount of memory to store, especially if they are very large. This can be a problem for applications that have limited memory resources.

Imbalanced trees: If a tree is not balanced, it can result in uneven search times. This can be a problem in applications where speed is critical.

Complexity: Trees can be complex data structures, and they can be difficult to understand and implement correctly. This can be a problem for developers who are not familiar with them.

Traversals in Binary Tree:

- ***Preorder Traversal:***

```
void preorder(Node *root)
{
    if (root != NULL)
    {
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
    else
    {
        return;
    }
}
```

```
void iterativePreorder(TreeNode *root)
{
    if (root == NULL)
        return;

    stack<TreeNode *> nodeStack;
    nodeStack.push(root);

    while (!nodeStack.empty())
```

```
    {
        TreeNode *node = nodeStack.top();
        cout << node->val << " ";
        nodeStack.pop();

        if (node->right)
            nodeStack.push(node->right);

        if (node->left)
            nodeStack.push(node->left);
    }
}
```

- ***In order Traversal:***

```
void inorder(Node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
    else
    {
        return;
    }
}
```

```
vector<int> inorderTraversal(TreeNode *root)
{
    vector<int> ans;
    stack<TreeNode *> s;
    TreeNode *node = root;
    while (true)
    {
        if (node != NULL)
        {
```

```
            s.push(node);
            node = node->left;
        }
        else
        {
            if (s.empty() == true)
                break;

            node = s.top();
            s.pop();
            ans.push_back(node->val);
            node = node->right;
        }
    }
    return ans;
}
```

- ***Post Order Traversal:***

```
void postorder(Node *root)
{
    if (root != NULL)
    {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << " ";
    }
    else
    {
        return;
    }
}
```

```
vector<int> postorderTraversal(TreeNode *root)
{
    if (root == NULL)
        return ans;
    stack<TreeNode *> s1;
    stack<TreeNode *> s2;
    s1.push(root);
```

```
    while (!s1.empty())
    {
        root = s1.top();
        s1.pop();
        s2.push(root);

        if (root->left != NULL)
            s1.push(root->left);
        if (root->right != NULL)
            s1.push(root->right);
    }

    while (!s2.empty())
    {
        ans.push_back(s2.top()->val);
        s2.pop();
    }

    return ans;
}
```

- ***Level Order Traversal:***

```
void levelorder(Node *root)
{
    if (root == NULL)
    {
        return;
    }
    queue<Node *> q;
    q.push(root);
    q.push(NULL);
    while (!q.empty())
    {
        Node *n = q.front();
        q.pop();
        if (n != NULL)
        {
            cout << n->data << " ";
            if (n->left)
            {
```

```
                q.push(n->left);
            }
            if (n->right)
            {
                q.push(n->right);
            }
        }
        else if (!q.empty())
        {
            q.push(NULL);
        }
    }
}
```

Built tree from preorder and inorder:

```
int search(int inorder[], int start, int end, int curr)
{
    for (int i = start; i <= end; i++)
    {
        if (inorder[i] == curr)
            return i;
    }
    return -1;
}

Node *builtTree(int preorder[], int inorder[], int start, int end)
{
    static int idx = 0;
    if (start > end)
    {
        return NULL;
    }
    int curr = preorder[idx];
    idx++;
    Node *node = new Node(curr);
    if (start == end)
    {
        return node;
```

```
    }
    int pos = search(inorder, start, end, curr);
    node->left = builtTree(preorder, inorder, start, pos - 1);
    node->right = builtTree(preorder, inorder, pos + 1, end);

    return node;
}
```

Built tree from post order and in order:

```
int search(int inorder[], int start, int end, int curr)
{
    for (int i = start; i <= end; i++)
    {
        if (inorder[i] == curr)
            return i;
    }
    return -1;
}

Node *builtTree(int postorder[], int inorder[], int start, int end)
{
    static int idx = 4;
    if (start > end)
    {
        return NULL;
    }
    int curr = postorder[idx];
    idx--;
    Node *node = new Node(curr);
    if (start == end)
    {
        return node;
    }
    int pos = search(inorder, start, end, curr);
    node->right = builtTree(postorder, inorder, pos + 1, end);
    node->left = builtTree(postorder, inorder, start, pos - 1);


    return node;
}
```

Sum at kth level:

```
void sumAtKLevel(Node *root, int k)
{
    if (root == NULL)
    {
        return;
    }
    int level = 0;
    int sum = 0;
    queue<Node *> q;
    q.push(root);
    q.push(NULL);
    while (!q.empty())
    {
        Node *n = q.front();
        q.pop();
        if (n != NULL)
        {
            if (level == k)
            {
                sum += n->data;
            }
            if (n->left)
            {
                q.push(n->left);
            }
            if (n->right)
            {
                q.push(n->right);
            }
        }
        else if (!q.empty())
        {
            q.push(NULL);
```

```
            level++;
        }
    }
    cout << sum;
}
```

Count Nodes in a tree:

```
int countNodes(Node *root)
{
    if (root == NULL)
    {
        return 0;
    }

    return countNodes(root->left) + countNodes(root->right) + 1;
}
```

Sum of all nodes in a tree:

```
int sumNodes(Node *root)
{
    if (root == NULL)
    {
        return 0;
    }

    return sumNodes(root->left) + sumNodes(root->right) + root->data;
}
```

Height of binary tree || Maximum depth of binary tree:

```
int calHeight(Node *root)
{
    if (root == NULL)
    {
        return 0;
    }

    int lheight = calHeight(root->left);
    int rheight = calHeight(root->right);
    return max(lheight, rheight) + 1;
}
```

Check for Balanced Binary tree:

A balanced binary tree is a binary tree that follows the 3 conditions:

The height of the left and right tree for any node does not differ by more than 1.

The left subtree of that node is also balanced.

The right subtree of that node is also balanced.

```
int checkHeight(TreeNode *root)
{
    if (root == NULL)
        return 0;

    int leftHeight = checkHeight(root->left);
    if (leftHeight == -1)
        return -1;

    int rightHeight = checkHeight(root->right);
    if (rightHeight == -1)
        return -1;

    int heightDiff = abs(leftHeight - rightHeight);
    if (heightDiff > 1)
        return -1;

    return 1 + max(leftHeight, rightHeight);
}
```

```
bool isBalanced(TreeNode *root)
{
    return (checkHeight(root) != -1);
}
```

Diameter of Binary tree:

The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root. The length of the path between two nodes is represented by the number of edges between them.

- Brute force

```
int maxi = 0;

// Function to find the height of a binary tree
int height(TreeNode *root)
{
    if (root == NULL)
        return 0;
    return 1 + max(height(root->left), height(root->right));
}

// Function to find the diameter of a binary tree
void diameter(TreeNode *root)
{
    if (root == NULL)
        return;

    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    maxi = max(maxi, leftHeight + rightHeight);
    diameter(root->left);
    diameter(root->right);
}
```

O (n^2).

- Efficient approach

```
int maxi = 0;
int diameter(TreeNode *root)
{
    if (root == NULL)
        return 0;

    int leftHeight = diameter(root->left);
    int rightHeight = diameter(root->right);
    maxi = max(maxi, leftHeight + rightHeight);
    return 1 + max(leftHeight, rightHeight);
}
```

Maximum Path sum in binary tree:

```
int maxi = 0;
int maxPathSum(TreeNode *node)
{
    if (node == NULL)
        return 0;

    int left = max(0, maxPathSum(node->left));
    int right = max(0, maxPathSum(node->right));
    maxi = max(maxi, left + right + node->val);
    return max(left, right) + node->val;
}
```

Check 2 trees are identical or not:

```
bool areIdentical(TreeNode *root1, TreeNode *root2)
{
    // Base cases: if both trees are empty, they are identical
    if (root1 == NULL && root2 == NULL)
        return true;
```

```
    // If one of the trees is empty and the other is not, they are not identical
    if (root1 == NULL || root2 == NULL)
        return false;

    // Check if the current nodes have the same value
    if (root1->val != root2->val)
        return false;

    // Recur for left and right subtrees
    return areIdentical(root1->left, root2->left) && areIdentical(root1->right,
root2->right);
}
```

Top view of Binary tree:

```
void topView(TreeNode *root)
{
    if (root == NULL)
        return;

    queue<pair<TreeNode *, int>> q; // Queue for level order traversal
    map<int, int> m;     // Map to store horizontal distance and node value

    q.push({root, 0}); // Push root and its horizontal distance

    while (!q.empty())
    {
        TreeNode *node = q.front().first;
        int hd = q.front().second;
        q.pop();

        if (m.find(hd) == m.end())
        {
            m[hd] = node->val; // Store the first node encountered at each
horizontal distance
        }

        if (node->left)
            q.push({node->left, hd - 1}); // Push left child
        if (node->right)
```

```
            q.push({node->right, hd + 1}); // Push right child
    }

    // Print the top view
    for (auto it = m.begin(); it != m.end(); it++)
    {
        cout << it->second << " ";
    }
}
```

Bottom view of Binary tree:

```
void bottomView(TreeNode *root)
{
    if (root == NULL)
        return;

    map<int, int> m;                // Map to store horizontal distance and
corresponding node value
    queue<pair<TreeNode *, int>> q; // Queue to perform level order traversal

    q.push({root, 0});

    while (!q.empty())
    {
        TreeNode *node = q.front().first;
        int hd = q.front().second;
        q.pop();

        m[hd] = node->val;

        if (node->left)
            q.push({node->left, hd - 1});
        if (node->right)
            q.push({node->right, hd + 1});
    }

    for (auto it : m)
```

```
    {
        cout << it.second << " ";
    }
}
```

Right view of Binary tree:

```
void rightView(TreeNode *root, int level, vector<int> &v)
{
    if (root == NULL)
        return;

    if (level == v.size())
    {
        v.push_back(root->val);
    }

    rightView(root->right, level + 1, v);
    rightView(root->left, level + 1, v);
}
```

Left view of Binary tree:

```
void leftView(TreeNode *root, int level, vector<int> &v)
{
    if (root == NULL)
        return;

    if (level == v.size())
    {
        v.push_back(root->val);
    }

    leftView(root->left, level + 1, v);
    leftView(root->right, level + 1, v);
}
```

Check for symmetrical Binary trees:

```
bool isSymmetric(TreeNode *leftNode, TreeNode *rightNode)
{
    if (leftNode == nullptr || rightNode == nullptr)
        return leftNode == rightNode;

    if (leftNode->val != rightNode->val)
        return false;

    return isSymmetric(leftNode->left, rightNode->right) && isSymmetric(leftNode-
>right, rightNode->left);
}
```

Print root to path node in binary tree:

```
Vector<int>ans;
bool printPath(TreeNode *root, vector<int> &ans, int target)
{
    if (root == NULL)
    {
        return false;
    }
    ans.push_back(root->val);
    if (root->val == target)
        return true;

    if (printPath(root->left, ans, target) || printPath(root->right, ans,
target))
        return true;

    ans.pop_back();

    return false;
}
```

Sum Replacement in binary tree:

```
void sumReplacement(Node *root)
{
    if (root == NULL)
        return;

    sumReplacement(root->left);
    sumReplacement(root->right);
    if (root->left != NULL)
    {
        root->data += root->left->data;
    }
    if (root->right != NULL)
    {
        root->data += root->right->data;
    }
}
```

Lowest Common ancestors in binary tree:

```
Node* Lca(TreeNode *root, TreeNode *p, TreeNode *q)
{
    if (root == NULL || root == p || root == q)
    {
        return root;
    }

    TreeNode *left = Lca(root->left, p, q);
    TreeNode *right = Lca(root->right, p, q);
    if (left == NULL)
        return right;

    else if (right == NULL)
        return left;
    else
    {
        return root;
    }
}
```

Flatten a binary tree to LinkedList:

```
Node *prev = NULL;
void flatten(Node *node)
{
    if (node == NULL)
        return;

    flatten(node->right);
    flatten(node->left);
    node->right = prev;
    node->left = NULL;
    prev = node;
}
```

Print all nodes at distance k in binary tree:

```
void makeParent(Node *node, unordered_map<Node *, Node *> &parentTrack)
{
    queue<Node *> q;
    q.push(node);
    while (!q.empty())
    {
        Node *curr = q.front();
        q.pop();
        if (curr->left)
        {
```

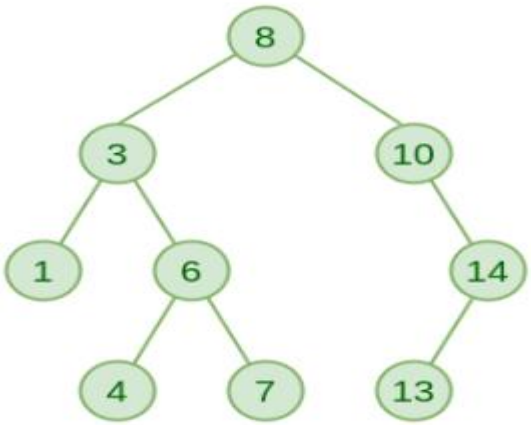
```
            parentTrack[curr->left] = curr;
            q.push(curr->left);
        }
        if (curr->right)
        {
            parentTrack[curr->right] = curr;
            q.push(curr->right);
        }
    }
}

vector<int> printKDistanceNodes(TreeNode *root, int target, int k)
{
    if (root == NULL)
        return;

    unordered_map<TreeNode *, TreeNode *> parent;
    makeParent(root, parent);

    unordered_map<TreeNode *, bool> visited;
    queue<TreeNode *> q;
    q.push(root);
    visited[root] = true;

    int distance = 0;

    while (!q.empty())
    {
        int levelSize = q.size();
        if (distance == k)
            break;
        for (int i = 0; i < levelSize; i++)
        {
            TreeNode *node = q.front();
            q.pop();

            if (node->left != NULL && !visited[node->left])
            {
                q.push(node->left);
                visited[node->left] = true;
            }

            if (node->right != NULL && !visited[node->right])
            {
                q.push(node->right);
```

```
                visited[node->right] = true;
            }

            TreeNode *parentNode = parentTrack[node];
            if (parentNode != NULL && !visited[parentNode])
            {
                q.push(parentNode);
                visited[parentNode] = true;
            }
        }
        distance++;
    }
    vector<int> result;
    while (!q.empty())
    {
        TreeNode *node = q.front();
        q.pop();
        result.push_back(node->val);
    }
    return result;
}
```

Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Inserting a node in BST:

```
Node *insert(Node *root, int val)
{
    if (root == NULL)
    {
        return new Node(val);
    }
    if (val < root->data)
    {
        root->left = insert(root->left, val);
    }
    else
    {
        root->right = insert(root->right, val);
    }
}
```

Note: *In case of binary search trees (BST), In-order traversal gives nodes in non-decreasing order.*

Print leaf nodes in BST:

```
void printLeafNodes(TreeNode *root)
{
    if (root == NULL)
        return;

    if (root->left == NULL && root->right == NULL)
    {
        cout << root->val << " ";
    }

    if (root->left != NULL)
    {
        printLeafNodes(root->left);
    }

    if (root->right != NULL)
    {
        printLeafNodes(root->right);
    }
}
```

Print non-leaf nodes in BST:

```
void printNonLeafNode(struct node *root)
{
    // Base Cases
    if (root == NULL || (root->left == NULL && root->right == NULL))
        return;

    // If current node is non-leaf,
    if (root->left != NULL || root->right != NULL)
    {
        cout << " " << root->key;
    }

    // If root is Not NULL and its one
    // of its child is also not NULL
    printNonLeafNode(root->left);
    printNonLeafNode(root->right);
}
```

Searching in BST:

```
bool search(Node *root, int key)
{ /// time complexity  O(logn)
    if (root == NULL)
    {
        return false;
    }
    if (key == root->data)
    {
        // cout << "Number found!";
        return true;
    }
    if (key < root->data)
    {
        search(root->left, key);
    }
    else
    {
        search(root->right, key);
    }
}
```

Deletion in BST:

```
Node *inOrderSucc(Node *root)
{
    Node *curr = root;
    while (curr && curr->left != NULL)
    {
        curr = curr->left;
    }
    return curr;
}
Node *deletion(Node *root, int del)
{
```

```
    if (del < root->data)
    {
        root->left = deletion(root->left, del);
    }
    else if (del > root->data)
    {
        root->right = deletion(root->right, del);
    }
    else
    {
        if (root->left == NULL)
        {
            Node *temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == NULL)
        {
            Node *temp = root->left;
            delete root;
            return temp;
        }

        Node *temp = inOrderSucc(root->right);
        root->data = temp->data;
        root->right = deletion(root->right, temp->data);
    }
}
```

Ceil in BST:

```
int findCeiling(Node *root, int value)
{
    if (root == nullptr)
    {
        return -1; // If tree is empty
    }
    int ceilValue = -1;
    while (root != nullptr)
    {
        if (root->data == value)
        {
            ceilValue = root->data; // If the current node's data is equal to the
value
```

```
            break;
        }
        else if (root->data < value)
        {
            root = root->right; // Search in the right subtree
        }
        else
        {
            ceilValue = root->data; // If the current node's data is greater than
the value
            root = root->left;      // Search in the left subtree
        }
    }
    return ceilValue;
}
```

Floor in BST:

```
int findFloor(Node *root, int value)
{
    if (root == nullptr)
    {
        return -1; // If tree is empty
    }
    int floorValue = -1;
    while (root != nullptr)
    {
        if (root->data == value)
        {
            floorValue = root->data; // If the current node's data is equal to
the value
            break;
        }
        else if (root->data > value)
        {
            root = root->left; // Search in the left subtree
        }
        else
        {
            floorValue = root->data; // If the current node's data is less than
the value
            root = root->right;      // Search in the right subtree
```

```
        }
    }
    return floorValue;
}
```

Kth smallest element in BST:

```
void kthSmallestUtil(TreeNode *root, int k, int &count, int &result)
{
    if (root == NULL || count >= k)
        return;

    kthSmallestUtil(root->left, k, count, result);

    count++;
    if (count == k)
    {
        result = root->val;
        return;
    }

    kthSmallestUtil(root->right, k, count, result);
}

int kthSmallest(TreeNode *root, int k)
{
    int count = 0;
    int result = -1;

    kthSmallestUtil(root, k, count, result);

    return result;
}
```

Kth largest element in BST:

```
void kthLargestUtil(TreeNode *root, int k, int &count, int &result)
{
    if (root == NULL || count >= k)
        return;

    kthLargestUtil(root->right, k, count, result);

    count++;
    if (count == k)
    {
        result = root->val;
        return;
    }

    kthLargestUtil(root->left, k, count, result);
}

int kthLargest(TreeNode *root, int k)
{
    int count = 0;
    int result = -1;

    kthLargestUtil(root, k, count, result);

    return result;
}
```

Check given tree is BST or not:

```
bool isValidBSTUtil(TreeNode *root, long long int min_val, long long int max_val)
{
    if (root == NULL)
        return true;

    if (root->val <= min_val || root->val >= max_val)
        return false;

    return isValidBSTUtil(root->left, min_val, root->val) && isValidBSTUtil(root->right, root->val, max_val);
}
```

```
bool isValidBST(TreeNode *root)
{
    return isValidBSTUtil(root, LLONG_MIN, LLONG_MAX);
}
```

Lowest Common ancestors in BST:

```
TreeNode *findLCA(TreeNode *root, int p, int q)
{
    if (root == NULL)
        return NULL;

    if (p < root->val && q < root->val)
    {
        return findLCA(root->left, p, q);
    }
    else if (p > root->val && q > root->val)
    {
        return findLCA(root->right, p, q);
    }

    return root;
}
```

Construct BST from preorder traversal:

```
TreeNode builtTreefrom Preorder(vector<int> a)
{
    int i = 0;
    return built(a, i, INT_MAX);
}

TreeNode *built(vector<int> a, int &i, int b)
```

```
{
    if (i == a.size() || a[i] > b)
        return NULL;
    TreeNode *root = new TreeNode(a[i++]);
    root->left = built(a, i, root->val);
    root->right = built(a, i, b);
    return root;
}
```

Finding in order successor of node in BST:

```
TreeNode *inorderSuccessor(TreeNode *root, TreeNode *p)
{
    TreeNode *successor = nullptr;

    while (root)
    {
        if (p->val >= root->val)
        {
            root = root->right;
        }
        else
        {
            successor = root;
            root = root->left;
        }
    }

    return successor;
}
```

BST Iterator:

```
class BSTIterator
{
public:
    BSTIterator(TreeNode *root)
    {
        // Initialize the iterator by pushing all left nodes onto the stack
        while (root != nullptr)
        {
            nodes.push(root);
            root = root->left;
        }
    }

    /** @return the next smallest number */
    int next()
    {
        // The smallest element is at the top of the stack
        TreeNode *current = nodes.top();
        nodes.pop();

        // If the current node has a right subtree, push its left nodes onto the
stack
        if (current->right != nullptr)
        {
            TreeNode *temp = current->right;
            while (temp != nullptr)
            {
                nodes.push(temp);
                temp = temp->left;
            }
        }

        return current->val;
    }


    bool hasNext()
    {
        return !nodes.empty();
    }

private:
```

```
    stack<TreeNode *> nodes; // Stack to store the nodes in the path to the
smallest element
};
```

Recover BST | Correct BST with 2 node swapping:

```
void inorderTraversal(TreeNode *root, vector<int> &values)
{
    if (root == nullptr)
        return;

    inorderTraversal(root->left, values);
    values.push_back(root->val);
    inorderTraversal(root->right, values);
}

void correctBST(TreeNode *root, const vector<int> &sortedValues, int &index)
{
    if (root == nullptr)
        return;

    correctBST(root->left, sortedValues, index);

    // Correct the node value
    root->val = sortedValues[index++];

    correctBST(root->right, sortedValues, index);
}

void recoverTree(TreeNode *root)
{
    vector<int> values;
    inorderTraversal(root, values);

    // Sort the values to identify the two swapped nodes
    sort(values.begin(), values.end());

    // Correct the BST
    int index = 0;
    correctBST(root, values, index);
}
```

Largest BST in Binary Tree:

```
struct BSTInfo
{
    int size;     // Size of the BST
    int minValue; // Minimum value in the BST
    int maxValue; // Maximum value in the BST
    BSTInfo(int size, int minValue, int maxValue)
    {
        size = size;
        minValue = minValue;
        maxValue = maxValue;
    }
};

BSTInfo largestBSTHelper(TreeNode *root)
{
    // Base case: for a null node, return a BSTInfo with size 0
    if (!root)
    {
        return BSTInfo(0, INT_MAX, INT_MIN);
    }

    // Recursively get information from left and right subtrees
    BSTInfo leftInfo = largestBSTHelper(root->left);
    BSTInfo rightInfo = largestBSTHelper(root->right);

    // Check if the current subtree is a valid BST
    if (
        root->val > leftInfo.maxValue && root->val < rightInfo.minValue)
    {
        // If valid, update the size, min, and max values
        int size = 1 + leftInfo.size + rightInfo.size;
        int minValue = min(root->val, leftInfo.minValue);
        int maxValue = max(root->val, rightInfo.maxValue);

        return BSTInfo(size, minValue, maxValue);
    }
```

```
    else
    {
        // If not valid, return information with isBST false
        // Set size to 0 and return INT_MAX and INT_MIN
        return BSTInfo(0, INT_MIN, INT_MAX);
    }
}

int largestBST(TreeNode *root)
{
    return largestBSTHelper(root).size;
}
```

Trie

A trie (pronounced "try") is a tree-like data structure used for efficiently storing and searching a dynamic set or associative array where the keys are usually strings. The word "trie" comes from the word "retrieval," and it is also sometimes referred to as a "prefix tree" because it efficiently handles operations based on prefixes.

Key Characteristics of a Trie:

- **Tree Structure**: A trie is a tree-like structure where each node represents a single character in a string.

- **Node Structure**: Each node in the trie typically contains an array or a set of pointers, each corresponding to a possible character in the alphabet.

- **Path to Nodes**: The path from the root to a particular node represents a string formed by concatenating the characters along that path.

- **End-of-Word Flag**: Each node may have a flag indicating whether a word ends at that node.

Operations:

Insertion: Adding a word to the trie involves creating nodes for each character in the word, starting from the root and following the path corresponding to the characters.

Search: Checking whether a specific word exists in the trie involves traversing the trie from the root, following the path of characters in the word.

Prefix Search (StartsWith): Determining if there are words with a given prefix involves traversing the trie based on the characters of the prefix.

```
#include <iostream>

struct Node
{
    Node *children[26];
    bool isEndOfWord;

    Node() : isEndOfWord(false)
    {
        for (int i = 0; i < 26; ++i)
        {
            children[i] = nullptr;
        }
    }

    // Destructor to free memory
    ~Node()
    {
        for (int i = 0; i < 26; ++i)
        {
            delete children[i];
        }
    }
};

class Trie
{
private:
    Node *root;
```

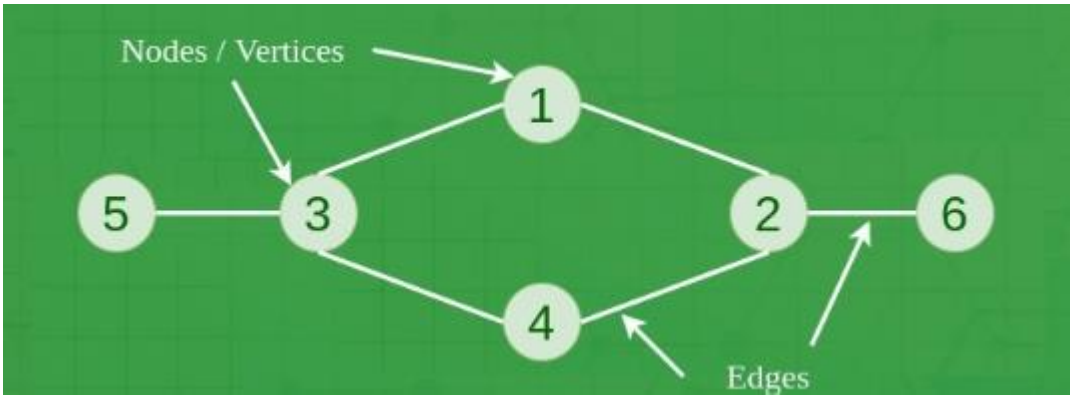
```
public:
    Trie()
    {
        root = new Node();
    }

    // Destructor to free memory
    ~Trie()
    {
        delete root;
    }

    void insert(const std::string &word)
    {
        Node *current = root;

        for (char ch : word)
        {
            int index = ch - 'a';
            if (!current->children[index])
            {
                current->children[index] = new Node();
            }
            current = current->children[index];
        }

        current->isEndOfWord = true;
    }

    bool search(const std::string &word) const
    {
        Node *current = root;

        for (char ch : word)
        {
            int index = ch - 'a';
            if (!current->children[index])
            {
                return false;
            }
            current = current->children[index];
        }

        return current->isEndOfWord; // Check if the word ends at this node
```

```
    }

    // Returns true if there is any word in the trie that starts with the given
prefix
    bool startsWith(const std::string &prefix) const
    {
        Node *current = root;

        for (char ch : prefix)
        {
            int index = ch - 'a';
            if (!current->children[index])
            {
                return false; // Prefix not found in the trie
            }
            current = current->children[index];
        }

        return true; // The prefix is found in the trie
    }
};

int main()
{
    Trie trie;

    // Insert words into the trie
    trie.insert("apple");
    trie.insert("app");
    trie.insert("application");
    trie.insert("banana");

    // Search for words
    std::cout << "Search 'app': " << (trie.search("app") ? "Found" : "Not found")
<< std::endl;
    std::cout << "Search 'orange': " << (trie.search("orange") ? "Found" : "Not
found") << std::endl;

    // Check if there are words with a given prefix
    std::cout << "Starts with 'app': " << (trie.startsWith("app") ? "Yes" : "No")
<< std::endl;
    std::cout << "Starts with 'ban': " << (trie.startsWith("ban") ? "Yes" : "No")
<< std::endl;

    return 0;
```

```
}
```

Graph

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).



Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or

circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Types of Graph:

There are several types of graph data structures, each with its own characteristics and use cases.

- **Undirected Graph:**

An undirected graph is a graph where edges have no direction. The relationship between nodes is symmetric.

Example: Social networks where friendships exist between individuals without a specific direction.

- **Directed Graph (Digraph):**

In a directed graph, edges have a direction from one node to another. The relationship between nodes is asymmetric.

Example: Web pages with hyperlinks. The links have a direction, pointing from one page to another.

- **Weighted Graph:**

In a weighted graph, each edge has an associated weight or cost. These weights can represent distances, costs, time, etc.

Example: Road networks where edges represent roads and weights represent distances.

- **Unweighted Graph:**

In contrast to weighted graphs, unweighted graphs don't have associated weights with their edges.

Example: Representing connections between airports without considering distances.

- **Cyclic Graph:**

A graph that contains at least one cycle (a path that starts and ends at the same node).

Example: A network of dependencies where A depends on B, B depends on C, and C depends on A.

- **Acyclic Graph (DAG - Directed Acyclic Graph):**

A directed graph without cycles. DAGs are often used to represent dependencies and are commonly used in scheduling and optimization problems.

Example: Dependency graph of tasks in a project where no task depends on itself.

- **Connected Graph:**

A graph in which there is a path between every pair of nodes. There are no isolated nodes.

Example: A fully connected social network.

- **Disconnected Graph:**

A graph where some nodes are not connected to any other nodes.

Example: Multiple isolated components in a social network.

Representation of Graph:

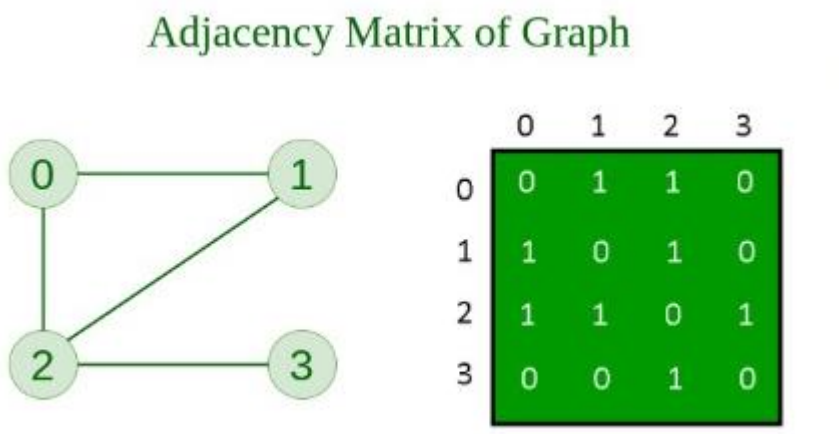
There are two ways to store a graph:

Adjacency Matrix

Adjacency List

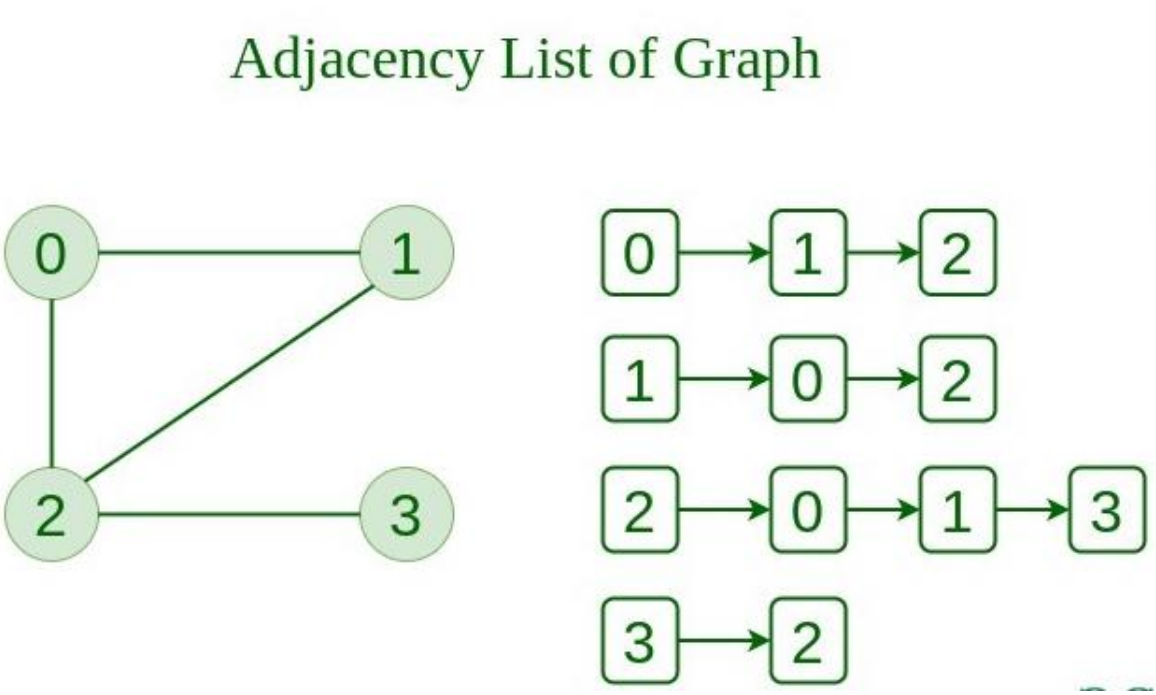
- **Adjacency Matrix**

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.

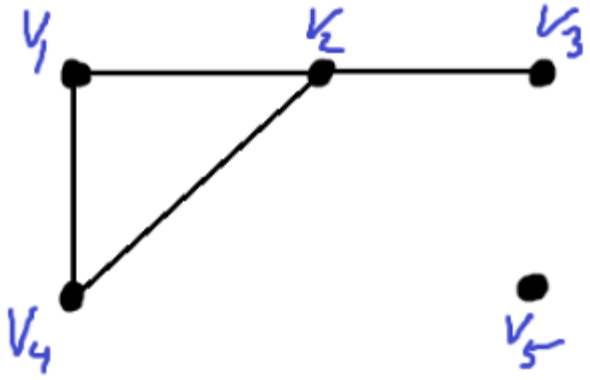


- **Adjacency List:**

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Degree of a vertex:



$deg(v_1) = 2$
$deg(v_2) = 3$
$deg(v_4) = 2$
$deg(v_3) = 1$
$deg(v_5) = 0$

Advantages:

- Graphs are a versatile data structure that can be used to represent a wide range of relationships and data structures.
- They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
- Graph algorithms are often very efficient and can be used to solve complex problems quickly and effectively.
- Graphs can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

Disadvantages:

- Graphs can be complex and difficult to understand, especially for people who are not familiar with graph theory or related algorithms.
- Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.
- Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.

- Graphs can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

Graph Traversal:

- BFS (Breath First Search) -> QUEUE
- DFS (Depth First Search) -> STACK

DFS traversal:

```
void DFSUtil(list<int> adj_list[], int v, bool visited[])
{
    visited[v] = true;
    cout << v << " ";

    for (int neighbor : adj_list[v])
    {
        if (!visited[neighbor])
        {
            DFSUtil(adj_list, neighbor, visited);
        }
    }
}

void DFS(list<int> adj_list[], int start)
{
    bool *visited = new bool[numvertices];
    for (int i = 0; i < numvertices; i++)
    {
        visited[i] = false;
    }

    cout << "Depth First Traversal: ";
```

```
    DFSUtil(adj_list, start, visited);

    delete[] visited;
    cout << endl;
}
```

BFS traversal:

```
void BFSUtil(list<int> adj_list[], vector<bool> &visited, queue<int> &q)
{
    if (q.empty())
        return;

    int current = q.front();
    cout << current << " ";
    q.pop();

    for (int neighbor : adj_list[current])
    {
        if (!visited[neighbor])
        {
            visited[neighbor] = true;
            q.push(neighbor);
        }
    }

    BFSUtil(adj_list, visited, q);
}

void BFS(list<int> adj_list[], int start)
{
    vector<bool> visited(numVertices, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    cout << "Breadth First Traversal: ";
    BFSUtil(adj_list, visited, q);
```

```
    cout << endl;
}
```

Adjacency Matrix Implementation:

```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;
int adjMatrix[20][20];
class Graph
{
private:
    int numVertices;
    bool *visited;

public:
    // Initialize the matrix to zero
    Graph(int numVertices)
    {
        this->numVertices = numVertices;
        for (int i = 0; i < numVertices; i++)
        {
            for (int j = 0; j < numVertices; j++)
            {
                adjMatrix[i][j] = 0;
            }
        }
    }

    // Add edges
    void add_Edge(int i, int j)
    {
        adjMatrix[i][j] = 1;
        adjMatrix[j][i] = 1;
```

```
    }

    // Remove edges
    void removeEdge(int i, int j)
    {
        adjMatrix[i][j] = 0;
        adjMatrix[j][i] = 0;
    }

    // Add a vertex

    void addVertex()
    {
        // increasing the number of vertices
        numVertices++;
        // initializing the new elements to 0
        for (int i = 0; i < numVertices; ++i)
        {
            adjMatrix[i][numVertices - 1] = 0;
            adjMatrix[numVertices - 1][i] = 0;
        }
    }

    // Remove vertex

    void removeVertex(int x)
    {
        // checking if the vertex is present
        if (x > numVertices)
        {
            cout << "\nVertex not present!";
            return;
        }
        else
        {
            int i;

            // removing the vertex
            while (x < numVertices)
            {

                // shifting the rows to left side
                for (i = 0; i < numVertices; ++i)
                {
                    adjMatrix[i][x] = adjMatrix[i][x + 1];
```

```
            }

            // shifting the columns upwards
            for (i = 0; i < numVertices; ++i)
            {
                adjMatrix[x][i] = adjMatrix[x + 1][i];
            }
            x++;
        }
        numVertices--;
    }
}

bool searchVertex(int x)
{
    if (x > numVertices)
        return false;
    else
        return true;
}

bool searchEdge(int x, int y)
{
    if (adjMatrix[x][y] == 1)
        return true;

    return false;
}

// bfs traversal

void Bfs(int source)
{
    visited = new bool[numVertices];
    for (int i = 0; i < numVertices; i++)
        visited[i] = false;
    queue<int> q;
    q.push(source);
    visited[source] = true;

    while (!q.empty())
    {
        int curr = q.front();
        q.pop();
        cout << curr << " ";
```

```
            for (int i = 0; i < numVertices; i++)
            {
                if (adjMatrix[curr][i] == 1 && visited[i] != true)
                {
                    q.push(i);
                    // cout << i << endl;
                    visited[i] = true;
                }
            }
        }
    }

    // dfs traversal

    void Dfs(int source)
    {
        visited = new bool[numVertices];
        for (int i = 0; i < numVertices; i++)
            visited[i] = false;
        stack<int> s;
        s.push(source);
        visited[source] = true;

        while (!s.empty())
        {
            int curr = s.top();
            s.pop();
            cout << curr << " ";
            for (int i = 0; i < numVertices; i++)
            {
                if (adjMatrix[curr][i] == 1 && visited[i] != true)
                {
                    s.push(i);
                    // cout << i << endl;
                    visited[i] = true;
                }
            }
        }
    }

    // dispaly vertices

    void displayVertices()
    {
        int i = 0;
```

```
        while (i < numVertices)
        {
            cout << i << ' ';
            i++;
        }
    }

    // Print the martix
    void toString()
    {
        for (int i = 0; i < numVertices; i++)
        {
            // cout << i << " : ";
            for (int j = 0; j < numVertices; j++)
                cout << adjMatrix[i][j] << " ";
            cout << "\n";
        }
    }
};
```

Adjacency List Implementation:

```
#include <iostream>
#include <list>
#include <iterator>
#include <queue>
#include <stack>
using namespace std;
class Graph
{
private:
    int numvertices;

public:
```

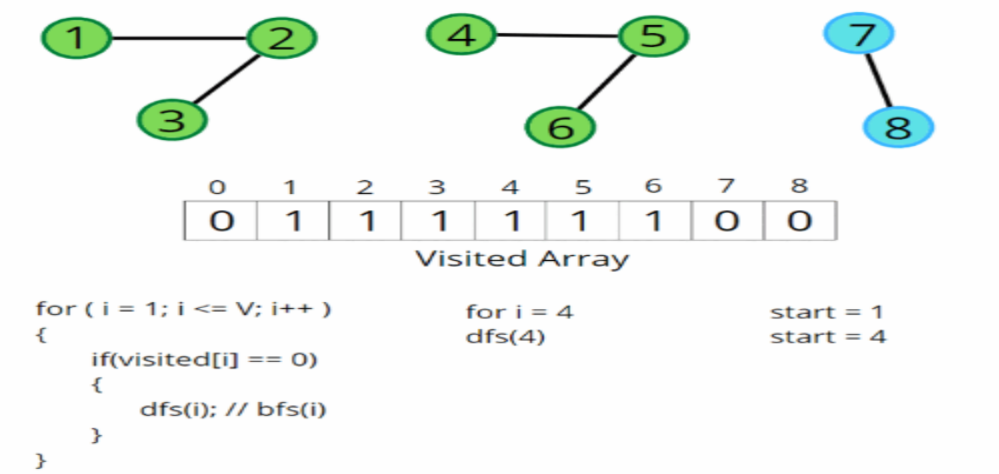
```
Graph(int v)
{
    this->numvertices = v;
}
void displayAdjList(list<int> adj_list[])
{
    for (int i = 0; i < numvertices; i++)
    {
        if (!adj_list[i].empty())
            cout << i << "--->";

        list<int>::iterator it;
        for (it = adj_list[i].begin(); it != adj_list[i].end(); ++it)
        {
            cout << *it << " ";
        }
        cout << endl;
    }
}
void add_edge(list<int> adj_list[], int u, int v)
{ // add v into the list u, and u into list v
    adj_list[u].push_back(v);
    adj_list[v].push_back(u);
}

void remove_edge(list<int> adj_list[], int u, int v)
{
    list<int>::iterator it;
    for (it = adj_list[u].begin(); it != adj_list[u].end(); ++it)
    {
        if (v == *it)
        {
            adj_list[u].erase(it);
            break;
        }
    }
    list<int>::iterator it1;
    for (it1 = adj_list[v].begin(); it1 != adj_list[v].end(); ++it1)
    {
        if (u == *it1)
        {
            adj_list[v].erase(it1);
            break;
        }
    }
```

```
}

void add_vertex(list<int> adj_list[])
{
    numvertices++;

    adj_list[numvertices].push_back(-1);
}
void remove_vertex(list<int> adj_list[], int u)
{
    list<int>::iterator it;
    for (int i = 0; i < numvertices; i++)
    {
        for (it = adj_list[i].begin(); it != adj_list[i].end(); ++it)
        {
            if (u == *it)
            {
                adj_list[i].erase(it);
                break;
            }
        }
    }
    adj_list[u].clear();
}
bool search_edge(list<int> adj_list[], int u, int v)
{
    for (int i : adj_list[u])
    {
        if (i == v)
            return true;
    }
    return false;
}
void BFS(list<int> adj_list[], int start)
{
    bool *visited = new bool[numvertices];
    for (int i = 0; i < numvertices; i++)
    {
        visited[i] = false;
    }

    queue<int> q;
    visited[start] = true;
    q.push(start);
```

```
        cout << "Breadth First Traversal: ";
        while (!q.empty())
        {
            int current = q.front();
            cout << current << " ";
            q.pop();

            for (int neighbor : adj_list[current])
            {
                if (!visited[neighbor])
                {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }

        delete[] visited;
        cout << endl;
    }
    void DFS(list<int> adj_list[], int start)
    {
        bool *visited = new bool[numvertices];
        for (int i = 0; i < numvertices; i++)
        {
            visited[i] = false;
        }

        stack<int> s;
        visited[start] = true;
        s.push(start);

        cout << "Depth First Traversal: ";
        while (!s.empty())
        {
            int current = s.top();
            cout << current << " ";
            s.pop();

            for (int neighbor : adj_list[current])
            {
                if (!visited[neighbor])
                {
                    visited[neighbor] = true;
                    s.push(neighbor);
```

```
                }
            }
        }

        delete[] visited;
        cout << endl;
    }
};
```

Number of Provinces:




```
class Solution
{
private:
    // dfs traversal function
    void dfs(int node, vector<int> adjLs[], int vis[])
    {
        // mark the more as visited
        vis[node] = 1;
        for (auto it : adjLs[node])
        {
            if (!vis[it])
            {
                dfs(it, adjLs, vis);
            }
        }
```

```
    }

public:
    int numProvinces(vector<vector<int>> adj, int V)
    {
        vector<int> adjLs[V];

        // to change adjacency matrix to list
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                // self nodes are not considered
                if (adj[i][j] == 1 && i != j)
                {
                    adjLs[i].push_back(j);
                    adjLs[j].push_back(i);
                }
            }
        }
        int vis[V] = {0};
        int cnt = 0;
        for (int i = 0; i < V; i++)
        {
            // if the node is not visited
            if (!vis[i])
            {
                // counter to count the number of provinces
                cnt++;
                dfs(i, adjLs, vis);
            }
        }
        return cnt;
    }
};
```

Number of Islands:

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

class Solution
{
public:
    int numIslands(vector<vector<char>> &grid)
    {
        if (grid.empty() || grid[0].empty())
        {
            return 0;
        }

        int rows = grid.size();
        int cols = grid[0].size();
        int numIslands = 0;

        // Create a 2D matrix to keep track of visited nodes
        vector<vector<bool>> visited(rows, vector<bool>(cols, false));

        for (int i = 0; i < rows; ++i)
        {
            for (int j = 0; j < cols; ++j)
            {
                if (grid[i][j] == '1' && !visited[i][j])
                {
                    ++numIslands;
                    bfs(grid, visited, i, j);
                }
            }
        }

        return numIslands;
    }

private:
    void bfs(vector<vector<char>> &grid, vector<vector<bool>> &visited, int row,
int col)
    {
        int rows = grid.size();
```

```
        int cols = grid[0].size();

        queue<pair<int, int>> q;
        q.push({row, col});
        visited[row][col] = true;

        vector<pair<int, int>> directions = {
            {-1, 0}, {1, 0}, {0, -1}, {0, 1}, // Horizontal and vertical
            {-1, -1},
            {-1, 1},
            {1, -1},
            {1, 1} // Diagonal
        };

        while (!q.empty())
        {
            pair<int, int> current = q.front();
            q.pop();

            for (const auto &dir : directions)
            {
                int newRow = current.first + dir.first;
                int newCol = current.second + dir.second;

                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols
&&
                    grid[newRow][newCol] == '1' && !visited[newRow][newCol])
                {
                    q.push({newRow, newCol});
                    visited[newRow][newCol] = true;
                }
            }
        }
    }
};
```

*Time complexity: O(n*n)* *Space complexity: O(n*n)*

Flood fill algorithm:

```cpp
class Solution
{
private:
    void dfs(int row, int col, vector<vector<int>> &ans,
             vector<vector<int>> &image, int newColor, int delRow[], int
delCol[],
             int iniColor)
    {
        // color with new color
        ans[row][col] = newColor;
        int n = image.size();
        int m = image[0].size();
        // there are exactly 4 neighbours
        for (int i = 0; i < 4; i++)
        {
            int nrow = row + delRow[i];
            int ncol = col + delCol[i];
            // check for valid coordinate
            // then check for same initial color and unvisited pixel
            if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
                image[nrow][ncol] == iniColor && ans[nrow][ncol] != newColor)
            {
                dfs(nrow, ncol, ans, image, newColor, delRow, delCol, iniColor);
            }
        }
    }

public:
    vector<vector<int>> floodFill(vector<vector<int>> &image,
                                  int sr, int sc, int newColor)
    {
        // get initial color
        int iniColor = image[sr][sc];
        vector<vector<int>> ans = image;
        // delta row and delta column for neighbours
        int delRow[] = {-1, 0, +1, 0};
        int delCol[] = {0, +1, 0, -1};
        dfs(sr, sc, ans, image, newColor, delRow, delCol, iniColor);
        return ans;
    }
};
```

*Time complexity: O(n*m)* *Space complexity: O(n*m)*

Rotten Oranges:

```cpp
class Solution
{
public:
    // Function to find minimum time required to rot all oranges.
    int orangesRotting(vector<vector<int>> &grid)
    {
        // figure out the grid size
        int n = grid.size();
        int m = grid[0].size();

        // store {{row, column}, time}
        queue<pair<pair<int, int>, int>> q;
        int vis[n][m];
        int cntFresh = 0;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                // if cell contains rotten orange
                if (grid[i][j] == 2)
                {
                    q.push({{i, j}, 0});
                    // mark as visited (rotten) in visited array
                    vis[i][j] = 2;
                }
                // if not rotten
                else
                {
                    vis[i][j] = 0;
                }
                // count fresh oranges
                if (grid[i][j] == 1)
                    cntFresh++;
            }
        }

        int tm = 0;
        // delta row and delta column
        int drow[] = {-1, 0, +1, 0};
        int dcol[] = {0, 1, 0, -1};
        int cnt = 0;
```

```
        // bfs traversal (until the queue becomes empty)
        while (!q.empty())
        {
            int r = q.front().first.first;
            int c = q.front().first.second;
            int t = q.front().second;
            tm = max(tm, t);
            q.pop();
            // exactly 4 neighbours
            for (int i = 0; i < 4; i++)
            {
                // neighbouring row and column
                int nrow = r + drow[i];
                int ncol = c + dcol[i];
                // check for valid cell and
                // then for unvisited fresh orange
                if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
                    vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1)
                {
                    // push in queue with timer increased
                    q.push({{nrow, ncol}, t + 1});
                    // mark as rotten
                    vis[nrow][ncol] = 2;
                    cnt++;
                }
            }
        }

        // if all oranges are not rotten
        if (cnt != cntFresh)
            return -1;

        return tm;
    }
};
```

*Time complexity: O(n*m)* *Space complexity: O(n*m)*

Detect Cycle in an undirected graph using BFS:

```cpp
class Solution
{
private:
    bool detect(int src, vector<int> adj[], int vis[])
    {
        vis[src] = 1;
        // store <source node, parent node>
        queue<pair<int, int>> q;
        q.push({src, -1});
        // traverse until queue is not empty
        while (!q.empty())
        {
            int node = q.front().first;
            int parent = q.front().second;
            q.pop();

            // go to all adjacent nodes
            for (auto adjacentNode : adj[node])
            {
                // if adjacent node is unvisited
                if (!vis[adjacentNode])
                {
                    vis[adjacentNode] = 1;
                    q.push({adjacentNode, node});
                }
                // if adjacent node is visited and is not it's own parent node
                else if (parent != adjacentNode)
                {
                    // yes it is a cycle
                    return true;
                }
            }
        }
        // there's no cycle
        return false;
    }

public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[])
    {
        // initialise them as unvisited
        int vis[V] = {0};
```

```
        for (int i = 0; i < V; i++)
        {
            if (!vis[i])
            {
                if (detect(i, adj, vis))
                    return true;
            }
        }
        return false;
    }
};
```

*Time complexity: O(N*2E)* *Space complexity: O(n)*

Detect Cycle in an undirected graph using DFS:

```
class Solution {
  private:
    bool dfs(int node, int parent, int vis[], vector<int> adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vis, adj) == true)
                    return true;
            }
            // visited node but not a parent node
            else if(adjacentNode != parent) return true;
        }
        return false;
    }
  public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
       int vis[V] = {0};
       // for graph with connected components
       for(int i = 0;i<V;i++) {
           if(!vis[i]) {
```

```
                if(dfs(i, -1, vis, adj) == true) return true;
            }
        }
        return false;
    }
};
```

*Time complexity: O(N*2E)* *Space complexity: O(n)*

Distance of nearest cell having 1:

```
class Solution
{
    public:
    //Function to find the distance of nearest 1 in the grid for each cell.
    vector<vector<int>>nearest(vector<vector<int>>grid)
    {
        int n = grid.size();
        int m = grid[0].size();
        // visited and distance matrix
        vector<vector<int>> vis(n, vector<int>(m, 0));
        vector<vector<int>> dist(n, vector<int>(m, 0));
        // <coordinates, steps>
        queue<pair<pair<int,int>, int>> q;
        // traverse the matrix
        for(int i = 0;i<n;i++) {
            for(int j = 0;j<m;j++) {
                // start BFS if cell contains 1
                if(grid[i][j] == 1) {
                    q.push({{i,j}, 0});
                    vis[i][j] = 1;
                }
                else {
                    // mark unvisited
                    vis[i][j] = 0;
                }
            }
        }

        int delrow[] = {-1, 0, +1, 0};
```

```
        int delcol[] = {0, +1, 0, -1};

        // traverse till queue becomes empty
        while(!q.empty()) {
            int row = q.front().first.first;
            int col = q.front().first.second;
            int steps = q.front().second;
            q.pop();
            dist[row][col] = steps;
            // for all 4 neighbours
            for(int i = 0;i<4;i++) {
                int nrow = row + delrow[i];
                int ncol = col + delcol[i];
                // check for valid unvisited cell
                if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m
                && vis[nrow][ncol] == 0) {
                    vis[nrow][ncol] = 1;
                    q.push({{nrow, ncol}, steps+1});
                }
            }
        }
        // return distance matrix
        return dist;
    }
};
```

*Time complexity: O(n*m)* *Space complexity: O(n*m)*

Replace o's with X's:

```
class Solution
{
private:
    void dfs(int row, int col, vector<vector<int>> &vis,
             vector<vector<char>> &mat, int delrow[], int delcol[])
    {
        vis[row][col] = 1;
        int n = mat.size();
```

```
        int m = mat[0].size();

        // check for top, right, bottom, left
        for (int i = 0; i < 4; i++)
        {
            int nrow = row + delrow[i];
            int ncol = col + delcol[i];
            // check for valid coordinates and unvisited Os
            if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
!vis[nrow][ncol] && mat[nrow][ncol] == 'O')
            {
                dfs(nrow, ncol, vis, mat, delrow, delcol);
            }
        }
    }

public:
    vector<vector<char>> fill(int n, int m,
                              vector<vector<char>> mat)
    {
        int delrow[] = {-1, 0, +1, 0};
        int delcol[] = {0, 1, 0, -1};
        vector<vector<int>> vis(n, vector<int>(m, 0));
        // traverse first row and last row
        for (int j = 0; j < m; j++)
        {
            // check for unvisited Os in the boundary rows
            // first row
            if (!vis[0][j] && mat[0][j] == 'O')
            {
                dfs(0, j, vis, mat, delrow, delcol);
            }

            // last row
            if (!vis[n - 1][j] && mat[n - 1][j] == 'O')
            {
                dfs(n - 1, j, vis, mat, delrow, delcol);
            }
        }

        for (int i = 0; i < n; i++)
        {
            // check for unvisited Os in the boundary columns
            // first column
            if (!vis[i][0] && mat[i][0] == 'O')
```

```
            {
                dfs(i, 0, vis, mat, delrow, delcol);
            }

            // last column
            if (!vis[i][m - 1] && mat[i][m - 1] == 'O')
            {
                dfs(i, m - 1, vis, mat, delrow, delcol);
            }
        }

        // if unvisited O then convert to X
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                if (!vis[i][j] && mat[i][j] == 'O')
                    mat[i][j] = 'X';
            }
        }

        return mat;
    }
};
```

*Time complexity: O(n*m)* *Space complexity: O(n*m)*

Number of enclaves:

```
class Solution
{
public:
    int numberOfEnclaves(vector<vector<int>> &grid)
    {
        queue<pair<int, int>> q;
        int n = grid.size();
        int m = grid[0].size();
        int vis[n][m] = {0};
```

```
        // traverse boundary elements
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                // first row, first col, last row, last col
                if (i == 0 || j == 0 || i == n - 1 || j == m - 1)
                {
                    // if it is a land then store it in queue
                    if (grid[i][j] == 1)
                    {
                        q.push({i, j});
                        vis[i][j] = 1;
                    }
                }
            }
        }

        int delrow[] = {-1, 0, +1, 0};
        int delcol[] = {0, +1, +0, -1};

        while (!q.empty())
        {
            int row = q.front().first;
            int col = q.front().second;
            q.pop();

            // traverses all 4 directions
            for (int i = 0; i < 4; i++)
            {
                int nrow = row + delrow[i];
                int ncol = col + delcol[i];
                // check for valid coordinates and for land cell
                if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1)
                {
                    q.push({nrow, ncol});
                    vis[nrow][ncol] = 1;
                }
            }
        }

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

```
            for (int j = 0; j < m; j++)
            {
                // check for unvisited land cell
                if (grid[i][j] == 1 & vis[i][j] == 0)
                    cnt++;
            }
        }
        return cnt;
    }
};
```

*Time complexity: O(n*m)* *Space complexity: O(n*m)*

Number of distinct Islands:

```
class Solution
{
private:
    void dfs(int row, int col, vector < vector<int> & vis, vector<vector<int>>
&grid, vector<pair<int, int>> &vec, int row0, int col0)
    {
        vis[row][col] = 1;
        vec.push_back({row - row0, col - col0});
        int n = grid.size();
        int m = grid[0].size();
        int delrow[] = {-1, 0, 1, 0};
        int delcol[] = {0, -1, 0, 1};
        for (int i = 0; i < 4; i++)
        {
            int nrow = row + delrow[i];
            int ncol = ncol + delcol[i];
            if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
!vis[nrow][ncol] && grid[nrow][ncol] == 1)
            {
                dfs(nrow, ncol, vis, grid, vec, row0, col0);
            }
        }
    }
```

```
public:
    int numOfDistinctIslands(vector<vector<int>> &grid)
    {
        int n = grid.size();
        int m = grid[0].size();
        vector < vector<int> vis(n, vector<int>(m, 0));
        set<vector<pair<int, int>>> st;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                if (!vis[i][j] && grid[i][j] == 1)
                {
                    vector<pair<int, int>> vec;
                    dfs(i, j, grid, vis, vec, i, j);
                    st.insert(vec);
                }
            }
        }
        return st.size();
    }
};
```

*Time complexity: O(n*m)* *Space complexity: O(n*m)*

Bipartite Graph using BFS:

```
class Solution
{
    // colors a component
private:
    bool check(int start, int V, vector<int> adj[], int color[])
    {
        queue<int> q;
        q.push(start);
        color[start] = 0;
        while (!q.empty())
        {
```

```
            int node = q.front();
            q.pop();

            for (auto it : adj[node])
            {
                // if the adjacent node is yet not colored
                // you will give the opposite color of the node
                if (color[it] == -1)
                {

                    color[it] = !color[node];
                    q.push(it);
                }
                // is the adjacent guy having the same color
                // someone did color it on some other path
                else if (color[it] == color[node])
                {
                    return false;
                }
            }
        }
        return true;
    }

public:
    bool isBipartite(int V, vector<int> adj[])
    {
        int color[V];
        for (int i = 0; i < V; i++)
            color[i] = -1;

        for (int i = 0; i < V; i++)
        {
            // if not coloured
            if (color[i] == -1)
            {
                if (check(i, V, adj, color) == false)
                {
                    return false;
                }
            }
        }
        return true;
    }
};
```

Time complexity: O(v+2E) *Space complexity: O(v)*

Bipartite Graph using DFS:

```
class Solution
{
private:
    bool dfs(int node, int col, int color[], vector<int> adj[])
    {
        color[node] = col;

        // traverse adjacent nodes
        for (auto it : adj[node])
        {
            // if uncoloured
            if (color[it] == -1)
            {
                if (dfs(it, !col, color, adj) == false)
                    return false;
            }
            // if previously coloured and have the same colour
            else if (color[it] == col)
            {
                return false;
            }
        }

        return true;
    }

public:
    bool isBipartite(int V, vector<int> adj[])
    {
        int color[V];
```

```
        for (int i = 0; i < V; i++)
            color[i] = -1;

        // for connected components
        for (int i = 0; i < V; i++)
        {
            if (color[i] == -1)
            {
                if (dfs(i, 0, color, adj) == false)
                    return false;
            }
        }
        return true;
    }
};
```

Time complexity: O(v+2E) *Space complexity: O(v)*

Detect Cycle in directed Graph using DFS:

```
class Solution
{
private:
    bool dfsCheck(int node, vector<int> adj[], int vis[], int pathVis[])
    {
        vis[node] = 1;
        pathVis[node] = 1;

        // traverse for adjacent nodes
        for (auto it : adj[node])
        {
            // when the node is not visited
            if (!vis[it])
            {
                if (dfsCheck(it, adj, vis, pathVis) == true)
                    return true;
```

```
            }
            // if the node has been previously visited
            // but it has to be visited on the same path
            else if (pathVis[it])
            {
                return true;
            }
        }

        pathVis[node] = 0;
        return false;
    }

public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[])
    {
        int vis[V] = {0};
        int pathVis[V] = {0};

        for (int i = 0; i < V; i++)
        {
            if (!vis[i])
            {
                if (dfsCheck(i, adj, vis, pathVis) == true)
                    return true;
            }
        }
        return false;
    }
};
```

Time complexity: O(v+2E) *Space complexity: O(v)*

Eventual safe states using DFS:

```
class Solution
{
private:
```

```
    bool dfsCheck(int node, vector<int> adj[], int vis[],
                  int pathVis[],
                  int check[])
    {
        vis[node] = 1;
        pathVis[node] = 1;
        check[node] = 0;
        // traverse for adjacent nodes
        for (auto it : adj[node])
        {
            // when the node is not visited
            if (!vis[it])
            {
                if (dfsCheck(it, adj, vis, pathVis, check) == true)
                {
                    check[node] = 0;
                    return true;
                }
            }
            // if the node has been previously visited
            // but it has to be visited on the same path
            else if (pathVis[it])
            {
                check[node] = 0;
                return true;
            }
        }
        check[node] = 1;
        pathVis[node] = 0;
        return false;
    }

public:
    vector<int> eventualSafeNodes(int V, vector<int> adj[])
    {
        int vis[V] = {0};
        int pathVis[V] = {0};
        int check[V] = {0};
        vector<int> safeNodes;
        for (int i = 0; i < V; i++)
        {
            if (!vis[i])
            {
                dfsCheck(i, adj, vis, pathVis, check);
            }
```

```
        }
        for (int i = 0; i < V; i++)
        {
            if (check[i] == 1)
                safeNodes.push_back(i);
        }
        return safeNodes;
    }
};
```

Time complexity: O(v+E) *Space complexity: O(N)*

Topological sort using DFS:

```
class Solution {
private:
    void dfs(int node, int vis[], stack<int> &st,
             vector<int> adj[]) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            if (!vis[it]) dfs(it, vis, st, adj);
        }
        st.push(node);
    }
public:
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int vis[V] = {0};
        stack<int> st;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(i, vis, st, adj);
            }
        }

        vector<int> ans;
```

```
        while (!st.empty()) {
            ans.push_back(st.top());
            st.pop();
        }
        return ans;
    }
};
```

Time complexity: O(v+E) *Space complexity: O(N)*

Kahn's algorithm | Topological sort using BFS:

```
class Solution
{
public:
    // Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++)
        {
            for (auto it : adj[i])
            {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++)
        {
            if (indegree[i] == 0)
            {
                q.push(i);
            }
        }
```

```
        vector<int> topo;
        while (!q.empty())
        {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node])
            {
                indegree[it]--;
                if (indegree[it] == 0)
                    q.push(it);
            }
        }

        return topo;
    }
};
```

Time complexity: O(v+E) *Space complexity: O(N)*

Detect cycle in directed graph using BFS:

```
class Solution
{
public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++)
        {
            for (auto it : adj[i])
            {
                indegree[it]++;
            }
        }
```

```
        queue<int> q;
        for (int i = 0; i < V; i++)
        {
            if (indegree[i] == 0)
            {
                q.push(i);
            }
        }

        int cnt = 0;
        // o(v + e)
        while (!q.empty())
        {
            int node = q.front();
            q.pop();
            cnt++;
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node])
            {
                indegree[it]--;
                if (indegree[it] == 0)
                    q.push(it);
            }
        }

        if (cnt == V)
            return false;
        return true;
    }
};
```

Time complexity: O(v+E) *Space complexity: O(N)*

Eventual safe states using BFS-Topological sort:

```cpp
class Solution
{
public:
    vector<int> eventualSafeNodes(int V, vector<int> adj[])
    {
        vector<int> adjRev[V];
        int indegree[V] = {0};
        for (int i = 0; i < V; i++)
        {
            // i -> it
            // it -> i
            for (auto it : adj[i])
            {
                adjRev[it].push_back(i);
                indegree[i]++;
            }
        }
        queue<int> q;
        vector<int> safeNodes;
        for (int i = 0; i < V; i++)
        {
            if (indegree[i] == 0)
            {
                q.push(i);
            }
        }

        while (!q.empty())
        {
            int node = q.front();
            q.pop();
            safeNodes.push_back(node);
            for (auto it : adjRev[node])
            {
                indegree[it]--;
                if (indegree[it] == 0)
                    q.push(it);
            }
        }

        sort(safeNodes.begin(), safeNodes.end());
        return safeNodes;
    }
```

```
};
```

*Time complexity: O(v+E)+O(n*logn)* *Space complexity: O(N)*

Alien Dictionary using BFS-Topological sort:

```
class Solution {
    // works for multiple components
private:
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        return topo;
```

```
    }
public:
    string findOrder(string dict[], int N, int K) {
        vector<int>adj[K];
        for (int i = 0; i < N - 1; i++) {
            string s1 = dict[i];
            string s2 = dict[i + 1];
            int len = min(s1.size(), s2.size());
            for (int ptr = 0; ptr < len; ptr++) {
                if (s1[ptr] != s2[ptr]) {
                    adj[s1[ptr] - 'a'].push_back(s2[ptr] - 'a');
                    break;
                }
            }
        }

        vector<int> topo = topoSort(K, adj);
        string ans = "";
        for (auto it : topo) {
            ans = ans + char(it + 'a');
        }
        return ans;
    }
};
```

*Time complexity: O(K+E)+O(N*len)* *Space complexity: O(N)*

Shortest Path in DAG using BFS-Topological sort:

```
class Solution {
  private:
    void topoSort(int node, vector < pair < int, int >> adj[],
      int vis[], stack < int > & st) {
      //This is the function to implement Topological sort.
      vis[node] = 1;
      for (auto it: adj[node]) {
        int v = it.first;
        if (!vis[v]) {
```

```
          topoSort(v, adj, vis, st);
        }
      }
      st.push(node);
    }
  public:
    vector < int > shortestPath(int N, int M, vector < vector < int >> & edges) {

      //We create a graph first in the form of an adjacency list.
      vector < pair < int, int >> adj[N];
      for (int i = 0; i < M; i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        int wt = edges[i][2];
        adj[u].push_back({v, wt});
      }
      // A visited array is created with initially
      // all the nodes marked as unvisited (0).
      int vis[N] = {
        0
      };
      //Now, we perform topo sort using DFS technique
      //and store the result in the stack st.
      stack < int > st;
      for (int i = 0; i < N; i++) {
        if (!vis[i]) {
          topoSort(i, adj, vis, st);
        }
      }
      //Further, we declare a vector 'dist' in which we update the value of the
nodes'
      //distance from the source vertex after relaxation of a particular node.

      vector < int > dist(N);
      for (int i = 0; i < N; i++) {
        dist[i] = 1e9;
      }

      dist[0] = 0;
      while (!st.empty()) {
        int node = st.top();
        st.pop();

        for (auto it: adj[node]) {
          int v = it.first;
```

```
          int wt = it.second;

          if (dist[node] + wt < dist[v]) {
            dist[v] = wt + dist[node];
          }
        }
      }

      for (int i = 0; i < N; i++) {
        if (dist[i] == 1e9) dist[i] = -1;
      }
      return dist;
    }
};
```

Time complexity: O(N+M) *Space complexity: O(N+M)*

Shortest Path in undirected graph with unit distance using BFS:

```
class Solution
{
public:
    vector<int> shortestPath(vector<vector<int>> &edges, int N, int M, int src)
    {
        // Create an adjacency list of size N for storing the undirected graph.
        vector<int> adj[N];
        for (auto it : edges)
        {
            adj[it[0]].push_back(it[1]);
            adj[it[1]].push_back(it[0]);
        }

        // A dist array of size N initialised with a large number to
        // indicate that initially all the nodes are untraversed.

        int dist[N];
        for (int i = 0; i < N; i++)
            dist[i] = 1e9;
        // BFS Implementation.
```

```
        dist[src] = 0;
        queue<int> q;
        q.push(src);
        while (!q.empty())
        {
            int node = q.front();
            q.pop();
            for (auto it : adj[node])
            {
                if (dist[node] + 1 < dist[it])
                {
                    dist[it] = 1 + dist[node];
                    q.push(it);
                }
            }
        }
        // Updated shortest distances are stored in the resultant array 'ans'.
        // Unreachable nodes are marked as -1.
        vector<int> ans(N, -1);
        for (int i = 0; i < N; i++)
        {
            if (dist[i] != 1e9)
            {
                ans[i] = dist[i];
            }
        }
        return ans;
    }
};
```

Time complexity: O(N+M) *Space complexity: O(N+M)*

Word ladder-1

```
class Solution
{
public:
    int wordLadderLength(string startWord, string targetWord,
                         vector<string> &wordList)
    {
        // Creating a queue ds of type {word,transitions to reach 'word'}.
```

```
        queue<pair<string, int>> q;

        // BFS traversal with pushing values in queue
        // when after a transformation, a word is found in wordList.
        q.push({startWord, 1});

        // Push all values of wordList into a set
        // to make deletion from it easier and in less time complexity.
        unordered_set<string> st(wordList.begin(), wordList.end());
        st.erase(startWord);
        while (!q.empty())
        {
            string word = q.front().first;
            int steps = q.front().second;
            q.pop();

            // we return the steps as soon as
            // the first occurence of targetWord is found.
            if (word == targetWord)
                return steps;

            for (int i = 0; i < word.size(); i++)
            {
                // Now, replace each character of 'word' with char
                // from a-z then check if 'word' exists in wordList.
                char original = word[i];
                for (char ch = 'a'; ch <= 'z'; ch++)
                {
                    word[i] = ch;
                    // check if it exists in the set and push it in the queue.
                    if (st.find(word) != st.end())
                    {
                        st.erase(word);
                        q.push({word, steps + 1});
                    }
                }
                word[i] = original;
            }
        }
        // If there is no transformation sequence possible
        return 0;
    }
};
```

*Time complexity: O(N*M*26)* *Space complexity: O(N)*

Dijkstra's Algorithm using priority queue:

```
class Solution
{
public:
    // Function to find the shortest distance of all the vertices
    // from the source vertex S.
    vector<int> dijkstra(int V, vector<vector<int>> adj[], int S)
    {

        // Create a priority queue for storing the nodes as a pair {dist,node}
        // where dist is the distance from source to the node.
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

        // Initialising distTo list with a large number to
        // indicate the nodes are unvisited initially.
        // This list contains distance from source to the nodes.
        vector<int> distTo(V, INT_MAX);

        // Source initialised with dist=0.
        distTo[S] = 0;
        pq.push({0, S});

        // Now, pop the minimum distance node first from the min-heap
        // and traverse for all its adjacent nodes.
        while (!pq.empty())
        {
            int node = pq.top().second;
            int dis = pq.top().first;
            pq.pop();

            // Check for all adjacent nodes of the popped out
            // element whether the prev dist is larger than current or not.
            for (auto it : adj[node])
            {
                int v = it[0];
```

```
                int w = it[1];
                if (dis + w < distTo[v])
                {
                    distTo[v] = dis + w;

                    // If current distance is smaller,
                    // push it into the queue.
                    pq.push({dis + w, v});
                }
            }
        }
        // Return the list containing shortest distances
        // from source to all the nodes.
        return distTo;
    }
};
```

Time complexity: O(E log (V)) *Space complexity: O(E+V)*

Dijkstra's Algorithm using sets:

```
class Solution
{
public:
    // Function to find the shortest distance of all the vertices
    // from the source vertex S.
    vector<int> dijkstra(int V, vector<vector<int>> adj[], int S)
    {
        // Create a set ds for storing the nodes as a pair {dist,node}
        // where dist is the distance from source to the node.
        // set stores the nodes in ascending order of the distances
        set<pair<int, int>> st;

        // Initialising dist list with a large number to
        // indicate the nodes are unvisited initially.
        // This list contains distance from source to the nodes.
        vector<int> dist(V, 1e9);

        st.insert({0, S});
```

```
        // Source initialised with dist=0
        dist[S] = 0;

        // Now, erase the minimum distance node first from the set
        // and traverse for all its adjacent nodes.
        while (!st.empty())
        {
            auto it = *(st.begin());
            int node = it.second;
            int dis = it.first;
            st.erase(it);

            // Check for all adjacent nodes of the erased
            // element whether the prev dist is larger than current or not.
            for (auto it : adj[node])
            {
                int adjNode = it[0];
                int edgW = it[1];

                if (dis + edgW < dist[adjNode])
                {
                    // erase if it was visited previously at
                    // a greater cost.
                    if (dist[adjNode] != 1e9)
                        st.erase({dist[adjNode], adjNode});

                    // If current distance is smaller,
                    // push it into the queue
                    dist[adjNode] = dis + edgW;
                    st.insert({dist[adjNode], adjNode});
                }
            }
        }
        // Return the list containing shortest distances
        // from source to all the nodes.
        return dist;
    }
};
```

Time complexity: O(E log (V)) *Space complexity: O(E+V)*

Shortest Path using Dijkstra's Algorithm:

```cpp
class Solution
{
public:
    vector<int> shortestPath(int n, int m, vector<vector<int>> &edges)
    {
        // Create an adjacency list of pairs of the form node1 -> {node2, edge
weight}
        // where the edge weight is the weight of the edge from node1 to node2.
        vector<pair<int, int>> adj[n + 1];
        for (auto it : edges)
        {
            adj[it[0]].push_back({it[1], it[2]});
            adj[it[1]].push_back({it[0], it[2]});
        }
        // Create a priority queue for storing the nodes along with distances
        // in the form of a pair { dist, node }.
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

        // Create a dist array for storing the updated distances and a parent
array
        // for storing the nodes from where the current nodes represented by
indices of
        // the parent array came from.
        vector<int> dist(n + 1, 1e9), parent(n + 1);
        for (int i = 1; i <= n; i++)
            parent[i] = i;

        dist[1] = 0;

        // Push the source node to the queue.
        pq.push({0, 1});
        while (!pq.empty())
        {
            // Topmost element of the priority queue is with minimum distance
value.
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int dis = it.first;

            // Iterate through the adjacent nodes of the current popped node.
            for (auto it : adj[node])
```

```
            {
                int adjNode = it.first;
                int edW = it.second;

                // Check if the previously stored distance value is
                // greater than the current computed value or not,
                // if yes then update the distance value.
                if (dis + edW < dist[adjNode])
                {
                    dist[adjNode] = dis + edW;
                    pq.push({dis + edW, adjNode});

                    // Update the parent of the adjNode to the recent
                    // node where it came from.
                    parent[adjNode] = node;
                }
            }
        }

        // If distance to a node could not be found, return an array containing -
1.
        if (dist[n] == 1e9)
            return {-1};

        // Store the final path in the 'path' array.
        vector<int> path;
        int node = n;

        // Iterate backwards from destination to source through the parent array.
        while (parent[node] != node)
        {
            path.push_back(node);
            node = parent[node];
        }
        path.push_back(1);

        // Since the path stored is in a reverse order, we reverse the array
        // to get the final answer and then return the array.
        reverse(path.begin(), path.end());
        return path;
    }
};
```

Time complexity: O(E log (V)) *Space complexity: O(E+V)*

Shortest distance in Binary Maze:

```
class Solution
{
public:
    int shortestPath(vector<vector<int>> &grid, pair<int, int> source,
                     pair<int, int> destination)
    {
        // Edge Case: if the source is only the destination.
        if (source.first == destination.first &&
            source.second == destination.second)
            return 0;

        // Create a queue for storing cells with their distances from source
        // in the form {dist,{cell coordinates pair}}.
        queue<pair<int, pair<int, int>>> q;
        int n = grid.size();
        int m = grid[0].size();

        // Create a distance matrix with initially all the cells marked as
        // unvisited and the source cell as 0.
        vector<vector<int>> dist(n, vector<int>(m, 1e9));
        dist[source.first][source.second] = 0;
        q.push({0, {source.first, source.second}});

        // The following delta rows and delts columns array are created such that
        // each index represents each adjacent node that a cell may have
        // in a direction.
        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};

        // Iterate through the maze by popping the elements out of the queue
        // and pushing whenever a shorter distance to a cell is found.
        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int dis = it.first;
            int r = it.second.first;
            int c = it.second.second;
```

```
            // Through this loop, we check the 4 direction adjacent nodes
            // for a shorter path to destination.
            for (int i = 0; i < 4; i++)
            {
                int newr = r + dr[i];
                int newc = c + dc[i];

                // Checking the validity of the cell and updating if dist is
shorter.
                if (newr >= 0 && newr < n && newc >= 0 && newc < m &&
grid[newr][newc]
                == 1 && dis + 1 < dist[newr][newc])
                {
                    dist[newr][newc] = 1 + dis;

                    // Return the distance until the point when
                    // we encounter the destination cell.
                    if (newr == destination.first &&
                        newc == destination.second)
                        return dis + 1;
                    q.push({1 + dis, {newr, newc}});
                }
            }
        }
        // If no path is found from source to destination.
        return -1;
    }
};
```

*Time complexity: O(4*n*m)* *Space complexity: O(n*m)*

Path with minimum effort:

```
class Solution
{
public:
    int MinimumEffort(vector<vector<int>> &heights)
```

```
{

    // Create a priority queue containing pairs of cells
    // and their respective distance from the source cell in the
    // form {diff, {row of cell, col of cell}}.
    priority_queue<pair<int, pair<int, int>>,
                   vector<pair<int, pair<int, int>>>,
                   greater<pair<int, pair<int, int>>>>
        pq;

    int n = heights.size();
    int m = heights[0].size();

    // Create a distance matrix with initially all the cells marked as
    // unvisited and the dist for source cell (0,0) as 0.
    vector<vector<int>> dist(n, vector<int>(m, 1e9));
    dist[0][0] = 0;
    pq.push({0, {0, 0}});

    // The following delta rows and delts columns array are created such that
    // each index represents each adjacent node that a cell may have
    // in a direction.
    int dr[] = {-1, 0, 1, 0};
    int dc[] = {0, 1, 0, -1};

    // Iterate through the matrix by popping the elements out of the queue
    // and pushing whenever a shorter distance to a cell is found.
    while (!pq.empty())
    {
        auto it = pq.top();
        pq.pop();
        int diff = it.first;
        int row = it.second.first;
        int col = it.second.second;

        // Check if we have reached the destination cell,
        // return the current value of difference (which will be min).
        if (row == n - 1 && col == m - 1)
            return diff;

        for (int i = 0; i < 4; i++)
        {
            // row - 1, col
            // row, col + 1
            // row - 1, col
```

```
                // row, col - 1
                int newr = row + dr[i];
                int newc = col + dc[i];

                // Checking validity of the cell.
                if (newr >= 0 && newc >= 0 && newr < n && newc < m)
                {
                    // Effort can be calculated as the max value of differences
                    // between the heights of the node and its adjacent nodes.
                    int newEffort = max(abs(heights[row][col] -
heights[newr][newc]), diff);

                    // If the calculated effort is less than the prev value
                    // we update as we need the min effort.
                    if (newEffort < dist[newr][newc])
                    {
                        dist[newr][newc] = newEffort;
                        pq.push({newEffort, {newr, newc}});
                    }
                }
            }
        }
        return 0; // if unreachable
    }
};
```

*Time complexity: O(4*n*m*log(n*m))* *Space complexity: O(n*m)*

Cheapest Flight within k stops:

```
  class Solution
{
public:
    int CheapestFLight(int n, vector<vector<int>> &flights,
                       int src, int dst, int K)
    {
        // Create the adjacency list to depict airports and flights in
        // the form of a graph.
```

```
        vector<pair<int, int>> adj[n];
        for (auto it : flights)
        {
            adj[it[0]].push_back({it[1], it[2]});
        }

        // Create a queue which stores the node and their distances from the
        // source in the form of {stops, {node, dist}} with 'stops' indicating
        // the no. of nodes between src and current node.
        queue<pair<int, pair<int, int>>> q;

        q.push({0, {src, 0}});

        // Distance array to store the updated distances from the source.
        vector<int> dist(n, 1e9);
        dist[src] = 0;

        // Iterate through the graph using a queue like in Dijkstra with
        // popping out the element with min stops first.
        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int stops = it.first;
            int node = it.second.first;
            int cost = it.second.second;

            // We stop the process as soon as the limit for the stops reaches.
            if (stops > K)
                continue;
            for (auto iter : adj[node])
            {
                int adjNode = iter.first;
                int edW = iter.second;

                // We only update the queue if the new calculated dist is
                // less than the prev and the stops are also within limits.
                if (cost + edW < dist[adjNode] && stops <= K)
                {
                    dist[adjNode] = cost + edW;
                    q.push({stops + 1, {adjNode, cost + edW}});
                }
            }
        }
        // If the destination node is unreachable return '-1'
```

```
        // else return the calculated dist from src to dst.
        if (dist[dst] == 1e9)
            return -1;
        return dist[dst];
    }
};
```

Time complexity: O(n) *Space complexity: O(e+v)*

Cheapest Flight within k stops:

```
class Solution
{
public:
    int minimumMultiplications(vector<int> &arr,
                               int start, int end)
    {
        // Create a queue for storing the numbers as a result of multiplication
        // of the numbers in the array and the start number.
        queue<pair<int, int>> q;
        q.push({start, 0});

        // Create a dist array to store the no. of multiplications to reach
        // a particular number from the start number.
        vector<int> dist(100000, 1e9);
        dist[start] = 0;
        int mod = 100000;

        // Multiply the start no. with each of numbers in the arr
        // until we get the end no.
        while (!q.empty())
        {
            int node = q.front().first;
            int steps = q.front().second;
            q.pop();
```

```
            for (auto it : arr)
            {
                int num = (it * node) % mod;

                // If the no. of multiplications are less than before
                // in order to reach a number, we update the dist array.
                if (steps + 1 < dist[num])
                {
                    dist[num] = steps + 1;

                    // Whenever we reach the end number
                    // return the calculated steps
                    if (num == end)
                        return steps + 1;
                    q.push({num, steps + 1});
                }
            }
        }
        // If the end no. is unattainable.
        return -1;
    }
};
```

*Time complexity: O(100000*n)* *Space complexity: O(100000*n)*

Number of ways to arrive at destination:

```
class Solution
{
public:
    int countPaths(int n, vector<vector<int>> &roads)
    {
        // Creating an adjacency list for the given graph.
```

```cpp
        vector<pair<int, int>> adj[n];
        for (auto it : roads)
        {
            adj[it[0]].push_back({it[1], it[2]});
            adj[it[1]].push_back({it[0], it[2]});
        }

        // Defining a priority queue (min heap).
        priority_queue<pair<int, int>,
                       vector<pair<int, int>>, greater<pair<int, int>>>
            pq;

        // Initializing the dist array and the ways array
        // along with their first indices.
        vector<int> dist(n, INT_MAX), ways(n, 0);
        dist[0] = 0;
        ways[0] = 1;
        pq.push({0, 0});

        // Define modulo value
        int mod = (int)(1e9 + 7);

        // Iterate through the graph with the help of priority queue
        // just as we do in Dijkstra's Algorithm.
        while (!pq.empty())
        {
            int dis = pq.top().first;
            int node = pq.top().second;
            pq.pop();

            for (auto it : adj[node])
            {
                int adjNode = it.first;
                int edW = it.second;

                // This 'if' condition signifies that this is the first
                // time we're coming with this short distance, so we push
                // in PQ and keep the no. of ways the same.
                if (dis + edW < dist[adjNode])
                {
                    dist[adjNode] = dis + edW;
                    pq.push({dis + edW, adjNode});
                    ways[adjNode] = ways[node];
                }
```

```
                // If we again encounter a node with the same short distance
                // as before, we simply increment the no. of ways.
                else if (dis + edW == dist[adjNode])
                {
                    ways[adjNode] = (ways[adjNode] + ways[node]) % mod;
                }
            }
        }
        // Finally, we return the no. of ways to reach
        // (n-1)th node modulo 10^9+7.
        return ways[n - 1] % mod;
    }
};
```

*Time complexity: O(E * Log V)* *Space complexity: O(N)*

Bellman ford Algorithm:

```
    vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S) {
        vector<int> dist(V, 1e8);
        dist[S] = 0;
        for (int i = 0; i < V - 1; i++) {
            for (auto it : edges) {
                int u = it[0];
                int v = it[1];
                int wt = it[2];
                if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                    dist[v] = dist[u] + wt;
                }
            }
        }
        // Nth relaxation to check negative cycle
        for (auto it : edges) {
            int u = it[0];
            int v = it[1];
            int wt = it[2];
            if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                return { -1};
            }
        }
```

```
        return dist;
    }
```

*Time complexity: O(V*E)* *Space complexity: O(V)*

Floyd Warshall Algorithm:

```
void shortest_distance(vector<vector<int>>&matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == -1) {
                    matrix[i][j] = 1e9;
                }
                if (i == j) matrix[i][j] = 0;
            }
        }

        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    matrix[i][j] = min(matrix[i][j],
                                       matrix[i][k] + matrix[k][j]);
                }
            }
        }




        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1e9) {
                    matrix[i][j] = -1;
                }
            }
        }
    }
```

Time complexity: O(V^3) *Space complexity: O(V^2)*

Find city with smallest number of neighbors at threshold distance:

```
int findCity(int n, int m, vector<vector<int>>& edges,
                 int distanceThreshold) {
        vector<vector<int>> dist(n, vector<int> (n, INT_MAX));
        for (auto it : edges) {
            dist[it[0]][it[1]] = it[2];
            dist[it[1]][it[0]] = it[2];
        }
        for (int i = 0; i < n; i++) dist[i][i] = 0;
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (dist[i][k] == INT_MAX || dist[k][j] == INT_MAX)
                        continue;
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }

        int cntCity = n;
        int cityNo = -1;
        for (int city = 0; city < n; city++) {
            int cnt = 0;
            for (int adjCity = 0; adjCity < n; adjCity++) {
                if (dist[city][adjCity] <= distanceThreshold)
                    cnt++;
            }

            if (cnt <= cntCity) {
                cntCity = cnt;
                cityNo = city;
            }
        }
        return cityNo;

    }
```

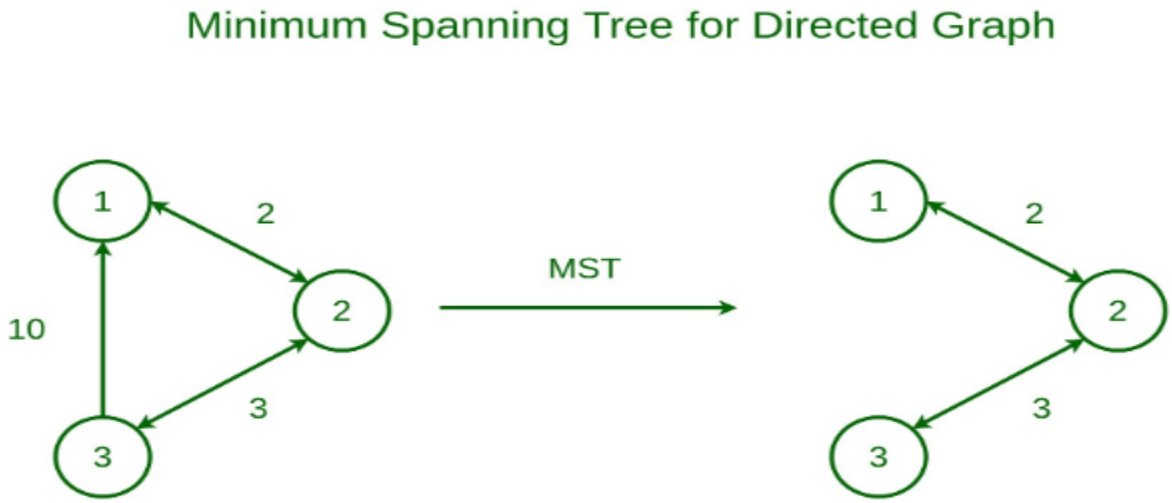
Time complexity: O(V^3) *Space complexity: O(V^2)*

Spanning Tree:

A spanning tree is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph.

Minimum Spanning Tree:

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum weight among all the possible spanning trees.



Find the sum of edges of MST using Prime's Algorithm:

```
int spanningTree(int V, vector<vector<int>> adj[])
{
    priority_queue<pair<int, int>,
                   vector<pair<int, int> >, greater<pair<int, int>>> pq;
```

```
        vector<int> vis(V, 0);
        // {wt, node}
        pq.push({0, 0});
        int sum = 0;
        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;

            if (vis[node] == 1) continue;
            // add it to the mst
            vis[node] = 1;
            sum += wt;
            for (auto it : adj[node]) {
                int adjNode = it[0];
                int edW = it[1];
                if (!vis[adjNode]) {
                    pq.push({edW, adjNode});
                }
            }
        }
        return sum;
    }
```

Time complexity: O(Elog E) *Space complexity: O(E)+O(V)*

Disjoint Set:

A disjoint-set data structure, also known as a union-find data structure, is a data structure that keeps track of a collection of disjoint (non-overlapping) sets. It's often used to solve problems involving the partitioning of a set into subsets and to efficiently determine whether two elements are in the same set or not.

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
```

```
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
int main() {
```

```
    DisjointSet ds(7);
    ds.unionBySize(1, 2);
    ds.unionBySize(2, 3);
    ds.unionBySize(4, 5);
    ds.unionBySize(6, 7);
    ds.unionBySize(5, 6);
    // if 3 and 7 same or not
    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";

    ds.unionBySize(3, 7);

    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";
    return 0;
}
```

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

Kruskal Algorithm:

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
```

```
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        // 1 - 2 wt = 5
        /// 1 - > (2, 5)
        // 2 -> (1, 5)

        // 5, 1, 2
        // 5, 2, 1
```

```
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWt;
    }
};
```

*Time complexity: O(N+E)+O(E log E) +O(E*4*2)* *Space complexity: O(N) + O(N) +O(E)*

Minimum Number of operations to make network connected:

```
class DisjointSet {
public:
    vector<int> rank, parent, size;
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
```

```
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution {
public:
    int Solve(int n, vector<vector<int>>& edge) {
        DisjointSet ds(n);
        int cntExtras = 0;
        for (auto it : edge) {
```

```
            int u = it[0];
            int v = it[1];
            if (ds.findUPar(u) == ds.findUPar(v)) {
                cntExtras++;
            }
            else {
                ds.unionBySize(u, v);
            }
        }
        int cntC = 0;
        for (int i = 0; i < n; i++) {
            if (ds.parent[i] == i) cntC++;
        }
        int ans = cntC - 1;
        if (cntExtras >= ans) return ans;
        return -1;
    }
};
```

*Time complexity: O(E*4@)+O(N*4@)* *Space complexity: O(N)*

Account Merge:

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }
```

```
    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>> &details) {
        int n = details.size();
        DisjointSet ds(n);
        sort(details.begin(), details.end());
        unordered_map<string, int> mapMailNode;
        for (int i = 0; i < n; i++) {
            for (int j = 1; j < details[i].size(); j++) {
                string mail = details[i][j];
                if (mapMailNode.find(mail) == mapMailNode.end()) {
                    mapMailNode[mail] = i;
                }
```

```
                else {
                    ds.unionBySize(i, mapMailNode[mail]);
                }
            }
        }

        vector<string> mergedMail[n];
        for (auto it : mapMailNode) {
            string mail = it.first;
            int node = ds.findUPar(it.second);
            mergedMail[node].push_back(mail);
        }

        vector<vector<string>> ans;

        for (int i = 0; i < n; i++) {
            if (mergedMail[i].size() == 0) continue;
            sort(mergedMail[i].begin(), mergedMail[i].end());
            vector<string> temp;
            temp.push_back(details[i][0]);
            for (auto it : mergedMail[i]) {
                temp.push_back(it);
            }
            ans.push_back(temp);
        }
        sort(ans.begin(), ans.end());
        return ans;
    }
```

*Time complexity: O(N+E) + O(E*4a) + O(N*(ElogE + E))* *Space complexity: O(N)*

Number of islands-II:

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
```

```
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution {
private:
```

```
    bool isValid(int adjr, int adjc, int n, int m) {
        return adjr >= 0 && adjr < n && adjc >= 0 && adjc < m;
    }
public:
    vector<int> numOfIslands(int n, int m,
                             vector<vector<int>> &operators) {
        DisjointSet ds(n * m);
        int vis[n][m];
        memset(vis, 0, sizeof vis);
        int cnt = 0;
        vector<int> ans;
        for (auto it : operators) {
            int row = it[0];
            int col = it[1];
            if (vis[row][col] == 1) {
                ans.push_back(cnt);
                continue;
            }
            vis[row][col] = 1;
            cnt++;
            // row - 1, col
            // row , col + 1
            // row + 1, col
            // row, col - 1;
            int dr[] = { -1, 0, 1, 0};
            int dc[] = {0, 1, 0, -1};
            for (int ind = 0; ind < 4; ind++) {
                int adjr = row + dr[ind];
                int adjc = col + dc[ind];
                if (isValid(adjr, adjc, n, m)) {
                    if (vis[adjr][adjc] == 1) {
                        int nodeNo = row * m + col;
                        int adjNodeNo = adjr * m + adjc;
                        if (ds.findUPar(nodeNo) != ds.findUPar(adjNodeNo)) {
                            cnt--;
                            ds.unionBySize(nodeNo, adjNodeNo);
                        }
                    }
                }
            }
            ans.push_back(cnt);
        }
        return ans;
    }
```

Time complexity: O(Q) *Space complexity: O(Q) + O(NM) + O(NM)*

Making a large island:

```
class DisjointSet {

public:
    vector<int> rank, parent, size;
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
```

```
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution {
private:
    bool isValid(int newr, int newc, int n) {
        return newr >= 0 && newr < n && newc >= 0 && newc < n;
    }
public:
    int MaxConnection(vector<vector<int>>& grid) {
        int n = grid.size();
        DisjointSet ds(n * n);
        // step - 1
        for (int row = 0; row < n ; row++) {
            for (int col = 0; col < n ; col++) {
                if (grid[row][col] == 0) continue;
                int dr[] = { -1, 0, 1, 0};
                int dc[] = {0, -1, 0, 1};
                for (int ind = 0; ind < 4; ind++) {
                    int newr = row + dr[ind];
                    int newc = col + dc[ind];
                    if (isValid(newr, newc, n) && grid[newr][newc] == 1) {
                        int nodeNo = row * n + col;
                        int adjNodeNo = newr * n + newc;
                        ds.unionBySize(nodeNo, adjNodeNo);
                    }
                }
            }
        }
        // step 2
        int mx = 0;
        for (int row = 0; row < n; row++) {
            for (int col = 0; col < n; col++) {
                if (grid[row][col] == 1) continue;
                int dr[] = { -1, 0, 1, 0};
                int dc[] = {0, -1, 0, 1};
                set<int> components;
                for (int ind = 0; ind < 4; ind++) {
                    int newr = row + dr[ind];
```

```
                    int newc = col + dc[ind];
                    if (isValid(newr, newc, n)) {
                        if (grid[newr][newc] == 1) {
                            components.insert(ds.findUPar(newr * n + newc));
                        }
                    }
                }
                int sizeTotal = 0;
                for (auto it : components) {
                    sizeTotal += ds.size[it];
                }
                mx = max(mx, sizeTotal + 1);
            }
        }
        for (int cellNo = 0; cellNo < n * n; cellNo++) {
            mx = max(mx, ds.size[ds.findUPar(cellNo)]);
        }
        return mx;
    }
};
```

Time complexity: O(N^2) *Space complexity: O(N^2)*

Most stones removed with same row or column:

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

class DisjointSet
{
    vector<int> rank, parent, size;

public:
    DisjointSet(int n)
    {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++)
```

```
        {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node)
    {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v)
    {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v)
            return;
        if (rank[ulp_u] < rank[ulp_v])
        {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u])
        {
            parent[ulp_v] = ulp_u;
        }
        else
        {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v)
    {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v)
            return;
        if (size[ulp_u] < size[ulp_v])
        {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
```

```
        else
        {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution
{
public:
    int maxRemove(vector<vector<int>> &stones, int n)
    {
        int maxRow = 0;
        int maxCol = 0;
        for (auto it : stones)
        {

            maxRow = max(maxRow, it[0]);
            maxCol = max(maxCol, it[1]);
        }

        DisjointSet ds(maxRow + maxCol + 1);
        unordered_map<int, int> stoneNodes;
        for (auto it : stones)
        {
            int nodeRow = it[0];
            int nodeCol = it[1] + maxRow + 1;
            ds.unionBySize(nodeRow, nodeCol);
            stoneNodes[nodeRow] = 1;
            stoneNodes[nodeCol] = 1;
        }

        int cnt = 0;
        for (auto it : stoneNodes)
        {
            cout << it.first << " " << it.second << endl;
            if (ds.findUPar(it.first) == it.first)
            {
                cnt++;
            }
        }
        return n - cnt;
    }
};
```

```cpp
int main()
{

    int n = 6;
    vector<vector<int>> stones = {
        {0, 0}, {0, 2}, {1, 3}, {3, 0}, {3, 2}, {4, 3}};

    Solution obj;
    int ans = obj.maxRemove(stones, n);
    cout << "The maximum number of stones we can remove is: " << ans << endl;
    return 0;
}
```

Strongly connected components (Kosaraju's algorithm):

```cpp
class Solution
{
private:
    void dfs(int node, vector<int> &vis, vector<int> adj[],
             stack<int> &st)
    {
        vis[node] = 1;
        for (auto it : adj[node])
        {
            if (!vis[it])
            {
                dfs(it, vis, adj, st);
            }
        }

        st.push(node);
    }

private:
    void dfs3(int node, vector<int> &vis, vector<int> adjT[])
    {
        vis[node] = 1;
```

```
        for (auto it : adjT[node])
        {
            if (!vis[it])
            {
                dfs3(it, vis, adjT);
            }
        }
    }

public:
    // Function to find number of strongly connected components in the graph.
    int kosaraju(int V, vector<int> adj[])
    {
        vector<int> vis(V, 0);
        stack<int> st;
        for (int i = 0; i < V; i++)
        {
            if (!vis[i])
            {
                dfs(i, vis, adj, st);
            }
        }

        vector<int> adjT[V];
        for (int i = 0; i < V; i++)
        {
            vis[i] = 0;
            for (auto it : adj[i])
            {
                // i -> it
                // it -> i
                adjT[it].push_back(i);
            }
        }
        int scc = 0;
        while (!st.empty())
        {
            int node = st.top();
            st.pop();
            if (!vis[node])
            {
                scc++;
                dfs3(node, vis, adjT);
            }
        }
```

```
        return scc;
    }
};

int main()
{

    int n = 5;
    int edges[5][2] = {
        {1, 0}, {0, 2}, {2, 1}, {0, 3}, {3, 4}};
    vector<int> adj[n];
    for (int i = 0; i < n; i++)
    {
        adj[edges[i][0]].push_back(edges[i][1]);
    }
    Solution obj;
    int ans = obj.kosaraju(n, adj);
    cout << "The number of strongly connected components is: " << ans << endl;
    return 0;
}
```

Time complexity: O(V+ E) *Space complexity: O(V + E)*

Hashing

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.

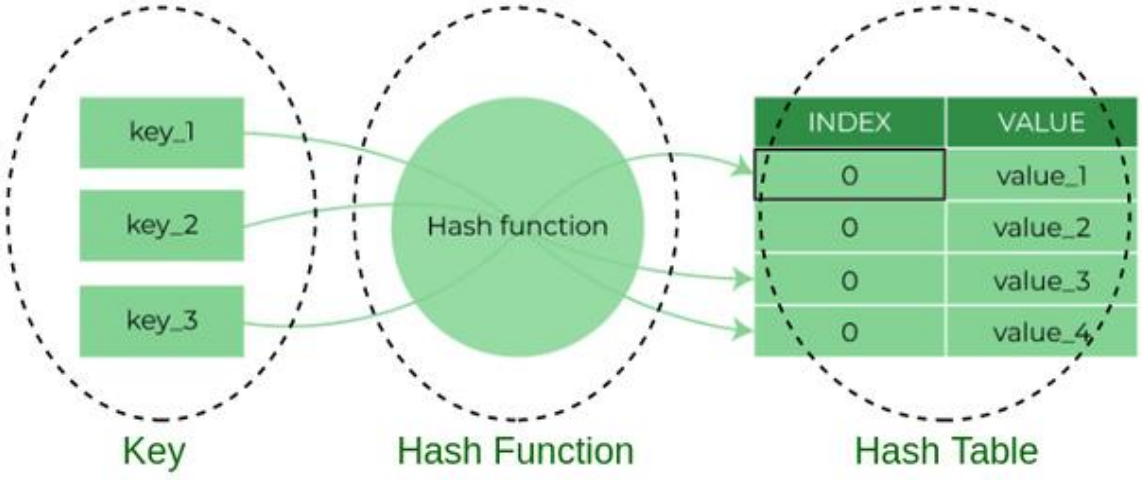
Need for hash data structure:

Every day, the data on the internet is increasing multifold and it is always a struggle to store this data efficiently. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure. Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word “efficiency“. Though storing in Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient. So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

Components of Hashing:

- **Key**: A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
- **Hash Function**: The hash function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.

- **Hash Table**: Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

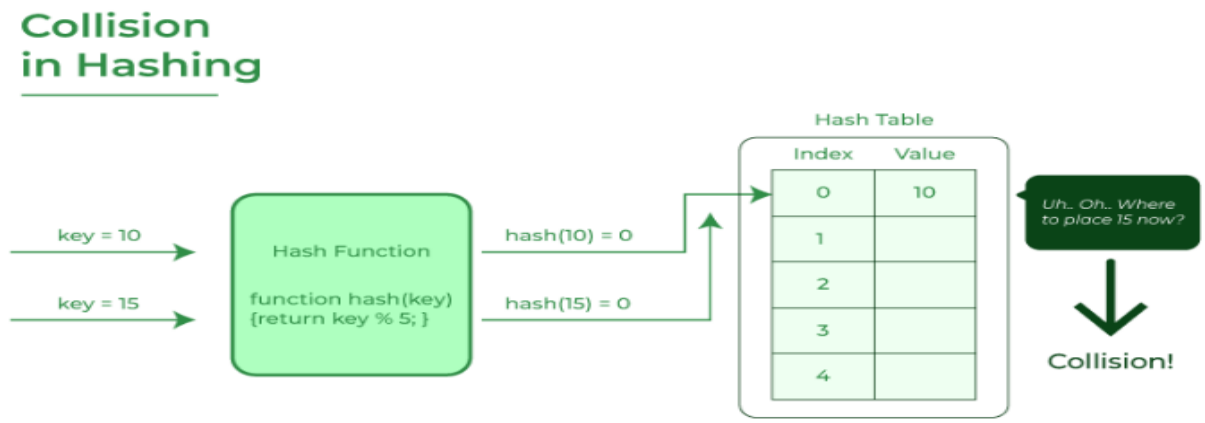



```cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// Hash function
int hashFunction(string key, int arraySize)
{
    int hash = 0;
    for (int i = 0; i < key.length(); i++)
    {
        cout << (int)key[i] << " ";
        hash = (hash + (int)key[i]) % arraySize;
    }
    cout << hash << endl;
    return hash;
}
```

```cpp
int main()
{
    int arraySize = 10;
    vector<string> keys = {"apple", "banana", "orange", "grape", "watermelon"};

    // Create an array to store the keys
    vector<string> hashTable(arraySize, "");

    // Hash and store the keys in the array
    for (int i = 0; i < keys.size(); i++)
    {
        int index = hashFunction(keys[i], arraySize);
        hashTable[index] = keys[i];
    }

    // Print the hash table
    for (int i = 0; i < arraySize; i++)
    {
        cout << "Index " << i << ": " << hashTable[i] << endl;
    }

    return 0;
}
```

Problem with collision:

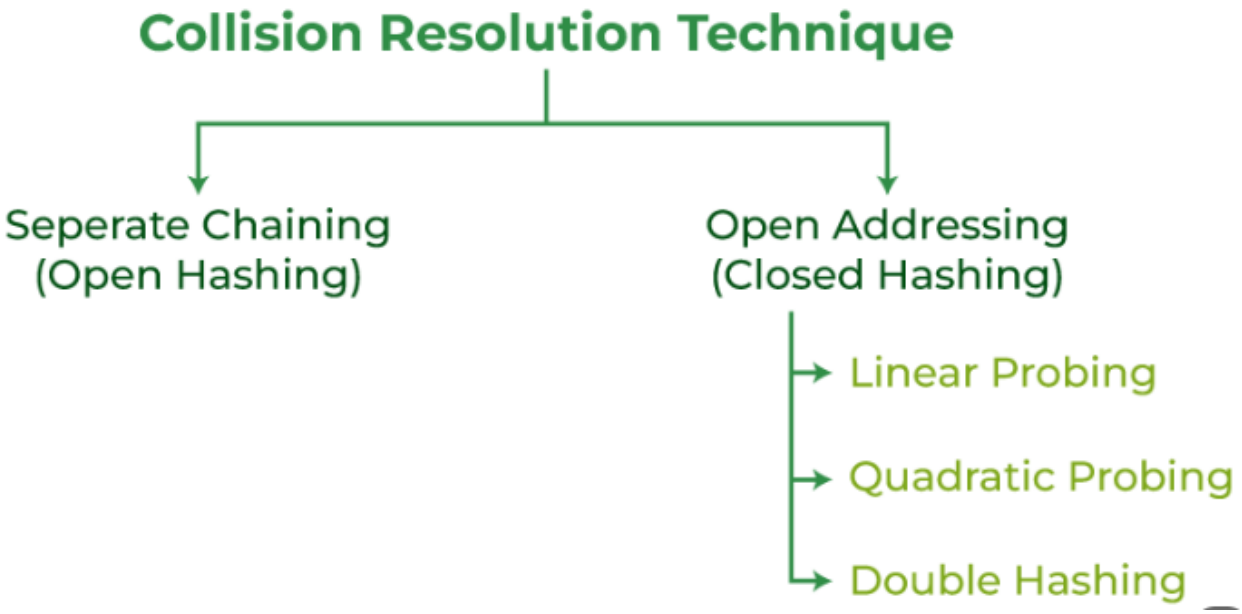
The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



How to handle collision:

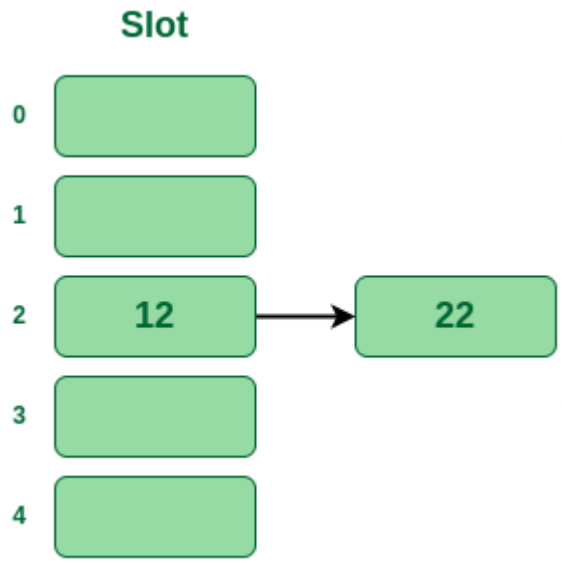
There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing



- **Separate Chaining:**

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.




```cpp
// CPP program to implement hashing with chaining

#include <iostream>
#include <list>
using namespace std;

class Hash
{
    int BUCKET;
    list<int> *table;

public:
    Hash(int V);
    void insertItem(int x);
    void deleteItem(int key);
    int hashFunction(int x)
    {
        return (x % BUCKET);
    }
    void displayHash();
```

```
};

Hash::Hash(int b)
{
    this->BUCKET = b;
    table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
{
    int index = hashFunction(key);
    table[index].push_back(key);
}

void Hash::deleteItem(int key)
{
    int index = hashFunction(key);
    list<int>::iterator i;
    for (i = table[index].begin();
         i != table[index].end(); i++)
    {
        if (*i == key)
            break;
    }
    if (i != table[index].end())
        table[index].erase(i);
}

void Hash::displayHash()
{
    for (int i = 0; i < BUCKET; i++)
    {
        cout << i;
        for (auto x : table[i])
            cout << " --> " << x;
        cout << endl;
    }
}

int main()
{
    int a[] = {15, 11, 27, 8, 12};
    int n = sizeof(a) / sizeof(a[0]);

    Hash h(7);
```

```
    for (int i = 0; i < n; i++)
        h.insertItem(a[i]);

    // h.deleteItem(8);

    h.displayHash();

    return 0;
}
```

- **Open Addressing:**

➢ **Linear Probing:**

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

```
// C++ program to implement hashing using linear probing

#include <iostream>
using namespace std;

// hash table
#define TABLE_SIZE 10
int hashTable[TABLE_SIZE];

// function to insert key into hash table
void insertHash(int key)
{
    // get index from hash function
    int index = key % TABLE_SIZE;

    // if hash table is full
    if (hashTable[index] != 0)
    {

        // linear probing
```

```
        int i = index;
        do
        {
            i = (i + 1) % TABLE_SIZE;
        } while (hashTable[i] != 0 && i != index);

        // if there is no space
        if (i == index)
        {
            cout << "No space in hash table";
            return;
        }
        else
        {
            hashTable[i] = key;
        }
    }

    // if hash table is not full
    else
    {
        hashTable[index] = key;
    }
}

// function to display hash table
void displayHash()
{
    for (int i = 0; i < TABLE_SIZE; i++)
    {
        if (hashTable[i] != 0)
            cout << i << " --> " << hashTable[i] << endl;
        else
            cout << i << endl;
    }
}

// Driver code
int main()
{
    insertHash(12);
    insertHash(25);
    insertHash(35);
    insertHash(26);
```

```
    displayHash();

    return 0;
}
```

- **Quadratic Probing:**

  Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

```
// // C++ program to implement quadratic Probing in C++

#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 11; // Size of hash table
const int EMPTY = -1;      // Empty cell value

// Hash table class
class HashTable
{
private:
    vector<int> table; // Vector to store hash table
public:
    HashTable()
    {
        table.resize(TABLE_SIZE, EMPTY); // Initialize table with EMPTY values
    }

    // Hash function to get hash value of key
    int hashFunction(int key)
    {
        return key % TABLE_SIZE;
    }
```

```
    // Quadratic probing function to resolve collisions
    int quadraticProbing(int hash, int i)
    {
        return (hash + i * i) % TABLE_SIZE;
    }

    // Function to insert key into hash table
    void insert(int key)
    {
        int hash = hashFunction(key);
        int i = 0;
        while (table[hash] != EMPTY)
        {                                          // While collision occurs
            hash = quadraticProbing(hash, ++i); // Resolve collision using
quadratic probing
        }
        table[hash] = key; // Insert key into table
    }

    // Function to search for key in hash table
    bool search(int key)
    {
        int hash = hashFunction(key);
        int i = 0;
        while (table[hash] != EMPTY)
        { // While collision occurs
            if (table[hash] == key)
            { // If key is found
                return true;
            }
            hash = quadraticProbing(hash, ++i); // Resolve collision using
quadratic probing
        }
        return false; // If key is not found
    }

    // Function to delete key from hash table
    bool remove(int key)
    {
        int hash = hashFunction(key);
        int i = 0;
        while (table[hash] != EMPTY)
        { // While collision occurs
            if (table[hash] == key)
            {                          // If key is found
```

```
                table[hash] = EMPTY; // Remove key from table
                return true;
            }
            hash = quadraticProbing(hash, ++i); // Resolve collision using
quadratic probing
        }
        return false; // If key is not found
    }

    // Function to display hash table
    void display()
    {
        cout << "Hash Table: " << endl;
        for (int i = 0; i < TABLE_SIZE; i++)
        {
            if (table[i] == EMPTY)
            {
                cout << i << "  --> "
                     << "EMPTY " << endl;
            }
            else
            {
                cout << i << "  --> " << table[i] << endl;
            }
        }
        cout << endl;
    }
};

int main()
{
    HashTable ht;

    // Insert some keys into hash table
    ht.insert(2);
    ht.insert(11);
    ht.insert(23);
    ht.insert(7);
    ht.insert(14);
    ht.insert(30);
    ht.insert(18);

    // Display hash table
    ht.display();
```

```
    // Search for some keys in hash table
    cout << "Search for key 11: " << (ht.search(11) ? "Found" : "Not found") <<
endl;
    cout << "Search for key 20: " << (ht.search(20) ? "Found" : "Not found") <<
endl;

    // Delete some keys from hash table
    cout << "Delete key 11: " << (ht.remove(11) ? "Deleted" : "Not found") <<
endl;
    cout << "Delete key 20: " << (ht.remove(20) ? "Deleted" : "Not found") <<
endl;

    ht.display();
}
```

- **Double Hashing:**

  Double hashing make use of two hash function, The first hash function is h1(k) which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key. But in case the location is occupied (collision) we will use secondary hash-function h2(k) in combination with the first hash-function h1(k) to find the new location on the hash table.

```
// C++ program to implement double hashing

#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 11; // Size of hash table
const int EMPTY = -1;      // Empty cell value

// Hash table class
class HashTable
{
private:
    vector<int> table; // Vector to store hash table
```

```
public:
    HashTable()
    {
        table.resize(TABLE_SIZE, EMPTY); // Initialize table with EMPTY values
    }

    // Hash function to get primary hash value of key
    int hashFunction(int key)
    {
        return key % TABLE_SIZE;
    }

    // Hash function to get secondary hash value of key
    int hashFunction2(int key)
    {
        return 7 - (key % 7); // 7 is a prime number smaller than TABLE_SIZE
    }

    // Double hashing function to resolve collisions
    int doubleHashing(int hash, int i, int key)
    {
        int hash2 = hashFunction2(key);
        return (hash + i * hash2) % TABLE_SIZE;
    }

    // Function to insert key into hash table
    void insert(int key)
    {
        int hash = hashFunction(key);
        int i = 0;
        while (table[hash] != EMPTY)
        {                                          // While collision occurs
            hash = doubleHashing(hash, ++i, key); // Resolve collision using
double hashing
        }
        table[hash] = key; // Insert key into table
    }

    // Function to search for key in hash table
    bool search(int key)
    {
        int hash = hashFunction(key);
        int i = 0;
        while (table[hash] != EMPTY)
        { // While collision occurs
```

```
            if (table[hash] == key)
            { // If key is found
                return true;
            }
            hash = doubleHashing(hash, ++i, key); // Resolve collision using
double hashing
        }
        return false; // If key is not found
    }

    // Function to display hash table
    void display()
    {
        cout << "Hash Table: " << endl;
        for (int i = 0; i < TABLE_SIZE; i++)
        {
            if (table[i] == EMPTY)
            {
                cout << i << "  --> "
                     << "EMPTY " << endl;
            }
            else
            {
                cout << i << "  --> " << table[i] << endl;
            }
        }
        cout << endl;
    }

    // Function to delete key from hash table
    void remove(int key)
    {
        int hash = hashFunction(key);
        int i = 0;
        while (table[hash] != EMPTY)
        {
            if (table[hash] == key)
            {
                table[hash] = EMPTY;
                cout << "Key " << key << " removed from hash table." << endl;
                return;
            }
            hash = doubleHashing(hash, ++i, key);
        }
        cout << "Key " << key << " not found in hash table." << endl;
```

```
    }
};

int main()
{
    HashTable ht;

    // Insert some keys into hash table
    ht.insert(2);
    ht.insert(11);
    ht.insert(23);
    ht.insert(7);
    ht.insert(14);
    ht.insert(30);
    ht.insert(18);

    // Display hash table
    ht.display();

    // Search for some keys in hash table
    cout << "Search for key 11: " << (ht.search(11) ? "Found" : "Not found") <<
endl;
    ht.remove(11);
    ht.display();
}
```

Advantages of Hash Data structure:

- Hash provides better synchronization than other data structures.
- Hash tables are more efficient than search trees or other data structures.
- Hash provides constant time for searching, insertion, and deletion operations on average.

Disadvantages of Hash Data structure:

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys. Hash does not allow null values.