Sieve of Eratosthenes

```
// C++ program to print all primes smaller than or equal to
// n using Sieve of Eratosthenes
#include <bits/stdc++.h>
using namespace std;

void SieveOfEratosthenes(int n)
{
	vector<bool> prime(n + 1, true);

  for (int p = 2; p * p <= n; p++) {
	if (prime[p] == true) {
	for (int i = p * p; i <= n; i += p)
		prime[i] = false;
	}
	}

	for (int p = 2; p <= n; p++)
	if (prime[p])
	cout << p << " ";
}


int main()
{
	int n = 30;
	cout << "Following are the prime numbers smaller "
	<< " than or equal to " << n << endl;
	SieveOfEratosthenes(n);
	return 0;
}
```

Time Complexity: O(N*log(log(N)))
Auxiliary Space: O(N)

Euler’s Totient Function

```
// C++ program to calculate Euler's
// Totient Function
#include <bits/stdc++.h>
```

```
using namespace std;

int phi(int n)
{
        int result = n;


        for(int p = 2; p * p <= n; ++p)
        {

        // Check if p is a prime factor.
        if (n % p == 0)
        {

        // If yes, then update n and result
        while (n % p == 0)
                n /= p;

        result -= result / p;
        }
        }

        if (n > 1)
        result -= result / n;

        return result;
}

// Driver code
int main()
{
        int n;
        for(n = 1; n <= 10; n++)
        {
        cout << "Phi" << "("
        << n << ")" << " = "
        << phi(n) << endl;
        }
        return 0;
}
```
Time complexity: O(√n log n)
Space complexity:O(1)

Print all prime factors of a given number

```
// C++ program to print all prime factors
#include <bits/stdc++.h>
using namespace std;

void primeFactors(int n) {

        while (n % 2 == 0) {

        cout << 2 << " ";
        n = n/2;
        }

        for (int i = 3; i*i <= n; i = i + 2) {

        while (n % i == 0) {

        cout << i << " ";
        n = n/i;
        }
        }

        if (n > 2)
        cout << n << " ";
}


int main() {

        int n = 315;
        primeFactors(n);
        return 0;
}
```

Find GCD or HCF of Two Numbers

```
// C++ program to find GCD of two numbers
#include <iostream>
using namespace std;
int gcd(int a, int b)
{
	return b == 0 ? a : gcd(b, a % b);
}
 int main()
{
	int a = 98, b = 56;
	cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);
	return 0;
}
```

LCM(a, b) = (a x b) / GCD(a, b)

https://www.geeksforgeeks.org/number-theory-competitive-programming/

Implementing upper_bound() and lower_bound()

```
// Function to implement lower_bound
int lower_bound(int arr[], int N, int X)
{
	int mid;

	// Initialise starting index and
	// ending index
	int low = 0;
	int high = N;

	// Till low is less than high
	while (low < high) {
	mid = low + (high - low) / 2;

	// If X is less than or equal
	// to arr[mid], then find in
	// left subarray
	if (X <= arr[mid]) {
	high = mid;
	}

	// If X is greater arr[mid]
	// then find in right subarray
```

```
        else {
        low = mid + 1;
        }
        }

        // if X is greater than arr[n-1]
        if(low < N && arr[low] < X) {
        low++;
        }

        // Return the lower_bound index
        return low;
}

// Function to implement upper_bound
int upper_bound(int arr[], int N, int X)
{
        int mid;

        // Initialise starting index and
        // ending index
        int low = 0;
        int high = N;

        // Till low is less than high
        while (low < high) {
        // Find the middle index
        mid = low + (high - low) / 2;

        // If X is greater than or equal
        // to arr[mid] then find
        // in right subarray
        if (X >= arr[mid]) {
        low = mid + 1;
        }

        // If X is less than arr[mid]
        // then find in left subarray
        else {
        high = mid;
        }
        }

        // if X is greater than arr[n-1]
```

```
	if(low < N && arr[low] <= X) {
	low++;
	}

	// Return the upper_bound index
	return low;
}
```

Dijkstra’s Algorithm

```
// C++ program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph
#include <iostream>
using namespace std;
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{

	// Initialize min value
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
	if (sptSet[v] == false && dist[v] <= min)
	min = dist[v], min_index = v;

	return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
	cout << "Vertex \t Distance from Source" << endl;
	for (int i = 0; i < V; i++)
```

```
		cout << i << " \t\t\t\t" << dist[i] << endl;
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
		int dist[V]; // The output array.  dist[i] will hold the
				// shortest
		// distance from src to i

		bool sptSet[V]; // sptSet[i] will be true if vertex i is
				// included in shortest
		// path tree or shortest distance from src to i is
		// finalized

		// Initialize all distances as INFINITE and stpSet[] as
		// false
		for (int i = 0; i < V; i++)
		dist[i] = INT_MAX, sptSet[i] = false;

		// Distance of source vertex from itself is always 0
		dist[src] = 0;

		// Find shortest path for all vertices
		for (int count = 0; count < V - 1; count++) {
		// Pick the minimum distance vertex from the set of
		// vertices not yet processed. u is always equal to
		// src in the first iteration.
		int u = minDistance(dist, sptSet);

		// Mark the picked vertex as processed
		sptSet[u] = true;

		// Update dist value of the adjacent vertices of the
		// picked vertex.
		for (int v = 0; v < V; v++)

		// Update dist[v] only if is not in sptSet,
		// there is an edge from u to v, and total
		// weight of path from src to  v through u is
		// smaller than current value of dist[v]
		if (!sptSet[v] && graph[u][v]
                    && dist[u] != INT_MAX
                    && dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist);
}

// driver's code
int main()
{

        /* Let us create the example graph discussed above */
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                    { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                    { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                    { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                    { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                    { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                    { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                    { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                    { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

        // Function call
        dijkstra(graph, 0);

        return 0;
}

// This code is contributed by shivanisinghss2110

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

// Define INF as a large value to represent infinity
#define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// Class representing a graph using adjacency list representation
class Graph {
        int V; // Number of vertices
```

```
	list<iPair> *adj; // Adjacency list

public:
	Graph(int V); // Constructor

	void addEdge(int u, int v, int w); // Function to add an edge
	void shortestPath(int s); // Function to print shortest path from source
};

// Constructor to allocate memory for the adjacency list
Graph::Graph(int V) {
	this->V = V;
	adj = new list<iPair>[V];
}

// Function to add an edge to the graph
void Graph::addEdge(int u, int v, int w) {
	adj[u].push_back(make_pair(v, w));
	adj[v].push_back(make_pair(u, w)); // Since the graph is undirected
}

// Function to print shortest paths from source
void Graph::shortestPath(int src) {
	// Create a priority queue to store vertices being processed
	// Priority queue sorted by the first element of the pair (distance)
	priority_queue<iPair, vector<iPair>, greater<iPair>> pq;

	// Create a vector to store distances and initialize all distances as INF
	vector<int> dist(V, INF);

	// Insert source into priority queue and initialize its distance as 0
	pq.push(make_pair(0, src));
	dist[src] = 0;

	// Process the priority queue
	while (!pq.empty()) {
	// Get the vertex with the minimum distance
	int u = pq.top().second;
	pq.pop();

	// Iterate through all adjacent vertices of the current vertex
	for (auto &neighbor : adj[u]) {
	int v = neighbor.first;
	int weight = neighbor.second;
```

```
        // If a shorter path to v is found
        if (dist[v] > dist[u] + weight) {
                // Update distance and push new distance to the priority queue
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
        }
        }
        }

        // Print the shortest distances
        cout << "Vertex Distance from Source" << endl;
        for (int i = 0; i < V; ++i)
        cout << i << " \t\t " << dist[i] << endl;
}

// Driver's code
int main() {
        int V = 9; // Number of vertices
        Graph g(V);

        // Add edges to the graph
        g.addEdge(0, 1, 4);
        g.addEdge(0, 7, 8);
        g.addEdge(1, 2, 8);
        g.addEdge(1, 7, 11);
        g.addEdge(2, 3, 7);
        g.addEdge(2, 8, 2);
        g.addEdge(2, 5, 4);
        g.addEdge(3, 4, 9);
        g.addEdge(3, 5, 14);
        g.addEdge(4, 5, 10);
        g.addEdge(5, 6, 2);
        g.addEdge(6, 7, 1);
        g.addEdge(6, 8, 6);
        g.addEdge(7, 8, 7);

        // Call the shortestPath function
        g.shortestPath(0);

        return 0;
}
```

Longest Common Subsequence (LCS)

```
int lcs(string &s1, string &s2) {
        int m = s1.size();
        int n = s2.size();

        // Initializing a matrix of size (m+1)*(n+1)
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

        // Building dp[m+1][n+1] in bottom-up fashion
        for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
        if (s1[i - 1] == s2[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
        else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
        }

        // dp[m][n] contains length of LCS for s1[0..m-1]
        // and s2[0..n-1]
        return dp[m][n];
}
```

Bellman–Ford Algorithm

```
vector<int> bellmanFord(int V, vector<vector<int>>& edges, int src) {

        // Initially distance from source to all
        // other vertices is not known(Infinite).
        vector<int> dist(V, 1e8);
        dist[src] = 0;

        // Relaxation of all the edges V times, not (V - 1) as we
        // need one additional relaxation to detect negative cycle
        for (int i = 0; i < V; i++) {
        for (vector<int> edge : edges) {
```

```
        int u = edge[0];
        int v = edge[1];
        int wt = edge[2];
        if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

                // If this is the Vth relaxation, then there is
                // a negative cycle
                if(i == V - 1)
                return {-1};

                // Update shortest distance to node v
                dist[v] = dist[u] + wt;
        }
        }
        }

        return dist;
}
```

Pseudo-Code of Floyd Warshall Algorithm :

```
For k = 0 to n – 1
 For i = 0 to n – 1
 For j = 0 to n – 1
 Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])
```

where i = source Node, j = Destination Node, k = Intermediate Node

Edit Distance

```
int m = s1.length();
int n = s2.length();

// Create a table to store results of subproblems
vector<vector<int>> dp(m + 1, vector<int>(n + 1));

// Fill the known entries in dp[][]
// If one string is empty, then answer
// is length of the other string
for (int i = 0; i <= m; i++)
dp[i][0] = i;
for (int j = 0; j <= n; j++)
```

```
        dp[0][j] = j;

        // Fill the rest of dp[][]
        for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
        if (s1[i - 1] == s2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];
        else
                dp[i][j] = 1 + min({dp[i][j - 1],
                        dp[i - 1][j],
                        dp[i - 1][j - 1]});
        }
        }
```

Program for nth Catalan Number

$$C_n = \frac{2n!}{(n+1)!*n!} \text{ or } C_n = \frac{1}{(n+1)}\binom{2n}{n}$$

$$\mathbf{C}(n,k) = \mathbf{C}_k^n = {}_nC_k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
// C++ program for nth Catalan Number

#include <iostream>
using namespace std;

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k) {
        int res = 1;

        // Since C(n, k) = C(n, n-k)
        if (k > n - k)
        k = n - k;

        // Calculate value of [n*(n-1)*---*(n-k+1)] /
        // [k*(k-1)*---*1]
        for (int i = 0; i < k; ++i)
        {
        res *= (n - i);
        res /= (i + 1);
        }

        return res;
}

// A Binomial coefficient based function to find nth catalan
// number in O(n) time
int findCatalan(int n) {

        // Calculate value of 2nCn
        int c = binomialCoeff(2 * n, n);

        // return 2nCn/(n+1)
        return c / (n + 1);
}

int main() {
        int n = 6;
        int res = findCatalan(n);
        cout << res;
        return 0;
}
```

Subset Sum Problem

```
bool isSubsetSum(vector<int> arr, int sum) {
        int n = arr.size();
        vector<bool> prev(sum + 1, false), curr(sum + 1);

        // Mark prev[0] = true as it is true
        // to make sum = 0 using 0 elements
        prev[0] = true;

        // Fill the subset table in
        // bottom up manner
        for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= sum; j++) {
        if (j < arr[i - 1])
```

```
                curr[j] = prev[j];
        else
                curr[j] = (prev[j] || prev[j - arr[i - 1]]);
        }
        prev = curr;
        }
        return prev[sum];
}
```

Coin Change

```
int count(vector<int> &coins, int sum) {
        int n = coins.size();

        // dp[i] will be storing the number of solutions for
        // value i. We need sum+1 rows as the dp is
        // constructed in bottom up manner using the base case
        // (sum = 0)
        vector<int> dp(sum + 1);

        // Base case (If given value is 0)
        dp[0] = 1;

        // Pick all coins one by one and update the table[]
        // values after the index greater than or equal to the
        // value of the picked coin
        for (int i = 0; i < n; i++)
        for (int j = coins[i]; j <= sum; j++)
        dp[j] += dp[j - coins[i]];
        return dp[sum];
}
```

0/1 Knapsack Problem

```
int knapSack(int W, vector<int>& wt, vector<int>& val) {
        // Making and initializing dp vector
        vector<int> dp(W + 1, 0);

        for (int i = 1; i <= wt.size(); i++) {
        for (int w = W; w >= 0; w--) {
        if (wt[i - 1] <= w)

                // Finding the maximum value
                dp[w] = max(dp[w], dp[w - wt[i - 1]] + val[i - 1]);
        }
        }
        return dp[W];
}
```

Union-Find Algorithm

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

class DisjointUnionSets {
        vector<int> rank, parent;

public:

        // Constructor to initialize sets
        DisjointUnionSets(int n) {
        rank.resize(n, 0);
        parent.resize(n);

        // Initially, each element is in its own set
        for (int i = 0; i < n; i++) {
        parent[i] = i;
        }
        }

        // Find the representative of the set that x belongs to
        int find(int x) {
        if (parent[x] != x) {

        // Path compression
        parent[x] = find(parent[x]);
        }
        return parent[x];
        }

        // Union of sets containing x and y
        void unionSets(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);

        // If they are in the same set, no need to union
        if (xRoot == yRoot) return;

        // Union by rank
        if (rank[xRoot] < rank[yRoot]) {
        parent[xRoot] = yRoot;
        } else if (rank[yRoot] < rank[xRoot]) {
        parent[yRoot] = xRoot;
        } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
        }
        }
};

int main() {
        // Let there be 5 persons with ids 0, 1, 2, 3, and 4
        int n = 5;
        DisjointUnionSets dus(n);

        // 0 is a friend of 2
        dus.unionSets(0, 2);

        // 4 is a friend of 2
        dus.unionSets(4, 2);

        // 3 is a friend of 1
        dus.unionSets(3, 1);

        // Check if 4 is a friend of 0
```

```
    if (dus.find(4) == dus.find(0))
    cout << "Yes\n";
    else
    cout << "No\n";

    // Check if 1 is a friend of 0
    if (dus.find(1) == dus.find(0))
    cout << "Yes\n";
    else
    cout << "No\n";

    return 0;
}
```

Breadth First Search or BFS for a Graph

```
void bfs(vector<vector<int>>& adj, int s)
{
    // Create a queue for BFS
    queue<int> q;

    // Initially mark all the vertices as not visited
    // When we push a vertex into the q, we mark it as
    // visited
    vector<bool> visited(adj.size(), false);

    // Mark the source node as visited and
    // enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (!q.empty()) {

    // Dequeue a vertex from queue and print it
    int curr = q.front();
    q.pop();
    cout << curr << " ";
```

```
        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (int x : adj[curr]) {
        if (!visited[x]) {
                visited[x] = true;
                q.push(x);
        }
        }
        }
}
```

Depth First Search or DFS for a Graph

```
void DFSRec(vector<vector<int>> &adj, vector<bool> &visited, int s){

        visited[s] = true;

        // Print the current vertex
        cout << s << " ";

        // Recursively visit all adjacent vertices
        // that are not visited yet
        for (int i : adj[s])
        if (visited[i] == false)
        DFSRec(adj, visited, i);
}

// Main DFS function that initializes the visited array
// and call DFSRec
void DFS(vector<vector<int>> &adj, int s){
        vector<bool> visited(adj.size(), false);
        DFSRec(adj, visited, s);
}
```

Detect cycle in an undirected graph

```
bool isCyclicUtil(int v, vector<vector<int>>& adj,
                bool visited[], int parent) {
        // Mark the current node as visited
        visited[v] = true;

        // Recur for all the vertices
        // adjacent to this vertex
        for (int i : adj[v]) {
        // If an adjacent vertex is not visited,
        // then recur for that adjacent
        if (!visited[i]) {
        if (isCyclicUtil(i, adj, visited, v))
                return true;
        }
        // If an adjacent vertex is visited and
        // is not parent of current vertex,
        // then there exists a cycle in the graph.
        else if (i != parent)
        return true;
        }
        return false;
}
```

Check whether a given graph is Bipartite or not

```
bool isBipartite(int V, vector<vector<int>> &adj) {

        // Vector to store colors of vertices.
        // Initialize all as -1 (uncolored)
        vector<int> color(V, -1);

        // Queue for BFS
        queue<int> q;

        // Iterate through all vertices to handle disconnected graphs
        for(int i = 0; i < V; i++) {

        // If the vertex is uncolored, start BFS from it
        if(color[i] == -1) {
```

```
        // Assign first color (0) to the starting vertex
        color[i] = 0;
        q.push(i);

        // Perform BFS
        while(!q.empty()) {
                int u = q.front();
                q.pop();

                // Traverse all adjacent vertices
                for(auto &v : adj[u]) {

                // If the adjacent vertex is uncolored,
                // assign alternate color
                if(color[v] == -1) {
                color[v] = 1 - color[u];
                q.push(v);
                }

                // If the adjacent vertex has the same color,
                // graph is not bipartite
                else if(color[v] == color[u]) {
                return false;
                }
                }
        }
        }
        }

        // If no conflicts in coloring, graph is bipartite
        return true;
}
```

The rule for modular addition is:

(a + b) mod m = ((a mod m) + (b mod m)) mod m

The Rule for modular multiplication is:

(a x b) mod m = ((a mod m) x (b mod m)) mod m

Modular multiplicative inverse

```
void modInverse(int A, int M) {
  int x, y;
  int g = gcdExtended(A, M, &x, &y);
  if (g != 1)
  	cout << "Inverse doesn't exist";
  else {

  	// m is added to handle negative x
  	int res = (x % M + M) % M;
  	cout << "Modular multiplicative inverse is " << res;
  }
}

// Function for extended Euclidean Algorithm
int gcdExtended(int a, int b, int* x, int* y)
{

  // Base Case
  if (a == 0) {
  	*x = 0, *y = 1;
  	return b;
  }


  // To store results of recursive call

  int x1, y1;

  int gcd = gcdExtended(b % a, a, &x1, &y1);


  // Update x and y using results of recursive

  // call

  *x = y1 - (b / a) * x1;

  *y = x1;


  return gcd;

}
```

deque::push_front() It is used to push elements into a deque from the front.

deque::push_back() This function is used to push elements into a deque from the back.

deque::pop_front() and deque::pop_back() pop_front() function is used to pop or remove elements from a deque from the front. pop_back() function is used to pop or remove elements from a deque from the back.

deque::front() and deque::back() front() function is used to reference the first element of the deque container. back() function is used to reference the last element of the deque container.

Infix to postfix

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following

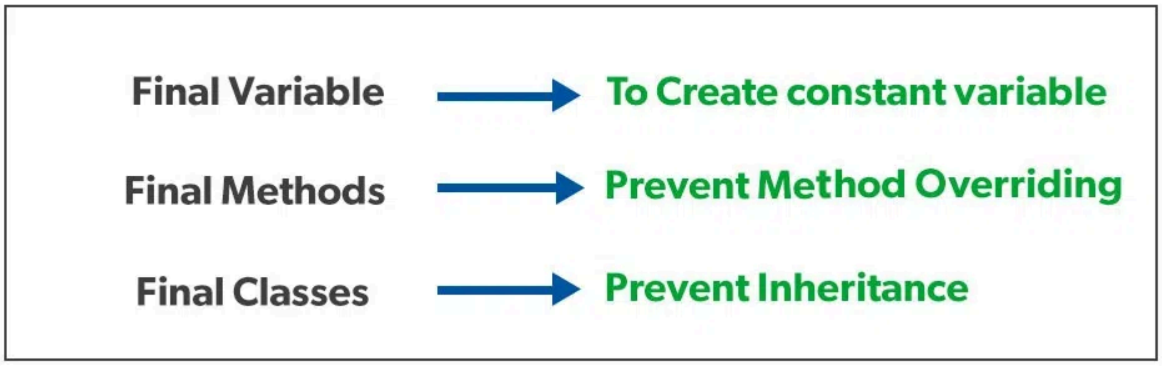
    - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
    - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

infix expression to prefix expression

To convert an infix expression to a prefix expression, we can use the **stack data structure**. The idea is as follows:

- **Step 1:** Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.
- **Step 2:** Convert the reversed **infix expression to "nearly" postfix expression**.
    - While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.
- **Step 3:** Reverse the postfix expression.

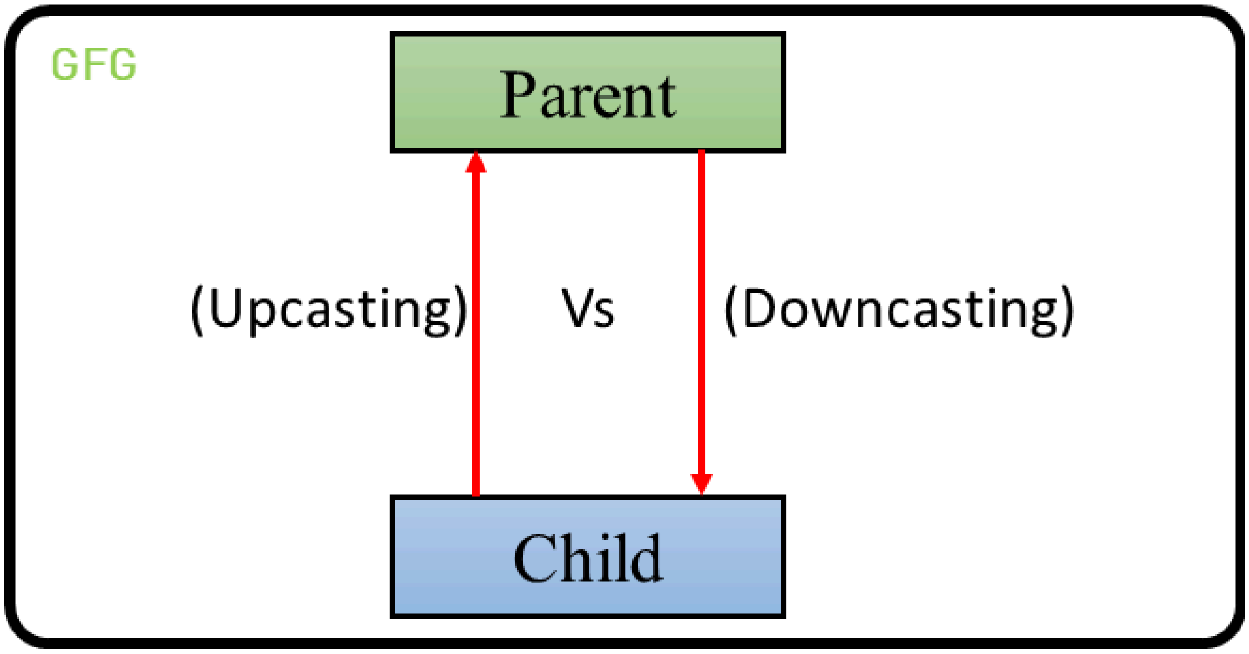
OOP



Characteristics of static keyword:

Here are some characteristics of the static keyword in Java:

- **Shared memory allocation**: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.
- **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.
- **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.
- **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.
- **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.



1) Physical layer

- The main functionality of the physical layer is to transmit the individual bits from one node to another node.
- It is the lowest layer of the OSI model.
- It establishes, maintains and deactivates the physical connection.
- It specifies the mechanical, electrical and procedural network interface specifications.

2) Data-Link Layer

Framing: The data link layer translates the physical's raw bit stream into packets known as Frames. The Data link layer adds the header and trailer to the frame. The header which is added to the frame contains the hardware destination and source address.

Physical Addressing: The Data link layer adds a header to the frame that contains a destination address. The frame is transmitted to the destination address mentioned in the header.

Flow Control: Flow control is the main functionality of the Data-link layer. It is the technique through which the constant data rate is maintained on both the sides so that no data get corrupted. It ensures that the transmitting station such as a server with higher processing speed does not exceed the receiving station, with lower processing speed.

- **Error Control:** Error control is achieved by adding a calculated value CRC (Cyclic Redundancy Check) that is placed to the Data link layer's trailer which is added to the message frame before it is sent to the physical layer. If any error seems to occurr, then the receiver sends the acknowledgment for the retransmission of the corrupted frames.
- **Access Control:** When two or more devices are connected to the same communication channel, then the data link layer protocols are used to determine which device has control over the link at a given time.

3) Network Layer

Functions of Network Layer:

- **Internetworking:** An internetworking is the main responsibility of the network layer. It provides a logical connection between different devices.
- **Addressing**: A Network layer adds the source and destination address to the header of the frame. Addressing is used to identify the device on the internet.
- **Routing**: Routing is the major component of the network layer, and it determines the best optimal path out of the multiple paths from source to the destination.
- **Packetizing:** A Network Layer receives the packets from the upper layer and converts them into packets. This process is known as Packetizing. It is achieved by internet protocol (IP).

4) Transport Layer

Functions of Transport Layer:

- **Service-point addressing:** Computers run several programs simultaneously due to this reason, the transmission of data from source to the destination not only from one computer to another computer but also from one process to another process. The transport layer adds the header that contains the address known as a service-point address or port address. The responsibility of the network layer is to transmit the data from one computer to another computer and the responsibility of the transport layer is to transmit the message to the correct process.
- **Segmentation and reassembly:** When the transport layer receives the message from the upper layer, it divides the message into multiple segments, and each segment is assigned with a sequence number that uniquely identifies each segment. When the message has arrived at the destination, then the transport layer reassembles the message based on their sequence numbers.
- **Connection control:** Transport layer provides two services Connection-oriented service and connectionless service. A connectionless service treats each segment as an individual packet, and they all travel in different routes to reach the destination. A connection-oriented service makes a connection with the transport layer at the destination machine before delivering the packets. In connection-oriented service, all the packets travel in the single route.
- **Flow control:** The transport layer also responsible for flow control but it is performed end-to-end rather than across a single link.
- **Error control:** The transport layer is also responsible for Error control. Error control is performed end-to-end rather than across the single link. The sender transport layer ensures that message reach at the destination without any error.

5) Session Layer

Functions of Session layer:

- **Dialog control:** Session layer acts as a dialog controller that creates a dialog between two processes or we can say that it allows the communication between two processes which can be either half-duplex or full-duplex.
- **Synchronization:** Session layer adds some checkpoints when transmitting the data in a sequence. If some error occurs in the middle of the transmission of data, then the transmission will take place again from the checkpoint. This process is known as Synchronization and recovery.

6) Presentation Layer

Functions of Presentation layer:

- **Translation:** The processes in two systems exchange the information in the form of character strings, numbers and so on. Different computers use different encoding methods, the presentation layer handles the interoperability between the different encoding methods. It converts the data from sender-dependent format into a common format and changes the common format into receiver-dependent format at the receiving end.
- **Encryption:** Encryption is needed to maintain privacy. Encryption is a process of converting the sender-transmitted information into another form and sends the resulting message over the network.
- **Compression:** Data compression is a process of compressing the data, i.e., it reduces the number of bits to be transmitted. Data compression is very important in multimedia such as text, audio, video.

7) Application Layer

- An application layer serves as a window for users and application processes to access network service.
- It handles issues such as network transparency, resource allocation, etc.

Functions of Application layer:

- **File transfer, access, and management (FTAM):** An application layer allows a user to access the files in a remote computer, to retrieve the files from a computer and to manage the files in a remote computer.
- **Mail services:** An application layer provides the facility for email forwarding and storage.
- Directory services: An application provides the distributed database sources and is used to provide that global information about various objects.

TCP/IP Model

The TCP/IP model is a fundamental framework for computer networking. It stands for Transmission Control Protocol/Internet Protocol, which are the core protocols of the Internet. This model defines how data is transmitted over networks, ensuring reliable communication between devices. It consists of four layers: the Link Layer, the Internet Layer, the Transport Layer, and the Application Layer. Each layer has specific functions that help manage different aspects of network communication, making it essential for understanding and working with modern networks.

How Does the TCP/IP Model Work?

Whenever we want to send something over the internet using the TCP/IP Model, the TCP/IP Model divides the data into packets at the sender's end and the same packets have to be recombined at the receiver's end to form the same data, and this thing happens to maintain the accuracy of the data. TCP/IP model divides the data into a 4-layer procedure, where the data first go into this layer in one order and again in reverse order to get organized in the same way at the receiver's end.

Layers of TCP/IP Model

- Application Layer
- Transport Layer(TCP/UDP)
- Network/Internet Layer(IP)
- Network Access Layer

1. Network Access Layer

It is a group of applications requiring network communications. This layer is responsible for generating the data and requesting connections. It acts on behalf of the sender and the Network Access layer on the behalf of the receiver. During this article, we will be talking on the behalf of the receiver.

The packet's network protocol type, in this case, TCP/IP, is identified by network access layer. Error prevention and "framing" are also provided by this layer. Point-to-Point Protocol (PPP) framing and Ethernet IEEE 802.2 framing are two examples of data-link layer protocols.

2. Internet or Network Layer

This layer parallels the functions of OSI's Network layer. It defines the protocols which are responsible for the logical transmission of data over the entire network. The main protocols residing at this layer are as follows:

- **IP:**IP stands for Internet Protocol and it is responsible for delivering packets from the source host to the destination host by looking at the IP addresses in the packet headers. IP has 2 versions: IPv4 and IPv6. IPv4 is the one that most websites are using currently. But IPv6 is growing as the number of IPv4 addresses is limited in number when compared to the number of users.
- **ICMP:**ICMP stands for Internet Control Message Protocol. It is encapsulated within IP datagrams and is responsible for providing hosts with information about network problems.
- **ARP:**ARP stands for Address Resolution Protocol. Its job is to find the hardware address of a host from a known IP address. ARP has several types: Reverse ARP, Proxy ARP, Gratuitous ARP, and Inverse ARP.

The Internet Layer is a layer in the Internet Protocol (IP) suite, which is the set of protocols that define the Internet. The Internet Layer is responsible for routing packets of data from one device to another across a network. It does this by assigning each device a unique IP address, which is used to identify the device and determine the route that packets should take to reach it.

3. Transport Layer

The TCP/IP transport layer protocols exchange data receipt acknowledgments and retransmit missing packets to ensure that packets arrive in order and without error. End-to-end communication is referred to as such. Transmission Control Protocol (TCP) and User Datagram Protocol are transport layer protocols at this level (UDP).

- **TCP:** Applications can interact with one another using TCP as though they were physically connected by a circuit. TCP transmits data in a way that resembles character-by-character transmission rather than separate packets. A starting point that establishes the connection, the whole transmission in byte order, and an ending point that closes the connection make up this transmission.

- **UDP:** The datagram delivery service is provided by UDP , the other transport layer protocol. Connections between receiving and sending hosts are not verified by UDP. Applications that transport little amounts of data use UDP rather than TCP because it eliminates the processes of establishing and validating connections.

4. Application Layer

This layer is analogous to the transport layer of the OSI model. It is responsible for end-to-end communication and error-free delivery of data. It shields the upper-layer applications from the complexities of data. The three main protocols present in this layer are:

HTTP and HTTPS:HTTP stands for Hypertext transfer protocol. It is used by the World Wide Web to manage communications between web browsers and servers.

- **SSH:**SSH stands for Secure Shell. It is a terminal emulations software similar to Telnet. The reason SSH is preferred is because of its ability to maintain the encrypted connection. It sets up a secure session over a TCP/IP connection.
- **NTP:**NTP stands for Network Time Protocol. It is used to synchronize the clocks on our computer to one standard time source. It is very useful in situations like bank transactions.

TCP/IP	**OSI**
TCP refers to Transmission Control Protocol.	OSI refers to Open Systems Interconnection.
TCP/IP uses both the session and presentation layer in the application layer itself.	OSI uses different session and presentation layers.
TCP/IP follows connectionless a horizontal approach.	OSI follows a vertical approach.

The Transport layer in TCP/IP does not provide assurance delivery of packets.	In the OSI model, the transport layer provides assurance delivery of packets.
Protocols cannot be replaced easily in TCP/IP model.	While in the OSI model, Protocols are better covered and are easy to replace with the technology change.
TCP/IP model network layer only provides connectionless (IP) services. The transport layer (TCP) provides connections.	Connectionless and connection-oriented services are provided by the network layer in the OSI model.

Public IP Address

- **Definition**: A **Public IP** address is assigned to a device (e.g., a router or a server) that is directly accessible over the internet. It is globally unique across the internet.
- **Purpose**:
  - It enables communication between devices on different networks.
  - It is used for web servers, email servers, or any device that needs to be accessible from the internet.
- **Assigned By**: Internet Service Providers (ISPs) assign public IP addresses.
- **Range**: Public IP addresses do not fall within specific reserved ranges, unlike private IPs.
- **Visibility**: Public IPs are visible to other devices on the internet.

Example:

- IPv4: `192.0.2.1`
- IPv6: `2001:0db8::1`

Private IP Address

- **Definition**: A **Private IP** address is used within a local network (e.g., home or office). These addresses are not routable on the public internet.
- **Purpose**:
  - It identifies devices within a local area network (LAN).
  - It helps conserve the number of public IP addresses used.

Assigned By: Private IPs are assigned either manually or dynamically by a **router** or a **DHCP server**.

Range: Private IPs fall within the following reserved ranges:

- `10.0.0.0` to `10.255.255.255` (Class A)
- `172.16.0.0` to `172.31.255.255` (Class B)
- `192.168.0.0` to `192.168.255.255` (Class C)

Visibility: Private IPs are only visible within the local network.

The concept of Public IP and Private IP addresses relates to how devices in a network are identified and communicate over the internet or within a local network. Here's an explanation of both:

Public IP Address

Definition: A Public IP address is assigned to a device (e.g., a router or a server) that is directly accessible over the internet. It is globally unique across the internet.

Purpose:

It enables communication between devices on different networks.

It is used for web servers, email servers, or any device that needs to be accessible from the internet.

Assigned By: Internet Service Providers (ISPs) assign public IP addresses.

Range: Public IP addresses do not fall within specific reserved ranges, unlike private IPs.

Visibility: Public IPs are visible to other devices on the internet.

Example:

IPv4: 192.0.2.1

IPv6: 2001:0db8::1

Private IP Address

Definition: A Private IP address is used within a local network (e.g., home or office). These addresses are not routable on the public internet.

Purpose:

It identifies devices within a local area network (LAN).

It helps conserve the number of public IP addresses used.

Assigned By: Private IPs are assigned either manually or dynamically by a router or a DHCP server.

Range: Private IPs fall within the following reserved ranges:

10.0.0.0 to 10.255.255.255 (Class A)

172.16.0.0 to 172.31.255.255 (Class B)

192.168.0.0 to 192.168.255.255 (Class C)

Visibility: Private IPs are only visible within the local network.

Example: 192.168.0.113 (commonly used for home networks).

Key Differences Between Public and Private IP Addresses

Aspect	**Public IP**	**Private IP**
Scope	Global (internet-wide)	Local (within a LAN)
Uniqueness	Unique across the internet	Unique only within the local network
Assigned By	Internet Service Provider (ISP)	Router or DHCP server
Accessibility	Accessible over the internet	Not directly accessible from the internet

Examples	`203.0.113.1`	`192.168.1.1`

What is a Router?

The router is a networking device that works at the network layer i.e., a third layer of the ISO-OSI model, and is the multiport device. It establishes a simple connection between the networks to provide the data flow between the networks. Router transfers data in the form of packets is used in LAN as well as MAN.

What is a Switch?

It is a point-to-point communication device. Basically, it is a kind of bridge that provides better connections. It is a kind of device that sets up and stops the connections according to the requirements needed at that time. It comes up with many features such as flooding, filtering, and frame transmission.

Router	**Switch**
The main objective of router is to connect various networks simultaneously.	While the main objective of switch is to connect various devices simultaneously.
It works in network layer.	While it works in data link layer.
Router is used by LAN as well as MAN.	While switch is used by only LAN.
Through the router, data is sent in the form of packets.	While through switch data is sent in the form of frame.

There is less collision taking place in the router.	While there is no collision taking place in full duplex switch.
Router is compatible with NAT.	While it is not compatible with NAT.
Router is a relatively much more expensive device than switch.	Switch is an expensive device than hub. but cheaper than router.
maximum speed for wireless is 1-10 Mbps and maximum speed for wired connections is 100 Mbps.	Maximum speed is 10Mbps to 100Mbps.
Router needs at least two networks to connect.	Switch needs at least single network is to connect.

What is a Subnet?

A subnet is like a smaller group within a large network. It is a way to split a large network into smaller networks so that devices present in one network can transmit data more easily. For example, in a company, different departments can each have their own subnet, keeping their data traffic separate from others. Subnet makes the network faster and easier to manage and also improves the security of the network.

Different Parts of IP Address

An IP address is made up of different parts, each serving a specific purpose in identifying a device on a network. In an IPv4 address, there are four parts, called “octets,” which are separated by dots (e.g., 192.168.1.1). Here’s what each part represents:

- **Network Portion**: The first few sections (octets) of an IP address identify the network that the device belongs to. This part of the IP address is common among all devices on the same network, allowing them to communicate with each other and share resources.

- **Host Portion**: The remaining sections of the IP address specify the individual device, or "host," within that network. This part makes each device unique within the network, allowing the router to distinguish between different devices.

The 32-bit IP address is divided into sub-classes. These are given below:

- **Class A:** The network ID is 8 bits long and the host ID is 24 bits long.
- **Class B:** The network ID is 16 bits long and the host ID is 16 bits long.
- **Class C:** The network ID is 24 bits long and the host ID is 8 bits long.

How Does Subnetting Work?

The working of subnets starts in such a way that firstly it divides the subnets into smaller subnets. For communicating between subnets, routers are used. Each subnet allows its linked devices to communicate with each other. Subnetting for a network should be done in such a way that it does not affect the network bits.

In class C the first 3 octets are network bits so it remains as it is.

- **For Subnet-1:** The first bit which is chosen from the host id part is zero and the range will be from (193.1.2.00000000 till you get all 1's in the host ID part i.e, 193.1.2.01111111) except for the first bit which is chosen zero for subnet id part.

Thus, the range of subnet 1 is: **193.1.2.0 to 193.1.2.127**

- **For Subnet-2:** The first bit chosen from the host id part is one and the range will be from (193.1.2.100000000 till you get all 1's in the host ID part i.e, 193.1.2.11111111).

Thus, the range of subnet-2 is: **193.1.2.128 to 193.1.2.255**

Note:

1. To divide a network into four (2^2) parts you need to choose two bits from the host id part for each subnet i.e, (00, 01, 10, 11).
2. To divide a network into eight (2^3) parts you need to choose three bits from the host id part for each subnet i.e, (000, 001, 010, 011, 100, 101, 110, 111) and so on.
3. We can say that if the total number of subnets in a network increases the total number of usable hosts decreases.

1. NIC(Network Interface Card)

NIC or Network Interface Card is a network adapter used to connect the computer to the network. It is installed in the computer to establish a LAN. It has a unique ID that is written on the chip, and it has a connector to connect the cable to it. The cable acts as an interface between the computer and the router or modem. NIC card is a layer 2 device, which means it works on the network model's physical and data link layers.

Types of NIC

- **Wired NIC:** Cables and Connectors use Wired NIC to transfer data.
- **Wireless NIC:** These connect to a wireless network such as Wifi, Bluetooth, etc.

2. HUB

A hub is a multi-port repeater. A hub connects multiple wires coming from different branches, for example, the connector in star topology which connects different stations. Hubs cannot filter data, so data packets are sent to all connected devices. In other words, the collision domain of all hosts connected through hub remains one. Hub does not have any routing table to store the data of ports and map destination addresses., the routing table is used to send/broadcast information across all the ports.

Types of HUB

- **Active HUB:** Active HUB regenerates and amplifies the electric signal before sending them to all connected device. This hub is suitable to transmit data for long distance connections over the network.
- **Passive HUB:** As the name suggests it does not amplify or regenerate electric signal, it is the simplest types of Hub among all and it is not suitable for long-distnace connections.
- **Switching HUB:** This is also known as intelligent HUB, they provide some additional functionality over active and passive hubs. They analyze data packets and make decisions based on MAC address and they are operated on DLL(Data Link Layer).

3. Router

A Router is a device like a switch that routes data packets based on their IP addresses. The router is mainly a Network Layer device. Routers normally connect LANs and WANs and have a dynamically updating routing table based on which they make decisions on routing the data packets. The router divides the broadcast domains of hosts connected through it.

4. Modem

A Modem is a short form of Modulator/Demodulator. The Modem is a hardware component/device that can connect computers and other devices such as routers and switches to the internet. Modems convert or modulate the analog signals coming from telephone wire into a digital form that is in the form of 0s and 1s.

5. Switch

A Switch is a multiport bridge with a buffer and a design that can boost its efficiency(a large number of ports implies less traffic) and performance. A switch is a data link layer device. The switch can perform error checking before forwarding data, which makes it very efficient as it does not forward packets that have errors and forward good packets selectively to the correct port only.

7. Media

It is also known as Link which is going to carry data from one side to another side. This link can be Wired Medium (Guided Medium) and Wireless Medium (Unguided Medium). It is of two types:

7.1 Wired Medium

- **Ethernet:** Ethernet is the most widely used LAN technology, which is defined under IEEE standards 802.3. There are two types of Ethernet:
- **Fibre Optic Cable:** In fibre optic cable data is transferred in the form of light waves.
- **Coaxial Cable:** Coaxial Cable is mainly used for audio and video communications.
- **USB Cable:** USB Stands for Universal Serial Bus it is mainly used to connect PCs and smartphones.

8. Repeater

Repeater is an important component of computer networks as it is used to regenerate and amplify signal in the computer networks. Repeaters are used to improve the quality of the networks and they are operated on the Physical Layer of the OSI Model.

9. Server

A server is a computer program that provides various functionality to another computer program. The server plays a vital role in facilitating communication, data storage, etc. Servers have more data storage as compared to normal computers. They are designed for the specific purpose of handling multiple requests from clients.