1. Standard Template Library (STL) in C++

1.1 Sequence Containers

Vector

- `vector<int> v;` // Declare
- `v.push_back(x);` // Insert at end
- `v.pop_back();` // Remove last element
- `v.size();` // Get size
- `v.begin(), v.end();` // Iterators

Deque

- `deque<int> dq;`
- `dq.push_front(x);` // Insert at front
- `dq.push_back(x);` // Insert at end
- `dq.pop_front();` // Remove front
- `dq.pop_back();` // Remove back

List

- `list<int> lst;`
- `lst.push_front(x);,lst.push_back(x);`
- `lst.insert(iterator, x);`
- `lst.erase(iterator);`

1.2 Associative Containers

Set (Stores unique, sorted elements)

- `set<int> s;`
- `s.insert(x);`
- `s.erase(x);`
- `s.count(x);` // Check existence

Map (Key-Value store, sorted by key)

- `map<string, int> mp;`
- `mp["Alice"] = 90;`
- `mp.erase("Alice");`
- `mp.count("Bob");` // Check existence

Unordered Map (Faster but unsorted)

- `unordered_map<int, int> ump;`

2. Important Algorithms

2.1 Sorting

- `sort(arr, arr+n);` // Ascending
- `sort(arr, arr+n, greater<int>());` // Descending

2.2 Searching

- `binary_search(arr, arr+n, x);`
- `lower_bound(arr, arr+n, x);`
- `upper_bound(arr, arr+n, x);`

3. Graph Algorithms

3.1 _____BFS

```
#include <iostream>

#include <queue>

#include <vector>



#define N 100 // Maximum number of nodes



std::vector<int> adj[N];  // Adjacency list for the graph

bool visited[N];  // Visited array to keep track of visited nodes



// BFS implementation in C++98

void bfs(int start) {

    std::queue<int> q;

    visited[start] = true;
```

```
    q.push(start);

    while (!q.empty()) {
        int node = q.front(); // Get the front node of the queue
        q.pop(); // Remove the front node from the queue

        std::cout << "Visited node " << node << std::endl;

        // Traverse all the neighbors of the current node
        for (int i = 0; i < adj[node].size(); ++i) {
            int neighbor = adj[node][i];
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    // Example graph for BFS
    adj[0].push_back(1);
    adj[0].push_back(2);
    adj[1].push_back(3);
    adj[2].push_back(3);
    adj[3].push_back(4);
```

```
    // Initialize visited array to false
    for (int i = 0; i < N; ++i) {
        visited[i] = false;
    }

    int start = 0;  // Starting node for BFS
    bfs(start);

    return 0;
}
```

_____DFS

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

#define N 100 // Adjust N based on your problem size

std::vector<int> adj[N];  // Adjacency list
bool visited[N];  // Visited array for DFS

// DFS implementation for C++98
void dfs(int node) {
    visited[node] = true;
    // Use a regular for loop instead of range-based for loop
```

```
    for (int i = 0; i < adj[node].size(); ++i) {
        int neighbor = adj[node][i];
        if (!visited[neighbor]) {
            dfs(neighbor);
        }
    }
}

int main() {
    // Example graph for DFS
    adj[0].push_back(1);
    adj[1].push_back(2);
    adj[1].push_back(3);
    adj[2].push_back(4);

    int start = 0;
    dfs(start);

    // Output visited nodes
    for (int i = 0; i < N; ++i) {
        if (visited[i]) std::cout << "Visited node " << i << std::endl;
    }

    return 0;
}
```

3.2 Dijkstra (Shortest Path)

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>



#define N 100 // Number of nodes in the graph

#define INF INT_MAX



std::vector<std::pair<int, int>> adj[N];  // Adjacency list with (neighbor, weight)



// Dijkstra’s algorithm using priority queue (compatible with C++98)

void dijkstra(int start) {

    std::vector<int> dist(N, INF);  // Distance array initialized to infinity

    dist[start] = 0;



    // Priority queue to store pairs (distance, node)

    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<std::pair<int, int>>> pq;

    pq.push(std::make_pair(0, start));  // Start node with distance 0



    while (!pq.empty()) {

        int u = pq.top().second;

        int d = pq.top().first;

        pq.pop();
```

```
        // If a shorter path is found, continue
        if (d > dist[u]) continue;

        // Iterate over all neighbors of node u
        for (int i = 0; i < adj[u].size(); ++i) {
            int v = adj[u][i].first;  // Neighbor node
            int weight = adj[u][i].second;  // Edge weight

            // Relaxation step
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push(std::make_pair(dist[v], v));
            }
        }
    }

    // Output the distances from the source node
    for (int i = 0; i < N; ++i) {
        if (dist[i] == INF) {
            std::cout << "Node " << i << " is unreachable." << std::endl;
        } else {
            std::cout << "Distance to node " << i << " is " << dist[i] <<
std::endl;
        }
    }
```

```
}

int main() {
    // Example graph for Dijkstra's algorithm
    adj[0].push_back(std::make_pair(1, 2));
    adj[0].push_back(std::make_pair(2, 4));
    adj[1].push_back(std::make_pair(2, 1));
    adj[2].push_back(std::make_pair(3, 1));
    adj[3].push_back(std::make_pair(4, 3));

    int start = 0;
    dijkstra(start);

    return 0;
}
```

4. Dynamic Programming (DP)

4.1 Fibonacci using DP

```
#include <iostream>
#define N 1000 // Maximum size for dp array

int dp[N];  // DP array to store Fibonacci numbers

// Function to calculate Fibonacci number using DP
int fib(int n) {
    // If n <= 1, return n (base case)
```

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

    // If the value has already been computed, return it
    if (dp[n] != -1) return dp[n];

    // Otherwise, calculate the value and store it
    dp[n] = fib(n-1) + fib(n-2);

    return dp[n];
}

int main() {
    // Initialize dp array to -1 for all elements
    for (int i = 0; i < N; i++) {
        dp[i] = -1;
    }

    int n = 10;  // Calculate the Fibonacci number for n = 10
    std::cout << "Fibonacci of " << n << " is: " << fib(n) << std::endl;

    return 0;
}
```

5. Number Theory

5.1 Prime Check (Sieve of Eratosthenes)

```
#include <iostream>
```

```
#include <vector>

#define N 100 // Choose a reasonable size for the array

// Function to initialize the isPrime array to true
void sieve(int n) {
    // Manually initialize the array to true
    bool isPrime[N];
    for (int i = 0; i <= n; i++) {
        isPrime[i] = true; // Initialize all entries to true
    }

    // Implement the sieve
    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false; // Mark multiples of i as not prime
            }
        }
    }

    // Print all primes
    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            std::cout << i << " ";
        }
```

```
    }

    std::cout << std::endl;

}


int main() {

    int n = 30; // Set a number limit to test the sieve

    sieve(n);

    return 0;

}
```

6. String Algorithms

6.1 KMP Algorithm (Substring Search)

```
#include <iostream>

#include <vector>

#include <string>

std::vector<int> kmp(std::string s, std::string pattern) {

    int m = pattern.size();



    // Initialize the LPS array manually for C++98 compatibility

    std::vector<int> lps(m, 0); // Initializes the vector of size m with all
values set to 0

    int length = 0; // length of the previous longest prefix suffix

    int i = 1; // index for pattern



    // Compute LPS array

    while (i < m) {
```

```
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }

    return lps;
}

int main() {
    std::string s = "ABABDABACDABABCABAB";
    std::string pattern = "ABABCABAB";

    std::vector<int> lps = kmp(s, pattern);

    // Print the LPS array
    for (size_t i = 0; i < lps.size(); ++i) {
        std::cout << lps[i] << " ";
```

```
    }


    return 0;

}
```

7. Bit Manipulation

- `x & (x - 1)` // Turns off rightmost set bit
- `x | (1 << n)` // Set nth bit
- `x & (1 << n)` // Check nth bit

Bitwise Swap :

```
#include <iostream>
#include <vector>
int main() {
   int a=10,b=5;
   a ^= b;
   b ^= a;
   a ^= b;
   std::cout <<a << " " << b;
   return 0;
}
```