Contents

1 Priority Queue

```cpp
priority_queue<int, vector<int>, greater<int>> p; // min heap
priority_queue<int, vector<int>, less<int>> p; // max heap
```

2 Connected Components

```cpp
int bfs(bool *visited, vector<int> *adjList, int *buffer, int size, int start) {
    int count = 0;
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        start = q.front();
        q.pop();
        for (auto i : adjList[start]) {
            if (!visited[i]) {
                visited[i] = true;
                buffer[count++] = i;
                q.push(i);
            }
        }
    }
    return count;
}
```

3 BFS

```cpp
/**
 * To compile and run:
 * clang++ -o bfs bfs.cpp && ./bfs
 */
#include <bits/stdc++.h>
```

```cpp
using namespace std;

int n;

/**
 * Unweighted shortest path with BFS
 * @param graph: adjacency list, directed or undirected
 * @param s: starting edge
 *
 * distance: min distance of nodes to s
 * path: last node to s
 */
void bfs(vector<int>* graph, int s){
    queue<int> q;
    int v, w;
    q.push(s);

    int* distance = new int[n];
    int* path = new int[n];
    memset(distance, -1, n * sizeof(int));
    memset(path, -1, n * sizeof(int));

    distance[s] = 0;
    while(!q.empty()){
        v = q.front();
        q.pop();
        for(int i = 0; i < graph[v].size(); i++){
            w = graph[v][i];
            if(distance[w] == -1){
                distance[w] = distance[v] + 1;
                path[w] = v;
                q.push(w);
                cout<<"("<<v<<", "<<w<<")"<<endl;
            }
        }
    }
}

int main(){
    cin>>n;
    auto graph = new vector<int>[n]();
    for(int i = 0; i < n; i ++){
        int t;
        cin>>t;
        for(int j = 0; j < t; j++){
            int s;
            cin>>s;
            graph[i].push_back(s);
        }
    }

    bfs(graph, 0);

    return 0;
```

```
}
```

4 Dijkstra

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

constexpr int SIZE = 20;
double graph[SIZE][SIZE];
double *dijkstra(int n, int v) {
    auto *distance = new double[n];
    distance[v] = 0;
    for (int j = 1; j < n; ++j)
        distance[j] = 10e300; //numeric_limits<double>::max();

    auto comp = [=](int a, int b) {
        return distance[a] > distance[b];
    };
    priority_queue<int, vector<int>, decltype(comp)> q(comp);
    q.push(v);
    while (!q.empty()) {
        v = q.top();
        q.pop();
        for (int i = 0; i < n; i++) {
            auto weight = graph[v][i];
            if (weight == 0) continue;
            if (distance[v] + weight < distance[i]) {
                distance[i] = distance[v] + weight;
                q.push(i);
            }
        }
    }
    return distance;
}

int main() {
    auto *dist = dijkstra(SIZE, 0);
    for (int i = 0; i < SIZE; i++) {
        cout << dist[i] << " ";
    }
    cout << endl;
    return 0;
}
```

5 Line Equations

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

/**
 * parametric line equation
 * <x, y> = s * <v1, v2> + <b1, b2>
 */
```

```cpp
struct Line {
    double v1;
    double v2;
    double b1;
    double b2;
};

struct Point {
    double x;
    double y;
};

Line lines[100];

Point intercept(const Line &l1, const Line &l2) {
    double temp = l2.v1 * l1.v2 - l2.v2 * l1.v1;
    if (abs(temp) < 1e-7)
        return {10e300, 10e300};
    double w1 = l1.b1 - l2.b1;
    double w2 = l1.b2 - l2.b2;
    double s = (l2.v2 * w1 - l2.v1 * w2) / temp;
    return {l1.v1 * s + l1.b1, l1.v2 * s + l1.b2};
}

double dist(const Point &p1, const Point &p2) {
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        int x1, y1, x2, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        double dx = x2 - x1;
        double dy = y2 - y1;
        lines[i].v1 = dx;
        lines[i].v2 = dy;
        lines[i].b1 = x1;
        lines[i].b2 = y1;
    }
}
```

6 Non Prime

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

constexpr int MAX = 2000001;
uint16_t not_prime[MAX] = { 0 };

```

```cpp
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    for (int x = 2; x * x < MAX; x++) {
        if (!not_prime[x]) {
            for (int i = x * x; i < MAX; i += x)
                not_prime[i] = 1;
        }
    }
    for (int x = 2; x < MAX; x++) {
        if (not_prime[x]) {
            for (int i = x; i < MAX; i += x)
                not_prime[i] += 1;
        }
    }
    for (int j = 2; j < MAX; ++j) {
        not_prime[j] += not_prime[j] == 0;
    }

    int q, i;
    cin >> q;
    for (int _ = 0; _ < q; ++_) {
        cin >> i;
        cout << not_prime[i] << '\n';
    }
    return 0;
}
```

7 Prim

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

constexpr int SIZE = 2000;
double graph[SIZE][SIZE];
double prim(int n) {
    int start = 0;
    auto comp = [](auto &a, auto &b) { return a.second > b.second; };
    priority_queue<pair<int, double>, pair<int, double>, decltype(comp)> q(comp);

    auto *visited = new bool[n]();
    auto *keys = new double[n]();
    keys[0] = 0;
    for (int i = 1; i < n; i++) {
        keys[i] = 10e300;
    }
    q.push({start, 0});
    while (!q.empty()) {
        start = q.top().first;
        q.pop();
        visited[start] = true;
        for (int i = 0; i < n; i++) {
```

```
            double weight = graph[start][i];
            if (!visited[i] && weight < keys[i]) {
                keys[i] = weight;
                q.push({i, weight});
            }
        }
    }

    delete[] visited;
    delete[] keys;
    return accumulate(keys, keys + n, 0.0);
}
```

8 Prime Sieve

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

#define MAX 123456
bool prime[MAX];
int main() {
    memset(prime, true, sizeof(prime));
    for (int x = 2; x * x < MAX; x++) {
        if (prime[x]) {
            for (int i = x * x; i < MAX; i += x)
                prime[i] = false;
        }
    }
}
```

9 TopSort

```
/**
 * To compile and run:
 * clang++ -o topsort topsort.cpp && ./topsort
 */
#include <bits/stdc++.h>
using namespace std;

/**
 * @param graph is the adjacency list, graph[a] contains b if a is pre-req of b
 * @param degrees is the in-degree of nodes in graph, e.g. 0 if no pre-req
 *
 * @returns topological order of the graph
 */
vector<int> topsort(vector<vector<int>>& graph, int* degrees){
    int counter = 0;
    queue<int> q;
    vector<int> result;

    for(int i = 0; i < graph.size(); i++){
        if(degrees[i] == 0){
            q.push(i);
```

```
        }
    }

    while(!q.empty()){
        int idx = q.top();
        q.pop();
        result.push_back(idx);
        for(auto v : graph[idx]){
            degrees[v] -= 1;
            if(degrees[v] == 0){
                q.push(v);
            }
        }
        counter++;
    }
}

int main(){

}
```