Everyone's Connected

Contents

1 Template

```cpp
#include <bits/stdc++.h>
#define endl '\n'
#define ll long long int
#define ull unsigned long long int
#define MOD7 1000000007
#define MOD9 1000000009
#define MAX 1000001

using namespace std;

/*------------------SOLBEGIN------------------*/

void solve() {
    return;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int t = 1; cin >> t;
    while (t--) solve();

    return 0;
}
```

2 Data structures

2.1 STL Algorithms

STL stands for Standard Template Library. It is a library that provides several generic classes and functions, allowing programmers to manipulate data structures in an easy and efficient way. The STL provides a range of algorithms which can be used to manipulate data stored in containers. The following list shows some of the algorithms provided by the STL and its functions:

Non-Manipulating Algorithms

- **sort(first_iterator, last_iterator)** - Sorts the elements in the range [first, last) in ascending order.
- **sort(frst_iterator, last_iterator, greater<int>())** - Sorts elements inside the vector, in descending order.

- **reverse(first_iterator, last_iterator)** - Reverses elements inside a vector.
- ***max_element(first_iterator, last_iterator)** - Finds the maximum element of a vector.
- ***min_element(first_iterator, last_iterator)** - Finds the minimum element of a vector.
- **accumulate(first_iterator, last_iterator, initial value of sum)** - Summates all the vector elements.
- **count(first_iterator, last_iterator, x)** - Counts all occurrences 'x' inside a vector.
- **find(first_iterator, last_iterator, x)** - Returns an iterator to the first occurrence of 'x' in vector and points to last address if the element is not present.
- **binary_search(first_iterator, last_iterator, x)** - Tests if 'x' exists in sorted vector or not.
- **lower_bound(first_iterator, last_iterator, x)** - Returns an element pointing to the first element in range [first, last), which has a value less than 'x'.
- **upper_bound(first_iterator, last_iterator, x)** - Returns an element pointing to the first element in range [first, last), which has a value greater than 'x'.

Manipulating Algorithms

- **arr.erase(position to delete)** - Erases selected element in vector and shifts and resizes it accordingly.
- **arr.erase(unique(arr.begin(), arr.end()), arr.end())** - Erases the duplicate occurrences in sorted vector in a single line.
- **next_permutation(first_iterator, last_iterator)** - Modifies the vector to its next permutation.
- **prev_permutation(first_iterator, last_iterator)** - Modifies the vector to its previous permutation.
- **distance(first_iterator, desired_iterator)** - Returns the distance of the desired position from the first iterator to a desired one.

2.2 Binary Search

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

vector<int> vec;

int binary_search_first_occurrence(const vector<int>& vec, int value) {
    // Binary search algorithm finds the first occurrence of a value in a sorted vector
    // Declare left and right pointers
    int left = 0;
    int right = vec.size() - 1;
    int result = -1;
    // While left and right pointers do not cross, keep searching
    while (left <= right) {
        // Calculate the middle element of the vector
        int mid = left + (right - left) / 2;
        // If the middle element is the value we are looking for, return its index
        if (vec[mid] == value) {
            result = mid;
            // left = mid + 1;  // Continue searching in the right half (for last occurrence)
            right = mid - 1;  // Continue searching in the left half
        // If the middle element is smaller than the value we are looking for, search in the right half
        } else if (vec[mid] < value) {
            left = mid + 1;
        // If the middle element is greater than the value we are looking for, search in the left half
        } else {
            right = mid - 1;
        }
    }
    return result;  // Returns -1 if value is not found
}

int main() {
    // Assign the variable value to the value you want to search
    int elements, value = 0;
    cin >> elements;
    // Read the elements of the vector
```

```
    for (int i = 0; i < elements; i++) {
        int x;
        cin >> x;
        vec.push_back(x);
    }
    cout << binary_search_first_occurrence(vec, value);

    return 0;
}
```

2.3 Simplified DSU (Stolen from GGDem)

2.4 Disjoint Set Union

2.5 Segment Tree

2.6 Segment Tree Lazy

2.7 Trie

3 Graphs

3.1 Graph Transversal

3.1.1 BFS

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

vector<bool> visited;
vector<vector<int>> adj;

void breadth_first_search(int node) {
    // BFS requieres queue data structure, starting from a given initial
        node
    queue<int> q;
    q.push(node);
    visited[node] = true;
    // While queue is not empty, pop the first element and push its
        children
    while (!q.empty()) {
        int v = q.front();
        cout << v << "␣";
        q.pop();
        // Push all children of v
        for (int u : adj[v]) {
            // If not visited, push and mark as visited
            if (!visited[u]) {
                q.push(u);
                visited[u] = true;
            }
        }
    }
}

int main() {
    int nodes, edges;
    cin >> nodes >> edges;
    // Initialize visited and adjacency list
    visited.assign(nodes, false);
    adj.assign(nodes, vector<int>());
    int u, v;
    // Values of nodes, given as pairs
    for (int i = 0; i < edges; i++) {
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // <- Assuming undirected graph
    }
    breadth_first_search(0); // Start BFS from node x

    return 0;
}
```

3.1.2 DFS

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

vector<bool> visited;
vector<vector<int>> adj;

void depth_first_search(int node) {
    // DFS requieres stack data structure, starting from a given initial
        node
    visited[node] = true;
    cout << node << '␣';
    // For each child of node, if it hasn't been visited, call DFS
        function
```

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

int main() {
    int nodes, edges;
    cin >> nodes >> edges;
    // Initialize visited and adjacency list
    visited.assign(nodes, false);
    adj.assign(nodes, vector<int>());
    // Values of nodes, given as pairs
    for(int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // <- Assuming undirected graph
    }
    // For each node, if it hasn't been visited, call DFS function
    for(int i = 0; i < nodes; i++) {
        if(!visited[i]) {
            depth_first_search(i);
        }
    }

    return 0;
}
```


4 Math

4.1 Identities

Coeficientes binomiales.

$(a+b)^n = \Sigma_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$

$\binom{n}{k} = \binom{n}{n-k}$

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

$k\binom{n}{k} = n\binom{n-1}{k-1}$

$\Sigma_{k=0} n\binom{n}{k} = 2^n$

$\Sigma_{k=0}^{n} (-1)^k \binom{n}{k} = 0$

$\binom{n+m}{t} = \Sigma_{k=0}^{t} \binom{n}{k}\binom{m}{t-k}$

$\Sigma_{j=k}^{n} \binom{j}{k} = \binom{n+1}{k+1}$

Numeros Catalanes.

$C_n = \frac{2(2n-1)}{n+1} C_{n-1}$

$C_n = \frac{1}{n+1}\binom{2n}{n}$

$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$

$\Sigma(n) = O(\log(\log(n)))$ (number of divisors of n)

$F_{2n+1} = F_n^2 + F_{n+1}^2$

$F_{2n} = F_{n+1}^2 - F_{n-1}^2$

$\sum_{i=1}^{n} F_i = F_{n+2} - 1$

$F_{n+i}F_{n+j} - F_n F_{n+i+j} = (-1)^n F_i F_j$

(Möbius Function)

0 if n is square-free

1 if n got even amount of distinct prime factors

0 if n got odd amount of distinct prime factors

(Möbius Inv. Formula)

Let $g(n) = \sum_{d|n} f(d)$, then $f(n) = \sum d \mid ng(d)\mu\left(\frac{n}{d}\right)$.

Permutaciones objetos repetidos

$P(n,k) = \frac{P(n,k)}{n_1! n_2! \ldots}$

Separadores, Ecuaciones lineares a variables $= b$

$\binom{\binom{a}{b}}{} = \binom{a+b-1}{b} = \binom{a+b-1}{a-1}$

Teorema chino

sean $\{n_1, n_2, ..., n_k\}$ primos relativos

$P = n_1 \cdot n_2 \cdot ... \cdot n_k$

$P_i = \frac{P}{n_i}$

$x \cong a_1(n_1)$

$x \cong a_2(n_2)$ ... $x \cong a_k(n_k)$

$P_1 S_1 \cong 1(n_1)$ Donde S soluciones.

$x = P_1 S_1 a_1 + P_2 S_2 a_2 ... P_k S_k a_k$

4.2 Binary Exponentiation and modArith

4.3 Modular Inverse (dividir mod)

4.4 Modular Binomial Coeficient and Permutations

4.5 Non-Mod Binomial Coeficient and Permutations

4.6 Modular Catalan Numbers

4.7 Ceil Fraccionario

```
long long int ceil(long long int numerator, long long int denominator) {
    return (numerator + denominator - 1) / denominator;
}
```

4.8 Numeros de Fibonacci

4.9 Sieve Of Eratosthenes

```
#include <bits/stdc++.h>
#define MAX 1000001
using namespace std;

// Define both prime and pfix arrays
bool prime[MAX];
int pfix[MAX];

// Sieve of Eratosthenes
void sieve() {
    // Set all numbers as prime
    memset(prime, true, sizeof(prime));
    // 0 and 1 are not prime
    prime[0] = prime[1] = false;
    // Iterate over all numbers
    for (int p = 2; p * p < MAX; p++)
        if (prime[p]) for (int i = p * p; i < MAX; i += p) prime[i] =
            false;
    // Calculate prefix sum of prime numbers
    for (int i = 2; i < MAX; i++) {pfix[i] = pfix[i - 1] + prime[i];}
}
```

4.10 Sieve-based Factorization

4.11 Cycle Finding

4.12 Berlekamp Massey

4.13 Modular Berlekamp Massey

4.14 Matrix exponentiation

4.15 Ecuaciones Diofantinas

4.16 Pollard-Rho, Stolen from GGDem

4.17 FFT, Stolen from GGDem

4.18 Euler Totient Function

5 Geometry

6 Strings

6.1 Explode by token

```
vector<string> explode_by_token(string const& s, char delimeter) {
  vector<string> result;
    // Create a string stream from the string, allowing to perform input
        /output operations on strings.
  istringstream iss(s);
    // Read the string stream, tokenizing it by the delimeter
  for(string token; getline(iss, token, delimeter);) {
        // Split the string by the delimeter and push it to the result
            vector
    result.push_back(move(token));
  }
    // Return the result vector
  return result;
```

```
12 }
```


Particularly useful when testing number theoretical solutions.