Codesnatchers

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
```

```cpp
    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

```cpp
#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

```cpp
#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;
}
```

3.2 Topological Sort

3.3 APSP: Floyd Warshall

3.4 SSSP

3.4.1 Lazy Dijkstra

```
// Lazy version of Dijkstra's algorithm usign priority queue
// Works with negative weights while there are no negative cycles
// If there are any negative cycles, the algorithm will not work
#include <bits/stdc++.h>
#define GS 1000
#define INF 100000000
using namespace std;

// Define the graph and the distance array
vector<pair<int, int>> graph[GS];
int distance[GS];

void dijkstra(int origin, int size) {
    // Set all distances to INF
    for (int i = 0; i <= size; i++) distance[i] = INF;
    // Create the priority queue and the current pair
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pair<int, int> current;

    // Set the distance to the origin to 0 and push it to the queue
    pq.push(make_pair(0, origin));

    // While the queue is not empty, get the top element and update the distances
    while (!pq.empty()) {
        // Get the top element and pop it
        current = pq.top();
        pq.pop();

        // If the distance is already smaller, continue to next iteration
        if (distance[current.second] < current.first) continue;
        // Update the distance
        distance[current.second] = current.first;

        // Iterate over the neighbors and update the distances
        for (pair<int, int> neighbor : graph[current.second]) {
            // If the new distance is smaller, push it to the queue
            if ((neighbor.second + current.first) < distance[neighbor.first]) {
                pq.push(make_pair(neighbor.second + current.first, neighbor.first));
            }
        }
    }
}
```

3.4.2 Bellman-Ford

3.5 Strongly Connected Components: Kosaraju

3.6 Articulation Points and Bridges: ModTarjan

4 Math

4.1 Identities

Coeficientes binomiales.

$$(a+b)^n = \Sigma_{k=0}^{n}\binom{n}{k}a^{n-k}b^k$$
$$\binom{n}{k} = \binom{n}{n-k}$$
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$
$$k\binom{n}{k} = n\binom{n-1}{k-1}$$
$$\Sigma_{k=0}n\binom{n}{k} = 2^n$$
$$\Sigma_{k=0}^{n}(-1)^k\binom{n}{k} = 0$$
$$\binom{n+m}{t} = \Sigma_{k=0}^{t}\binom{n}{k}\binom{m}{t-k}$$
$$\Sigma_{j=k}^{n}\binom{j}{k} = \binom{n+1}{k+1}$$

Números 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)

$$F_{2n+1} = F_n^2 + F_{n+1}^2$$
$$F_{2n} = F_{n+1}^2 - F_{n-1}^2$$
$$\sum_{i=1}^{n} F_i = F_{n+2} - 1$$
$$F_{n+i}F_{n+j} - F_nF_{n+i+j} = (-1)^n F_iF_j$$

(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

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

Separadores, Ecuaciones lineares a variables $= b$

$$\left(\binom{a}{b}\right) = \binom{a+b-1}{b} = \binom{a+b-1}{a-1}$$

Teorema chino

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

$$P = n_1 \cdot n_2 \cdot ... \cdot n_k$$
$$P_i = \frac{P}{n_i}$$
$$x \cong a_1(n_1)$$
$$x \cong a_2(n_2) \; ... \; x \cong a_k(n_k)$$

$P_1S_1 \cong 1(n_1)$ Donde S soluciones.

$$x = P_1S_1a_1 + P_2S_2a_2...P_kS_ka_k$$

4.2 Binary Exponentiation and Modular Arithmetic

4.2.1 Binary Exponentiation

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

ll inf = 10000000007;

ll bitPow(ll a, ll e) {
    ll res = 1;
    a %= inf;

    // while exponent is greater than zero
    while (e > 0) {
        // if exponent is odd, multiply result by base
        if (e & 1)
            // multiply result by base and take the remainder
            res = (res * a) % inf;
        // square the base and take the remainder
        a = (a * a) % inf;
        // divide the exponent by 2
        e >>= 1;
    }

    return res;
}
```

4.2.2 Modular Arithmetic

Modular airhmetic is a system of arithmetic for integers, which considers the remainder. In modulus, numbers "wrap around" upon reaching a fixed value.

Congruence

A number x mod N is the equivalent of the remainder of the division of x by N. Two numbers a and b are congruent modulo N if they have the same remainder upon division by N. We say that N if a mod $N = b$ mod N.

- **For example:** $54 \equiv 24 \pmod 7$
  Both numbers are congruent modulo 7, since $54 \bmod 7 = 3$ and $24 \bmod 7 = 3$.

Another way of defining this is by saying that a and b are congruent modulo N if their difference $(a - b)$ is an integer multiple of n, that is, if $\frac{a-b}{n}$ has a reminder of 0.

- **For example:** $36 \equiv 10 \pmod{13}$
  36 and 10 are congruent modulo 13, since their difference $36 - 10 = 26$ is a multiple of 13 ($n = 13$).

Addition
Properties of addition in Modular Arithmetic:

1. If $a + b = c$ then $a \pmod N + b \pmod N \equiv c \pmod N$.
2. If $a \equiv b \pmod N$, then $a + k \equiv b + k \pmod N$ for any integer k.
3. If $a \equiv b \pmod N$ and $c \equiv d \pmod N$, then $a + c \equiv b + d \pmod N$.
4. If $a \equiv b \pmod N$, then $-a \equiv -b \pmod N$.

- **For example:** Find the sum of 31 and 148 in modulo 24.
  31 in modulo 24 is 7 and 148 in modulo 24 is 4. Thus, $31 + 148 \equiv 7 + 4 \equiv 11 \pmod{24}$.
- **Another example:** Find the remainder when $123 + 234 + 32 + 56 + 22 + 12 + 78$ is divided by 3.
  We know that $123 \bmod 3 = 0$, $234 \bmod 3 = 0$, $32 \bmod 3 = 2$, $56 \bmod 3 = 2$, $22 \bmod 3 = 1$, $12 \bmod 3 = 0$, and $78 \bmod 3 = 0$. Thus, the sum of all these numbers is $0 + 0 + 2 + 2 + 1 + 0 + 0 = 5$, and $5 \bmod 3 = 2$.

Multiplication
Properties of multiplication in Modular Arithmetic:

1. If $a \cdot b = c$, then $a \pmod N \cdot b \pmod N \equiv c \pmod N$.
2. If $a \equiv b \pmod N$, then $a \cdot k \equiv b \cdot k \pmod N$ for any integer k.
3. If $a \equiv b$ and $c \equiv d \pmod N$, then $a \cdot c \equiv b \cdot d \pmod N$.

- **For example:** What is $(8 \cdot 16) \pmod 7$.
  Since $8 \equiv 1 \pmod 7$ and $16 \equiv 2 \pmod 7$, then $(8 \cdot 16) \equiv (1 \cdot 2) \equiv 2 \pmod 7$.
- **Another example:** What is the remainder when $123 \cdot 234 \cdot 32 \cdot 56 \cdot 22 \cdot 12 \cdot 78$ is divided by 3.
  We know that $123 \equiv 1$, $134 \equiv 2$, $23 \equiv 2$, $49 \equiv 1$, $235 \equiv 1$ and $13 \equiv 1$, therefore: $123 \cdot 234 \cdot 32 \cdot 56 \cdot 22 \cdot 12 \cdot 78 \equiv 1 \cdot 2 \cdot 2 \cdot 1 \cdot 1 \cdot 1 \equiv 4 \equiv 1 \pmod 3$. Leaving a remainder of 1.

```cpp
void modArithmetic (int a, int b, int x) {
    // If the result of adding a and b is greater than x, take the
        remainder of the division by x
    (a + b) % x;
    // If the result of subtracting a and b is less than 0, add x to the
         result and take the modulus again
    (a - b %x + x) % x;
    // If the result of multiplying a and b is greater than x, take the
        remainder of the division by x
    (a * b) % x;
}
```

4.3 Modular Inverse

The modular inverse of an integer a modulo m is an integer x such that $ax \equiv 1 \pmod m$.

- If a and N are integers such that $gcd(a, N) = 1$, then there exists an integer x such that $ax \equiv 1 \pmod N$.
  x is called the modular inverse of a modulo N.

However, $\frac{a}{b} \pmod N$ is not the same as $(\frac{(a \bmod N)}{(b \bmod N)}) \pmod N$.

- Lets take $a = 10$, $b = 2$, and $N = 3$.
  $\frac{10}{2} \pmod 3 = 5 \pmod 3 = 2$; $(\frac{10 \bmod 3}{2 \bmod 3}) \pmod 3 = (\frac{1}{2}) \pmod 3 = 0.5$.
  This discrepancy is due to the fact that division is not always compatible with modular arithmetic.

On the other hand, using the extended Euclidean algorithm, we can find the modular inverse of a modulo N:

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

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

    int x1, y1;
```

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

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

    return gcd;
}
```

4.4 Modular Binomial Coeficient and Permutations

4.5 Non-Mod Binomial Coeficient and Permutations

4.6 Modular Catalan Numbers

4.7 Fractional Ceiling

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

4.8 Fibonacci Numbers

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

int fibonacci(int x) {
    if (x == 0) return 0;
    if (x == 1) return 1;
    return fibonacci(x - 1) + fibonacci(x - 2);
}
```

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];

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

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

void sieveFactorization() {
    // smallest_prime[i] stores the smallest prime factor of i
    int smallest_prime[MAX];

    // Initialize the samllest prime factor of each number
    for (int i = 2; i < MAX; i++)
        // If i is prime, then the smallest prime factor of i is i,
            otherwise is the smallest prime factor of i
        smallest_prime[i] = (i % 2 == 0 ? 2 : i);

    // Iterate over all odd numbers
    for (int i = 3; i * i < MAX; i += 2)
        if (smallest_prime[i] == i)
            // Marks the smallest prime factor of all multiples of i as
                i, but only if it is the smallest prime factor
            for (int j = i * i; j < MAX; j += i)
                smallest_prime[j] = min(smallest_prime[j],
                    smallest_prime[i]);
}
```

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;
}
```

6.2 Multiple Hashings DS

6.3 Permute chars of string

6.4 Longest common subsequence

6.5 KMP

6.6 Suffix Array

6.7 STL Suffix Array

7 Classics

7.1 Job scheduling

7.1.1 One machine, linear penalty

7.1.2 One machine, deadlines

7.1.3 One machine, profit

7.1.4 Two machines, min time

8 Flow

8.1 Dinic, thx GGDem

9 Miscellaneous

9.1 PBDS

9.2 Bit Manipulation

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

// Bitmasks are represented from 30 to 62 bits using signed int and
   signed long long int to avoid problems with two's complement
int main() {
  signed int a, b;

  // To multiply a number by two, just apply a left shift
  a = 1;
  a= a << 3;

```

```
  // To divide a number by two, just apply a right shift
  a = 32;
  a = a >> 3;

    // To turn on the n-th bit of a number, just apply a bitwise OR with
        2^(n-1), turns on the third bit
  a = 1;
  b = 1 << 2;
  a = a | b;

  // To turn off the n-th bit of a number, just apply a bitwise AND with
      the complement of ~2^(n-1), turns off the third bit
  a = 5;
  b = 1 << 2;
  a &= ~b;

    // To check if the n-th bit of a number is on, just apply a bitwise
        AND with 2^(n-1) and check if the result is turned on
  a = 5;
  b = 1 << 2;
  a = a & b;
  cout << (a ? "YES" : "NO") << endl;

    // To reverse the n-th bit of a number, just power the n-th bit with
        2^(n-1)
  a = 5;
  b = 1 << 2;
  a = a ^ b;

    // To obtain the least significant bit of a number that is turned on
        , just apply a bitwise AND with the complement of the number and
        add one
  a = 12;
    log2(a & ((-1) * a)) + 1

    // To turn on all bits of a number
  a = (1<<4)-1;
}
```

9.2.1 Bitmasking

Bitmasking is a technique used in computer programming to solve problems using individual or groups of bits within a binary number.
It is a powerful tool to solve problems that involve subsets, permutations, and combinations; using bitwise operations such as AND, OR, XOR and NOT.
Some bitmasking utilities are:

- **Memory efficiency:** Bitmasks can be used for compact and memory efficient storage big collections of data.
- **Subset, permutation and combination generation:** Can be used to generate all possible subsets, permutations and combinations of a set.
- **Set operations:** Can be used to perform set operations such as union, intersection, and difference.
- **Data masking and filtering:** By selectively turning on or off bits, we can filter out or mask certain data.
- **Optimization:** Algorithm optimization can be achieved using bitmasking, substituing bit-level operations for arithmetic operations.

10 Testing

10.1 Gen and AutoRun testcases

10.1.1 Gen.cpp

10.1.2 Stress testing

10.1.3 Autorun

10.2 Highly Composite Numbers

Particularly useful when testing number theoretical solutions.