

Competitive Programming Library

Too bad to be Accepted 2023/2024

Contents

1 Templates

1.1	Setup	2
1.1.1	IO Manipulation	2
1.1.2	GCC Compiler Optimization (Vectorization)	2
1.2	MOD Template	2
1.3	Macros	3
1.4	Grid Navigation	3
1.5	Integer 128	3

2 Dynamic Programming

2.1	Some dp patterns	4
2.2	DP solutions	5
2.2.1	Max Subarray sum (Kadane's Algorithm)	5
2.2.2	Maximum Subarray Alternating Sum	6
2.2.3	Count number of DISTINCT ordered ways to produce coins sums to x	6
2.2.4	Min absolute difference between 2 elements from (L, R) (DP Ranges)	6
2.2.5	Longest common subsequence between 2 Strings	7

3 Bit Manipulation

3.1	Subset Operations	7
-----	-----------------------------	---

4 Algorithms

4.1	MO	9
4.2	Intervals	9

2

4.2.1	Prefix Sum (L, R) intervals	9
4.2.2	Find subarrays intervals that sum to K Using Map	10
4.3	Ad-hoc	10
4.3.1	Find duplicate	10
4.4	Sorting Algorithms	11
4.4.1	Radix Sort	11
4.4.2	Counting Sort	11

5 Data Structures

12

5.1	Strings	12
5.1.1	Trie (Prefix Tree)	12
5.2	Range Queries	12
5.2.1	Segment Tree	12
5.2.2	Lazy Propegation	14
5.2.3	Fenwick Tree	14
5.2.4	Fenwick UpdateRange	15
5.2.5	2D BIT	15
5.2.6	Sparse Table	16
5.3	Ordered Set	17

6 Counting Principles

17

6.1	nCr	17
6.1.1	Fast nCr	17
6.1.2	Method 1: Pascal's Triangle (Dynamic Programming) - $\mathcal{O}(n^2)$	17
6.1.3	Method 2: Factorial Definition (Modular Inverses) - $\mathcal{O}(n + \log MOD)$	18

7 Graph Theory

7.1	Shortest Path algorithms	18
7.1.1	Dijkstra Algorithm	18
7.1.2	Floyd Warshal Algorithm	19
7.1.3	Bellman Ford Algorithm	19
7.2	Cycle Detection	20
7.2.1	DFS Implementation	20
7.2.2	Another way for undirected graphs	20
7.2.3	General Way	20
7.2.4	DSU Implementation	21
7.3	Algorithms	21
7.3.1	Heavy Light Decomposition	21
7.3.2	Heavy Light Decomposition with lazy SegTree	23
7.3.3	Topological Sort	25

8 Techniques

8.1	Coordinate Compression	25
8.2	Binary to decimal	26
8.3	Decimal to binary	26

9 Number Theory

9.1	Divisors	26
9.1.1	formulas	26
9.2	Primes	27

10 Geometry

10.1	Linearity	28
10.1.1	co-linear points	28
10.2	polygons	28
10.2.1	Polygon formation	28
10.2.2	Rectangle Intersection	28

11 Miscellaneous

11.1	Faster implementations	29
11.1.1	hashes	29
11.1.2	Binary Search the value	29

18 1 Templates**1.1 Setup****1.1.1 IO Manipulation***Input/Output*

```
#include <bits/stdc++.h>

freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);

#define fastIO \
    ios_base::sync_with_stdio(false), cin.tie(nullptr), cout.tie(nullptr);
```

1.1.2 GCC Compiler Optimization (Vectorization)*GCC Opt*

```
// Ref: USACO guide
// will make GCC auto-vectorize for loops and optimizes floating points
// better (assumes associativity and turns off denormals).
#pragma GCC optimize ("Ofast")
// can double performance of vectorized code, but causes crashes on old
// machines.
#pragma GCC target ("avx,avx2,fma")

// slows down run time but throws a Runtime Error if an overflow occurred
#pragma GCC optimize("trapv")
```

1.2 MOD Template

```
constexpr int MOD = 1e9+7; // must be a prime number

int add(int a, int b) {
    int res = a+b;
    if(res >= MOD) return res -= MOD;
}
```

```
int sub(int a, int b) {
    int res = a-b;
    if(res < 0) return res += MOD;
}

int power(int a, int e) {
    int res = 1;
    while(e) {if(e & 1) res = res * a % MOD; a = a * a % MOD;
    e >>= 1;}
    return res;
}

int inverse(int a) {
    return power(a, MOD-2);
}

int div(int a, int b) {
    return a * inverse(b) % MOD;
}
```

MOD Template

1.3 Macros

Macros

```
#define getBit(n, k) (n >> k)
#define ON(n, idx) (n | (1ll << idx))
#define OFF(n, idx) (n & ~(1ll << idx))
#define toggle(n, idx) ((n) ^ (1ll<<(idx)))
#define gray(n) (n ^ (n >> 1))
#define bitCount(x) (__builtin_popcountll(x))
#define clz(x) (__builtin_clzll(x))
#define ctz(x) (__builtin_ctzll(x))
#define uniq(x) x.resize(unique(x.begin(), x.end())-x.begin());

#define angle(a) (atan2((a).imag(), (a).real()))
//#define vec(a, b) ((b)-(a))
#define same(v1, v2) (dp(vec(v1,v2),vec(v1,v2)) < EPS)
#define dotProduct(a, b) ((conj(a)*(b)).real()) // a*b cos(T), if zero ->
prep
```

```
#define crossProduct(a, b) ((conj(a)*(b)).imag()) // a*b sin(T), if zero
-> parallel
//#define length(a) (hypot((a).imag(), (a).real()))
#define normalize(a) ((a)/length(a))
#define rotate0(v, ang) ((v)*exp(point(0,ang)))
#define rotateA(p, ang, about) (rotate0(vec(about,p),ang)+about)
#define reflect0(v, m) (conj((v)/(m))*(m))
#define ceil_i(a, b) (((1l)(a)+(1l)(b-1))/(1l)(b))
#define floor_i(a, b) (a/b)
#define round_i(a, b) ((a+(b/2))/b) // if a>0
#define round_m(a, b) ((a-(b/2))/b) // if a<0
#define round_multiple(n, m) round_i(n,m)*m // round to multiple if
specified element

const double PI = acos(-1.0);
```

1.4 Grid Navigation

Grid Nav

```
// knight moves on a chess board
int dx[] = { -2, -1, 1, 2, -2, -1, 1, 2 };
int dy[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

// Grid up, down, right, left (Moves for Chess Rook)
int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

// Grid cell all neighbours
const int dx[8] = {1, 0, -1, 0, 1, 1, -1, -1}
const int dy[8] = {0, 1, 0, -1, -1, 1, -1, 1};

// Grid Diagonal (Moves for Chess Bishop)
int dx[] = {1, 1, -1, -1};
int dy[] = {1, -1, 1, -1};
```

1.5 Integer 128

i128

```

typedef __int128 i128;

__int128 read() {
    __int128 x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        if (ch == '-') f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}

void print(__int128 x) {
    if (x < 0) {
        putchar('-');
        x = -x;
    }
    if (x > 9) print(x / 10);
    putchar(x % 10 + '0');
}

bool cmp(__int128 x, __int128 y) { return x > y; }

```

2 Dynamic Programming

2.1 Some dp patterns

Maximumu/Minimum path cost

```

const int MAX = 21;
int grid[MAX][MAX];
int mem[MAX][MAX];
int n = 20;
bool valid(int r, int c){
    return r >= 0 && r < n && c >= 0 && c < n;
}

```

```

}

int maxPathSum(int r, int c){
    if(!valid(r,c)){
        return 0;
    }

    if(r == n-1 && c == n-1){
        return mem[r][c] = grid[r][c];
    }
    // available moves
    int path1 = maxPathSum(r+1,c);
    int path2 = maxPathSum(r,c+1);

    return grid[r][c] + max(path1,path2);
}

```

add operators between numbers to get max prod/sum

```

// put +, -, between sequence of numbers such that the sum is divisible by
// k, and maximum as possible
const int MAX = 21;
long long mem[MAX][MAX];
const int n = 20;
int k = 4; // example
int v[20];
int fix(int a){
    return (a % k + k) % k;
}

long long tryAll(int pos, int mod){
    long long &ret = mem[pos][mod];
    if(ret != -1){
        return ret;
    }
    if(pos == n){
        return ret = mod == 0;
    }
    if(tryAll(pos+1,fix(mod + v[pos])) || tryAll(pos+1,fix(mod-v[pos]))){
        return ret = 1;
    }
    return ret = 0;
}

```

pick choices with no two similar consecutive choices

```
// pick minimum of choinces costs with no two similar consecutive choices
const int choices = 4;
const int n = 20;
int MAX = n;
int mem[MAX][choices];
const int OO = 1e6+1;
int minCost(int pos, int lastChoice){
    if(pos == n){
        return 0; // invalid move
    }
    int &ret = mem[pos][lastChoice];

    if(ret != -1){
        return ret;
    }

    ret = OO; // want to minimize
    // let choices are 0, 1, 2
    if(lastChoice != 0){
        ret = min(ret, minCost(pos+1,0));
    }
    if(lastChoice != 1){
        ret = min(ret, minCost(pos+1,1));
    }
    if(lastChoice != 2){
        ret = min(ret, minCost(pos+1,2));
    }
    return ret;
}
```

sum S and max/min Product

```
int maxK;

ll mem[21][101]; // k, and s

// You are given an integer s and an integer k. Find k positive integers
// a1, a2, ..., ak
// such that their sum is equal to s and their product is the maximal
// possible. Return their product.
```

```
ll maxProd(int k, int rem)
{
    if(k == maxK){
        // base case
        if(rem == 0)
            return 1;
        return 0;
    }

    if(rem == 0) // invalid case
        return 0;

    ll &ret = mem[k][rem];

    if(ret != -1)
        return ret;

    ret = 0;

    for (int i = 1; i <= rem; ++i) {
        ll sol = maxProd(k+1, rem - i) * i;
        ret = max(ret, sol);
    }

    return ret;
}
```

2.2 DP solutions

2.2.1 Max Subarray sum (Kadane's Algorithm)

Max Subarray sum

```
int maxSubarraySum(vector<int>& arr, int len) {
    int ans = INT_MIN, cur = 0;

    for (int i = 0; i < len; i++) {
        cur = cur + arr[i];
        if (ans < cur)
            ans = cur;

        if (cur < 0)
```

```
        cur = 0;
    }

    return ans;
}
```

2.2.2 Maximum Subarray Alternating Sum

Maximum Subarray Alternating Sum

```
/* REF: GeeksForGeeks
Input: arr[] = {-4, -10, 3, 5}
Output: 9
Explanation: Subarray {arr[0], arr[2]} = {-4, -10, 3}. Therefore, the sum
of this subarray is 9.
*/
int maxSubarraySumALT(vector<int>& a, int len) {
    int ans = INT_MIN, cur = 0;
    for (int i = 0; i < len; i++) {
        if (i % 2 == 0)
            cur = max(cur + a[i], a[i]);
        else
            cur = max(cur - a[i], -a[i]);

        ans = max(ans, cur);
    }

    cur = 0;

    for (int i = 0; i < len; i++) {
        if (i % 2 == 1)
            cur = max(cur + a[i], a[i]);
        else
            cur = max(cur - a[i], -a[i]);

        ans = max(ans, cur);
    }

    return ans;
}
```

2.2.3 Count number of DISTINCT ordered ways to produce coins sums to x

Count distinct

/*
For example, if the coins are {2,3,5} and the desired sum is 9, there
are 3 ways:

2+2+5
3+3+3
2+2+2+3
*/

```
int n, x;
cin >> n >> x;
vector<int> coins(n);
read(coins);

vector dp(x + 1, 0);

dp[0] = 1;
for (int i = 0; i < n; ++i) {
    for (int j = coins[i]; j <= x; ++j) {
        dp[j] = add(dp[j], dp[j - coins[i]]);
    }
}

cout << dp[x] << endl;
```

2.2.4 Min absolute difference between 2 elements from (L, R) (DP Ranges)

Min absolute difference

```
const int N = 1e4 + 1;

int dp[N][N];

int n;
cin >> n;
vector<int> a(n);
read(a);
```

```
for (int i = 0; i < n; ++i) dp[i][i] = 1e6; // INF, you can't take the
    element with it self
for (int i = 1; i < n; ++i) dp[i - 1][i] = abs(a[i] - a[i - 1]);

for (int len = 3; len <= n; ++len) {
    for (int l = 0, r = len - 1; r < n; ++l, ++r) {
        dp[l][r] = min(dp[l][r - 1], dp[l + 1][r]);
        dp[l][r] = min(dp[l][r], abs(a[l] - a[r]));
    }
}

int q;
cin >> q;
while (q--) {
    int l, r;
    cin >> l >> r;
    --l, --r;

    cout << dp[l][r] << el;
}
```

2.2.5 Longest common subsequence between 2 Strings

$$dp[i][j] = \begin{cases} \max(dp[i-1][j], dp[i][j-1]) & \text{if } A_i \neq B_j \\ dp[i-1][j-1] + 1 & \text{if } A_i = B_j \end{cases}$$

LIS 2 Strings

```
// REF: USACO guide
int longestCommonSubsequence(string a, string b) {
    int dp[a.size()][b.size()];
    for (int i = 0; i < a.size(); i++) { fill(dp[i], dp[i] + b.size(), 0); }
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == b[0]) dp[i][0] = 1;
        if (i != 0) dp[i][0] = max(dp[i][0], dp[i - 1][0]);
    }
    for (int i = 0; i < b.size(); i++) {
        if (a[0] == b[i]) dp[0][i] = 1;
        if (i != 0) dp[0][i] = max(dp[0][i], dp[0][i - 1]);
    }
    for (int i = 1; i < a.size(); i++) {
        for (int j = 1; j < b.size(); j++) {
```

```
if (a[i] == b[j]) {
    dp[i][j] = dp[i - 1][j - 1] + 1;
} else {
    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
}
}
return dp[a.size() - 1][b.size() - 1];
}
```

3 Bit Manipulation

3.1 Subset Operations

count subsets with give sum

```
int countDistinctSubsetsWithSum(vector<int>& arr, int n, int k) {
    // Count distinct subsets of array arr that sum up to k
    vector<int> dp(k + 1, 0);
    dp[0] = 1;
    for (int i = 0; i < n; ++i) {
        for (int j = k; j >= arr[i]; --j) {
            dp[j] += dp[j - arr[i]];
        }
    }
    return dp[k]; // Number of distinct subsets with sum k
}
```

max xor of any subset of elements in the array

```
int maximalSubsetXOR(vector<int>& arr, int n) {
    // Find the maximum XOR of any subset of elements in array arr
    int maxXor = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        int xorSum = 0;
        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) {
                xorSum ^= arr[i];
            }
        }
    }
}
```

```
        maxXor = max(maxXor, xorSum);
    }
    return maxXor;
}
```

min xor of any subset

```
int minimumSubsetXOR(vector<int>& arr, int n) {
    // Find the minimum XOR of any pair of elements in array arr
    int minSubsetXor = INT_MAX;
    for (int mask = 0; mask < (1 << n); ++mask) {
        int xorSum = 0;
        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) {
                xorSum ^= arr[i];
            }
        }
        minSubsetXor = min(minSubsetXor, xorSum);
    }
    return minSubsetXor;
}
```

subset generation

```
void subsetGeneration(int x, int n) {
    // Generate all non-empty subsets of a set represented by an integer x
    for (int subset = x; subset > 0; subset = (subset - 1) & x) {
        // Process subset
        cout << subset << endl;
    }
}
```

check if subset of elements in the array sum up to k

```
void subsetSumCheck(vector<int>& arr, int n, int k) {
    // Check if a subset of elements in array arr sums up to k
    for (int subset = 0; subset < (1 << n); ++subset) {
        int sum = 0;
        for (int i = 0; i < n; ++i) {
            if (subset & (1 << i)) {
```

```
                sum += arr[i];
            }
        }
        if (sum == k) {
            // Found subset with sum k
            cout << "Subset with sum " << k << ": " << subset << endl;
        }
    }
}
```

max subset sum mod m

```
int subsetWithMaxSumModuloM(vector<int>& arr, int n, int m) {
    // Find the maximum subset sum modulo m
    vector<int> dp(m, -1);
    dp[0] = 0;
    int currentMod = 0;
    for (int i = 0; i < n; ++i) {
        currentMod = (currentMod + arr[i]) % m;
        for (int j = 0; j < m; ++j) {
            if (dp[j] != -1) {
                dp[(j + currentMod) % m] = max(dp[(j + currentMod) % m], dp[j] + arr[i]);
            }
        }
        dp[currentMod] = max(dp[currentMod], arr[i]);
    }
    return dp[0]; // Maximum subset sum modulo m
}
```

iterate over all supersets represented by x

```
void iterateOverSupersets(int x, int n) {
    // Iterate over all supersets of a set represented by x
    int subset = x;
    do {
        // Process subset
        cout << subset << endl;
        subset = (subset + 1) | x;
    } while (subset <= (1 << n) - 1);
}
```

4 Algorithms

4.1 MO

MO Algorithm

```
// MO      -> O(N+Q SQRT(N)) <= 10^5

const int N = 1e5+5, M = 1e5+5;
int n, m;
int nums[N], q_ans[M];

struct query {
    int idx, block_idx, l, r;

    query() = default;
    query(int _l, int _r, int _idx) {
        idx = _idx;
        r = _r - 1;
        l = _l - 1;
        block_idx = _l / sqrt(n);
    }

    bool operator <(const query & y) const {
        if(y.block_idx == block_idx) return r < y.r;
        return block_idx < y.block_idx;
    }
};

int freq[N], ans;

void add(int idx) {
    freq[nums[idx]]++;
    if (freq[nums[idx]] == 2) ans++;
}

void remove(int idx) {
    freq[nums[idx]]--;
    if (freq[nums[idx]] == 1) ans--;
}

cin >> n >> m;
for (int i = 0; i < n; ++i) cin >> nums[i];
```

```
vector<query> Query(m);
for (int i = 0; i < m; ++i) {
    int l, r; cin >> l >> r;
    Query[i] = query(l, r, i);
}

sort(Query.begin(), Query.end());
int l0 = 1, r0 = 0;
for (int i = 0; i < m; ++i) {
    while (l0 < Query[i].l) remove(l0++);
    while (l0 > Query[i].l) add(--l0);
    while (r0 < Query[i].r) add(++r0);
    while (r0 > Query[i].r) remove(r0--);
    q_ans[Query[i].idx] = ans;
}
for (int i = 0; i < m; ++i) {
    cout << q_ans[i] << '\n';
}
```

4.2 Intervals

4.2.1 Prefix Sum (L, R) intervals

Prefix Sum (L, R) intervals

// NOTE: works fine with small n or with large memory

```
int main() {
    int n, k;
    cin >> n >> k;

    vector<int> a(n + 1);
    vector<vector<int>> rangesPrefix(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; ++i)
        cin >> a[i];

    int l = 1, r = 1, sum = 0;
    // validate your intervals
    // here the intervals are the ones that have a sum of k
    while (r <= n) {
        sum += a[r];
```

```

while (sum > k) {
    sum -= a[l];
    ++l;
}

while (l <= r && a[l] == 0) {
    if (sum != k)
        break;

    rangesPrefix[r][l]++;

    ++l;
}

if (sum == k) {
    rangesPrefix[r][l]++;
}

++r;
}

// prefix sum the columns
for (int i = 1; i <= n; ++i) {
    for (int j = n - 1; j >= 0; --j) {
        rangesPrefix[i][j] += rangesPrefix[i][j + 1];
    }
}

// prefix sum the rows
for (int i = 0; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        rangesPrefix[j][i] += rangesPrefix[j - 1][i];
    }
}

int q; cin >> q;

while (q--) {
    cin >> l >> r;
    // answer the number of intervals (X, Y) X <= Y that are included
    // between L, R
    cout << rangesPrefix[r][l] - rangesPrefix[l - 1][l] << endl;
}

```

```

}
}

```

4.2.2 Find subarrays intervals that sum to K Using Map

Find subarray intervals that sum to K Using Map

```

int n, k;
cin >> n >> k;

vector<int> a(n + 1);
vector<pair<int, int>> rng;
for (int i = 1; i <= n; ++i)
    cin >> a[i];

map<int, set<int>> prev;
int currSum = 0;

for (int i = 1; i <= n; ++i) {
    currSum += a[i];
    if (currSum == k) {
        rng.push_back({1, i});
    }
    if (prev.find(currSum - k) != prev.end()) {
        for (auto &j : prev[currSum - k]) {
            rng.push_back({j + 1, i});
        }
    }
    prev[currSum].insert(i);
}

```

4.3 Ad-hoc

4.3.1 Find duplicate

Find duplicate using XOR

```

int findDuplicate(int arr[] , int n)
{
    int answer=0;
    //XOR all the elements with 0
}

```

```

for(int i=0; i<n; i++){
    answer=answer^arr[i];
}
//XOR all the elements with no from 1 to n
// i.e answer^0 = answer
for(int i=1; i<n; i++){
    answer=answer^i;
}
return answer;
}

```

4.4 Sorting Algorithms

4.4.1 Radix Sort

Radix Sort

```

// O(n + b), where n is the number of elements and b is the base of the
// number system
// A function to do counting sort of arr[] according to the digit
// represented by exp.
void countingSort(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n); // output array
    int count[10] = {0};

    // Store count of occurrences in count[]
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains the actual
    // position of this digit in output[]
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr now

```

```

// contains sorted numbers according to the current digit
for (int i = 0; i < n; i++)
    arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using Radix Sort
void radixSort(vector<int>& arr) {
    // Find the maximum number to know the number of digits
    int mx = *max_element(arr.begin(), arr.end());

    // Do counting sort for every digit. Note that instead
    // of passing the digit number, exp is passed. exp is 10^i
    // where i is the current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countingSort(arr, exp);
}

```

4.4.2 Counting Sort

Counting Sort

```

// O(N+M), where N and M are the size of inputArray[] and countArray[]
// The main function that sorts arr[] of size n using Counting Sort
void countingSort(vector<int>& arr) {
    int maxElement = *max_element(arr.begin(), arr.end());
    int minElement = *min_element(arr.begin(), arr.end());
    int range = maxElement - minElement + 1;

    vector<int> count(range), output(arr.size());
    for (int i = 0; i < arr.size(); i++)
        count[arr[i] - minElement]++;

    for (int i = 1; i < count.size(); i++)
        count[i] += count[i - 1];

    for (int i = arr.size() - 1; i >= 0; i--) {
        output[count[arr[i] - minElement] - 1] = arr[i];
        count[arr[i] - minElement]--;
    }

    for (int i = 0; i < arr.size(); i++)
        arr[i] = output[i];
}

```

5 Data Structures

5.1 Strings

5.1.1 Trie (Prefix Tree)

Basic Implementation

```
#define MAX_CHAR 26

struct TrieNode {
    TrieNode *pTrieNode[MAX_CHAR]{};
    bool isWord;

    TrieNode() {
        isWord = false;
        fill(pTrieNode, pTrieNode + 26, (TrieNode *) NULL);
    }

    virtual ~TrieNode() = default;
};

class Trie {
private:
    TrieNode *root;
public:
    Trie() {
        root = new TrieNode();
    }

    virtual ~Trie() = default;

    TrieNode *getTrieNode() {
        return this->root;
    }

    void insert(const string &word) {
        TrieNode *current = root;
        for (char c: word) {
            int i = c - 'a';
            if (current->pTrieNode[i] == nullptr)
```

```
                current->pTrieNode[i] = new TrieNode();
            current = current->pTrieNode[i];
        }
        current->isWord = true;
    }

    bool search(const string &word) {
        TrieNode *current = root;
        int ch = 0;
        for (char c: word) {
            ch = c - 'a';
            if (current->pTrieNode[ch] == nullptr)
                return false;
            current = current->pTrieNode[ch];
        }
        return current->isWord;
    }

    bool startsWith(const string &prefix) {
        TrieNode *current = root;
        int ch = 0;
        for (char c: prefix) {
            ch = c - 'a';
            if (current->pTrieNode[ch] == nullptr)
                return false;
            current = current->pTrieNode[ch];
        }
        return true;
    }
};
```

5.2 Range Queries

5.2.1 Segment Tree

Basic Implementation

```
struct Node {
    long long val;
};

struct SegTree {
```

```
private:
    const Node NEUTRAL = {INT_MIN};

    static Node merge(const Node& x1, const Node& x2) {
        return {x1.val + x2.val};
    }

    void set(const int& idx, const int& val, int x, int lx, int rx) {
        if (rx - lx == 1) return void(values[x].val = val);

        int mid = (rx + lx) / 2;

        if (idx < mid)
            set(idx, val, 2 * x + 1, lx, mid);
        else
            set(idx, val, 2 * x + 2, mid, rx);

        values[x] = merge(values[2 * x + 1], values[2 * x + 2]);
    }

    Node query(const int& l, const int& r, int x, int lx, int rx) {
        if (lx >= r || l >= rx) return NEUTRAL;
        if (lx >= l && rx <= r) return values[x];

        int mid = (rx + lx) / 2;

        return merge(query(l, r, 2 * x + 1, lx, mid), query(l, r, 2 * x + 2, mid, rx));
    }

    void build(vector<int> &a, int x, int lx, int rx) {
        if (rx - lx == 1) {
            if (lx < a.size()) {
                values[x].val = a[lx];
            }
            return;
        }
        int m = (lx + rx) / 2;
        build(a, 2 * x + 1, lx, m);
        build(a, 2 * x + 2, m, rx);
        values[x] = merge(values[2 * x + 1], values[2 * x + 2]);
    }

    void assign_range(int l, int r, int node, int lx, int rx, int time,
        int val) {
```

```
        if (lx > r || l > rx) return;
        if (lx >= l && rx <= r) {
            lazy[node] = {time, val};
            return;
        }
        int mid = (lx+rx) / 2;

        assign_range(l, r, 2*node+1, lx, mid, time, val);
        assign_range(l, r, 2*node+2, mid+1, rx, time, val);
    }

    pair<int, int> point_query(int lx, int rx, int node, int idx) {
        if(rx == lx) return lazy[node];
        int mid = (lx+rx) / 2;

        if(idx <= mid) {
            auto x = point_query(lx, mid, 2*node+1, idx);
            if(x.first > lazy[node].first) return x;
            return lazy[node];
        }
        auto x = point_query(mid+1, rx, 2*node+2, idx);
        if(x.first > lazy[node].first) return x;
        return lazy[node];
    }

public:
    int size{};
    vector<Node> values;

    void build(vector<int> &a) {
        build(a, 0, 0, size);
    }

    void init(int _size) {
        size = 1;
        while (size < _size) size *= 2;
        values.assign(2 * size, NEUTRAL);
    }

    void set(int idx, int val) {
        set(idx, val, 0, 0, size);
    }

    Node query(const int& l, const int& r) {
```

```

        return query(l, r, 0, 0, size);
    }
};

```

5.2.2 Lazy Propagation

Lazy Propagation

```

struct SegTree {
private:
    void propagate(int lx, int rx, int node) {
        if(!lazy[node]) return;

        if(lx != rx) {
            lazy[2*node+1] = lazy[node];
            lazy[2*node+2] = lazy[node];
        }
        values[node] = lazy[node] * (rx - lx + 1);
        lazy[node] = 0;
    }

    // assign val in range [l, r]
    void update_range(int l, int r, int node, int lx, int rx, int val,
        bool f) {
        propagate(lx, rx, node);
        if (lx > r || l > rx) return;
        if (lx >= l && rx <= r) {
            lazy[node] = val;
            propagate(lx, rx, node);
            return;
        }
    }
    int mid = (lx+rx) / 2;

    update_range(l, r, 2*node+1, lx, mid, val, f);
    update_range(l, r, 2*node+2, mid+1, rx, val, f);
    values[node] = values[2*node+1] + values[2*node+2];
}

// get sum in range [l, r]
int range_query(int l, int r, int lx, int rx, int node) {
    propagate(lx, rx, node);
    if (lx > r || l > rx) return 0;
    if (lx >= l && rx <= r) return values[node];
}

```

```

        int mid = (lx+rx) / 2;
        return range_query(l, r, lx, mid, 2*node+1) + range_query(l, r, mid
            +1, rx, 2*node+2);
    }
};

public:
    int size{};
    vector<int> values, lazy;

    void init(int _size) {
        size = 1;
        while (size < _size) size *= 2;
        values.assign(2 * size, 0);
        lazy.assign(2 * size, 0);
    }

    void update_range(int l, int r, int v, bool f) {
        update_range(l, r, 0, 0, size-1, v, f);
    }

    int range_query(int l, int r) {
        return range_query(l, r, 0, size-1, 0);
    }
};

```

5.2.3 Fenwick Tree

Fenwick Tree

```

struct Fenwick {
    // One Based
    vector<int> tree;

    explicit Fenwick(int n) {tree.assign(n + 5, {});}

    // Computes the prefix sum from [1, i], O(log(n))
    int query(int i) {
        int res = 0;
        while (i > 0) {
            res += tree[i];
            i &= ~(i & -i);
        }
    }
}

```

```

        return res;
    }

    int query(int l, int r) {
        return query(r) - query(l-1);
    }

    // Get the value at index i
    int get(int i) {
        return query(i, i);
    }

    // Add 'v' to index 'i', O(log(n))
    void update(int i, int v) {
        while (i < tree.size()) {
            tree[i] += v;
            i += (i & -i);
        }
    }

    // Update range, Point query
    // To get(k) do prefix sum [1, k] and in insert update_range(i, i, a[i])
    void update_range(int l, int r, int v) {
        update(l, v);
        update(r+1, -v);
    }
};

```

5.2.4 Fenwick UpdateRange

BIT UpdateRange

```

struct BITUpdateRange {
private:
    int n;
    vector<int> B1, B2;

    void add(vector<int> &b, int idx, int x) {
        while (idx <= n) {
            b[idx] += x;
            idx += idx & -idx;
        }
    }
};

```

```

    }

    int sum(vector<int> &b, int idx) {
        int total = 0;
        while (idx > 0) {
            total += b[idx];
            idx &= ~(idx & -idx);
        }
        return total;
    }

    int prefix(int idx) {
        return sum(B1, idx) * idx - sum(B2, idx);
    }

public:
    explicit BITUpdateRange(int n) : n(n) {
        B1.assign(n + 1, {});
        B2.assign(n + 1, {});
    }

    void update(int l, int r, int x) {
        add(B1, l, x);
        add(B1, r + 1, -x);
        add(B2, l, x * (l - 1));
        add(B2, r + 1, -x * r);
    }

    int query(int i) {
        return prefix(i) - prefix(i - 1);
    }

    int query(int l, int r) {
        return prefix(r) - prefix(l - 1);
    }
};

```

5.2.5 2D BIT

2D BIT

```

struct BIT2D {
    int n, m;
};

```

```

vector<vector<int>> bit;

BIT2D(int n, int m) : n(n), m(m) {
    bit.assign(n + 2, vector<int>(m + 2));
}

void update(int x, int y, int val) {
    for (; x <= n; x += x & -x) {
        for (int i = y; i <= m; i += i & -i) {
            bit[x][i] += val;
        }
    }
}

int prefix(int x, int y) {
    int res = 0;
    for (; x > 0; x &= ~(x & -x)) {
        for (int i = y; i > 0; i &= ~(i & -i)) {
            res += bit[x][i];
        }
    }
    return res;
}

int query(int sx, int sy, int ex, int ey) {
    int ans = 0;
    ans += prefix(ex, ey);
    ans -= prefix(ex, sy - 1);
    ans -= prefix(sx - 1, ey);
    ans += prefix(sx - 1, sy - 1);
    return ans;
}
};

```

5.2.6 Sparse Table

Impl with the index

```

// storing the index also
struct SNode {
    int val;
    int index;
};

```

```

class SparseTable {
private:
    vector<vector<SNode>> table;

    function<SNode(const SNode&, const SNode&)> merge;

    static SNode StaticMerge(const SNode& a, const SNode& b) {
        return a.val < b.val ? a : b;
    }

public:
    explicit SparseTable(const vector<int>& arr, const function<SNode(
        const SNode&, const SNode&)>& mergeFunc = StaticMerge) {
        int n = static_cast<int>(arr.size());
        int log_n = static_cast<int>(log2(n)) + 1;
        this->merge = mergeFunc;

        table.resize(n, vector<SNode>(log_n));

        for (int i = 0; i < n; i++) {
            table[i][0] = {arr[i], i};
        }

        for (int j = 1; (1 << j) <= n; j++) {
            for (int i = 0; i + (1 << j) <= n; i++) {
                table[i][j] = mergeFunc(table[i][j - 1], table[i + (1 << (j
                    - 1))][j - 1]);
            }
        }

        SNode query(int left, int right) {
            int j = static_cast<int>(log2(right - left + 1));
            return merge(table[left][j], table[right - (1 << j) + 1][j]);
        }
    };

    int main(void) {
        int n;
        cin >> n;
        vector<int> arr(n);
        for (auto& element : arr) cin >> element;
    }
};

```



```
SparseTable minSt(arr, [](const SNode& a, const SNode& b) -> SNode {
    return a.val < b.val ? a : b;
});

SparseTable maxSt(arr, [](const SNode& a, const SNode& b) -> SNode {
    return a.val > b.val ? a : b;
});
}
```

5.3 Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void erase_set(ordered_set &os, int v) {
    // Number of elements less than v
    int rank = os.order_of_key(v);

    auto it = os.find_by_order(rank);
    os.erase(it);
}
```

Ordered Set

6 Counting Principles

6.1 nCr

$$C(n, k) = \frac{n!}{(n-k)!k!} = \frac{n * (n-1) * (n-2) * \dots * (n-k+1)}{k!}$$

6.1.1 Fast nCr

$$C(n, k) = \frac{n * (n-1) * (n-2) * \dots * (n-k+1)}{1 * 2 * 3 * \dots * k} = \prod_{i=0}^{k-1} \frac{n-i}{i+1} = \prod_{i=0}^{k-1} (n-i)(i+1)^{-1}$$

Fast nCr

```
int nCr(const int& n, const int& r) {
    double res = 1;
    for (int i = 1; i <= r; ++i)
        res = res * (n - r + i) / i;
    return (int)(res + 0.01);
}
```

6.1.2 Method 1: Pascal's Triangle (Dynamic Programming) - $\mathcal{O}(n^2)$

nCk using dp

```
// REF: USACO guide

/** @return nCk mod p using dynamic programming */
int binomial(int n, int k, int p) {
    // dp[i][j] stores iCj
    vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));

    // base cases described above
    for (int i = 0; i <= n; i++) {
        /*
         * i choose 0 is always 1 since there is exactly one way
         * to choose 0 elements from a set of i elements
         * (don't choose anything)
         */
        dp[i][0] = 1;
        /*
         * i choose i is always 1 since there is exactly one way
         * to choose i elements from a set of i elements
         * (choose every element in the set)
         */
        if (i <= k) { dp[i][i] = 1; }
    }
}
```

```
for (int i = 0; i <= n; i++) {
    for (int j = 1; j <= min(i, k); j++) {
        if (i != j) { // skips over the base cases
            // uses the recurrence relation above
            dp[i][j] = (dp[i - 1][j - 1] + dp[i - 1][j]) % p;
        }
    }
}

return dp[n][k]; // returns nCk modulo p
}
```

6.1.3 Method 2: Factorial Definition (Modular Inverses) - $\mathcal{O}(n + \log MOD)$

nCk using Modular Inverses

// REF: USACO guide

```
const int MAXN = 1e6;
```

```
long long fac[MAXN + 1];
long long inv[MAXN + 1];
```

```
/** @return x^n modulo m in O(log p) time. */
long long exp(long long x, long long n, long long m) {
    x %= m; // note: m * m must be less than 2^63 to avoid ll overflow
    long long res = 1;
    while (n > 0) {
        if (n % 2 == 1) { res = res * x % m; }
        x = x * x % m;
        n /= 2;
    }
    return res;
}
```

```
/** Precomputes n! from 0 to MAXN. */
void factorial(long long p) {
    fac[0] = 1;
    for (int i = 1; i <= MAXN; i++) { fac[i] = fac[i - 1] * i % p; }
}
```

```
/**
 * Precomputes all modular inverse factorials
 * from 0 to MAXN in O(n + log p) time
 */
void inverses(long long p) {
    inv[MAXN] = exp(fac[MAXN], p - 2, p);
    for (int i = MAXN; i >= 1; i--) { inv[i - 1] = inv[i] * i % p; }
}

/** @return nCr mod p */
long long choose(long long n, long long r, long long p) {
    return fac[n] * inv[r] % p * inv[n - r] % p;
}
```

```
int main() {
    factorial();
    inverses();
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int a, b;
        cin >> a >> b;
        cout << choose(a, b) << '\n';
    }
}
```

7 Graph Theory

7.1 Shortest Path algorithms

7.1.1 Dijkstra Algorithm

Dijkstra Implementation

```
#define INF (1e18) // for int defined as ll

int n, m;
vector<vector<pair<int, int>>> adj;
vector<int> cost;
vector<int> parent;
```

```
void dijkstra(int startNode = 1) {
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<>> pq;

    cost[startNode] = 0;
    pq.emplace(0, startNode);

    while (!pq.empty()) {
        int u = pq.top().second;
        ll d = pq.top().first;
        pq.pop();

        if (d > cost[u]) continue;

        for (auto &p: adj[u]) {
            int v = p.first;
            int w = p.second;
            if (cost[v] > cost[u] + w) {
                cost[v] = cost[u] + w;
                parent[v] = u;
                pq.emplace(cost[v], v);
            }
        }
    }
}

void run_test_case(int testNum) {
    cin >> n >> m;

    adj.assign(n + 1, {});
    cost.assign(n + 1, INF);
    parent.assign(n + 1, -1);

    while (m--) {
        // Read Edges
    }

    dijkstra();

    if (cost[n] == INF) {
        cout << -1 << el; // not connected {Depends on you use case}
        return;
    }

    stack<int> ans;
```

```
for (int v = n; v != -1; v = parent[v]) ans.push(v);

while (!ans.empty()) { // printing the path
    cout << ans.top() << ' ';
    ans.pop();
}
cout << el;
}
```

7.1.2 Floyd Warshal Algorithm

Floyd Warshal Implementation

```
int main() {
    int n, m; cin >> n >> m;
    vector <vector <int>> adj(n + 1, vector <int> (n + 1, 2e9));
    for (int i = 0; i < n; i++) adj[i][i] = 0;

    while(m--) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u][v] = min(adj[u][v], w);
        adj[v][u] = min(adj[v][u], w);
    }

    for (int mid = 1; mid <= n; mid++) {
        for (int start = 1; start <= n; start++) {
            for (int end = 1; end <= n; end++) {
                adj[start][end] = min(adj[start][end], adj[start][mid] +
                    adj[mid][end]);
            }
        }
    }

    return 0;
}
```

7.1.3 Bellman Ford Algorithm

BellmanFord Implementation

```
vector <vector <pair<int, int>>> &adj

vector <long long> BellmanFord(int src) {
    int n = (int)adj.size();
    vector <long long> dist(n, 2e18);

    dist[src] = 0;
    for (int it = 0; it < n-1; it++) {
        bool in = false;
        for (int i = 0; i < n; i++) { // iterate on the edges
            for (auto &j, w : adj[i]) {
                if (dist[j] > dist[i] + w) {
                    in = true;
                    dist[j] = dist[i] + w;
                }
            }
        }
        if (!in) return dist;
    }

    for (int i = 0; i < n; i++) {
        for (auto &j, w : adj[i]) {
            if (dist[j] > dist[i] + w) { //negative cycle
                return vector <long long> (n, -1); // or any flag
            }
        }
    }

    return dist;
}
```

7.2 Cycle Detection

7.2.1 DFS Implementation

DFS Implementation

```
// return true with number of nodes in the cycle, either odd cycle or even
bool cycle_detection(unordered_map<int, vector<int>> &graph, int source,
    int par, unordered_map<int, bool> vis, int c){
```

```
    if(vis[source]) return true;

    vis[source] = true;

    for(int v: graph[source]){
        if(v != par){
            c++;
            if(dfs(graph,v, source, vis, c)) return true;
        }
    }
    return false;
}
```

7.2.2 Another way for undirected graphs

Another way for undirected graphs

```
// this is true only for undirected graphs
bool dfs1(int cur, int par) {
    bool ret = false;
    vis[cur] = true;
    for (auto &i : adj[cur]) {
        if (!vis[i]) ret|=dfs1(i, cur);
        else if (par != i) ret = true;
    }
    return ret;
}
```

7.2.3 General Way

General Way

```
// general algorithm
vector <bool> cyc;
bool dfs(int cur, int par) {
    bool ret = false;
    vis[cur] = cyc[cur] = true;
    for (auto &i : adj[cur]) {
        if (par == i) continue;
        if (!vis[i]) ret|=dfs(i, cur);
        else if (cyc[i]) ret = true;
    }
}
```

```
    }  
    cyc[cur] = false;  
    return ret;  
}
```

7.2.4 DSU Implementation

DSU Implementation

```
#include <iostream>  
#include <vector>  
  
class UnionFind {  
public:  
    UnionFind(int n) {  
        parent.resize(n);  
        rank.resize(n, 0);  
        for (int i = 0; i < n; ++i) {  
            parent[i] = i;  
        }  
    }  
  
    int find(int u) {  
        if (parent[u] != u) {  
            parent[u] = find(parent[u]);  
        }  
        return parent[u];  
    }  
  
    void unionSets(int u, int v) {  
        int rootU = find(u);  
        int rootV = find(v);  
  
        if (rootU != rootV) {  
            if (rank[rootU] > rank[rootV]) {  
                parent[rootV] = rootU;  
            } else if (rank[rootU] < rank[rootV]) {  
                parent[rootU] = rootV;  
            } else {  
                parent[rootV] = rootU;  
                ++rank[rootU];  
            }  
        }  
    }  
}
```

```
    }  
  
private:  
    std::vector<int> parent;  
    std::vector<int> rank;  
};  
  
bool detectCycle(const std::vector<std::pair<int, int>>& edges, int n) {  
    UnionFind uf(n);  
  
    for (const auto& edge : edges) {  
        int u = edge.first;  
        int v = edge.second;  
  
        if (uf.find(u) == uf.find(v)) {  
            return true;  
        }  
        uf.unionSets(u, v);  
    }  
  
    return false;  
}  
  
int main() {  
    std::vector<std::pair<int, int>> edges = { {0, 1}, {1, 2}, {2, 3}, {3, 0} };  
    int n = 4; // Number of vertices  
  
    if (detectCycle(edges, n)) {  
        std::cout << "Cycle detected" << std::endl;  
    } else {  
        std::cout << "No cycle detected" << std::endl;  
    }  
  
    return 0;  
}
```

7.3 Algorithms

7.3.1 Heavy Light Decomposition

Basic HLD Impl

```
struct Node {
    int val;
};

const Node nullNode = {0};

const int N = 2e5 + 5, S = 1 << 19;
int n, q;
int val[N];
int sz[N], par[N], dep[N], id[N], top[N];
vector<int> adj[N];

Node st[S];

Node merge(const Node& a, const Node& b) {
    return {a.val + b.val};
}

void update(int idx, Node val) {
    st[idx += n] = val;
    for (idx /= 2; idx; idx /= 2) st[idx] = merge(st[idx * 2], st[idx * 2
        + 1]);
}

Node query(int lo, int hi) {
    Node ra = nullNode, rb = nullNode;

    for (lo += n, hi += n + 1; lo < hi; lo /= 2, hi /= 2) {
        if (lo & 1) ra = merge(ra, st[lo++]);
        if (hi & 1) rb = merge(st[--hi], rb);
    }

    return merge(ra, rb);
}

int dfs_size(const int& node, const int& parent) {
    sz[node] = 1;
    par[node] = parent;
    for (const int& ch : adj[node]) {
        if (ch == parent) continue;
        dep[ch] = dep[node] + 1;
        par[ch] = node;
        sz[node] += dfs_size(ch, node);
    }
}
```

```
    }
    return sz[node];
}

int curId = 0;

void dfs_hld(const int& cur, const int& parent, const int& curTop) {
    id[cur] = curId++;
    top[cur] = curTop;
    update(id[cur], {val[cur]});
    int heavyChild = -1, heavyMax = -1;
    for (const int& ch : adj[cur]) {
        if (ch == parent) continue;
        if (sz[ch] > heavyMax) {
            heavyMax = sz[ch];
            heavyChild = ch;
        }
    }

    if (heavyChild == -1) return;
    dfs_hld(heavyChild, cur, curTop);
    for (int ch : adj[cur]) {
        if (ch == parent || ch == heavyChild) continue;

        dfs_hld(ch, cur, ch);
    }
}

Node path(int u, int v) {
    Node ans = nullNode;

    while (top[u] != top[v]) {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        ans = merge(ans, query(id[top[u]], id[u]));
        u = par[top[u]];
    }

    if (dep[u] > dep[v]) swap(u, v);
    ans = merge(ans, query(id[u], id[v]));
    return ans;
}

void init() {
    for (int i = 0; i < S; i++) st[i] = nullNode;
}
```

```

    dfs_size(1, 1);
    dfs_hld(1, 1, 1);
}

int main() {
    cin >> n >> q;
    for (int i = 1; i <= n; i++) cin >> val[i];

    int a, b;
    for (int i = 2; i <= n; i++) {
        cin >> a >> b;
        adj[a].pb(b);
        adj[b].pb(a);
    }

    init(); // <----- DON'T FORGET TO CALL THIS FUNCTION

    int type;
    while (q--) {
        cin >> type;
        if (type == 1) {
            cin >> a >> b;
            val[a] = b;
            update(id[a], {val[a]});
        }
        else {
            cin >> a;
            cout << path(1, a).val << endl;
        }
    }
}

```

7.3.2 Heavy Light Decomposition with lazy SegTree

Basic HLD Impl

```

struct Node {
    int val;
};

const Node nullNode = {0};

const int N = 2e5 + 5, S = 1 << 19;

```

```

int n, q;
int val[N];
int sz[N], par[N], dep[N], id[N], top[N];
vector<int> adj[N];

Node st[S];
int lazy[S];

Node merge(const Node& a, const Node& b) {
    return {a.val + b.val};
}

void push(int idx, int l, int r) {
    if (lazy[idx] != 0) {
        st[idx].val += lazy[idx] * (r - l + 1);
        if (l != r) {
            lazy[idx * 2] += lazy[idx];
            lazy[idx * 2 + 1] += lazy[idx];
        }
        lazy[idx] = 0;
    }
}

void update_range(int lo, int hi, int l, int r, int idx, int value) {
    push(idx, l, r);
    if (lo > r || hi < l) return;
    if (lo <= l && r <= hi) {
        lazy[idx] += value;
        push(idx, l, r);
        return;
    }
    int mid = (l + r) / 2;
    update_range(lo, hi, l, mid, idx * 2, value);
    update_range(lo, hi, mid + 1, r, idx * 2 + 1, value);
    st[idx] = merge(st[idx * 2], st[idx * 2 + 1]);
}

void update(int idx, Node val) {
    update_range(idx, idx, 0, n - 1, 1, val.val);
}

void update_range(int lo, int hi, int value) {
    update_range(lo, hi, 0, n - 1, 1, value);
}

```

```

Node query(int lo, int hi, int l, int r, int idx) {
    push(idx, l, r);
    if (lo > r || hi < l) return nullNode;
    if (lo <= l && r <= hi) return st[idx];
    int mid = (l + r) / 2;
    return merge(query(lo, hi, l, mid, idx * 2), query(lo, hi, mid + 1, r,
        idx * 2 + 1));
}

Node query(int lo, int hi) {
    return query(lo, hi, 0, n - 1, 1);
}

int dfs_size(const int& node, const int& parent) {
    sz[node] = 1;
    par[node] = parent;
    for (const int& ch : adj[node]) {
        if (ch == parent) continue;
        dep[ch] = dep[node] + 1;
        par[ch] = node;
        sz[node] += dfs_size(ch, node);
    }
    return sz[node];
}

int curId = 0;

void dfs_hld(const int& cur, const int& parent, const int& curTop) {
    id[cur] = curId++;
    top[cur] = curTop;
    update(id[cur], {val[cur]});
    int heavyChild = -1, heavyMax = -1;
    for (const int& ch : adj[cur]) {
        if (ch == parent) continue;
        if (sz[ch] > heavyMax) {
            heavyMax = sz[ch];
            heavyChild = ch;
        }
    }

    if (heavyChild == -1) return;
    dfs_hld(heavyChild, cur, curTop);
    for (int ch : adj[cur]) {

```

```

        if (ch == parent || ch == heavyChild) continue;

        dfs_hld(ch, cur, ch);
    }
}

int get(int u) {
    return query(id[u], id[u]).val;
}

void path(int u, int v, int val) {
    // Node ans = nullNode;

    while (top[u] != top[v]) {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        // ans = merge(ans, query(id[top[u]], id[u]));
        update_range(id[top[u]], id[u], val);
        u = par[top[u]];
    }

    if (dep[u] > dep[v]) swap(u, v);
    // ans = merge(ans, query(id[u], id[v]));
    update_range(id[u], id[v], val);
    // return ans;
}

void init() {
    for (int i = 0; i < S; i++) st[i] = nullNode;
    memset(lazy, 0, sizeof(lazy));
    dfs_size(1, 1);
    dfs_hld(1, 1, 1);
}

int main(void) {
    cin >> n >> q;
    for (int i = 1; i <= n; i++) val[i] = 0;
    int a, b;
    for (int i = 2; i <= n; i++) {
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    init(); // <----- DON'T FORGET TO CALL THIS FUNCTION

```



```
int v;
while (q--) {
    cin >> a >> b >> v;
    path(a, b, v);
}

for (int i = 1; i <= n; i++) {
    cout << get(i) << " ";
}
cout << el;
}
```

7.3.3 Topological Sort

Topological Sort Using DFS

```
//REF: USACO Guide
vector<int> top_sort;
vector<vector<int>> graph;
vector<bool> visited;

void dfs(int node) {
    for (int next : graph[node]) {
        if (!visited[next]) {
            visited[next] = true;
            dfs(next);
        }
    }
    top_sort.push_back(node);
}

int main() {
    int n, m; // The number of nodes and edges respectively
    std::cin >> n >> m;

    graph = vector<vector<int>>(n);
    for (int i = 0; i < m; i++) {
        int a, b;
        std::cin >> a >> b;
        graph[a - 1].push_back(b - 1);
    }

    visited = vector<bool>(n);
```

```
for (int i = 0; i < n; i++) {
    if (!visited[i]) {
        visited[i] = true;
        dfs(i);
    }
}
std::reverse(top_sort.begin(), top_sort.end());

vector<int> ind(n);
for (int i = 0; i < n; i++) { ind[top_sort[i]] = i; }

// Check if the topological sort is valid
bool valid = true;
for (int i = 0; i < n; i++) {
    for (int j : graph[i]) {
        if (ind[j] <= ind[i]) {
            valid = false;
            goto answer;
        }
    }
}
answer;;

if (valid) {
    for (int i = 0; i < n - 1; i++) { cout << top_sort[i] + 1 << ' '; }
    cout << top_sort.back() + 1 << endl;
} else {
    cout << "IMPOSSIBLE" << endl;
}
}
```

8 Techniques

8.1 Coordinate Compression

```
void coordinate_compress(vector<int> &x, int start=0, int
    step=1) {
    set unique(x.begin(), x.end());
    map<int, int> valPos;

    int idx=0;
```

```
for (auto i: unique) {
    valPos[i] = start + idx * step;
    ++idx;
}
for(auto &i: x) i = valPos[i];
}
```

Coordinate Compression

8.2 Binary to decimal

Binary to decimal

```
// Function to convert binary to decimal
// O(32)
int binaryToDecimal(string str)
{
    int dec_num = 0;
    int power = 0 ;
    int n = str.length() ;

    for(int i = n-1 ; i>=0 ; i--){
        if(str[i] == '1'){
            dec_num += (1<<power) ;
        }
        power++ ;
    }

    return dec_num;
}
```

8.3 Decimal to binary

Decimal to binary

```
// Function that convert Decimal to binary
// O(32)
void decToBinary(int n)
{

```

```
// Size of an integer is assumed to be 32 bits
for (int i = 31; i >= 0; i--) {
    int k = n >> i;
    if (k & 1)
        cout << "1";
    else
        cout << "0";
    }
}

// O(logn)
string DecimalToBinary(int num)
{
    string str;
    while(num){
        if(num & 1) // 1
            str+='1';
        else // 0
            str+='0';
        num>>=1; // Right Shift by 1
    }
    return str;
}
```

9 Number Theory

9.1 Divisors

9.1.1 formulas

number of divisors

```
int d(int n){
    unordered_map<int, int> factors = pf(n);
    int c = 1;
    for(const auto& factor: factors){
        c *= (factor.second+1);
    }
    return c;
}
```

```
// range Count Divisors backward thinking MAXN = 2e6
for(int i=1; i <= n; ++i) {
    for(int j = i; j <= n; j += i) {
        numFactors[j]++;
    }
}

int countDivisors(int n) {
    int count = 0;
    for (int i = 1; i * i <= n; ++i) {
        if (n % i == 0) {
            if (i == n / i) {
                count++; // Perfect square
            } else {
                count += 2; // Pair of divisors
            }
        }
    }
    return count;
}
```

sum of divisors

```
int s(int n){
    unordered_map<int,int> factors = pf(n);
    int sum = 1;
    for(const auto& factor: factors){
        int p = factor.first;
        int exp = factor.second;
        sum *= (pow(p,exp+1)-1)/p-1;
    }
    return sum;
}
```

9.2 Primes

prime factorization

```
void factorize(int x, unordered_map<int, int>& factors) {
    while (x % 2 == 0) {
```

```
        factors[2]++;
        x /= 2;
    }
    for (int i = 3; i * i <= x; i += 2) {
        while (x % i == 0) {
            factors[i]++;
            x /= i;
        }
    }
    if (x > 2) factors[x]++;
}
```

number of co-primes with n

```
int eulerTotient(int n){
    int result = n;

    for(int i = 2; SQ(i) <= n; i++){
        if(n%i == 0){
            while(n%i == 0){
                n/=i;
            }
            result -= result/i;
        }
    }

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

//Phi(n) = n * (1 - 1/P1) * (1 - 1/P2) * ...

//NOTE: summation of Euler function over divisors of n is equal to n
```

Prime Check

```
vector<bool> isPrime(MAXN, true);

void sieve() {
    isPrime[0] = isPrime[1] = false;
```

```
for (int i=2; i * i <= isPrime.size(); ++i) {
    if(isPrime[i]) {
        for (int j = 2 * i; i <= isPrime.size(); j += i)
            prime[j] = false;
    }
}

bool Prime(int n) {
    if(n == 2) return true;
    if(n < 2 || n % 2 == 0) return false;

    for(int i=3; i * i <= n; i += 2) {
        if(n % i == 0) return false;
    }
    return true;
}

// Generate Primes
const int sz = sqrt(MAXN);
vector<int> prime;
vector<bool> vis(sz);

void pre() {
    prime.push_back(2);
    for (int j = 4; j < sz; j += 2) vis[j] = true;
    for (int i = 3; i < sz; i += 2) {
        if (vis[i]) continue;
        prime.push_back(i);
        for (int j = i * i; j < sz; j += i) vis[j] = true;
    }
}

// Preprocessing Prime Factorization of range numbers
constexpr int N = 5e6+1;
int a[N];

for(int i=2; i < N; ++i) {
    if(!a[i]) {
        for(int j=1; i*j < N; ++j) {
            for(int k=i*j; k%i==0; k/=i) a[i*j]++;
        }
    }
}
```

```
    a[i] += a[i-1];
}
```

10 Geometry

10.1 Linearity

10.1.1 co-linear points

check if two points are co-linear

```
bool co_linear(int x1, int y1, int x2, int y2, int x3, int y3){
    int area = x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2);
    return area == 0;
}
```

10.2 polygons

10.2.1 Polygon formation

check if can form polygon with given angle

```
bool possible(double angle){
    if(angle <= 0 || angle >= 180) return false;

    double sides = 360.0/(180.0-angle);

    return (sides == static_cast<int>(sides) && sides >= 3);
}
```

10.2.2 Rectangle Intersection

intersection area between 2 rectangles

```
struct Rectangle {
    int x1, y1; // Bottom-left corner
    int x2, y2; // Top-right corner
};
```

```
int intersectionArea(const Rectangle& rect1, const Rectangle& rect2){

    int x_left = max(rect1.x1, rect2.x1);
    int y_bottom = max(rect1.y1, rect2.y1);
    int x_right = min(rect1.x2, rect2.x2);
    int y_top = min(rect1.y2, rect2.y2);

    int intersection_width = x_right - x_left;
    int intersection_height = y_top - y_bottom;

    if (intersection_width > 0 && intersection_height > 0) {
        return intersection_width * intersection_height;
    }

    return 0;
}
```

11 Miscellaneous

11.1 Faster implementations

11.1.1 hashes

custom hash

```
#define safe hash unordered_map<type, type, custom_hash> // same for
gp_hash_table
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().
            time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
}
```

```
    }
};
```

gb hash table

```
//policy based ds (faster hash table)
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table;
```

11.1.2 Binary Search the value

nearest sqrt

```
long long my_sqrt(long long a)
{
    long long l=0,r=5000000001;
    while(r-l>1)
    {
        long long mid=(l+r)/2;
        if(1ll*mid*mid<=a)l=mid;
        else r=mid;
    }
    return l;
}
```
