

# Competitive Programming Library

Too bad to be Accepted

## Contents

<b>1</b>	<b>Dynamic Programming</b>	<b>2</b>	<b>6</b>	<b>Graph Theory</b>	<b>12</b>
1.1	Some dp patterns . . . . .	2	6.1	Shortest Path algorithms . . . . .	12
			6.1.1	Dijkstra Algorithm . . . . .	12
<b>2</b>	<b>Bit Manipulation</b>	<b>4</b>	6.1.2	Floyd Warshal Algorithm . . . . .	13
2.1	Subset Operations . . . . .	4	6.1.3	Bellman Ford Algorithm . . . . .	13
<b>3</b>	<b>Algorithms</b>	<b>5</b>	6.2	Cycle Detection . . . . .	14
3.1	MO . . . . .	5	6.2.1	DFS Implementation . . . . .	14
<b>4</b>	<b>Data Structures</b>	<b>6</b>	6.2.2	Another way for undirected graphs . . . . .	14
4.1	Strings . . . . .	6	6.2.3	General Way . . . . .	14
4.1.1	Trie (Prefix Tree) . . . . .	6	6.2.4	DSU Implementation . . . . .	14
4.2	Range Queries . . . . .	7	6.3	Algorithms . . . . .	15
4.2.1	Segment Tree . . . . .	7	6.3.1	Heavy Light Decomposition . . . . .	15
4.2.2	Lazy Propegation . . . . .	8	<b>7</b>	<b>Techniques</b>	<b>17</b>
4.2.3	Fenwick Tree . . . . .	9	7.1	Coordinate Compression . . . . .	17
4.2.4	Fenwick UpdateRange . . . . .	9	<b>8</b>	<b>Templates</b>	<b>18</b>
4.2.5	2D BIT . . . . .	10	8.1	MOD Template . . . . .	18
4.2.6	Sparse Table . . . . .	10	8.2	Macros . . . . .	18
4.3	Ordered Set . . . . .	11	<b>9</b>	<b>Miscellaneous</b>	<b>19</b>
<b>5</b>	<b>Counting Principles</b>	<b>12</b>	<b>10</b>	<b>Faster implementations</b>	<b>19</b>
5.1	nCr . . . . .	12	10.1	hashes . . . . .	19
5.1.1	Fast nCr . . . . .	12	10.2	Binary Search the value . . . . .	19
			<b>11</b>	<b>Number Theory</b>	<b>20</b>

## 12 Divisors

12.1 factorization . . . . .	20
12.2 divisors . . . . .	20
12.3 co-primes . . . . .	20

## 13 Geometry

## 14 Linearity

14.1 co-linear points . . . . .	21
14.2 polygons . . . . .	21

# 1 Dynamic Programming

## 1.1 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 Bit Manipulation

### 2.1 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);
}
```

---

## 3 Algorithms

### 3.1 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 Data Structures

### 4.1 Strings

#### 4.1.1 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;
    }
};

```

---

## 4.2 Range Queries

### 4.2.1 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(1, r, 2*node+1, lx, mid, time, val);
    assign_range(1, 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(1, r, 0, 0, size);
    }
};

```

## 4.2.2 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(1, r, 2*node+1, lx, mid, val, f);
        update_range(1, 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(1, r, lx, mid, 2*node+1) + range_query(1, 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(1, r, 0, 0, size-1, v, f);
    }

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

```

### 4.2.3 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);
}
};

```

### 4.2.4 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);
    }
};

```

#### 4.2.5 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;
    }
};

```

#### 4.2.6 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;
    });
}
```

---

### 4.3 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*

---

## 5 Counting Principles

### 5.1 nCr

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

#### 5.1.1 Fast nCr

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

#### *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 Graph Theory

### 6.1 Shortest Path algorithms

#### 6.1.1 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;
}

```

---

## 6.1.2 Floyd Warshal Algorithm

### *FloydWarshal 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;
}

```

---

## 6.1.3 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;
}

```

## 6.2 Cycle Detection

### 6.2.1 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;
}
```

### 6.2.2 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;
}
```

### 6.2.3 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;
}
```

### 6.2.4 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;
}
```

---

## 6.3 Algorithms

### 6.3.1 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 << el;
        }
    }
}

```



---

## 7 Techniques

### 7.1 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 Templates

### 8.1 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*

### 8.2 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 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) (((1ll)(a)+(1ll)(b-1))/(1ll)(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);

int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

int dx[8] = {1, 1, -1, -1, 2, 2, -2, -2};
int dy[8] = {2, -2, 2, -2, 1, -1, 1, -1};
```

## 9 Miscellaneous

## 10 Faster implementations

### 10.1 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;
```

---

### 10.2 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;
```

---

## 11 Number Theory

## 12 Divisors

### 12.1 factorization

*prime factorization*

```
unordered_map<int,int> pf(int n){
    unordered_map<int, int> factors;
    while(n%2 == 0){
        factors[2]++;
        n/=2;
    }

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

---

### 12.2 divisors

*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;
}
```

---

*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;
}
```

---

### 12.3 co-primes

*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
```

---

## 13 Geometry

## 14 Linearity

### 14.1 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;
}
```

---

### 14.2 polygons

*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);
}
```

---

*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;
}
```

---