树状数组

```
int origin[N]; //原本数组，下标从1开始
int bit[N]; //树状数组，下标从1开始

int lowbit(int x){
    return x & (-x);
}

void update(int i, int x) {
    //注意这里是大于等于号
    while( i <= n ){
        bit[i] += x;
        i += lowbit(i);
    }
}

int getSum(int i){
    int sum = 0;
    //注意这里是大于号
    while( i > 0 ) {
        sum += bit[i];
        i -= lowbit(i);
    }
    return sum;
}
```

线段树

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

using namespace std;
const long long N = 1e6 + 7;

typedef long long ll;
struct node {
    long long l, r, sum, add, mul;
} tree[N];

//快读
template<typename T>
void read(T &x) {
    x = 0;
    char ch = (char) getchar();
    ll f = 1;
    if (!isdigit(ch)) {
        if (ch == '-') f = -1;
        ch = (char) getchar();
    }
    while (isdigit(ch)) {
        x = (x << 1) + (x << 3) + (ch ^ 48);
        ch = (char) getchar();
    }
```

```
    x *= f;
}

long long init[N]; //存放初始值，下标从1开始
long long n, m, mod;

//建树
void build(long long i, long long l, long long r) {
    tree[i].add = 0;
    tree[i].mul = 1;
    tree[i].l = l;
    tree[i].r = r;
    if (l == r) {
        tree[i].sum = init[l] % mod;
        return;
    }
    long long mid = (l + r) / 2;
    build(2 * i, l, mid);
    build(2 * i + 1, mid + 1, r);
    tree[i].sum = (tree[2 * i].sum + tree[2 * i + 1].sum) % mod;
}

inline void push_down(long long i) {
    tree[2 * i].sum = (tree[i].mul * tree[i * 2].sum + ((tree[i * 2].r - tree[2 * i].l + 1) * tree[i].add) % mod) % mod;
    tree[2 * i + 1].sum = (tree[i].mul * tree[i * 2 + 1].sum + ((tree[i * 2 + 1].r - tree[2 * i + 1].l + 1) * tree[i].add) % mod) % mod;

    tree[i * 2].mul = (tree[2 * i].mul * tree[i].mul) % mod;
    tree[i * 2 + 1].mul = (tree[2 * i + 1].mul * tree[i].mul) % mod;

    tree[i * 2].add = (tree[2 * i].add * tree[i].mul + tree[i].add) % mod;
    tree[i * 2 + 1].add = (tree[2 * i + 1].add * tree[i].mul + tree[i].add) % mod;

    tree[i].mul = 1;
    tree[i].add = 0;
}

//区间[l,r]所有元素各加上k
inline void add(long long i, long long l, long long r, long long k) {
    if (tree[i].l >= l && tree[i].r <= r) {
        tree[i].sum = (tree[i].sum + k * (tree[i].r - tree[i].l + 1)) % mod;
        tree[i].add = (tree[i].add + k) % mod;
        return;
    }
    push_down(i);
    if (tree[i * 2].r >= l)
        add(i * 2, l, r, k);
    if (tree[i * 2 + 1].l <= r)
        add(i * 2 + 1, l, r, k);
    tree[i].sum = (tree[i * 2].sum + tree[i * 2 + 1].sum) % mod;
}


//区间 乘
inline void mult(long long i, long long l, long long r, long long k) {
```

```
    if (tree[i].l >= l && tree[i].r <= r) {
        tree[i].mul = (tree[i].mul * k) % mod;
        tree[i].add = (tree[i].add * k) % mod;
        tree[i].sum = (tree[i].sum * k) % mod;
        return;
    }
    push_down(i);
    if (tree[i * 2].r >= l)
        mult(i * 2, l, r, k);
    if (tree[i * 2 + 1].l <= r)
        mult(i * 2 + 1, l, r, k);
    tree[i].sum = (tree[i * 2].sum + tree[i * 2 + 1].sum) % mod;
}


//查询
inline long long search(long long i, long long l, long long r) {
    if (tree[i].l >= l && tree[i].r <= r) {
        return tree[i].sum;
    }
    push_down(i);
    long long s = 0;
    if (tree[i * 2].r >= l)
        s = (s + search(i * 2, l, r)) % mod;
    if (tree[i * 2 + 1].l <= r)
        s = (s + search(i * 2 + 1, l, r)) % mod;
    return s;
}


int main() {
    read(n), read(m), read(mod);
    long long flag, cn, cm, cw;
    for (int i = 1; i <= n; ++i)
        read(init[i]);
    build(1, 1, n);
    for (int i = 1; i <= m; ++i) {
        read(flag);
        if (flag == 1) {
            read(cn), read(cm), read(cw);
            mult(1, cn, cm, cw);
        } else if (flag == 2) {
            read(cn), read(cm), read(cw);
            add(1, cn, cm, cw);
        } else {
            read(cn), read(cm);
            cout << search(1, cn, cm) << endl;
        }
    }
}

```

滑动窗口框架

```
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
```

```
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /*********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

回溯算法框架

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

单调队列模板

```
class MonotonicQueue {
    // 双链表，支持头部和尾部增删元素
    private LinkedList<Integer> q = new LinkedList<>();

    public void push(int n) {
        // 将前面小于自己的元素都删除
        while (!q.isEmpty() && q.getLast() < n) {
            q.pollLast();
        }
        q.addLast(n);
    }

```

```
    public int max() {
        // 队头的元素肯定是最大的
        return q.getFirst();
    }

    public void pop(int n) {
        if (n == q.getFirst()) {
            q.pollFirst();
        }
    }
}

/* 单调队列的实现 */
class MonotonicQueue {
    LinkedList<Integer> q = new LinkedList<>();
    public void push(int n) {
        // 将小于 n 的元素全部删除
        while (!q.isEmpty() && q.getLast() < n) {
            q.pollLast();
        }
        // 然后将 n 加入尾部
        q.addLast(n);
    }

    public int max() {
        return q.getFirst();
    }

    public void pop(int n) {
        if (n == q.getFirst()) {
            q.pollFirst();
        }
    }
}

/* 解题函数的实现 */
int[] maxSlidingWindow(int[] nums, int k) {
    MonotonicQueue window = new MonotonicQueue();
    List<Integer> res = new ArrayList<>();

    for (int i = 0; i < nums.length; i++) {
        if (i < k - 1) {
            //先填满窗口的前 k - 1
            window.push(nums[i]);
        } else {
            // 窗口向前滑动，加入新数字
            window.push(nums[i]);
            // 记录当前窗口的最大值
            res.add(window.max());
            // 移出旧数字
            window.pop(nums[i - k + 1]);
        }
    }
    // 需要转成 int[] 数组再返回
    int[] arr = new int[res.size()];
    for (int i = 0; i < res.size(); i++) {
        arr[i] = res.get(i);
    }
```

```
    return arr;
}
```

单调栈模板

```
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> res(nums.size()); // 存放答案的数组
    stack<int> s;
    // 倒着往栈里放
    for (int i = nums.size() - 1; i >= 0; i--) {
        // 判定个子高矮
        while (!s.empty() && s.top() <= nums[i]) {
            // 矮个起开，反正也被挡着了。。。
            s.pop();
        }
        // nums[i] 身后的 next great number
        res[i] = s.empty() ? -1 : s.top();
        //
        s.push(nums[i]);
    }
    return res;
}
```

BFS框架

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

最短路径 dijkstra算法

```cpp
void dijkstra(vector<vector<int> >&graph, int start, int prev[], int dist[])
{
    int i, j, k;
    int min;
    int tmp;
    int flag[20] ;      // flag[i]=1表示"顶点start"到"顶点i"的最短路径已成功获取。

    // 初始化
    for (i = 0; i < 20; i++) {
        flag[i] = 0;              // 顶点i的最短路径还没获取到。
        prev[i] = 0;              // 顶点i的前驱顶点为0
        dist[i] = graph[start][i];    // 顶点i的最短路径为"顶点start"到"顶点i"的权。
    }

    // 对"顶点start"自身进行初始化
    flag[start] = 1;
    dist[start] = 0;

    // 每次找出一个顶点的最短路径。
    for (i = 1; i < 20; i++) {
        // 寻找当前最小的路径；
        // 即，在未获取最短路径的顶点中，找到离start最近的顶点k。
        min = INT_MAX;
        for (j = 0; j < 20; j++) {
            if (flag[j] == 0 && dist[j] < min) {
                min = dist[j];
                k = j;
            }
        }
        // 标记"顶点k"为已经获取到最短路径
        flag[k] = 1;

        // 修正当前最短路径和前驱顶点
        // 即，当已经"顶点k的最短路径"之后，更新"未获取最短路径的顶点的最短路径和前驱顶点"。
        for (j = 0; j < 20; j++) {
            tmp = (graph[k][j] == INT_MAX ? INT_MAX : (min + graph[k][j]));
//防止溢出
            if (flag[j] == 0 && (tmp < dist[j])) {
                dist[j] = tmp;
                prev[j] = k;
            }
        }
    }
}
//打印路径
void printPath( int pre[], int tar, int start){
    if( tar == start ){
        cout << city[tar]<<endl;
        return ;
    }
    printPath(pre, pre[tar], start);
    cout << city[tar] << endl;
}
```

并查集

```
int X = 10010;
int f[X];

void init(){
    for(int i = 0;i <= X - 1;i++){
        f[i] = i;
    }
}

int find_f(int x){
    if(x != f[x]){
        return f[x] = find_f(f[x]);//在递归的时候，就直接将遇到的当前帮派的英雄的掌门修改了
    }
    return f[x];//如果找到了掌门，就直接返回掌门编号
}

void join(int x,int y){
    int fx = find_f(x),
        fy = find_f(y);//找到两位英雄的掌门
    if(fx != fy){
        f[fy] = fx;//合并子集
    }
}

#include <bits/stdc++.h>

using namespace std;
int f[10001];

int find(int x) {
    if (x != f[x]) {
        return f[x] = find(f[x]);
    }
    return f[x];
}

int main() {
    int N, M;
    cin >> N >> M;

    for (int i = 1; i <= N; i++) {
        f[i] = i;
    }

    for (int i = 1; i <= M; i++) {
        int z, x, y;
        cin >> z >> x >> y;
        if (z == 1) {
            int a = find(x);
            int b = find(y);
            if (a != b) {
                f[a] = b;
            }
        } else {
```

```
            int a = find(x);
            int b = find(y);
            if (a == b) {
                cout << "Y" << endl;
            } else {
                cout << "N" << endl;
            }
        }
    }
    return 0;
}

```

其他

```
//友元重载<号
friend bool operator<(book b1, book b2) {
    return b1.high < b2.high;
}

//cmp函数
bool cmp(int a, int b){
    return a < b;
}

//优先队列
priority_queue<int, vector<int>, greater<int> > pq;//小顶堆

```