

树状数组

```
1  int origin[N]; //原本数组，下标从1开始
2  int bit[N]; //树状数组，下标从1开始
3
4  int lowbit(int x){
5      return x & (-x);
6  }
7
8  void update(int i, int x) {
9      //注意这里是大于等于号
10     while( i <= n ){
11         bit[i] += x;
12         i += lowbit(i);
13     }
14 }
15
16 int getSum(int i){
17     int sum = 0;
18     //注意这里是大于号
19     while( i > 0 ) {
20         sum += bit[i];
21         i -= lowbit(i);
22     }
23     return sum;
24 }
```

线段树

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4  const long long N = 1e6 + 7;
5
6  typedef long long ll;
7  struct node {
8      long long l, r, sum, add, mul;
9  } tree[N];
10
11 //快读
12 template<typename T>
13 void read(T &x) {
14     x = 0;
15     char ch = (char) getchar();
16     ll f = 1;
17     if (!isdigit(ch)) {
18         if (ch == '-') f = -1;
19         ch = (char) getchar();
20     }
21     while (isdigit(ch)) {
22         x = (x << 1) + (x << 3) + (ch ^ 48);
23         ch = (char) getchar();
24     }
```

```

25     x *= f;
26 }
27
28 long long init[N]; //存放初始值, 下标从1开始
29 long long n, m, mod;
30
31 //建树
32 void build(long long i, long long l, long long r) {
33     tree[i].add = 0;
34     tree[i].mul = 1;
35     tree[i].l = l;
36     tree[i].r = r;
37     if (l == r) {
38         tree[i].sum = init[l] % mod;
39         return;
40     }
41     long long mid = (l + r) / 2;
42     build(2 * i, l, mid);
43     build(2 * i + 1, mid + 1, r);
44     tree[i].sum = (tree[2 * i].sum + tree[2 * i + 1].sum) % mod;
45 }
46
47 inline void push_down(long long i) {
48     tree[2 * i].sum = (tree[i].mul * tree[i * 2].sum + ((tree[i * 2].r -
49 tree[2 * i].l + 1) * tree[i].add) % mod) % mod;
50     tree[2 * i + 1].sum = (tree[i].mul * tree[i * 2 + 1].sum + ((tree[i * 2
51 + 1].r - tree[2 * i + 1].l + 1) * tree[i].add) % mod) % mod;
52
53     tree[i * 2].mul = (tree[2 * i].mul * tree[i].mul) % mod;
54     tree[i * 2 + 1].mul = (tree[2 * i + 1].mul * tree[i].mul) % mod;
55
56     tree[i * 2].add = (tree[2 * i].add * tree[i].mul + tree[i].add) % mod;
57     tree[i * 2 + 1].add = (tree[2 * i + 1].add * tree[i].mul + tree[i].add)
58 % mod;
59
60     tree[i].mul = 1;
61     tree[i].add = 0;
62 }
63
64 //区间[l,r]所有元素各加上k
65 inline void add(long long i, long long l, long long r, long long k) {
66     if (tree[i].l >= l && tree[i].r <= r) {
67         tree[i].sum = (tree[i].sum + k * (tree[i].r - tree[i].l + 1)) %
68 mod;
69         tree[i].add = (tree[i].add + k) % mod;
70         return;
71     }
72     push_down(i);
73     if (tree[i * 2].r >= l)
74         add(i * 2, l, r, k);
75     if (tree[i * 2 + 1].l <= r)
76         add(i * 2 + 1, l, r, k);
77     tree[i].sum = (tree[i * 2].sum + tree[i * 2 + 1].sum) % mod;
78 }
79
80 //区间 乘
81 inline void mult(long long i, long long l, long long r, long long k) {

```

```

79     if (tree[i].l >= 1 && tree[i].r <= r) {
80         tree[i].mul = (tree[i].mul * k) % mod;
81         tree[i].add = (tree[i].add * k) % mod;
82         tree[i].sum = (tree[i].sum * k) % mod;
83         return;
84     }
85     push_down(i);
86     if (tree[i * 2].r >= 1)
87         mult(i * 2, 1, r, k);
88     if (tree[i * 2 + 1].l <= r)
89         mult(i * 2 + 1, 1, r, k);
90     tree[i].sum = (tree[i * 2].sum + tree[i * 2 + 1].sum) % mod;
91 }
92
93
94 //查询
95 inline long long search(long long i, long long l, long long r) {
96     if (tree[i].l >= 1 && tree[i].r <= r) {
97         return tree[i].sum;
98     }
99     push_down(i);
100     long long s = 0;
101     if (tree[i * 2].r >= 1)
102         s = (s + search(i * 2, 1, r)) % mod;
103     if (tree[i * 2 + 1].l <= r)
104         s = (s + search(i * 2 + 1, 1, r)) % mod;
105     return s;
106 }
107
108
109 int main() {
110     read(n), read(m), read(mod);
111     long long flag, cn, cm, cw;
112     for (int i = 1; i <= n; ++i)
113         read(init[i]);
114     build(1, 1, n);
115     for (int i = 1; i <= m; ++i) {
116         read(flag);
117         if (flag == 1) {
118             read(cn), read(cm), read(cw);
119             mult(1, cn, cm, cw);
120         } else if (flag == 2) {
121             read(cn), read(cm), read(cw);
122             add(1, cn, cm, cw);
123         } else {
124             read(cn), read(cm);
125             cout << search(1, cn, cm) << endl;
126         }
127     }
128 }
129

```

滑动窗口框架

```

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

```

```

4     for (char c : t) need[c]++;
5
6     int left = 0, right = 0;
7     int valid = 0;
8     while (right < s.size()) {
9         // c 是将移入窗口的字符
10        char c = s[right];
11        // 右移窗口
12        right++;
13        // 进行窗口内数据的一系列更新
14        ...
15
16        /** debug 输出的位置 **/
17        printf("window: [%d, %d]\n", left, right);
18        /***/
19
20        // 判断左侧窗口是否要收缩
21        while (window needs shrink) {
22            // d 是将移出窗口的字符
23            char d = s[left];
24            // 左移窗口
25            left++;
26            // 进行窗口内数据的一系列更新
27            ...
28        }
29    }
30 }

```

回溯算法框架

```

1  result = []
2  def backtrack(路径, 选择列表):
3      if 满足结束条件:
4          result.add(路径)
5          return
6
7      for 选择 in 选择列表:
8          做选择
9          backtrack(路径, 选择列表)
10         撤销选择

```

单调队列模板

```

1  class MonotonicQueue {
2      // 双链表, 支持头部和尾部增删元素
3      private LinkedList<Integer> q = new LinkedList<>();
4
5      public void push(int n) {
6          // 将前面小于自己的元素都删除
7          while (!q.isEmpty() && q.getLast() < n) {
8              q.pollLast();
9          }
10         q.addLast(n);
11     }
12 }

```

```

13     public int max() {
14         // 队头的元素肯定是最大的
15         return q.getFirst();
16     }
17
18     public void pop(int n) {
19         if (n == q.getFirst()) {
20             q.pollFirst();
21         }
22     }
23 }
24
25 /* 单调队列的实现 */
26 class MonotonicQueue {
27     LinkedList<Integer> q = new LinkedList<>();
28     public void push(int n) {
29         // 将小于 n 的元素全部删除
30         while (!q.isEmpty() && q.getLast() < n) {
31             q.pollLast();
32         }
33         // 然后将 n 加入尾部
34         q.addLast(n);
35     }
36
37     public int max() {
38         return q.getFirst();
39     }
40
41     public void pop(int n) {
42         if (n == q.getFirst()) {
43             q.pollFirst();
44         }
45     }
46 }
47
48 /* 解题函数的实现 */
49 int[] maxSlidingWindow(int[] nums, int k) {
50     MonotonicQueue window = new MonotonicQueue();
51     List<Integer> res = new ArrayList<>();
52
53     for (int i = 0; i < nums.length; i++) {
54         if (i < k - 1) {
55             // 先填满窗口的前 k - 1
56             window.push(nums[i]);
57         } else {
58             // 窗口向前滑动，加入新数字
59             window.push(nums[i]);
60             // 记录当前窗口的最大值
61             res.add(window.max());
62             // 移出旧数字
63             window.pop(nums[i - k + 1]);
64         }
65     }
66     // 需要转成 int[] 数组再返回
67     int[] arr = new int[res.size()];
68     for (int i = 0; i < res.size(); i++) {
69         arr[i] = res.get(i);
70     }

```

```
71     return arr;
72 }
```

单调栈模板

```
1  vector<int> nextGreaterElement(vector<int>& nums) {
2      vector<int> res(nums.size()); // 存放答案的数组
3      stack<int> s;
4      // 倒着往栈里放
5      for (int i = nums.size() - 1; i >= 0; i--) {
6          // 判定个子高矮
7          while (!s.empty() && s.top() <= nums[i]) {
8              // 矮个起开，反正也被挡着了。。。
9              s.pop();
10         }
11         // nums[i] 身后的 next great number
12         res[i] = s.empty() ? -1 : s.top();
13         //
14         s.push(nums[i]);
15     }
16     return res;
17 }
```

BFS框架

```
1  // 计算从起点 start 到终点 target 的最短距离
2  int BFS(Node start, Node target) {
3      Queue<Node> q; // 核心数据结构
4      Set<Node> visited; // 避免走回头路
5
6      q.offer(start); // 将起点加入队列
7      visited.add(start);
8      int step = 0; // 记录扩散的步数
9
10     while (q not empty) {
11         int sz = q.size();
12         /* 将当前队列中的所有节点向四周扩散 */
13         for (int i = 0; i < sz; i++) {
14             Node cur = q.poll();
15             /* 划重点：这里判断是否到达终点 */
16             if (cur is target)
17                 return step;
18             /* 将 cur 的相邻节点加入队列 */
19             for (Node x : cur.adj())
20                 if (x not in visited) {
21                     q.offer(x);
22                     visited.add(x);
23                 }
24         }
25         /* 划重点：更新步数在这里 */
26         step++;
27     }
28 }
```

最短路径 dijkstra算法

```

1 void dijkstra(vector<vector<int> >&graph, int start, int prev[], int dist[])
2 {
3     int i, j, k;
4     int min;
5     int tmp;
6     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。
7
8     // 初始化
9     for (i = 0; i < 20; i++) {
10         flag[i] = 0;          // 顶点i的最短路径还没获取到。
11         prev[i] = 0;          // 顶点i的前驱顶点为0
12         dist[i] = graph[start][i];    // 顶点i的最短路径为"顶点start"到"顶点i"的
13                                     权。
14     }
15
16     // 对"顶点start"自身进行初始化
17     flag[start] = 1;
18     dist[start] = 0;
19
20     // 每次找出一个顶点的最短路径。
21     for (i = 1; i < 20; i++) {
22         // 寻找当前最小的路径:
23         // 即, 在未获取最短路径的顶点中, 找到离start最近的顶点k。
24         min = INT_MAX;
25         for (j = 0; j < 20; j++) {
26             if (flag[j] == 0 && dist[j] < min) {
27                 min = dist[j];
28                 k = j;
29             }
30         }
31         // 标记"顶点k"为已经获取到最短路径
32         flag[k] = 1;
33
34         // 修正当前最短路径和前驱顶点
35         // 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
36         点"。
37         for (j = 0; j < 20; j++) {
38             tmp = (graph[k][j] == INT_MAX ? INT_MAX : (min + graph[k][j]));
39             //防止溢出
40             if (flag[j] == 0 && (tmp < dist[j])) {
41                 dist[j] = tmp;
42                 prev[j] = k;
43             }
44         }
45     }
46
47     //打印路径
48     void printPath( int pre[], int tar, int start){
49         if( tar == start ){
50             cout << city[tar]<<endl;
51             return ;
52         }
53         printPath(pre, pre[tar], start);
54         cout << city[tar] << endl;
55     }
56 }

```

并查集

```
1  int x = 10010;
2  int f[x];
3
4  void init(){
5      for(int i = 0;i <= x - 1;i++){
6          f[i] = i;
7      }
8  }
9
10 int find_f(int x){
11     if(x != f[x]){
12         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
13     }
14     return f[x]; //如果找到了掌门，就直接返回掌门编号
15 }
16
17 void join(int x,int y){
18     int fx = find_f(x),
19         fy = find_f(y); //找到两位英雄的掌门
20     if(fx != fy){
21         f[fy] = fx; //合并子集
22     }
23 }
24
25 #include <bits/stdc++.h>
26
27 using namespace std;
28 int f[10001];
29
30 int find(int x) {
31     if (x != f[x]) {
32         return f[x] = find(f[x]);
33     }
34     return f[x];
35 }
36
37 int main() {
38     int N, M;
39     cin >> N >> M;
40
41     for (int i = 1; i <= N; i++) {
42         f[i] = i;
43     }
44
45     for (int i = 1; i <= M; i++) {
46         int z, x, y;
47         cin >> z >> x >> y;
48         if (z == 1) {
49             int a = find(x);
50             int b = find(y);
51             if (a != b) {
52                 f[a] = b;
53             }
54         } else {
```



```

55         int a = find(x);
56         int b = find(y);
57         if (a == b) {
58             cout << "Y" << endl;
59         } else {
60             cout << "N" << endl;
61         }
62     }
63 }
64 return 0;
65 }
66

```

其他

```

1 //友元重载<号
2 friend bool operator<(book b1, book b2) {
3     return b1.high < b2.high;
4 }
5
6 //cmp函数
7 bool cmp(int a, int b){
8     return a < b;
9 }
10
11 //优先队列
12 priority_queue<int, vector<int>, greater<int> > pq; //小顶堆
13

```