HƯỚNG DẪN GIẢI — Ban Chuyên môn Tin học

Bài 1: Đọc sách

Subtask 1: Vét cạn

Ở mỗi truy vấn, tính tổng các quyển sách đứng trước quyển sách thứ p_i, sau đó lần lượt di chuyển lần lượt các quyển sách $p_i - 1$ qua vị trí p_i, quyển thứ $p_i - 2$ qua $p_i - 1$,... quyển thứ 1 qua vị trí 2. Cuối cùng, đặt quyển thứ p_i cũ vào vị trí đầu tiên.

Độ phức tạp của mỗi truy vấn là $\mathcal{O}(n)$, độ phức tạp cuối cùng là $\mathcal{O}(qn)$ với q là số lượng số truy vấn.

Subtask 2: Fenwick Tree

Nhận xét: Lucian sẽ đọc sách liên tục trong q ngày, tức là chỉ có q lần rút sách thứ p_i ra rồi đặt lên trên cùng.

Do đó, ta có thể tạo một mảng B có $q+n$ phần tử với q phần tử đầu tiên tượng trưng các cuốn sách được lấy ra rồi đặt lên trên. Ban đầu, q phần tử đầu tiên đều mang giá trị 0. Còn n phần tử còn lại tượng trưng cho số sách được để ở lúc đầu. Quyển sách thứ i có vị trí là $q+i$ trong mảng B.

Vào ngày đọc sách thứ j, đáp án cần tìm là tổng giá trị các phần tử đứng trước nó. Để cập nhật, đặt giá trị phần tử ở vị trí cũ này bằng 0, và vị trí mới của quyển sách này là $q - j + 1$.

Các thao tác cập nhật và truy vấn này có thể được quản lý bằng **Fenwick Tree** kinh điển, giảm độ phức tạp của mỗi truy vấn xuống còn $\mathcal{O}(\log(q+n))$.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}((n+q)\log(q+n))$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n+q)$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
#define FOR(i,a,b) for (int i = (a); i <= (int)(b); ++i)
#define FOD(i,a,b) for (int i = (a); i >= (int)(b); --i)

int n,q;
long long BIT[2000005];
int pos[2000005];
int a[1000005];

void update(int id, int v) {
  int idx = id;
  while (idx <= q+n) {
    BIT[idx] += v;
    idx += (idx & (-idx));
  }
}

long long getSum(int p) {
  int idx = p;
  long long ans = 0;
  while (idx > 0) {
    ans += BIT[idx];
```

```
    idx -= (idx & (-idx));
  }
  return ans;
}

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

void solve() {
  std::cin >> n >> q;
  FOR(i,1,n) {
    std::cin >> a[i];
    pos[i] = q + i;
  }
  FOR(i,1,n) update(pos[i], a[i]);
  FOR(t,1,q) {
    int i; std::cin >> i;
    int tmp = a[i], p = pos[i];
    std::cout << query(1, p-1) << "\n";

    update(p, -tmp);
    pos[i] = q-t+1;
    update(q-t+1, tmp);
  }
}

int main() {
  std::ios_base::sync_with_stdio(0);
  std::cin.tie(0);
  std::cout.tie(0);

  solve();
        return 0;
}
```

Bài 2: Phần tử nhỏ nhất

Subtask 1: Vét cạn

Với giới hạn mảng và số lượng truy vấn là 5000, ta có thể thực hiện truy vấn loại 1 trong $\mathcal{O}(1)$ và loại 2 trong $\mathcal{O}(r-l+1)$ hay $\mathcal{O}(n)$. Tổng độ phức tạp là $\mathcal{O}(qn)$.

Subtask 2: Segment Tree

Đối với subtask 2, chúng ta sử dụng cấu trúc dữ liệu Segment Tree kinh điển để xử lý các truy vấn trong độ phức tạp $\mathcal{O}(\log n)$ mỗi truy vấn. Tuy nhiên, ta sẽ không đào sâu vào lời giải này vì trọng tâm của contest là cấu trúc dữ liệu Fenwick Tree.

Subtask 3: Fenwick Tree (BIT)

Cấu trúc dữ liệu Fenwick Tree thông thường chỉ xử lý được các truy vấn có tính bù trừ (phép cộng, bitwise XOR,...). Tuy nhiên, vào năm 2015, Mircea Dima và Rodica Ceterchi đã trình bày cách sử dụng Fenwick Tree để xử lý truy vấn tìm phần tử nhỏ nhất. Bạn đọc có thể tham khảo bài báo cáo khoa học tại đây.

Trong phần này, chúng ta sẽ tìm hiểu phương pháp được trình bày trong bài báo cáo trên.

Xây dựng Fenwick Tree ngược

Ý tưởng của thuật toán là ta xây dựng song song hai Fenwick Tree: một Fenwick Tree truyền thống và một Fenwick Tree ngược. Gọi $p(x)$ là bit bật nhỏ nhất trong biểu diễn nhị phân của x. Nút thứ k trong Fenwick Tree ngược quản lý đoạn $[k; k+p(k)-1]$ (thay vì là $[k-p(k)+1; k]$), tức **đoạn có độ dài $p(k)$ bắt đầu tại**

k. Lúc này, việc di chuyển trên cây Fenwick Tree thứ hai sẽ được thực hiện ngược lại so với cây truyền thống. Nói cách khác, ở bước cập nhật, ta di chuyển bằng phép gán $k \leftarrow k - p(k)$, và ở truy vấn, ta di chuyển bằng phép gán $k \leftarrow k + p(k)$.

Truy vấn đoạn

Với một truy vấn tìm phần tử nhỏ nhất trên đoạn $[l; r]$, ta bắt đầu từ nút thứ r trên cây đầu tiên (Fenwick Tree truyền thống), nhảy trên cây (bằng phép gán $k \leftarrow k - p(k)$) để lấy nhiều nút có đoạn quản lý nằm hoàn toàn trong $[l; r]$ nhiều nhất có thể. Thực hiện thao tác tương tự trên cây thứ hai, bắt đầu từ nút thứ l và nhảy bằng phép gán $k \leftarrow k + p(k)$.

Có thể chứng minh rằng sau khi thực hiện hai thao tác trên, số lượng phần tử chưa được phủ tối đa là 1. Nếu có phần tử chưa được phủ, chỉ số của nó cũng chính là chỉ số trên hai con trỏ của hai cây sau khi nhảy (trong trường hợp này chúng chắc chắn có cùng chỉ số), ta chỉ cần truy cập điểm là xong.

Như vậy, ta có truy vấn đoạn trong $\mathcal{O}(\log n)$.

Cập nhật điểm

Ý tưởng tìm các nút để cập nhật giống Fenwick Tree truyền thống. Tuy nhiên, với một nút quản lý đoạn $[l; r]$ chưa điểm p cần cập nhật, ta cần tính phần tử nhỏ nhất của đoạn $[l; p-1]$ và $[p+1; r]$.

Ta sẽ giải quyết vấn đề này trên từng cây:

- Để tính đoạn $[l; p-1]$ trên cây thứ nhất, ta duy trì một con trỏ tại nút $p-1$ trên chính cây này. Trong lúc cập nhật, biên trái của đoạn cần tính sẽ chạy sang trái (tức l giảm), lúc đó, ta chỉ việc nhảy con trỏ này sang trái để lấy các đoạn cần thiết. Có thể chứng minh rằng tổng số nút cần nhảy là $\mathcal{O}(\log n)$ và ta luôn có thể phủ đoạn cần tính.
- Để tính đoạn $[p+1; r]$ trên cây thứ nhất, ta duy trì một con trỏ tại nút $p+1$ trên cây thứ hai. Tương tự đoạn $[l; p-1]$, ta nhảy con trỏ này sang phải để lấy các đoạn cần thiết trong lúc cập nhật, sau đó lấy thêm giá trị của phần tử tại nút đó nếu chưa được phủ. Tổng số nút cũng là $\mathcal{O}(\log n)$ và ta luôn có thể phủ đoạn cần tính.
- Đối với cây thứ hai, để tính đoạn $[l; p-1]$, ta dùng con trỏ bắt đầu tại nút $p-1$ trên cây thứ nhất. Để tính đoạn $[p+1; r]$, ta dùng con trỏ bắt đầu tại nút $p+1$ trên chính cây thứ hai. Sử dụng ý tưởng tương tự cây thứ nhất.

Tổng độ phức tạp của một truy vấn vẫn là $\mathcal{O}(\log n)$.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}((n+q)\log n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n)$.

Cài đặt (tham khảo)

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

typedef long long ll;
typedef long double ld;
typedef pair<ll,ll> pl;
typedef pair<int,int> pii;
typedef tuple<int,int,int> tpl;

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

struct BIT {
```

```
    vector<int> t1, t2, a;
    int n;

    BIT (int sz) : t1(sz + 1), t2(sz + 1), a(sz + 1), n(sz) {}

    int p (int k) { return k & -k; }

    int query (int l, int r) {
        int ans = INT_MAX;
        for (; l + p(l) - 1 <= r && l <= r; l += p(l)) ans = min(ans, t2[l]);
        for (; r - p(r) + 1 >= l && l <= r; r -= p(r)) ans = min(ans, t1[r]);
        return min(ans, (l == r ? a[l] : INT_MAX));
    }

    void update (int pos, int val) {
        a[pos] = val;

        // update tr-1
        int keep = val, posL = pos - 1, posR = pos + 1;
        int nL = posL - p(posL) + 1, nR = posR + p(posR) - 1;
        for (int k = pos; k < t1.size(); k += p(k)) {
            int l = k - p(k) + 1, r = k; // current node's range
            while (posL >= 1 && nL >= l) keep = min(keep, t1[posL]), posL = nL - 1, nL = posL - p(posL) + 1;
            while (posR <= n && nR <= r) keep = min(keep, t2[posR]), posR = nR + 1, nR = posR + p(posR) - 1;
            t1[k] = min(keep, a[k]);
        }

        // update tr-2
        keep = val, posL = pos - 1, posR = pos + 1;
        nL = posL - p(posL) + 1, nR = posR + p(posR) - 1;
        for (int k = pos; k > 0; k -= p(k)) {
            int l = k, r = k + p(k) - 1; // current node's range
            while (posL >= 1 && nL >= l) keep = min(keep, t1[posL]), posL = nL - 1, nL = posL - p(posL) + 1;
            while (posR <= n && nR <= r) keep = min(keep, t2[posR]), posR = nR + 1, nR = posR + p(posR) - 1;
            t2[k] = min(keep, a[k]);
        }
    }
};

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, q; cin >> n >> q;
    BIT tree(n);

    for (int i = 1; i <= n; i++) {
        int a; cin >> a;
        tree.update(i, a);
    }

    while (q--) {
        int type; cin >> type;
        if (type == 1) {
            int k, val; cin >> k >> val;
            tree.update(k, val);
        }
        else {
            int l, r; cin >> l >> r;
            cout << tree.query(l, r) << "\n";
        }
    }
```

```
77
78      return 0;
79  }
```

Bài 3: Truy vấn đếm

Subtask 1: Vét cạn

Trong subtask này ta sẽ vét cạn thông thường:

- Với truy vấn loại 1 (gán $a_k \leftarrow x$): ta gán giá trị a_k là giá trị x của đề vào mảng a hiện tại.
- Với truy vấn loại 2 (đếm số phần tử có giá trị y trong đoạn $[l; r]$): duyệt trâu từ a_l tới a_r và đếm xem có bao nhiêu giá trị bằng y và trả lời truy vấn.

Và với phương pháp trên thì độ phức tạp của thuật toán là $\mathcal{O}(nq)$.

Subtask 2: Fenwick Tree

Trong subtask này, do $1 \leq y \leq 10$ nên ta chỉ quan tâm 10 giá trị khác nhau. Khi đó ta có thể tạo 10 Fenwick tree tính tổng sao cho Fenwick tree thứ j, gọi là $\texttt{bit}_j$ sẽ thực hiện trả lời truy vấn trong đoạn $[l; r]$ có bao nhiêu phần tử có giá trị j.

- Với truy vấn loại 1 (gán $a_k \leftarrow x$): cập nhật $\texttt{bit}_{a_k}$, sau đó gán a_k là x và cập nhập $\texttt{bit}_x$.
- Với truy vấn loại 2 (đếm số phần tử có giá trị y trong đoạn $[l; r]$): tính tổng đoạn $[l; r]$ trên $\texttt{bit}_y$.

Và với phương pháp trên thì độ phức tạp của thuật toán là $\mathcal{O}(10 \cdot n \log n)$.

Subtask 3: Fenwick Tree + Cấu trúc dữ liệu

Ở subtask này, ta kết hợp cấu trúc dữ liệu `std::map` và Fenwick tree với mỗi nút (đỉnh) của Fenwick Tree được lưu dưới dạng `std::map`. Ở mỗi truy vấn, ta thêm phần tử vào tối đa $\log n$ nút khác nhau. Do đó, tổng số lượng phần tử trong các nút là $\mathcal{O}((n + q) \log n)$. Gọi $\texttt{bit}_i[x]$ là số lượng phần tử có giá trị là x trong nút thứ i của cây.

- Với truy vấn loại 1 (gán $a_k \leftarrow x$): cập nhật $\texttt{bit}_i[a_k] \leftarrow \texttt{bit}_i[a_k] - 1$ tại các đỉnh i có vùng quản lý chứa k, sau đó gán $a_k \leftarrow x$ và cập nhật $\texttt{bit}_i[x] \leftarrow \texttt{bit}_i[x] + 1$.
- Với truy vấn loại 2 (đếm số phần tử có giá trị y trong đoạn $[l; r]$): Tính tổng $\texttt{bit}_i[y]$ với i là các nút phủ đoạn cần truy vấn.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}(n \log^2 n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n \log n)$.

Cài đặt (tham khảo)

```cpp
#include <bits/stdc++.h>
#define fi first
#define se second
#define pii pair <int , int>
#define ar3 array <int , 3>

using namespace std;

const int INF = 1e9 + 7;
const int maxn = 2e5 + 7;
```

```cpp

int n , q , cur[maxn];
map <int,int> bit[maxn];


void update(int u, int v)
{
    int idx = u;

    while(idx <= n)
    {
        bit[idx][v]++;
        idx += (idx & (-idx));
    }

    idx = u;

    while(idx <= n)
    {
        bit[idx][cur[u]]--;
        idx += (idx & (-idx));
    }
    cur[u] = v;
}

int get(int p, int v)
{
    int idx = p;
    int ans = 0;

    while(idx > 0)
    {
        ans += bit[idx][v];
        idx -= (idx & (-idx));
    }
    return ans;
}

int getrange(int u, int v, int x)
{
    return get(v, x) - get(u-1, x);
}

void solve()
{
    cin >> n >> q;

    for(int i = 1; i <= n; i++) cur[i] = -INF;

    for(int i = 1; i <= n; i++)
    {
        int a_i; cin >> a_i; update(i , a_i);
    }

    while(q--)
    {
        int t; cin >> t;
        if(t == 2)
        {
            int l , r , x; cin >> l >> r >> x;
            cout << getrange(l , r , x) << '\n';
        }
        else
        {
            int p , x; cin >> p >> x;
            update(p , x);
        }
```

```
    }
}


signed main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    solve();
    return 0;
}
```

Bài 4: Truy vấn đồng dư

Subtask 1: Vét cạn

Ta nhận thấy n đủ nhỏ để chạy hai vòng lặp tính tổng các số đồng dư y modulo m. Vậy độ phức tạp của lời giải trên sẽ là $\mathcal{O}(nq)$.

Subtask 2: Fenwick Tree

Ở subtask này, ta thấy rằng $m = 1$. Như vậy các số sau khi modulo cho 1 đều bằng 0 (nghĩa là chia cho 1 dư 0). Vậy ta chỉ cần tính tổng các số trong đoạn từ $l \to r$ khi $y = 0$. Vì $n \leq 10^4$, độ phức tạp sẽ khá lớn nếu dùng cách tính như lời giải subtask 1, ta sẽ dùng Fenwick Tree thay vào đó để giảm độ phức tạp xuống để trả lời các truy vấn. Từ đó độ phức tạp của lời giải là $\mathcal{O}(Q \log n)$.

Subtask 3: Fenwick Tree + tối ưu

Ở subtask này, $m \leq 10$ nên mọi số khi modulo cho m đều sẽ nằm trong khoảng từ $0 \to m - 1$. Vì vậy ta gọi $f[i][j]$ tính tổng các số đồng dư i modulo m trong đoạn từ $1 \to j$ bằng Fenwick Tree.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}(10 \cdot (n + q) \log n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(10n)$.

Cài đặt (tham khảo)

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e4;
long long bit[10][N+5];
long long a[N+5];
int n,m,q;
void update(long long mm,long long id,long long val)
{
    for(;id<=n;id+=id & (-id))
        bit[mm][id]+=val;
    return;
}
long long get(long long id,long long mm)
{
    long long ans = 0;
    for(;id;id-=id & (-id))
        ans+=bit[mm][id];
    return ans;
}
int main()
{
    cin >> n >> m >> q;
```

```
    for(int i = 1;i<=n;i++)
    {
        cin >> a[i];
        update(a[i]%m,i,a[i]);
    }
    while(q--)
    {
        int s;
        cin >> s;
        if(s == 2)
        {
            int l,r,mod;
            cin >> l >> r >> mod;
            cout << get(r,mod) - get(l-1,mod) << "\n";
            continue;
        }
        int p;long long val;
        cin >> p >> val;
        update(a[p]%m,p,-a[p]);
        a[p]+=val;
        update(a[p]%m,p,a[p]);
    }
}
```

Bài 5: Tìm kho báu

Subtask 1: Vét cạn

Duyệt qua toàn bộ bản đồ và cộng các ô có khoảng cách Manhattan với ô (i, j) không quá d. Độ phức tạp $\mathcal{O}(q \cdot d^2)$.

Subtask 2: Fenwick Tree hai chiều

Tổng giá trị kho báu của các ô (x, y) có khoảng cách Manhattan với ô (i, j) không quá d chính là tổng giá trị của các ô nằm trong hình thoi có tâm tại (i, j) và có khoảng cách từ tâm tới 4 đỉnh là d. Rõ ràng, việc xử lý vùng hình thoi khó hơn vùng hình vuông nhiều.

Để ý rằng điều kiện $|i - x| + |j - y| \leq d$ có thể biến đổi thành:

$$\begin{cases} -d \leq i - x + j - y \leq d \\ -d \leq i - x - j + y \leq d \end{cases} \iff \begin{cases} i + j - d \leq x + y \leq i + j + d \\ i - j - d \leq x - y \leq i - j + d \end{cases}$$

Nói cách khác, ta chỉ xét các ô có chỉ số đường chéo thứ nhất $x + y$ trong khoảng $[i + j - d; i + j + d]$ và đường chéo thứ hai $x - y$ trong khoảng $[i - j - d; i - j + d]$.

Từ đây, ý tưởng của bài toán là xoay hệ trục tọa độ: ta đặt giá trị của ô (i, j) vào ô $(i+j, i-j)$ của Fenwick Tree hai chiều. Với các truy vấn tính tổng, ta chỉnh tổng hình chữ nhật con có gốc trên trái là ô $(i + j - d, i - j - d)$ và gốc dưới phải là ô $(i + j + d, i - j + d)$.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}(nm \log n \log m + q \log n \log m)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(nm)$.

Cài đặt (tham khảo)

Trong code dưới đây, chỉ số đường chéo $i - j$ đã được cộng lên một lượng `offset` $= m + 1$ để tránh chỉ số âm.

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

using ll = long long;
using ld = long double;
using pl = pair<ll,ll>;
using pii = pair<int,int>;
using tpl = tuple<int,int,int>;

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

struct BIT {
    vector<vector<ll>> tr;
    BIT (int n, int m) : tr(n + 1, vector<ll>(m + 1)) {}

    int p (int k) { return k & -k; }

    void update (int i, int j, ll incr) {
        for (int u = i; u < tr.size(); u += p(u))
            for (int v = j; v < tr[u].size(); v += p(v))
                tr[u][v] += incr;
    }

    ll preSum (int i, int j) {
        ll ans = 0;
        for (int u = i; u; u -= p(u))
            for (int v = j; v; v -= p(v))
                ans += tr[u][v];
        return ans;
    }

    ll query (int i, int j, int u, int v) {
        return preSum(u, v) - preSum(i - 1, v) - preSum(u, j - 1) + preSum(i - 1, j - 1);
    }
};

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, m, q; cin >> n >> m >> q;
    int offset = m + 1;
    BIT tree(n + m, n + offset);
    vector<vector<int>> a(n + 1, vector<int>(m + 1));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> a[i][j];
            tree.update(i + j, i - j + offset, a[i][j]);
        }
    }

    while (q--) {
        int type, i, j, d; cin >> type >> i >> j >> d;
        if (type == 1) {
            tree.update(i + j, i - j + offset, d - a[i][j]);
            a[i][j] = d;
        }
        else cout << tree.query(max(1, i + j - d), max(1, i - j + offset - d), min(n + m, i + j + d), min(n, i - j + d) + offset) << "\n";
    }

    return 0;
}
```

Bài 6: Đếm cặp nghịch thế

Subtask 1: Vét cạn

Cách tính $f(l, r)$ đơn giản nhất là sử dụng hai vòng lặp lồng nhau:

- Vòng lặp ngoài duyệt i từ l đến $r - 1$
- Vòng lặp trong duyệt j từ $i + 1$ đến r, đếm số cặp thoả điều kiện $a[i] > a[j]$.

Do giới hạn $n \leq 100$ nên ta có thể duyệt qua từng đoạn con $[l; r]$ bằng hai vòng lặp lồng nhau trong $\mathcal{O}(n^2)$. Rồi tính $f(l, r)$ cho từng đoạn con đó bằng cách trên trong $\mathcal{O}(n^2)$.

- Độ phức tạp thời gian: $\mathcal{O}(n^4)$

Subtask 2 + 3: Fenwick Tree

Đầu tiên, chúng ta có thể nén giá trị của mảng a xuống thành các giá trị nhỏ hơn bắt đầu từ 1, do số cặp nghịch thế của [100 30 70 12 13] và [5 3 4 1 2] là như nhau. Giá trị cụ thể của các phần tử không quan trọng, mà là thứ tự lớn dần của các phần tử, hay mỗi phần tử lớn hơn bao nhiêu phần tử khác.

Để nén giá trị, chúng ta có thể lưu một mảng khác đã được sắp xếp theo thứ tự tăng dần, rồi tìm nhị phân từng phần tử trong mảng ban đầu trong mảng đã sắp xếp để được mảng đã nén giá trị. Độ phức tạp thời gian cho quá trình này là $O(n \log n)$.

Chúng ta cài đặt BIT (Fenwick Tree) dưới dạng truy vấn tổng và cập nhật giá trị. BIT ở đây sẽ có vai trò lưu trữ số lượng các phần tử đứng trước với phần tử đã xét từ đầu đến phần tử đang xét hiện tại. Chúng ta làm vậy bằng cách sau mỗi vòng lặp, tăng phần tử thứ $a[i]$ cho i đơn vị.

Duyệt i qua từng phần tử từ 1 đến n. Chúng ta sử dụng BIT truy vấn tổng từ $[a[i] + 1; n]$ (không truy vấn từ $a[i]$ do chúng ta chỉ quan tâm đến những phần tử lớn hơn $a[i]$). Việc truy vấn tổng như vậy sẽ cho ta biết được tổng số lượng phần tử đứng trước các phần tử lớn hơn $a[i]$. Sau đó lấy tổng nhân với $(n - i + 1)$, là số lượng phần tử đứng sau $a[i]$.

Giải thích vì sao lại lấy tổng nhân với $(n - i + 1)$: Gọi 1 phần tử lớn hơn $a[i]$ xuất hiện trước nó là x. Các phần tử đứng trước x có thể là những điểm bắt đầu của đoạn con chứa cặp nghịch thế $(x, a[i])$. Các phần tử đứng sau $a[i]$ có thể là những điểm kết thúc của đoạn con chứa cặp nghịch thế $(x, a[i])$. Do đó, số lượng đoạn con chứa cặp nghịch thế $(x, a[i])$ được tính bằng cách nhân số lượng phần tử đứng trước x với số lượng phần tử đứng sau $a[i]$, tức là $(n - i + 1)$.

Sau đó chúng ta cập nhật biến kết quả. Sau khi truy vấn thì chúng ta cập nhật giá trị: tăng phần tử thứ $a[i]$ cho i đơn vị.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}(n \log n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n)$.

Cài đặt (tham khảo)

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

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

const int MOD = 1e9 + 22213;

int add (int a, int b) { return a + b - (a + b < MOD ? 0 : MOD); }
int sub (int a, int b) { return a - b + (a - b >= 0 ? 0 : MOD); }
int mul (int a, int b) { return 1LL * a * b % MOD; }

```

```cpp
struct BIT {
    vector<int> tr;
    BIT (int sz) : tr(sz + 1) {}

    int p (int k) { return k & -k; }

    void update (int k, int val) {
        for (; k < tr.size(); k += p(k)) tr[k] = add(tr[k], val);
    }

    int preSum (int k, int ans = 0) {
        for (; k; k -= p(k)) ans = add(ans, tr[k]);
        return ans;
    }

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

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n; cin >> n;
    vector<int> a(n + 1), cmp;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        cmp.push_back(a[i]);
    }

    cmp.push_back(0);
    sort(all(cmp)), filter(cmp);
    for (int i = 1; i <= n; i++)
        a[i] = lower_bound(all(cmp), a[i]) - cmp.begin();

    BIT tree(n); int ans = 0;
    for (int i = 1; i <= n; i++) {
        int cur = mul(tree.query(a[i] + 1, n), n - i + 1);
        ans = add(ans, cur);
        tree.update(a[i], i);
    }
    cout << ans;

    return 0;
}
```

Bài 7: Phần tử nhỏ thứ k

Subtask 1: Vét cạn

Với mỗi truy vấn loại 3, ta sắp xếp lại mảng a rồi in ra phần tử thứ k. Độ phức tạp $\mathcal{O}(qn \log n)$.

Subtask 2: Vét cạn + tối ưu

Ta duy trì mảng a luôn được sắp xếp. Do đó, với mọi truy vấn loại 1, ta tìm vị trí cần chèn phần tử mới và thêm phần tử vào đúng vị trí đó. Khi này, ta có thể trả lời truy vấn loại 3 trong $\mathcal{O}(1)$. Độ phức tạp $\mathcal{O}(qn)$.

Subtask 3: Tìm kiếm nhị phân + Walk on Fenwick Tree

Đầu tiên, nén các giá trị của mảng a (kể cả các giá trị cập nhật) rồi xây dựng mảng thống kê.

Ý tưởng tìm kiếm nhị phân

Gọi $\texttt{pre}[i]$ là số phần tử trong mảng a có giá trị không quá i. Để tìm phần tử thứ k, ta tìm M lớn nhất sao cho $\texttt{pre}[M] < k$, khi đó, đáp án là $M + 1$.

Để tìm M, ta cần quản lý mảng thống kê của a bằng Fenwick Tree và tìm kiếm nhị phân trên tổng tiền tố (prefix sum) của mảng thống kê này. Tuy nhiên, độ phức tạp là $\mathcal{O}(n \log n + q \log^2 n)$, chưa đủ để xử lý bài toán.

Walk on Fenwick Tree

Để tối ưu hơn nữa, ta sử dụng ý tưởng Walk on Fenwick Tree. Cụ thể, ta sẽ tìm kiếm nhị phân trên bit và tận dụng cấu trúc của Fenwick Tree để tăng tốc thuật toán.

Với tìm kiếm nhị phân trên bit, ta sẽ tham lam thử bật các bit từ lớn đến nhỏ. Giả sử hiện tại, đáp án tìm được là $\texttt{ans}$ và ta thử bật bit thứ i. Điều này đồng nghĩa với việc ta cần tính $\texttt{pre}[\texttt{ans} + 2^i]$ một cách nhanh chóng. Tuy nhiên, để ý rằng $p(\texttt{ans} + 2^i) = 2^i$ nên $\texttt{pre}[\texttt{ans} + 2^i] = \texttt{pre}[\texttt{ans}] + \texttt{tr}[\texttt{ans} + 2^i]$, với $\texttt{tr}[k]$ là nút thứ k của Fenwick Tree. Như vậy, ta chỉ cần lưu lại biến $\texttt{sum} = \texttt{pre}[\texttt{ans}]$ là có thể tính tổng tiền tố mới trong $\mathcal{O}(1)$.

Đến đây, ta đã có thể xử lý truy vấn trong $\mathcal{O}(\log n)$.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}((n + q) \log n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n + q)$.

Cài đặt (tham khảo)

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

typedef long long ll;
typedef long double ld;
typedef pair<ll,ll> pl;
typedef pair<int,int> pii;
typedef tuple<int,int,int> tpl;

#define all(a) a.begin(), a.end()
#define filter(a) a.erase(unique(all(a)), a.end())

struct BIT {
    vector<int> tr;
    BIT (int sz) : tr(sz + 1) {}

    int p (int k) { return k & -k; }

    void update (int k, int val) {
        for (; k < tr.size(); k += p(k)) tr[k] += val;
    }

    int walk (int targ) {
        int ans = 0, sum = 0;
        for (int mask = 1 << 19; mask; mask >>= 1)
            if ((ans | mask) < tr.size() && tr[ans | mask] + sum < targ) ans |= mask, sum += tr
    [ans];
        return ans + 1;
    }

    int preSum (int k, int ans = 0) {
        for (; k; k -= p(k)) ans += tr[k];
        return ans;
    }

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

```cpp
};

const int mn = 6e5 + 5;
pii saveQuery[mn];
vector<int> cmp;

int get (int targ) { return lower_bound(all(cmp), targ) - cmp.begin(); }

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, q; cin >> n >> q;
    int counter = 0;

    /// read initial array and queries
    for (int i = 1; i <= n; i++) {
        int a; cin >> a;
        saveQuery[++counter] = {1, a}; // each element is treated as a type-1 query
        cmp.push_back(a);
    }
    for (int i = 1; i <= q; i++) {
        int type, x; cin >> type >> x;
        if (type == 1) cmp.push_back(x);
        saveQuery[++counter] = {type, x};
    }

    /// compress values
    cmp.push_back(0); // make compressed value 1-indexed
    sort(all(cmp)), filter(cmp);

    /// process queries
    BIT tree(cmp.size());
    for (int i = 1; i <= counter; i++) {
        int type, x; tie(type, x) = saveQuery[i];
        if (type == 1) tree.update(get(x), 1);
        else if (type == 2) {
            int cur = get(x);
            if (cur >= cmp.size() || cmp[cur] != x) continue;
            if (tree.query(cur, cur)) tree.update(cur, -1);
        }
        else {
            int k = tree.walk(x);
            cout << cmp[k] << "\n";
        }
    }

    return 0;
}
```

Bài 8: Truy cập lịch sử

Subtask 1: Vét cạn

Trong subtask này ta sẽ thực hiện theo tư tưởng "đề bài nói gì làm nấy". Để cụ thể hóa hơn, nhận thấy rằng $n, q \leq 2000$, nên ta có thể lưu mỗi trạng thái của mảng sau mỗi lần truy vấn. Từ đó, với mỗi truy vấn loại 2, ta có thể dễ dàng truy lại mảng cần thiết và tính kết quả. Độ phức tạp $\mathcal{O}(n \cdot q)$.

Subtask 2: Xử lý offline

Ta nhận thấy rằng khi $t = 0$, ta sẽ không cần phải khôi phục truy vấn nữa.

Cách 1

Ta sẽ sort lại các truy vấn theo thời gian của nó (đối với truy vấn loại 1 thì thời gian của nó là index của nó trong dãy các truy vấn). Từ đây ta sẽ sử dụng CTDL BIT, việc sort các truy vấn xong đảm bảo được khi có các truy vấn loại 2 thì lúc đó sẽ đang sử dụng đúng mảng BIT. Độ phức tạp $\mathcal{O}(q \cdot \log q + q \cdot \log n)$.

Cách 2

Cách này sẽ xử lý theo một cách khéo léo khác, tư tưởng là khi ta cập nhật đến đâu thì sẽ xử lý truy vấn tìm kết quả cần thiết đến đó trước khi cập nhật khác. Cách này cũng khá tương tự với cách 2, tiết kiệm được thời gian sort truy vấn nhưng ngược lại tốn bộ nhớ nhiều hơn. Độ phức tạp $\mathcal{O}(q \cdot \log n)$.

Subtask 3: Fenwick Tree + xử lý online

Tại mỗi nút, ta sẽ có một vector để lưu lại các giá trị đã cập nhật lần trước, nói cách khác là lịch sử của nút này. Vì mỗi lần cập nhật ta chỉ cập nhật tối đa $\log n$ nút, nên nếu chúng ta có q lần cập nhật thì tối đa sẽ có $q \cdot \log n$ nút khác nhau.

Chi tiết hơn về cách cài đặt, mỗi nút sẽ lưu vector gồm dãy các pair {thời gian, giá trị}. Nếu ta cần cập nhật tại nút này chỉ cần lấy giá trị của nút cuối cùng rồi lưu thêm thời gian mới và cập nhật thôi. Đối với truy vấn đáp án, ta binary search nút cần thiết và lấy giá trị đó để thêm vào kết quả. Kỹ thuật này có tên gọi là **Fat node**.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}(n \log n + q \log^2 n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n \log n)$.

Cài đặt (tham khảo)

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

#define int unsigned long long
#define INF 1e18
#define f first
#define s second
#define pii pair<int, int>
#define vi vector<int>

const int MAXN = 2e5;
int n;
int a[MAXN + 5];
vector<pii> bit[MAXN + 5];
int ans = 0, flag = 0, cur = 0;

void dc(int &num) { // Decode queries
    num = num ^ (flag * ans * ans);
}

int get_val(int pos, int time) {
    return prev(upper_bound(bit[pos].begin(), bit[pos].end(), make_pair(time, INF)))->s;
}

void upd(int pos, int val) {
    ++cur;
    int delta = val ^ a[pos];
    a[pos] = val;
    while(pos <= n) {
        bit[pos].push_back({cur, delta ^ bit[pos].back().s});
        pos += (pos & (-pos));
    }
```

```cpp
}

int get(int pos, int time) {
    int res = 0;
    if(pos == 0) return 0;
    while(pos > 0) {
        res ^= get_val(pos, time);
        pos -= (pos & (-pos));
    }
    return res;
}

void solve()
{
    cin >> n;
    int q; cin >> q;
    cin >> flag;

    // Initialize BIT array
    for(int i = 1; i <= n; i++) bit[i].push_back({0, 0});

    // Add elements
    for(int i = 1; i <= n; i++) {
        int x; cin >> x;
        upd(i, x);
    }


    vi v = {cur}; // Array for storing states after queries

    while(q--) {
        int t; cin >> t;
        if(t == 1) {
            int k, x; cin >> k >> x;
            dc(k); dc(x);
            // cout << k << ' ' << x << '\n';
            upd(k, x);
        } else {
            int l, r, p; cin >> l >> r >> p;
            dc(l); dc(r); dc(p);
            ans = get(r, v[p]) ^ get(l - 1, v[p]);
            cout << ans << '\n';
        }
        v.push_back(cur);
    }
}

signed main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);
    // freopen("inp.txt", "r", stdin);
    // freopen("out.txt", "w", stdout);
    int t = 1;
    // cin >> t;
    while (t--)
        solve();
}
```

Bài 9: Thứ tự từ điển

Subtask 1: Vét cạn

Với giới hạn mảng và số lượng truy vấn là 5000, ta có thể thực hiện truy vấn loại 1 trong $\mathcal{O}(1)$ và loại 2 trong $\mathcal{O}(k)$ hay $\mathcal{O}(n)$ bằng cách so sánh tuần tự từng ký tự của 2 xâu truy vấn. Tổng độ phức tạp là $\mathcal{O}(q \cdot n)$.

Subtask 2 + 3: Fenwick tree + Hashing

Để giải được toàn bộ bài toán ta sẽ dùng Hash và CTDL Fenwick tree (trong phần này mình sẽ gọi là BIT - Binary Indexed Tree). Thực hiện lưu trữ mảng prefix Hash cho xâu s bằng BIT để có thể thay đổi xâu s và lấy giá trị Hash của xâu con của xâu s sau khi đã bị thay đổi để hỗ trợ thực hiện truy vấn so sánh.

Phân tích BIT

Phân tích sơ lược: CTDL BIT sẽ gồm một biến n là kích thước của cây và mảng `tree` có độ dài n sẽ chứa giá trị trong từng node của cây.

Khởi tạo một BIT dùng để lấy tổng tiền tố của một mảng $a[]$. Giá trị của $\texttt{tree}[i]$ là:

$$\texttt{tree}[i] = a[i] + a[i-1] + \cdots + a[i-p(i)+1]$$

Với $p(i) = i \ \& \ -i$. Phép toán $-i \ \& \ i$ sẽ trả về 2^k lớn nhất sao cho i chia hết cho 2^k – một phép toán thường thấy khi sử dụng BIT. Ví dụ nếu BIT trên có $n = 5$:

- $\texttt{tree}[1] = a[1]$.
- $\texttt{tree}[2] = a[2] + a[1]$.
- $\texttt{tree}[3] = a[2]$.
- $\texttt{tree}[4] = a[4] + a[3] + a[2] + a[1]$.
- $\texttt{tree}[5] = a[5]$.

Tổng quát hơn, tại đỉnh thứ i của BIT quản lý mảng a thì sẽ chứa kết quả giá trị dựa trên các phần tử $a[i],\ a[i-1],\ \cdots,\ a[i-p(i)+1]$.

Rolling Hash

Bạn đọc có thể tìm hiểu rõ hơn về thuật toán Rolling Hash tại đây, phần lời giải chỉ sẽ tập trung vào cách kết hợp thuật toán này với BIT.

Lời giải sử dụng Hash với modulo $\texttt{MOD} = 10^9 + 7$ và cơ số $\texttt{base} = 307$. Đồng thời, ta khởi tạo mảng `bp` nhau sau:

$$\texttt{bp}[i] = \texttt{base}^i \bmod \texttt{MOD} \quad \forall 0 \le i \le 10^6$$

"Dynamic Hash" với BIT

Lời giải sử dụng xâu ký tự được đánh số từ 1.

Khi dùng BIT để khởi tạo dãy Hash trên tiền tố của xâu s (gọi là Hash BIT để phân biệt với BIT thông thường), thì mảng $\texttt{tree}[i]$ sẽ chứa mã Hash của xâu con $s_{i-p(i)+1}s_{i-p(i)+2}\ldots s_i$. Ví dụ, xâu s = "abcde":

- $\texttt{tree}[1]$ là mã Hash của xâu con "a".
- $\texttt{tree}[2]$ là mã Hash của xâu con "ab".
- $\texttt{tree}[3]$ là mã Hash của xâu con "c".
- $\texttt{tree}[4]$ là mã Hash của xâu con "abcd".
- $\texttt{tree}[5]$ là mã Hash của xâu con "e".

Quay lại với Hash, trong một xâu s có độ dài n thì mã Hash của s là:

$$\texttt{hash}(s) = s_1 \cdot \texttt{base}^{n-1} + s_2 \cdot \texttt{base}^{n-2} + \cdots + s_n \cdot \texttt{base}^{n-n}$$

Vậy, ký tự thứ i trong xâu s sẽ đóng góp vào mã Hash một giá trị $s_i \cdot \texttt{base}^{n-i}$. Do đó khi thực hiện truy vấn thay đổi một ký tự trong xâu s ta thực hiện như sau:

```
void update(int pos, int val){
    for(int t=0, idx=pos; idx<=n; t+=-idx&idx, idx+=-idx&idx){
        tree[idx] = (tree[idx] + (val * bp[t]))%MOD;
    }
}
```

- `val` $= c - s_i$. Sau đó gán $s_i = c$.
- t là giá trị đại diện cho khoảng cách từ vị trí `pos` đến đầu mút bên phải của xâu con mà node `idx` đang quản lý.

Tương tự, hàm tính mã Hash của tiền tố / xâu con của xâu s được thực hiện như sau:

```
int get(int pos){
    int res = 0;
    for(int t=0, idx=pos; idx; t+=-idx&idx, idx-=-idx&idx){
        res = (res + (tree[idx] * bp[t])) % MOD;
    }
    return res;
}

int get(int l, int r){
    return (get(r) + -(get(l - 1) * bp[r - l + 1])) % MOD;
}
```

- t là giá trị đại diện cho độ lớn xâu mà mã Hash `res` đã tạo được.

Lời giải

Để so sánh 2 xâu con a, b có độ dài k về thứ tự từ điển ta sẽ thực hiện tìm độ dài tiền tố chung dài nhất của 2 xâu con gọi là p. Sau đó ta sẽ xét các trường hợp:

- $p = k \Rightarrow a = b$.
- $a[p] < b[p] \Rightarrow a < b$.
- $a[p] > b[p] \Rightarrow a > b$.

Để tìm được p ta sẽ dùng tìm kiếm nhị phân theo độ dài của tiền tố chung cần của 2 xâu a, b cộng với Hash BIT để so khớp chuỗi.

Như vậy, độ phức tạp của truy vấn 1 là $\mathcal{O}(\log n)$, độ phức tạp của 2 là $\mathcal{O}(\log^2 n)$.

Độ phức tạp

- Độ phức tạp thời gian: $\mathcal{O}(n \log n + q \log^2 n)$.
- Độ phức tạp bộ nhớ: $\mathcal{O}(n)$.

Cài đặt (tham khảo)

```

#include <bits/stdc++.h>
#define endl '\textbackslashn'
using namespace std;x

const int MOD = 998244353;
const int base = 307;
const int MX = 200005;
int n, q;
int bp[MX];
string s;
```

```cpp

int add(int a, int b){
    return (a += b) + (a >= MOD? -MOD : a<0? MOD : 0);
}

int mul(long long a, long long b){
    return a * b % MOD;
}

struct BIT{
    int n;
    vector<int> tree;
    BIT(int n = 0) : n(n), tree(n + 5, 0) {}

    void update(int idx, int val){
        for(int t=0; idx<=n; t+=-idx&idx, idx+=-idx&idx){
            tree[idx] = add(tree[idx], mul(val, bp[t]));
        }
    }

    int get(int idx){
        int res = 0;
        for(int t=0; idx; t+=-idx&idx, idx-=-idx&idx){
            res = add(mul(tree[idx], bp[t]), res);
        }
        return res;
    }

    int get(int l, int r){
        return add(get(r), -mul(get(l - 1), bp[r - l + 1]));
    }
};

int32_t main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    bp[0] = 1;
    for(int i=1; i<MX; i++) bp[i] = mul(bp[i-1], base);
    cin >> n >> q;
    cin >> s;
    s = ' ' + s;
    BIT bit(n);
    for(int i=1; i<=n; i++) bit.update(i, s[i]);
    while(q--){
        int oper;
        cin >> oper;
        if(oper == 1){
            int i; char x;
            cin >> i >> x;
            bit.update(i, x - s[i]);
            s[i] = x;
        }
        if(oper == 2){
            int i, j, k;
            cin >> i >> j >> k;
            int d = 0;
            for(int lo=1, hi=k; lo<=hi;){
                int mid = (lo + hi)>>1;
                if(bit.get(i, i + mid - 1) == bit.get(j, j + mid - 1)) d = mid, lo = mid + 1;
                else hi = mid - 1;
            }
            if(d == k) cout << "=\textbackslashn";
            else cout << (s[i + d] < s[j + d]? "<\textbackslashn" : ">\textbackslashn");
        }
    }

```

```
79      return 0;
80  }
```

— HẾT —