**Cây phân đoạn (Segment Tree)**

Cây phân đoạn (Segment Tree) là gì? Nó dùng để giải quyết bài toán nào?

**a. Định nghĩa:**

Cây phân đoạn (Segment Tree) là một cấu trúc dữ liệu dạng cây nhị phân, được thiết kế để lưu trữ thông tin về các đoạn (khoảng) trong một mảng. Mỗi nút của cây biểu diễn một đoạn con (subsegment) của mảng ban đầu, và lưu trữ thông tin đặc trưng cho đoạn đó (ví dụ: tổng, giá trị nhỏ nhất, giá trị lớn nhất,...).

**b. Ứng dụng chính:**

Segment Tree chủ yếu được sử dụng để giải quyết **các bài toán truy vấn và cập nhật trên đoạn con của mảng một cách hiệu quả**, đặc biệt là khi có **nhiều truy vấn** và/hoặc **cập nhật giá trị** trong mảng.

3. Các bài toán thường gặp có thể giải bằng Segment Tree:

|  |  |  |
| --- | --- | --- |
| **Bài toán** | **Miêu tả** | **Thời gian xử lý** |
| Tính tổng đoạn [L, R] | Tìm tổng các phần tử từ chỉ số L đến R trong mảng | O(log n) |
| Tìm giá trị nhỏ nhất/lớn nhất trong đoạn [L, R] | Tìm min/max trên đoạn con | O(log n) |
| Cập nhật giá trị tại vị trí i | Thay đổi giá trị phần tử tại vị trí i trong mảng | O(log n) |
| Tăng/giảm tất cả phần tử trong đoạn [L, R] | Dạng nâng cao, dùng Lazy Propagation | *O(log n)* |

**c. Ưu điểm của Segment Tree:**

- **Hiệu quả:** Truy vấn và cập nhật đều trong thời gian **O(log n)**, nhanh hơn so với cách duyệt tuyến tính O(n).

- **Linh hoạt:** Có thể tùy biến để xử lý nhiều loại phép toán: tổng, min, max, GCD, XOR,...

- **Phù hợp với bài toán động:** Tức là các bài toán mà dữ liệu thay đổi theo thời gian, cần cập nhật thường xuyên.

Cấu trúc cây Segment Tree

Segment Tree là một cấu trúc dữ liệu được sử dụng rất nhiều trong các kỳ thi, đặc biệt là trong những bài toán xử lý trên dãy số.

### **Cấu trúc và cách xây dựng Segment Tree**

Segment Tree được xây dựng dựa trên nguyên lý **chia để trị**:

* Với một mảng gồm N phần tử, nút gốc của cây sẽ quản lý toàn bộ đoạn [1, N].
* Nút con trái sẽ quản lý nửa trái: [1, ⌊N/2⌋].
* Nút con phải sẽ quản lý nửa phải: [⌊N/2⌋ + 1, N].

tổng quát hơn:  
Nếu một nút quản lý đoạn [i, j] thì:

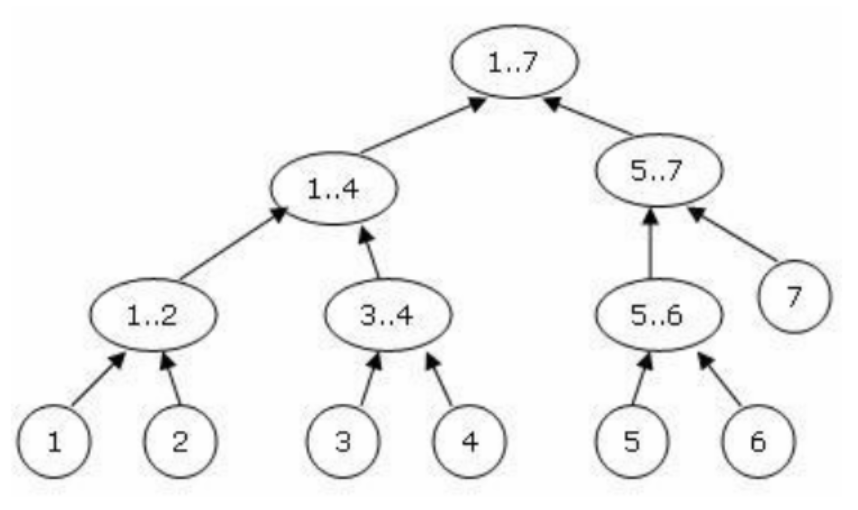
* Con trái quản lý đoạn [i, mid], với mid = ⌊(i + j) / 2⌋.
* Con phải quản lý đoạn [mid + 1, j].

**Mỗi nút của cây lưu trữ một thông tin nào đó về đoạn mà nó quản lý**, chẳng hạn như:

* Tổng các phần tử trong đoạn.
* Giá trị nhỏ nhất/lớn nhất trong đoạn.
* Số lượng các phần tử thỏa mãn điều kiện nào đó.

Cây này có chiều cao xấp xỉ log₂N, nên mỗi thao tác **truy vấn hoặc cập nhật** chỉ mất **O(log N)** thời gian.

**Ví dụ**: Xét một dãy gồm 7 phần tử, Segment Tree sẽ quản lý các đoạn như sau:



**Cài đặt**

Để cài đặt, ta có thể dùng một mảng 1 chiều, phần tử thứ nhất của mảng thể hiện nút gốc. Phần tử thứ id sẽ có 2 con là 2∗id (con trái) và 2∗id + 1 (con phải). Với cách cài đặt này, người ta đã chứng minh được bộ nhớ cần dùng cho ST không quá 4∗N phần tử.

**Áp dụng**

**Áp dụng**

Để dễ hình dung, ta lấy 1 ví dụ cụ thể:

* Cho dãy N phần tử (N ≤ 105). Ban đầu mỗi phần tử có giá trị 0.
* Có Q truy vấn (Q ≤ 105). Mỗi truy vấn có 1 trong 2 loại:

1. Gán giá trị v cho phần tử ở vị trí i.
2. Tìm giá trị lớn nhất cho đoạn [i,j].

|  |  |
| --- | --- |
| **input** | **ouput** |
| 5 6  1 2 10  1 4 5  2 1 5  1 3 7  2 2 4  2 1 3 | 10  10  10 |

Cách đơn giản nhất là dùng 1 mảng A duy trì giá trị các phần tử. Với thao tác 1 thì ta gán A[i]=v. Với thao tác 2 thì ta dùng 1 vòng lặp từ i đến j để tìm giá trị lớn nhất. Rõ ràng cách này có độ phức tạp là O(N∗Q) và không thể chạy trong thời gian cho phép.

**Cách dùng Segment Tree như sau:**

* Với truy vấn loại 1, ta sẽ cập nhật thông tin của các nút trên cây ST mà đoạn nó quản lý chứa phần tử i.
* Với truy vấn loại 2, ta sẽ tìm tất cả các nút trên cây ST mà đoạn nó quản lý nằm trong [i,j], rồi lấy max của các nút này.

**Có 3 thao tác:**

**1. Thao tác xây dựng dữ liệu cho cây (build)**

+) Dùng để **khởi tạo cây** từ dãy số ban đầu.

+) Thường được gọi **một lần duy nhất**, trước khi thực hiện các truy vấn.

+) Cách làm: xây cây nhị phân từ dưới lên, mỗi nút là **tổng, min, max, UCLN, số lượng…** tùy theo yêu cầu đề bài.

|  |
| --- |
| void build(int id, int l, int r) {  if (l == r) {  st[id] = a[l]; // hoặc khởi tạo giá trị gì đó  return;  }  int mid = (l + r) / 2;  build(2\*id, l, mid);  build(2\*id+1, mid+1, r);  st[id] = combine(st[2\*id], st[2\*id+1]); // ví dụ: max, sum, gcd, ...  } |

**2.1 Thao tác update dữ liệu vào cây**

* Thao tác truy xuất dữ liệu 🡪 kết quả
* Dùng để **thay đổi giá trị phần tử** hoặc **cập nhật đoạn** trong dãy.

 Có 2 loại cập nhật:

* **Cập nhật 1 phần tử** (thường update lên cây theo đường đi từ gốc đến lá).
* **Cập nhật 1 đoạn** (thường dùng Lazy Propagation để tối ưu).

|  |
| --- |
| // Cập nhật 1 phần tử tại vị trí i thành giá trị v  void update(int id, int l, int r, int i, int v) {  if (i < l || i > r) return;  if (l == r) {  st[id] = v;  return;  }  int mid = (l + r) / 2;  update(2\*id, l, mid, i, v);  update(2\*id+1, mid+1, r, i, v);  st[id] = combine(st[2\*id], st[2\*id+1]);  } |

### **3. Thao tác truy vấn (Query / Get)**

* Dùng để **tính toán và lấy kết quả** trên một đoạn [u, v].
* Phổ biến nhất là: **tổng đoạn, giá trị lớn nhất, nhỏ nhất, GCD đoạn,...**

|  |
| --- |
| // Truy vấn kết quả đoạn [u, v]  int get(int id, int l, int r, int u, int v) {  if (v < l || r < u) return base\_case; // VD: 0 cho tổng, -∞ cho max,...  if (u <= l && r <= v) return st[id];  int mid = (l + r) / 2;  return combine(  get(2\*id, l, mid, u, v),  get(2\*id+1, mid+1, r, u, v)  );  } |

**Code bài toán:**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  const int MAXN = 1e5 + 5;  const int INF = 1e9 + 7;  // Cây phân đoạn lưu trữ max  int ST[4 \* MAXN];  int N, Q;  void update(int id, int l, int r, int i, int v) {  if (i < l || i > r) return; // Không nằm trong đoạn quản lý  if (l == r) {  ST[id] = v;  return;  }  int mid = (l + r) / 2;  update(id \* 2, l, mid, i, v);  update(id \* 2 + 1, mid + 1, r, i, v);  ST[id] = max(ST[id \* 2], ST[id \* 2 + 1]);  }  // Truy vấn giá trị lớn nhất trên đoạn [u, v]  int getMax(int id, int l, int r, int u, int v) {  if (v < l || r < u) return -INF; // Không giao nhau  if (u <= l && r <= v) return ST[id]; // Nằm hoàn toàn trong đoạn truy vấn  int mid = (l + r) / 2;  return max(getMax(id \* 2, l, mid, u, v),  getMax(id \* 2 + 1, mid + 1, r, u, v));  }  main()  {  ios::sync\_with\_stdio(false);  cin.tie(nullptr);  cin >> N;  cin >> Q;  for (int q = 1; q <= Q; ++q) {  int type;  cin >> type;  if (type == 1) {  int i, v;  cin >> i >> v;  update(1, 1, N, i, v);  } else if (type == 2) {  int l, r;  cin >> l >> r;  cout << getMax(1, 1, N, l, r) << "\n";  }  }  } |

**Phân tích thời gian chạy**

Mỗi thao tác truy vấn trên cây ST có độ phức tạp O(logN). Để chứng minh điều này, ta xét 2 loại thao tác trên cây ST:

1. Truy vấn 1 phần tử trên ST (giống thao tác update ở trên)
2. Truy vấn nhiều phần tử trên ST (giống thao tác get ở trên)

Đầu tiên ta có thể chứng minh được:

* Độ cao của cây ST không quá O(logN).
* Tại mỗi độ sâu của cây, không có phần tử nào nằm trong 2 nút khác nhau của cây.

**Thao tác loại 1**

Với thao tác này, ở mỗi độ sâu của cây, ta chỉ gọi đệ quy các con của nó không quá 1 nút. Phân tích đoạn code trên, ta xét các trường hợp:

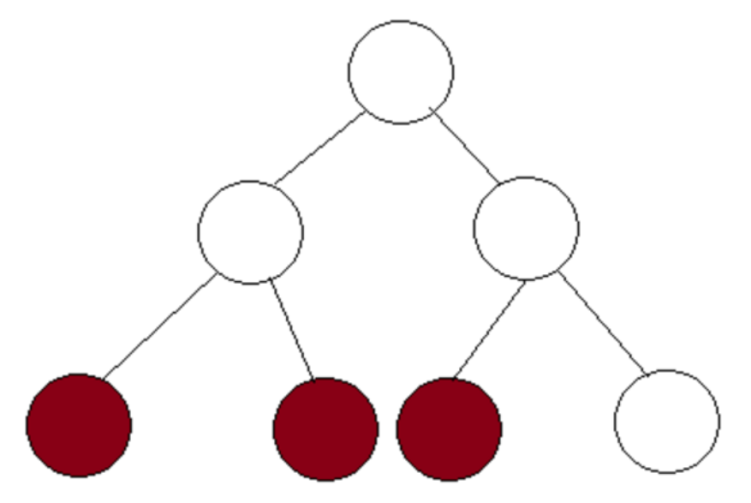
* Phần tử cần xét không nằm trong đoạn [l,r] do nút id quản lý. Trường hợp này ta dừng lại, không xét tiếp.
* Phần tử cần xét nằm trong đoạn [l,r] do nút id quản lý. Ta xét các con của nút id. Tuy nhiên chỉ có 1 con của nút id chứa phần tử cần xét và ta sẽ phải xét tiếp các con của nút này. Với con còn lại, ta sẽ dừng ngay mà không xét các con của nó nữa.

Do đó độ phức tạp của thao tác này không quá O(logN).

**Thao tác loại 2**

Với thao này, ta cũng chứng minh tương tự, nhưng ở mỗi độ sâu của cây, ta chỉ gọi hàm đệ quy với các con của nó không quá 2 nút.

Ta chứng minh bằng phản chứng, giả sử ta gọi đệ quy với 3 nút khác nhau của cây ST (đánh dấu màu đỏ):



Trong trường hợp này, rõ ràng toàn bộ đoạn của nút ở giữa quản lý nằm trong đoạn đang truy vấn. Do đó ta không cần phải gọi đệ quy các con của nút ở giữa. Từ đó suy ra vô lý, nghĩa là ở mỗi độ sâu ta chỉ gọi đệ quy với không quá 2 nút.

**Phân tích bộ nhớ**

Ta xét 2 trường hợp:

* N = 2k: Cây ST đầy đủ, ở độ sâu cuối cùng có đúng 2k lá, và các độ sâu thấp hơn không có nút lá nào (và các nút này đều có đúng 2 con). Như vậy:
* Tầng k: có 2k nút
* Tầng k−1: có 2k−1 nút
* ... Tổng số nút không quá 2k+1.
* Với N > 2k và N < 2k+1. Số nút của cây ST không quá số nút của cây ST với N=2k+1.

Do đó, số nút của cây cho dãy N phần tử, với N ≤ 2k là không quá 2k+1, giá trị này xấp xỉ 4∗N. Bằng thực nghiệm, ta thấy dùng 4∗N là đủ.

So sánh hiệu năng

|  |  |  |
| --- | --- | --- |
| **Phương pháp** | **Độ phức tạp** | **Có chạy được khi N, Q lớn?** |
| Duyệt trực tiếp | O(N × Q) | ❌ Quá chậm |
| Segment Tree | O(Q × logN) | ✅ Rất nhanh và hiệu quả |

**Từ bài toán 1: Xây dựng bài toán 2**

* Cho dãy số gồm N phần tử (N ≤ 105).
* Có Q truy vấn (Q ≤ 105). Mỗi truy vấn có 1 trong 2 loại:

1. Gán giá trị v cho phần tử ở vị trí i.
2. Tìm giá trị lớn nhất cho đoạn [i,j].

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  const int MAXN = 1e5 + 5;  int a[MAXN], st[4 \* MAXN];  int n, q;  // Hàm xây dựng segment tree  void build(int id, int l, int r) {  if (l == r) {  st[id] = a[l];  return;  }  int mid = (l + r) / 2;  build(2 \* id, l, mid);  build(2 \* id + 1, mid + 1, r);  st[id] = max(st[2 \* id], st[2 \* id + 1]);  }  // Hàm cập nhật giá trị tại vị trí pos  void update(int id, int l, int r, int pos, int val) {  if (l == r) {  st[id] = val;  return;  }  int mid = (l + r) / 2;  if (pos <= mid)  update(id \* 2, l, mid, pos, val);  else  update(id \* 2 + 1, mid + 1, r, pos, val);  st[id] = max(st[id \* 2], st[id \* 2 + 1]);  }  // Hàm truy vấn giá trị lớn nhất trong đoạn [u, v]  int get\_max(int id, int l, int r, int u, int v) {  if (v < l || r < u) return INT\_MIN;  if (u <= l && r <= v) return st[id];  int mid = (l + r) / 2;  int left = get\_max(id \* 2, l, mid, u, v);  int right = get\_max(id \* 2 + 1, mid + 1, r, u, v);  return max(left, right);  }  int main() {  ios::sync\_with\_stdio(false);  cin.tie(0);  cin >> n >> q;  // Khởi tạo mảng a ban đầu = 0  for (int i = 1; i <= n; i++) a[i] = 0;  build(1, 1, n);  while (q--) {  int type;  cin >> type;  if (type == 1) {  int i, v;  cin >> i >> v;  update(1, 1, n, i, v);  } else {  int i, j;  cin >> i >> j;  cout << get\_max(1, 1, n, i, j) << '\n';  }  }  return 0;  } |

### **Bài toán 3: Đề bài**:

Cho một mảng có **N phần tử** ban đầu rỗng (chưa có giá trị), bạn phải thực hiện **Q thao tác**, mỗi thao tác là một trong hai dạng:

* **Thao tác 1:**  
  1 i v — Gán giá trị v vào vị trí thứ i của mảng.  
  (Nếu vị trí đó đã có giá trị trước đó thì sẽ bị ghi đè.)
* **Thao tác 2:**  
  2 l r — Truy vấn ước chung lớn nhất (GCD) của các phần tử trong đoạn [l, r].

### **Yêu cầu**:

* Sau mỗi truy vấn loại 2, in ra **GCD của đoạn [l, r]**.
* Mảng có chỉ số từ 1 đến N.

### **Dữ liệu vào**:

* Dòng 1: Hai số nguyên N Q — số lượng phần tử của mảng và số lượng thao tác (1 ≤ N, Q ≤ 10⁵).
* Q dòng tiếp theo, mỗi dòng là một thao tác theo định dạng:
  + 1 i v (1 ≤ i ≤ N, 1 ≤ v ≤ 10⁹)
  + 2 l r (1 ≤ l ≤ r ≤ N)

### **Dữ liệu ra**:

* Với mỗi thao tác loại 2, in ra một dòng là **ước chung lớn nhất** của các số trong đoạn [l, r].

|  |  |
| --- | --- |
| **input** | **output** |
| 5 5  1 1 12  1 2 18  1 3 24  2 1 3  2 2 3 | 6  6 |

**Code tham khảo:**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  const int MAXN = 1e5 + 5;  int ST[4 \* MAXN]; // Segment Tree lưu GCD  int N, Q;  // Cập nhật giá trị tại vị trí i thành v  void update(int id, int l, int r, int i, int v) {  if (i < l || i > r) return; // Không thuộc đoạn quản lý  if (l == r) {  ST[id] = v; // Cập nhật tại lá  return;  }  int mid = (l + r) / 2;  update(id \* 2, l, mid, i, v);  update(id \* 2 + 1, mid + 1, r, i, v);  ST[id] = \_\_gcd(ST[id \* 2], ST[id \* 2 + 1]); // GCD 2 con  }  // Truy vấn GCD trên đoạn [u, v]  int getGCD(int id, int l, int r, int u, int v) {  if (v < l || r < u) return 0;  if (u <= l && r <= v) return ST[id];  int mid = (l + r) / 2;  return \_\_gcd(getGCD(id \* 2, l, mid, u, v),  getGCD(id \* 2 + 1, mid + 1, r, u, v));  }  main() {  ios::sync\_with\_stdio(false);  cin.tie(nullptr);  cin >> N >> Q;  while (Q--) {  int type;  cin >> type;  if (type == 1) {  int i, v;  cin >> i >> v;  update(1, 1, N, i, v);  } else if (type == 2) {  int l, r;  cin >> l >> r;  cout << getGCD(1, 1, N, l, r) << "\n";  }  }  } |

**II. KỸ THUẬT CẬP NHẬT LƯỜI (LAZY PROPAGATION)**

## **1. Đặt vấn đề**

### **Bài toán thường gặp:**

Bạn có một mảng A[1..N], và cần xử lý các truy vấn sau:

1. **Cập nhật đoạn**: Cộng thêm một giá trị val cho **tất cả các phần tử trong đoạn [l, r]**.
2. **Truy vấn đoạn**: Lấy giá trị lớn nhất hoặc tổng các phần tử trong đoạn [l, r].

**2. Ý tưởng của Lazy Propagation**

### Tư tưởng chính:

Thay vì cập nhật ngay lập tức tất cả các phần tử trong đoạn [l, r], ta **"hoãn lại" (lười biếng)** và **chỉ cập nhật khi thực sự cần**.

### Cơ chế hoạt động:

* Khi gặp một nút id trong cây Segment Tree quản lý một đoạn con nằm trong [l, r]:
  + Ta **chưa cần đi sâu xuống các nút con**.
  + Thay vào đó:
    - **Lưu lại rằng cần cập nhật sau** thông qua biến lazy[id].
    - Cập nhật **giá trị tổng quát** tại nút id ngay (để truy vấn sau không sai).
* Khi sau này cần truy vấn hoặc cập nhật các đoạn con:
  + Ta **đẩy giá trị lười từ nút cha xuống các con**.

|  |
| --- |
| struct Node {  int val; // Giá trị lớn nhất (hoặc tổng) trong đoạn  int lazy; // Giá trị cần "lười" cộng thêm vào các phần tử trong đoạn con  };  Node st[MAXN \* 4]; // Segment Tree |

## **a. Hàm đẩy lười (push / down)**

Trước khi gọi đệ quy xuống hai con 2\*id và 2\*id+1, ta cần **"đẩy" giá trị lazy** xuống:

|  |
| --- |
| void down(int id) {  int t = st[id].lazy;  st[id \* 2].val += t;  st[id \* 2].lazy += t;  st[id \* 2 + 1].val += t;  st[id \* 2 + 1].lazy += t;  st[id].lazy = 0; // Đã đẩy xong, reset lazy  } |

**b. Hàm cập nhật đoạn [u, v] với giá trị val**

|  |
| --- |
| void update(int id, int l, int r, int u, int v, int val) {  if (v < l || r < u) return; // Không giao nhau  if (u <= l && r <= v) {  // Nút [l, r] nằm hoàn toàn trong [u, v]  st[id].val += val;  st[id].lazy += val;  return;  }  // Có giao nhau một phần  down(id); // Đẩy lazy xuống trước  int mid = (l + r) / 2;  update(id \* 2, l, mid, u, v, val);  update(id \* 2 + 1, mid + 1, r, u, v, val);  // Cập nhật lại giá trị của nút hiện tại từ 2 con  st[id].val = max(st[id \* 2].val, st[id \* 2 + 1].val);  } |

c. Hàm truy vấn đoạn [u, v] (lấy max hoặc tổng)

|  |
| --- |
| **int get(int id, int l, int r, int u, int v) {**  **if (v < l || r < u) return -INFINITY; // Không giao**  **if (u <= l && r <= v) return st[id].val; // Nằm hoàn toàn trong**  **down(id); // Đẩy lazy xuống**  **int mid = (l + r) / 2;**  **return max(**  **get(id \* 2, l, mid, u, v),**  **get(id \* 2 + 1, mid + 1, r, u, v)**  **)**  **}** |

**3. Tại sao phải dùng Lazy Propagation?**

**Lazy Propagation** là một kỹ thuật rất mạnh để xử lý **cập nhật đoạn** nhanh trên cây Segment Tree.

Thay vì cập nhật từng phần tử, bạn **lưu lại việc cập nhật**, và chỉ thực hiện khi cần thiết.

Rất phù hợp cho:

- Bài toán tổng/giá trị lớn nhất trong đoạn.

- Khi cần cập nhật hàng loạt phần tử nhiều lần.

## **Vận dụng: Bài toán cụ thể :**

Link nộp bài:

<https://lequydon.ntucoder.net/Problem/Details/6120>

Cho mảng A[1..N], ban đầu gồm toàn số 0.

Yêu cầu thực hiện 2 loại truy vấn:

1. 0 l r val: Cộng thêm val vào tất cả các phần tử trong đoạn [l, r].
2. 1 l r: In ra **giá trị lớn nhất** trong đoạn [l, r].

**Giới hạn**:

* 1 ≤ N ≤ 10^5
* 1 ≤ Q ≤ 105
* 1 ≤ l ≤ r ≤ N
* |val| ≤ 106

|  |  |
| --- | --- |
| input | output |
| 5 6  0 1 3 5  1 1 5  0 2 5 2  1 1 5  0 3 4 -3  1 3 5 | 5  7  4 |

**Code tham khảo:**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  const int MAXN = 1e5 + 5;  struct Node {  int val; // giá trị lớn nhất trong đoạn  int lazy; // giá trị cần đẩy xuống  };  Node st[MAXN \* 4];  // Đẩy lazy từ nút id xuống hai con  void down(int id) {  int t = st[id].lazy;  st[id\*2].lazy += t;  st[id\*2].val += t;  st[id\*2+1].lazy += t;  st[id\*2+1].val += t;  st[id].lazy = 0;  }  // Cập nhật đoạn [u, v] tăng thêm val  void update(int id, int l, int r, int u, int v, int val) {  if (v < l || r < u) return;  if (u <= l && r <= v) {  st[id].val += val;  st[id].lazy += val;  return;  }  down(id);  int mid = (l + r) / 2;  update(id\*2, l, mid, u, v, val);  update(id\*2+1, mid+1, r, u, v, val);  st[id].val = max(st[id\*2].val, st[id\*2+1].val);  }  // Lấy max trong đoạn [u, v]  int get(int id, int l, int r, int u, int v) {  if (v < l || r < u) return INT\_MIN;  if (u <= l && r <= v) return st[id].val;  down(id);  int mid = (l + r) / 2;  return max(  get(id\*2, l, mid, u, v),  get(id\*2+1, mid+1, r, u, v)  );  }  int main() {  ios::sync\_with\_stdio(false);  cin.tie(nullptr);  int n, q;  cin >> n >> q;  while (q--) {  int type;  cin >> type;  if (type == 0) {  int l, r, val;  cin >> l >> r >> val;  update(1, 1, n, l, r, val);  } else {  int l, r;  cin >> l >> r;  cout << get(1, 1, n, l, r) << '\n';  }  }  } |

**II. BÀI TẬP ÁP DỤNG**

**Bài toán vận dụng:**

**Bài 1: https://lequydon.ntucoder.net/Problem/Details/6119**

Cho một dãy gồm *n* phần tử có giá trị ban đầu bằng 0.

Cho *m* phép biến đổi, mỗi phép có dạng (*u, v, k*): tăng mỗi phần tử từ vị trí *u* đến vị trí *v* lên *k* đơn vị.

Cho *q* câu hỏi, mỗi câu có dạng (u, v): cho biết phần tử có giá trị lớn nhất thuộc đoạn [*u, v*]

**Dữ liệu**

* Dòng 1*: n, m*
* *m* dòng tiếp theo, mỗi dòng chứa u, v, k cho biết một phép biến đổi
* Dòng thứ *m+2*: *p*
* *p* dòng tiếp theo, mỗi dòng chứa *u, v* cho biết một phép biến đổi

(n, m, q ≤ 105)

**Kết quả**

Gồm *p*dòng chứa kết quả tương ứng cho từng câu hỏi.

|  |  |
| --- | --- |
| **input** | **output** |
| 6 2 1 3 2 4 6 3 1 3 4 | 3 |

**Bài 2:** **https://lequydon.ntucoder.net/Problem/Details/6112**

Hàng ngày khi lấy sữa, N con bò của bác John (1 ≤ N ≤ 50000) luôn xếp hàng theo thứ tự không đổi. Một hôm bác John quyết định tổ chức một trò chơi cho một số con bò. Để đơn giản, bác John sẽ chọn ra một đoạn liên tiếp các con bò để tham dự trò chơi. Tuy nhiên để trò chơi diễn ra vui vẻ, các con bò phải không quá chênh lệch về chiều cao.

Bác John đã chuẩn bị một danh sách gồm Q (1 ≤ Q ≤ 200000) đoạn các con bò và chiều cao của chúng (trong phạm vi [1, 1000000]). Với mỗi đoạn, bác John muốn xác định chênh lệch chiều cao giữa con bò thấp nhất và cao nhất. Bạn hãy giúp bác John thực hiện công việc này!

### **Dữ liệu**

* Dòng đầu tiên chứa 2 số nguyên N và Q.
* Dòng thứ i trong số N dòng sau chứa 1 số nguyên duy nhất, là độ cao của con bò thứ i.
* Dòng thứ i trong số Q trong tiếp theo chứa 2 số nguyên A, B (1 ≤ A ≤ B ≤ N), cho biết đoạn các con bò từ A đến B.

### **Kết quả**

Gồm Q dòng, mỗi dòng chứa 1 số nguyên, là chênh lệch chiều cao giữa con bò thấp nhất và cao nhất thuộc đoạn tương ứng.

**Ví dụ:**

|  |  |
| --- | --- |
| **input** | **output** |
| 6 3 1 7 3 4 2 5 1 5 4 6 2 2 | 6 3 0 |

**Bài 3: https://lequydon.ntucoder.net/Problem/Details/6454**

Bạn được cung cấp một **tập hợp ban đầu rỗng**, và cần thực hiện lần lượt **N thao tác**. Mỗi thao tác thuộc một trong hai loại sau:

* **Thao tác loại 1**: 1 x — thêm số nguyên x vào tập hợp.
* **Thao tác loại 2**: 2 x — xóa **một lần xuất hiện** của số x khỏi tập hợp.  
  Dữ liệu luôn đảm bảo rằng **trước khi xóa, số x đã có ít nhất một lần trong tập**.

Sau **mỗi thao tác**, bạn cần in ra **ước chung lớn nhất (GCD)** của tất cả các số hiện có trong tập hợp.  
Nếu **tập hợp rỗng**, hãy in ra số **1**.

### **Dữ liệu vào**

* Dòng đầu tiên là một số nguyên **N** — số lượng thao tác (1 ≤ N ≤ 10⁵).
* N dòng tiếp theo, mỗi dòng gồm hai số nguyên **t** và **x**:
  + t là loại thao tác (1 hoặc 2).
  + x là số nguyên được thêm hoặc xóa (1 ≤ x ≤ 10⁹).

### **Dữ liệu ra**

* Gồm N dòng, mỗi dòng là **GCD của tập hợp** sau khi thực hiện thao tác tương ứng.
* Nếu tập hợp rỗng tại thời điểm đó, in ra **1**.

|  |  |  |
| --- | --- | --- |
| **input** | **output** | **Giải thích** |
| 6 1 8 1 12 1 10 1 8 2 8 2 8 | 8 4 2 2 2 2 | Tập hợp sau mỗi thao tác: • Thao tác 1: 8 • Thao tác 2: 8, 12 • Thao tác 3: 8, 12, 10 • Thao tác 4: 8, 12, 10, 8 • Thao tác 5: 8, 12, 10 • Thao tác 6: 12, 10 |

**Bài 4: Hiệu lớn nhất**

**https://oj.vnoi.info/problem/tinhyeumangtheo\_maxdif**

**1.1. Đề bài**

Cho dãy số nguyên 𝑎1,2,…,𝑎𝑛 và số nguyên dương 𝑘. Thực hiện phép xóa 𝑘 phần tử, sau đó sắp xếp các phần tử theo thứ tự tăng dần, gọi 𝑊 là hiệu lớn nhất giữa hai phần tử liên tiếp.

**Yêu cầu**: Tìm cách xóa để 𝑊 nhận giá trị nhỏ nhất.

**Input**

* + Dòng đầu chứa hai số nguyên dương 𝑛,𝑘 (𝑘 ≤ 𝑛 − 2);
  + Dòng thứ hai chứa 𝑛 số nguyên 𝑎1,𝑎2,…,𝑎𝑛 (|𝑎𝑖| ≤ 109);

**Output**

* Gồm một dòng chứa một số là giá trị 𝑊 nhỏ nhất tìm được.

|  |  |
| --- | --- |
| MAXDIF.INP | MAXDIF.OUT |
| 5 1  4 1 2 3 9 | 1 |

**Subtask 1**: 𝑛 ≤ 100;

**Subtask 2**: 𝑛 ≤ 2000;

**Hướng dẫn giải thuật**

Ta có *n* phần tử, suy ra có *n* - 1 khoảng cách giữa hai phần tử liên tiếp nhau.

Khi xóa đi *k* phần tử, ta còn lại *n - k* phần tử, suy ra có *n – k* – 1 khoảng cách giữa hai số liên tiếp nhau.

Gọi *ai* là khoảng cách giữa hai phần tử *i* và *i* + 1 (*i*: 1 🡪 *n*-1).

Vậy bài toán trở thành: Chọn đoạn gồm *n - k* – 1 phần tử liên tiếp trong *n* - 1 phần tử trong mảng *a* sao cho max của *n – k* - 1 phần tử là nhỏ nhất.

Dùng **Segment Tree** để tìm phần tử lớn nhất trong đoạn gồm *m* phần tử liên tiếp

**Bài 5:** <https://lequydon.ntucoder.net/Problem/Details/5924>

Nhân ngày lễ tình nhân, Ami quyết định đi leo núi. Ami đứng trước n ngọn núi được đánh số từ 1 đến n, mỗi ngọn núi có chiều cao là hi và một chỉ số boosting là di. Ami có thể bắt đầu leo núi ở bất kì ngọn núi nào và khi vượt qua ngọn núi i, Ami có thể dừng lại hoặc phải leo ở những ngọn núi có chỉ số lớn hơn i và có độ cao lớn hơn độ cao ngọn núi hiện tại ít nhất là di. Hệ thức hóa, giả sử Ami đang ở ngọn núi i có chiều cao hi và chỉ số boosting di, Ami sẽ được leo ngọn núi j nếu j > i và hj – hi >= di. Ami muốn leo nhiều núi nhất có thể, do đó các bạn được phép giúp Ami tìm ra lịch trình leo núi tối ưu.

**Dữ liệu vào**

Dòng đầu tiên là một số nguyên dương n (n <= 2 \* 105) là số ngọn núi

Dòng tiếp theo là n số nguyên dương hi (hi <= 106) là độ cao ngọn núi thứ i.

Dòng cuối cùng là n số nguyên dương di (di <= 106) là chỉ số boosting của ngọn núi i.

**Dữ liệu ra**

Một số nguyên là số ngọn núi nhiều nhất Ami có thể leo.

Ví dụ:

|  |  |
| --- | --- |
| **input** | **output** |
| 5  1 2 3 4 5  2 1 3 1 1 | 3 |