

**TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA CÔNG NGHỆ THÔNG TIN**



**ĐỒ ÁN LẬP TRÌNH TÍNH TOÁN
XÂY DỰNG THU VIỆN CÂY PHÂN ĐOẠN
CÀI ĐẶT BẰNG CON TRỎ**

Người hướng dẫn: TS. **Đặng Thiên Bình**

Sinh viên thực hiện:

Lưu Duy Quang 102220036 **NHÓM: 22Nh15B**

Nguyễn Văn Quyết 102220038 **NHÓM: 22Nh15B**

Đà Nẵng, tháng 6 năm 2023

MỤC LỤC

MỤC LỤC	1
DANH MỤC HÌNH VẼ	2
MỞ ĐẦU	3
1. TỔNG QUAN ĐỀ TÀI	4
1.1. Bài toán thực tế	4
1.2. Cách giải quyết tự nhiên	4
1.3. Hướng giải quyết bằng kỹ thuật mảng cộng dồn	4
1.4. Hướng giải quyết bằng cấu trúc dữ liệu cây phân đoạn	5
1.5. Cải thiện nhược điểm của cách cài đặt cây phân đoạn bằng phương pháp truyền thống	5
1.6. Tổng quan đề tài	5
2. CƠ SỞ LÝ THUYẾT	6
2.1. Ý tưởng và cơ sở lý thuyết của cây phân đoạn	6
2.2. Bài toán mở rộng, kỹ thuật Lazy Propagation	9
3. TỔ CHỨC CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN	9
3.1. Cấu trúc mã trong thư viện	9
3.2. Phát biểu bài toán	9
3.3. Cài đặt cấu trúc dữ liệu cây phân đoạn	9
3.4. Phát biểu bài toán mở rộng (ứng dụng kỹ thuật Lazy Propagation)	14
3.5. Cài đặt kỹ thuật Lazy Propagation	14
4. CHƯƠNG TRÌNH VÀ KẾT QUẢ	15
4.1. Ngôn ngữ cài đặt	15
4.2. Tính đơn giản, dễ sử dụng của thư viện	15
4.3. Tính tùy biến cao của thư viện	17
4.4. Sử dụng thư viện để giải quyết các bài toán trên trang cses	18
5. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN	19
5.1. Kết luận	19
5.2. Hướng phát triển	20
TÀI LIỆU THAM KHẢO	21
PHỤ LỤC	22

DANH MỤC HÌNH VẼ

Hình 1: Ảnh minh họa cây phân đoạn	7
Hình 2: Ảnh minh họa cách thức hoạt động của hàm cập nhật	8
Hình 3: Ảnh minh họa cách thức hoạt động của hàm lấy dữ liệu trong đoạn	8
Hình 4: Ảnh mô tả tổng quan các biến và hàm trong file segmentTreeNode.h	10
Hình 5: Ảnh mô tả tổng quan các biến và hàm trong file segmentTree.h	11
Hình 6: Ảnh mô tả tổng quan các biến và hàm trong file templateGeneralFunction.h	13
Hình 7: Ảnh mô tả tổng quan các biến và hàm trong kỹ thuật Lazy Propagation	15
Hình 8: Ảnh thể hiện sự gọn nhẹ trong mã nguồn khi xài thư viện	16
Hình 9: Ảnh thể hiện sự gọn nhẹ trong mã nguồn khi xài thư viện	17
Hình 10: Ảnh thể hiện khả năng tùy biến mã nguồn cao khi xài thư viện	18
Hình 11: Kết quả kiểm thử thư viện	19
Hình 12: Kết quả kiểm thử thư viện	19

MỞ ĐẦU

Cá trong lập trình thi đấu và lập trình ứng dụng, ta thường xuyên gặp những bài toán theo dạng cần trả lời những truy vấn trên đoạn và thay đổi giá trị của một phần tử trong mảng cho trước bằng một giá trị khác. Có nhiều cách để xử lý những bài toán dạng này, tùy thuộc vào những bộ dữ liệu khác nhau và các yêu cầu về thời gian, bộ nhớ khác nhau. Tuy nhiên, có một cấu trúc dữ liệu có thể giải quyết hiệu quả bài toán với đa số trường hợp với tốc độ nhanh và không chiếm quá nhiều bộ nhớ. Đó là cấu trúc dữ liệu cây phân đoạn.

Mặc dù là một cấu trúc dữ liệu mạnh mẽ như vậy, nhưng trong ngôn ngữ C++ chưa hề hỗ trợ thư viện chuẩn để giúp người lập trình cài đặt nhanh chóng và hiệu quả. Do vậy đề tài sẽ nghiên cứu xây dựng một thư viện hỗ trợ các thao tác trên cây phân đoạn đơn giản, trực quan và có độ tùy biến cao. Trong quá trình nghiên cứu, có một bài toán tối ưu các thao tác cập nhật đoạn trên cây phân đoạn là Lazy Propagation. Trong đề tài này cũng sẽ nghiên cứu và hỗ trợ thêm kỹ thuật trên thư viện.

Sau khi cài đặt và xây dựng xong thư viện, thư viện sẽ được chuyển về những file header (.h) để người lập trình dễ dàng sử dụng những tính năng có trong thư viện chỉ với một dòng include. Để đảm bảo độ chính xác về cả kết quả và thời gian, không gian thực thi, thư viện sẽ được kiểm thử thông qua những bài toán trên trang online judge nổi tiếng cses.

1. TỔNG QUAN ĐỀ TÀI

1.1. Bài toán thực tế:

Cho 1 dãy số có n số nguyên $x[]$, cần xử lý q truy vấn gồm 2 loại truy vấn sau:

- Truy vấn loại 1: Gán giá trị phần tử ở vị trí u bằng k
- Truy vấn loại 2: Tìm tổng các giá trị trong mảng $x[]$ trong đoạn $[a, b]$

1.2. Cách giải quyết tự nhiên

Cách giải quyết tự nhiên hay còn được gọi là vét cạn, nghĩa là, ta sẽ làm theo những gì đề bài yêu cầu:

Với truy vấn loại 1: Chúng ta chỉ cần gán $x[u] = k$

Với truy vấn loại 2: Chúng ta sẽ dùng một vòng lặp từ a đến b để tính tổng các phần tử

Với phương pháp giải quyết tự nhiên, ở trường hợp tệ nhất ta sẽ mất độ phức tạp thời gian là $O(n*q)$ vì ta:

- Xử lý truy vấn loại 1 trong độ phức tạp thời gian $O(1)$
- Xử lý truy vấn loại 2 trong độ phức tạp thời gian $O(n)$

Nếu bài toán có quá nhiều truy vấn 2, chương trình của chúng ta sẽ chạy rất chậm.

1.3. Hướng giải quyết bằng kỹ thuật mảng cộng dồn

Cách giải quyết này ta sẽ sử dụng kỹ thuật mảng cộng dồn để tối ưu truy vấn loại 2, là một trong những kỹ thuật cơ bản trong lập trình. Kỹ thuật được trình bày sơ lược như sau:

Ta sẽ gọi $f[i]$ là tổng các phần tử của mảng $x[]$ trong đoạn $[1, i]$. Ta xây được mảng cộng dồn bằng công thức quy hoạch động: $f[i] = f[i - 1] + x[i]$. Vậy, ta có thể tính tổng một đoạn con $[a, b]$ bằng công thức: $f[b] - f[a - 1]$.

Vậy ta sẽ xử lý bài toán 1.1 như sau:

- VỚI truy vấn loại 1: Ta sẽ xây dựng lại mảng cộng dồn từ vị trí u đến vị trí n
- VỚI truy vấn loại 2: Ta có kết quả của truy vấn sẽ là $f[b] - f[a - 1]$

Với phương pháp giải quyết trên ở trường hợp tệ nhất ta vẫn sẽ mất độ phức tạp là $O(n*q)$ vì ta:

- Xử lý truy vấn loại 1 trong độ phức tạp thời gian $O(n)$
- Xử lý truy vấn loại 2 trong độ phức tạp thời gian $O(1)$

Nếu bài toán có quá nhiều truy vấn 1, chương trình của chúng ta sẽ chạy rất chậm.

1.4. Hướng giải quyết bằng cấu trúc dữ liệu cây phân đoạn

Cây phân đoạn là một cấu trúc dữ liệu, giúp chúng ta có thể:

- Xử lý truy vấn loại 1 trong độ phức tạp thời gian $O(\log_2(n))$
- Xử lý truy vấn loại 2 trong độ phức tạp thời gian $O(\log_2(n))$

Nhưng độ phức tạp không gian của ta sẽ lên đến $O(4^*n)$ để lưu trữ hết tất cả các nút trong cây phân đoạn, nếu cài đặt bằng mảng bình thường. Hơn nữa, khi cài đặt bằng mảng, chúng ta sẽ phải tạo trước độ dài của mảng. Việc này dẫn đến những nhược điểm như sau:

- Nếu chúng ta đặt trước độ dài mảng quá nhỏ, khi gặp những bài toán yêu cầu kích thước cây phân đoạn lớn, chương trình không thể chạy được và sẽ báo lỗi trong quá trình chạy (Runtime error).
- Nếu chúng ta đặt trước độ dài mảng quá lớn, nếu không gặp những bài toán yêu cầu kích thước cây phân đoạn lớn, ta sẽ bỏ phí rất nhiều tài nguyên.

1.5. Cải thiện nhược điểm của cách cài đặt cây phân đoạn bằng phương pháp truyền thống

Phương pháp cài đặt tối ưu không gian hơn cho cây phân đoạn là sử dụng con trỏ. Khi đó, chúng ta giải quyết bài toán 1.1 với các ưu điểm so với những phương pháp khác như sau:

- Vẫn đảm bảo được độ phức tạp thời gian $O(\log_2(n))$ cho cả 2 loại truy vấn
- Độ phức tạp không gian nhỏ hơn cài đặt bằng mảng, vì chúng ta chỉ cần lưu trữ các nút con tồn tại thực của cây
- Dễ dàng xóa bộ nhớ sau khi không cần dùng cây nữa hoặc dùng chính cây đó để xử lý những bài toán tương đương

1.6. Tổng quan đề tài

Vì C++ chưa hỗ trợ thư viện cho cây phân đoạn nên nếu muốn sử dụng nó, người lập trình phải tự cài đặt. Như vậy sẽ tốn rất nhiều thời gian, công sức, chưa kể trong lúc cài đặt sẽ có nhiều nguy cơ sai sót dẫn đến chậm trễ tiến trình dự án và những hệ lụy lớn hơn. Do vậy, việc tạo ra một thư viện cây phân đoạn là cần thiết. Và như đã chỉ ra những ưu và nhược điểm trong hai phần 1.4 và 1.5, thư viện cho cây phân đoạn

được cài đặt bằng con trỏ thay vì mảng, và thỏa mãn được những nhu cầu như sau:

- Quan trọng nhất: đảm bảo được độ phức tạp thời gian và độ phức tạp không gian tối ưu cho bài toán
- Cách sử dụng thư viện đơn giản, thân thiện, mã ứng dụng ngắn gọn cho những bài toán chỉ áp dụng cây phân đoạn ở mức cơ bản.
- Khả năng tùy biến của thư viện tốt để đảm bảo giải quyết được những bài toán khó và phức tạp

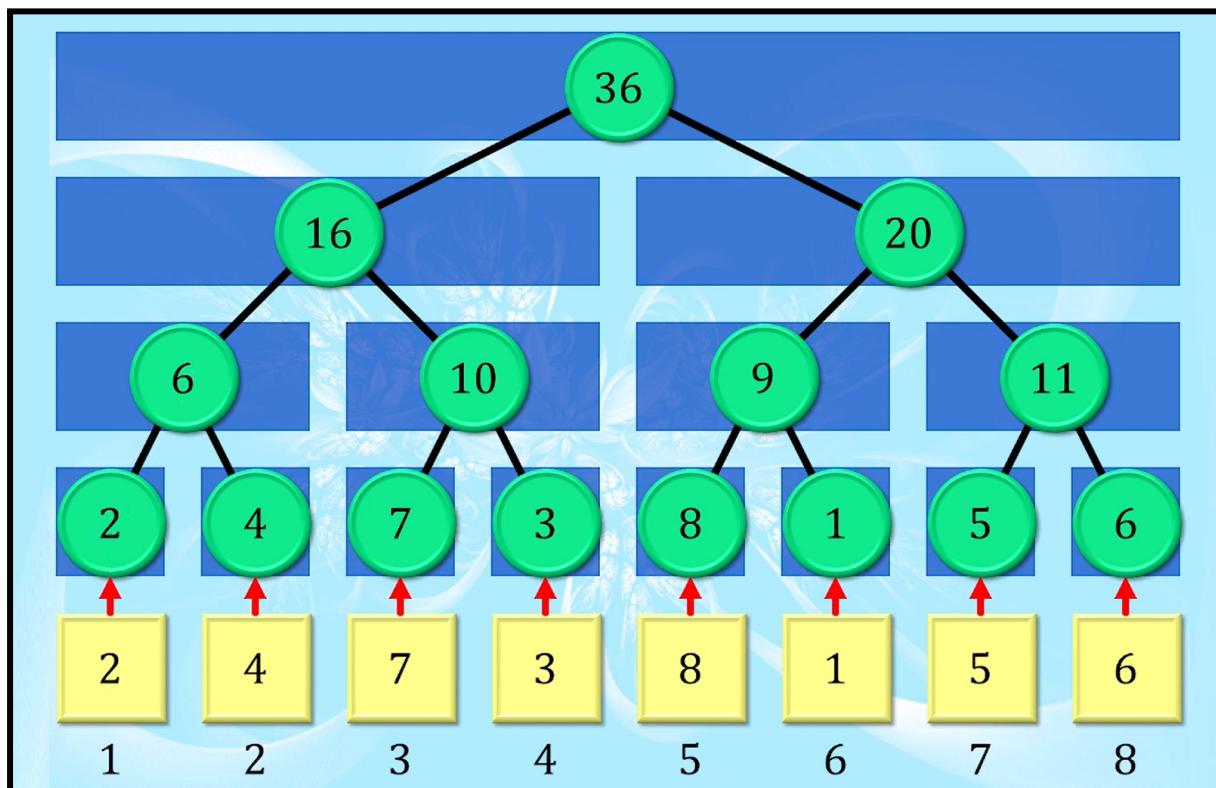
2. CƠ SỞ LÝ THUYẾT

2.1. Ý tưởng và cơ sở lý thuyết của cây phân đoạn

Cây phân đoạn là một cấu trúc dữ liệu cho phép thực hiện các thao tác truy vấn và cập nhật trên một đoạn các phần tử của mảng với chi phí thực hiện mỗi thao tác có độ phức tạp là hàm logarit. Cây phân đoạn được sử dụng rất nhiều trong các bài toán xử lý trên dãy số.

Cây phân đoạn được xây dựng dựa trên cấu trúc cây nhị phân. Mỗi nút của cây phân đoạn tương ứng với một đoạn của mảng. Nút gốc của cây phân đoạn tương ứng với toàn bộ mảng. Các nút con của một nút cha tương ứng với hai nửa của đoạn tương ứng.

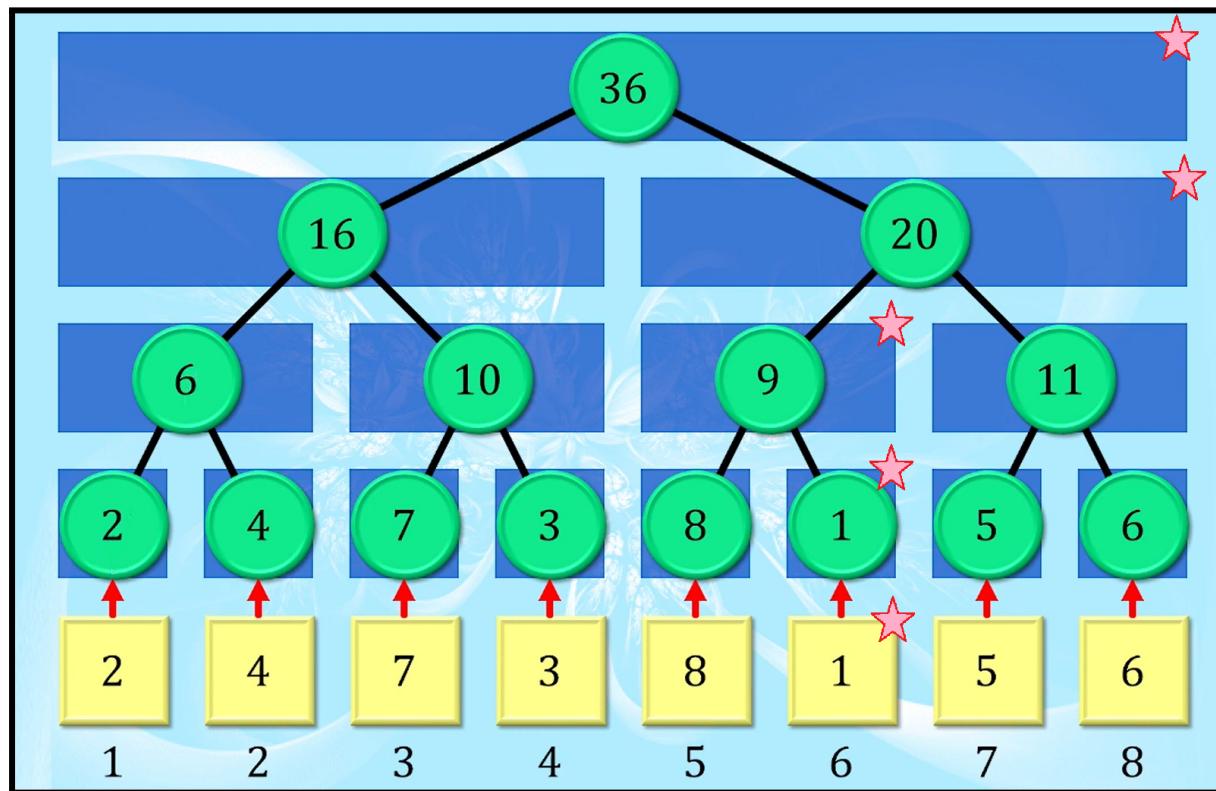
Mỗi nút của cây phân đoạn chứa thông tin về giá trị của đoạn tương ứng và thông tin liên quan đến các thao tác truy vấn và cập nhật. Các thao tác truy vấn và cập nhật được thực hiện trên cây phân đoạn bằng cách duyệt cây theo chiều từ gốc xuống các nút lá.



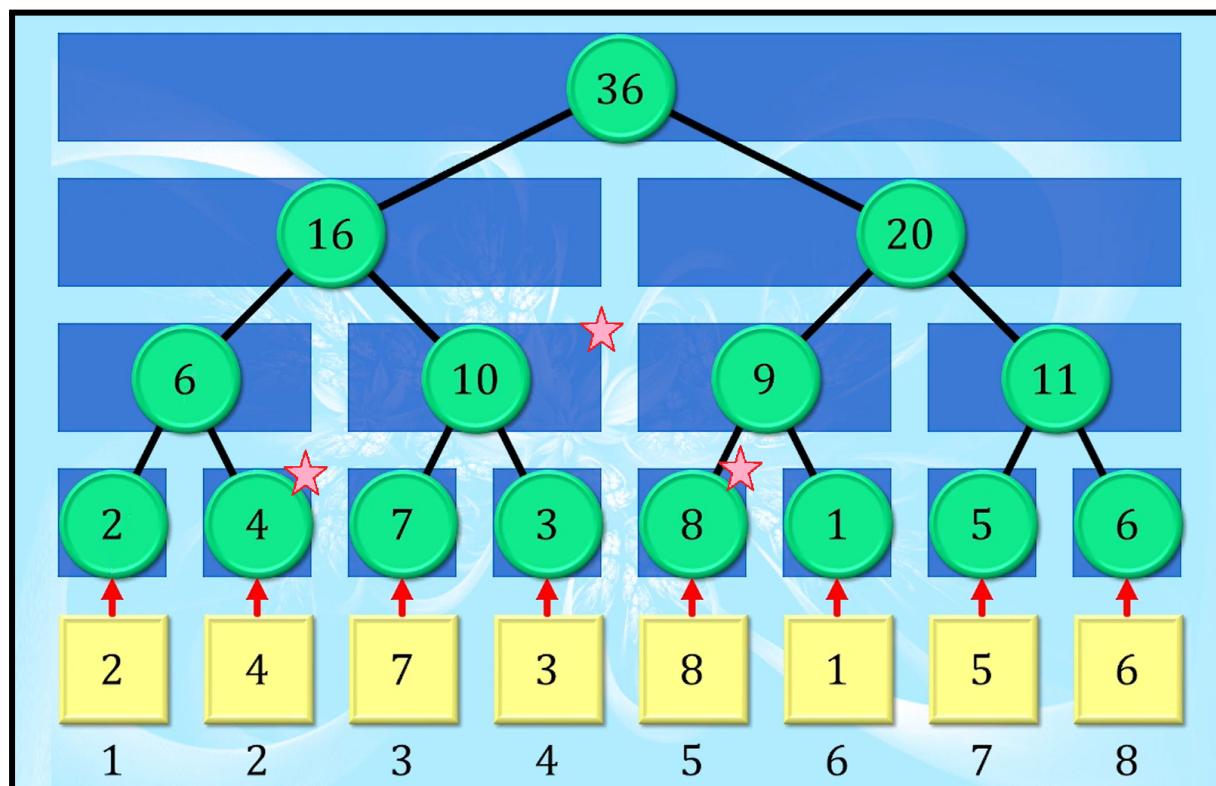
Cây phân đoạn có 2 thao tác chính: cập nhật và lấy giá trị trên một đoạn. Cập nhật là thao tác thực hiện thay đổi các giá trị trên mảng như: tăng phần tử ở vị trí u lên k đơn vị hay gán giá trị của phần tử ở vị trí u bằng k . Thao tác lấy giá trị sẽ thực hiện lấy thông tin cần thiết trên đoạn $[u, v]$ cho trong truy vấn. Thông tin đó có thể là tổng, tích, ước chung lớn nhất, ...

Với thao tác cập nhật giá trị trên phần tử ở vị trí i , cây phân đoạn sẽ cập nhật tất cả các nút có quản lý thông tin của nút i .

Còn với thao tác lấy giá trị trên một đoạn $[u, v]$ trong truy vấn, cây phân đoạn sẽ lấy thông tin của các nút trên cây sao cho các nút đó bao phủ đúng đoạn $[u, v]$, từ đó xác định kết quả cần tìm.



Hình ảnh trên minh họa ví dụ nếu ta thay đổi giá trị của phần tử ở vị trí thứ 6, tất cả nút có quản lý nút 6 sẽ được cập nhật.



Hình ảnh trên minh họa truy vấn lấy giá trị trong đoạn [2, 5]. Những nút được đánh dấu sao sẽ được lấy giá trị để tính toán giá trị của đoạn [2, 5]. Trong ví dụ này, đoạn [2, 5] sẽ có tổng là tổng của các đoạn [2, 2], [3, 4], [5, 5].

2.2. Bài toán mở rộng, kỹ thuật Lazy Propagation

Kỹ thuật Lazy Propagation được sử dụng trong cây phân đoạn để giảm độ phức tạp của các truy vấn cập nhật đoạn. Ý tưởng của kỹ thuật này là không thực hiện các truy vấn cập nhật ngay lập tức mà chỉ đánh dấu các nút cần cập nhật và trì hoãn việc thực hiện cho đến khi có truy vấn xuất đến nút đó.

Ví dụ như chúng ta cần tăng tất cả các phần tử của mảng trong đoạn [1, 4] lên k đơn vị. Thay vì cập nhật các nút con quản lý đoạn [1, 1], [2, 2], [3, 3], [4, 4], chỉ cần cập nhật giá trị ở nút [1, 4] và lưu một giá trị lazy ở nút [1, 4] thông báo rằng: “Tất cả những phần tử trong mảng do nút này quản lý đều sẽ được tăng lên k đơn vị”. Sau này, nếu cần sử dụng những nút con của nút [1, 4], chúng ta mới thực sự cần đi xuống và cập nhật.

3. TỔ CHỨC CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

3.1. Cấu trúc mã trong thư viện

1. segmentTreeNode.h: quản lý thông tin được lưu trữ trên một nút của cây nhị phân
2. templateTreeNode.h: các template có sẵn giúp việc sử dụng thư viện dễ dàng hơn
3. segmentTree.h: quản lý thông tin được lưu trữ trong một cây phân đoạn
4. main.cpp: file mẫu mô phỏng cách sử dụng thư viện giải quyết một bài toán thực tế

3.2. Phát biểu bài toán

Cho 1 dãy số có n số nguyên $x[]$, cần xử lý q truy vấn gồm 2 loại truy vấn sau:

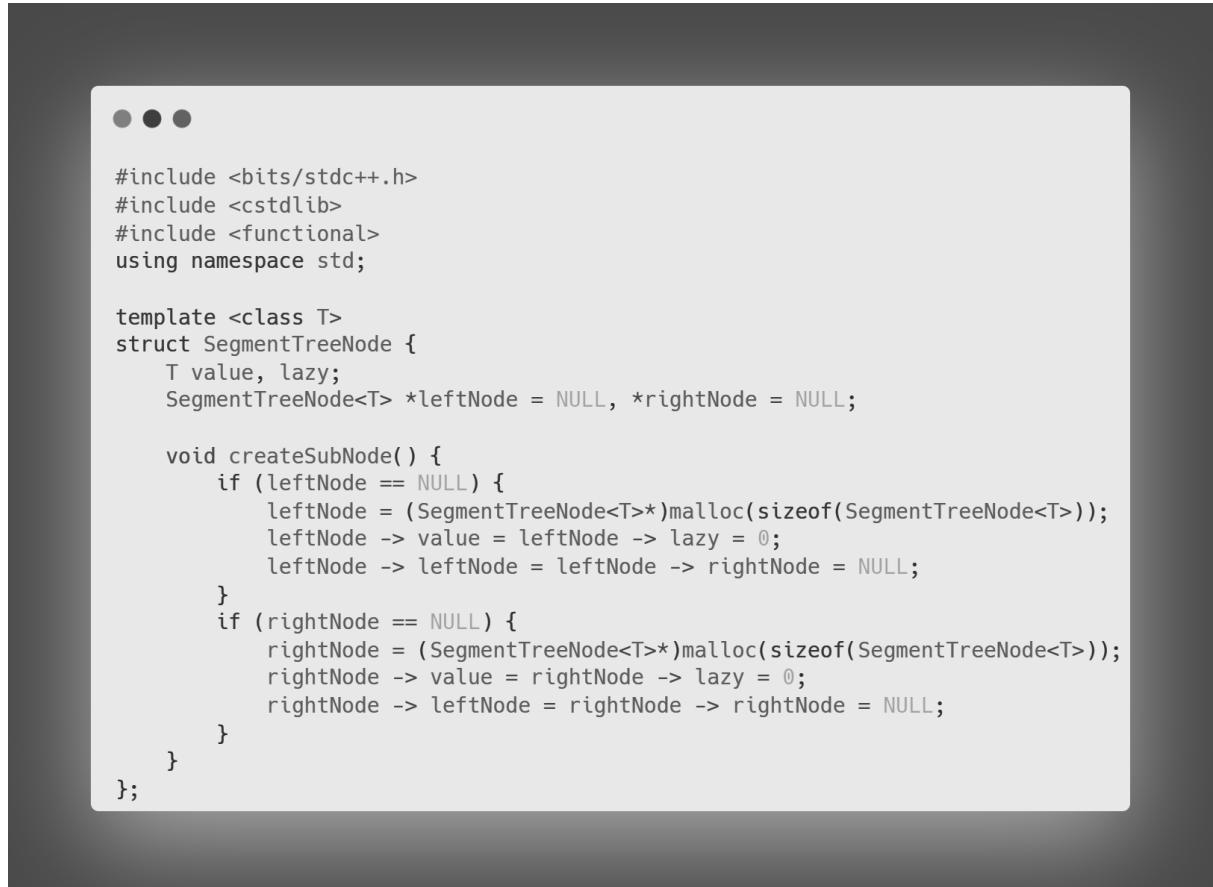
- Truy vấn loại 1: Gán giá trị phần tử ở vị trí u bằng k
- Truy vấn loại 2: Tìm giá trị nhỏ nhất trong khoảng $[a, b]$

Giới hạn bài toán: $1 \leq n, q \leq 2 * 10^5$

3.3. Cài đặt cấu trúc dữ liệu cây phân đoạn

Vì một nút của cây phân đoạn sẽ chứa rất nhiều thông tin, nên ta cần tạo một

struct để lưu trữ những thông tin đó. Để tiện quản lý và tránh gây xung đột code, nên tách hẳn struct này vào một file header riêng. Code trong file segmentTreeNode.h:



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

template <class T>
struct SegmentTreeNode {
    T value, lazy;
    SegmentTreeNode<T> *leftNode = NULL, *rightNode = NULL;

    void createSubNode() {
        if (leftNode == NULL) {
            leftNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
            leftNode -> value = leftNode -> lazy = 0;
            leftNode -> leftNode = leftNode -> rightNode = NULL;
        }
        if (rightNode == NULL) {
            rightNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
            rightNode -> value = rightNode -> lazy = 0;
            rightNode -> leftNode = rightNode -> rightNode = NULL;
        }
    }
};
```

Trong đó: T là kiểu dữ liệu của giá trị lưu trong một nút của cây phân đoạn, tùy trường hợp thực tế mà có thể là một biến số nguyên (int, long long), hoặc là những kiểu dữ liệu phức tạp hơn như một mảng hay một xâu ký tự (array, string). Thậm chí một nút của cây phân đoạn có thể lưu trữ một cây phân đoạn khác để giải quyết những bài toán phức tạp hơn như cây phân đoạn 2 chiều (Segment Tree 2D).

- Biến value: là giá trị được lưu trong một nút của cây phân đoạn
- Biến lazy: là giá trị của kỹ thuật Lazy Propagation
- Biến leftNode: lưu con trỏ dẫn đến nút con bên trái của nút hiện tại, nếu nút hiện tại không có nút con bên trái sẽ mang giá trị NULL.
- Biến rightNode: lưu con trỏ dẫn đến nút con bên phải của nút hiện tại, nếu nút hiện tại không có nút con bên phải sẽ mang giá trị NULL.
- Hàm createSubNode(): tạo 2 nút con cho nút hiện tại

Tương tự một nút trong cây phân đoạn, bản thân cây phân đoạn cũng cần lưu trữ rất nhiều thông tin, nên ta cần tạo một struct để lưu trữ những thông tin đó. Để tiện quản lý và tránh gây xung đột code, nên tách hẳn struct này vào một file header riêng. Code trong file segmentTree.h:

```
struct SegmentTree {
    SegmentTreeNode<T> *root, *defaultNode;
    int head = -1, tail = -1;
    function<void(SegmentTreeNode<T>*, T)> defaultChangeMethod;
    function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)> defaultMergeMethod;

    void createTree() {
        root = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
        root->value = root->lazy = 0;
        root->leftNode = root->rightNode = NULL;
    }

    void createTree(int headValue, int tailValue, int defaultValue,
                   function<void(SegmentTreeNode<T>, T)> change,
                   function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode) {
        createTree();
        head = headValue;
        tail = tailValue;
        defaultNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
        defaultNode->value = defaultValue;
        defaultChangeMethod = change;
        defaultMergeMethod = mergeSubNode;
    }

    void update(int pos, T value) {
        update(root, head, tail, pos, value, defaultChangeMethod, defaultMergeMethod);
    }

    void update(SegmentTreeNode<T> *currNode, int l, int r, int pos, T value,
               function<void(SegmentTreeNode<T>*, T)> change, function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode) {
        if (l > pos || r < pos) return;
        if (l == r) {
            change(currNode, value);
            return;
        }
        currNode->createSubNode();

        int mid = (l + r) >> 1;
        update(currNode->leftNode, l, mid, pos, value, change, mergeSubNode);
        update(currNode->rightNode, mid + 1, r, pos, value, change, mergeSubNode);
        mergeSubNode(currNode, currNode->leftNode, currNode->rightNode);
    }

    SegmentTreeNode<T>* get(int u, int v) {
        return get(root, head, tail, u, v, defaultNode, defaultMergeMethod);
    }

    SegmentTreeNode<T>* get(SegmentTreeNode<T> *currNode, int l, int r, int u, int v, SegmentTreeNode<T>* defaultValue,
                           function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode) {
        if (l > v || r < u) return defaultValue;
        if (u <= l && r <= v) return currNode;

        int mid = (l + r) >> 1;
        currNode->createSubNode();
        SegmentTreeNode<T>* result = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
        mergeSubNode(
            result,
            get(currNode->leftNode, l, mid, u, v, defaultValue, mergeSubNode),
            get(currNode->rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode)
        );
        return result;
    }
};
```

Trong đó:

- Biến root: chứa con trỏ dẫn đến nút gốc của cây phân đoạn
- Biến defaultNode: nút "rỗng". Cụ thể hơn, khi thực hiện truy vấn "get", nếu đoạn đang xét không nằm trong khoảng cần lấy, hàm "get" sẽ về nút "rỗng". Nút "rỗng" phải thỏa mãn điều kiện là khi hợp với kết quả của truy vấn, thì kết quả của truy vấn vẫn không thay đổi.
- Biến head và tail: lưu trữ vị trí đầu tiên và vị trí cuối cùng mà cây phân đoạn đang quản lý.
- Biến hàm defaultChangeMethod: lưu trữ hàm làm thay đổi các nút khi thực hiện truy vấn 1
- Biến hàm defaultMergeMethod: lưu trữ hàm hợp nhất hai nút con trái và phải để tạo nên giá trị của nút cha.
- Hàm createTree(): Hàm tạo nút gốc của cây
- Hàm createTree(...): Hàm vừa tạo nút gốc của cây vừa gán các biến và hàm mặc định sẵn trên cây
- Hàm update(): Thực hiện truy vấn 1
- Hàm get(): Thực hiện truy vấn 2

Để tiện cho người sử dụng thư viện, những hàm change(hàm biến đổi các nút khi thực hiện truy vấn 1) và hàm merge(hàm hợp nhất hai nút con trái phải để tạo nên giá trị của nút cha) đơn giản được viết trước sẵn. Tập hợp những hàm được viết sẵn sẽ được lưu trong file templateTreeNode.h:

```
#include <bits/stdc++.h>
#include <cstdlib>
#include <functional>
#include "SegmentTreeNode.h"
using namespace std;

template <class T> __lcm(T a, T b) {return a * b / __gcd(a, b);}

template <class T> void change_assign(SegmentTreeNode<T>* node, T value) {
    node->value = value;
}

template <class T> void change_add(SegmentTreeNode<T>* node, T value) {
    node->value += value;
};

template <class T> void merge_sum(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r)
{
    par->value = l->value + r->value;
};

template <class T> void merge_max(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r)
{
    par->value = max(l->value, r->value);
};

template <class T> void merge_min(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r)
{
    par->value = min(l->value, r->value);
};

template <class T> void merge_gcd(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r)
{
    par->value = __gcd(l->value, r->value);
};

template <class T> void merge_lcm(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r)
{
    par->value = __lcm(l->value, r->value);
};

template <class T> void merge_or(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r) {
    par->value = l->value | r->value;
};

template <class T> void merge_and(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l, SegmentTreeNode<T>* r) {
    par->value = l->value & r->value;
};
```

Trong đó:

- Hàm `change_assign`: sử dụng khi truy vấn 1 yêu cầu gán giá trị phần tử ở vị trí u là k
- Hàm `change_add`: sử dụng khi truy vấn 1 yêu cầu tăng phần tử ở vị trí u lên k đơn vị
- Hàm `merge_sum`: sử dụng khi truy vấn 2 yêu cầu tính tổng của các số của mảng cho trước trong đoạn $[u, v]$
- Hàm `merge_max`: sử dụng khi truy vấn 2 yêu cầu tính số lớn nhất của các số

của mảng cho trước trong đoạn $[u, v]$

- Hàm merge_min: sử dụng khi truy vấn 2 yêu cầu tính số bé nhất của các số của mảng cho trước trong đoạn $[u, v]$
- Hàm merge_gcd: sử dụng khi truy vấn 2 yêu cầu tính ước chung lớn nhất của các số của mảng cho trước trong đoạn $[u, v]$
- Hàm merge_lcm: sử dụng khi truy vấn 2 yêu cầu tính bội chung nhỏ nhất của các số của mảng cho trước trong đoạn $[u, v]$
- Hàm merge_or: sử dụng khi truy vấn 2 yêu cầu tính phép bit or của các số của mảng cho trước trong đoạn $[u, v]$
- Hàm merge_and: sử dụng khi truy vấn 2 yêu cầu tính phép bit and của các số của mảng cho trước trong đoạn $[u, v]$

3.4. Phát biểu bài toán mở rộng (Ứng dụng kỹ thuật Lazy Propagation)

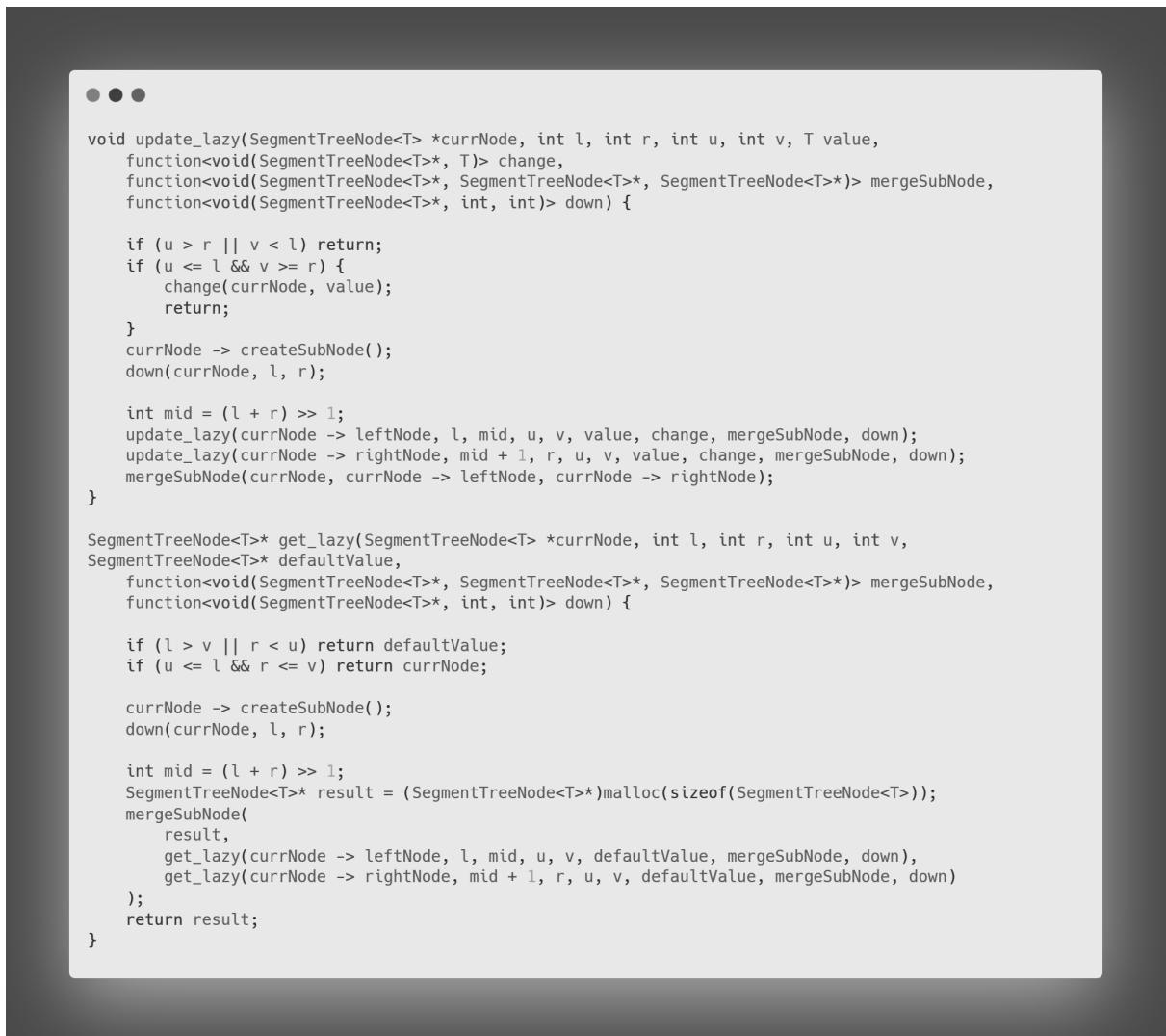
Cho 1 dãy số có n số nguyên $x[]$, cần xử lý q truy vấn gồm 2 loại truy vấn sau:

- Truy vấn loại 1: Tăng giá trị phần tử từ vị trí u đến vị trí v lên k đơn vị
- Truy vấn loại 2: Tìm giá trị nhỏ nhất trong khoảng $[a, b]$

Giới hạn bài toán: $1 \leq n, q \leq 2 * 10^5$

3.5. Cài đặt kỹ thuật Lazy Propagation

Về cách cài đặt kỹ thuật Lazy Propagation, chúng ta chỉ cần viết thêm hai hàm update_lazy và get_lazy dựa theo cơ sở lý thuyết ở phần 2.2. Code phần kỹ thuật Lazy Propagation trong file segmentTree.h:



```
void update_lazy(SegmentTreeNode<T> *currNode, int l, int r, int u, int v, T value,
    function<void(SegmentTreeNode<T>*, T)> change,
    function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode,
    function<void(SegmentTreeNode<T>, int, int)> down) {

    if (u > r || v < l) return;
    if (u <= l && v >= r) {
        change(currNode, value);
        return;
    }
    currNode -> createSubNode();
    down(currNode, l, r);

    int mid = (l + r) >> 1;
    update_lazy(currNode -> leftNode, l, mid, u, v, value, change, mergeSubNode, down);
    update_lazy(currNode -> rightNode, mid + 1, r, u, v, value, change, mergeSubNode, down);
    mergeSubNode(currNode, currNode -> leftNode, currNode -> rightNode);
}

SegmentTreeNode<T>* get_lazy(SegmentTreeNode<T> *currNode, int l, int r, int u, int v,
    SegmentTreeNode<T>* defaultValue,
    function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode,
    function<void(SegmentTreeNode<T>, int, int)> down) {

    if (l > v || r < u) return defaultValue;
    if (u <= l && r <= v) return currNode;

    currNode -> createSubNode();
    down(currNode, l, r);

    int mid = (l + r) >> 1;
    SegmentTreeNode<T>* result = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
    mergeSubNode(
        result,
        get_lazy(currNode -> leftNode, l, mid, u, v, defaultValue, mergeSubNode, down),
        get_lazy(currNode -> rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode, down)
    );
    return result;
}
```

4. CHƯƠNG TRÌNH VÀ KẾT QUẢ

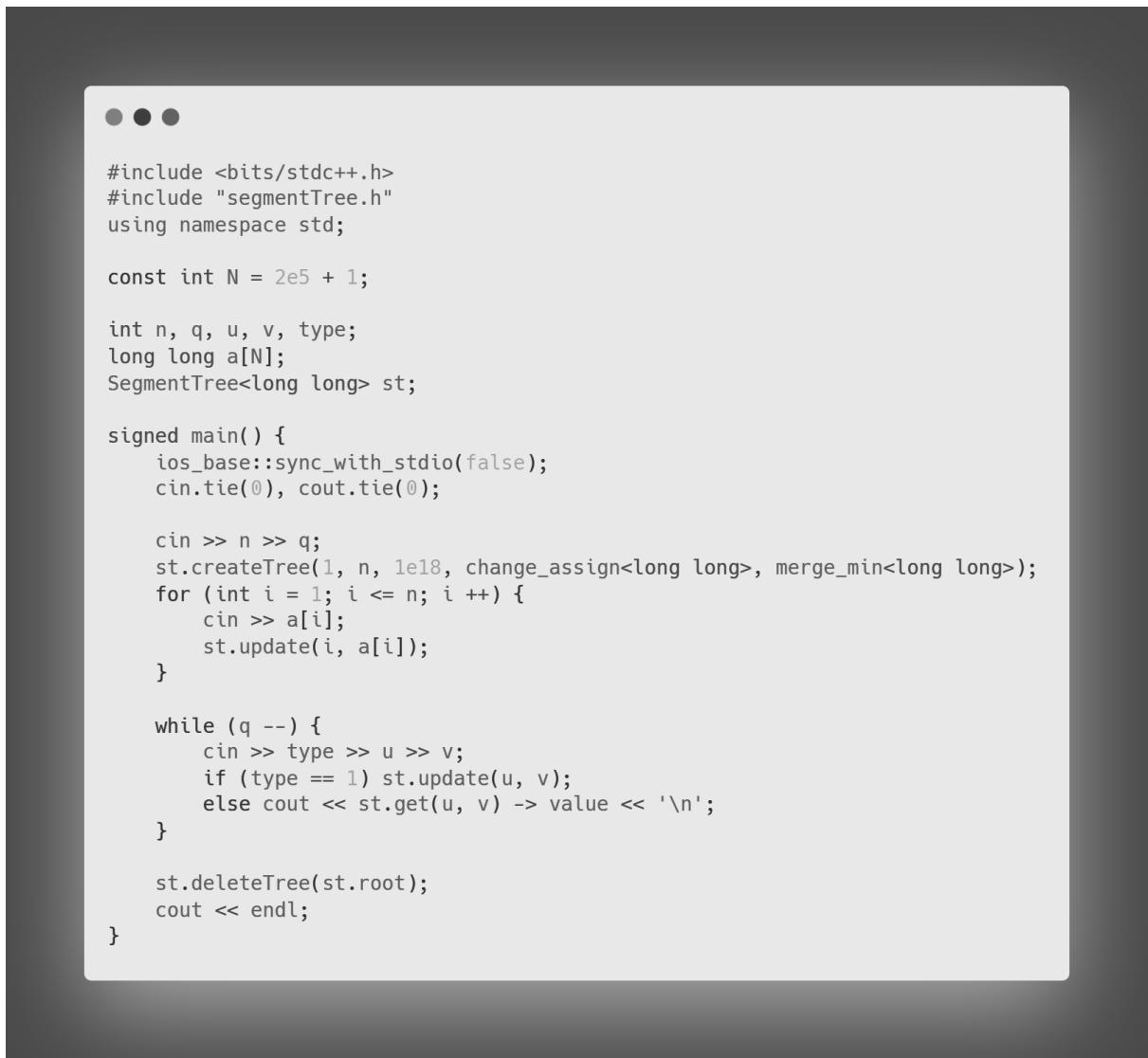
4.1. Ngôn ngữ cài đặt (C++ 11)

4.2. Tính đơn giản, dễ sử dụng của thư viện

Với những bài toán đơn giản, cũng là những bài toán thường gặp nhất khi cần sử dụng tới cây phân đoạn, việc cài đặt cây phân đoạn khi sử dụng thư viện là vô cùng đơn giản. Giúp mã nguồn gọn nhẹ, trực quan và dễ dàng cho việc sửa lỗi hơn.

Một vài ví dụ của việc sử dụng thư viện cho những bài toán đơn giản:

Ví dụ 1: Bài toán ở mục 3.2



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

const int N = 2e5 + 1;

int n, q, u, v, type;
long long a[N];
SegmentTree<long long> st;

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);

    cin >> n >> q;
    st.createTree(1, n, 1e18, change_assign<long long>, merge_min<long long>);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        st.update(i, a[i]);
    }

    while (q--) {
        cin >> type >> u >> v;
        if (type == 1) st.update(u, v);
        else cout << st.get(u, v) -> value << '\n';
    }

    st.deleteTree(st.root);
    cout << endl;
}
```

Ví dụ 2: Bài toán ở mục 3.2 nhưng thay đổi lại yêu cầu trong các loại truy vấn:

- Truy vấn 1: Tăng giá trị phần tử ở vị trí u lên k đơn vị
- Truy vấn 2: Tìm ước chung lớn nhất của các số trong mảng trong đoạn $[a, b]$

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

const int N = 2e5 + 1;

int n, q, u, v, type;
long long a[N];
SegmentTree<long long> st;

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);

    cin >> n >> q;
    st.createTree(1, n, 0, change_add<long long>, merge_gcd<long long>);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        st.update(i, a[i]);
    }

    while (q--) {
        cin >> type >> u >> v;
        if (type == 1) st.update(u, v);
        else cout << st.get(u, v) -> value << '\n';
    }

    st.deleteTree(st.root);
    cout << endl;
}
```

4.3 Tính tùy biến cao của thư viện

Với những bài toán phức tạp hơn, như những bài toán sử dụng kỹ thuật Lazy Propagation, là lúc cây phân đoạn cần phải tùy biến nhiều hơn để có thể giải quyết được. Thư viện cho phép hỗ trợ những tùy biến sau:

- Tùy biến về kiểu dữ liệu trong giá trị của các nút con. Cho phép lưu trữ những thông tin phức tạp hơn, thay vì chỉ là những biến số nguyên. Những thông tin đó có thể là: một xâu, một mảng, một cây nhị phân khác,... nằm trong một nút của cây phân đoạn.
- Hàm change(), để tùy biến theo các yêu cầu trong truy vấn 1 của bài toán

- Hàm merge(), để tùy biến theo các yêu cầu trong truy vấn 2 của bài toán
- Hàm down(), để tùy biến theo các yêu cầu trong kỹ thuật Lazy Propagation

Mã ví dụ của bài toán mục 3.4 khi tùy biến các hàm trong thư viện:

```
● ● ●

#include <bits/stdc++.h>
#include "segmentTree.h"
using namespace std;

const int N = 2e5 + 1;

int n, q, u, v, type;
long long a[N];
SegmentTree<long long> st;

void changeNode(SegmentTreeNode<long long>* node, long long value) {
    node->value += value;
    node->lazy += value;
}

void mergeSubNode(SegmentTreeNode<long long>* par, SegmentTreeNode<long long>* l, SegmentTreeNode<long long>* r) {
    par->value = max(l->value, r->value);
}

void down(SegmentTreeNode<long long>* currNode, int l, int r) {
    if (currNode->lazy == 0) return;

    currNode->leftNode->value += currNode->lazy;
    currNode->rightNode->value += currNode->lazy;
    currNode->leftNode->lazy += currNode->lazy;
    currNode->rightNode->lazy += currNode->lazy;

    currNode->lazy = 0;
}

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);

    cin >> n >> q;
    st.createTree();
    while (q--) {
        cin >> type;
        if (type == 0) {
            cin >> u >> v >> val;
            st.update_lazy(st.root, l, n, u, v, val, changeNode, mergeSubNode, down);
        } else {
            cin >> u >> v;
            cout << st.get_lazy(st.root, l, n, u, v, defaultValue, mergeSubNode, down)->value << '\n';
        }
    }

    st.deleteTree();
    cout << endl;
}
```

4.4 Sử dụng thư viện để giải quyết các bài toán trên trang cses

2 bài toán sử dụng để kiểm thử thư viện là bài Dynamic Range Sum Queries và Range Update Queries trên trang cses. Nếu đối với bài Dynamic Range Sum Queries là một bài toán ứng dụng cơ bản của cây phân đoạn thì bài Range Update Queries là

bài toán cần áp dụng kỹ thuật Lazy Propagation. Bởi vì trang cses chỉ cho nộp bài bằng một file duy nhất nên dành phải gộp các mã ở các file header vào file main.cpp. Mặc dù mã sẽ dài và rối hơn nhưng về cách hoạt động, bản chất là không thay đổi.

Kết quả cho 2 lần kiểm thử:

Range Update Queries					
TASK SUBMIT RESULTS STATISTICS HACKING					
Number of submissions: 1					
< 1 >					
time	lang	code time	code size	result	
	C++	0.81 s	4670 ch.	✓	»

Dynamic Range Sum Queries					
TASK SUBMIT RESULTS STATISTICS HACKING					
Number of submissions: 2					
< 1 >					
time	lang	code time	code size	result	
	C++	0.67 s	2452 ch.	✓	»
	C++	--	2453 ch.	✗	»

Vì bộ test trên cses rất lớn, cả độ dài mảng và số lượng truy vấn đều có thể lên đến 2×10^5 . Nên ta có thể biết được thư viện đã cho ra kết quả chính xác với độ phức tạp về thời gian thực thi được đảm bảo.

5. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1. Kết luận

Đã hoàn thành mục tiêu tạo thư viện cây phân tập có tính sử dụng cao. Thư viện phù hợp cho cả những lập trình viên có thể sử dụng để giải quyết những bài toán

đơn giản đến phức tạp đòi hỏi phải có sự tùy biến cao trong cây phân đoạn.

Cây phân đoạn sử dụng trong thư viện được cài đặt bằng con trỏ, phù hợp với việc sử dụng thực tế.

Kiểm thử thư viện thành công khi đạt kết quả ACCEPTED cho những bài toán trên trang cses.

5.2. Hướng phát triển

Mặc dù đã thành công trong việc xây dựng thư viện nhưng vẫn tồn tại nhiều điểm có thể cải tiến:

- Khử đệ quy để có thời gian thực thi và bộ nhớ sử dụng tốt hơn
- Thêm nhiều hàm tùy biến sẵn cho thư viện để phù hợp hơn với các nhu cầu sử dụng khác nhau
- Phát triển một tài liệu hướng dẫn sử dụng thư viện cụ thể
- Thêm tính năng quản lý bộ nhớ mà cây phân tập hiện tại đang sử dụng và số nút đang có trên cây
- Sửa lại mã nguồn sạch và chuyên nghiệp hơn, thêm phần hướng dẫn trong code (comment)

TÀI LIỆU THAM KHẢO

https://cp-algorithms.com/data_structures/segment_tree.html

https://en.wikipedia.org/wiki/Segment_tree

<https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/>

<https://codeforces.com/blog/entry/106923>

<https://codeforces.com/blog/entry/15890>

https://www.w3schools.com/cpp/cpp_pointers.asp

PHỤ LỤC

Link code github: <https://github.com/Canuc80k/PBL1>

File: segmentTreeNode.h

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

template <class T>
struct SegmentTreeNode {
    T value, lazy;
    SegmentTreeNode<T> *leftNode = NULL, *rightNode = NULL;

    void createSubNode() {
        if (leftNode == NULL) {
            leftNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
            leftNode -> value = leftNode -> lazy = 0;
            leftNode -> leftNode = leftNode -> rightNode = NULL;
        }
        if (rightNode == NULL) {
            rightNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
            rightNode -> value = rightNode -> lazy = 0;
            rightNode -> leftNode = rightNode -> rightNode = NULL;
        }
    }
};
```

File: templateGeneralFunction.h

```
#include <bits/stdc++.h>
#include <cstdlib>
#include <functional>
#include "SegmentTreeNode.h"
using namespace std;

template <class T> __lcm(T a, T b) {return a * b / __gcd(a, b);}

template <class T> void change_assign(SegmentTreeNode<T>* node, T value) {
    node -> value = value;
}
```

```
template <class T> void change_add(SegmentTreeNode<T>* node, T value) {
    node->value += value;
}

template <class T> void merge_sum(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = l->value + r->value;
}

template <class T> void merge_max(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = max(l->value, r->value);
}

template <class T> void merge_min(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = min(l->value, r->value);
}

template <class T> void merge_gcd(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = __gcd(l->value, r->value);
}

template <class T> void merge_lcm(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = __lcm(l->value, r->value);
}

template <class T> void merge_or(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = l->value | r->value;
}

template <class T> void merge_and(SegmentTreeNode<T>* par, SegmentTreeNode<T>* l,
SegmentTreeNode<T>* r) {
    par->value = l->value & r->value;
}
```

File segmentTree.h

```
#include <bits/stdc++.h>
#include <cstdlib>
#include <functional>
#include "templateGeneralFunction.h"
using namespace std;

template <class T>
struct SegmentTree {
    SegmentTreeNode<T> *root, *defaultNode;
    int head = -1, tail = -1;
    function<void(SegmentTreeNode<T>*, T)> defaultChangeMethod;
    function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
    defaultMergeMethod;

    void createTree() {
        root = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
        root->value = root->lazy = 0;
        root->leftNode = root->rightNode = NULL;
    }

    void createTree(int headValue, int tailValue, int defaultValue,
                  function<void(SegmentTreeNode<T>*, T)> change,
                  function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
                  mergeSubNode) {
        createTree();
        head = headValue;
        tail = tailValue;
        defaultNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
        defaultNode->value = defaultValue;
        defaultChangeMethod = change;
        defaultMergeMethod = mergeSubNode;
    }

    void update(int pos, T value) {
        update(root, head, tail, pos, value, defaultChangeMethod, defaultMergeMethod);
    }

    void update(SegmentTreeNode<T> *currNode, int l, int r, int pos, T value,
               function<void(SegmentTreeNode<T>*, T)> change, function<void(SegmentTreeNode<T>*,
               SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode) {
```

```
if(l > pos || r < pos) return;
if(l == r) {
    change(currNode, value);
    return;
}
currNode -> createSubNode();

int mid = (l + r) >> 1;
update(currNode -> leftNode, l, mid, pos, value, change, mergeSubNode);
update(currNode -> rightNode, mid + 1, r, pos, value, change, mergeSubNode);
mergeSubNode(currNode, currNode -> leftNode, currNode -> rightNode);
}

void update_lazy(SegmentTreeNode<T> *currNode, int l, int r, int u, int v, T value,
function<void(SegmentTreeNode<T>*, T)> change,
function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
mergeSubNode,
function<void(SegmentTreeNode<T>*, int, int)> down) {

if(u > r || v < l) return;
if(u <= l && v >= r) {
    change(currNode, value);
    return;
}
currNode -> createSubNode();
down(currNode, l, r);

int mid = (l + r) >> 1;
update_lazy(currNode -> leftNode, l, mid, u, v, value, change, mergeSubNode, down);
update_lazy(currNode -> rightNode, mid + 1, r, u, v, value, change, mergeSubNode, down);
mergeSubNode(currNode, currNode -> leftNode, currNode -> rightNode);
}

SegmentTreeNode<T>* get(int u, int v) {
    return get(root, head, tail, u, v, defaultNode, defaultMergeMethod);
}

SegmentTreeNode<T>* get(SegmentTreeNode<T> *currNode, int l, int r, int u, int v,
SegmentTreeNode<T>* defaultValue,
function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
mergeSubNode) {
```

```
if (l > v || r < u) return defaultValue;
if (u <= l && r <= v) return currNode;

int mid = (l + r) >> 1;
currNode -> createSubNode();
SegmentTreeNode<T>* result = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
mergeSubNode(
    result,
    get(currNode -> leftNode, l, mid, u, v, defaultValue, mergeSubNode),
    get(currNode -> rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode)
);
return result;
}

SegmentTreeNode<T>* get_lazy(SegmentTreeNode<T> *currNode, int l, int r, int u, int v,
SegmentTreeNode<T>* defaultValue,
function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
mergeSubNode,
function<void(SegmentTreeNode<T>*, int, int)> down) {

if (l > v || r < u) return defaultValue;
if (u <= l && r <= v) return currNode;

currNode -> createSubNode();
down(currNode, l, r);

int mid = (l + r) >> 1;
SegmentTreeNode<T>* result = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
mergeSubNode(
    result,
    get_lazy(currNode -> leftNode, l, mid, u, v, defaultValue, mergeSubNode, down),
    get_lazy(currNode -> rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode, down)
);
return result;
}

void deleteTree(SegmentTreeNode<T> *currNode) {
    if (currNode == NULL) return;

    deleteTree(currNode -> leftNode);
    deleteTree(currNode -> rightNode);
```

```
    free(currNode);
}
};
```

File code AC bài Dynamic Range Sum Queries trên cses:

```
#include <bits/stdc++.h>
#include <cstdlib>
#include <functional>
// #include "segmentTree.h"
using namespace std;

struct SegmentTreeNode {
    long long value, defaultValue = 0;
    SegmentTreeNode *leftNode = NULL, *rightNode = NULL;

    void createSubNode() {
        if (leftNode == NULL) {
            leftNode = (SegmentTreeNode*)malloc(sizeof(SegmentTreeNode));
            leftNode -> value = defaultValue;
            leftNode -> leftNode = leftNode -> rightNode = NULL;
        }
        if (rightNode == NULL) {
            rightNode = (SegmentTreeNode*)malloc(sizeof(SegmentTreeNode));
            rightNode -> value = defaultValue;
            rightNode -> leftNode = rightNode -> rightNode = NULL;
        }
    }
};

struct SegmentTree {
    SegmentTreeNode *root;
    long long activeNode = 0;

    void createTree() {
        root = (SegmentTreeNode*)malloc(sizeof(SegmentTreeNode));
        root -> value = 0;
        root -> leftNode = root -> rightNode = NULL;
    }

    void update(SegmentTreeNode *currNode, long long l, long long r, long long pos, long long value,
               function<void(SegmentTreeNode*, long long)> change, function<void(SegmentTreeNode*,
```

```
SegmentTreeNode*, SegmentTreeNode*)> mergeSubNode) {

    if(l > pos || r < pos) return;
    if(l == r) {
        change(currNode, value);
        return;
    }
    currNode -> createSubNode();

    long long mid = (l + r) >> 1;
    update(currNode -> leftNode, l, mid, pos, value, change, mergeSubNode);
    update(currNode -> rightNode, mid + 1, r, pos, value, change, mergeSubNode);
    mergeSubNode(currNode, currNode -> leftNode, currNode -> rightNode);
}

SegmentTreeNode* get(SegmentTreeNode *currNode, long long l, long long r, long long u, long long v,
SegmentTreeNode* defaultValue,
    function<void(SegmentTreeNode*, SegmentTreeNode*, SegmentTreeNode*)> mergeSubNode) {

    if(l > v || r < u) return defaultValue;
    if(u <= l && r <= v) return currNode;

    long long mid = (l + r) >> 1;
    currNode -> createSubNode();
    SegmentTreeNode* result = (SegmentTreeNode*)malloc(sizeof(SegmentTreeNode));
    mergeSubNode(
        result,
        get(currNode -> leftNode, l, mid, u, v, defaultValue, mergeSubNode),
        get(currNode -> rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode)
    );
    return result;
}
};

const long long N = 1e6 + 1;

long long n, q, type, u, v;
long long a[N];
SegmentTree st;

void changeNode(SegmentTreeNode* node, long long value) {
    node -> value = value;
```

}

```
void mergeSubNode(SegmentTreeNode* par, SegmentTreeNode* l, SegmentTreeNode* r) {
    par -> value = l -> value + r -> value;
}

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    st.createTree();
    SegmentTreeNode *defaultValue = (SegmentTreeNode*)malloc(sizeof(defaultValue));
    defaultValue -> value = 0;

    cin >> n >> q;
    for (long long i = 1; i <= n; i++) {
        cin >> a[i];
        st.update(st.root, 1, n, i, a[i], changeNode, mergeSubNode);
    }

    while (q--) {
        cin >> type >> u >> v;
        if (type == 1) st.update(st.root, 1, n, u, v, changeNode, mergeSubNode);
        else cout << st.get(st.root, 1, n, u, v, defaultValue) -> value << '\n';
    }
}
```

File: code AC bài Range Update Queries trên trang cses

```
#include <bits/stdc++.h>
#include <cstdlib>
#include <functional>
// #include "segmentTree.h"
using namespace std;

template <class T>
struct SegmentTreeNode {
    T value, lazy;
    SegmentTreeNode<T> *leftNode = NULL, *rightNode = NULL;

    void createSubNode() {
        if (leftNode == NULL) {
            leftNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
            leftNode -> value = leftNode -> lazy = 0;
```

```
    leftNode -> leftNode = leftNode -> rightNode = NULL;
}
if (rightNode == NULL) {
    rightNode = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
    rightNode -> value = rightNode -> lazy = 0;
    rightNode -> leftNode = rightNode -> rightNode = NULL;
}
}
};

template <class T>
struct SegmentTree {
    SegmentTreeNode<T> *root;
    int activeNode = 0, head = -1, tail = -1;
    function<void(SegmentTreeNode<T>*, T)> defaultChangeMethod;
    function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
defaultMergeMethod;

void createTree() {
    root = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
    root -> value = root -> lazy = 0;
    root -> leftNode = root -> rightNode = NULL;
}

void createTree(int headValue, int tailValue,
    function<void(SegmentTreeNode<T>*, T)> change, function<void(SegmentTreeNode<T>*,
SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode) {

    createTree();
    head = headValue;
    tail = tailValue;
    defaultChangeMethod = change;
    defaultMergeMethod = mergeSubNode;
}

void update(int pos, T value) {
    update(root, head, tail, pos, value, defaultChangeMethod, defaultMergeMethod);
}

void update(SegmentTreeNode<T> *currNode, int l, int r, int pos, T value,
    function<void(SegmentTreeNode<T>*, T)> change, function<void(SegmentTreeNode<T>*,
SegmentTreeNode<T>*, SegmentTreeNode<T>*)> mergeSubNode) {
```

```
if(l > pos || r < pos) return;
if(l == r) {
    change(currNode, value);
    return;
}
currNode -> createSubNode();

int mid = (l + r) >> 1;
update(currNode -> leftNode, l, mid, pos, value, change, mergeSubNode);
update(currNode -> rightNode, mid + 1, r, pos, value, change, mergeSubNode);
mergeSubNode(currNode, currNode -> leftNode, currNode -> rightNode);
}

void update_lazy(SegmentTreeNode<T> *currNode, int l, int r, int u, int v, T value,
function<void(SegmentTreeNode<T>*, T)> change,
function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
mergeSubNode,
function<void(SegmentTreeNode<T>*, int, int)> down) {

if(u > r || v < l) return;
if(u <= l && v >= r) {
    change(currNode, value);
    return;
}
currNode -> createSubNode();
down(currNode, l, r);

int mid = (l + r) >> 1;
update_lazy(currNode -> leftNode, l, mid, u, v, value, change, mergeSubNode, down);
update_lazy(currNode -> rightNode, mid + 1, r, u, v, value, change, mergeSubNode, down);
mergeSubNode(currNode, currNode -> leftNode, currNode -> rightNode);
}

SegmentTreeNode<T>* get(SegmentTreeNode<T> *currNode, int l, int r, int u, int v,
SegmentTreeNode<T>* defaultValue,
function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
mergeSubNode) {

if(l > v || r < u) return defaultValue;
if(u <= l && r <= v) return currNode;
```

```
int mid = (l + r) >> 1;
currNode -> createSubNode();
SegmentTreeNode<T>* result = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
mergeSubNode(
    result,
    get(currNode -> leftNode, l, mid, u, v, defaultValue, mergeSubNode),
    get(currNode -> rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode)
);
return result;
}

SegmentTreeNode<T>* get_lazy(SegmentTreeNode<T> *currNode, int l, int r, int u, int v,
SegmentTreeNode<T>* defaultValue,
    function<void(SegmentTreeNode<T>*, SegmentTreeNode<T>*, SegmentTreeNode<T>*)>
mergeSubNode,
    function<void(SegmentTreeNode<T>*, int, int)> down) {

    if (l > v || r < u) return defaultValue;
    if (u <= l && r <= v) return currNode;

    currNode -> createSubNode();
    down(currNode, l, r);

    int mid = (l + r) >> 1;
    SegmentTreeNode<T>* result = (SegmentTreeNode<T>*)malloc(sizeof(SegmentTreeNode<T>));
    mergeSubNode(
        result,
        get_lazy(currNode -> leftNode, l, mid, u, v, defaultValue, mergeSubNode, down),
        get_lazy(currNode -> rightNode, mid + 1, r, u, v, defaultValue, mergeSubNode, down)
    );
    return result;
}
};

const long long N = 1e6 + 1;

long long n, q, type, u, v, val;
long long a[N];
SegmentTree<long long> st;

void changeNode(SegmentTreeNode<long long>* node, long long value) {
    node -> value += value;
```

```
    node -> lazy += value;
}

void mergeSubNode(SegmentTreeNode<long long>* par, SegmentTreeNode<long long>* l,
SegmentTreeNode<long long>* r) {
    par -> value = max(l -> value, r -> value);
}

void down(SegmentTreeNode<long long>* currNode, int l, int r) {
    if (currNode -> lazy == 0) return;

    currNode -> leftNode -> value += currNode -> lazy;
    currNode -> rightNode -> value += currNode -> lazy;
    currNode -> leftNode -> lazy += currNode -> lazy;
    currNode -> rightNode -> lazy += currNode -> lazy;

    currNode -> lazy = 0;
}
signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    st.createTree();
    SegmentTreeNode<long long> *defaultValue = (SegmentTreeNode<long
long>*)malloc(sizeof(defaultValue));
    defaultValue -> value = -1e18;
    cin >> n >> q;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        st.update_lazy(st.root, 1, n, i, i, a[i], changeNode, mergeSubNode, down);
    }
    while (q--) {
        cin >> type;
        if (type == 1) {
            cin >> u >> v >> val;
            st.update_lazy(st.root, 1, n, u, v, val, changeNode, mergeSubNode, down);
        } else {
            cin >> u;
            cout << st.get_lazy(st.root, 1, n, u, u, defaultValue, mergeSubNode, down) -> value << '\n';
        }
    }
}
```