TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA TOÁN - CƠ - TIN HỌC

BÁO CÁO CUỐI KÌ

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

Giảng viên: Nguyễn Thị Hồng Minh

Đề tài:

FENWICK TREE

Thành viên

Bùi Khánh Duy – 20001898

Đỗ Anh Đức – 20001906

Lê Thị Thùy Dung – 20001895

HÀ NỘI - 2022

LỜI CẢM ƠN

Lời đầu tiên, chúng em xin gửi lời cảm ơn sâu sắc đến giảng viên bộ môn - Cô Nguyễn Thị Hồng Minh. Cô đã giảng dạy, truyền đạt, chia sẻ cho chúng em những kiến thức quý báu trong suốt học kì qua. Nhờ sự giúp đỡ của cô mà chúng em được tiếp thu những kiến thức rất hữu ích cả trong và ngoài chuyên ngành.

Vì thời gian làm bài có hạn nên nhóm không tránh khỏi những thiếu sót. Chúng em rất mong nhận được lời nhận xét từ cô để cải thiện hơn đồng thời được bổ sung, nâng cao kiến thức của bản thân.

Chúng em xin chân thành cảm ơn cô vì những kiến thúc bổ ích trong thời gian học tập vừa qua. Kính chúc cô thật nhiều sức khỏe, luôn vui vẻ, hạnh phúc, thành công trong cuộc sống.

Mục lục

I. Fenwick Tree

1. Tổng quan

Fenwick Tree, hay còn gọi là cây chỉ số nhị phân (Binary Indexed Tree - BIT), là một cấu trúc dữ liệu tối ưu cho việc cập nhật giá trị một phần tử và tìm tổng, min/max giữa 2 vị trí bất kì trong mảng. Độ phức tạp cho mỗi lần cập nhật, truy xuất là $O(logN)$ với N là độ dài dãy cần quản lý. BIT được sử dụng trong những bài toán về mảng, truy vấn trên mảng.

2. Bài toán

Cho mảng A gồm N phần tử (đánh số từ 1). Có Q truy vấn thuộc 2 loại:
1 u v : cộng v đơn vị vào A[u]
2 p : Tính tổng các phần từ $A[1], A[2], ...A[p]$. *(Lưu ý mảng bắt đầu từ 1)*
Giới hạn: N, Q $\leq 2 \cdot 10^5$

3. Ý tưởng

3.1 Ngây thơ 1

- Với truy vấn loại 1, chỉ tăng phần tử $A[u]$ thêm v đơn vị.
- Với truy vấn loại 2: Duyệt qua tất cả các phần tử trong đoạn $[1, \ldots, p]$ và cộng giá trị vào biến kết quả

```
public class Naive1 {
    final int N = 100005;
    int a[] = new int[N];
    void update(int u, int v) {
        a[u] += v;
    }
    int getSum(int p) {
        int answer = 0;
        for (int i = 1; i <= p; ++i) {
            answer += a[i];
        }
        return answer;
    }
```

```
}
```

Phân tích:

- Độ phức tạp khi update: $O(1)$
- Độ phức tạp khi truy vấn: $O(p) = O(N)$
- Với Q truy vấn thì độ phức tạp là: $O(Q + Q \cdot N) = O(Q \cdot N)$

3.2 Ngây thơ 2

Áp dụng mảng cộng dồn (prefix sum) để tính nhanh tổng một đoạn. Khi cập nhật 1 phần tử, ta sẽ cập nhật từ đoạn đó trở về sau.

```
public class Naive2 {
    final int N = (int)1e5+5;
    int n; // size of a, n < N
    int a[] = new int[N];
    int sum[] = new int[N];
    void preProcess() {
        sum[1] = a[1];
        for(int i = 2; i <= n; ++i) {
            sum[i] = sum[i-1] + a[i];
        }
    }
    void update(int u, int v) {
        for(int i = u; i <= n; ++i) {
            sum[i] += v;
        }
    }
    int getSum(int p) {
        return sum[p];
    }
}
```

Phân tích:

- Độ phức tạp tiền xử lý là: $O(N)$
- Độ phức tạp khi update: $O(N)$

• Độ phức tạp khi truy vấn: *O(1)*

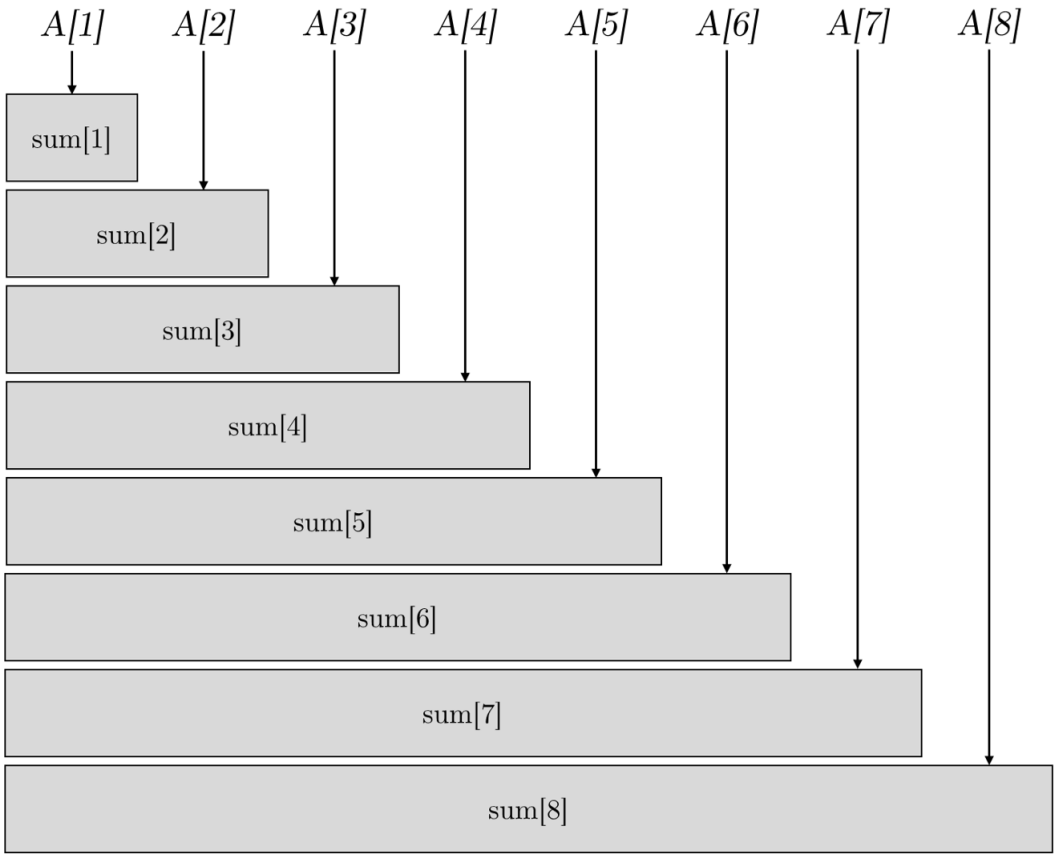
Nếu bài toán không có truy vấn cập nhật thì độ phức tạp sẽ chỉ là $O(N+Q)$ đủ nhanh.

Nhưng khi có thao tác cập nhật, độ phức tạp sẽ trở thành $O(N \cdot Q)$, không khác gì phương pháp 1.

4. Giới thiệu

Một loại cây có tính chất của prefix sum, đồng thời có thể nhanh chóng cập nhật giá trị của prefix.

4.1 Cấu trúc cây prefix sum - Tổng tiền tố



Hình 1: Cây tiền tố

Mỗi phần tử sum[i]$= \sum_1^i$a[i] $\Rightarrow a[j] \in$ sum[i] nếu thỏa mãn j $\leq i$.

Nếu cập nhật $a[j]$ thì số phần tử cần cập nhật sẽ gần như là cả đoạn(như cách ngây thơ 2)

Có cách nào để bố trí lại phần tử hay không?

5. Tổng quát

5.1 Ý tưởng hình thành nên Fenwick Tree

Mỗi số n đều có thể biểu diễn dưới dạng tổng của các luỹ thừa cơ số 2.
$\Rightarrow$ Để tính tổng đoạn $[1,\ldots,n]$ ta có thể tách thành các đoạn nhỏ có độ dài 2^k và cộng lại tổng trên.

$$
\begin{array}{lll}
1 & = 2^0 & = [1...1] \\
2 & = 2^1 & = [1...2] \\
3 & = 2^1 + 2^0 & = [1...2] + [3..3] \\
4 & = 2^2 & = [1...4] \\
5 & = 2^2 + 2^0 & = [1...4] + [5...5] \\
6 & = 2^2 + 2^1 & = [1...4] + [5...6] \\
7 & = 2^2 + 2^1 + 2^0 & = [1...4] + [5...6] + [7...7] \\
8 & = 2^3 & = [1...8]
\end{array}
$$

Hình 2: Ví dụ 8 số đầu biểu diễn thành các đoạn nhỏ

Cụ thể, đặt $n = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_k} (i_1 > i_2 > \ldots > i_k \geq 0)$.

$\Rightarrow$ Để tính tổng từ $[1,\ldots,n]$ ta tính tổng các đoạn

$$[1;2^{i_1}] + [2^{i_1}+1;2^{i_1}+2^{i_2}] + \ldots + [2^{i_1}+2^{i_2}+\ldots+2^{i_k-1}+1;n]$$

Với số n có thể có tối đa $log_2 n$ bits $\Rightarrow$ độ phức tạp là $O(C\ log\ n)$, với $O(C)$ là độ phức tạp khi tính tổng của đoạn.

5.2 Tiểu sử

Fenwick tree (BIT a.k.a Binary Index Tree) được nhắc đến lần đầu trong bài báo có tiêu đề: "A new data structure for cumulative frequency tables"(Peter M. Fenwick, 1994).

5.3 Tên gọi

Thường được gọi là cây BIT (Binary Index Tree). Tuy nhiên, đây không phải là Binary Tree, vì các node được đánh số từ 1 đến N và sử dụng các biểu diễn nhị phân của nó để định nghĩa nên node cha cho các node khác.

6. Các phép toán với bit

Các thao tác trên bit	Kí hiệu
Phép AND	&
Phép OR	|
Phép phủ đinh NOT	~
Phép XOR	^
Phép dịch trái - Shift left	«
Phép dịch phải - Shift right	»

6.1 Phép AND

Kí hiệu: &

Bảng chân trị:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Phép AND chỉ có giá trị là 1 nếu cả hai toán hạng đều có giá trị là 1.

Ví dụ:

```
A           0000 1100
B           0101 0101
C = A & B   0000 0100
```

6.2 Phép OR

Kí hiệu: |

Bảng chân trị:

A	**B**	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Phép OR chỉ có giá trị là 0 nếu cả hai toán hạng đều có giá trị là 0.

Ví dụ:

```
A           0000 1100
B           0101 0101
C = A | B   0101 1101
```

6.3 Phép phủ đinh NOT

Kí hiệu: $\sim$

Bảng chân trị:

A	$\sim A$
0	1
1	0

Phép phủ định NOT đảo bit 1 thành 0 và ngược lại.

Ví dụ:

```
A        0000 1100
B = ~ A   1111 0011
```

6.4 Phép XOR

Kí hiệu: $\wedge$

Bảng chân trị:

A	B	A $\wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Phép XOR chỉ có giá trị 0 nếu cả hai toán hạng có cùng giá trị, cùng là giá trị 1, hay cùng là giá trị 0.

Ví dụ:

```
A           0000 1100
B           0101 0101
C = A ^ B   0101 1001
```

6.5 Phép dịch trái $<<$

Kí hiệu: «

Phép dịch trái n bit tương đương với phép nhân cho 2n.

Ví dụ:

```
A            0000 1100
                  <-
B = A << 2   0011 0000
```

6.6 Phép dịch phải $>>$

Kí hiệu: »

Phép dịch phải n bit tương đương với phép chia cho 2n.

Ví dụ:

```
A            0000 1100
                  ->
B = A >> 2   0000 0011
```

6.7 Giải thích về $i\ \&(-i)$:

Lấy ví dụ cho kiểu 8 bit (để đơn giản hoá cách viết)

$x = 5_{(10)} = 00000101_{(2)}$

$\bar{x} = 11111010_{(2)}$

$x + \bar{x} = 11111111_{(2)} = -1_{(10)}$

$-x = \bar{x} + 1$

Phép toán này được gọi là "phép bù hai", đây cũng chính là cách biểu diễn số âm trong máy tính. Từ ví dụ ở trên, ta có thể hiểu cách các kiểu dữ liệu như 'int' và 'long' trong Java và C++ biểu diễn số âm.

$\bar{x} + 1 = 11111010_{(2)} + 1 = 11111011_{(2)}$

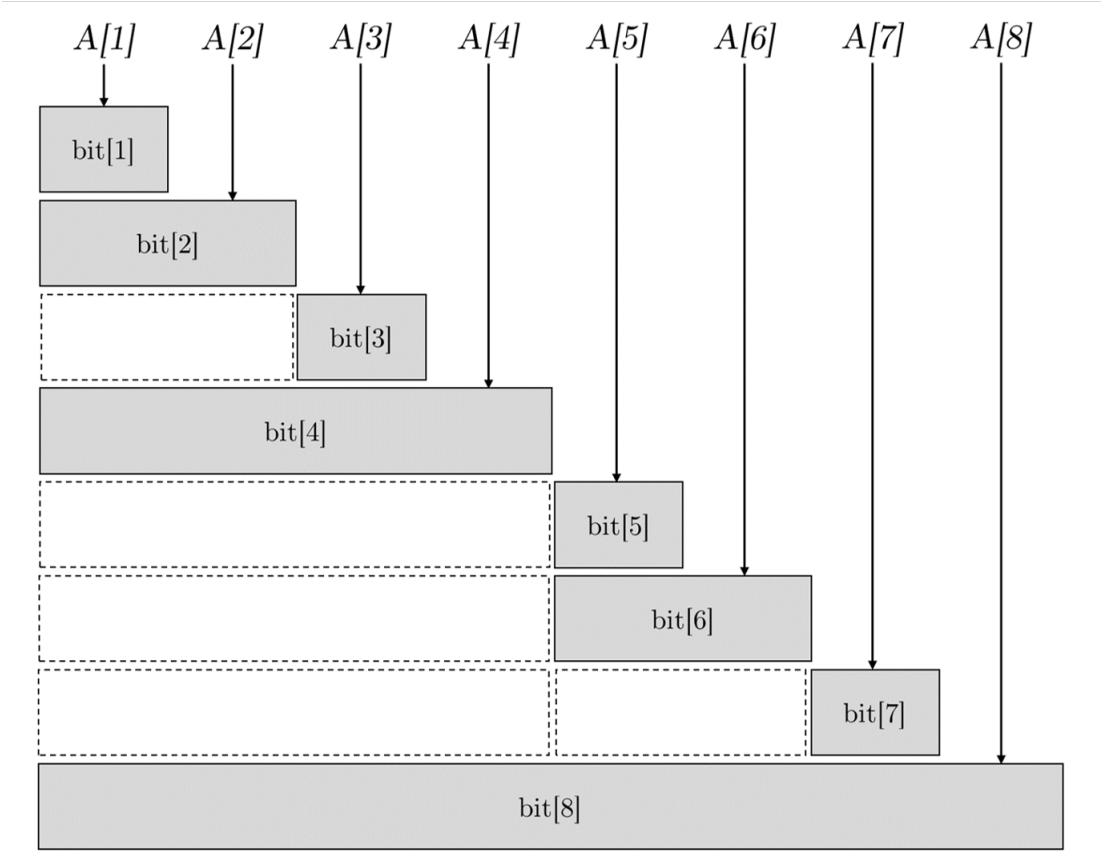
$x\&(-x) = x\&(\bar{x} + 1) \Rightarrow$ sẽ lấy ra được bit ngoài cùng lớn nhất. Thử lấy thêm ví dụ khác:

$x = 14_{(10)} = 00001110_{(2)}$

$-x = -14_{(10)} = 11110010_{(2)}$

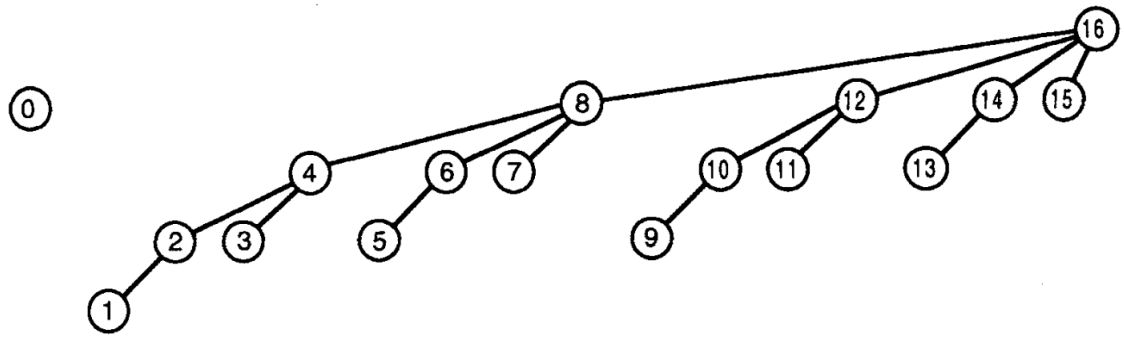
$x\&(-x) = 10_{(2)}$ là vị trí của bit 1 cao nhất trong số

7. Biểu diễn của Fenwick Tree



Hình 3: Minh hoạ Fenwick tree dưới dạng prefix sum

Trong hình trên, những đoạn được tô đậm là đoạn của phần tử chỉ số n được BIT lưu trữ; những đoạn được tô nét mảnh không được lưu trữ trực tiếp mà sẽ được truy cập gián tiếp.



Hình 4: Minh hoạ BIT dưới dạng node và cạnh

Đây là biểu diễn của BIT dưới dạng cây, ở đây ta sẽ phân tích về mối

quan hệ cha - con của cấu trúc dữ liệu này.

Với x là vị trí node hiện tại

7.1 Truy cập code cha

$cha(x) = x + 2^k$ với k là số nhỏ nhất mà x chia hết cho 2^k.

$\Leftrightarrow$ cha(x) $= x + x\&(-x) = 2^k$

Giải thích

2^k là số nhỏ nhất mà x chia hết cho.

Biểu diễn của x dưới dạng nhị phân

$$\begin{array}{r|cccccccc} \text{vị trí} & n & n-1 & \dots & k+1 & k & k-1 & \dots & 0 \\ x = & y_n & y_{n-1} & \dots & y_{k+1} & 1 & 0 & \dots & 0 \end{array}$$

$\forall\, y_j \mid j \in [k+1, n],\ y = 0$ hoặc 1.

- Nếu $y_j = 0 \rightarrow$ chia hết cho 2^k
- Nếu $y_j = 1 \rightarrow 100..0_{(\text{j số 0})} = 100..0_{(\text{k số 0})} << (j-k) = 2^k \cdot 2^{j-k}$

(toán tử $<< n$ là phép dịch sang trái n bit, tương đương với phép $\times 2^n$)

$\rightarrow 2^k$ là số nhỏ nhất mà x chia hết (điều hiển nhiên)

Ví dụ:

$$9_{(10)} = 1001_{(2)} \rightarrow 1010_{(2)} = 10_{(10)} \rightarrow 1100_{(2)} = 12_{(10)} \rightarrow 10000_{(2)} = 16_{(10)}$$

Logic: Khi mình cập nhật 1 đoạn, phải cập nhật tất cả các đoạn chứa nó (ở đây là node cha).

Đây chính là thao tác update cây khi mình thêm 1 giá trị vào

```
    for(int i = u; i < N; i += i & (-i)) {
     bit[i] += v;
}
```

7.2 Truy cập node không liền kề với nó

Để tính tổng các phần tử từ $A[1]...A[p]$, ta lấy giá trị 2^k giống ở trên, nhưng sẽ là loại bỏ đi để đi đến các node cùng bậc:

next(x) $= x - 2^k$ với k là số nhỏ nhất mà x chia hết cho 2^k

$\Leftrightarrow$ next(x) $= x - x\&(-x)$

Giải thích

next(x) là node lớn nhất mà không phải là con của x

Biểu diễn của x dưới dạng nhị phân

vị trí	n	$n-1$	$\ldots$	$k+1$	k	$k-1$	$\ldots$	0
$x =$	y_n	y_{n-1}	$\ldots$	y_{k+1}	1	0	$\ldots$	0

Lấy j bất kì trong đoạn $[k-1, 0]$

Xét: $x_j = x - 2^j$.

vị trí	n	$n-1$	$\ldots$	$k+1$	k	$\ldots$	j	$\ldots$	0
$x_j =$	y_n	y_{n-1}	$\ldots$	y_{k+1}	0	$\ldots$	1	$\ldots$	0

(phần $\ldots$ là viết gọn của các vị trí mang bit 0)

Áp dụng giải thích ở phần Truy cập node cha, ta có: $x_j + x_j\&(-x_j) = x$ $\rightarrow x_j$ là node con của x

Sử dụng phản chứng, ta chứng minh được $x - x\&(-x)$ là node lớn nhất không phải con của x

Từ đây ta có thao tác để lấy tổng các phần tử từ $A[1], \ldots A[p]$

```
int answer = 0;
for(int i = p; i > 0; i -= i & (-i)) {
    answer += bit[i];
}
```

7.3 Cách cài đặt

Cấu trúc dữ liệu này được gọi là cây, bởi vì ta có thể biểu diễn nó dưới dạng cây (như trên hình 4). Tuy nhiên, không nhất thiết phải sử dụng cấu trúc của cây (nốt và cạnh) mà chỉ cần dùng một mảng bình thường 'bit' là đã giải quyết được các vấn đề. (giống với việc dùng mảng cho cấu trúc Binary Search Tree)

Dưới đây là đoạn code cho Fenwick Tree, với mảng được bắt đầu từ 1 (one-based array)

```
public class FenwickTree {
    int n; // size
    final int N = (int) 1e5+5;
    int bit[] = new int[N];
    FenwickTree(int size) {
        n = size;
    }
    FenwickTree(int[] a) {
        n = a.length;
        for(int i = 1; i < n; ++i) {
            bit[i] += a[i];
            int j = i + (i&(-i));
            if (j < n) {
                bit[j] += bit[i];
            }
        }
    }
    void add(int u, int v) {
        for(int i = u; i <= n; i += i & (-i)) {
            bit[i] += v;
        }
    }
    int getSum(int p) {
        int answer = 0;
        for(int i = p; i > 0; i -= i & (-i)) {
            answer += bit[i];
        }
        return answer;
    }
}
```

Ngoài ra, nhóm đã tìm được phương pháp để xây dựng (build) cây BIT với độ phức tạp $O(N)$ khi có mảng cho trước.

8. Cập nhật đoạn

Bài toán

1 v l r : cộng v vào tất cả các phần tử A[l], A[l+1], ..., A[r]

2 u : Tính giá trị hiện tại của A[u]

3 l r : Tính tổng các phần từ từ A[l], A[l+1], ..., A[r]

Phần phía trên đã trình bày về việc cập nhật 1 điểm và các thao tác với nó (truy vấn điểm, truy vấn đoạn). Tuy nhiên đấy chỉ là một phần nhỏ công dụng của cấu trúc dữ liệu này. Mặt khác, xử lý bài toán này chỉ với việc gọi hàm add cho cập nhật điểm như trên thì độ phức tạp là $O(Q \cdot NlogN)$. Đương nhiên là cách này quá chậm, và dưới đây là giải pháp để giữ nguyên độ phức tạp $O(QlogN)$

8.1 Truy vấn điểm

Bài toán; Tăng trên đoạn $[l,r]$ lên v

Dựa trên bản chất của tổng tích luỹ, ta xây dựng nên mảng hiệu (difference array) để lưu hiệu giữa các phần tử liền kề với nhau. Mảng hiệu được xây dựng như sau:

- Với $i = 1$ thì $diff[i] = A[i]$
- Với $2 \leq i \leq N$ thì $diff[i] = A[i] - A[i-1]$

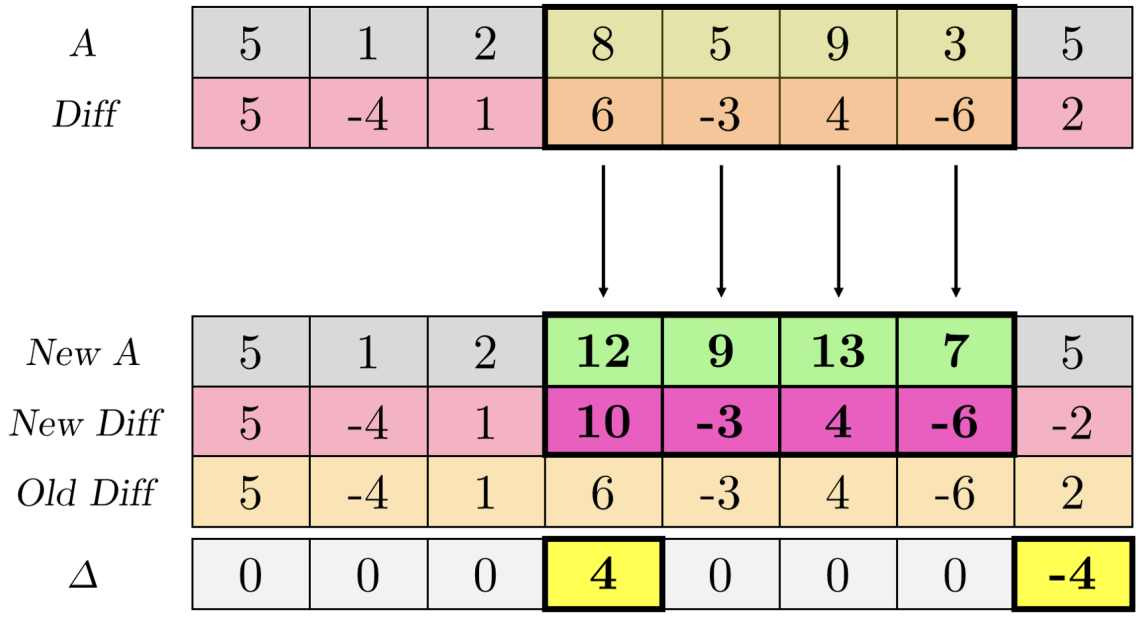
Ví dụ:

	1	2	3	4	5	6	7	8
A	5	1	2	8	5	9	3	5
Diff	5	-4	1	6	-3	4	-6	2

Khi cộng tất cả phần tử $diff$ từ 1 đến i ta có:

$$\begin{aligned}\sum_{j=1}^{i} diff[i] &= \text{diff[1] + diff[2] + ... + diff[i]} \\ &= \text{a[1] + a[2] - a[1] + a[3] - a[2] + ... + a[i] - a[i-1]} \\ &= \text{a[1] - a[1] + a[2] - a[2] + ... + a[i-1] - a[i-1] + a[i]} \\ &= \text{a[i]}\end{aligned}$$

Từ tính chất này, khi cần tính giá trị $A[i]$, ta lấy tổng của i phần tử $diff$ đầu tiên $\rightarrow$ Bài toán trở thành tính tổng trên mảng $diff$.



Hình 5: Minh hoạ thao tác cập nhật trên đoạn $[l,\ldots,r]$ từ mảng hiệu, và cộng $\Delta = 4$ vào đoạn $[4,\ldots,7]$

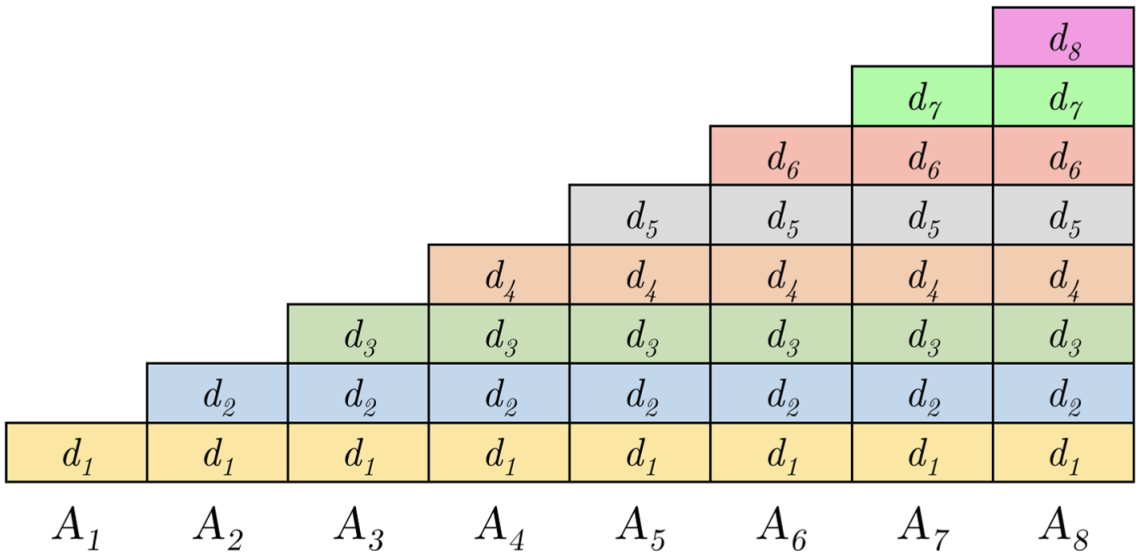
Nguyên lý: Do cả đoạn $[l,\ldots,r]$ đều cộng thêm giá trị Δ (theo bài toán là v) nên hiệu của chúng không đổi. Khác biệt nằm ở 2 biên (a_{l-1}, a_l) và (a_r, a_{r+1}); vì thế ta chỉ cần cập nhật điểm tại 2 biên của hiệu và gọi truy vấn tính tổng để lấy giá trị hiện tại của a_i

Thao tác add ở trên sẽ được xử lý lại như sau (từ giờ sẽ sửa thành update cho đúng với tính chất cập nhật):

```
1       void updatePoint(int u, int v) {
2           for(int i = u; i < n; i += i & -i) {
3               bit[i] += v;
4           }
5       }
```

```
    void updateRange(int l, int r, int v) {
        updatePoint(l, v);
        updatePoint(r + 1, -v);
    }

    int get(int p) {
        int ans = 0;
        for(int i = p; i > 0; i -= i & -i) {
            ans += bit[i];
        }
        return ans;
    }
```

8.2 Truy vấn trên đoạn



Hình 6: Minh hoạ trực quan mối quan hệ về tổng các phần tử với mảng A và mảng hiệu $diff$

Nhắc lại phần 1: $A[i] = \sum_{j=1}^{i} diff[i]$. Dựa vào hình, ta có tính lần lượt tổng các phần tử A_1 đến A_i như sau:

- $sum[1] = diff[1]$
- $sum[2] = 2 \cdot diff[1] + diff[2]$
- $sum[3] = 3 \cdot diff[1] + 2 \cdot diff[2] + diff[3]$

- ...
- $sum[i] = i \cdot diff[1] + (i-1) \cdot diff[2] + \ldots + (i-j+1) \cdot diff[j] + \ldots + 2 \cdot diff[i-1] + diff[i]$

Tuy nhiên, ta thấy hệ số nhân i bị biến động nên không thuận tiện cho việc truy vấn liên tục. Để dễ dàng hơn, ta sẽ cố định $diff[i]$ nhân với hệ số $n-i+1$. Lúc này trở thành

- $sum[1] = n \cdot diff[1] - (n-1) \cdot diff[i]$
- $sum[2] = n \cdot diff[1] + (n-1) \cdot diff[2] - (n-2) \cdot (diff[1] + diff[2])$
- $sum[3] = n \cdot diff[1] + (n-1) \cdot diff[2] + (n-2) \cdot diff[3] - (n-3) \cdot (diff[1] + diff[2] + diff[3])$
- ...
- $sum[i] = n \cdot diff[1] + (n-1) \cdot diff[2] + \ldots + (n-j+1) \cdot diff[j] + \ldots + (n-i+1) \cdot diff[i] - (n-i) \cdot (diff[1] + diff[2] + \ldots + diff[i])$

Tóm lại ta được: $sum[i] = \sum_{j=1}^{i} (n-j+1) \cdot diff[j] - (n-i) \cdot \sum_{j=1}^{i} diff[j]$.

Để đơn giản hoá, ta lưu toàn bộ giá trị $(n-j+1) \cdot diff[i]$ vào mảng S_1 và $diff[j]$ vào mảng S_2.

Ta dựng 2 mảng cộng dồn dựa trên 2 mảng này:

- S_2 thì giống như việc Truy vấn điểm ở phần 1,
- S_1 thì hơi khác ở hệ số nhân $(n-j+1)$. Tuy vậy, hệ số này không đổi trong quá trình tính toán với từng phần tử trên đoạn nên ta chỉ cần nhân hệ số này với giá trị cần cập nhật và áp dụng phương thức tương tự ở cập nhật trên.

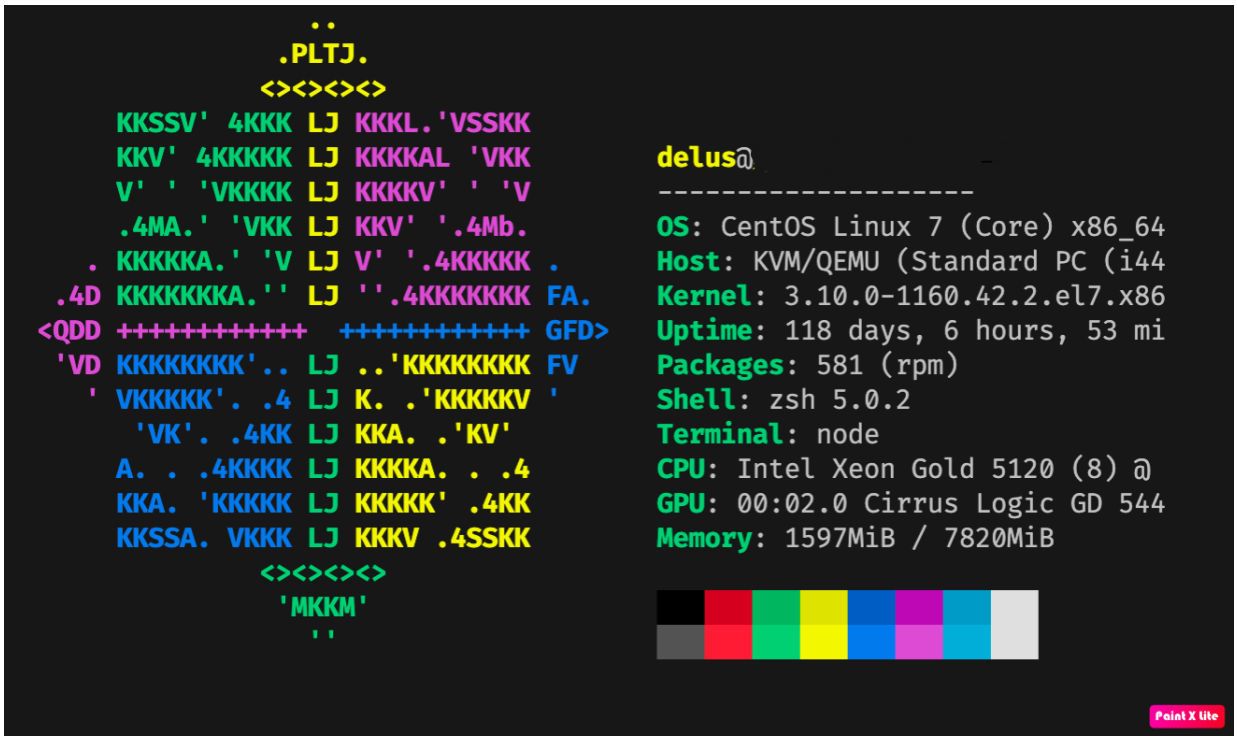
Code:

```
1       void updatePoint(int[] b, int u, int v) {
2           for (int i = u; i < n; i += i & -i) {
3               b[i] += v;
4           }
```

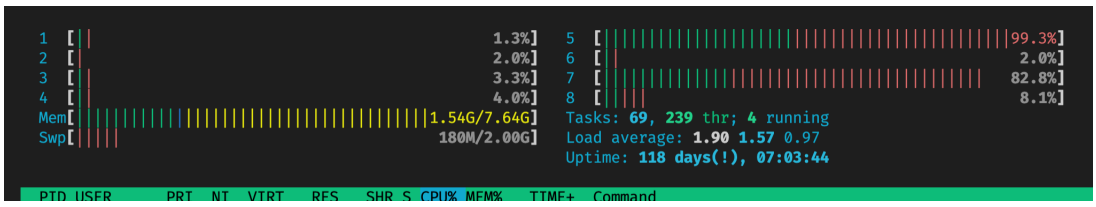
```
    }

    void updateRange(int l, int r, int v) {
        updatePoint(bit1, l, (n - l + 1) * v);
        updatePoint(bit1, r + 1, -(n - r) * v);
        updatePoint(bit2, l, v);
        updatePoint(bit2, r + 1, -v);
    }

    int getSumOnBIT(int[] b, int p) {
        int ans = 0;
        for (int i = p; i > 0; i -= i & -i) {
            ans += bit[i];
        }
        return ans;
    }

    int prefixSum(int u) {
        return getSumOnBIT(bit1, u) - getSumOnBIT(bit2, u) * (n - u);
    }

    int rangeSum(int l, int r) {
        return prefixSum(r) - prefixSum(l - 1);
    }
```

9. Benchmark

9.1 Thông tin máy thực hiện chạy



Hình 7: Cấu hình máy chạy benchmark



Hình 8: Hình ảnh lúc chạy benchmark

9.2 Bảng kết quả

Thông tin về bộ test:

- Có 2 loại test: $N \leq 10^3$ và $N \leq 10^5$
- Có 1000 bộ test cho mỗi loại
- Sử dụng *python* để sinh các bộ test
- Các code để chạy test được viết bằng *C++* vì *Java* chạy quá chậm

	Fenwick Tree	Fast Segment Tree	Slow Segment Tree
1000	364352733	355223024	388960366
100000	453302026	440214056	511036297

Bảng 1: Bảng tổng thời gian chạy, đơn vị μs

9.3 Giải thích

Code Fenwick Tree tương tự như trong phần Cài đặt ở trên.

Code Fast Segment tree được cài đặt với bộ nhớ là $2 \cdot N$ và sử dụng các toán tử bit $(>>, <<)$ để tối ưu hết mức tốc độ chạy.

Code Slow Segment tree sử dụng cách cài đặt cổ điển, tốn $4 \cdot N$ ô nhớ, không dùng toán tử bit để tối ưu.

10. Ưu, nhược điểm

10.1 Ưu điểm

Fenwick Tree là cấu trúc dữ liệu được dựa trên tính chất của prefix sum và cập nhật giá trị chỉ trong $O(logn)$, mặt khác CTDL này dễ dàng cài đặt và không tốn quá nhiều thời gian để code

Sử dụng tính chất của Prefix sum, ta dễ dàng tính được tổng trên đoạn $[l \cdots r]$ thông qua phép *getSum(r) - getSum(l-1)*.

10.2 Nhược điểm

Vì là CTDL dựa trên Prefix sum, nên khi sử dụng với các phép toán không có tính chất đấy là việc không dễ dàng.

Các phép max, min, gcd không tồn tại phép hiệu cho phép ta lấy kết quả của một đoạn như việc tính tổng.

10.3 Kết luận

- Fenwick Tree (BIT) là cấu trúc dữ liệu dễ cài đặt, tốc độ chạy nhanh, tốn ít bộ nhớ, nhưng chỉ sử dụng được cho ít bài toán với các điều kiện chặt chẽ.

- Cần nắm rõ tính chất và các bài toán của của Fenwick Tree để quyết định có nên dùng hay không.
- Tốc độ chạy của Fenwick Tree và Segment tree được tối ưu không thua kém nhau, nhưng Segment tree lại giải quyết được nhiều bài toán hơn

11. Ứng dụng

1. Tính tổng mảng một chiều
2. Tính tổng của đoạn $[l, r]$

   Dựa trên tính chất của Prefix sum

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

3. BIT2D

```
        final int N = (int) 1e3+3;
        int[][] node = new int[N][N];
        // lấy tổng hình chữ nhật (1, 1) - (u, v)
        int get2D(int u, int v) {
            int res = 0;
            // đang xét đến cây u
            for (; u > 0; u -= u & -u) {
                // xét trên cây node[u]
                for(int x = v; x > 0; x -= x & -x) {
                    res += node[u][x];
                }
            }
            return res;
        }
        // cập nhật điểm (u, v)
        void update2D(int u, int v, int val) {
            // đang xét cây u
            for(; u < n; u += u & -u) {
                for(int x = v; x < n; x += x & -x) {
                    node[u][x] += val;
                }
            }
```

```
23          }
```

Ý tưởng: mỗi node là 1 cây BIT con

12. Mở rộng

- Kết hợp sử dụng HashMap
- Đếm số lượng phần tử lớn hơn K trong khoảng [L, R] sử dụng Fenwick tree
- Dãy con không giảm dài nhất (LIS) → Dùng max BIT
- Dãy nghịch thế

13. Bonus

Cách cài đặt cho mảng bắt đầu từ 0:

Ngoài ra thì đây là đoạn code dùng mảng bắt đầu từ 0 (zero-based array)

Lý thuyết của việc di chuyển từ node cha sang node con tuy không quá phức tạp (chỉ là các phép toán với bit), nhưng vì nhìn loằng ngoằng khó nhớ nên không mấy khi dùng.

Tuy nhiên, cần cẩn trọng với việc gọi truy vấn đoạ n*get(l, r)* như ở phần trên)

```
1           public class FenwickTree_0 {
2               int n; // size
3               final int N = (int) 1e5+5;
4               int bit[] = new int[N];
5               FenwickTree_0(int size) {
6                   n = size;
7               }
8               FenwickTree_0(int[] a) {
9                   n = a.length;
10                  for(int i = 0; i < n; ++i) {
11                      add(i, a[i]);
12                  }
13              }
14              void add(int u, int v) {
```

```
15                  for(int i = u; i < n; i = i | (i + 1)) {
16                      bit[i] += v;
17                  }
18              }
19              int getSum(int p) {
20                  int answer = 0;
21                  for(int i = p; i >= 0; i = (i & (i+1)) - 1) {
22                      answer += bit[i];
23                  }
24                  return answer;
25              }
26          }
```

Tài liệu tham khảo

[1] https://vnoi.info/wiki/algo/data-structures/fenwick.md

[2] https://www.youtube.com/watch?v=tWMn5n8FZA8

[3] https://www.stdio.vn/modern-cpp/cac-thao-tac-tren-bit-1QnH8

[4] https://cp-algorithms.com/data_structures/fenwick.html#finding-minimum-of-0-r-in-one-dimensional-array

[5] https://www.semanticscholar.org/paper/A-new-data-structure-for-cumulative-frequency-Fenwick/769fb8055fbe0997ef8d9dab6c9abf37489c6575?p2df

[6] https://www.researchgate.net/publication/282222122_Efficient_Range_Minimum_Queries_using_Binary_Indexed_Trees

[7] https://stackoverflow.com/questions/31068521/is-it-possible-to-build-a-fenwick-tree-in-on