Học Binary Indexed Trees từ các kĩ thuật đơn giản nhất.

Nguyễn Như Thắng- GV THPT Chuyên Lào Cai.

0. Mở đầu:

Bài toán cơ sở: Cho dãy số nguyên $a_1, a_2, ...a_n$, có m câu hỏi, mỗi câu hỏi yêu cầu tính tổng 1 đoạn liên tiếp từ l_i đến r_i với $i = 1..m$.

Với cách giải thông thường mỗi yêu cầu ta duyệt từ l_i đến r_i để tính tổng, như vậy độ phức tạp thuật toán là O(*m*.*n*). Nếu *m* và *n* cỡ 10^5 chắc chắn sẽ không thể thực hiện trong thời gian cho phép.

Để giải quyết vấn đề, ta sẽ cải tiến bằng cách tính trước tổng cộng dồn *s[i]* là tổng các số từ $a_1 \to a_i$ ngay khi đọc dữ liệu vào mất thời gian chung là O(n). Sau đó, mỗi câu hỏi ta chỉ thực hiện trong thời gian O(1) bằng cách lấy $s[r_i] - s[l_i - 1]$, như vậy độ phức tạp của bài toán còn O(*m*+*n*).

Tuy nhiên ta mở rộng bài toán bằng cách bổ sung thêm *m* yêu cầu nữa được đan xen vào m yêu cầu ban đầu là: cập nhật lại trị $a_i = val$. Khi đó, việc tính sẵn tổng cộng dồn s[i] lại trở nên vô nghĩa. Vì dãy số liên tục biến động, s[i] cũng sẽ luôn biến động, do đó độ phức tạp của bài toán quay về nguyên trạng ban đầu là O(m.n). Tuy nhiên, ý tưởng về việc cộng dồn lại không hề vô nghĩa khi ta kết hợp với chỉ số nhị phân để tạo ra giá trị cộng dồn được quản lý bằng cây chỉ số nhị phân (Binary Index Tree). Mỗi khi cần cập nhật, tính tổng một đoạn, ta hoàn toàn thực hiện được trong thời gian chung là O(logN). Độ phức tạp bài toán sẽ là O(NlogN+2MlogN).

Cấu trúc dữ liệu BIT có bản chất là ý tưởng cộng dồn và mã hóa các đoạn cộng dồn được quản lý bằng chỉ số nhị phân. Đây là một loại cấu trúc dữ liệu để các thuật toán thực hiện nhanh hơn. Trong bài viết này chúng ta sẽ thảo luận về cấu trúc dữ liệu Binary Indexed Trees (cây nhị phân chỉ số). Theo Peter M. Fenwick thì cấu trúc này lần đầu tiên được sử dụng để nén dữ liệu. Bây giờ nó thường được sử dụng để lưu trữ các tần số và thao tác với bảng tần số tích lũy.

Yêu cầu của bài toán cơ sở có thể được thay thế bằng việc tìm min, max,...

1. Giới thiệu bài toán

Xét bài toán sau đây. Chúng ta có *n* chiếc hộp và các truy vấn có thể là:

1. Thêm một số viên bi vào hộp *i*.
2. Tính số lượng các viên bi từ hộp *k* tới hộp *l*.

Giải pháp đơn giản có độ phức tạp thời gian là *O*(1) cho truy vấn 1 và *O*(*n*) cho truy vấn 2. Giả sử chúng ta thực hiện *m* truy vấn. Trường hợp xấu nhất (khi tất cả đều là truy vấn 2) có độ phức tạp thời gian là *O*(*n*×*m*). Sử dụng cấu trúc segment tree, chúng ta có thể giải quyết bài toán này với trường hợp xấu nhất có độ phức tạp thời gian là *O*(*m*.log$_2$*n*). Một cách khác là sử dụng cấu trúc Binary Indexed Trees, cũng với sự phức tạp thời gian trong trường hợp xấu nhất là *O*(*m*.log$_2$*n*), nhưng Binary Indexed Trees dễ viết mã hơn và yêu cầu không gian bộ nhớ ít hơn so với segment tree. Tương ứng là O(n) so với O(4n).

Để hiểu được BIT, bạn cần hiểu kĩ phần Mở đầu (nói về cộng dồn), biết cách biểu diễn số nhị phân, các phép toán logic về bit 0 và 1.

2. Ký hiệu

- *BIT* - Binary Indexed Tree.
- *MaxVal* - giá trị lớn nhất của các tần số khác không.
- *f*[*i*] - tần số (số lần xuất hiện) của giá trị với chỉ số *i*, *i* = 1 … *MaxVal*.
- *c*[*i*] - tần số tích lũy cho chỉ số *i* (*c*[*i*] = *f*[1] + *f*[2] + ... + *f*[*i*]).
- *tree*[*i*] - tổng của các tần số *f* được lưu trữ trong *BIT* với chỉ số *i* (phần sau chúng ta sẽ mô tả chỉ số này có nghĩa là gì). Đôi khi chúng ta viết cây tần số thay vì tổng các tần số được lưu trữ trong *BIT*.
- $\overline{num}$ - số bù 1 của số nguyên *num* (đảo ngược các chữ số nhị phân của *num*: 0 → 1, 1 → 0).

Chú ý: Thông thường chúng ta đặt *f*[0] = 0, *c*[0] = 0, *tree*[0] = 0, vì vậy đôi khi ta bỏ qua chỉ số 0.

3. Ý tưởng nền tảng

Mỗi số nguyên có thể được biểu diễn như là tổng các lũy thừa của 2, hay biểu diễn được trong hệ cơ số 2. Theo cách này, tần số tích lũy có thể được biểu diễn như là tổng của các tập của các tần số con. Trong bài toán này, mỗi tập sẽ chứa một số liên tiếp các tần số.

idx là một chỉ số nào đó của *BIT*. *r* là vị trí của chữ số 1 cuối cùng (chữ số 1 bên phải nhất) trong biểu diễn nhị phân của *idx*.

tree[*idx*] là tổng các tần số *f* từ chỉ số (*idx* - 2^r + 1) tới chỉ số *idx* (xem bảng 1 để hiểu rõ hơn). Chúng ta cũng nói rằng *idx* quản lí các chỉ số từ (*idx* - 2^r + 1) tới *idx* (chú ý rằng việc quản lí này là chìa khóa trong thuật toán của chúng ta và là cách thao tác với cây). Ví dụ: Theo chỉ số nhị phân: $12_{(10)}=1100_{(2)}$, do đó nút 12 sẽ quản lý các nút có chỉ số từ 9 (=12-2^2+1) đến 12.

Bảng cộng dồn các tần số (tree) bằng tổng giá trị các nút nó quản lý: ví dụ: tree[12]=f[9]+f[10]+f[11]+f[12]; tree[14]=f[13]+f[14]; tree[1]=f[1]; tree[2]=f[1]+f[2]; tree[4]=f[1]+f[2]+f[3]+f[4]; tree[5]=f[5]; tree[6]=f[5]+f[6];…

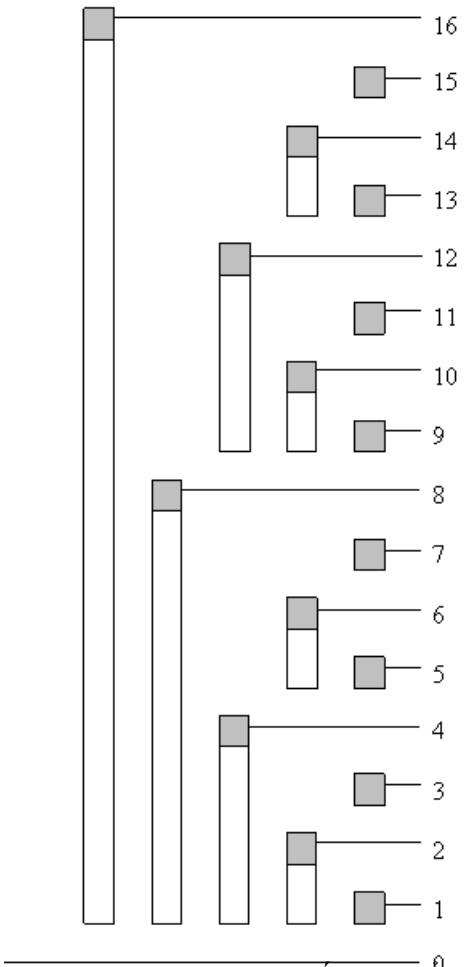
Hãy quan sát bảng giá trị minh họa bên dưới:

idx	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
***f*[*idx*]**	1	0	2	1	1	3	0	4	2	5	2	2	3	1	0	2
***c*[*idx*]**	1	1	3	4	5	8	8	12	14	19	21	23	26	27	27	29
***tree*[*idx*]**	1	1	2	4	1	4	0	12	2	7	2	11	3	4	0	29

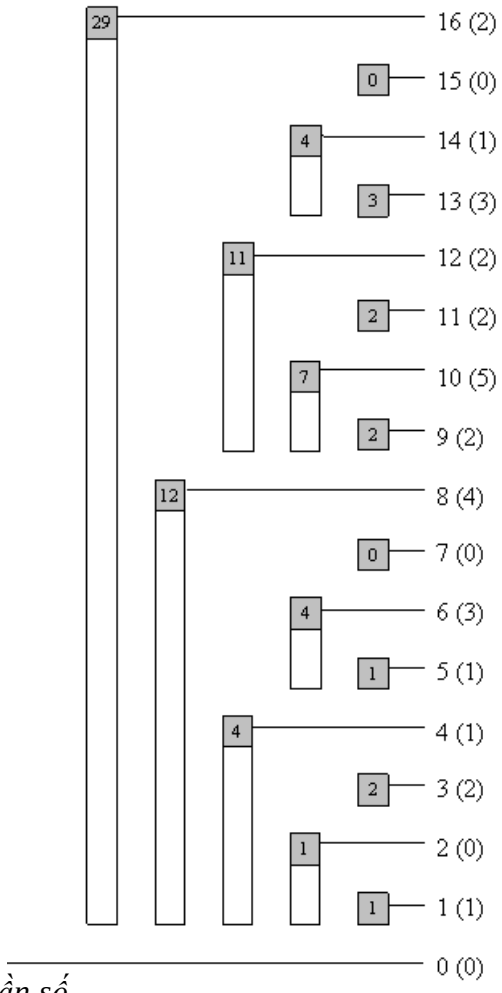
Bảng 1

idx	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Các chỉ số mà *idx* quản lí	1	1..2	3	1..4	5	5..6	7	1..8
idx	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Các chỉ số mà *idx* quản lí	9	9..10	11	9..12	13	13..14	15	1..16

Bảng 2 - Bảng quản lí các chỉ số



Hình 3. Cây quản lí chỉ số (cột hiển thị đoạn các tần số tích lũy trong phần tử đầu)



Hình 4. Cây tần số (tree)

Giả sử chúng ta đang tìm tần số tích lũy của 13 phần tử đầu tiên như sau: Trong biểu diễn nhị phân, 13 là bằng 1101. Vì vậy, chúng ta sẽ tính *c*[1101] = *tree*[1101] + *tree*[1100] + *tree*[1000] = 3 + 11 + 12 = 26. Tổng đoạn từ 1 đến 13 là c[13]=tree[13]+tree[12]+tree[8]. Tính tổng từ đoạn có chỉ số 12 đến 13 như sau: c[13]-c[11]=tree[13]+tree[12]+tree[8]-(tree[11]+tree[10]+tree[8])=tree[13]+tree[12]-tree[11]-tree[10]=3+11-2-7=5 đúng bằng f[12]+f[13]=2+3.

4. Lấy chữ số 1 bên phải nhất.

Chữ số 1 bên phải nhất hay còn gọi là chữ số 1 cuối cùng. Để lấy được số 1 cuối cùng đó, ta gọi *num* là số nguyên ta cần lấy số 1 cuối cũng. Giả sử *num* có dạng nhị phân $a1b$, ở đó a là biểu diễn các chữ số nhị phân trước số 1 cuối cùng thì b chỉ gồm các chữ số không đằng sau số 1 cuối cùng (b=000...000➔nghịch đảo của b là (~b) = $\bar{b}$ =111…111).

Số bù 1 của số num là ~num, ~num được xác định bằng cách nghịch đảo toàn bộ các bit biểu diễn num.

Số bù 2 là cách biểu diễn số đối của số num. Được xác định bằng cách lấy số bù 1 cộng thêm 1. Vậy ta có:

(num)= $a1b$ ➔(~num)= $\bar{a}0\bar{b}$ ➔ (-num)= $\bar{a}1b$ ➔(num)+(-num)=00…00.

Phép toán AND (num)&(-num)= ($a1b$)&($\bar{a}1b$) = 0..010..0= 2^r (ở đó r là vị trí của số 1 cuối cùng). Vậy thì ta dễ dàng cô lập được số 1 cuối cùng bằng phép toán (num&-num) trong C++.

$a1b$
& $\bar{a}1b$

= 0..010..0

5. Tính tần số tích lũy

Nếu chúng ta muốn đọc tần số tích lũy của một số nguyên *idx*, chúng ta cộng *tree*[*idx*] vào *sum*, sau đó loại bỏ bit cuối cùng của *idx* từ chính nó (tức là thay đổi chữ số cuối cùng bằng không) và lặp lại điều này trong khi *idx* vẫn lớn hơn 0. Chúng ta có thể sử dụng hàm sau (viết bằng C++):

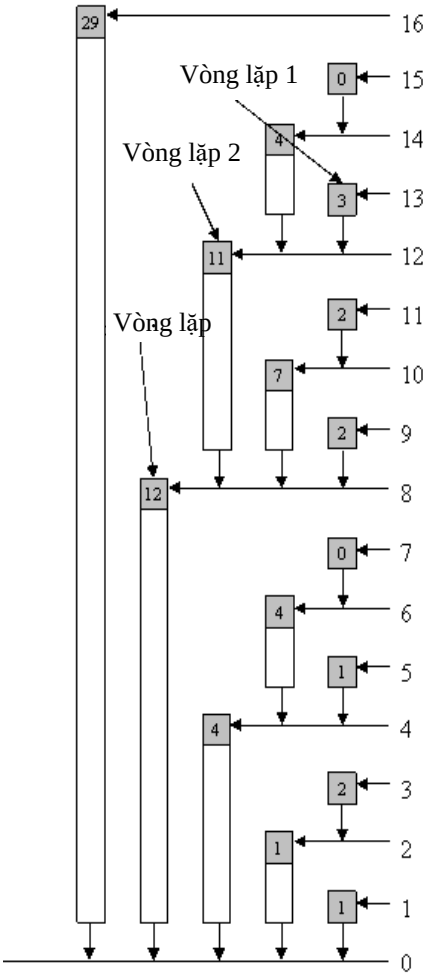
```
int read(int idx) {
    int sum = 0;
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}
```

Ví dụ với *idx* = 13, *sum* = 0:

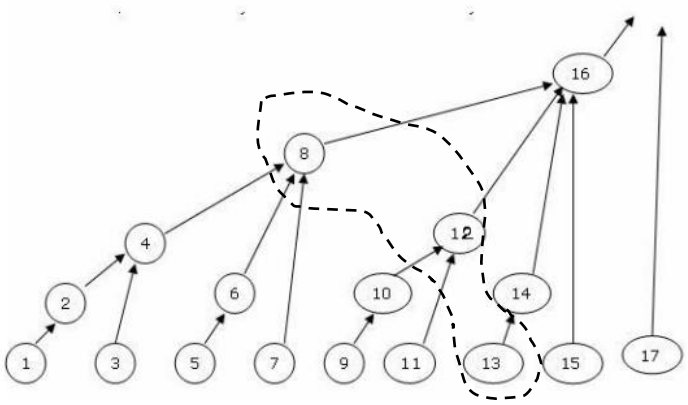
Vòng lặp	*idx*	Vị trí số 1 cuối cùng	Tree[idx]	*idx* & *-idx*	*Sum*
1	13 = 1101	0	3	1 (2^0)	3
2	12 = 1100	2	11	4 (2^2)	14
3	8 = 1000	3	12	8 (2^3)	26
4	0 = 0	---	---	---	---

Vì vậy, kết quả tần số tích lũy của chỉ số 13 là 26. Số lần lặp trong hàm này là số bit 1 trong *idx*, vì vậy số lần lặp nhiều nhất là $\log_2 MaxVal$.

Độ phức tạp thời gian của hàm *read*: $O(\log_2 MaxVal)$.



Hình 5 – Mũi tên minh họa đường đi từ chỉ số 13 tới 0 trong việc tính sum.



6. Cập nhật một lại giá trị của một nút.

Để cập nhật giá trị tại một nút f[idx] thì nó sẽ tác động lên tất cả các nút quản lý nút tree[idx].

Ví dụ: để cập nhật lại giá trị của nút idx=9; ta cần cập nhật các nút 10, 12,16,32,64,…2^k (trong đó k<=$\log_2 MaxVal$). Cách làm: lặp lại việc cộng thêm 1 vào số 1 bên phải nhất cho đến khi *idx>MaxVal*. Biểu diễn các nút bằng chỉ số nhị phân theo khi cập nhật nút 9 theo thứ tự sau: Bắt đầu từ nút *idx*=$9=1001_2 \rightarrow 1010_2 \rightarrow 1100_2 \rightarrow 10000_2 \rightarrow 100000_2 \rightarrow 1000000_2 \rightarrow \ldots$

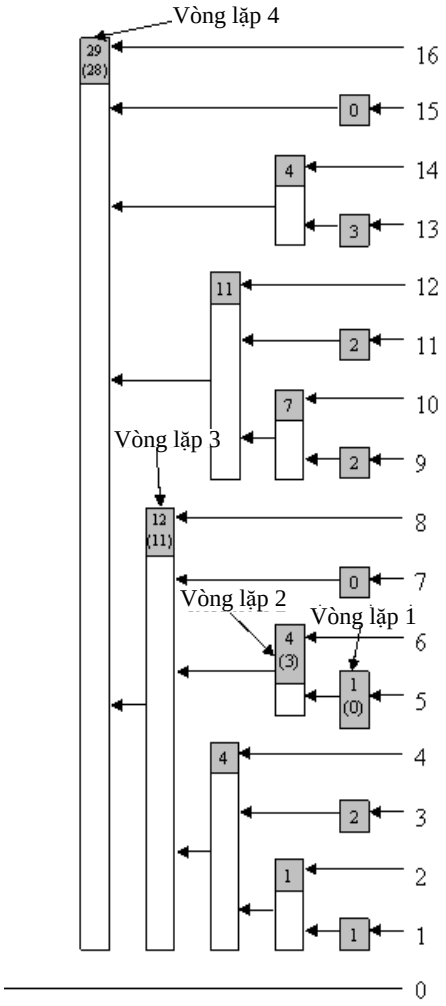
Hàm trong trong C++ được cài đặt như sau:

```
void update(int idx, int val) {
    while (idx <= MaxVal) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}
```

Hãy xem ví dụ với *idx* = 5

Vòng lặp	*idx*	Vị trí của chữ số cuối cùng	*idx* & *-idx*
1	5 = 101	0	1 (2^0)
2	6 = 110	1	2 (2^1)
3	8 = 1000	3	8 (2^3)
4	16 = 10000	4	16 (2^4)
5	32 = 100000	---	---

Độ phức tạp thời gian của hàm *update*: $O(\log_2 MaxVal)$.



Hình 6 – Cập nhật cây (trong cặp ngoặc đơn là tần số trước khi cập nhật); Mũi tên minh họa đường đi trong khi chúng ta cập nhật cây từ chỉ số tới MaxVal (Hình vẽ minh họa ví dụ cho chỉ số 5)

7. BIT 2D

BIT có thể được sử dụng như là một cấu trúc dữ liệu đa chiều. Giả sử bạn có một mặt phẳng với các dấu chấm (có tọa độ không âm). Bạn thực hiện ba truy vấn:

1. Đặt dấu chấm ở (x, y).
2. Loại bỏ dấu chấm ở (x, y).
3. Đếm số chấm nằm trong hình chữ nhật (0, 0), (x, y) - ở đó (0, 0) là góc dưới bên trái, (x, y) là góc trên bên phải và các cạnh song song với trục hoành và trục tung.

Nếu m là số lượng các truy vấn, *max_x* là hoành độ lớn nhất và *max_y* là tung độ lớn nhất, thì bài toán sẽ được giải với độ phức tạp thời gian là $\boldsymbol{O(m \times \log_2(max\_x) \times \log_2(max\_y))}$. Trong trường hợp này, mỗi phần tử của cây sẽ chứa một mảng *tree*[*max_x*][*max_y*]. Việc cập nhật các chỉ số của hoành độ giống như trước. Ví dụ, giả sử chúng ta đặt hoặc gỡ bỏ dấu chấm ở (a, b) thì chúng ta sẽ gọi *update*(a, b, 1) hoặc *update*(a, b, -1), ở đó hàm update được cài đặt như sau:

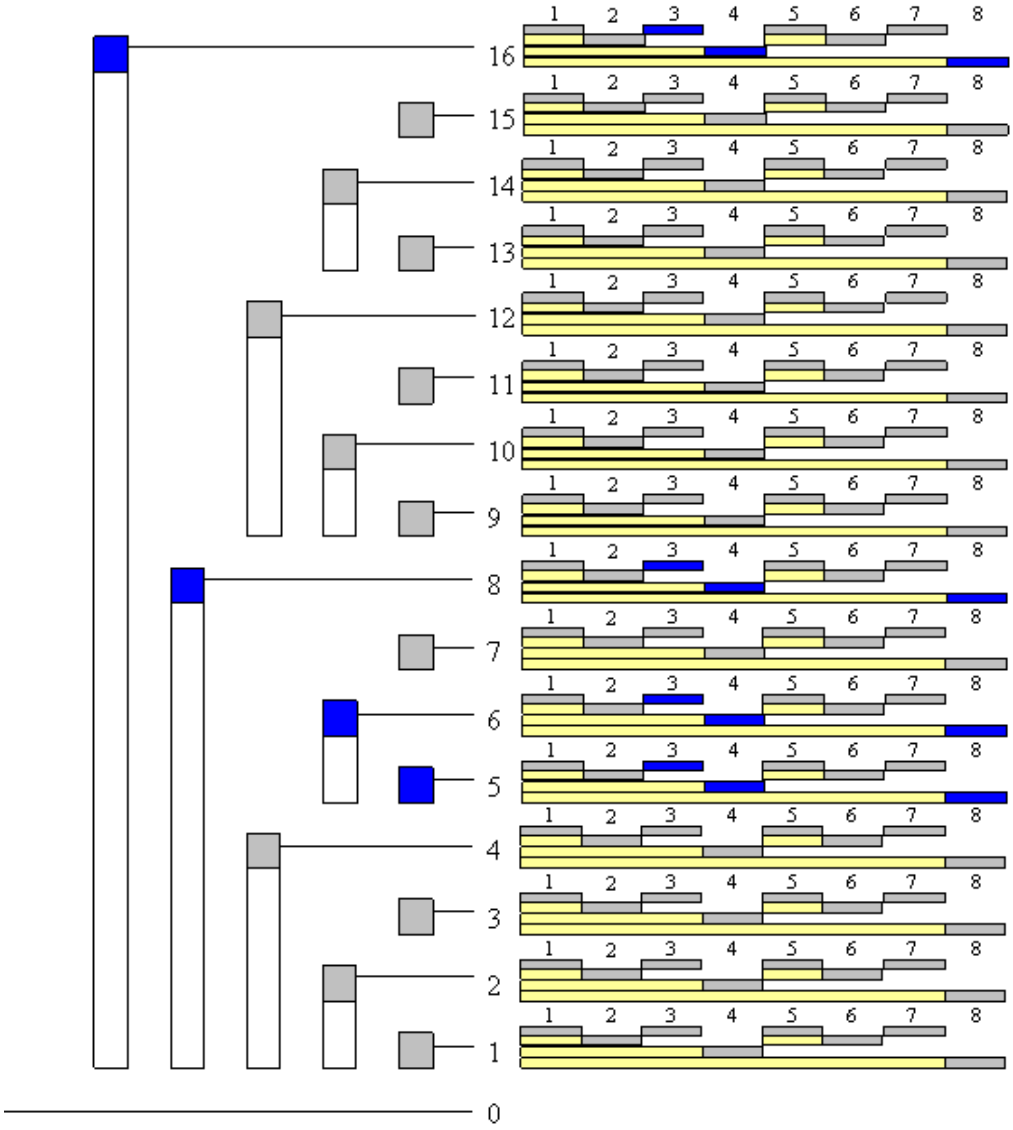
```
void update(int x, int y, int val) {
    while (x <= max_x) {
        updatey(x , y , val);
        // this function should update array tree[x]
        x += (x & -x);
    }
}
```

Hàm *updatey* là giống hàm *update*:

```
void updatey(int x, int y, int val) {
    while (y <= max_y) {
        tree[x][y] += val;
        y += (y & -y);
    }
}
```

Bạn có thể viết gộp lại trong một hàm:

```
void update(int x, int y, int val) {
    int y1;
    while (x <= max_x) {
        y1 = y;
        while (y1 <= max_y) {
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
```



Hình 7 – BIT là mảng của mảng, vì vậy BIT là mảng 2 chiều (kích thước 16×8). Trường mầu xanh là trường được cập nhật khi chúng ta cập nhật chỉ số idx(5, 3).

Việc thay đổi cho các hàm khác là rất giống nhau. Ngoài ra, lưu ý rằng BIT có thể được sử dụng như là một cấu trúc dữ liệu n chiều.

8. Một số bài toán áp dụng

8.1. Bài toán "Range Sum Query"

Có n cái hộp được đánh số từ 1 đến n ($1 \le n \le 100.000$), ban đầu tất cả các hộp này đều rỗng. Có m ($1 \le m \le 100.000$) truy vấn, mỗi truy vấn có 1 trong 2 dạng sau:

- "+ i v": Thêm v viên bi vào hộp i ($1 \le i \le n$, $0 \le v \le 100.000$).
- "? i j": Tính tổng số lượng các viên bi nằm trong các hộp từ i đến j ($1 \le i \le j \le n$).

Dữ liệu: Dòng đầu tiên chứa 2 số nguyên n và m. Tiếp theo có m dòng, mỗi dòng chứa một phép một truy vấn như mô tả ở trên. Các số trên cùng một dòng ngăn cách nhau bởi một dấu cách.

Kết quả: Đưa ra lần lượt các câu trả lời cho mỗi truy vấn dạng thứ hai. Mỗi câu trả lời ghi trên một dòng.

Ví dụ:

input	Output
6 5 + 1 8	21 26

+ 2 13 ? 1 3 + 4 5 ? 1 6	

Phân tích và thiết kế thuật toán: Đây chính là bài toán đã nêu ở phần giới thiệu. Bài này có thể giải bằng cấu trúc dữ liệu Binary Indexed Trees và Segment Trees. Để tiện cho việc so sánh, dưới đây là hai lời giải sử dụng các cấu trúc dữ liệu trên.

Chương trình cài đặt bằng cấu trúc dữ liệu Binary Indexed Trees:

```
#include <cstdio>
#include <cstring>
using namespace std;

int n, m;
long long tree[100001];

void update(int idx, int val) {
    while (idx <= n) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

long long read(int idx) {
    long long sum = 0;
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

int main () {
    scanf("%d%d\n", &n, &m);
    memset(tree, 0, sizeof(tree));
    for ( ; m > 0; m--) {
        char c;
        int i, j;
        scanf("%c%d%d\n", &c, &i, &j);
        if (c == '+')
            update(i, j);
        else
            printf("%lld\n", read(j)-read(i-1));
    }
    return 0;
}
```

Chương trình cài đặt bằng cấu trúc dữ liệu Segment Trees:

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

int n, m;
long long tree[262144];

long long query(int node, int l, int r, int i, int j) {
    if (l == i && r == j)
```

```
        return tree[node];

    long long ans = 0;
    int c = (l+r)/2;
    if (i <= c) ans += query(2*node + 1, l, c, i, min(c, j));
    if (j > c) ans += query(2*node + 2, c+1, r, max(c+1, i), j);
    return ans;
}

void update(int node, int l, int r, int i, int v) {
    if (l == i && i == r) {
        tree[node] += v;
        return;
    }

    int p1 = 2*node + 1, p2 = 2*node + 2, c = (l+r)/2;
    if (i <= c) update(p1, l, c, i, v);
    if (i > c) update(p2, c+1, r, i, v);
    tree[node] = tree[p1] + tree[p2];
}

int main () {
    scanf("%d%d\n", &n, &m);
    memset(tree, 0, sizeof(tree));
    for ( ; m > 0; m--) {
        char c;
        int i, j;
        scanf("%c%d%d\n", &c, &i, &j);
        if (c == '+')
            update(0, 1, n, i, j);
        else
            printf("%lld\n", query(0, 1, n, i, j));
    }
    return 0;
}
```

8.2. Bài toán "Tổng ma trận"

Cho một ma trận $N \times N$ được làm đầy với các con số. BuggyD đang phân tích ma trận và bây giờ ông ta muốn tính tổng của một ma trận con nào đó. Vì vậy ông ta muốn có một hệ thống mà ở đó ông ta có thể nhận được kết quả từ mỗi truy vấn của mình. Ngoài ra ma trận là động và giá trị của một ô bất kỳ có thể bị thay đổi bởi một lệnh trong hệ thống đó.

Giả sử ban đầu, tất cả các ô của ma trận được làm đầy với số 0. Hãy thiết kế một hệ thống như vậy cho BuggyD. Đọc các mô tả dữ liệu vào ra để biết thêm chi tiết.

Dữ liệu: Dòng đầu tiên chứa chứa một số nguyên N ($1 \le N \le 1024$), mô tả kích thước của ma trận. Tiếp theo là danh sách các câu lệnh (có không quá 100.000 câu lệnh), mỗi câu lệnh chứa trên một dòng và có 3 dạng sau:

1. "SET x y num" – Gán giá trị của ô (x, y) giá trị num ($0 \le x, y < N$).
2. "SUM $x1$ $y1$ $x2$ $y2$" – Tính và ghi ra tổng giá trị của các ô trong hình chữ nhật từ $(x1, y1)$ tới $(x2, y2)$. Giả thiết $0 \le x1 \le x2 < N$, $0 \le y1 \le y2 < N$ và kết quả vừa với kiểu số nguyên 32-bit có dấu.
3. "END" – Kết thúc danh sách các câu lệnh.

Kết quả: Ghi ra câu trả lời trên một dòng cho mỗi câu lệnh "SUM".

Ví dụ:

input	**Output**
4 SET 0 0 1	1 12

SUM 0 0 3 3 SET 2 2 12 SUM 2 2 2 2 SUM 2 2 3 3 SUM 0 0 2 2 END	12 13

Phân tích và thiết kế thuật toán: Bài toán này được giải bằng sử dụng cấu trúc dữ liệu BIT 2D. Chương trình được cài đặt như sau.

```
#include <cstdio>
#include <cstring>
using namespace std;

int n, tree[1025][1025];

void update(int x, int y, int num) {
    int y1;
    while (x <= n) {
        y1 = y;
        while (y1 <= n) {
            tree[x][y1] += num;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

int query(int x, int y) {
    int y1, sum = 0;
    while (x > 0) {
        y1 = y;
        while (y1 > 0) {
            sum += tree[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return sum;
}

int main () {
    int t, x1, y1, x2, y2, num;
    char com[5];

    scanf("%d", &n);
    memset(tree, 0, sizeof(tree));
    while (true) {
        scanf("%s", com);
        if (!strcmp(com,"END")) break;
        if (!strcmp(com,"SET")) {
            scanf("%d %d %d", &x1, &y1, &num);
            x1++, y1++;
            int s1 = query(x1, y1);
            int s2 = query(x1, y1-1);
            int s3 = query(x1-1, y1);
            int s4 = query(x1-1, y1-1);
            update(x1, y1, num - (s1-s2-s3+s4));
        }
        else {
            scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
            x1++, y1++, x2++, y2++;
```

```
            printf("%d\n",query(x2,y2)-query(x2,y1-1)-query(x1-1,y2)+query(x1-
1,y1-1));
        }
    }

    return 0;
}
```

8.3. Bài toán "Dãy nghịch thế"

Cho một dãy số a_1, a_2, ..., a_N. Một nghịch thế là một cặp số u, v sao cho $1 \le u < v \le N$ và $a_u > a_v$. Nhiệm vụ của bạn là đếm số nghịch thế.

Dữ liệu: Dòng đầu ghi số nguyên N ($1 \le N \le 60.000$). Dòng thứ hai ghi các số a_1, a_2, ..., a_N ($1 \le a_i \le 60.000$).

Kết quả: Ghi ra một số duy nhất là số nghịch thế.

Ví dụ:

input	Output
3 3 1 2	2

8.4. Bài toán "Range Sum Query 2"

Cho một dãy gồm n phần tử được đánh số từ 1 đến n ($1 \le n \le 50.000$) có giá trị ban đầu bằng 0. Có m ($1 \le m \le 100.000$) phép biến đổi và truy vấn:

- Biến đổi có dạng "+ i j v": cộng vào các phần tử từ vị trí i đến j với v ($1 \le i \le j \le n$, $|v| \le 100.000$).
- Truy vấn có dạng "? i j": cho biết tổng của các phần tử từ vị trí i đến j ($1 \le i \le j \le n$).

Đưa ra câu trả lời cho lần lượt các truy vấn dạng thứ hai.

Dữ liệu: Dòng đầu tiên chứa 2 số nguyên n và m. Tiếp theo có m dòng, mỗi dòng chứa một phép biến đổi hoặc một truy vấn.

Kết quả: Đưa ra câu trả lời cho lần lượt các truy vấn dạng thứ hai. Mỗi câu trả lời ghi trên một dòng.

Ví dụ:

input	Output
10 7 ? 1 10 + 3 8 5 ? 2 5 + 1 5 -3 + 1 10 2 ? 1 10 ? 2 6	0 15 35 18

8.5. Bài toán "Floating Median"

Trong khí tượng có một công cụ thống kê chung là tính số trung vị của một tập các phép đo. Cho K số, ta sắp xếp các số đó theo thứ tự không giảm, khi đó số trung vị của chúng là số thứ $\left\lfloor \frac{K+1}{2} \right\rfloor$. Ví dụ, số trung vị của (1, 2, 6, 5, 4, 3) là 3 và số trung vị (11, 13, 12, 14, 15) là 13.

Bạn hãy viết một phần mềm cho thiết bị đo nhiệt độ mỗi giây một lần. Thiết bị này có màn hình hiển thị kỹ thuật số nhỏ. Bất cứ lúc nào, màn hình sẽ hiển thị nhiệt độ trung bình đo được trong K giây cuối cùng.

Trước khi cài đặt phần mềm của bạn vào thiết bị, bạn cần kiểm tra nó trên máy tính. Thay vì đo nhiệt độ, chúng ta sẽ sử dụng bộ sinh số ngẫu nhiên (RNG - Random Number Generator) để tạo ra nhiệt độ "giả". Cho ba số nguyên *seed*, *mul* và *add*, chúng ta xác định dãy các nhiệt độ như sau:

- $t_0 = seed$
- $t_{i+1} = (t_i \times mul + add) \bmod 65536$

Ngoài các thông số của RNG, bạn sẽ nhận được hai số nguyên N và K.

Xét dãy gồm N nhiệt độ đầu tiên được tạo ra bởi RNG (tức là các giá trị t_0 đến t_{n-1}). Dãy này có $N-K+1$ dãy con liên tiếp độ dài K. Với mỗi dãy như vậy, chúng ta cần tính số trung vị của nó.

Cho các số *seed*, *mul*, *add*, N và K. Hãy tính tất cả các số trung vị như mô tả ở trên và đưa ra tổng của chúng.

Dữ liệu: Gồm 5 dòng chứa lần lượt các số nguyên *seed*, *mul*, *add*, N và K ($0 \le seed, mul, add \le 65.535$; $1 \le N \le 250.000$; $1 \le K \le 5.000$; $K \le N$).

Kết quả: Đưa ra tổng các số trung vị như mô tả ở trên.

Ví dụ:

input	Output
10 0 13 5 2	49
3 1 1 10 3	60

Giải thích ví dụ 1: Các nhiệt độ tạo ra là 10, 13, 13, 13, 13. Các dãy con liên tiếp độ dài 2 là (10, 13), (13, 13), (13, 13), (13, 13). Trung vị của chúng là 10, 13, 13, 13. Tổng của các trung vị là 10 + 13 + 13 + 13 = 49.
Giải thích ví dụ 2: Các nhiệt độ được tạo ra là: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12. Các dãy con liên tiếp độ dài 3 là (3, 4, 5), (4, 5, 6), ..., (10, 11, 12). Trung vị của chúng là 4, 5, ..., 11. Tổng của các trung vị là 4 + 5 + ... + 11 = 60.
Một số bài tập khác trên SPOJ:
http://vn.spoj.com/problems/NKINV/ - solution - code

http://vn.spoj.com/problems/CRATE/ - solution - code

http://www.spoj.com/problems/RATING/ - vn - solution - code

http://vn.spoj.com/problems/KINV/ - solution - code

http://vn.spoj.com/problems/C11SEQ/ - solution - code

http://vn.spoj.com/problems/KMEDIAN/ - vn - sotution - code

http://vn.spoj.com/problems/NKLUCK/ - sotution - code

http://vn.spoj.com/problems/FOCUS/ - solution - code

http://vn.spoj.com/problems/NKMAXSEQ/ - solution - code

http://vn.spoj.com/problems/NKREZ/ - solution - code

http://vn.spoj.com/problems/MEDIAN/ - solution - code

http://vn.spoj.com/problems/PVOI14_4/ - solution - code

http://www.spoj.com/problems/MATSUM/ - vn - solution - code

http://www.spoj.com/problems/CCOST/ - vn - solution - code

http://vn.spoj.com/problems/KQUERY/ - solution - code

http://vn.spoj.com/problems/DQUERY/ - en - solution - code

http://vn.spoj.com/problems/SWAPB/ - solution

http://vn.spoj.com/problems/SKWLTH/ - solution

http://vn.spoj.com/problems/PLACE/ - solution

http://vn.spoj.com/problems/MCHAOS/ - solution

http://www.spoj.com/problems/INCSEQ/ - code

http://www.spoj.com/problems/INVCNT/ - code

http://www.spoj.com/problems/NICEDAY/ - code

9. Kết luận

- Viết chương trình với cấu trúc Binary Indexed Trees là rất dễ dàng.
- Mỗi truy vấn trên Binary Indexed Trees có độ phức tạp thời gian logarit.
- Binary Indexed Trees cần không gian bộ nhớ tuyến tính.
- Bạn có thể sử dụng nó như là một cấu trúc dữ liệu *n* chiều.

10. Tài liệu tham khảo

- http://community.topcoder.com
- http://vnoi.info
- http://vn.spoj.com
- http://www.spoj.com