

Mảng cộng dồn và mảng hiệu

Mảng cộng dồn và mảng hiệu

Tác giả:

- Bùi Nguyễn Đức Tân - VNU-HCM, High School for the Gifted

Reviewer:

- Trần Quang Lộc - ITMO University
- Hoàng Xuân Nhật - VNU-HCM, University of Science
- Nguyễn Phú Bình - Hung Vuong High School for the Gifted, Binh Duong Province

Khái niệm

Mảng cộng dồn (prefix sum)

Cho một mảng A có n phần tử được đánh số từ 0 đến $n - 1$, ta dựng mảng $S(A)$ theo quy tắc sau:

- $S_0 = c$, với c là một hằng số thực
- $S_i = S_{i-1} + A_{i-1} = c + \sum_{j=0}^{i-1} A_j$, với $1 \leq i < n$

Mảng $S(A)$ được gọi là **mảng cộng dồn (tiền tố)** theo c của A , gọi cách khác là prefix sum của A . Từ một mảng A , ta có thể sinh ra vô hạn mảng $S(A)$ bằng cách chọn một số thực c tùy ý; trên thực tế, ta thường chọn $c = 0$ để thuận tiện hơn khi tính toán.

	0	1	2	3	4	5	6	7
A	3	-1	-4	1	5	9	-2	-6

S									
	0	1	2	3	4	5	6	7	8

$$S_i = 0 \quad \text{với } i = 0$$

$$S_i = S_{i-1} + A_{i-1} \quad \text{với } i > 0$$

Mảng hiệu (difference array)

Cũng với mảng A , ta có thể dựng mảng $D(A)$ theo quy tắc:
 $D_i = A_{i+1} - A_i$ ($0 \leq i < n - 1$).

Mảng $D(A)$ được gọi là **mảng hiệu** của A , có tên tiếng Anh là difference array.

	0	1	2	3	4	5	6	7
A	3	-1	-4	1	5	9	-2	-6

D							
	0	1	2	3	4	5	6

$$D_i = A_{i+1} - A_i \quad \text{với } 0 \leq i < n - 1$$

Cài đặt

Mảng cộng dồn

Để dựng mảng cộng dồn, ta có thể áp dụng định nghĩa ở trên để dựng trực tiếp mảng:

```

1 | vector<int> buildPrefixSum(const vector<int>& a, int C = 0) {
2 |     int n = (int)a.size();
3 |     vector<int> prefixSum(n + 1);
4 |
5 |     prefixSum[0] = C;
6 |
7 |     for (int i = 0; i < n; i++)
8 |         prefixSum[i + 1] = prefixSum[i] + a[i];
9 | }
```

```
10 |  
11 |     return prefixSum;  
    | }
```


Ngoài ra, thư viện C++ STL cũng cung cấp hàm `partial_sum` để phục vụ quá trình dựng mảng cộng dồn, cú pháp của hàm như sau:

```
1 | partial_sum(first, last, result, binary_op)
```

Hàm trên sẽ thực hiện tính mảng cộng dồn với toán tử `binary_op` trên các container của C++ có iterator trở từ `[first, last)` và trả mảng cộng dồn sang container bắt đầu từ iterator trở về `result`.

Có hai lưu ý quan trọng khi sử dụng hàm này:

- Hàm `partial_sum` duyệt qua các phần tử của container theo tính chất của iterator của container đó; vì thế giá trị của mảng cộng dồn sẽ phụ thuộc vào thứ tự xuất hiện của các phần tử trong container đó.
- Tham số `binary_op` có thể được để trống. Khi này, toán tử mặc định là phép cộng (+).

Bạn đọc có thể tham khảo thêm về hàm này tại trang [cppreference](#) .

Code minh họa:

```
1 | void printArray(const vector<int>& arr) {  
2 |     for (int v : arr) cout << v << " ";  
3 |     cout << endl;  
4 | }  
5 |  
6 | vector<int> a = {3, -1, -4, 1, 5, 9, -2, -6};  
7 | int n = (int)a.size();  
8 |  
9 | // Dựng thủ công  
10 | vector<int> prefOne = buildPrefixSum(a);  
11 | printArray(prefOne); // 0 3 2 -2 -1 4 13 11 5  
12 |  
13 | // Dựng bằng partial_sum  
14 | vector<int> prefTwo(n);  
15 | partial_sum(a.begin(), a.end(), prefTwo.begin());  
16 | printArray(prefTwo); // 3 2 -2 -1 4 13 11 5
```

Trong cả hai cách trên, độ phức tạp của quá trình dựng là $\mathcal{O}(n)$.

Mảng hiệu

Tương tự, ta cũng có thể áp dụng định nghĩa để dựng trực tiếp mảng hiệu:


```
1 | vector<int> buildDifferenceArray(const vector<int>& a) {
2 |     int n = (int)a.size();
3 |
4 |     vector<int> differenceArray(n - 1);
5 |
6 |     for (int i = 0; i < n - 1; i++)
7 |         differenceArray[i] = a[i + 1] - a[i];
8 |
9 |     return differenceArray;
10| }
```

Ngoài ra, thư viện C++ STL cũng cung cấp hàm `adjacent_difference` để phục vụ quá trình dựng mảng cộng dồn, cú pháp của hàm như sau:

```
1 | adjacent_difference(first, last, result, binary_op)
```

Hàm trên sẽ thực hiện tính mảng hiệu với toán tử `binary_op` trên các container của C++ có iterator trở từ `[first, last)` và trả mảng hiệu sang container bắt đầu từ iterator trở về `result`.

Các lưu ý ở phần `partial_sum` cũng được áp dụng cho hàm này.

Bạn đọc có thể tham khảo thêm về hàm này tại trang [cppreference](#) .

Code minh họa:

```
1 | // Dựng thủ công
2 | vector<int> diffOne = buildDifferenceArray(a);
3 | printArray(diffOne);
4 | // -4 -3 5 4 4 -11 -4
5 |
6 | // Dựng bằng partial_sum
7 | vector<int> diffTwo(n);
8 | adjacent_difference(a.begin(), a.end(), diffTwo.begin());
9 | printArray(diffTwo);
10| // 3 -4 -3 5 4 4 -11 -4
```

Trong cả hai cách trên, độ phức tạp của quá trình dựng là $\mathcal{O}(n)$.

Tính chất

Độ dài mảng

- Đối với **mảng cộng dồn**, do ta cần thêm một hằng số c ở đầu mảng, độ dài của mảng $S(c, A)$ là $n + 1$, nhiều hơn 1 phần tử so với mảng A gốc.
- Ngược lại, **mảng hiệu** được dựng dựa trên hiệu của hai phần tử liên kề nhau. Tuy nhiên, trong mảng A chỉ có $n - 1$ cặp như vậy, vì thế độ dài của $D(A)$ là $n - 1$, ít hơn 1 phần tử so với mảng A gốc.

Tính riêng biệt

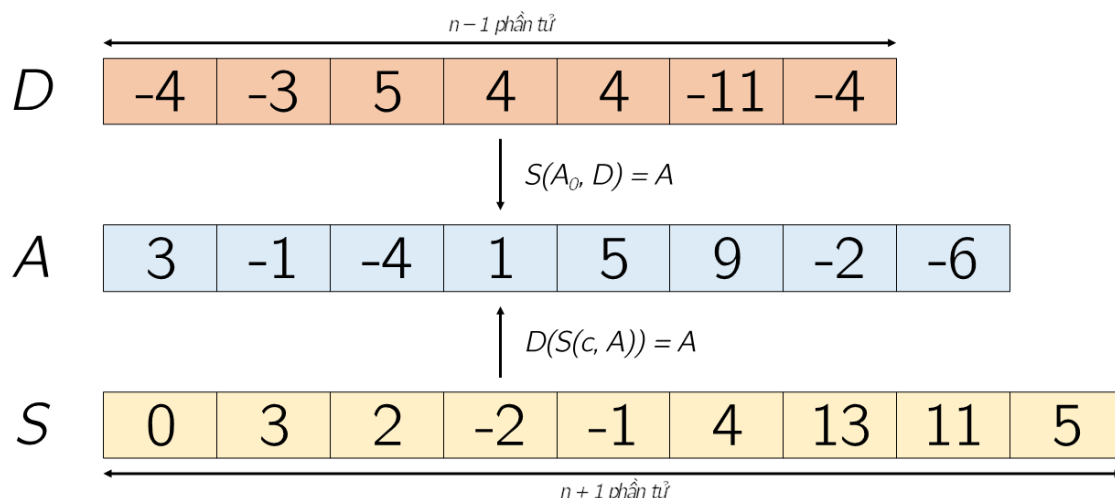
- Từ một mảng A bất kỳ, ta sinh được vô hạn mảng cộng dồn $S(c, A)$ từ A . Tuy nhiên, các mảng cộng dồn này chỉ khác nhau ở giá trị c được chọn.
- Cũng với mảng A đó, ta sinh được **một và chỉ một** mảng hiệu $D(A)$ từ A .

Liên hệ giữa mảng cộng dồn và mảng hiệu

Cho mảng cộng dồn $S(c, A)$ và mảng hiệu $D(A)$, ta có thể dễ dàng khôi phục nội dung của mảng A thông qua các phép sau:

- $S(A_0, D(A)) = A$
- $D(S(c, A)) = A$ với mọi c thực

Hình dưới đây mô tả rõ hơn mối liên hệ giữa mảng gốc, mảng hiệu và mảng cộng dồn sinh ra từ nó:



Hàm `partial_sum` và `adjacent_difference` trong C++ STL cũng tuân theo quy tắc này trên. Tuy nhiên, các thao tác trên hai hàm này có phần phức tạp hơn so với thao tác trên mảng mà ta cài đặt thủ công.

Ứng dụng của mảng cộng dồn

Mảng cộng dồn có một tính chất quan trọng: các phần tử được cộng lại chồng chất lên nhau một cách liên tiếp, vì thế, với mọi nửa khoảng $[l, r)$ ($0 \leq l < r \leq n$), ta chỉ cần tính $S_r - S_l$ để tính tổng của các phần tử $A_l, A_{l+1}, \dots, A_{r-2}, A_{r-1}$. Việc trừ này cũng sẽ khử đi hằng số c của S , vì thế ta có thể dùng bất kỳ mảng S nào được sinh từ A để tính tổng.

Chứng minh:

Theo định nghĩa: $S_i = c + \sum_{j=0}^{i-1} A_j$

Khi này:

$$\begin{aligned} S_r - S_l &= c + \sum_{j=0}^{r-1} A_j - \left(c + \sum_{j=0}^{l-1} A_j \right) \\ &= c + \sum_{j=0}^{r-1} A_j - c - \sum_{j=0}^{l-1} A_j \\ &= c - c + \sum_{j=0}^{r-1} A_j - \sum_{j=0}^{l-1} A_j \\ &= \sum_{j=l}^{r-1} A_j + \sum_{j=0}^{l-1} A_j - \sum_{j=0}^{l-1} A_j \\ &= \sum_{j=l}^{r-1} A_j \quad \blacksquare \end{aligned}$$

Trong đa số trường hợp, mảng cộng dồn thường được sử dụng nếu bài toán yêu cầu tính tổng một đoạn con nhiều lần liên tiếp. Dưới đây, ta sẽ đề cập một số bài toán có điều kiện trên.

Bài toán minh họa

Nguồn: [CSES - Maximum Subarray Sum](#) 

Đề bài: Cho một mảng A gồm n phần tử. Tìm đoạn con khác rỗng có tổng lớn nhất. Giới hạn: $1 \leq n \leq 2 \cdot 10^5, |A_i| \leq 10^9$

Trước hết, ta tạo mảng $pref = S(0, A)$ để lưu mảng cộng dồn của A . Giả sử với r cố định, ta cần tìm $l < r$ sao cho tổng các phần tử trong nửa khoảng $[l, r)$ đạt cực đại. Ta viết lại bài toán theo công thức sau:

$$\begin{aligned} ans_r &= \max_{0 \leq l < r} (pref_r - pref_l) \\ &= pref_r + \max_{0 \leq l < r} (-pref_l) \\ &= pref_r - \min_{0 \leq l < r} pref_l \end{aligned}$$

Nếu ta chạy r từ 1 đến n , ta có thể cập nhật cuốn chiếu min theo từng $pref_r$; việc này cho phép chúng ta tính ans_r với độ phức tạp $O(1)$. Đáp án của bài toán là $\max_r ans_r$ với $1 \leq r \leq n$.

Độ phức tạp của cách trên là $O(n)$. Code tham khảo:

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int MAXN = 200003;
6  const long long INF = 1e18;
7
8  int n, a[MAXN];
9  long long prefSum[MAXN], prefMin[MAXN], ans = -INF;
10
11 int main() {
12     cin >> n;
13     for (int i = 1; i <= n; i++) cin >> a[i];
14     prefSum[0] = prefMin[0] = 0;
15     for (int i = 1; i <= n; i++)
16         prefSum[i] = prefSum[i - 1] + a[i], prefMin[i] = min(prefMin
17     for (int i = 1; i <= n; i++)
18         ans = max(ans, prefSum[i] - prefMin[i - 1]);
19     cout << ans;
20 }
```

Lưu ý, ta có thể thu gọn `prefSum` và `prefMin` thành một biến duy nhất để tối ưu bộ nhớ sử dụng.

Bên cạnh cách giải đã đề cập, bài toán này cũng có thể giải bằng phương pháp quy hoạch động hoặc chia để trị.

Ứng dụng của mảng hiệu

Giả sử, ta cần cộng thêm một lượng k vào một đoạn con $[l, r]$ của mảng A . Thay vì cộng lần lượt từng phần tử với độ phức tạp $O(n)$, ta có thể dựng mảng hiệu $D(A)$ và cập nhật trên đó với độ phức tạp $O(1)$. Dựng mảng hiệu D từ A lưu trữ chênh lệch của các cặp phần tử liên kề nhau, ta chia các trường hợp sau để nhận xét:

- Nếu A_i và A_{i+1} đều nằm ngoài $[l, r] \Rightarrow$ Giá trị của hai phần tử không đổi $\Rightarrow D_i$ không đổi
- Nếu A_i và A_{i+1} đều nằm trong $[l, r] \Rightarrow$ Giá trị của hai phần tử đều được cộng một lượng $k \Rightarrow D_i$ không đổi

- Nếu chỉ một trong A_i hoặc A_{i+1} nằm trong $[l, r] \Rightarrow$ Giá trị của một phần tử giữ nguyên còn phần tử còn lại được cộng một lượng $k \Rightarrow D_i$ thay đổi

Chỉ duy nhất trường hợp cuối cùng ta cần tác động trực tiếp lên D . Nhận thấy, trường hợp cuối chỉ thỏa khi $i = l - 1$ hoặc $i = r$, ta chỉ cần tác động trực tiếp lên D_{l-1} và D_r để cập nhật đoạn. Sau khi cập nhật hoàn tất, ta áp dụng tính chất $S(D, A_0) = A$ để lấy giá trị cuối cùng của A .

Bài toán minh họa

Nguồn: [Codeforces - Karen and Coffee](#) 

Đề bài: Cho một mảng A có vô số phần tử mang giá trị 0. Có n truy vấn cập nhật, mỗi truy vấn yêu cầu cập nhật toàn bộ phần tử từ l_i đến r_i thêm 1.

Sau khi cập nhật xong, trả lời q câu hỏi với nội dung sau: có bao nhiêu vị trí i thỏa $a \leq i \leq b$ và $A_i \geq k$?

Giới hạn: $k \leq n \leq 2 \cdot 10^5, q \leq 2 \cdot 10^5, 1 \leq l_i \leq r_i \leq 2 \cdot 10^5, 1 \leq a \leq b \leq 2 \cdot 10^5$

Do điều kiện l_i, r_i, a, b của đề bài, mảng A sẽ chỉ lưu trữ tối đa 200 nghìn phần tử, toàn bộ phần tử này đều có chỉ số dương, vì thế ta sẽ đơn thuần lưu 2 mảng này dưới dạng mảng thường.

Gọi D là mảng hiệu của A . Để xử lý truy vấn cập nhật, ta chỉ cần cập nhật giá trị của mảng D tại 2 biên l và $r + 1$ bằng cách tăng D_l thêm 1 và trừ 1 khỏi D_{r+1} . Sau khi xử lý toàn bộ truy vấn cập nhật, ta sử dụng hệ thức $S(D, 0) = A$ để cập nhật lại giá trị của mảng A .

Để trả lời câu hỏi, ta có thể hình dung mỗi chỉ số trong đoạn $[a, b]$ mang giá trị 1 nếu thỏa điều kiện đề bài và 0 nếu không thỏa; số chỉ số thỏa mãn cũng chỉ là tổng của các phần tử nằm trong đoạn $[a, b]$. Từ tính chất này, ta có thể ứng dụng mảng cộng dồn để trả lời nhau câu hỏi trong $\mathcal{O}(1)$. Ta đặt mảng A' đánh dấu giá trị A_i tương ứng có thỏa điều kiện $\geq k$ không, rồi dựng mảng cộng dồn S trên A' . Đáp án của mỗi câu hỏi sẽ là kết quả của phép tính $S_b - S_{a-1}$.

Độ phức tạp thời gian và không gian của cách này đều là $\mathcal{O}(n)$. Code tham khảo:

```
1 | #include <bits/stdc++.h>
2 |
3 | using namespace std;
4 |
5 | const int MAXN = 200003;
6 |
7 | int n, k, q, d[MAXN], a[MAXN], s[MAXN];
8 |
9 | void update(int l, int r) {
10 |     ++d[l], --d[r + 1];
11 | }
12 |
```



```

13 void buildPrefixSum() {
14     a[0] = s[0] = 0;
15     for (int i = 1; i < MAXN; i++) {
16         a[i] = a[i - 1] + d[i];
17         s[i] = s[i - 1] + (a[i] >= k);
18     }
19 }
20
21 int query(int a, int b) {
22     return s[b] - s[a - 1];
23 }
24
25 int main() {
26     cin >> n >> k >> q;
27     for (int i = 0; i < n; i++) {
28         int l, r; cin >> l >> r;
29         update(l, r);
30     }
31     buildPrefixSum();
32     for (int i = 0; i < q; i++) {
33         int a, b; cin >> a >> b;
34         cout << query(a, b) << endl;
35     }
36 }

```

Mở rộng sang mảng nhiều chiều

Ta có thể mở rộng mảng cộng dồn và mảng hiệu để thao tác trên mảng nhiều chiều.

Mảng cộng dồn hai chiều

Cho mảng hai chiều A có kích thước $m \times n$ (chỉ số hàng và cột đầu tiên đều là 1), mảng cộng dồn $S(A)$ được dựng theo công thức sau:

$$S_{i,j} = \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i,t_j}$$

Các phần tử trong mảng cộng dồn lưu tổng của toàn bộ phần tử chứa trong hình chữ nhật $[1, i] \times [1, j]$.

Điểm khác biệt so với mảng cộng dồn 1 chiều ở đây là sự lược bỏ của hằng số C , ta ngầm quy ước: $S_{0,x} = S_{y,0} = 0$ với x, y nguyên không âm khi dựng mảng cộng dồn.

	1	2	3	4
1	3	-6	7	8
2	-7	9	4	-4
3	1	-1	7	1
4	4	4	-2	-3

A

	0	1	2	3	4
0	0	0	0	0	0
1	0	3	-3	4	12
2	0	-4	-1	10	14
3	0	-3	-1	17	22
4	0	1	7	23	25

S

Từ công thức quy ước trên, ta thực hiện biến đổi sau để dựng mảng cộng dồn:

$$\begin{aligned}
S_{i,j} &= \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i,t_j} \\
&= \sum_{t_i=1}^{i-1} \sum_{t_j=1}^{j-1} A_{t_i,t_j} + \sum_{t_i=1}^{i-1} A_{t_i,j} + \sum_{t_j=1}^{j-1} A_{i,t_j} + A_{i,j} \\
&= \sum_{t_i=1}^{i-1} \sum_{t_j=1}^{j-1} A_{t_i,t_j} + \sum_{t_i=1}^{i-1} A_{t_i,j} + \sum_{t_j=1}^{j-1} A_{i,t_j} + A_{i,j} \\
&= \sum_{t_i=1}^{i-1} \sum_{t_j=1}^{j-1} A_{t_i,t_j} + \left(\sum_{t_i=1}^{i-1} \sum_{t_j=1}^j A_{t_i,t_j} - \sum_{t_i=1}^{i-1} \sum_{t_j=1}^{j-1} A_{t_i,t_j} \right) + \left(\sum_{t_i=1}^i \sum_{t_j=1}^{j-1} A_{t_i,t_j} - \sum_{t_i=1}^{i-1} \sum_{t_j=1}^{j-1} A_{t_i,t_j} \right) + A_{i,j} \\
&= S_{i-1,j-1} + S_{i-1,j} - S_{i-1,j-1} + S_{i,j-1} - S_{i-1,j-1} + A_{i,j} \\
&= S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1} + A_{i,j}
\end{aligned}$$

Để hình dung rõ hơn công thức biến đổi trên, bạn đọc có thể tham khảo hình ảnh dưới:

	1	2	3	4
1	3	-6	7	8
2	-7	9	4	-4
3	1	-1	7	1
4	4	4	-2	-3

A

	0	1	2	3	4
0	0	0	0	0	0
1	0	3	-3	4	12
2	0	-4	-1	10	14
3					
4					

S

$$S_{i,j} = S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1} + A_{i-1,j-1}$$

Các phần tử A_i tô màu xanh nhạt được đánh dấu 1 lần, tô màu xanh đậm được đánh dấu tới 2 lần

Code dưới đây dựng mảng cộng dồn hai chiều:

```

1  vector< vector<int> > build2DPrefixSum(const vector< vector<int> >
2      int m = (int)a.size(), n = (int)a[0].size();
3
4      vector< vector<int> > prefixSum(m + 1, vector<int> (n + 1, 0))
5
6      for (int i = 1; i <= m; i++)
7          for (int j = 1; j <= n; j++)
8              prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1] + a[i - 1][j - 1];
9      return prefixSum;
10 }
```

Để tính tổng của toàn bộ các phần tử nằm trong hình chữ nhật có góc trái trên là (x_1, y_1) và góc phải dưới (x_2, y_2) , ta biến đổi tương tự để thu được công thức tính sau:

$$\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} A_{i,j} = S_{x_2, y_2} - S_{x_1-1, y_2} - S_{x_2, y_1-1} + S_{x_1-1, y_1-1}$$

Phần chứng minh công thức trên xin được nhường lại cho bạn đọc. Hình ảnh dưới minh họa vì sao công thức trên cho kết quả chính xác:

	1	2	3	4
1	3	-6	7	8
2	-7	9	4	-4
3	1	-1	7	1
4	4	4	-2	-3

A

	0	1	2	3	4
0	0	0	0	0	0
1	0	3	-3	4	12
2	0	-4	-1	10	14
3	0	-3	-1	17	22
4	0	1	7	23	25

S

	1	2	3	4
1	3	-6	7	8
2	-7	9	4	-4
3	1	-1	7	1
4	4	4	-2	-3

A

	0	1	2	3	4
0	0	0	0	0	0
1	0	3	-3	4	12
2	0	-4	-1	10	14
3	0	-3	-1	17	22
4	0	1	7	23	25

S

1. Cộng vào toàn bộ phần tử nằm trong hình chữ nhật

	1	2	3	4
1	3	-6	7	8
2	-7	9	4	-4
3	1	-1	7	1
4	4	4	-2	-3

A

	0	1	2	3	4
0	0	0	0	0	0
1	0	3	-3	4	12
2	0	-4	-1	10	14
3	0	-3	-1	17	22
4	0	1	7	23	25

S

2. Trừ các phần tử nằm trong hình chữ nhật

Các phần tử A_i tô màu đỏ bị trừ tới 2 lần, vì thế cần phải cộng bù lại

Mảng cộng dồn ba chiều


Giả sử ta có mảng A trong không gian 3 chiều với kích thước $m \times n \times p$, ta dựng mảng $S(A)$ theo quy tắc sau:
$$S_{i,j,k} = \sum_{t_i=1}^i \sum_{t_j=1}^j \sum_{t_k=1}^k A_{t_i,t_j,t_k}$$

Công thức sau được sử dụng để dựng mảng cộng dồn 3 chiều:

$$S_{t_i,t_j,t_k} = A_{t_i,t_j,t_k} + S_{t_i-1,t_j,t_k} + S_{t_i,t_j-1,t_k} + S_{t_i,t_j,t_k-1} - S_{t_i-1,t_j-1,t_k} - S_{t_i-1,t_j,t_k-1} - S_{t_i,t_j-1,t_k-1} + S_{t_i-1,t_j-1,t_k-1}$$

Tương tự, ta tính tổng các phần tử trong không gian $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ qua công thức sau:

$$\begin{aligned} \sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} \sum_{k=z_1}^{z_2} A_{i,j,k} &= S_{x_2,y_2,z_2} - S_{x_1-1,y_2,z_2} - S_{x_2,y_1-1,z_2} - S_{x_2,y_2,z_1-1} \\ &\quad + S_{x_1-1,y_1-1,z_2} + S_{x_1-1,y_2,z_1-1} + S_{x_2,y_1-1,z_1-1} - S_{x_1-1,y_1-1,z_1-1} \end{aligned}$$

Hai công thức trên được xây dựng thông qua phương pháp bao hàm - loại trừ. Bạn đọc có thể tham khảo bài viết [Bao hàm - Loại trừ](#)  trên VNOI Wiki để hiểu rõ hơn lý do có được công thức trên.

Ta cũng có thể áp dụng phương pháp này để mở rộng cho các mảng n -chiều. Tuy nhiên, do số lượng bài toán liên quan đến mảng trong không gian từ 4 chiều trở lên là cực hiếm, mảng cộng dồn trong không gian này gần như không có ứng dụng thực tiễn. Vì thế, bài viết xin giới hạn lại tại mảng cộng dồn trong không gian 3 chiều trở xuống.

Mảng hiệu hai chiều

Trước khi bắt đầu xây dựng mảng hiệu 2 chiều, ta cần định nghĩa thêm 2 khái niệm sau cho một mảng A hai chiều có kích thước $m \times n$:

- ▶ $D_{hàng}(A)$ gồm m hàng, hàng thứ i biểu thị mảng hiệu của mảng gồm toàn bộ phần tử nằm trên hàng đó.
- ▶ $D_{cột}(A)$ gồm n cột, cột thứ i biểu thị mảng hiệu của mảng gồm toàn bộ phần tử nằm trên cột đó.

Trong không gian 1 chiều, ta thấy được $S(D(A)) = A$. Để tính chất này được áp dụng cho mảng 2 chiều, ta tạo mảng A' thỏa: $A'_{i,j} = A_{i,j}$ với i, j dương và $A_{i,j} = 0$ với $i = 0$ hoặc $j = 0$. Mảng A' này đánh số theo dạng 0-indexed và có kích thước là $(m+1) \times (n+1)$. Khi này, mảng hiệu của A sẽ là mảng D thỏa $S(D) = A'$.

Đặt $D = D_{cột}(D_{hàng}(A))$, khi này, ta có:

$$\begin{aligned} D_{i,j} &= [D_{cột}(D_{hàng}(A))]_{i,j} \\ &= [D_{hàng}(A_i)]_j - [D_{hàng}(A_{i-1})]_j \\ &= (A_{i,j} - A_{i,j-1}) - (A_{i-1,j} - A_{i-1,j-1}) \\ &= A_{i,j} - A_{i,j-1} - A_{i-1,j} + A_{i-1,j-1} \end{aligned}$$

Mảng $S(D)$ có giá trị như sau:

$$\begin{aligned}
S(D)_{i,j} &= \sum_{t_i=1}^i \sum_{t_j=1}^j D_{t_i,t_j} \\
&= \sum_{t_i=1}^i \sum_{t_j=1}^j (A_{t_i,t_j} - A_{t_i,t_j-1} - A_{t_i-1,t_j} + A_{t_i-1,t_j-1}) \\
&= \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i,t_j} - \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i,t_j-1} - \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i-1,t_j} + \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i-1,t_j-1} \\
&= \sum_{t_i=1}^i \sum_{t_j=1}^j A_{t_i,t_j} - \sum_{t_i=1}^i \sum_{t_j=1}^{j-1} A_{t_i,t_j} - \sum_{t_i=1}^{i-1} \sum_{t_j=1}^j A_{t_i,t_j} + \sum_{t_i=1}^{i-1} \sum_{t_j=1}^{j-1} A_{t_i,t_j} \\
&= S_{i,j} - S_{i-1,j} - S_{i,j-1} + S_{i-1,j-1} \\
&= A'_{i,j}
\end{aligned}$$

Ta kết luận rằng $S(D) = A'$, mảng D ta vừa dựng chính là mảng hiệu của A .

	1	2	3	4
1	3	-6	7	8
2	-7	9	4	-4
3	1	-1	7	1
4	4	4	-2	-3

A

	1	2	3	4
1	3	-9	13	1
2	-7	16	-5	-8
3	1	-2	8	-6
4	4	0	-6	-1

$D_{\text{hàng}}(A)$

	1	2	3	4
1	3	-9	13	1
2	-7	16	-5	-8
3	1	-2	8	-6
4	4	0	-6	-1

	1	2	3	4
1	3	-9	13	1
2	-7	16	-5	-8
3	1	-2	8	-6
4	4	0	-6	-1

Từ các quan sát trên, ta có thể dựng mảng hiệu của A bằng hai cách:

- Sử dụng trực tiếp công thức:

$$D_{i,j} = A_{i,j} - A_{i,j-1} - A_{i-1,j} + A_{i-1,j-1}$$

- Tính $D_{\text{hàng}}$ cho từng hàng của A và gán kết quả vào A' , sau đó tính $D_{\text{cột}}$ cho từng cột của A' .

Code dưới đây dựng mảng hiệu 2 chiều D theo A theo cách thứ nhất:

```
1  vector< vector<int> > build2DDifferenceArray(const vector< vector<
2      int m = (int)a.size(), n = (int)a[0].size();
3      vector< vector<int> > differenceArray(m, vector<int>(n, 0)));
4
5      for (int i = 0; i < m; i++) {
6          for (int j = 0; j < n; j++) {
7              differenceArray[i][j] = a[i][j];
8              if (i > 0) differenceArray[i][j] -= a[i - 1][j];
9              if (j > 0) differenceArray[i][j] -= a[i][j - 1];
10             if (i > 0 && j > 0) differenceArray[i][j] += a[i - 1][
11         }
12     }
13
14     return differenceArray;
15 }
```

Quay lại bài toán cũ trong không gian 1 chiều: làm thế nào để ta tăng thêm một lượng k lên toàn bộ phần tử trong lưới $[r_1, r_2] \times [c_1, c_2]$ một cách hiệu quả? Khi ta tính $D_{hàng}$ cho từng hàng của A , ta nhận thấy chỉ giá trị của phần tử ở biên trái (tại c_1) và biên phải (tại $c_2 + 1$). Tiếp tục tính $D_{cột}$ cho từng cột của mảng $D_{hàng}(A)$, quan sát tương tự cho thấy chỉ phần tử tại r_1 và $r_2 + 1$ sẽ bị tác động bởi thao tác cập nhật này.


Từ nhận xét trên, ta thấy chỉ có 4 phần tử của $D(A)$ sẽ bị tác động bởi thao tác này là tọa độ (r_1, c_1) , $(r_1, c_2 + 1)$, $(r_2 + 1, c_1)$ và $(r_2 + 1, c_2 + 1)$ - trong đó, phần tử tại (r_1, c_1) và $(r_2 + 1, c_2 + 1)$ tăng thêm lượng k , phần tử tại $(r_1, c_2 + 1)$ và $(r_2 + 1, c_1)$ trừ đi lượng k . Ta dễ dàng cập nhật đoạn trên mảng hai chiều với độ phức tạp $\mathcal{O}(1)$.

Mảng hiệu ba chiều

Cũng như mảng cộng dồn, ta cũng có thể dựng mảng hiệu của các mảng trong không gian 3 chiều. Tương tự, nếu ta coi D là mảng sinh ra mảng cộng dồn A , ta có công thức dựng D sau:

$$D_{i,j,k} = A_{i,j,k} - A_{i-1,j,k} - A_{i,j-1,k} - A_{i,j,k-1} \\ + A_{i-1,j-1,k} + A_{i-1,j,k-1} + A_{i,j-1,k-1} - A_{i-1,j-1,k-1}$$

Để xử lý truy vấn cập nhật toàn bộ phần tử trong không gian $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$, ta cần cập nhật giá trị tại 8 vị trí, các vị trí đều nằm tại biên của không gian. Nếu ta coi mảng n -chiều như một hình lập phương chứa số, vị trí cần cập nhật sẽ tương ứng với đỉnh của hình lập phương đại diện cho không gian cần cập nhật. Ta sẽ tô xen kẽ các đỉnh này theo hai màu đen - trắng, đỉnh có tọa độ (x_1, y_1, z_1) được tô màu trắng. Phần tử D ứng với các đỉnh trắng được cộng thêm lượng k , ngược lại, phần tử ứng với đỉnh đen thì trừ đi lượng k .


Hình trên minh họa những vị trí mà ta cần cập nhật trên mảng hiệu. Tương tự mảng cộng dồn, phương pháp [bao hàm - loại trừ](#)  được áp dụng để đưa đến kết luận này.



Mở rộng sang mảng động

Trong các ví dụ đã đề cập, các bài toán chúng ta phải giải đều không có truy vấn cập nhật hoặc toàn bộ truy vấn cập nhật được thực hiện trước truy vấn hỏi. Tuy nhiên, trong một số bài toán yêu cầu phải thực hiện xen kẽ hai loại truy vấn này, ta cần sử dụng các cấu trúc dữ liệu để giải quyết hiệu quả các truy vấn này.

Có hai dạng bài toán liên quan đến mảng cộng dồn và mảng hiệu:

- **Dạng 1:** Cập nhật giá trị của A_i hoặc tính tổng của i phần tử đầu tiên.
- **Dạng 2:** Cập nhật toàn bộ giá trị từ A_i đến A_j ($i \leq j$) hoặc cho biết giá trị hiện tại của A_i .

Nếu bài toán chỉ xử lý một trong hai dạng nói trên, ta có thể áp dụng cấu trúc dữ liệu [Binary Indexed Tree](#)  để giải quyết các truy vấn trên. Độ phức tạp của từng truy vấn sẽ phụ thuộc vào số chiều của mảng, thí dụ, thao tác trên mảng 1D sẽ cho độ phức tạp $\mathcal{O}(\log n)$ còn trên mảng 2D sẽ là $\mathcal{O}(\log^2 n)$.

Trong một số bài toán yêu cầu xử lý kết hợp 2 dạng (cập nhật đoạn và tính tổng đoạn), ta thường áp dụng [Segment Tree](#)  có lazy propagation (cập nhật lười). Mặc dù có chung độ phức tạp, cách cài đặt này thường khó hơn, có thời gian chạy lâu hơn và dùng nhiều bộ nhớ hơn so với cài đặt Binary Indexed Tree. Nếu ta làm việc trên mảng 1 chiều, ta cũng có thể biến đổi hệ thức giữa mảng hiệu và mảng cộng dồn để cài đặt trực tiếp BIT làm việc trên các truy vấn này. Bạn đọc có thể tham khảo thêm cách cài đặt này tại [đây](#) .

Bài tập

Mảng cộng dồn 1 chiều

[VNOJ - NKSEQ](#) 

[CSES - Subarray Sums II](#) 

[CSES - Subarray Divisibility](#) 



[Codeforces - Good Subarrays](#) 

[Codechef - XXOR](#) 

Mảng hiệu 1 chiều

[Codeforces - Little Girl and Maximum Sum](#) 

[Codeforces - Greg and Array](#) 

[Codeforces Gym - 319055E](#)  (lưu ý: để xem nội dung bài tập cần tham gia nhóm tại [link đây](#) )

Mảng cộng dồn nhiều chiều

CSES - Forest Queries [↗](#)

USACO - The Lazy Cow [↗](#)

USACO - Painting the Barn [↗](#)

VNOJ - MAXCUB [↗](#)

Mảng hiệu nhiều chiều

Codechef - COW3E [↗](#)

WCIPEG - CAKE [↗](#)

VNOJ có phân loại riêng các bài tập về mảng cộng dồn, bạn đọc có thể tham khảo tại [đây](#) [↗](#) .

References

WCIPEG - Prefix sum array and difference array [↗](#)

cppreference.com - partial_sum [↗](#)

cppreference.com - adjacent_difference [↗](#)

Được cung cấp bởi [Wiki.js](#)