Monotonic Stack - Monotonic Queue

The Gifted Battlefield

15 - 09 - 2023

Mục lục

1 Monotonic Stack

Monotonic stack là 1 loại CTDL dùng để lưu trữ 1 dãy số tăng/giảm dần , nó được sử dụng để tối ưu nhiều bài toán liên quan tới thao tác tìm kiếm phần tử gần nhất thoả mãn 1 yêu cầu gì đó. Trong phần dưới đây , bài viết sẽ giới thiệu qua cách hoạt động cũng như các bài tập vận dung cơ bản của CTDL này.

1.1 Bài toán áp dụng

Link đề bài :Nearest Smaller Values
Tóm tắt đề: Cho mảng A gồm n phần tử. Với mỗi A_i, tìm 1 A_j sao cho $1 \leq j < i$, $A_j < A_i$ và j lớn nhất có thể.

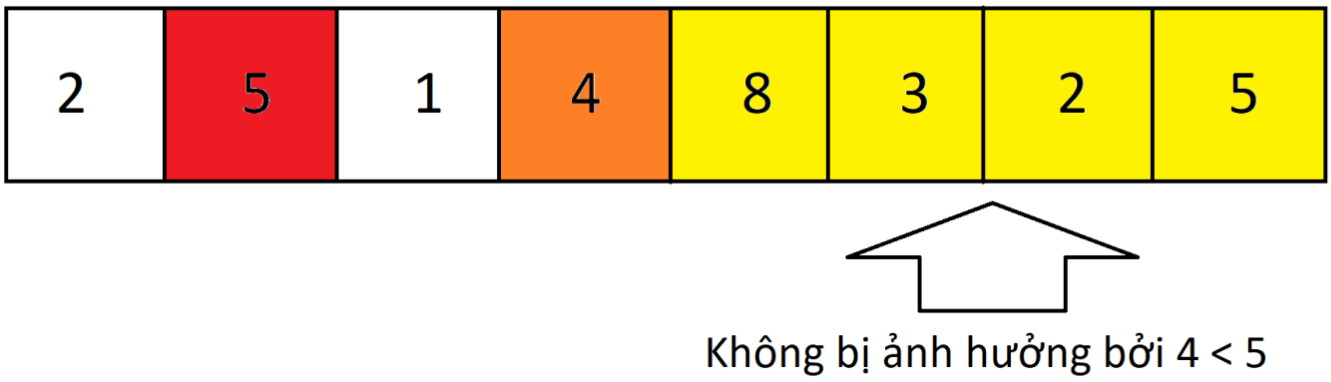
Thuật toán thô thiển

Với mỗi A_i , ta thấy có thể chạy thêm 1 vòng lặp ngược về đầu mảng để tìm kiếm giá trị thoả yêu cầu đề bài. Code mẫu:

```
for (int i = 1; i <= n; ++i) {
    bool found = 0;
    for (int j = i - 1; j > 0 && found == 0; --j) {
        if (arr[j] < arr[i]) {
            cout << j << " ";
            found = 1;
        }
    }
    if (found == 0) {
        cout << 0 << " ";
    }
}
```

Thuật toán chuẩn

Ta có thể thấy cách làm trên vô cùng đơn giản, dễ cài đặt nhưng lại có độ phức tạp về thời gian là $O(N^2)$, sẽ kém hiệu quả đối với những dữ liệu N lớn.
Để giải quyết bài toán này hiệu quả hơn , ta có nhận xét sau :
- Gọi ans_i là đáp án của bài toán tại vị trí i , ta thấy tại i bất kì , tất cả giá trị A_j sao cho $1 \leq j < i$ và $A_j > A_i$ sẽ không ảnh hưởng tới bất kì ans_k nào với $k > i$ nữa. Hình mình họa:



- Sử dụng stack hay bất kì CTDL nào hỗ trợ thao tác **Last in First out (LIFO)** với độ phức tạp thời gian $O(1)$, ta sẽ liên tục đẩy các giá trị vừa duyệt qua lên trên cùng CTDL.
- Với mỗi A_i, ta có thể xoá hết những giá trị ở trên cùng của CTDL nếu chúng có giá trị lớn hơn A_i hiện tại. Vị trí đầu tiên trong CTDL có giá trị nhỏ hơn A_i chính là đáp án cho bài toán tại vị trí i.
- Vì ta liên tục xoá đi các giá trị trên cùng của CTDL và thay thế nó bằng các giá trị khác lớn/nhỏ hơn nên CTDL duy trì được tính đơn điệu , nghĩa là nó sẽ tăng/giảm dần từ dưới đáy lên trên.
- Vì mỗi phần tử trong mảng được đẩy vào và xoá đi tối đa 1 lần , ta đạt được độ phức tạp cuối cùng là $O(N)$, hiệu quả hơn nhiều so với cách duyệt ban đầu.
Code mẫu :

```
int n;
stack<pair<int, int>> st;
cin >> n;
for (int i = 1, a; i <= n; ++i) {
    cin >> a;
    while (st.size() > 0 && st.top().first >= a) {
        st.pop();
    }
    if (st.empty() == true) {
        cout << 0 << " ";
    } else {
        cout << st.top().second() << " ";
    }
}
```

2 Monotonic Queue và bài toán min/max tịnh tiến

2.1 Bài toán áp dụng

Link đề bài: MINK

Tóm tắt đề: Cho mảng A gồm n phần tử và một số k, với mỗi i trong khoảng $[1, n-k+1]$, hãy tính $min(A_i, A_{i+1}, A_{i+2}, ..., A$

Cách làm cơ bản

Ta có thể dễ dàng làm bài này với độ phức tạp $O(N * K)$ bằng cách xét hết tất cả các khoảng và tính min tương ứng.

Code mẫu:

```
int n,k;
int arr[17001];
void solve() {
    cin >> n >> k;
    for (int i = 1; i <= n; i++) cin>>arr[i];;
    for (int i = 1; i <= n - k + 1; i++) {
        int mn = INT_MAX;
        for (int j = 0; j < k; j++) {
            mn = min(mn, arr[i + j]);
        }
        cout << mn << ' ';
    }
    cout << "\n";
}
```

Thuật toán chuẩn

Tất nhiên, ta có thể sử dụng các cấu trúc dữ liệu như *segment tree* hay *sparse table* để giải bài toán này trong độ phức tạp $O(NlogN)$

Nhưng, team TGB sẽ giới thiệu cho bạn các giải bài này trong độ phức tạp là $O(N)$ bằng cách sử dụng Monotonic

Queue.
Mặc dù gọi là Monotonic Queue nhưng nó sẽ được cài đặt bằng Deque. Monotonic Queue hoạt động dựa trên nguyên lý của kỹ thuật cửa sổ trượt. Vì thế, nó cũng sẽ có những áp dụng giống cửa sổ trượt. Rõ hơn, nó sẽ dùng trong các bài toán là min/max tịnh tiến hoặc các bài toán dùng hai con trỏ để tìm min/max.

Phần này sẽ giải quyết bài toán min tịnh tiến, bài toán max tịnh tiến bạn đọc có thể tự suy ra từ phần này. Giả sử, ta đang có đáp án là min của một đoạn $[l, r]$ thì làm sao để ta có được đáp án của đoạn $[l+1, r+1]$? Nếu đây là bài toán tổng và ta dùng kỹ thuật cửa sổ trượt thì ta sẽ trừ đi $A[l]$ và cộng vào $A[r+1]$. Nhưng vì đây là ta tìm min và nó không có tính giao hoán nên ta không làm vậy được.

Nếu ta xem Monotonic Queue của ta làm một Monotonic Stack thì việc thêm vào phần tử $A[r+1]$ sẽ cực kỳ đơn giản. Ngoài ra, ta cũng có thể nhận xét là nếu ta xem Monotonic Queue của ta là một Monotonic Stack thì các phần tử trong Queue sẽ tăng dần, hay phần tử đầu tiên của Queue sẽ là min của đoạn ta xét. Nhưng làm cách nào để ta loại bỏ phần tử $A[l]$ khỏi Monotonic Queue của ta. Đây là lúc ta cần sử dụng cấu trúc dữ liệu *Deque*. *Deque* cho ta xóa bỏ và thêm phần tử vào hai đầu của một queue, vì thế, nếu phần tử đầu tiên của Deque là phần tử ở vị trí l thì ta sẽ xóa nó đi và ta chắc chắn sẽ được đáp án của cửa sổ $[l+1, r+1]$.

Để hiểu rõ hơn thì ta hãy xét ví dụ sau.
$n = 8, k = 4$
$A = [1, 3, 5, 7, 4, 5, 9, 5]$

Lưu ý, trong phần này thì Deque sẽ chứa vị trí của các phần tử chứ không chứa giá trị còn khi cài đặt thì người đọc có thể dùng kiểu pair để thuận tiện cài đặt. Ngoài ra, mảng đáp án sẽ gọi là ans.
Gọi monotonic queue của ta là q.

Ban đầu: $q = []$
tương ứng với $l = 0, r = 0$.

Khi ta xét đoạn đầu tiên là $l = 1, r = 4$ thì:
$q = [1, 2, 3, 4]$
nên đáp án của đoạn $[1, 4]$ sẽ là $A[1]$ và $ans = [1]$.

Khi ta thêm phần tử $r = 5$ thì q của ta sẽ trở thành
$q = [1, 2, 5]$
rồi nếu ta xóa phần tử $l = 1$ đi thì q của ta sẽ trở thành
$q = [2, 5]$
vì thế, đáp án của đoạn $[2, 5]$ sẽ là $A[2]$ và $ans = [1, 3]$

Khi ta thêm phần tử $r = 6$ thì q của ta sẽ trở thành
$q = [2, 5, 6]$
rồi nếu ta xóa phần tử $l = 2$ đi thì q của ta sẽ trở thành
$q = [5, 6]$
vì thế, đáp án của đoạn $[3, 6]$ sẽ là $A[5]$ và $ans = [1, 3, 4]$

Khi ta thêm phần tử $r = 7$ thì q của ta sẽ trở thành
$q = [5, 6, 7]$

rồi nếu ta xóa phần tử $l = 3$ đi thì q của ta sẽ trở thành
$q = [5, 6, 7]$
vì thế, đáp án của đoạn $[4, 7]$ sẽ là $A[5]$ và $ans = [1, 3, 4, 4]$

Khi ta thêm phần tử $r = 8$ thì q của ta sẽ trở thành
$q = [5, 6, 8]$
rồi nếu ta xóa phần tử $l = 4$ đi thì q của ta sẽ trở thành
$q = [5, 6, 8]$
vì thế, đáp án của đoạn $[5, 8]$ sẽ là $A[5]$ và $ans = [1, 3, 4, 4, 4]$

Nếu ta check lại thử bằng tay thì ta thấy đáp án $ans = [1, 3, 4, 4, 4]$ đúng.

Ta thấy, với mỗi phần tử thì nó sẽ được cho vào deque một lần và bị bỏ ra nhiều nhất 1 lần và việc cho vô và bỏ ra một phần tử trong deque có độ phức tạp là $O(1)$ và ta sẽ bỏ vô tất cả là N phần tử nên độ phức tạp của thuật toán này là $O(N)$.
Cài đặt cho bài toán MINK:

```
deque <int> dq;
dq={};
for (int i = 1; i <= N; ++i) {
    while (dq.size() && A[dq.back()] >= A[i]) dq.pop_back();

    dq.push_back(i);

    if (dq.front() + k <= i) dq.pop_front();

    if (i >= k) ans[i] = A[dq.front()];
}
```

Nhận xét chung

Monotonic Queue là một kỹ thuật khá hữu dụng trong lập trình thi đấu. Dù vậy, áp dụng của nó khá là hạn hẹp và nó có nhược điểm là nó chỉ áp dụng cho việc tính min/max tịnh tiến và không có cập nhật. Nên khi so với *segment tree* hay *sparse table* thì nó không thể so sánh bằng được. Dưới đây là bảng so sánh 4 thuật toán làm bài này là vét cạn, segment tree, sparse table, Monotonic Queue.

Thuật toán	Độ phức tạp	Ưu điểm	Nhược điểm
Vét cạn	$O(NK)$	Code ngắn	Chậm
Segment tree	$O(N \log N)$	Cập nhật được và không nhất thiết là tịnh tiến	Cài khá dài
Sparse table	$O(N \log N)$	Không nhất thiết là tịnh tiến và cài ngắn	Không cập nhật được
Monotonic queue	$O(N)$	Nhanh nhất trong 4 thuật	Bắt buộc phải là tịnh tiến và không cập nhật được

3 Monotonic Stack vs Monotonic Queue

Monotonic Stack và Monotonic Queue đều được dùng để lưu trữ 1 giá trị max / min của 1 dãy số . Tuy vậy nhưng cách cả hai loại CTDL hoạt động và những bài tập được hướng đến đều có những khác biệt đáng kể. Ta có bảng so sánh sau :

	Montonic Stack	Monotonic Queue
Cách thức cài đặt	Sử dụng các CTDL hỗ trở thao tác Last in First out (LIFO)	Sử dụng các CTDL dạng hàng đợi hỗ trợ thao tác thêm vào / loại bỏ các giá trị ở hai đầu
Công dụng	Giải quyết các bài toán liên quan tới "tìm 1 phần tử gần nhất"trên 1 dãy số	Giải quyết các bài toán tìm max / min trên 1 đoạn tịnh tiến của 1 dãy số
Cách lưu trữ phần tử	Dưới dạng đơn điệu tăng / giảm dần bên trong CTDL	

Qua bảng so sánh trên, ta thấy cả hai kỹ thuật đều có khả năng tối ưu các bài toán liên quan đến dãy số khác nhau, và việc thành thạo hai kỹ thuật này sẽ giúp ích rất nhiều trong quá trình học Tin sau này.

4 Bài tập vận dụng

Lưu ý, bấm vào mã bài để hiện đề.

4.1 Monotonic Stack

KPLANK
Discrete Centrifugal Jump

4.2 Monotonic Queue

STOCKS
CLIMBING

5 Lời giải

5.1 KPLANK

Với mỗi i từ 1 đến N, nếu ta xét hình vuông có cạnh là $A[i]$ thì việc ta cần làm chỉ là tìm xem vị trí đầu tiên bên phải và bên trái bé hơn $A[i]$ thôi. Giả sử hai vị trí đó là l_i và r_i, khi đó nếu $r_i - l_i - 1 >= A[i]$ thì ta có thể tạo hình vuông có cạnh là $A[i]$ và ta có thể cập nhật đáp án tương ứng. Việc tìm l_i và r_i có thể dễ dàng tìm bằng cách dùng Monotonic Stack. Độ phức tạp của thuật toán $O(N)$. Code mẫu:

```
#include<bits/stdc++.h>
#define ll long long
using namespace std;
signed main() {
    ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    ll n;
```

```
    cin >> n;
    ll arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    vector<ll> l(n), r(n);
    stack<ll> st;
    for (int i = 0; i < n; i++) {
        while (!st.empty() && arr[st.top()] >= arr[i]) {
            st.pop();
        }
        if (st.empty()) {
            l[i] = -1;
        } else {
            l[i] = st.top();
        }
        st.push(i);
    }
    st={};
    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && arr[st.top()] >= arr[i]) {
            st.pop();
        }
        if (st.empty()) {
            r[i] = n;
        } else {
            r[i] = st.top();
        }
        st.push(i);
    }
    st={};
    ll ans=0;
    for (int i = 0; i < n; i++) {
        if (r[i] - l[i] - 1 >= arr[i]) ans = max(ans, arr[i]);
    }
    cout<<ans;
    return 0;
}
```

5.2 Discrete Centrifugal Jump

Ta sẽ xem bài này là một bài toán đồ thị. Hiển nhiên, sẽ có cạnh từ i đến $i+1$.

Gọi R_i, L_i, r_i, l_i lần lượt là vị trí gần nhất bên phải lớn hơn $A[i]$, vị trí gần nhất bên trái lớn hơn $A[i]$, vị trí gần nhất bên phải nhỏ hơn $A[i]$ và vị trí gần nhất bên trái nhỏ hơn $A[i]$. Khi đó, ta sẽ có một cạnh nối một chiều từ L_i đến R_i và một cạnh nối một chiều từ l_i đến r_i. Tại sao điều này đúng, theo định nghĩa, mọi phần tử từ $l_i + 1$ đến i sẽ lớn hơn $A[i]$, mọi phần tử từ i đến $r_i - 1$ sẽ lớn hơn $A[i]$ nên sẽ thỏa được điều kiện 3. Tương tự với L_i, R_i và điều kiện 2. Như thế, ta đã chuyển bài toán về tìm đường đi ngắn nhất từ 1 đến N trên đồ thị có hướng. Bài toán này có thể giải bằng BFS hoặc Quy hoạch động trên đồ thị có hướng không vòng lặp trong độ phức tạp là $O(N + M)$. Nhưng, ta nhận xét là với mỗi phần tử i, sẽ có nhiều nhất là 3 cạnh đi ra từ nó tương ứng với 3 trường hợp. Nên $M \leq 3 * N$ và độ phức tạp của chúng ta chỉ à $O(N)$.

Code mẫu:

```
#include<bits/stdc++.h>

#define ll long long
using namespace std;
signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    ll n;
    cin >> n;
    ll arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    vector < ll > L(n), l(n), R(n), r(n);
    stack < ll > st;
    for (int i = 0; i < n; i++) {
        while (!st.empty() && arr[st.top()] >= arr[i]) {
            st.pop();
        }
        if (st.empty()) {
            l[i] = -1;
        } else {
            l[i] = st.top();
        }
        st.push(i);
    }
    st = {};
    for (int i = 0; i < n; i++) {
        while (!st.empty() && arr[st.top()] <= arr[i]) {
            st.pop();
        }
        if (st.empty()) {
            L[i] = -1;
        } else {
            L[i] = st.top();
        }
        st.push(i);
    }
    st = {};
    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && arr[st.top()] >= arr[i]) {
            st.pop();
        }
        if (st.empty()) {
            r[i] = -1;
        } else {
            r[i] = st.top();
        }
```

```
        st.push(i);
    }
    st = {};
    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && arr[st.top()] <= arr[i]) {
            st.pop();
        }
        if (st.empty()) {
            R[i] = -1;
        } else {
            R[i] = st.top();
        }
        st.push(i);
    }
    st = {};
    vector < ll > adj[n];
    for (int i = 0; i < n - 1; i++) {
        adj[i].push_back(i + 1);
    }
    for (int i = 0; i < n; i++) {
        if (R[i] != -1 && L[i] != -1) {
            adj[L[i]].push_back(R[i]);
            //adj[R[i]].push_back(L[i]);
        }
        if (r[i] != -1 && l[i] != -1) {
            adj[l[i]].push_back(r[i]);
            //adj[r[i]].push_back(l[i]);
        }
    }
    queue < ll > q;
    vector < ll > check(n, 0), dis(n, 1e18);
    q.push(0);
    dis[0] = 0;
    while (!q.empty()) {
        ll u = q.front();
        q.pop();
        if (check[u] == 1) continue;
        check[u] = 1;
        for (auto i: adj[u]) {
            q.push(i);
            dis[i] = min(dis[i], dis[u] + 1);
        }
    }
    cout << dis.back();
    return 0;
}
```

5.3 STOCKS

Với bài này, nếu ta dùng các cấu trúc dữ liệu như segment tree hay Sparse table thì chắc chắn sẽ không qua được tất cả các test do độ phức tạp là $O(NlogN)$ và $N \leq 3 * 10^6$. Vì thế, ta phải sử dụng một thuật toán $O(N)$. Khi

đó, ta sẽ có nhận xét sau: $max - min$ của một đoạn chỉ có thể tăng lên khi ta tăng hai biên chứ không thể nào giảm đi được. Lí do của việc này khá dễ nhìn thấy. Khi ta tăng hai biến của một đoạn lên, max chỉ có thể tăng và min chỉ có thể giảm nên $max - min$ chỉ có thể tăng thôi. Vì thế, ta có thể sử dụng kỹ thuật hai con trỏ cho bài toán này. Nhưng như ta đã biết được từ phần Monotonic Queue, kỹ thuật hai con trỏ trên max và min thì ta có thể sử dụng Deque. Vì thế, ta có thuật toán như sau:

Ta xét các r từ 1 đến N là biên phải của đoạn ta xét. Ta sẽ duy trì song song một con trỏ l là biên trái của đoạn ta xét sao cho nó cũng là vị trí đầu tiên mà $max - min$ của đoạn $[l, r]$ không quá t. Nếu $max - min$ quá t, ta sẽ tăng l lên và thay đổi deque tương ứng (xem phần Monotonic Queue). Sau đó, cập nhật đáp án lại là $ans = max(ans, r - l + 1)$.

Code mẫu:

```
#include<bits/stdc++.h>

using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int n, t;
    cin >> t >> n;
    int arr[n + 1];
    for (int i = 1; i <= n; i++) cin >> arr[i];
    deque < pair < int, int >> mn, mx;
    int ans = 0;
    int p = 0;
    for (int i = 1; i <= n; i++) {
        while (!mx.empty()) {
            if (mx.back().first < arr[i]) {
                mx.pop_back();
            } else {
                break;
            }
        }
        while (!mn.empty()) {
            if (mn.back().first > arr[i]) {
                mn.pop_back();
            } else {
                break;
            }
        }
        mx.push_back({arr[i], i});
        mn.push_back({arr[i], i});
        while (!mn.empty() && !mx.empty()) {
            if (mx.front().first - mn.front().first <= t) {
                break;
            }
            p++;
            if (mn.front().second <= p) mn.pop_front();
```

```
            if (mx.front().second <= p) mx.pop_front();
        }
        ans = max(ans, i - p);
    }
    cout << ans;
}
```

5.4 CLIMBING

Bài này cơ bản chỉ là tìm min,max của tịnh tiến của đoạn độ dài k rồi kiểm tra xem $max - min$ có quá t hay không thôi. Bài này ta có thể làm trong $O(N)$ với Monotonic Queue.

```
#include <bits/stdc++.h>
#define ll long long

using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    ll n, k, c;
    cin >> n >> k >> c;
    ll arr[n];
    for (int i = 0; i < n; i++) cin >> arr[i];
    deque<pair<ll, ll>> mx, mn;
    ll cnt = 0;
    for (int i = 0; i < k; i++) {
        while (!mx.empty()) {
            if (mx.back().first < arr[i]) {
                mx.pop_back();
            } else {
                break;
            }
        }
        mx.push_back({arr[i], i});
        while (!mn.empty()) {
            if (mn.back().first > arr[i]) {
                mn.pop_back();
            } else {
                break;
            }
        }
        mn.push_back({arr[i], i});
    }

    if (mx.front().first - mn.front().first <= c) {
        cout << 1 << ntr;
        cnt++;
    }
```

```
    for (int i = k; i < n; i++) {
        while (!mx.empty()) {
            if (mx.back().first < arr[i]) {
                mx.pop_back();
            } else {
                break;
            }
        }
        mx.push_back({arr[i], i});

        while (!mn.empty()) {
            if (mn.back().first > arr[i]) {
                mn.pop_back();
            } else {
                break;
            }
        }
        mn.push_back({arr[i], i});
        if (mn.front().second <= i - k) mn.pop_front();
        if (mx.front().second <= i - k) mx.pop_front();
        if (mx.front().first - mn.front().first <= c) {
            cout << i - k + 2 << ntr;
            cnt++;
        }
    }
    if (cnt == 0) {
        cout << "NONE";
    }
}
```