2940.Find Building Where Alice and Bob Can Meet

Huinan Chen

2024 年 12 月 27 日

问题描述

给定一个由正整数构成的数组 `heights`，其中 `heights[i]` 表示第 i 栋建筑的高度。

如果一个人处于建筑 i，他们可以移动到另一个建筑 j，当且仅当 $i < j$ 且 `heights[i]` $<$ `heights[j]`。

你还给定了一个二维数组 `queries`，其中 `queries[i] = [ai, bi]` 表示在第 i 个查询中，Alice 在建筑 ai 处，Bob 在建筑 bi 处。

返回一个数组 `ans`，其中 `ans[i]` 表示 Alice 和 Bob 可以相遇的最左侧建筑的索引。如果 Alice 和 Bob 无法相遇，则 `ans[i]` = -1。给定建筑物的高度数组 `heights` 和若干查询 `queries`，每个查询表示 Alice 和 Bob 分别从不同的建筑出发，要求找到最左侧的建筑，使得 Alice 和 Bob 都能移动到该建筑。我们可以利用单调栈和排序的方法来优化查询处理。

算法分析

- **单调栈：**我们使用单调栈来存储建筑的高度，保证栈中建筑的高度是递减的。这有助于我们高效地找出 Alice 和 Bob 能够相遇的最左建筑。
- **查询排序：我们首先按照查询中 Bob 的建筑高度排序，然后从右到左处理这些查询。这样我们可以一次性处理所有能到达的建筑。**
- **二分查找：**使用 `upper_bound` 函数来在栈中找到符合条件的最左建筑，进一步加速查找过程。

算法步骤

1. 初始化答案数组 `ans`，并对查询进行预处理。对于每个查询，如果 Alice 和 Bob 在同一个建筑上，直接返回该建筑；如果 Alice 的建筑高度小于 Bob 的建筑高度，则直接返回 Bob 的建筑。
2. 对剩余的查询按 Bob 的建筑高度降序排序。
3. 使用单调栈来处理建筑，从右到左遍历，更新栈中存储的建筑，并查找最左侧可以相遇的建筑。
4. 对每个查询，利用栈中的信息，查找符合条件的建筑，并更新答案数组。

代码实现

示例：

```
class Solution {
public:
    using int2 = pair<int, int>;
    vector<int> leftmostBuildingQueries(vector<int>& heights,
                                        vector<vector<int>>& queries) {
        int n = heights.size(), qz = queries.size();
        vector<int> ans(qz, -1);
        vector<int2> idx;

        for (int i = 0; i < qz; i++) {
            int& x = queries[i][0], & y = queries[i][1];
            if (x > y) // let x <= y
                swap(x, y);
            if (x == y || heights[x] < heights[y])
                ans[i] = y;
            else idx.emplace_back(y, i);
        }

        sort(idx.begin(), idx.end(), greater<>());
        vector<int2> stack;

        int j = n - 1;
        for (auto [_, i] : idx) {
            int x = queries[i][0];
            int y = queries[i][1];
            for (; j > y; j--) {
                while (!stack.empty() && heights[stack.back().second] < heights[j])
                    stack.pop_back();
                stack.emplace_back(heights[j], j);
            }

            auto it = upper_bound(stack.rbegin(), stack.rend(), make_pair(heights[x], n));
            ans[i] = (it == stack.rend()) ? -1 : it->second;
        }
        return ans;
    }
};
```

我的代码:

```
vector<int> leftmostBuildingQueries(vector<int> &heights, vector<vector<int>> &queries) {
        auto numberOfHeights = heights.size();
        auto numberOfQueries = queries.size();
        vector<int> indices;
```

```
        vector<int> res(numberOfQueries);
        for (auto i = 0; i < numberOfQueries; ++i) {
            auto &a = queries[i][0];
            auto &b = queries[i][1];
            if (a > b) std::swap(a, b);
            if (a == b || heights[b] > heights[a]) {
                res[i] = b;
                continue;
            }
            indices.emplace_back(i);
        }
        std::sort(indices.begin(), indices.end(), [&](auto a, auto b) {
            return queries[a][1] < queries[b][1];
        });
        vector<int> s;
        int right = heights.size() - 1;
        for (int i = indices.size() - 1; i >= 0; --i) {
            auto index = indices[i];
            auto a = queries[index][0];
            auto b = queries[index][1];
            while (right > b) {
                while (s.size() && heights[s.back()] <= heights[right]) {
                    s.pop_back();
                }
                s.emplace_back(right);
                --right;
            }
            auto iter = std::upper_bound(s.rbegin(), s.rend(), a, [&](auto x, auto y) {
                return heights[x] < heights[y];
            });
            if (iter == s.rend()) {
                res[index] = -1;
            } else {
                res[index] = *iter;
            }
        }
        return res;
    }
```

注意

比较 2 行代码的区别

```
auto iter = std::upper_bound(s.rbegin(), s.rend(), a, [&](auto x, auto y) {
                return heights[x] < heights[y];
            });
auto iter = std::upper_bound(s.begin(), s.end(), a, [&](auto x, auto y) {
                return heights[x] > heights[y];
            });
```

std::upper_bound 是一个标准库算法，用于在一个已排序的区间内查找第一个大于给定值的元素的位置。它的函数签名如下：

```
iterator upper_bound(iterator first, iterator last, const T& val);
iterator upper_bound(iterator first, iterator last, const T& val, Compare comp);
```

1. first 和 last 定义了查找的区间范围。
2. val 是我们要查找的目标值。
3. comp 是一个可选的自定义比较函数，用来定义元素之间的大小关系。

两个版本的差异

(a) auto iter = std::upper_bound(s.rbegin(), s.rend(), a, [&](auto x, auto y) return heights[x] < heights[y];);

- s.rbegin() 和 s.rend() 是 s 容器的逆向迭代器，也就是从容器的末尾到开头的迭代器。
- std::upper_bound(s.rbegin(), s.rend(), a, [&](auto x, auto y) return heights[x] < heights[y];) 会在这个反向区间内查找第一个比 heights[a] 小的元素的下一个位置（即找到 heights[x] < heights[a] 的位置）。逆序迭代器意味着在查找过程中，算法会从容器的末尾开始查找。

逆序迭代器的作用：

- rbegin() 和 rend() 使得迭代器从末尾到开头遍历。
- 由于是从大到小的顺序遍历，所以查找是基于倒序的。
- 通过 heights[x] < heights[y] 来比较元素，使得 upper_bound 会找到 “第一个小于目标值 heights[a] 的位置”。

(b) auto iter = std::upper_bound(s.begin(), s.end(), a, [&](auto x, auto y) return heights[x] > heights[y];);

解释：

- s.begin() 和 s.end() 是 s 容器的正向迭代器，也就是从容器的开头到末尾的迭代器。
- std::upper_bound(s.begin(), s.end(), a, [&](auto x, auto y) return heights[x] > heights[y];) 会在这个正向区间内查找第一个比 heights[a] 小的元素的下一个位置（即找到 heights[x] > heights[a] 的位置）。正序迭代器意味着查找是从容器的开始到末尾。

正序迭代器的作用：

- begin() 和 end() 使得迭代器从开头到末尾遍历。

- 查找的是 “第一个大于目标值 heights[a] 的位置”，因此使用 heights[x] > heights[y] 进行比较。

具体差异：

- 迭代器方向的不同（正向与反向）正向迭代器（s.begin() 和 s.end()）：表示从容器的起始位置（最左侧）到结束位置（最右侧），用于查找的顺序是从小到大的。逆向迭代器（s.rbegin() 和 s.rend()）：表示从容器的末尾（最右侧）到起始位置（最左侧），用于查找的顺序是从大到小的。

- 比较函数的不同（< vs >）在 正向迭代器 中，std::upper_bound 是根据 heights[x] > heights[y] 来查找。由于元素是按从小到大的顺序排列的，因此该比较会帮助我们找到第一个 “比目标值大的位置”。在 逆向迭代器 中，std::upper_bound 是根据 heights[x] < heights[y] 来查找。逆向迭代器让元素按从大到小的顺序排列，因此我们查找的是第一个 “比目标值小的下一个位置”。

两者的关系与影响：

正向迭代器的查找顺序与我们通常的直觉一致，元素是从小到大排布，查找的是 “第一个比目标值大的元素”。逆向迭代器则是从大到小排布，查找的是 “第一个比目标值小的元素”。

换句话说，正向迭代器的比较 heights[x] > heights[y] 使得我们要寻找的是第一个满足 heights[x] > heights[a] 的 x，而逆向迭代器的比较 heights[x] < heights[y] 使得我们要寻找的是第一个满足 heights[x] < heights[a] 的 x。