「集思广益一波」

CleanBlue

2022 年 11 月 16 日

目录

1 数据结构

1.1 单调队列/滑动窗口

有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

```cpp
int n, k;
std::cin >> n >> k;

std::vector<int> a(n);
for (int &i : a) {
    std::cin >> i;
}

std::deque<int> q;
for (int i = 0; i < n; ++i) {  //输出最小值
    while (!q.empty() && a[q.back()] >= a[i]) {
        q.pop_back();
    }
    q.emplace_back(i);
    if (i >= k - 1) {
        while (!q.empty() && q.front() <= i - k)
            {
            q.pop_front();
        }
        std::cout << a[q.front()] << ' ';
    }
}
std::cout << '\n';

while (!q.empty()) {
    q.pop_back();
}

for (int i = 0; i < n; ++i) {  //输出最大值
    while (!q.empty() && a[q.back()] <= a[i]) {
        q.pop_back();
    }
    q.emplace_back(i);
    if (i >= k - 1) {
        while (!q.empty() && q.front() <= i - k)
            {
            q.pop_front();
        }
        std::cout << a[q.front()] << ' ';
    }
}
std::cout << '\n';
```

1.2 单调栈

输出右边第一个比 a_i 大的数的下标。

```cpp
int n;
std::cin >> n;

std::vector<int> a(n), ans(n);
for (int &i : a) {
    std::cin >> i;
}

std::stack<int> s;
for (int i = n - 1; i >= 0; --i) {
    while (!s.empty() && a[s.top()] <= a[i]) {
        s.pop();
    }
    if (s.empty()) {
        ans[i] = -1;
    } else {
        ans[i] = s.top();
    }
    s.emplace(i);
}

for (int i = 0; i < n; ++i) {
    std::cout << ans[i] + 1 << "\n";
}
```

2 基本算法

2.1 二分

```cpp
int binary_search_1(const std::vector<int> &a,
    int x) {
    // a 单调递增
    // 在 [0, n) 中查找 x 或 x 的后继的位置
    int n = a.size(), l = 0, r = n;
    while (l < r) {
        int mid = (l + r) / 2;
        if (a[mid] >= x) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }
    return l;
}

int binary_search_2(const std::vector<int> &a,
    int x) {
    // a 单调递增
    //在 [0, n) 中查找 x 或 x 的前驱的位置
    int n = a.size(), l = 0, r = n;
    while (l < r) {
        int mid = (l + r + 1) / 2;
        if (a[mid] <= x) {
            l = mid;
        } else {
            r = mid - 1;
        }
    }
    return l;
}

double float_binary_search(/*...*/) {
    constexpr double eps = 1e-7;
    double l = 0, r = 1e18;
    while (r - l > eps) {
        double mid = (l + r) / 2;
        if (/*...*/) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return l;
}
```

2.2 三分

如何取 mid1 和 mid2? 有以下两种基本方法。

(1) 三等分：mid1 和 mid2 为 $[l, r]$ 的三等分点，区间每次可以缩小 1 / 3。

(2) 近似三等分：计算 $[l, r]$ 的中间点 mid $= (l + r)/2$，然后让 mid1 和 mid2 非常接近 mid，如 mid1 = mid - eps, mid2 = mid + eps。区间每次可以缩小接近一半。

近似三等分比三等分要稍微快一点。不过，在有些情况下 eps 过小可能导致 mid1 和 mid2 算出的结果相等，从而判断错方向。

三等分：

```
while (r - l > eps) {
    double k = (r - l) / 3;
    double mid1 = l + k, mid2 = r - k;
    if (/*...*/) {
        r = mid2;
    } else {
        l = mid1;
    }
}
```

近似三等分：

```
while (r - l > eps) {
    double mid1 = (l + r) / 2;
    double mid2 = mid1 + eps;
    if (/*...*/) {
        r = mid2;
    } else {
        l = mid1;
    }
}
```

整数三分：

```
while (r - l > 2) {
    int mid1 = l + (r - l) / 3;
    int mid2 = r - (r - l) / 3;
    if (/*...*/) {
        r = mid2;
    } else {
        l = mid1;
    }
}
```