GDSC: Algorithms & Data Structures: Practice

Vladislav Artiukhov

March 23, 2024

Contents

1. Basic Data Structures

1.1 Queue & Deque

***Problem.* Maximum of Sliding Window**

Given an array of integers a, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the array that contains maximum elements of each position of the sliding window.

Time complexity: $O(n)$.

Space complexity: $O(k)$.

Note: you can solve it on LeetCode

```
Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

```
Example 2:

Input: nums = [1], k = 1
Output: [1]
```

***Solution.* №1, Using deque + two pointers**

Credits to **monster0Freason** for a great solution.

Let's look at the following example where we see 4 elements and the window size is $k = 3$:

$a_0 = 1, a_1 = 3, a_2 = -1, a_3 = 2$ - for any sliding windows that contain both a_0 and a_1 can never have a_0 as a maximum; same for the sliding windows that contain a_2 and either a_1 or a_3 since $a_2 < a_1, a_3$ $\implies$ we do not care about the elements that are smaller than those we currently capture in a sliding window of size k.

Let's then capture only elements such that $a_{i_1} \geq a_{i_2} \geq ... \geq a_{i_{\bar{k}}} (\bar{k} \leq k)$. Then for the current sliding window the result is a_{i_1} since it is the greatest element among those captured by this sliding window. In order to move the sliding window to the next element we need to consider the next element a_r and remove all $a_{i_j}..a_{i_{\bar{k}}} < a_r$ to preserve the defined above property. In order to support such an algorithm we could use **deque**:

```cpp
    void solve(const std::vector<int>& a, int k) {
        // keeping indexes because it eases the removal of the leftmost element
        // once it gets out of the sliding window range.
        std::deque<int> d;

        int left = 0;
        int right = 0;

        while (right < a.size()) {
            while(!d.empty() && a[d.back()] < a[right]) {
                d.pop_back();
            }
```

```
            d.push_back(right);
            // now we preserve the invariant: a[d[0]] >= a[d[1]] >= a[d[2]] ...

            if (left > d.front()) {
                d.pop_front();
            }

            // once we have observed enough elements for the 1st sliding window
            if (right >= k - 1) {
                std::cout << a[d.front()] << " ";
                left++;
            }
            right++;
        }
    }
```

№2, Using std::multiset + two pointers

We have not yet studied sets and multisets (i.e. **std::set<T>** and **std::multiset<T>**) but it is also a possible solution which works in $O(n \cdot \log k)$ of time complexity.

In C++ **std::set** and **std::multiset** are the data structures that keep the sorted set of the elements witout or with duplicates respectively. By default both of them sort elements in ascending order but it is possible to sort in the descending order as well (**std::multiset<T, std::greater<T> >**, e.g., for int: **std::multiset<int, std::greater<int> >** (same for **std::set<T>**); see template parameters of the structures on cppreference.com). In order to retrieve the maximum element in the set we need to call its **begin()** method which returns an iterator to the beginning (**works for** $O(\log(size))$**!**):

```
std::set<int, std::greater<int>> s1;
s1.insert(2);
s1.insert(1);
s1.insert(3);
// (*s1.begin()) == 3; s1 = {3, 2, 1}

std::set<int> s2;
s2.insert(2);
s2.insert(1);
s2.insert(3);
// (*s2.begin()) == 1; s2 = {1, 2, 3}
```

Here we keep a sliding window of size k by both **left** and **right** pointers and add the next element under **right** and remove an element under **left** once we shift the sliding window. After shifting the sliding window we retrieve the maximum and print it:

```
void solve(const std::vector<int>& a, int k) {
    std::multiset<int, std::greater<int>> st;
    int left = 0;
    int right = 0;

    for (; right < std::min(a.size(), k); ++right) {
        st.insert(a[right]);
    }

    std::cout << (*st.begin()) << " ";

    while(right < a.size()) {
        // removing via iterator, but the element.
        // st.erase(val) would remove all elements that are equal to 'val'
        st.erase(st.find(a[left++]));
        st.add(a[right++]);
        std::cout << (*st.begin()) << " ";
```

```
        }
    }
```

Problem. **Max Value of Equation**

Given an array p of size $n \leq 10^5$ containing the coordinates of points on a 2D plane, sorted by the x-values, i.e. $\{x_i\}$ form a strictly increasing sequence ($x_i < x_j,\ i < j$), where $p_i = (x_i, y_i)$ ($-10^8 \leq x_i, y_i \leq 10^8$). You are also given an integer $k \leq 2 \cdot 10^8$.

Return the **maximum value of the equation** $y_i + y_j + |x_i - x_j|$ where $|x_i - x_j| \leq k$ and $0 \leq i < j < n$.

It is guaranteed that there exists at least one pair of points that satisfy the constraint $|x_i - x_j| \leq k$.

Time complexity: $O(n)$ or $O(n \cdot \log n)$.

Space complexity: $O(k)$.

Note: you can solve it on LeetCode

Example 1:

> **Input:** points = [[1,3],[2,0],[5,10],[6,-10]], k = 1
> **Output:** 4
> **Explanation:** The first two points satisfy the condition $|x_i - x_j|$ <= 1 and if we calculate the equation we get 3 + 0 + |1 - 2| = 4. Third and fourth points also satisfy the condition and give a value of 10 + -10 + |5 - 6| = 1.
> No other pairs satisfy the condition, so we return the max of 4 and 1.

Example 2:

> **Input:** points = [[0,0],[3,0],[9,2]], k = 3
> **Output:** 3
> **Explanation:** Only the first two points have an absolute difference of 3 or less in the x-values, and give the value of 0 + 0 + |0 - 3| = 3.

Solution. Notice that the points are sorted in ascending order by their x-coordinates, thus we can keep a sliding window $(p_{i_1}, p_{i_2}, ..., p_{i_j}), \forall m = 1..j:\ p_{i_m} = (x_{i_m}, y_{i_m})$ that satisfies $x_{i_n} \leq x_{i_m},\ n < m$ and $|x_{i_1} - x_{i_j}| \leq k$, by keeping such a sliding window we automatically satisfy the condition $|x_i - x_j| \leq k$.

Now let's work with the given formula:

$y_i + y_j + |x_i - x_j|$ - assume that $i < j \implies (y_i - x_i) + (y_j + x_j)$. Since the answer always exists, therefore, such j exists either $\implies$ let's assume that every time when we add a next point $p = (x, y)$ into own sliding window, this added point p substitutes its coordinate components x, y in the formula as x_j, y_j, i.e. $x_j := x$ and $y_j := y$.

Notice that now we only have $y_i - x_i$ as unfixed part of the sum, thus we need to maximize it $\implies$ we need to find such a point $\bar{p}_i$ whose $y_i - x_i \to \max$; we already know that this p is contained in our sliding window by definition $\implies$ we need to find maximum in the sliding window over the function $y_i - x_i$, we can easily do it via **max-queue** over pairs $(y_i - x_i,\ x_i)$ where maximum is built over 1st component of the pair:

```
    void solve(const std::vector<pair<int, int>>& points, int k) {
        MaxQueue<pair<int, int>> q;
        int ans = INT_MIN;

        for (int i = 0; i < points.size(); ++i) {
            while(!q.empty() && std::abs(q.front().second - points[i].x) > k
                /* <=> |xi - xj| > k */) {
                q.pop_front();
            }
            // before adding current point we relax the answer
```

```

            // q.retrieveMax() returns a pair {yi-xi, xi}
            // see: structured binding in C++
            auto [diff, x] = q.retrieveMax();

            ans = std::max(ans, diff + points[i].y + points[i].x);
            q.push_back({ points[i].y - points[i].x, points[i].x });
        }

        std::cout << ans << std::endl;
    }
```

2. Binary Search

2.1 Binary search over the answer

***Problem.* Poles installation**

Given n points c_i, $i = 0..n-1$ on a coordinate axis x sorted in the ascending order and k poles. You need to place all of the k poles on the coordinate points c_i. so that the **minimum distance** between **two consequent** poles is maximized.

Note: the actual problem is called "Cows in the stalls", you can find it here (in Russian): "Cows in the stalls".

Solution. The statement can be rewritten in a mathematical notation as follows:

$d := d_{min}(c_i, c_j) \to max,\ \ i < j$ - where d is the minimum distance between the two points c_i and c_j where two consequent poles are installed.

The maximization problem may seam not obvious because the function d has a complex representation, and there is no a clear way of its optimization.

Let's assume that we want d to be greater than some value x. What do we need to satisfy to claim that $d > x$?

Indeed, we need to prove that all the k poles can be placed on the coordinate points c_i in such a way that $\forall i : d(p_{i-1}, p_i) = d(c_{j_{i-1}}, c_{j_i}) = |c_{j_{i-1}} - c_{j_i}| \geq x$, i.e. the distance between all neighbouring poles is at least x.

Let's introduce a function $f(x) = (\#$ of poles placed, so that min distance $d \geq x)$. This function is extremely easy to calculate: merely go over all available coordinates c_i and for the current pole p_j select a coordinate, so that distance between p_j and the previous pole p_{j-1} is at least x and increment the number of placed poles; if at the end the number of placed poles is k, then $d \geq x$.

Obviously enough, $f(x)$ is a monotonically decreasing function, because the greater the constraint x for the min distance d, the fewer number of poles can be placed.

Now, we are interested in finding first point x where $f(x) = k$, this can be done easily via binary search.

```
std::vector<int> coords;

bool func(int x) {
    int poles_placed = 1;
    int last_pole_pos = coords[0];
    for (int c : coords) {
        if (c - last_pole_pos >= x) {
            poles_placed++;
            last_pole_pos = c;
        }
    }
    return poles_placed;
}

int solve() {
    std::sort(coords.begin(), coords.end());
    // minimum x, always sufficient since d_min >= 0
    int l = 0;
    // distance between first and last coordinates + 1
    int r = coords.back() - coords.first() + 1;

    while (r - l > 1) {
        int m = (l + r) / 2;
        if (func(m) >= k) l = m;
        else r = m;
    }
```

```

    return l;
}
```

Problem. **Printers**

There are two printers. One prints one page in x minutes, the other prints in y minutes. In what time will both prints print n pages?

Solution. There is a formula solution, but let's think about the problem differently.

Given t minutes, how many pages can both printers print? Let it be a function $f(t) = p$ where p is the number of pages.

$f(t) = \lfloor \frac{t}{x} \rfloor + \lfloor \frac{t}{y} \rfloor$

If in time t we can print p pages, then we can print $1, 2, 3, .., p-1$ pages as well, thus $f(t)$ is monotonic over t and the time t_0 for n pages can be found via binary search.

As for the left boundary we can take 0 and as for the right boundary, the $x \cdot n$ (time of the first printer working alone) will be always sufficient.

3. Hash table

3.1 Hash table with Sliding Window

Problem. Given an integer array a and an integer k, return the number of **good subarrays** of a.

A **good array** is an array where the number of different (distinct) integers in that array is exactly k. A subarray is a **contiguous part** of an array.

For example, $[1, 2, 3, 1, 2]$ has 3 different integers: $1, 2, 3$.

Asymptotics: $O(n)$ in time and space.

<u>Note</u>: you can solve this problem on LeetCode here 992. Subarrays with K Different Integers.

Example. 1. $a = [1, 2, 1, 2, 3]$, $k = 2$, the answer will be 7, the good subarrays are:

$[1, 2], [2, 1], [1, 2], [2, 3], [1, 2, 1], [2, 1, 2], [1, 2, 1, 2]$.

2. $a = [1, 2, 1, 3, 4]$, $k = 3$, the answer will be 3, the good subarrays are:

$[1, 2, 1, 3], [2, 1, 3], [1, 3, 4]$.

Solution. Idea: $exactly(k) = atMost(k) - atMost(k - 1)$, thus let's count the number of subarrays with at most k ($\leq k$) distinct numbers and the number of subarrays with at most $k - 1$ distinct numbers.

In order to count distinct numbers if a sliding window we need keep the count of each distinct value in the sliding window.

Once the right border of the window goes forward we increment the count of a new value a_j, and if it the count became 1, then increment cnt (since the number of distinct numbers increased by one).

Once the left border of the window goes forward we decrement the count of a tracked value a_i (before incrementing i) and if this value becomes 0, we decrement the number of distinct values.

Let's keep a sliding window $[i, j]$ that will contain $\leq k$ distinct elements:

1. If current number of distinct values is $\leq k$ we only need to move forward the right boundary j and on each iteration increase the result by $(j - i + 1)$ because the number of subarrays that end in j and contain $\leq k$ distinct values is the number of suffixes the subarray $a[i..j]$ has (suffixes are: $[i + 1..j]$, $[i + 2..j]$, ..., $[j - 1..j]$, $[j..j]$). This way we will count all the subarrays of a that contain $\leq k$ elements.

2. Once the number of distinct values cnt inside the sliding window has become $(k + 1)$ we need to move forward the left boundary i until cnt becomes equal to k.

By the above algorithm we found the result of the function $atMost(k)$, now do the same for $atMost(k - 1)$ and return the their difference.

```cpp
int atMost(vector<int>& nums, int k) {
    unordered_map<int, int> counts;
    int i = 0;
    int cnt = 0;
    int result = 0;

    for (int j = 0; j < nums.size(); ++j) {
        if (counts[nums[j]]++ == 0) {
            ++cnt;
        }

        if (cnt <= k) {
            result += (j - i + 1);
        }

        while (i <= j && cnt > k) {
            --counts[nums[i]];
```

```
            if (counts[nums[i]] == 0) {
                --cnt;
            }
            ++i;

            if (cnt == k) {
                result += (j - i + 1);
            }
        }
    }

    return result;
}

int exact(vector<int>& nums, int k) {
    return atMost(k) - atMost(k-1);
}
```

4. Graphs and basic depth-first-search (dfs)

Problem. **Cycle in Graph**

Given a graph g of n verticies and m edges. Find any cycle in the given graph and print its verticies.

Solution. The idea is to implement dfs with **two colors**:

1. *Color 1* denotes that this vertex was visited but yet not left by the dfs.
2. *Color 2* denotes that this vertex was left by the dfs.

The cycle is a situation in the dfs algorithm when it encounters any vertex that has a color 1. Once such vertex is found the graph has a cycle.

In order to print its verticies we need to keep track of the visited but yet not left verticies (i.e. verticies of color 1). We can implement it by inserting a currently considered vertex inside an vector and remove it from the vector once all its neighbouring verticies are traversed. Once the cycle is found ending at the vertex v of color 1, we will go through the elements of this tracking vector until and print its elements until v is found.

```cpp
bool dfs(int v, vector<int>& trace) {
    if (marked[v] == 2) return false;
    if (marked[v] == 1) {
        // found cycle
        std::cout << v << " ";
        while(trace.back() != v) {
            std::cout << trace.back() << " ";
            trace.pop_back();
        }
        return true;
    }

    marked[v] = 1;
    trace.push_back(v);

    // traverse neighbouring verticies
    bool found = false;
    for (int u : graph[v]) {
        found = dfs(u);
        if (found) break;
    }

    trace.pop_back();
    return found;
}
```

Problem. **2285. Maximum Total Importance of Roads**

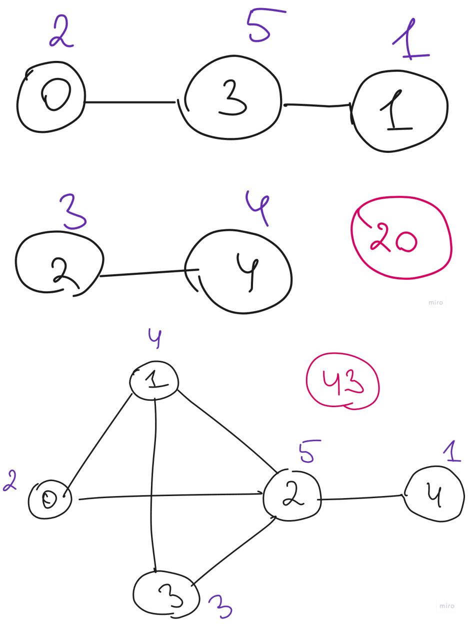
You are given an integer n denoting the number of cities in a country. The cities are numbered from 0 to $n-1$. You are given the set of roads r between the cities $a_i \to b_i$.

You need to assign integer values from 1 to n to all the cities, where each value can only be used **once**.

Definition: The **importance** of a road is the sum of the values of the two cities it connects.

Return the **maximum total importance** of all roads possible after assigning the values optimally.

Note: you may solve this problem on LeetCode, 2285. Maximum Total Importance of Roads.



Solution. Let's consider a single vertex v with assigned value i_v and its contribution to the result: $\deg v \cdot i_v$.

The total importance is calculated as follows: $S := \sum_{v \in V} i_v \cdot \deg v \to \max$. Notice that S takes its maximum when we assign greater values i_v to verticies with greater deg. Thus, let's sort the verticies according their degrees $\deg v$ in the ascending order and assign values from 1 to n in such order:

```cpp
int maximumImportance(vector<pair<int>>& roads) {
    vector<int> deg(n);
    for (auto [v, u] : roads) {
        ++deg[v];
        ++deg[u];
    }

    vector<int> verticies(n);
    for (int i = 0; i < n; ++i) {
        verticies[i] = i;
    }

    std::sort(std::begin(verticies), std::end(verticies), [&](int v, int
        u) {
        return deg[v] < deg[u];
    });

    int S = 0;
    for (int value = 1; value <= n; ++value) {
        int v = verticies[value - 1];
        S += value * deg[v];
    }

    return S;
}
```