STL Application - 3

STL Applications and Problem Ideas-3

1. Problem: Given an array of integers arr and an integer k, write a function to print the minimum element of each k-size subarray.

One approach could be using multiset.

Algorithm Explanation:

- Start iterating through the array arr from index 0 to n-1, where n is the size of the array.
- Check if the current index i is greater than or equal to k. If true, it means we have a k-size window to consider.
- Remove the element at index i - k from the multiset. This ensures that the multiset always contains elements from the current window.
- Check if the current index i is greater than or equal to k - 1. If true, it means we have a valid k-size window.
- Print the minimum element of the current k-size window, which is the value at *s.begin().

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

void printMinOfSubarrays(const vector<int> &arr, int k)
{
    multiset<int> s;
    for (int i = 0; i < arr.size(); ++i)
    {
        s.insert(arr[i]);


        if (i >= k)
        {
            s.erase(s.find(arr[i - k]));
        }


        if (i >= k - 1)
        {
            cout << *s.begin() << endl;
        }
    }
}


int main()
{
    // Example usage:
    vector<int> arr = {3, 1, 4, 2, 8, 6, 5, 7};
    int k = 3;

    cout << "Minimum of each " << k << "-size subarray:" << endl;
    printMinOfSubarrays(arr, k);

    return 0;
}
```

Time Complexity is O(nlogk) as the time complexity of the insert and erase operations in a multiset is O(log k), where k is the size of the multiset. Why logk not logn? because the size of a multiset is always k.

But a better approach is possible and the idea is Monotone deque. Let's first learn about it.

A monotonic deque, short for monotonic dequeue or monotone decreasing dequeue, is a data structure that maintains elements in non-increasing order while allowing for efficient insertion and removal operations. It is often used in problems where you need to track the minimum (or maximum) element in a sliding window of a sequence.

`Insert Operation (insert):` The insert operation in the monotone deque is used to insert an element into the deque. It maintains the monotonicity property by popping elements from the back of the deque until the back element is less than or equal to the new element.

`Erase Operation (erase):` The erase operation is used to remove an element from the deque. It removes the front element if it matches the specified value.

`Get Minimum Operation (getmin):` The getmin operation returns the minimum element in the deque, which is always at the front due to the monotonic property.

`Why "Monotone" Deque:` The term "monotone" refers to the property that the elements in the deque maintain either non-increasing or non-decreasing order. In this case, the monotone deque is non-increasing. As elements are inserted, the deque remains sorted in non-increasing order.

`Why Use Monotone Deque:` The monotone deque efficiently maintains the minimum element in a sliding window of size k. It allows constant time complexity for insertion, erasure, and retrieval of the minimum element. The use of a deque ensures that elements within the current window are sorted in non-increasing order, making it easy to access the minimum.

Code:

Copy

```cpp
#include <bits/stdc++.h>
using namespace std;

struct monotone_deque
{
    deque<int> dq;

    void insert(int x)
    { // O(k)
        while (!dq.empty() && dq.back() > x)
            dq.pop_back();
        dq.push_back(x);
    }
    void erase(int x)
    { // O(1)
        if (dq.front() == x)
            dq.pop_front();
    }
    int getmin()
    { // O(1)
        return dq.front();
    }
};


int main()
{ // O(n)
    int n, k;
    cin >> n >> k;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
```

```
    monotone_deque mt;
    for (int i = 0; i < n; i++)
    {                       // O(n)
        mt.insert(arr[i]); // O(1)
        if ((i - k) >= 0)
            mt.erase(arr[i - k]); // O(1)
        if (i >= (k - 1))
            cout << mt.getmin() << endl; // O(1)
    }
}
```

The time complexity is O(n), where n is the size of the array. The easy way to understand the complexity is that at each element two operations are done, one is insertion and second is pop, nothing else.

Mark as Read

Write a comment

B *I* U

• **Before you post...** I Understand

- **No Cheating or Violation of Honor Code** : Users are usually prohibited from discussing or sharing solutions for active contests. This is to prevent cheating and to maintain the integrity of the platform.
- **Avoid Posting Full Solutions** : Users are encouraged not to post entire solutions to problems, especially for active contests. Instead, it's common to share hints, tips, or discuss general approaches without giving away the entire solution.
- **Be Respectful** : Users are expected to be respectful and considerate of others. Avoid using offensive language, making personal attacks, or engaging in any form of harassment.
- **Stay On Topic** : Discussions should stay relevant to the problem at hand. Off-topic discussions or spamming are generally discouraged.