Two Pointers & Sliding Window

Agenda

1. What is The Two-Pointer Technique?

❖ The Two Pointers Technique is a broad category in algorithm design, used to simplify problems that involve sequences (like arrays or lists) by using two pointers to traverse the sequence, often in a single pass. This technique can significantly reduce the complexity of problems that might otherwise require nested iterations.

❖ In the Two Pointers Technique, we iterate two **monotonic** pointers across an array to search for a pair of indices satisfying some condition in linear time.

❖ For example, the Two Pointers Technique can be used to track:

➢ The start and end of an interval (subarray, substring, etc..).

➢ Two values in the same array or two different arrays.

2. Types of Problems It Solves

The Two Pointers Technique solves a wide range of problems that vary in nature, let's categorize them into three categories:

a. **Foundational Two Pointers Problems**: Scenarios that do not involve a window or segment of the array. Such as:

➢ Finding two values in a sorted array that sum up to T.

➢ Merging two sorted arrays in linear time.

b. **Variable Size Windows**: Problems where you need to find a subarray within an array that meets a **certain condition**, and the size of this window is **not fixed**. The window size **adjusts dynamically** based on the condition being met.

➢ Finding the longest subarray with sum $\leq k$ (or the shortest subarray with sum $> k$).

➢ Finding the longest subarray with at most k unique characters.

➢ Finding the longest substring without repeating characters.

c. **Fixed Size Windows (Sliding Window)**: A specific instance of the Two Pointers Technique where the window size is **fixed**, and the window **"slides"** across the array to find a segment satisfying a particular condition.

➢ Finding the maximum sum subarray of size k.

➢ Finding the maximum of every subarray of size k.

➢ Finding the first negative integer in every window of size k.

3. Foundational Two Pointers Problems

a. Two-Sum Problem

Given a sorted array A of length N and a target T, you need to find whether there are two values in the array that add up to T or not.

$Array\ A,\ T = 8$**:**

0	1	2	3	4	5
1	2	3	4	5	8

Solution:

The naive solution is to brute force all different pairs $i\ \&\ j$ such that $0 \le i < j \le n$, for each pair check if $A[i] + A[j]$ adds up to T, until you either find a valid pair or you finish with no valid solution.

Code:

```
bool twoSum(vector<int>& A, int T) {
    for (int i = 0; i < A.size(); ++i) {
        for (int j = i + 1; j < A.size(); ++j) {
            if (A[i] + A[j] > T) break;
            if (A[i] + A[j] == T) return true;
        }
    }
    return false;
}
```

Time complexity: $O(N^2)$

Let's analyze how $i\ \&\ j$ move with every iteration:

- i moves forward with every iteration.
- j moves forward and breaks when $A[i] + A[j]$ is greater than T.

Let's suppose that in the i-th iteration, j breaks at index X. Since array A is sorted, $A[i]$ increases as i increases. That is $A[i + 1] \ge A[i]$. That means that in the $(i + 1)$-th iteration, j will break at index Y such that $Y \le X$. Let's rewrite the code so that j is decreasing.

```
bool twoSum(vector<int>& A, int T) {
   for (int i = 0; i < A.size(); ++i) {
       for (int j = A.size() - 1; j > i; --j) {
           if (A[i] + A[j] < T) break;
           if (A[i] + A[j] == T) return true;
       }
```

```
    }
    return false;
}
```

Same analysis: as i increases, $A[i]$ also increases.
With the same logic as before: if in the i-th iteration j breaks at index X, in the $(i + 1)$-th iteration j will break at index Y such that $Y \leq X$. But do we really need to reset j for each iteration of i?

- If $A[i] + A[j] > T$,
- Then $A[i + 1] + A[j]$ is also greater than T, as $A[i + 1] \geq A[i]$. This means if we have tried j for i, then when moving to $i + 1$, we need to try all values of J such that $J < j$ only.
- This concludes that j should be **monotonically** decreasing, and we could rewrite the code as:

Code:

```
bool twoSum(vector<int>& A, int T) {
    int j = A.size() - 1;
    for (int i = 0; i < j; ++i) {
        while (j > i && A[i] + A[j] > T) --j;
            if (A[i] + A[j] == T) return true;
     }
    return false;
}
```

Time Complexity: $O(N)$, every element is visited at most once by either i or j. That's how we can use the two-pointers technique to optimize the brute force approach.

b. Merge Two Sorted Arrays

Given two arrays A and B, sorted in ascending order, we want to merge the elements of these arrays into one big array C, also sorted in ascending order.

Array A:

1	3	4	7	8

Array B:

2	3	5	6	8	10	11

Solution:

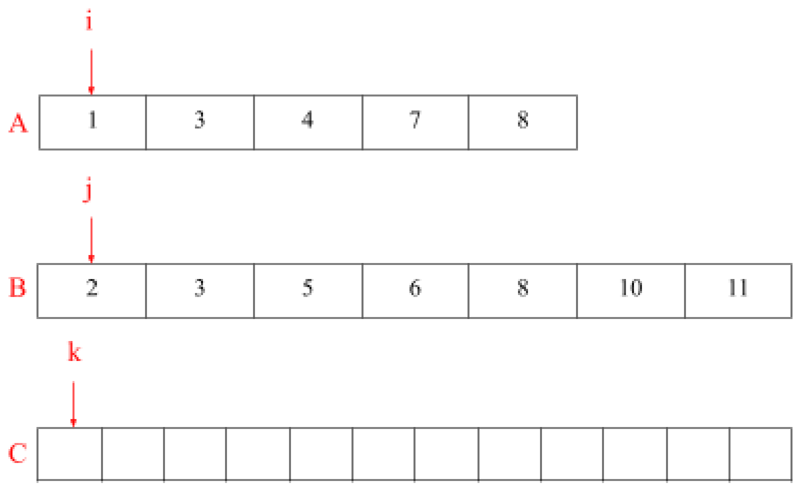
The easiest way to solve this problem is to combine both arrays into one array, and then sort it. This algorithm takes $O(nlog(n))$ time complexity, where n is the length of the combined array C.

Let's see how we can solve it without sorting and in linear time.

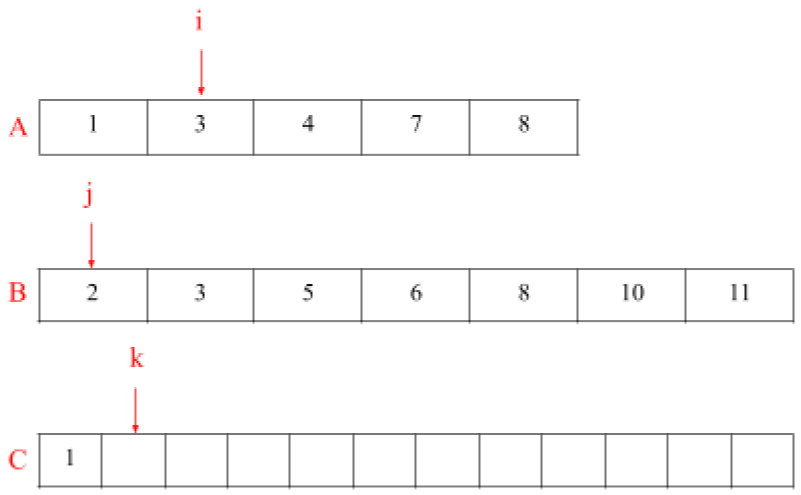
The key to solving this problem is to make use of the fact that the two arrays are already sorted.

Since the two arrays are sorted, we know that the smallest element in the final combined array must be either the smallest element in A or the smallest element in B, which is at index 0 in both arrays.

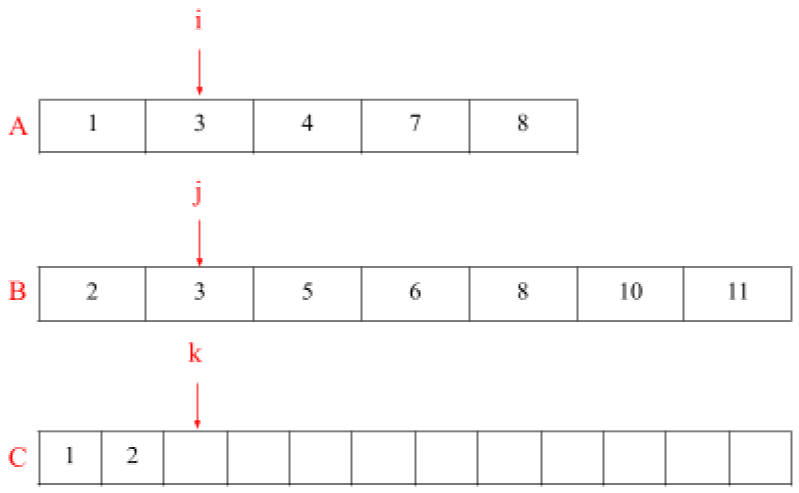
Let's initialize a pointer $i = 0$ that iterates over array A, a pointer $j = 0$ that iterates over array B, and a pointer k that iterates over the new array C to place all combined elements sorted ascendingly.



- The smallest element in C ($C[0]$) is the minimum between $A[0]$ and $B[0]$.
- Here, $C[0] = A[0]$. Since $A[0]$ is now taken, let's move pointer i one step forward.



- Now, the next smallest element in C ($C[1]$) is the minimum between $A[i]$ and $B[j]$.
- Here, $C[1] = B[j]$. Since $B[j]$ is now taken, we move pointer j one step forward.



- And so on, we keep comparing the smallest untaken value in A with the smallest untaken value in B and place the smaller of them at index k in C.
- Every time the smaller is $A[i]$, we move i and k one step forward.
- Similarly, every time the smaller is $B[j]$, we move j and k one step forward.
- At any moment, if the end of A or B is reached, then the rest of the other array is taken in order, to fill C.
- Finally, when both pointers reach the end, this means that all elements were merged into array C. Thus, the algorithm terminates.

Code:

```
vector<int> mergeTwoSortedArrays(vector<int> &A, vector<int>
&B) {
    int n = A.size(), m = B.size();
    vector<int> C(n + m);
    int i = 0, j = 0, k = 0;
    while (i < n && j < m) {
        if (A[i] <= B[j]) {
            C[k] = A[i];
            i++;
        } else {
            C[k] = B[j];
            j++;
        }
        k++;
    }


    while (i < n) {
        C[k] = A[i];
        i++;
        k++;
    }
    while (j < m) {
        C[k] = B[j];
        j++;
        k++;
    }
    return C;
}
```

Time Complexity: $O(A.size() + B.size())$.

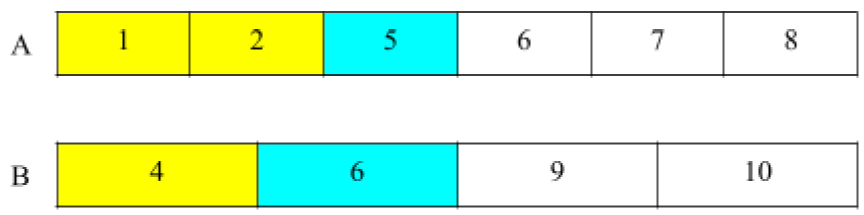
c. Count the Number of Smaller Elements

Given two arrays A and B, we want to calculate for each element B_j how many indices i exist in A such that $A[i] < B[j]$.

Solution:

The naive approach is to iterate over array B and for each element iterate over array A and count the number of elements satisfying $A[i] < B[j]$. This costs $O(A.size() \times B.size())$ time complexity.

To solve this problem efficiently, let's observe the following example assuming both arrays are sorted.



- Yellow elements in A are less than 4 in B. Also, yellow and blue elements in A are less than 6 in B. Is this a coincidence?
- The main observation here is that for any pair of indices x and y in B such that $B[x] < B[y]$, all elements that are smaller than $B[x]$ are also smaller than $B[y]$.
- Therefore, instead of iterating over A for each element in B, we can iterate a monotonic pointer i which will keep moving forward only without having to reset it.
- Whenever $A[i] \geq B[j]$, we store the result and move j one step forward.

Code:

```
vector<int> countSmaller(vector<int> &A, vector<int> &B) {
    int n = A.size(), m = B.size();
    vector<int> res(m);
    int i = 0;
    for (int j = 0; j < m; ++j) {
        while (i < n && A[i] < B[j]) {
            i++;
        }
        res[j] = i;
    }
    return res;
}
```

Time Complexity: $O(A.size() + B.size())$.

4. Variable Size Windows

a. Longest Subarray with Sum k

Given an unsorted array A of N positive integers and a Target sum T, find a subarray whose sum = T.

$Array\ A\ (T\ =\ 9)$:

0	1	2	3	4	5
2	5	3	1	6	8

Solution:

Let's start with the naive way:

We need to find a start S and an end E such that the sum of $A[S{:}E]\ =\ T$. So we will try every start and end in the array.

Code:

```
void subarraySum(vector<int>& A, int T) {
   for (int S = 0; S < A.size(); ++S) {
       int sum = 0;
       for (int E = S; E < A.size(); ++E) {
           sum += A[E];
           if (sum > T) break;
           if (sum == T) {
               cout<< S <<" "<< E <<"\n";
               return;
           }
       }
   }
   cout<< -1 <<"\n"; // no solution
}
```

Time complexity: $O(N^2)$.

Let's analyze how moves of S & E affect the subarray sum. When E moves, the subarray sum increases. We keep moving E until either we reach our target sum and stop, or the sum becomes greater than T. Then E breaks, S moves one step forward, and
E is reset.

Initial state: S & E at index 0.

S, E					
0	1	2	3	4	5
2	5	3	1	6	8

E is expanded till index 1, sum is $2 + 5 = 7$. We try index 2: $7 + 3 = 10 > T$, E stops here.

S		E			
0	1	2	3	4	5
🟩 2	🟩 5	3	1	6	8

We move S one step forward and reset E.

	S, E				
0	1	2	3	4	5
2	5	3	1	6	8

Now we keep expanding E again until we reach index 3 with sum $= 5 + 3 + 1 = 9 = T$. The target subarray sum was found, so the algorithm terminates.

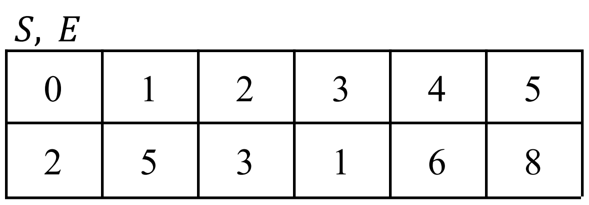
	S		E		
0	1	2	3	4	5
2	🟩 5	🟩 3	🟩 1	6	8

Observation:

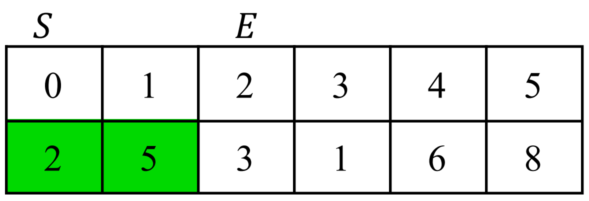
- Moving E one step forward increases the sum.
- Moving S one step forward decreases the sum.
- At each starting point S, we keep expanding E as long as possible. Once E reaches an element that if added will make the sum exceed T, E stops without adding it, meaning that the sum remains smaller than the target at this point.
- Then, S moves one step forward, meaning that the front element is excluded from the sum. Since the sum was already smaller than T and now it decreased, it's guaranteed that it will remain smaller than T after this move.
- Therefore, there is no need to reset E.

This is how it works:

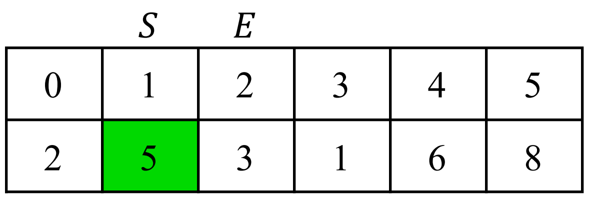
Initial state: S & E at index 0.



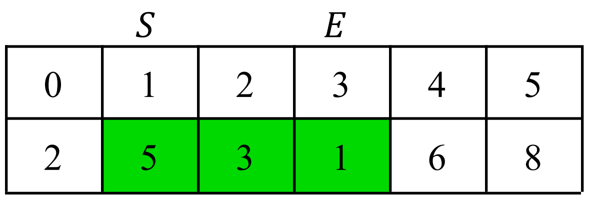
E is expanded till index 1, sum is $2 + 5 = 7$. We try index 2: $7 + 3 = 10 > T$, E stops here.



We remove $A[0]$ from the sum and move S one step forward, sum = 7 - 2 = 5.



Now we keep expanding E again and reach index 3 with sum $= 5 + 3 + 1 = 9 = T$. The target subarray sum is found, and the algorithm terminates.



Code:

```cpp
void subarraySum(vector<int>& A, int T) {
   int S = 0, E = 0, sum = 0;
   while (S < A.size()) {
       // keep expanding E as long as the new sum <= T
       while (E < A.size() && sum + A[E] <= T) {
           sum += A[E];
           ++E;
       }
       // we found a solution
       if (sum == T) {
           cout<< S <<" "<< E - 1 <<"\n";
           return;
       }
       sum -= A[S]; // remove the front element
       ++S; // move S one step forward
   }
   cout<< -1 <<"\n"; // no solution
}
```

Time Complexity: In the worst case scenario S & E move till the end. Every index will be visited twice by S & E. So we have a maximum of $2 * N$ steps and a Time complexity of $O(N)$.

b. Longest Substring Without Repeating Characters (Link)

Given a string S, find the length of the longest substring without repeating characters.

Example:

```
S = "abcabcbb"
Output: 3
```

Solution:

If a substring $s[i:j]$ has no repeating characters, then any prefix of it also has no repeating characters.

- Let mx be the maximum valid substring length, initialize $mx = 0$.
- Let's place two pointers i and j at index 0 and declare a set $taken$ that includes all characters in the current substring.
- As long as $i < s.size()$:
    - As long as $j < s.size()$ and $S[j] \notin taken$:
        - Increment j.
        - Add $S[j]$ to the $taken$ set.
    - Update mx if the current substring length $(j - i)$ is greater.
    - Exclude $S[i]$ from the current substring by removing it from the $taken$ set and incrementing i.

Code:

```
string s;
cin >> s;
int i = 0, j = 0, mx = 0;
unordered_set<char> taken;
while (i < s.size()) {
    while (j < s.size() && taken.find(s[j]) == taken.end()) {
        taken.insert(s[j]);
        j++;
    }
    mx = max(mx, j - i);
    taken.erase(s[i]);
    i++;
}
cout << mx << "\n";
```

Time Complexity: $O(s.size())$.

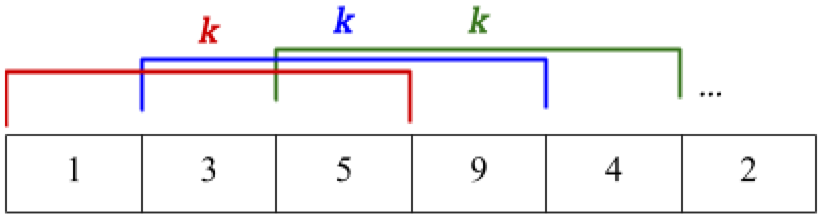
5. Fixed Size Windows (Sliding Window)

a. Maximum Sum Subarray of Size k

Given an array A of N integers and an integer k, we want to find the sum of the maximum sum subarray of size k.

Solution:

Since the substring size is fixed, we use the sliding window technique.



- Initialize $mx = 0$ and $sum = 0$.
- Let's place two pointers i and j at index 0.
- While the end of the current window $(i + k - 1)$ is $< N$:
    - While the size of the current window is $< k$:
        - Add $A[j]$ to sum and increment j.
    - At this point, the length of the current window is k. Update mx if the current window sum is greater.
    - Exclude $A[i]$ by subtracting it from sum and increment i to explore the next window.

Code:

```cpp
int n, k;
cin >> n >> k;
vector<int> arr(n);
for (int i = 0; i < n; ++i)
    cin >> arr[i];
int i = 0, j = 0, mx = 0, sum = 0;
while (i + k - 1 < n) { // i+k-1 is the current window end
    while (j - i < k) {
        sum += arr[j];
        j++;
    }
    mx = max(mx, sum);
    sum -= arr[i];
    i++;
}
cout << mx << "\n";
```

Time Complexity: $O(N)$.

Another Sliding Window Implementation:

We can process the first window of size k, then iterate through each ending index i starting from k to $N - 1$. For each ending index, update the window by adding $A[i]$ to the sum and removing $A[i - k]$ from it.

Code:

```
int n, k;
cin >> n >> k;
vector<int> arr(n);
for (int i = 0; i < n; ++i) {
    cin >> arr[i];
}
int sum = 0, mx;
for (int i = 0; i < k; ++i) {
    sum += arr[i];
}
mx = sum;
for (int i = k; i < n; ++i) {
    sum += arr[i];
    sum -= arr[i - k];
    mx = max(mx, sum);
}
cout << mx << "\n";
```

Time Complexity is the same, $O(N)$.

b. Maximum of All Subarrays of Size k (Link)

Given an array A of size N and an integer k, we want to find the maximum element of each window of size k in the array.

Solution:

Let's maintain a data structure by which we can directly access the maximum element, add new elements, and remove elements from it.

A priority queue or a multiset is suitable for this task.

We will do as usual. First, we need to find the maximum element in the first window, then we slide the window by adding one new element and removing elements that are outside the current window.

- To keep track of the maximum element in the window, we add the newly added element in the window to the priority queue (max-heap) along with its index in the array $\{A[index],\ index\}$.

- The maximum element in the window is simply the top of the priority queue.
- How do we remove the front element of the window?
    - Using a priority queue, we can't access or search for an element easily, so we will simply remove the front element of the window when it's at the top of the queue, but how will we know that it should be removed? That's exactly where we need the index we inserted with each element. We know that the current window range is $[i: j]$, so any element whose index is before i should be removed.
    - Using a multiset, we can simply use $find$ to search for the front element of the window and erase it.

Code #1 (Using Priority Queue):

```
vector<int> maxSlidingWindow(vector<int> &arr, int k) {
    int n = arr.size();
    vector<int> mx(n - k + 1);
    priority_queue<pair<int,int>> pq;
    int i = 0, j = 0;
    while (i + k - 1 < n) {
        while (j - i < k) {
            pq.push({arr[j], j});
            j++;
        }
        while (pq.top().second < i) {
            pq.pop();
        }
        mx[i] = pq.top().first;
        i++;
    }
    return mx;
}
```

Code #2 (Using Multiset):

```
vector<int> maxSlidingWindow(vector<int> &arr, int k) {
    int n = arr.size();
    vector<int> mx(n - k + 1);
    multiset<int> seen;
    int i = 0, j = 0;
    while (i + k - 1 < n) {
        while (j - i < k) {
            seen.insert(arr[j]);
            j++;
        }
        mx[i] = *seen.rbegin();
        seen.erase(seen.find(arr[i]));
        i++;
    }
    return mx;
}
```

Time Complexity: $O(N * logN)$.

6. Try By Yourself

a. Longest Substring with at most k Unique Characters

Given a string S and an integer k, find the length of the longest substring that contains at most k unique characters.

Example:

```
S = "aksacbbaxc", k = 3
Output: 5
```

String `"acbba"` is the longest substring that has at most 3 unique characters.

b. First Negative Integer in Every Window of Size k

Given an array A of size N and an integer k, find the first negative integer in every window of size k.

Example:

```
S = {1, 3, -1, -5, 2, -10, 8}, k = 3
Output: {-1, -1, -1, -5, -10}
```

That's it, good luck and happy coding 😁