Unit-1

Syllabus

Sliding Window – Introduction- Applications – Naive Approach, Diet Plan Performance, Distinct Numbers in Each Sub array, Kth Smallest Sub array Sum, Maximum of all sub arrays of size k.

Two Pointer Approach -Introduction –Palindrome Linked List, Find the Closest pair from two sorted arrays, Valid Word Abbreviation.

Sliding Window – Introduction:

What is Sliding Window?

The **Sliding Window** technique is an efficient way to solve problems involving subarrays or substrings by avoiding redundant computations. It is commonly used for problems requiring contiguous window-based calculations in **arrays** or **strings**.

These problems are painless to solve using a brute force approach in $\mathbf{O(n^2)}$ or $\mathbf{O(n^3)}$. However, the **Sliding window** technique can reduce the time complexity to **O(n)**.

```
[ 5, 7, 1, 4, 3, 6, 2, 9, 2 ]
[ 5, 7, 1, 4, 3, 6, 2, 9, 2 ]
```

The basic idea behind the sliding window technique is to transform two nested loops into a single loop.

Below are some fundamental clues to identify such kind of problem:

- The problem will be based on an array, list or string type of data structure.
- It will ask to find subrange in that array or string will have to give longest, shortest, or target values.
- Its concept is mainly based on ideas like the longest sequence or shortest sequence of something that satisfies a given condition perfectly.

Let's say that if you have an array like below:

[a, b, c, d, e, f, g, h

A sliding window of **size 3** would run over it like below:

[a, b, c]

[b, c, d]

[c, d, e]

[d, e, f]

[e, f, g]

[f, g, h]

Basic Steps to Solve Sliding Window Problem

- Find the size of the window on which the algorithm has to be performed.
- Calculate the result of the first window, as we calculate in the naive approach.
- Maintain a pointer on the start position.
- Then run a loop and keep sliding the window by one step at a time and also sliding that pointer one at a time, and keep track of the results of every window.

OR

- Take **HashMap** or dictionary to count specific array input and uphold on increasing the window towards right using an outer loop.
- Take one inside a loop to reduce the window side by sliding towards the right. This loop will be very short.
- Store the current maximum or minimum window size or count based on the problem statement.

Example of sliding to find the largest sum of five consecutive elements.

[5, 7, 1, 4, 6, 3, 2, 9, 2 , 7]

Step 1:Understand the Problem Statement

We are given an array:

arr = [5, 7, 1, 4, 6, 3, 2, 9, 2, 7]

We need to find the maximum sum of any five consecutive elements in this array.

Step 2: Identify the Sliding Window Properties

- The window size (K) = 5.
- We start by computing the sum of the first 5 elements.
- Then, we slide the window by removing the leftmost element and adding the next element in the array.
- We keep track of the maximum sum encountered.

Step 3: Initialize the Window

Compute the first window sum

We sum the first 5 elements:

Initial window (first 5 elements): [5, 7, 1, 4, 6]

Sum = 5 + 7 + 1 + 4 + 6 = 23

So, our initial maximum sum = 23.

Step 4: Slide the Window

Now, we move the window one step right at a time:

Iteration 1: Slide Right (Window Moves Right)

- Remove the first element (5).
- Add the next element (3).
- Compute the new sum:

New window: [7, 1, 4, 6, 3]

New sum = (Previous sum - outgoing element + incoming element)

= 23 - 5 + 3

= 21

Max Sum So Far: max(23, 21) = 23

Iteration 2: Slide Right

- Remove 7, Add 2.
- Compute new sum:

New window: [1, 4, 6, 3, 2]

New sum = 21 - 7 + 2 = 16

- Max Sum So Far: max(23, 16) = 23

Iteration 3: Slide Right

- Remove 1, Add 9.
- Compute new sum:

New window: [4, 6, 3, 2, 9]

New sum = 16 - 1 + 9 = 24

- Max Sum So Far: max(23, 24) = 24

Iteration 4: Slide Right

- Remove 4, Add 2.
- Compute new sum:

New window: [6, 3, 2, 9, 2]

New sum = 24 - 4 + 2 = 22

- Max Sum So Far: max(24, 22) = 24

Iteration 5: Slide Right

- Remove 6, Add 7.
- Compute new sum:

New window: [3, 2, 9, 2, 7]

New sum = 22 - 6 + 7 = 23

- Max Sum So Far: max(24, 23) = 24

Step 5: Conclusion

The largest sum of any five consecutive elements is 24, and the corresponding subarray is:

[4, 6, 3, 2, 9]

Instead of recalculating results for every possible subarray, the **sliding window technique** maintains a window (a subset of elements) and efficiently slides it across the data structure, updating results dynamically.

Types of Sliding Window

1. **Fixed-size Sliding Window** – Used when the window size is constant.
2. **Variable-size Sliding Window** – Used when the window size changes dynamically based on a condition.

1. Fixed-size Sliding Window

Concept

- The window size **remains constant** (K elements).
- Used in problems where we analyse **subarrays of a fixed size**.

Example: ***Running Average of Temperature***

Imagine a weather station recording temperatures. You want to calculate the **average temperature of the last 7 days** every day. Here, the **window size is fixed at 7 days**, and every new day, you slide the window to exclude the oldest reading and include the newest one.

2. Variable-size Sliding Window

Concept

- The window size **changes dynamically** based on a condition.
- Used when the problem involves **constraints (sum, distinct elements, etc.)**.

Example: ***Longest Substring Without Repeating Characters***

Imagine typing a password. You want to find the **longest substring of unique characters**. The window size **expands when new unique characters are added** and **shrinks when a duplicate is found**.

Fixed-Size Sliding Window

Let's look at an example to better understand this idea.

The Problem:

Suppose the problem gives us an array of length 'n' and a number 'k'. **The problem asks us to find the maximum sum of 'k' consecutive elements inside the array**.

In other words, first, we need to calculate the sum of all ranges of length 'k' inside the array. After that, we must return the maximum sum among all the calculated sums.

Naive Approach:

Let's take a look at the naive approach to solving this problem:

Algorithm 1: Naive Approach for maximum sum over ranges

```
Data: A: The array to calculate the answer for
      n: Length of the array
      k: Size of the ranges
Result: Returns the maximum sum among all ranges of length k
answer ← 0;
for L ← 1 to n − k + 1 do
    sum ← 0;
    for i ← L to L + k − 1 do
        sum ← sum + A[i];
    end
    answer ← maximum(answer, sum);
end
return answer;
```

- First, we iterate over all the possible beginnings of the ranges. For each range, we iterate over its elements from **L to L+K-1** and calculate their sum. After each step, we update the best answer so far. Finally, the answer becomes the maximum between the old answer and the currently calculated sum.
- In the end, we return the best answer we managed to find among all ranges.
- **The time complexity is $O(n^2)$ in the worst case**, where 'n' is the length of the array.

Brute Force: Total Complexity = O(N * K)

```
public class BF_Demo
{
    public static int maxSumBruteForce(int[] arr, int k)
    {
        int n = arr.length;
        if (n < k) return -1; // Edge case: Not enough elements
```

```
        int maxSum = Integer.MIN_VALUE;

        // Iterate over all possible subarrays of size K
        for (int i = 0; i <= n - k; i++)
        {
            int currentSum = 0;

            // Calculate the sum of subarray from i to i + k - 1
            for (int j = i; j < i + k; j++)
            {
                currentSum += arr[j];
            }

            // Update maxSum if we found a larger sum
            maxSum = Math.max(maxSum, currentSum);
        }

        return maxSum;
    }

    public static void main(String[] args)
    {
        int[] arr = {5, 7, 1, 4, 6, 3, 2, 9, 2, 7};
        int k = 5;
        System.out.println("Maximum sum of 5 consecutive elements (Brute Force): "
        + maxSumBruteForce(arr, k));
    }
}
```

Sliding Window Algorithm:

Let’s try to improve on our naive approach to achieve a better complexity.

First, let’s find the relation between every two consecutive ranges. The first range is obviously [1,k] . However, the second range will be [2,k+1].

We perform **two operations** to move from the first range to the second one:

1. **The first operation is adding the element with index ‘k+1’ to the answer**.
2. **The second operation is removing the element with index 1 from the answer**.

Every time, after we calculate the answer to the corresponding range, we just maximize our calculated total answer.

Let’s take a look at the solution to the described problem:

Algorithm 2: Sliding window technique for maximum sum over ranges

```
Data: A: The array to calculate the answer for
      n: Length of the array
      k: Size of the ranges
Result: Returns the maximum sum among all ranges of length k
sum ← 0;
for i ← 1 to k do
    sum ← sum + A[i];
end
answer ← sum;
for i ← k + 1 to n do
    sum ← sum + A[i] − A[i − k];
    answer ← maximum(answer, sum);
end
return answer;
```

Firstly, we calculate the sum for the first range which is [1,K]. Secondly, we store its sum as the answer so far.

After that, we iterate over the possible ends of the ranges that are inside the range [k+1,n]. In each step, we update the sum of the current range. Hence, we add the value of the element at index and delete the value of the element at index 'i -k'.

Every time, we update the best answer we found so far to become the maximum between the original answer and the newly calculated sum. In the end, we return the best answer we found among all the ranges we tested.

The time complexity of the described approach is O(n), where 'n' is the length of the array.

Sliding Window: Total Complexity: O(N) (efficient compared to O(N*K) brute force)

```
public class SlidingWindowMaxSum
{
    public static int maxSumSubarray(int[] arr, int k)
    {
        int n = arr.length;
        if (n < k) return -1; // Edge case: Not enough elements

        int maxSum = 0, windowSum = 0;

        // Compute sum of the first window
        for (int i = 0; i < k; i++)
        {
            windowSum += arr[i];
        }
        maxSum = windowSum;

        // Slide the window over the array
        for (int i = k; i < n; i++)
```

```
            {
                windowSum += arr[i] - arr[i - k]; // Add next element, remove first element of previous window
                maxSum = Math.max(maxSum, windowSum);
            }
            return maxSum;
        }

        public static void main(String[] args)
        {
            int[] arr = {5, 7, 1, 4, 6, 3, 2, 9, 2, 7};
            int k = 5;
            System.out.println("Maximum sum of 5 consecutive elements: " +
                                                    maxSumSubarray(arr,
                                                    k));
        }
}
```

Variable-Size Sliding Window:

We refer to the flexible-size sliding window technique as the two-pointers technique. We'll take an example of this technique to better explain it too.

Problem:

You are given **n books** placed in a row, where each book takes a certain number of minutes to read. You also have **k free minutes** available. Your goal is to read the **maximum number of consecutive books** without exceeding the total available time **k**.

In other words, you need to find the longest **continuous subarray** of books whose total reading time does not exceed **k**.

Input:

- An integer array books[], where books[i] represents the time required to read the i-th book.
- An integer k, representing the total time available.

Output:

- An integer representing the **maximum number of consecutive books** that can be read within **k** minutes.

Naive Approach (Brute Force)

Approach

1. Iterate over all possible **starting points** in the array.
2. For each starting point, iterate over all possible **ending points**.
3. Compute the sum of the selected subarray and check if it is ≤ **k**.

4. If valid, update the maximum number of books that can be read.

Time Complexity

- Since we check all subarrays, the worst case requires **$O(N^2)$** operations.

Java Code:

```
public class NaiveMaxBooks
{
	public static int maxBooksNaive(int[] books, int k)
	{
		int n = books.length;
		int maxBooks = 0;

		// Try all possible subarrays (i to j)
		for (int i = 0; i < n; i++)
		{
			int sum = 0;
			for (int j = i; j < n; j++)
			{
				sum += books[j];

				if (sum > k) break; // Stop if sum exceeds k
				maxBooks = Math.max(maxBooks, j - i + 1);
			}
		}
		return maxBooks;
	}

	public static void main(String[] args)
	{
		int[] books = {3, 1, 2, 1, 4, 5, 2};
		nt k = 6;
		System.out.println("Max books (Naïve): " + maxBooksNaive(books, k));
	}
}
```

Sliding Window Approach:

1. We maintain a **window** of books (a subarray) whose total reading time is **≤ k**.
2. We **expand the window** from the right (add books) as long as the total reading time stays within **k**.
3. If the total reading time exceeds **k**, we **shrink the window** from the left (remove books) until we satisfy the constraint.
4. At every step, we **track the maximum window size** encountered.

Step-by-Step Execution with Example:

books = {3, 1, 2, 1, 4, 5, 2}

k = 6

Visualization of Sliding Window Progress

Step	Right Pointer (right)	Left Pointer (left)	Window (books read)	Current Sum	Max Books
1	0 → 3	0	[3]	3	1
2	1 → 1	0	[3, 1]	4	2
3	2 → 2	0	[3, 1, 2]	6	3
4	3 → 1	0	[3, 1, 2, 1]	7 (Exceeds k)	Shrink
5	3 → 1	1	[1, 2, 1]	4	3
6	4 → 4	1	[1, 2, 1, 4]	8 (Exceeds k)	Shrink
7	4 → 4	2	[2, 1, 4]	7 (Exceeds k)	Shrink
8	4 → 4	3	[1, 4]	5	3
9	5 → 5	3	[1, 4, 5]	10 (Exceeds k)	Shrink
10	5 → 5	4	[4, 5]	9 (Exceeds k)	Shrink
11	5 → 5	5	[5]	5	3
12	6 → 2	5	[5, 2]	7 (Exceeds k)	Shrink
13	6 → 2	6	[2]	2	3

Final Answer: 3 (Max books that can be read)

The maximum number of books that can be read consecutively within k = 6 minutes is 3.

Complexity Analysis

Approach	Time Complexity	Why?
Naïve (Brute Force)	$O(N^2)$	Tries all subarrays
Sliding Window	O(N)	Moves left & right pointers only

Program Code:

```
public class SlidingWindowMaxBooks
{
	public static int maxBooksSlidingWindow(int[] books, int k)
	{
		int n = books.length;
		int maxBooks = 0, sum = 0, left = 0;

		for (int right = 0; right < n; right++)
		{
			sum += books[right]; // Expand window by adding right book

			// If sum exceeds k, shrink the window from the left
			while (sum > k)
			{
```

```
                    sum -= books[left];
                    left++;
                }

                // Update maxBooks with the current window size
                maxBooks = Math.max(maxBooks, right - left + 1);
            }

            return maxBooks;
        }

        public static void main(String[] args)
        {
            int[] books = {3, 1, 2, 1, 4, 5, 2};
            int k = 6;
            System.out.println("Max books (Sliding Window): " +
                                                maxBooksSlidingWindow(books, k));
        }
}
```

Applications:

1. **Diet Plan Performance.**
2. **Distinct Numbers in Each Subarray.**
3. **Kth Smallest Subarray Sum.**
4. **Maximum of all subarrays of size k.**

1. Diet Plan Performance

A dieter consumes calories[i] calories on the i-th day.

Given an integer k, for **every** consecutive sequence of k days (calories[i], calories[i+1], ..., calories[i+k-1] for all 0 <= i <= n-k), they look at *T*, the total calories consumed during that sequence of k days (calories[i] + calories[i+1] + ... + calories[i+k-1]):

- If T < lower, they performed poorly on their diet and lose 1 point;
- If T > upper, they performed well on their diet and gain 1 point;
- Otherwise, they performed normally and there is no change in points.

Initially, the dieter has zero points. Return the total number of points the dieter has after dieting for calories.length days.

Note that the total points can be negative.

Example 1:

Input: calories = [1,2,3,4,5], k = 1, lower = 3, upper = 3

Output: 0

Explanation: Since k = 1, we consider each element of the array separately and compare it to lower and upper.

calories[0] and calories[1] are less than lower so 2 points are lost.

calories[3] and calories[4] are greater than upper so 2 points are gained.

Example 2:

Input: calories = [3,2], k = 2, lower = 0, upper = 1

Output: 1

Explanation: Since k = 2, we consider subarrays of length 2.

calories[0] + calories[1] > upper so 1 point is gained.

Example 3:

Input: calories = [6,5,0,0], k = 2, lower = 1, upper = 5

Output: 0

Explanation:

calories[0] + calories[1] > upper so 1 point is gained.

lower <= calories[1] + calories[2] <= upper so no change in points.

calories[2] + calories[3] < lower so 1 point is lost.

Constraints:

- 1 <= k <= calories.length <= 10^5

- 0 <= calories[i] <= 20000
- 0 <= lower <= upper

Solution

- Use the idea of sliding window. Initially, calculate the calories consumed during the first consecutive k days, which is calories[0] + calories[1] + ... + calories[k - 1]. Let sum be the calories consumed during the first consecutive k days.
- If sum < lower, then lose 1 point.
- If sum > upper, then gain 1 point.
- Each time, remove the first element from the window and add the next element into the window and calculate the sum, and decide whether the point is increased, decreased or unchanged.

Java Program for DietPlanPerformance using Sliding Window Technique:

DietPlanPerformance.java

```
import java.util.*;

class DietPlanPerformance
{
	public int dietPlanPerformance(int[] calories, int k, int lower, int upper)
	{
		int points = 0;
		int sum = 0;
		for (int i = 0; i < k; i++)
			sum += calories[i];
		if (sum > upper)
			points++;
		else if (sum < lower)
			points--;
		int length = calories.length;
		for (int i = k; i < length; i++)
		{
			sum += calories[i];
			sum -= calories[i - k];
			if (sum > upper)
				points++;
			else if (sum < lower)
				points--;
		}
		return points;
	}

	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
```

```
                int n=sc.nextInt();
                int calories[]=new int[n];

                for(int i=0;i<n;i++)
                {
                        calories[i]=sc.nextInt();
                }
                int k=sc.nextInt();
                int l=sc.nextInt();
                int u=sc.nextInt();
                System.out.println(new
                DietPlanPerformance().dietPlanPerformance(calories,k,l,u));
        }
}
```

Test Case-1

input=6

6 5 4 3 2 1

4 1 6

output=3

Test Case-1

input=10

9 18 27 36 45 54 63 72 81 90

5 72 81

output=6

2. Distinct Numbers in Each Subarray:

Given an integer array nums and an integer k, you are asked to construct the array ans of size n-k+1 where ans[i] is the number of **distinct** numbers in the subarray nums[i:i+k-1] = [nums[i], nums[i+1], ..., nums[i+k-1]].

Return *the array* ans.

Example 1:

Input: nums = [1,2,3,2,2,1,3], k = 3

Output: [3,2,2,2,3]

Explanation: The number of distinct elements in each subarray goes as follows:

- nums[0:2] = [1,2,3] so ans[0] = 3
- nums[1:3] = [2,3,2] so ans[1] = 2
- nums[2:4] = [3,2,2] so ans[2] = 2
- nums[3:5] = [2,2,1] so ans[3] = 2
- nums[4:6] = [2,1,3] so ans[4] = 3

Example 2:

Input: nums = [1,1,1,1,2,3,4], k = 4

Output: [1,2,3,4]

Explanation: The number of distinct elements in each subarray goes as follows:

- nums[0:3] = [1,1,1,1] so ans[0] = 1
- nums[1:4] = [1,1,1,2] so ans[1] = 2
- nums[2:5] = [1,1,2,3] so ans[2] = 3
- nums[3:6] = [1,2,3,4] so ans[3] = 4

Constraints:

- $1 <= k <= nums.length <= 10^5$
- $1 <= nums[i] <= 10^5$

The time complexity to **O(n)** by using the **sliding window** technique

The idea is to store the frequency of elements in the current window in a map and keep track of the distinct elements count in the current window (of size `k`). The code can be optimized to derive the count of elements in any window using the count of elements in the previous window by inserting the new element to the previous window from its right and removing an element from its left.

Java Program for Distinct Numbers in Each Subarray using Sliding Window Technique:

DistinctCount.java

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

class DistinctCount
{
	// Function to find the count of distinct elements in every subarray/ of size `k` in the
	array
	public static void findDistinctCount(int[] A, int k)
	{
		// map to store the frequency of elements in the current window of size `k`
		Map<Integer, Integer> freq = new HashMap<>();

		// maintains the count of distinct elements in every subarray of size `k`
		int distinct = 0;

		// loop through the array
		for (int i = 0; i < A.length; i++)
		{
			// ignore the first `k` elements
			if (i >= k)
			{
				/* remove the first element from the subarray by reducing its
				frequency in the map*/
				freq.put(A[i - k], freq.getOrDefault(A[i - k], 0) - 1);

				// reduce the distinct count if we are left with 0
				if (freq.get(A[i - k]) == 0)
				{
					distinct--;
				}
			}
			// add the current element to the subarray by incrementing its count in the
			map
			freq.put(A[i], freq.getOrDefault(A[i], 0) + 1);

			/* increment distinct count by 1 if element occurs for the first time in
			the current window */
			if (freq.get(A[i]) == 1)
			{
				distinct++;
			}

			// print count of distinct elements in the current subarray
			if (i >= k - 1)
```

```
                {
                        System.out.println(distinct);
                }
        }
    }
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        int array[]=new int[n];
        int k=sc.nextInt();
        for(int i=0;i<n;i++)
        {
            array[i]=sc.nextInt();
        }
        findDistinctCount(array, k);
    }
}
```

Test Case-1:

Input: 7 3

2 3 4 3 3 2 4

Output : 3 2 2 2 3

Test Case-2:

Input =15 4

4 1 5 3 2 3 4 2 3 1 3 3 2 4 3

Output =4 4 3 3 3 3 4 3 2 3 3 3

3. Kth Smallest Subarray Sum

Given an integer array nums of length n and an integer k, return *the k-th* ***smallest subarray sum***.

A **subarray** is defined as a **non-empty** contiguous sequence of elements in an array. A **subarray sum** is the sum of all elements in the subarray.

Example 1:

Input: nums = [2,1,3], k = 4

Output: 3

Explanation: The subarrays of [2,1,3] are:

- [2] with sum 2
- [1] with sum 1
- [3] with sum 3
- [2,1] with sum 3
- [1,3] with sum 4
- [2,1,3] with sum 6

Ordering the sums from smallest to largest gives 1, 2, 3, 3, 4, 6. The 4th smallest is 3.

Example 2:

Input: nums = [3,3,5,5], k = 7

Output: 10

Explanation: The subarrays of [3,3,5,5] are:

- [3] with sum 3
- [3] with sum 3
- [5] with sum 5
- [5] with sum 5
- [3,3] with sum 6
- [3,5] with sum 8
- [5,5] with sum 10
- [3,3,5], with sum 11
- [3,5,5] with sum 13
- [3,3,5,5] with sum 16

Ordering the sums from smallest to largest gives 3, 3, 5, 5, 6, 8, 10, 11, 13, 16. The 7th smallest is 10.

Constraints:

- n == nums.length

- 1 <= n <= 2 * 10^4
- 1 <= nums[i] <= 5 * 10^4
- 1 <= k <= n * (n + 1) / 2

Solution

Use binary search. The maximum subarray sum is the sum of all elements in nums and the minimum subarray sum is the minimum element in nums. Initialize high and low as the maximum subarray sum and the minimum subarray sum. Each time let mid be the mean of high and low and count the number of subarrays that have sum less than or equal to mid, and adjust high and low accordingly. Finally, the k-th smallest subarray sum can be obtained.

To count the number of subarrays that have sum less than or equal to mid, **use sliding window** over nums and for each index, count the number of subarrays that end at the index with sum less than or equal to mid.

Write a java Program for Kth Smallest Subarray Sum using Sliding Window Technique:

KthSmallestSubarraySum.java

```
import java.util.*;
class KthSmallestSubarraySum
{
	public int kthSmallestSubarraySum(int[] nums, int k)
	{
		int min = Integer.MAX_VALUE, sum = 0;
		for (int num : nums)
		{
			min = Math.min(min, num);
			sum += num;
		}
		int low = min, high = sum;
		while (low < high)
		{
			int mid = (low+high) / 2 ;
			int count = countSubarrays(nums, mid);
			if (count < k)
			low = mid + 1;
			else
			high = mid;
		}
		return low;
	}
	public int countSubarrays(int[] nums, int mid)
	{
		int count = 0;
		int sum = 0;
		int length = nums.length;
		int left = 0, right = 0;
```

```
                while (right < length)
                {
                        sum += nums[right];
                        while (sum > mid)
                        {
                                sum -= nums[left];
                                left++;
                        }
                        count += right - left + 1;
                        right++;
                }
                return count;
        }
        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);

                int n=sc.nextInt();
                int array[]=new int[n];
                int k=sc.nextInt();

                for(int i=0;i<n;i++)
                        array[i]=sc.nextInt();

                System.out.println( new
                                KthSmallestSubarraySum().kthSmallestSubarraySum(array,
                                k));
        }
}
```

Test Case1:

Input:

3 4
3 2 4

Output:

5

Explanation:

The subarrays of 3 2 4 are:
1st subarray: 3 the sum is 3
2nd subarray: 2 the sum is 2
3rd subarray: 4 the sum is 4
4th subarray: 3,2 the sum is 5
5th subarray: 2,4 the sum is 6
6th subarray: 3,2,4 the sum is 9

The 4th smallest is 5

4. Maximum of all subarrays of size k

You are given an array of integers nums, 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 max sliding window.

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]

Example-3

Input: a[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}, k = 3

Output: 3 3 4 5 5 5 6

Explanation:

Maximum of subarray {1, 2, 3} is 3
Maximum of subarray {2, 3, 1} is 3
Maximum of subarray {3, 1, 4} is 4
Maximum of subarray {1, 4, 5} is 5
Maximum of subarray {4, 5, 2} is 5
Maximum of subarray {5, 2, 3} is 5
Maximum of subarray {2, 3, 6} is 6

Time Complexity: O(n)

Solution:

This problem is a variant of the problem First negative number in every window of size k.

If you closely observe the way we calculate the maximum in each k-sized subarray, you will notice that we're doing repetitive work in the inner loop. Let's understand it with an example:

Input: a[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}, k = 3

For the above example, we first calculate the maximum in the subarray {1, 2, 3}, then we move on to the next subarray {2, 3, 1} and calculate the maximum. Notice that, there is an overlap in both the subarrays:

1 2 3

2 3 1

So, we're calculating the maximum in the overlapping part (2, 3) twice. We can utilize the **sliding window technique** to save us from re-calculating the maximum in the overlapping subarray and improve the run-time complexity.

In the sliding window technique, we consider each k-sized subarray as a sliding window. To calculate the maximum in a particular window (2, 3, 1), we utilize the calculation done for the previous window (1, 2, 3).

Since we want to utilize the calculation from the previous window, we need a way to store the calculation for the previous window. We'll use a PriorityQueue to store the elements of the sliding window in decreasing order (maximum first).

The remaining explanation is same as the previous problem we solved using the same technique: First negative number in every window of size k.

Java Program for Maximum of all subarrays of size k using Sliding Window Technique:

MaxOfAllSubarraysOfSizeK.java

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

public class MaxOfAllSubarraysOfSizeK
{
	private static int[] maxofAllSubarray(int[] a, int k)
	{
		int n = a.length;
		int[] maxOfSubarrays = new int[n-k+1];
		int idx = 0;

		PriorityQueue<Integer> q = new
		PriorityQueue<>(Comparator.reverseOrder());
		int windowStart = 0;
		for(int windowEnd = 0; windowEnd < n; windowEnd++)
		{
			q.add(a[windowEnd]);
			if(windowEnd-windowStart+1 == k)
```

```
                {
                    /* We've hit the window size. Find the maximum in the current
                    window and Slide the window ahead*/
                    int maxElement = q.peek();
                    maxOfSubarrays[idx++] = maxElement;

                    if(maxElement == a[windowStart])
                    {
                        /* Discard a[windowStart] since we are sliding the
                        window now. So, it is going out of the window.*/
                        q.remove();
                    }

                    windowStart++; // Slide the window ahead
                }
            }
            return maxOfSubarrays;
        }

        public static void main(String[] args)
        {
            Scanner sc = new Scanner(System.in);
            int n = sc.nextInt();
            int[] a = new int[n];
            for(int i = 0; i < n; i++)
            {
                a[i] = sc.nextInt();
            }
            int k = sc.nextInt();
            int[] result = maxofAllSubarray(a, k);
            for(int i = 0; i < result.length; i++)
            {
                System.out.print(result[i] + " ");
            }
        }
}
```

Test Case-1

Input:9

1 2 3 1 4 5 2 3 6

3

Output:3 3 4 5 5 5 6

Test Case-2

input=8

1 5 7 2 8 18 12 10

2

output=5 7 7 8 18 18 12

Two Pointer Approach

Introduction:

- The **two-pointer method** is a helpful technique to keep in mind when working with strings and arrays.
- It's a clever optimization that can help reduce time complexity with no added space complexity (a win-win!) by utilizing extra pointers to avoid repetitive operations.

Why Two Pointers?

- Many questions involving data stored in arrays or linked lists ask us to find sets of data that fit a certain condition or criterion.
- Enter two pointers: we could solve these problems by brute force using a single iterator, but this often involves nesting loops, which increases the time complexity of a solution exponentially.
- Instead, we can use two pointers, or iterator variables, to track the beginning and end of subsets within the data, and then check for the condition or criterion**.**

Recognizing a Two-Pointer Problem:

- The **first clue** that a two-pointer approach might be needed is that the problem asks for a set of elements — that fit a certain pattern or constraint — to be found in a sorted array or linked list.
- In a sorted array, there is additional information about the endpoints as compared to an unsorted array.
- For example, if an array of integers is sorted in ascending order, we know that the start position of the array is the smallest or most negative integer, and the end position is the greatest or most positive.
- Therefore, if a problem presents a sorted array, it is very likely able to be solved using a two-pointer technique where the pointers are initially assigned the values of the first and last members of the array.
- An example of this is illustrated below for further clarification. In a linked list, by contrast, the most likely solution will involve pointers moving in tandem to ‘look’ for the constraint or condition of the problem.
- The **second clue** that a two-pointer approach might be needed is that the problem asks for something to be found in, inserted into, or removed from a linked list.
- Knowing the exact position or length of a linked list means moving through every node before that position (all nodes if length is needed).
- We know each node in the list is connected to the next node in the list until the tail of the list. In this case, tandem pointers or a fast and slow pointer approach will likely serve to solve the problem.
- An example of the fast and slow pointer approach is illustrated below.

Problem -1: Two Sum Problem with Sorted Input Array.

Statement:

Given a 1-indexed array of integers numbers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.

Return the indices of the two numbers, index1 and index2, added by one as an integer array [index1, index2] of length 2.

The tests are generated such that there is exactly one solution. You may not use the same element twice.

Your solution must use only constant extra space.

Example 1:

Input: N=4 numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

Example 2:

Input: N=3 numbers = [2,3,4], target = 6
Output: [1,3]
Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].

Example 3:

Input: N=2 numbers = [-1,0], target = -1
Output: [1,2]
Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1, 2].

Pseudocode

```
function(array, target)
{
        set a left pointer to the first element of the array
        set a right pointer to the last element of the array
        loop through the array; check if left and right add to target
        if sum is less than the target, increase left pointer
        if sum is greater than the target, decrease right pointer
        once their sum equals the target, return their indices
}
```

The pseudo code illustrates the most classic case of a two-pointer problem, where the pointers start out as the ends of a sorted array. In this case, we are looking for two members of an array to add to a specific target value.

Java code for Two Sum Problem with Sorted Input Array using Two pointer Approach:

```
class TwoSum
{
        public static int[] twoSum(int[] numbers, int target)
        {
                int slow = 0, fast = numbers.length - 1;
                while(slow < fast)
                {
                    int sum = numbers[slow] + numbers[fast];
                    if(sum == target)
                            return new int[]{slow + 1, fast + 1};
                    else if(sum < target)
                            slow++;
                    else
                            fast--;
                }
                return new int[]{-1, -1};
        }
        public static void main(String[] args)
        {
              Scanner sc = new Scanner(System.in);
              int n = sc.nextInt();
              int[] arr= new int[n];
              for(int i=0;i<n;i++)
                    arr[i]=sc.nextInt();
              int k = sc.nextInt();
              System.out.println(Arrays.toString(twoSum(arr, k)));
```

```
        }
}
```

Time Complexity is: O(n) & Space Complexity is: O(1).

Problem-2: Rotate Array *k* Steps

Given an array, rotate the array to the right by *k* steps, where *k* is non-negative. For example, if our input array is *[1, 2, 3, 4, 5, 6, 7]* and *k* is *4*, then the output should be *[4, 5, 6, 7, 1, 2, 3]*.

We can solve this by having two loops again which will make the time complexity *O(n^2)* or by using an extra, temporary array, but that will make the space complexity *O(n)*.

Let's solve this using the two-pointer technique instead:

```
import java.util.*;
public class ArrayRotation
{
   public static void rotateArray(int[] arr, int k)
  {
      if (arr == null || arr.length <= 1 || k % arr.length == 0)
         return;
      int n = arr.length;
      k = k % n;
      reverse(arr, 0, n - 1); // Reverse the whole array
      reverse(arr, 0, k - 1); // Reverse the first k elements
      reverse(arr, k, n - 1); // Reverse the remaining elements
   }
   public static void reverse(int[] arr, int start, int end)
  {
      while (start < end)
     {
         // Swap elements at start and end positions
         int temp = arr[start];
         arr[start] = arr[end];
         arr[end] = temp;
         // Move the pointers towards the center
         start++;
         end--;
      }
   }
   public static void main(String[] args)
   {
         Scanner sc = new Scanner(System.in);
         int n = sc.nextInt();
         int[] arr= new int[n];
         for(int i=0;i<n;i++)
                  arr[i]=sc.nextInt();
         int k = sc.nextInt();
         rotateArray(arr, k);
         System.out.println(Arrays.toString(arr));
   }
}
```

test cases

case =1

input=12

1 2 3 4 5 6 7 8 9 10 11 12

2

output=[11, 12, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

case =2

input=9

9 8 7 6 5 4 3 2 1

0

Problem-3: Middle Element in a LinkedList

Given a singly *LinkedList*, find its middle element. For example, if our input *LinkedList* is *1->2->3->4->5*, then the output should be *3*.

We can also use the two-pointer technique in other data-structures similar to arrays like a *LinkedList*:

/*Traverse linked list using two-pointers.

Move one pointer by one and the other pointers by two.

When the Right pointer reaches the end, the Left pointer will reach the middle of the linked list.*/

```
import java.util.*;

class MiddleElementLL
{
   Node head;
   Node tail;

   // Linked list node
   class Node {
      int data;
      Node next;

      Node(int d) {
         data = d;
         next = null;
      }
   }

   // Function to add a node to the linked list
   public void addNode(int data)
{
      Node newNode = new Node(data);
      if (head == null) {
         head = newNode;
         tail = newNode;
      } else {
         tail.next = newNode;
         tail = newNode;
      }
   }

         void printMiddle(){
                  Node slowPtr = head;
                  Node fastPtr = head;
                  // System.out.println(head.data);

                  while (fastPtr.next != null && fastPtr.next.next != null) {
```

```
                    fastPtr = fastPtr.next.next;
                    slowPtr = slowPtr.next;
                }
        System.out.println("The middle element is [" + slowPtr.data + "] \n");
        }

  // This function prints the contents of the linked list starting from the given node
  public void printList() {
     Node tnode = head;
     while (tnode != null) {
        System.out.print(tnode.data + "->");
        tnode = tnode.next;
     }
     System.out.println("NULL");
  }

  public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     String list[] = sc.nextLine().split(" ");
     MiddleElementLL llist = new MiddleElementLL();

     for (int i = 0; i < list.length; i++) {
        llist.addNode(Integer.parseInt(list[i]));
     }

     llist.printList();
     llist.printMiddle();
  }
}
```

Test Cases:

case =1

input= 5 10 20 15 25 12 30 35 32

output=25

case =2

input=1 2 3 4 5 6 7 8 9

output= 5

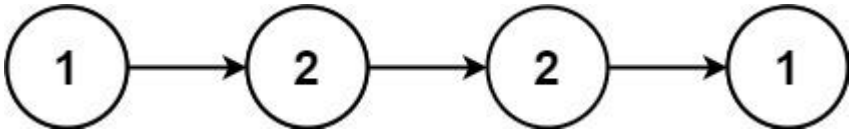
APPLICATIONS:

1. **Palindrome Linked List.**
2. **Find the Closest pair from two sorted arrays.**
3. **Valid Word Abbreviation.**

1. Palindrome Linked List:

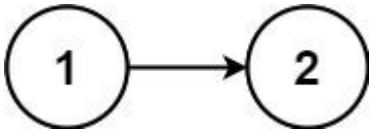
- Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:



Input:
head = [1,2,2,1]
Output:
true

Example 2:



Input:
head = [1,2]
Output:
false

Time Complexity: O(n)
Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1).

Java Program for Palindrome LinkedList using Two-Pointer Approach:

PalindromeLinkedList.java

```
import java.util.*;

class Node
{
	int data;
	Node next;

	public Node(int data)
	{
		this.data = data;
		this.next = null;
	}
}

class Solution
{
	//Function to check whether the list is palindrome.
	Node getmid (Node head)
	{
		Node slow =  head ;
		Node fast =  head.next ;

		while(fast !=  null && fast.next != null )
		{
			fast =  fast.next.next ;
			slow =  slow.next ;
		}
		return slow ;
	}

	Node reverse(Node head)
	{
		Node curr  =  head ;
		Node prev =  null  ;
		Node next =  null ;

		while(curr !=  null)
		{
			next =  curr.next ;
			curr.next  =  prev  ;
			prev = curr ;
			curr =  next ;
		}
		return prev ;
	}
```

```
boolean isPalindrome(Node head)
{
        if(head.next ==  null)
        {
                return true ;
        }

        Node middle =  getmid(head);

        Node temp  =  middle.next;
        middle.next =  reverse(temp);

        Node head1  =  head;
        Node head2 =  middle.next;

        while(head2 != null)
        {
                if(head1.data != head2.data)
                {
                        return false;
                }
                head1 =  head1.next;
                head2 =  head2.next;
        }
        temp  =  middle.next;
        middle.next =  reverse(temp);
        return true ;
        }
}

public class PalindromeList
{
        public Node head = null;
        public Node tail = null;

        public void addNode(int data)
        {
        Node newNode = new Node(data);
        if(head == null)
        {
                head = newNode;
                tail = newNode;
        }
        else
        {
                tail.next = newNode;
                tail = newNode;
        }
}
```

```
        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);
                PalindromeList list = new PalindromeList();
                String list2[]=sc.nextLine().split(" ");
                for(int i=0;i<list2.length;i++)
                        list.addNode(Integer.parseInt(list2[i]));
                Solution sl=new Solution();
                System.out.println(sl.isPalindrome(list.head));
        }
}
```

=== testcases ===
case =1
input =1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
output =true

case =2
input =1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1
output =True

case =3
input =1 2 3 4 5 6 7 8 9 8 9 7 6 5 4 3 2 1
output =false

2. Find the Closest pair from two sorted arrays:

- Given two sorted arrays and a number x, find the pair whose sum is closest to x and **the pair has an element from each array**.
- We are given two arrays ar1[0…m-1] and ar2[0..n-1] and a number x, we need to find the pair ar1[i] + ar2[j] such that absolute value of (ar1[i] + ar2[j] – x) is minimum.

Example-1:

Input: ar1[] = {1, 4, 5, 7};

ar2[] = {10, 20, 30, 40};

x = 32

Output: 1 and 30

Example-2:

Input: ar1[] = {1, 4, 5, 7};

ar2[] = {10, 20, 30, 40};

x = 50

Output: 7 and 40

Solution:

A **Simple Solution** is to run two loops. The outer loop considers every element of first array and inner loop checks for the pair in second array. We keep track of minimum difference between ar1[i] + ar2[j] and x.

We can do it **in O(n) time** using following steps.

1. Merge given two arrays into an auxiliary array of size m+n using merge process of merge sort. While merging keeps another boolean array of size m+n to indicate whether the current element in merged array is from ar1[] or ar2[].

2. Consider the merged array and use the linear time algorithm to find the pair with sum closest to x. One extra thing we need to consider only those pairs which have one element from ar1[] and other from ar2[], we use the boolean array for this purpose.

Can we do it in a single pass and O(1) extra space?

The idea is to start from left side of one array and right side of another array, and use the algorithm same as step 2 of above approach. Following is detailed algorithm.

1) Initialize a variable diff as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.

2) Initialize two index variables l and r in the given sorted array.

(a) Initialize first to the leftmost index in ar1: l = 0

(b) Initialize second the rightmost index in ar2: r = n-1

3) Loop while l< length.ar1 and r>=0

(a) If abs(ar1[l] + ar2[r] - sum) < diff then

update diff and result

(b) If (ar1[l] + ar2[r] < sum) then l++

(c) Else r--

4) Print the result.

Java program to find closest pair in an array using Two pointer approach:

ClosestPair.java

```
import java.util.*;

class ClosestPair
{
    /* arr1[0..m-1] and arr2[0..n-1] are two given sorted arrays/ and x is given number.
    This function prints the pair from both arrays such that the sum of the pair is closest
    to x.*/
    void printClosest(int ar1[], int ar2[], int m, int n, int x)
    {
        // Initialize the diff between pair sum and x.
        int diff = Integer.MAX_VALUE;
        // res_l and res_r are result indexes from ar1[] and ar2[] respectively
        int res_l = 0, res_r = 0;
        // Start from left side of ar1[] and right side of ar2[]
        int l = 0, r = n-1;
        while (l<m && r>=0)
        {
            /* If this pair is closer to x than the previously found closest, then
            update res_l, res_r and diff*/

            if (Math.abs(ar1[l] + ar2[r] - x) < diff)
            {
                res_l = l;
                res_r = r;
                diff = Math.abs(ar1[l] + ar2[r] - x);
            }
```

```
                    // If sum of this pair is more than x, move to smaller side
                    if (ar1[l] + ar2[r] > x)
                        r--;
                    else // move to the greater side
                        l++;
                }
                // Print the result
                System.out.print("The closest pair is [" + ar1[res_l] + ", " + ar2[res_r] + "]");
        }

        public static void main(String args[])
        {
                ClosestPair ob = new ClosestPair();
                Scanner sc=new Scanner(System.in);
                System.out.println("enter size of array_1");
                int n1=sc.nextInt();
                int arr1[]=new int[n1];

                System.out.println("enter the values of array_1");
                for(int i=0;i<n1;i++)
                        arr1[i]=sc.nextInt();
                System.out.println("enter size of array_2");
                int n2=sc.nextInt();
                int arr2[]=new int[n2];

                System.out.println("enter the values of array_2");
                for(int i=0;i<n2;i++)
                        arr2[i]=sc.nextInt();
                System.out.println("enter closest number");
                int x=sc.nextInt();
                ob.printClosest(arr1, arr2, n1, n2, x);
        }
}
```

Input:

```
enter size of array_1
4
enter the values of array_1
1 8 10 12
enter size of array_2
4
enter the values of array_2
2 4 9 15
```

enter closest number
11

Output:

The closest pair is [1, 9]

3. Valid Word Abbreviation:

Given a non-empty string s and an abbreviation abbr, return whether the string matches with the given abbreviation.

A string such as "word" contains only the following valid abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Notice that only the above abbreviations are valid abbreviations of the string "word". Any other string is not a valid abbreviation of "word".

Note: Assume s contains only lowercase letters and abbr contains only lowercase letters and digits.

Example 1:

Given s = "internationalization", abbr = "i12iz4n":

Return true.

Example 2:

Given s = "apple", abbr = "a2e":

Return false.

Time Complexity: O(n) where n=max(len(word),len(abbr))
Auxiliary Space: O(1).

Solution: Two Pointers

- We maintain two pointers, i pointing at word and j pointing at abbr.
- There are only two scenarios:
  - j points to a letter. We compare the value i and j points to. If equal, we increment them. Otherwise, return False.
  - j points to a digit. We need to find out the complete number that j is pointing to, e.g. 123. Then we would increment i by 123. We know that next we will:
    - either break out of the while loop if i or j is too large
    - or we will return to scenario 1.

Java program for Valid Word Abbreviation using Two Pointer approach:

ValidWordAbbreviation.java

```
import java.util.*;
class ValidWordAbbreviation
{
	public boolean validWordAbbreviation(String word, String abbr)
	{
		int i = 0, j = 0;
		while (i < word.length() && j < abbr.length())
		{
			if (word.charAt(i) == abbr.charAt(j))
			{
				++i;
				++j;
				continue;
			}
			if (abbr.charAt(j) <= '0' || abbr.charAt(j) > '9')
			{
				return false;
			}
			int start = j;
			while (j < abbr.length() && abbr.charAt(j) >= '0'&&abbr.charAt(j) <=
			'9')
			{
				++j;
			}
			int num = Integer.valueOf(abbr.substring(start, j));
			i += num;
		}
		return i == word.length() && j == abbr.length();
	}
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		System.out.println("Enter word");

		String word = in.next();
		System.out.println("Enter Abbreviation");
		String abbrv = in.next();
		System.out.println(new
				ValidWordAbbreviation().validWordAbbreviation(word,abbrv
				));
```

```
        }
}
```

Case-1:
input=
Enter word
kmit
Enter Abbreviation
4
output=true

Case=2

input=
Enter word
Diary
Enter Abbreviation
d2ary
output=false