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:

Window Sliding Technique is a computational technique which aims to reduce the use of nested loop and replace it with a single loop, thereby reducing the time complexity.

What is Sliding Window?

The Sliding window is a problem-solving technique of data structure and algorithm for problems that apply arrays or lists. 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)**.

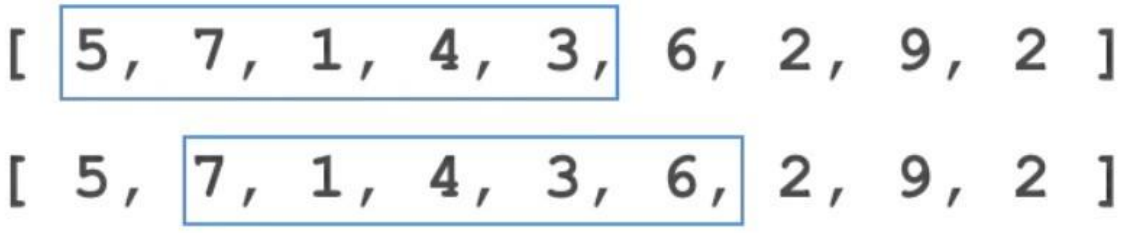


Figure 1: Sliding window technique to find the largest sum of 5 consecutive numbers.

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:

Figure 2: Array of values

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

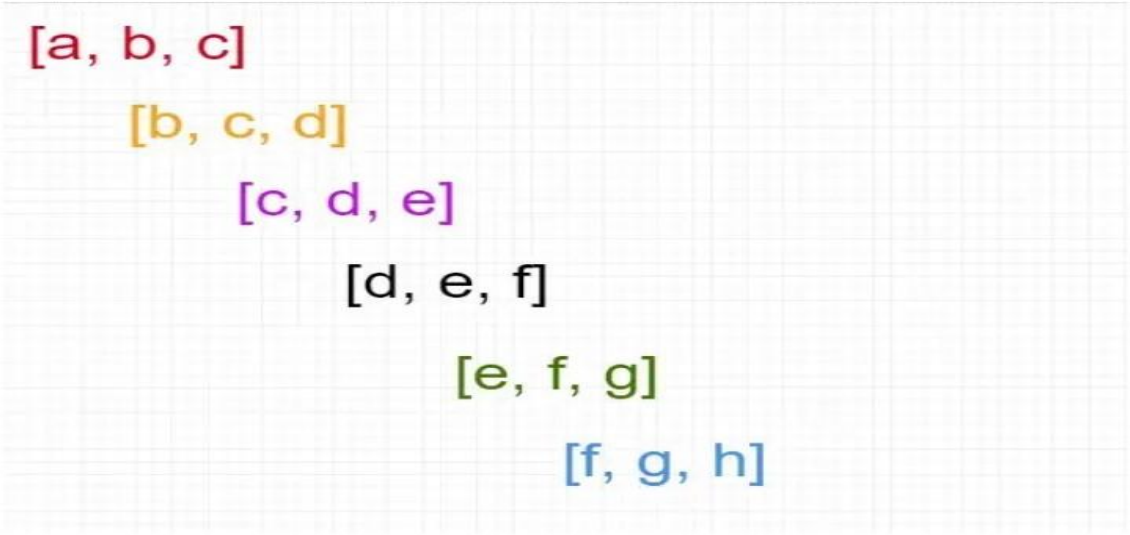


Figure 3: Sliding window of size 3 (Sublist of 3 items)

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, 3, 6, 2, 9, 2]

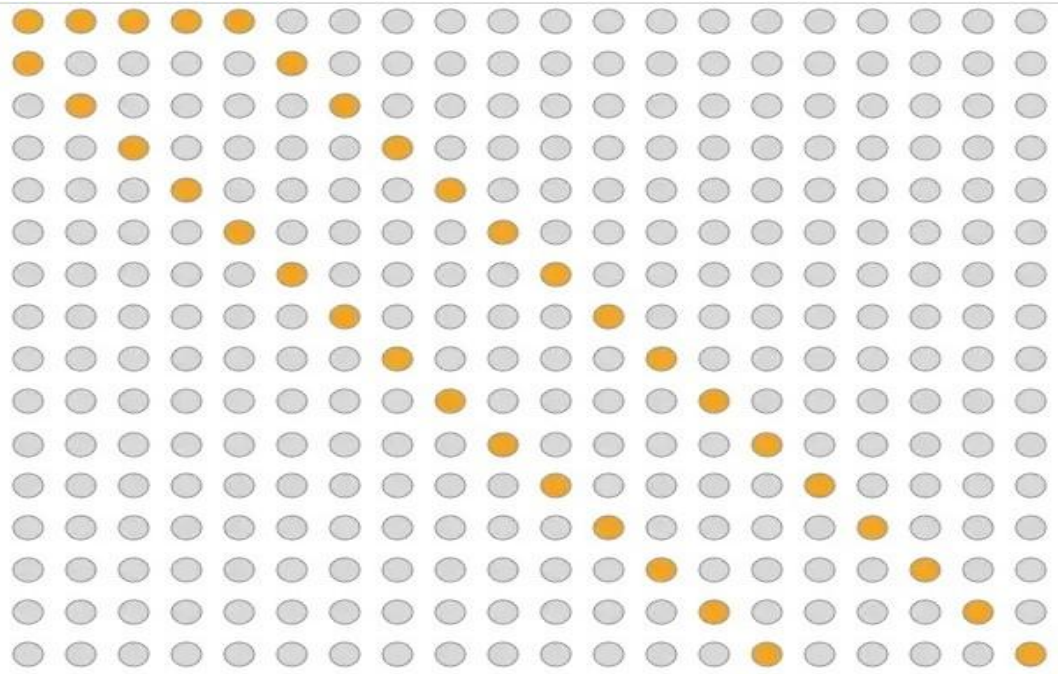


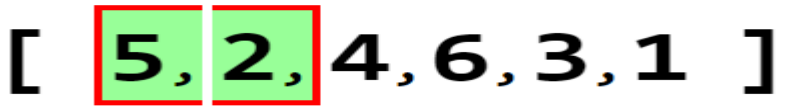
Figure 4: Sliding window technique to find the largest sum of five consecutive elements

Example: Given an array as input, extract the pair of contiguous integers that have the highest sum of all pairs. Return the pair as an array.

- Contiguous means sequential, which means the elements must be right next to each other to count as a pair. This is a **giant** clue that a sliding window may work beautifully with this question.
- In the context of this question, contiguous also means that one cannot simply sort the array and return the last two integers in an array, since we want the pair of **contiguous** integers which have the highest sum in the array as it is already ordered.

For this question, take the input [5, 2, 4, 6, 3, 1] as the input.

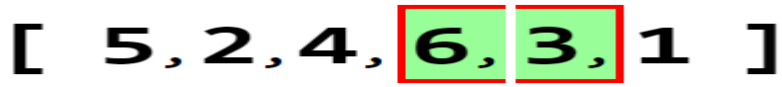
- The sliding window will not be explained thoroughly here but it will be in the next section. This part serves as a visual introduction with light notes. The following picture is of the first iteration of the sliding window for this problem.



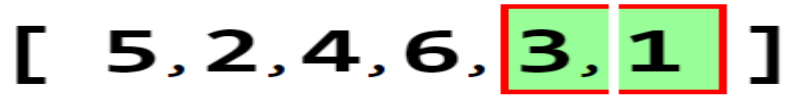
- The window has a red border and a mint background. The current sum of the pair, [5,2] is 7. Continue iterating through the entire array to see if that's still the highest sum. One must **slide** the window up by 1 to reach the next iteration.

- This window evaluates to 6 so it is not higher. Continue iterating.

- This window's pair is the highest so far. Keep track of that and compare it to the last two iterations.



- Close, but still not higher than the previous pair.



- This iteration's pair definitely was not the highest. After all the iterations, the highest sum pair was [4, 6].
- The number of iterations is worth noticing here: the number of iterations was 5, which ranks the sliding window solution for this problem slightly faster than **O(n).** For small and large inputs, linear time is an ideal goal.
- Now that the mental model is starting to form, one may be wondering **how** to recognize problems that can use a sliding window successfully.

In general, any problem where the author is asking for any of the following return values can use a sliding window:

- Minimum value
- Maximum value
- Longest value
- Shortest value
- *K*-sized value

In case the length of the ranges is fixed, we call this the fixed-size sliding window technique. However, if the lengths of the ranges are changed, we call this the flexible window size technique. We'll provide examples of both of these options.

Applications:

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

1. Naive Approach:

A. Fixed-Size Sliding Window:

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

A. 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.

Java Program for Maximum subarray sum with k size using Naïve Approach

```
import java.util.*;

class Maxsum
 {
   public int maxSum(int a[],int k)
    {
      int n=a.length;
      int sum,answer=Integer.MIN_VALUE;
      for (int i=0; i <n-k+1; i++)
      {
         sum=0;

           for (int j=i; j<=i+k-1;j++)
           {
               sum += a[j];
           }
        answer=Math.max(answer,sum);
      }

          return answer;
   }
   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();

          System.out.println(new Maxsum ().maxSum(a,k));
   }
 }
```

Example1:

input=
6
5 2 4 6 3 1
3
output=
13

Example2:

input=
8
4 -1 2 1 6 -5 3 2
3
output=
9

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.

Java Program for Maximum subarray sum with k size using Slidng windowApproach

```
import java.util.*;

class Maxsum
 {
   public int maxSum(int a[],int k)
    {
      int n=a.length;

      int sum=0;

      for (int i=0; i<k; i++)
      {
         sum=sum+a[i];
      }

      answer=sum;

       for (int i=k;i<n;i++)
            {
                sum =sum+a[i]-a[i-k];

        answer=Math.max(answer,sum);
      }

          return answer;
   }
   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();

          System.out.println(new Test().maxSum(a,k));
    }
  }
```

Example1:

input=
6
5 2 4 6 3 1
3
output=
13

Example2:

input=
8
4 -1 2 1 6 -5 3 2
3
output=
9

B. Flexible-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:

Suppose we have 'n' books aligned in a row. For each book, we know the number of minutes needed to read it. However, we only have 'k' free minutes to read.

Also, we should read some consecutive books from the row. In other words, we can choose a range from the books in the row and read them. Of course, the condition is that the sum of time needed to read the books mustn't exceed 'k'.

Therefore, the problem asks us to find the maximum number of books we can read. Namely, **we need to find a range from the array whose sum is at most 'k' such that this range's length is the maximum possible**.

Naive Approach:

Take a look at the naive approach for solving the problem:

Algorithm 3: Naive approach to maximum length range

```
Data: A: The array of time needed to read each book
      n: Length of the array
      k: Maximum number of minutes to read
Result: Returns the maximum length of a range whose sum is at
        most k
answer ← 0;
for L ← 1 to n do
    sum ← 0;
    i ← L;
    length ← 0;
    while i ≤ n AND sum + A[i] ≤ k do
        sum ← sum + A[i];
        length ← length + 1;
        i ← i + 1;
    end
    answer ← maximum(answer, length);
end
return answer;
```

- First, we initialize the best answer so far with zero. Next, we iterate over all the possible beginnings of the range. For each beginning, we iterate forward as long as we can read more books. Once we can't read any more books, we update the best answer so far as the maximum between the old one and the length of the range we found.
- In the end, we return the best answer we managed to find.
- **The complexity of this approach is $O(n^2)$** , where 'n' is the length of the array of books.

Sliding Window Algorithm:

We'll try to improve the naive approach, in order to get a linear complexity.

First, let's assume we managed to find the answer for the range that starts at the beginning of the array. The next range starts from the second index inside the array. However, the end of the second range is surely after the end of the first range.

The reason for this is that the second range doesn't use the first element. Therefore, the second range can further extend its end since it has more free time now to use.

Therefore, when moving from one range to the other, we first delete the old beginning from the current answer. Also, we try to extend the end of the current range as far as we can.

Hence, by the end, we'll iterate over all possible ranges and store the best answer we found.

The following algorithm corresponds to the explained idea:

Algorithm 4: Sliding window approach to maximum length range

```
Data: A: The array of time needed to read each book
      n: Length of the array
      k: Maximum number of minutes to be used
Result: Returns the maximum length of a range whose sum is at
        most k
answer ← 0;
sum ← 0;
R ← 1;
for L ← 1 to n do
    if L > 1 then
        sum ← sum − A[L − 1];
    end
    while R ≤ n AND sum + A[R] ≤ k do
        sum ← sum + A[R];
        R ← R + 1;
    end
    answer ← maximum(answer, R − L);
end
return answer;
```

Just as with the naive approach, we iterate over all the possible beginnings of the range. For each beginning, we'll first subtract the value of the index L-1 from the current sum.

After that, we'll try to move 'R' as far as possible. Therefore, we continue to move 'R' as long as the sum is still at most 'K'. Finally, we update the best answer so far. Since the length of the current range is R-L, we maximize the best answer with this value.

Although the algorithm may seem to have a **$O(n^2)$** complexity, let's examine the algorithm carefully. The variable 'R' always keeps its value. Therefore, it only moves forward until it reaches the value of 'n'.

Therefore, the number of times we execute the *while* loop in total is at most 'n' times.

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

C. Differences:

The main difference comes from the fact that in some problems we are asked to check a certain property among all range of the same size. On the other hand, on some other problems, we are asked to check a certain property among all ranges who satisfy a certain condition. In these cases, this condition could make the ranges vary in their length.

In case these ranges had an already known size (like our consecutive elements problem), we'll certainly go with the fixed-size sliding window technique. However, if the sizes of the ranges were different (like our book-length problem), we'll certainly go with the flexible-size sliding window technique.

Also, always keep in mind the following condition to use the sliding window technique that we covered in the beginning: We must guarantee that moving the **L** pointer forward will certainly make us either keep **R** in its place or move it forward as well.

2. 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.

Solution Approach:

The Java solution uses an efficient sliding window approach to solve the diet plan performance problem. The sliding window concept is applicable when it is required to calculate something among all contiguous subarrays of a certain length in an efficient manner.

Here is a breakdown of the implementation steps:

1. **Initial Sum Calculation**: The initial sum s of the calories for the first k days is calculated using the built-in sum() function on the slice of the first k elements of the calories list.
2. **Initial Points Calculation**: The initial sum s is then passed to the check(s) function, which compares s with the lower and upper bounds. Depending on the comparison, the function returns -1, 1, or 0, representing losing a point, gaining a point, or no change in points, respectively. This initial points value is stored in the ans variable.
3. **Sliding the Window**: The code enters a loop that will iterate starting from the k-th day to the last day. For each iteration, it adjusts the sum by subtracting the calorie count of the day that's exiting the window (calories[i - k]) and adding the count of the new day entering the window (calories[i]). This keeps the sum s up-to-date with the calorie count of the current window without having to re-calculate the sum from scratch.
4. **Update Points**: After updating the sum for the new window position, the check(s) function is used again to determine if points should be gained or lost based on the new sum. The result is added to the ans variable, which accumulates the total points.
5. **Returning the Result**: After all windows have been processed, the function returns ans, which contains the total points the dieter has after completing the diet.

This approach has a **time complexity of O(n),** where n is the number of days, and a **space complexity of O(1),** which makes it a very efficient solution.

Write a 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)
   {
        // Initialize the sum of the first 'k' elements.
       int windowSum = 0;
```

```
        // Calculate the sum of the first 'k' calories.
        for (int i = 0; i < k; ++i)
        {
            windowSum += calories[i];
        }

        // Initialize the performance points.
        int points = 0;

        // Check if the initial 'k' day period is below or above the threshold.
        if (windowSum < lower)
        {
            points--;
        }
       else if (windowSum > upper)
      {
            points++;
        }

        // Iterate through the array starting from the 'k'th day.
        for (int i = k; i < calories.length; ++i)
        {
            // Slide the window by 1: remove the first element and add the new one.
            windowSum += calories[i] - calories[i - k];

            // Adjust points based on the new sum.
            if (windowSum < lower)
           {
                points--;
            }
           else if (windowSum > upper)
           {
                points++;
            }
        }
         // Return the calculated points.
        return points;
    }

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

        System.out.println("enter calories size");
```

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

        System.out.println("enter the calroties");
          for(int i=0;i<n;i++)
          {
            calories[i]=sc.nextInt();
          }
           System.out.println("enter the days");

           int k=sc.nextInt();
            System.out.println("enter the Lower value");
             int l=sc.nextInt();
             System.out.println("enter the Upper value");
             int u=sc.nextInt();
             System.out.println(new DietPlanPerformance().dietPlanPerformance (calories,k,l,u));
    }
 }
```

Input:

enter calories size
5
enter the calroties
1 2 3 4 5
enter the days
1
enter the Lower value
3
enter the Upper value
3

Output:

0

3. 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

Solution Approach:

- We use a hash table count to record the occurrence times of each number in the subarray of length k.
- Next, we first traverse the first k elements of the array, record the occurrence times of each element, and after the traversal, we take the size of the hash table as the first element of the answer array.
- Then, we continue to traverse the array from the index k. Each time we traverse, we increase the occurrence times of the current element by one, and decrease the occurrence times of the element on the left of the current element by one.
- If the occurrence times of the left element become 0 after subtraction, we remove it from the hash table.
- Then we take the size of the hash table as the next element of the answer array, and continue to traverse.
- After the traversal, we return the answer array.

The **time complexity is O(n),** and the **space complexity is O(k)**. Where n is the length of the array nums, and k is the parameter given by the problem.

Write a java Program for Distinct Numbers in Each Subarray using Sliding Window Technique:

DistinctCount.java

```
import java.util.*;

class DistinctCount
{
  // Function to find the count of distinct elements in every subarray/ of size `k` in the array
   public static void findDistinctCount(int arr[], int K)
   {

      // Creates an empty hashMap hM
      HashMap<Integer, Integer> hM = new HashMap<Integer, Integer>();

      // Traverse the first window and store count of every element in hash map

      for (int i = 0; i < K; i++)

         hM.put(arr[i], hM.getOrDefault(arr[i], 0) + 1);

      // Print count of first window
      System.out.print(hM.size()+" ");

      // Traverse through the remaining array
      for (int i = K; i < arr.length; i++)
      {
          // Remove first element of previous window If there was only one occurrence

         if (hM.get(arr[i - K]) == 1)
        {
            hM.remove(arr[i - K]);
         }

         else // reduce count of the removed element
            hM.put(arr[i - K], hM.get(arr[i - K]) - 1);

      // Add new element of current window If this element appears first time, set its count as 1

         hM.put(arr[i], hM.getOrDefault(arr[i], 0) + 1);

         // Print count of current window
         System.out.print(hM.size()+" ");
```

```
        }
    }

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

Input:

```
enter array elements size
7
enter the elements
1 2 3 2 2 1 3
enter the subarray size
3
```

Output:

```
3 2 2 2 3
```

4. 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.

Solution Approach

The kthSmallestSubarraySum function in the provided Java code leverages binary search, which is a classic optimization for problems involving sorted arrays or monotonic functions. In this case, the monotonic function is the number of subarrays with sums less than or equal to a given value.

The code defines a helper function f(s) to use within the binary search. We can describe what each part of this function does and how it integrates with the binary search mechanism:

1. **Sliding Window via Two Pointers**
   - f(s) uses a sliding window with two pointers, i and j, which represent the start and end of the current subarray being considered.
   - As we iterate over the array with i, we add each nums[i] to the temporary sum t.

- If the sum t exceeds the value s, we subtract elements starting from nums[j] and increment j to reduce t. This maintains the window where the subarray sum is less than or equal to s.

2. **Counting Subarrays**
   - The line cnt += i - j + 1 is crucial. It adds the number of subarrays ending at i for which the sum is less than or equal to s.
   - This works because for every new element added to the window, there are i - j + 1 subarrays. For instance, if our window is [nums[j], ..., nums[i]], then [nums[i]], [nums[i-1], nums[i]], up to [nums[j], ..., nums[i]] are all valid subarrays.
3. **Binary Search**
   - The actual search takes place over a range of potential subarray sums, from min(nums) to sum(nums). We use the binary search algorithm to efficiently find the smallest sum that has at least k subarrays less than or equal to it.
   - The bisect_left function from the bisect module is used with a custom key function, which is the f(s) we defined earlier. The key function transforms the value we're comparing against in the binary search.
   - bisect_left will find the smallest value within the range [l, r] for which f(s) returns True, indicating it is the kth smallest sum.

The combination of these techniques results in an efficient solution where, rather than having to explicitly sort all the subarray sums, we can deduce the kth smallest sum by narrowing down our search space logarithmically with binary search and counting subarrays linearly with the two-pointer technique.

Time Complexity:
Combining the binary search and the sliding window, the total time complexity is O(n * log(r - l)).

Write a java Program for KthSmallestSubarraySum using Sliding Window Technique:

KthSmallestSubarraySum.java

```
import java.util.*;

class KthSmallestSubarraySum
{
  public int kthSmallestSubarraySum(int[] nums, int k)
    {
     // Initialize the left and right boundaries for the binary search.
    // Assume the smallest subarray sum is large, and find the smallest element of the array.

    int left = Integer.MAX_VALUE, right = 0;
       for (int num : nums)
       {
          left = Math.min(left, num);
          right += num;
       }
```

```
    // Perform binary search to find the kth smallest subarray sum.
    while (left < right)
    {
        int mid = left + (right - left) / 2;
        // If there are more than k subarrays with a sum <= mid, move the right pointer.
        if (countSubarraysWithSumAtMost(nums, mid) >= k)
        {
            right = mid;
        }
        else
        {
            // Otherwise, move the left pointer.
            left = mid + 1;
        }
    }
    // The left pointer points to the kth smallest subarray sum.
    return left;
}
// Helper method to count the number of subarrays with a sum at most 's'.
private int countSubarraysWithSumAtMost(int[] nums, int s)
 {
    int currentSum = 0, start = 0;
    int count = 0;
    for (int end = 0; end < nums.length; ++end)
     {
        currentSum += nums[end];
        // If the current sum exceeds 's', shrink the window from the left.
        while (currentSum > s)
        {
            currentSum -= nums[start++];
        }
        // Add the number of subarrays ending at index 'end' with a sum at most 's'.
        count += end - start + 1;
    }
    return count;
}
 public static void main(String[] args)
{
Scanner sc=new Scanner(System.in);
    System.out.println("enter array elements size");
    int n=sc.nextInt();
     int array[]=new int[n];
    System.out.println("enter the elements");
      for(int i=0;i<n;i++)
       {
```

```
            array[i]=sc.nextInt();
          }
           System.out.println("enter the kth samllest subarray size");
           int k=sc.nextInt();
          System.out.println(new KthSmallestSubarraySum().kthSmallestSubarraySum(array, k));
    }
}
```

Input:

enter array elements size
3
enter the elements
2 1 3
enter the kth samllest subarray size
4

Output:

3

5. 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]

Solution Approach:

Create a Deque, dq of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on right side of it in current window. Process all array elements one by one and maintain dq to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the dq is the largest and element at rear/back of dq is the smallest of current window.

Step-by-Step Algorithm:

- Create a deque to store only useful elements of current window.
- Run a loop and insert the first **k** elements in the deque. Before inserting the element, check if the element at the back of the queue is **smaller** than the current element, if it is so remove the element from the back of the deque until all elements left in the deque are **greater** than the current element. Then insert the current element, at the back of the deque.
- Now, run a loop from k to the end of the array.
  - Print the **front** element of the deque.
  - Remove the element from the **front** of the queue if they are out of the current window.
  - Insert the **next** element in the deque. Before inserting the element, check if the element at the back of the queue is **smaller** than the current element, if it is so remove the element from the back of the deque until all elements left in the deque are **greater** than the current element. Then insert the current element, at the **back** of the deque.
  - Print the maximum element of the last window.

Time Complexity: O(n). It seems more than O(n) at first look. It can be observed that every element of the array is added and removed at most once. So there are a total of 2n operations.
Auxiliary Space: O(k). Elements stored in the dequeue take O(k) space.

Write a java Program for Maximum of all subarrays of size k using Sliding Window Technique (Using Deque): **MaxOfAllSubarraysOfSizeK.java**

```
import java.util.Deque;
import java.util.*;

class Maximumofallsubarraysofsizek
{
   // A Dequeue (Double ended queue) based method for printing maximum element of all
//subarrays of size K

 public static void printMax(int arr[], int N, int K)
    {
// Create a Double Ended Queue, Qi that will store indexes of array elements The queue will
//store indexes of useful elements in every window and it will maintain decreasing order of
//values from front to rear in Qi, i.e., arr[Qi.front[]] to arr[Qi.rear() are sorted in decreasing
//order

    Deque<Integer> Qi = new LinkedList<Integer>();

        int i;
        for (i = 0; i < K; ++i)
         {
             while (!Qi.isEmpty() && arr[i] >= arr[Qi.peekLast()])
               Qi.removeLast();
               Qi.addLast(i);
        }

        // Process rest of the elements, i.e., from arr[k] to arr[n-1]
        for (i=K; i < N; ++i)
        {
          // The element at the front of the queue is the largest element of previous window, so print it
            System.out.print(arr[Qi.peek()] + " ");

            // Remove the elements which are out of this window
            while ((!Qi.isEmpty()) && Qi.peek() <= i - K)
             Qi.removeFirst();

  // Remove all elements smaller than the currently being added element (remove useless elements)
              while ((!Qi.isEmpty())  && arr[i] >= arr[Qi.peekLast()])
               Qi.removeLast();
```

```
            // Add current element at the rear of Qi
            Qi.addLast(i);
        }
        // Print the maximum element of last window
        System.out.print(arr[Qi.peek()]);
    }
     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();
                    printMax(a,n,k);
    }
}
```

(OR)

Using PriorityQueue

```
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);
System.out.println("enter array elements size");
      int n = sc.nextInt();

      int[] a = new int[n];
       System.out.println("enter the elements");
      for(int i = 0; i < n; i++) {
         a[i] = sc.nextInt();
      }
      System.out.println("enter the subarray size");
      int k = sc.nextInt();
      sc.close();
      int[] result = maxofAllSubarray(a, k);
      for(int i = 0; i < result.length; i++) {
         System.out.print(result[i] + " ");
      }
      System.out.println();
   }
}
```

Input:

```
enter array elements size
9
enter the elements
1 2 3 1 4 5 2 3 6
enter the suarray size
3
```

Output:

```
3 3 4 5 5 5 6
```

Part-2: 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: 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: 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: 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
  sum is less than the target, increase left pointer
  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 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};
  }
}
```

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:

```
public void rotate(int[] input, int step)
{
   step %= input.length;
```

```
    reverse(input, 0, input.length - 1);
    reverse(input, 0, step - 1);
    reverse(input, step, input.length - 1);
}

private void reverse(int[] input, int start, int end)
{
    while (start < end) {
        int temp = input[start];
        input[start] = input[end];
        input[end] = temp;
        start++;
        end--;
    }
}
```

- In the above methods, we reverse the sections of the input array in-place, multiple times, to get the required result. For reversing the sections, we used the two-pointer approach where swapping of elements was done at both ends of the array section.
- Specifically, we first reverse all the elements of the array. Then, we reverse the first *k* elements followed by reversing the rest of the elements. **The time complexity of this solution is *O(n)* and space complexity is *O(1)***.

<u>Problem-3</u>: 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*:

```
public <T> T findMiddle(MyNode<T> head)
{
MyNode<T> slowPointer = head;
MyNode<T> fastPointer = head;
while (fastPointer.next != null && fastPointer.next.next != null)
{
 fastPointer = fastPointer.next.next;
slowPointer = slowPointer.next;
 }
return slowPointer.data;
 }
```

- In this approach, we traverse the linked list using two pointers. One pointer is incremented by one while the other is incremented by two.
- When the fast pointer reaches the end, the slow pointer will be at the middle of the linked list. **The time complexity of this solution is *O(n)*, and space complexity is *O(1)*.**

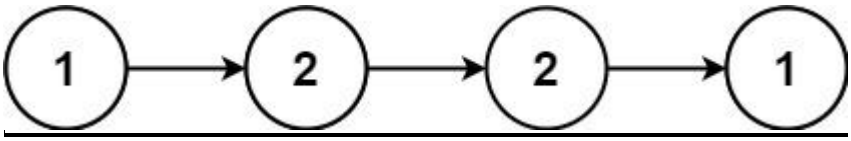
APPLICATIONS:

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

1. Palindrome Linked List:

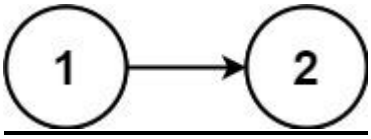
- Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:



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

Example 2:



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

The solution approach follows the intuition which is broken down into the following algorithms and patterns:

1. **Two-pointer technique:** To find the middle of the list, we use two pointers (slow and fast). The slow pointer is incremented by one node, while the fast pointer is incremented by two nodes on each iteration. When the fast pointer reaches the end, slow will be pointing at the middle node.

1.slow, fast = head, head.next

2.while fast and fast.next:
3.slow, fast = slow.next, fast.next.next

2. **Reversing the second half of the list:** Once we have the middle node, we reverse the second half of the list starting from slow.next. To do this, we initialize two pointers pre (to keep track of the previous node) and cur (the current node). We then iterate until cur is not None, each time setting cur.next to pre, effectively reversing the links between the nodes.

1. pre, cur = None, slow.next
2. while cur:
3. t = cur.next
4. cur.next = pre
5. pre, cur = cur, t

3. **Comparison of two halves:** After reversing the second half, pre will point to the head of the reversed second half. We compare the values of the nodes starting from head and pre. If at any point the values differ, we return False indicating that the list is not a palindrome. Otherwise, we keep advancing both pointers until pre is None. If we successfully reach the end of both halves without mismatches, the list is a palindrome, so we return True.

```
1 while pre:
2    if pre.val != head.val:
3        return False
4    pre, head = pre.next, head.next
5 return True
```

The code uses the two-pointer technique and the reversal of a linked list to solve the problem very effectively.

The total time complexity of the algorithm is O(n), and the **space complexity is O(1),** because no additional space is used proportional to the input size; we're just manipulating the existing nodes in the linked list.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the linked list [1 -> 2 -> 3 -> 2 -> 1]. The goal is to determine if this list represents a palindrome.

Step 1: Finding the Middle

We use the two-pointer technique. Initially, both slow and fast point to the first element, with fast moving to the next immediately for comparison purposes.

1. Initial state:
2. slow -> 1
3. fast -> 2
4. List: 1 -> 2 -> 3 -> 2 -> 1

Now we begin traversal:

1. Iteration 1:
2. slow -> 2
3. fast -> 3

4. List: 1 -> 2 -> 3 -> 2 -> 1
5. Iteration 2:
6. slow -> 3
7. fast -> 1 (fast reaches the end of the list so we stop here)
8. List: 1 -> 2 -> 3 -> 2 -> 1

At this stage, slow is pointing to the middle of the list.

Step 2: Reverse the Second Half

Starting from the middle node (where slow is currently pointing), we proceed to reverse the second half of the list. We'll use two pointers pre and cur to achieve this:

1. pre points to None
2. cur points to 3 (slow.next)
3. List: 1 -> 2 -> 3 -> 2 -> 1

We now iterate and reverse the link between the nodes until cur is None:

1. Iteration 1:
2. pre -> 3
3. cur -> 2
4. Reversed part: None <- 3 List: 1 -> 2 -> 3 -> 2 -> 1
5. Iteration 2:
6. pre -> 2
7. cur -> 1
8. Reversed part: None <- 3 <- 2 List: 1 -> 2 -> 3 -> 2 -> 1
9. Iteration 3:
10. pre -> 1
11. cur -> None
12. Reversed part: None <- 3 <- 2 <- 1

After reversing, we have pre pointing to the new head of the reversed second half, which is the node with the value 1.

Step 3: Compare Two Halves

We now have two pointers, head pointing to the first node of the list and pre pointing to the head of the reversed second half. We need to compare the values of both halves:

1. pre points to 1, head points to 1

We move both pointers and compare their values:

1. pre -> 2, head -> 2 (values match, move forward)
2. pre -> 3, head -> 3 (values match, move forward)

When pre becomes None, we've successfully compared all nodes of the reversed half with the corresponding nodes of the first half and found that all the values match, which implies that the list represents a palindrome. Hence, we return True.

When implementing these steps in a programming language like Python, the overall result of this example would be that the function confirms the linked list [1 -> 2 -> 3 -> 2 -> 1] is indeed a palindrome.

Java Program for PalindromeLinkedList 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;
      }   list
      else
     {
```

```
            tail.next = newNode;
            tail = newNode;
        }
    }

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

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

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 keep 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 4 5 7
enter size of array_2
4
enter the values of array_2
10 20 30 40
enter closest number
32
```

Output:

```
The closest pair is [1, 30]
```

3. Valid Word Abbreviation:

A string can be abbreviated by replacing any number of non-adjacent, non-empty substrings with their lengths.
The lengths should not have leading zeros.

For example, a string such as "substitution" could be abbreviated as (but not limited to):

"s10n" ("s ubstitutio n")
"sub4u4" ("sub stit u tion")
"12" ("substitution")
"su3i1u2on" ("su bst i t u ti on")
"substitution" (no substrings replaced)

The following are not valid abbreviations:

"s55n" ("s ubsti tutio n", the replaced substrings are adjacent)
"s010n" (has leading zeros)
"s0ubstitution" (replaces an empty substring)

Given a string word and an abbreviation abbr, return whether the string matches the given abbreviation.

A substring is a contiguous non-empty sequence of characters within a string.

Example 1:
Given s = "internationalization", abbr = "i12iz4n":
Return true.

Example 2:
Given s = "apple", abbr = "a2e":
Return false.

Time Complexity: O(n)
Auxiliary Space: O(1).

Sloution Approach:

We can directly simulate character matching and replacement.

Assume the lengths of the string word and the string abbr are m and n respectively. We use two pointers i and j to point to the initial positions of the string word and the string abbrabbr respectively, and use an integer variable xx to record the current matched number in abbr.

Loop to match each character of the string word and the string abbr:

If the character abbr[j] pointed by the pointer jj is a number, if abbr[j] is '0' and xx is 0, it means that the number in abbr has leading zeros, so it is not a valid abbreviation, return false; otherwise, update x tox×10+abbr[j]−‘0′.

If the character abbr[j] pointed by the pointer j is not a number, then we move the pointer ii forward by x positions at this time, and then reset x to 0. i≥m or word[i]≠abbr[j] at this time, it means that the two strings cannot match, return false; otherwise, move the pointer ii forward by 11 position.

Then we move the pointer jj forward by 11 position, repeat the above process, until ii exceeds the length of the string wordword or jj exceeds the length of the string abbr.

Finally, if i+x equals mm and j equals nn, it means that the string wordword can be abbreviated as the string abbrabbr, return true; otherwise return false.

The time complexity is O(m+n), where mm and nn are the lengths of the string wordword and the string abbr respectively. The space complexity is O(1).

Java program for Valid Word Abbreviation using Two Pointer approach:

ValidWordAbbreviation.java

```
import java.util.*;

public class ValidWordAbbreviation
{
   public static boolean isValidWordAbbreviation(String word, String abbr)
   {
      int i = 0, j = 0;
      int m = word.length(), n = abbr.length();

      while (i < m && j < n) {
         if (Character.isDigit(abbr.charAt(j))) {
            if (abbr.charAt(j) == '0') return false; // no leading zeros
            int num = 0;
            while (j < n && Character.isDigit(abbr.charAt(j)))
            {
               /*
                *abbr = "i12iz4n"
               At j = 1, we find digit '1'.
               num = 0 * 10 + (1 - 0) = 1.
```

```
                At j = 2, we find digit '2'.
                num = 1 * 10 + (2 - 0) = 12.
                Now, num holds the value 12, meaning we skip 12 characters in the word!
                 */

                num = num * 10 + (abbr.charAt(j) - '0');
                j++;
            }
            i += num;
        } else {
            if (word.charAt(i) != abbr.charAt(j)) return false;
            i++;
            j++;
        }
    }

    return i == m && j == n;
  }
  public static void main(String args[])
  {
    Scanner sc=new Scanner(System.in);
    String word=sc.next();
    String abbrivation=sc.next();
    System.out.println(new
ValidWordAbbreviation().isValidWordAbbreviation(word,abbrivation));
  }
}
```

input=

Enter word

kmit

Enter Abbreviation

4

output=

true