

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:

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 $O(n^2)$ or $O(n^3)$. However, the **Sliding window** technique can reduce the time complexity to $O(n)$.

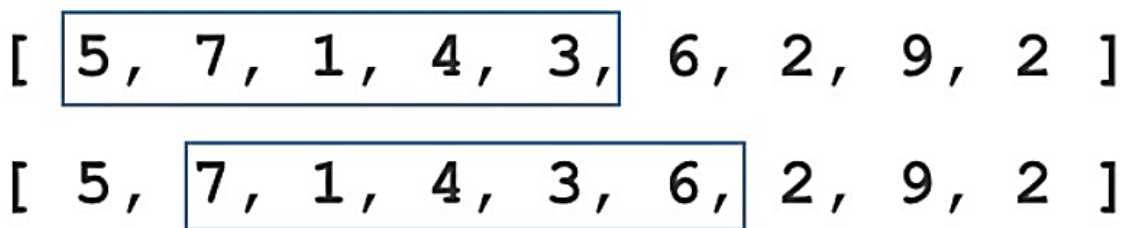


Figure 1: Sliding window technique to find the largest sum of 5 consecutive numbers.

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.

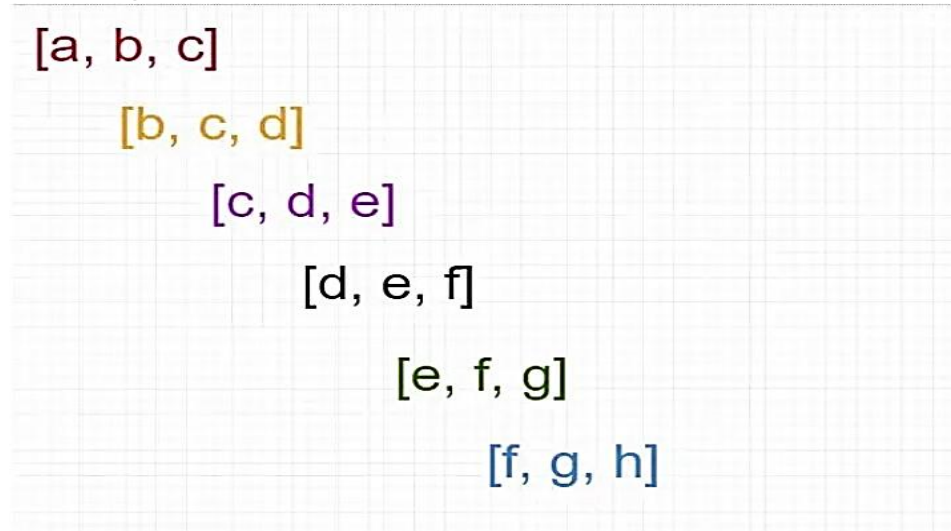
Let's say that if you have an array like below:



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

Figure 2: Array of values

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]

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]

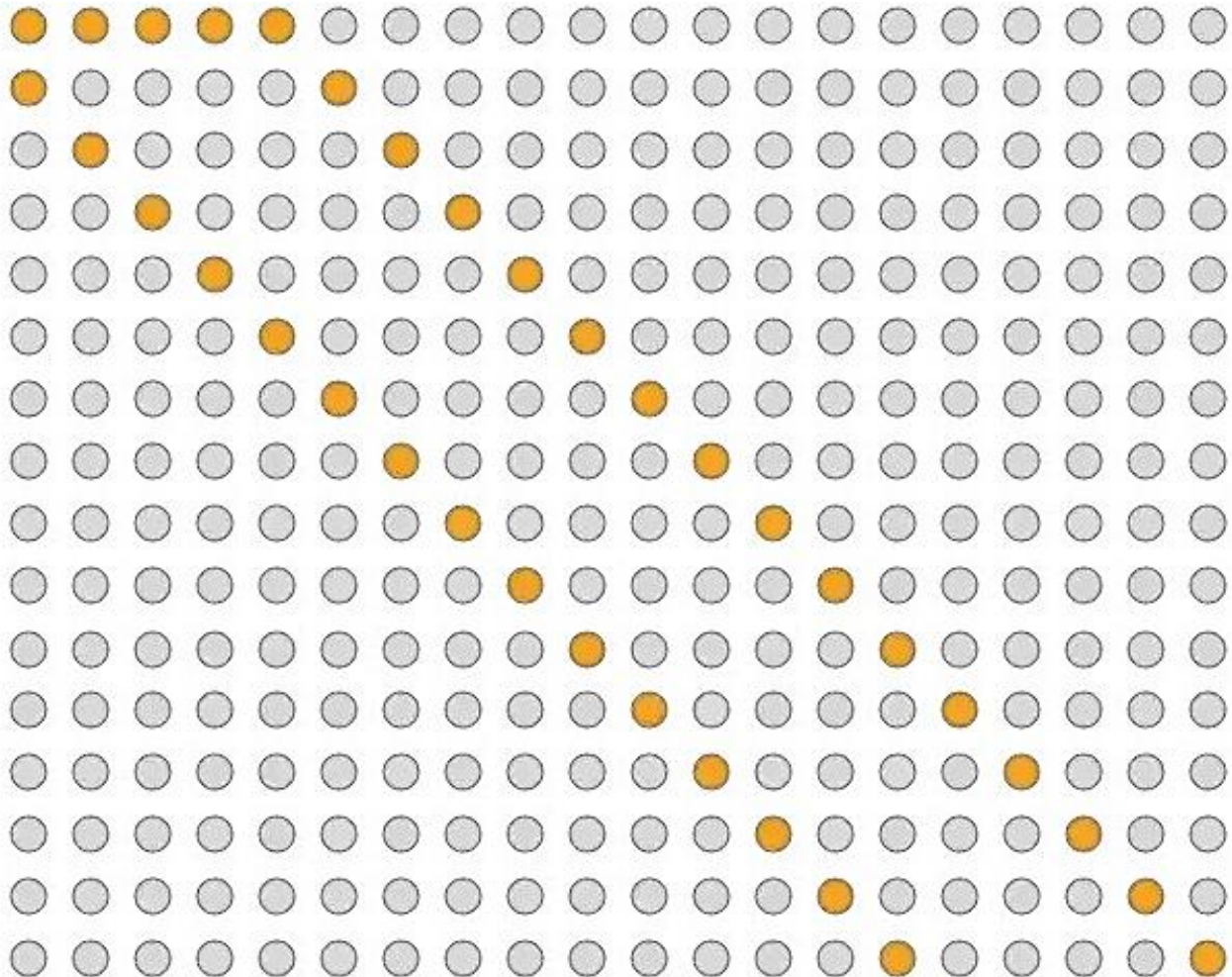


Figure 4: Sliding window technique to find the largest sum of five consecutive elements

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.

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

- 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.

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

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

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

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

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

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

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

- 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

Fixed-Size Sliding Window

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

i. 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.

ii. 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 \leftarrow 0;

for L \leftarrow 1 to n - k + 1 do

 sum \leftarrow 0;

 for i \leftarrow L to L + k - 1 do

 sum \leftarrow sum + A[i];

 end

 answer \leftarrow maximum(answer, sum);

end

return answer;

Re

- 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.

iii. 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 i 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.

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.

i. 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.**

ii. 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 \leftarrow 0;

for L \leftarrow 1 to n do

 sum \leftarrow 0;

 i \leftarrow L;

 length \leftarrow 0;

 while i \leq n AND sum + A[i] \leq k do

 sum \leftarrow sum + A[i];

 length \leftarrow length + 1;

 i \leftarrow i + 1;

 end

 answer \leftarrow 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.

iii. 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 \leftarrow 0;

sum \leftarrow 0;

R \leftarrow 1;

for L \leftarrow 1 to n do

 if L > 1 then

 sum \leftarrow sum - A[L - 1];

 end

 while R \leq n AND sum + A[R] \leq k do

 sum \leftarrow sum + A[R];

 R \leftarrow R + 1;

 end

 answer \leftarrow 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.

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 $\text{calories}[i]$ calories on the i -th day.

Given an integer k , for **every** consecutive sequence of k days ($\text{calories}[i]$, $\text{calories}[i+1]$, ..., $\text{calories}[i+k-1]$ for all $0 \leq i \leq n-k$), they look at T , the total calories consumed during that sequence of k days ($\text{calories}[i] + \text{calories}[i+1] + \dots + \text{calories}[i+k-1]$):

- If $T < \text{lower}$, they performed poorly on their diet and lose 1 point;
- If $T > \text{upper}$, 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: $\text{calories} = [1,2,3,4,5]$, $k = 1$, $\text{lower} = 3$, $\text{upper} = 3$

Output: 0

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

$\text{calories}[0]$ and $\text{calories}[1]$ are less than lower so 2 points are lost.

$\text{calories}[3]$ and $\text{calories}[4]$ are greater than upper so 2 points are gained.

Example 2:

Input: $\text{calories} = [3,2]$, $k = 2$, $\text{lower} = 0$, $\text{upper} = 1$

Output: 1

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

$\text{calories}[0] + \text{calories}[1] > \text{upper}$ so 1 point is gained.

Example 3:

Input: $\text{calories} = [6,5,0,0]$, $k = 2$, $\text{lower} = 1$, $\text{upper} = 5$

Output: 0

Explanation:

$\text{calories}[0] + \text{calories}[1] > \text{upper}$ so 1 point is gained.

$\text{lower} \leq \text{calories}[1] + \text{calories}[2] \leq \text{upper}$ so no change in points.

$\text{calories}[2] + \text{calories}[3] < \text{lower}$ so 1 point is lost.

Constraints:

- $1 \leq k \leq \text{calories.length} \leq 10^5$
- $0 \leq \text{calories}[i] \leq 20000$
- $0 \leq \text{lower} \leq \text{upper}$

Solution

- Use the idea of sliding window. Initially, calculate the calories consumed during the first consecutive k days, which is $\text{calories}[0] + \text{calories}[1] + \dots + \text{calories}[k - 1]$. Let sum be the calories consumed during the first consecutive k days.
- If $\text{sum} < \text{lower}$, then lose 1 point.
- If $\text{sum} > \text{upper}$, 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

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`

Constraints:

- $1 \leq k \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^5$

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

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.

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 2 * 10^4$
- $1 \leq \text{nums}[i] \leq 5 * 10^4$
- $1 \leq k \leq n * (n + 1) / 2$

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

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]`

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 $1 \leq \text{index1} < \text{index2} \leq \text{numbers.length}$.

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.io.*;
class MiddleOftheLinkedList
{
    Node head;
    // Linked list node
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    // Function to print the middle element of the linked list
    void printMiddle()
    {
        Node slowPtr = head;
        Node fastPtr = head;

        while (fastPtr != null && fastPtr.next != null)
        {
            fastPtr = fastPtr.next.next;
            slowPtr = slowPtr.next;
        }
        System.out.println("The middle element is [" + slowPtr.data + "] \n");
    }

    // Inserts a new Node at front of the list.
    public void push(int new_data)
    {
        // 1 & 2: Allocate the Node & Put in the data
```

```
        Node new_node = new Node(new_data);
        // 3. Make next of new Node as head
        new_node.next = head;
        // 4. Move the head to point to the new Node
        head = new_node;
    }

    // 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(" ");
        int val;
        MiddleOftheLinkedList llist = new MiddleOftheLinkedList();
        for (int i = 0; i < list.length; i++)
        {
            llist.push(Integer.parseInt(list[i]));
        }
        //llist.printList();
        llist.printMiddle();
    }
}
```

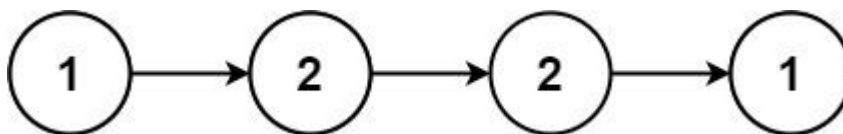
Test Cases:**case =1****input=** 5 10 20 15 25 12 30 35 32**output=**25**case =2****input=**1 2 3 4 5 6 7 8 9**output=** 5

APPLICATIONS:

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

1. Palindrome Linked List:

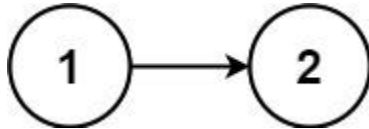
- Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

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 $ar1[0 \dots m-1]$ and $ar2[0 \dots n-1]$ 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 $l < \text{length.ar1}$ and $r \geq 0$
 - (a) If $\text{abs}(\text{ar1}[l] + \text{ar2}[r] - \text{sum}) < \text{diff}$ then
update diff and result
 - (b) If $(\text{ar1}[l] + \text{ar2}[r] < \text{sum})$ 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", "lrd", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "lo1d", "lor1", "w1r1", "lo2", "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 $n = \max(\text{len}(\text{word}), \text{len}(\text{abbr}))$

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