SLiding window

Imagine you have a long row of colorful marbles lined up, and you're holding a small, see-through ruler. This ruler is special because it's as long as exactly four marbles. Your task is to find the prettiest section of four marbles in the whole row.

You start at the beginning of the row. You place your ruler so that it covers the first four marbles. This is your first "window" of marbles. You look at them closely, remember how pretty they look together, and then you slide your ruler one marble to the right. Now, the ruler covers the second to the fifth marble. This is your second window.

You keep doing this, sliding your ruler one marble at a time, and looking at each group of four marbles. Every time you slide your ruler to a new group of marbles, you compare it with what you remember about the previous groups. You're trying to remember which group of four marbles looked the prettiest.

The sliding window technique in computer programming is quite similar to this. Instead of marbles, you have numbers in an array (which is just a list of numbers), and instead of a ruler, you have a range of numbers that you're looking at.

For example, let's say you have an array of numbers like this: [2, 4, 6, 8, 10, 12, 14], and you want to find the largest sum you can get from any three numbers next to each other. You would start with the first three numbers: [2, 4, 6]. This is your first "window". You add them up to get 12.

Then, you "slide" your window one number to the right, so now it covers [4, 6, 8]. You add these up to get 18. You remember this sum because it's larger than the first sum. You keep sliding your window and adding up the numbers, always remembering the largest sum you've found.

So, the sliding window technique is like using your ruler to look at different parts of your marble row, or different parts of your number array, one section at a time, and comparing these sections to find the prettiest or the largest sum. It's a simple way to break down a big problem (like looking at all the marbles or all the numbers) into smaller, more manageable problems (like looking at just four marbles or three numbers at a time).

**Two-Pointer**

Imagine you and your friend are on a long path filled with stones. Each stone has a number written on it. Your task is to find two stones that add up to a special number, let's say 10. You start at one end of the path, and your friend starts at the other end. You both know that the path is arranged in a way that the numbers on the stones increase as you walk towards each other.

1. **Starting Position**: You start at the beginning of the path (the first stone), and your friend starts at the end of the path (the last stone). These are your two pointers.
2. **Moving Towards Each Other**: You both start walking towards each other. Every time you meet at a pair of stones, you add up the numbers on those stones.
3. **Finding the Sum**: If the sum is too small, you move one step forward to a stone with a bigger number. If the sum is too big, your friend moves one step backward to a stone with a smaller number.
4. **Success or Meet in the Middle**: You keep doing this until you find two stones that add up to 10, or until you meet in the middle of the path.

This is the essence of the two-pointer technique. It's about having two positions in a list (or array) and moving them intelligently towards each other to find a solution to your problem.

### Applications of Two-Pointer Technique

1. **Finding a Pair with a Given Sum**: Like in the stones example, finding two numbers that add up to a specific sum in a sorted array.
2. **Reverse an Array**: One pointer starts at the beginning, and the other at the end. Swap their elements and move the pointers towards each other.
3. **Palindrome Check**: Check if a string is a palindrome by comparing characters from both ends moving towards the center.
4. **Removing Duplicates from a Sorted Array**: Use one pointer to iterate through the array and another to store the position of the next unique element.
5. **Trapping Rain Water**: Calculating the amount of water that can be trapped between buildings of different heights represented in an array.

\_\_\_\_

### Divide and Conquer

Imagine you have a huge jigsaw puzzle that seems really hard to solve all at once. To make it easier, you decide to split the puzzle into smaller sections, solve each section separately, and then put all the solved sections back together. This is what we call the "Divide and Conquer" strategy.

1. **Divide**: First, you break the puzzle into smaller, more manageable pieces. It's like taking a big problem and dividing it into smaller problems.
2. **Conquer**: Then, you solve each smaller puzzle on its own. This is easier because each piece is not as complex as the whole puzzle.
3. **Combine**: Finally, you combine the solved pieces back together to complete the entire puzzle.

**Applications of Divide and Conquer**

1. **Sorting Algorithms**: Like in Merge Sort, where you divide the array into halves, sort each half, and then merge them back together.
2. **Searching Algorithms**: In Binary Search, you divide the array in half, decide which half will contain the element, and then search only in that half.
3. **Finding Maximum and Minimum**: Divide the array into parts, find the maximum and minimum in each part, and then compare them to find the overall maximum and minimum.
4. **Calculating Power of a Number**: By dividing the power into halves, calculating for each half, and then combining the results.
5. **Multiplying Large Numbers**: In algorithms like Karatsuba's algorithm, where large numbers are split into smaller numbers, multiplied separately, and then combined.

\_\_