# Two pointers

Two-pointers is an extremely common technique used to solve array and string problems. It involves having two integer variables that both move along an iterable.

There are several ways to implement two-points. One method is:

Start the pointers at the edges of the input. Move them towards each other until they meet.

Deciding which pointers to move and when will depend on the problem we are trying to solve.

```
function fn(arr):
    left = 0
    right = arr.length - 1

    while left < right:
        Do some logic here depending on the problem
        Do some more logic here to decide on one of the following:
            1. left++
            2. right--
            3. Both left++ and right--
```

The strength of this technique is that we will never have more than O(n) iterations for the while loop because the pointers start n away from each other and move at least one step closer in every iteration. If we keep the work inside each iteration at O(1), this technique will result in a linear runtime, which is usually the best possible runtime.

---

Example 1: Given a string `s`, return `true` if it is a palindrome, `false` otherwise

```
def check_if_palindrome(s):
    left = 0
    right = len(s) - 1

    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
```

```
        right -= 1

    return True
```

Our algorithm is very efficient as not only does it run in O(n), but it also uses only O(1) space. No matter how big the input is, we will always only use two integer variables.

---

Example 2: Given a **sorted** array of unique integers and a target integer, return `true` if there exists a pair of numbers that sum to target, `false` otherwise. This problem is similar to the problem: Two Sum (but in Two Sum, the input is not sorted).

For example, given `nums = [1, 2, 4, 6, 8, 9, 14, 15]` and `target = 13`, return true because `4 + 9 = 13`.

Brute force would yield O(n^2) by trying exhaustively all pairs of numbers, but we can reduce this to O(n) using two pointers. This works because the numbers are sorted. Moving the left pointer to the right permanently increases the value the left pointer points to; moving the right pointer to the left permanently decreases the value the right pointer points to. If we have `x + y > target`, then we can never have a solution with `y` because `x` can only increase. So if a solution exists, we can only find it by decreasing `y`. Similar logic applies to `x`.

```
def check_for_target(nums, target):
    left = 0
    right = len(nums) - 1

    while left < right:
        # curr is the current sum
        curr = nums[left] + nums[right]
        if curr == target:
            return True
        if curr > target:
            right -= 1
        else:
            left += 1

    return False
```

When the problem has two iterables in the input (2 strings, 2 arrays), using two pointers means: Move along both inputs simultaneously until all elements have been checked.

```
function fn(arr1, arr2):
    i = j = 0
    while i < arr1.length AND j < arr2.length:
        Do some logic here depending on the problem
        Do some more logic here to decide on one of the following:
            1. i++
            2. j++
            3. Both i++ and j++

    // Step 4: make sure both iterables are exhausted
    // Note that only one of these loops would run
    while i < arr1.length:
        Do some logic here depending on the problem
        i++

    while j < arr2.length:
        Do some logic here depending on the problem
        j++
```

The runtime will be linear in the two array lengths, O(n + m) if the work inside the while loop is O(1).

---

Example 3: Given two sorted integer arrays `arr1` and `arr2`, return a new array that combines both of them and is also sorted.

The first thing that might come to mind is to combine both arrays and then perform a sort. If we have `n = arr1.length + arr2.length` (a different n from the n in the code which is the length of the first array), then we get a time complexity of O(nlogn) (the cost of sorting). This would be a good approach if the input arrays were not sorted, but because they are sorted we can use the two pointers technique to improve it to O(n).

```
def combine(arr1, arr2):
    i, j, k = 0, 0, 0
    n = len(arr1)
    m = len(arr2)
    result = [0] * (n + m)

    while i < n and j < m:
```

```
        if arr1[i] < arr2[j]:
            result[k] = arr1[i]
            i += 1
            k += 1
        else:
            result[k] = arr2[j]
            j += 1
            k += 1

    while i < n:
        result[k] = arr1[i]
        i += 1
        k += 1

    while j < m:
        result[k] = arr2[j]
        j += 1
        k += 1

    return result
```

---

Example 4: 392. Is Subsequence. Given two strings `s` and `t`, return `true` if `s` is a subsequence of `t`, or `false` otherwise.

A subsequence of a string is a sequence of characters that can be obtained by deleting some (or none) of the characters from the original string, while maintaining the relative order of the remaining characters. For example, "ace" is a subsequence of "abcde" while "aec" is not.

We can use two pointers to solve this in linear time. If we find that `s[i] == t[j]`, that means we "found" the letter at position `i` for `s`, and we can move on to the next one by incrementing `i`. We should increment `j` at each iteration no matter what (which means we could also implement this algorithm using a for loop). `s` is a subsequence of `t` if we can "find" all the letters of `s`, which means that `i == s.length` at the end of the algorithm.

```
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        i = j = 0
        while i < len(s) and j < len(t):
            if s[i] == t[j]:
```

```
                i += 1
            j += 1

        return i == len(s)
```

---

All the methods like starting the two pointers at the ends are just guidelines. They might not always start there. But the ideas are the same. Some problems even use three pointers!

---

SEE PROBLEMS 344, 977