2-2. Two pointers

#_area/coding_test/leetcode/2_arrays_and_strings

> The format of this article is the format that many articles in this course will take. We will introduce ideas, and then walk through example problems to demonstrate the ideas. These walkthrough problems are not meant to be solved by you.
> 이 강의의 많은 글들은 이러한 형식을 취합니다. 아이디어를 소개한 후, 예시 문제를 통해 해당 아이디어를 어떻게 적용하는지 보여줍니다. 이 예시 문제들은 여러분이 직접 풀어보라고 드리는 것이 아니라, 아이디어를 설명하기 위한 예시일 뿐입니다.

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. In this article, we are focusing on arrays and strings. This means we will have two integers, usually named something like `i` and `j`, or `left` and `right` which each represent an index of the array or string.
투 포인터(Two Pointers)는 배열이나 문자열 문제를 해결할 때 매우 자주 사용되는 기법입니다. 이는 두 개의 정수 변수를 사용하여, 둘 다 어떤 순회 가능한 객체(여기서는 배열이나 문자열)를 따라 이동하게 하는 방식입니다. 배열과 문자열에 대해서만 포커스해 보면, 보통 `i`, `j` 또는 `left`, `right` 같은 이름을 갖는 두 정수가 각각 배열(또는 문자열)의 인덱스를 가리키게 됩니다.

There are several ways to implement two pointers. To start, let's look at the following method:
투 포인터를 구현하는 데는 여러 가지 방법이 있지만, 우선 다음 방법을 살펴봅시다:

> Start the pointers at the edges of the input. Move them towards each other until they meet.
> 입력의 양 끝에서 시작하는 포인터를 서로 향해 이동시키면서 만날 때까지 진행한다.

Converting this idea into instructions:

1. Start one pointer at the first index `0` and the other pointer at the last index `input.length - 1`.
2. Use a while loop until the pointers are equal to each other.
3. At each iteration of the loop, move the pointers towards each other. This means either increment the pointer that started at the first index, decrement the pointer that started at the last index, or both. Deciding which pointers to move will depend on the problem we are trying to solve.

이 아이디어를 단계별로 정리해 보면:

1. 첫 번째 포인터는 시작 인덱스인 `0`에서, 두 번째 포인터는 마지막 인덱스인 `input.length - 1`에서 시작한다.
2. 두 포인터가 서로 같아질 때까지 `while` 루프로 반복한다.
3. 각 반복에서, 포인터들을 서로를 향해 움직인다. 즉, 시작 인덱스 쪽 포인터를 증가시키거나, 끝 인덱스 쪽 포인터

를 감소시키거나, 또는 둘 다 이동시킬 수 있다. 어떤 포인터를 어떻게 이동시킬지는 우리가 풀고자 하는 문제의 요구 사항에 따라 달라집니다.

```
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. Therefore, if we can keep the work inside each iteration at `O(1)`, this technique will result in a linear runtime, which is usually the best possible runtime. Let's look at some examples.
이 기법의 강점은, 포인터들이 서로 `n` 거리(배열 또는 문자열의 길이만큼)에서 시작해 매 반복마다 최소 한 걸음씩 가까워지므로, while 루프가 `O(n)` 이상 반복되지 않는다는 점입니다. 따라서 각 반복에서의 작업을 `O(1)`로 유지할 수 있다면, 이 기법은 전체적으로 선형 시간(Linear Time)을 달성하게 됩니다. 이는 일반적으로 달성할 수 있는 최선의 시간 복잡도입니다. 이제 몇 가지 예시를 살펴봅시다.

> **Example 1**: Given a string `s`, return `true` if it is a palindrome, `false` otherwise.
> A string is a palindrome if it reads the same forward as backward. That means, after reversing it, it is still the same string. For example: `"abcdcba"`, or `"racecar"`.
> **예시 1**: 문자열 `s`가 주어졌을 때, 만약 `s`가 팰린드롬이면 `true`를, 그렇지 않다면 `false`를 반환하라.
> 팰린드롬이란, 앞뒤가 똑같이 읽히는 문자열을 말한다. 즉, 문자열을 뒤집어도 원본과 동일해야 한다. 예를 들어 `"abcdcba"`나 `"racecar"`와 같은 경우다.

After reversing a string, the first character becomes the last character. If a string is the same after being reversed, that means the first character is the same as the last character, the second character is the same as the second last character, and so on. We can use the two pointers technique here to check that all corresponding characters are equal. To start, we check the first and last characters using

two separate pointers. To check the next pair of characters, we just need to move our pointers toward each other one position. We continue until the pointers meet each other or we find a mismatch.
문자열을 뒤집었을 때, 첫 글자가 맨 끝 글자가 됩니다. 문자열이 뒤집었을 때 동일하다는 것은, 첫 글자와 맨 끝 글자가 같고, 두 번째 글자와 끝에서 두 번째 글자가 같으며, 이 과정을 전체에 걸쳐 동일하게 유지해야 함을 의미합니다. 여기서 투 포인터 기법을 사용할 수 있습니다. 먼저, 두 개의 포인터(예: `left`, `right`)를 사용해 문자열의 첫 글자와 마지막 글자를 각각 가리키도록 설정합니다. 이후 각 반복마다 포인터들을 안쪽으로 한 칸씩 움직이면서 대응되는 문자가 모두 같은지를 확인합니다. 포인터가 서로 만나기 전까지, 또는 불일치가 발생하기 전까지 이를 반복합니다.

Detailed explanation

We keep track of two indices: a left one, and a right one. In the beginning, the left index points to the first character, and the right index points to the last character. If these characters are not equal to each other, we know the string can't be a palindrome, so we return false. Otherwise, the string may be a palindrome; we need to check the next pair. To move on to the next pair, we move the left index forward by one, and the right index backward by one. Again, we check if the pair of characters are equal, and if they aren't, we return false.
우리는 왼쪽 인덱스(`left`)와 오른쪽 인덱스(`right`)를 추적합니다. 초기에는 왼쪽 인덱스는 첫 글자(인덱스 0), 오른쪽 인덱스는 마지막 글자(인덱스 `s.length - 1`)를 가리킵니다. 이 두 글자가 다르다면, 해당 문자열은 팰린드롬이 될 수 없으므로 즉시 `false`를 반환합니다. 만약 같다면 팰린드롬일 가능성이 있으니, 다음 쌍을 검사해야 합니다. 다음 쌍을 확인하기 위해서는 왼쪽 인덱스를 1 증가시키고, 오른쪽 인덱스를 1 감소시킵니다. 다시 두 글자를 비교하고, 만약 다르다면 `false`를 반환합니다.

We continue this process until we either find a mismatch (in which case the string cannot be a palindrome, so we return false), or the pointers meet each other (which indicates we have gone through the entire string, checking all pairs). If we get through all pairs without a mismatch, we know the string is a palindrome, so we can return true.
이 과정을 통해 불일치가 발견되면(즉, 글자가 다르면) `false`를 반환하고, 포인터가 서로 만나거나 교차할 때까지 모든 쌍이 일치한다면 그 문자열은 팰린드롬이므로 `true`를 반환합니다.

To run the algorithm until the pointers meet each other, we can use a while loop. Each iteration in the while loop checks one pair. If the check is successful, we increment `left` and decrement `right` to move to the next pair. If the check is unsuccessful, we return false.
포인터가 서로 만날 때까지 알고리즘을 실행하기 위해, 우리는 `while` 루프를 사용할 수 있습니다. 각 반복에서 한 쌍의 문자를 확인하고, 만약 동일하면 `left`를 1 증가, `right`를 1 감소시켜 다음 쌍을 확인합니다. 만약 다르다면 즉시 `false`를 반환합니다.

Video

```java
public boolean checkIfPalindrome(String s) {
    int left = 0;
    int right = s.length() - 1;

    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }

        left++;
        right--;
    }

    return true;
}
```

Notice that if the input was an array of characters instead of a string, the algorithm wouldn't change. The two pointers technique works as long as the index variables are moving along some abstract iterable.
문자열 대신 문자 배열이 주어졌다고 해도, 알고리즘 자체는 변하지 않습니다. 투 포인터 기법은 인덱스 변수가 어떤 순회 가능한 객체(Iterables)를 따라 이동하기만 하면 항상 적용할 수 있기 때문입니다.

This 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 always only use two integer variables. The time complexity is `O(n)` because the while loop iterations cost `O(1)` each, and there can never be more than `O(n)` iterations of the while loop - the pointers start at a distance of `n` from each other and move closer by one step each iteration.
이 알고리즘은 매우 효율적입니다. 시간 복잡도가 `O(n)`일 뿐 아니라, 메모리를 `O(1)`만 사용합니다. 입력 크기가 얼마나 크든, 우리는 항상 두 개의 정수 변수만 사용합니다. 시간 복잡도가 `O(n)`인 이유는, `while` 루프 한 번의 반복이 `O(1)` 시간이 걸리고, 포인터가 처음에 `n`만큼 떨어진 상태에서 시작하여 매 반복마다 서로 한 칸씩 가까워져 최대 `O(n)`번 반복될 수 있기 때문입니다.

> **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 Two

Sum (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`.
Note: a similar version of this problem can be found on LeetCode: 167. Two Sum II - Input Array Is Sorted
예시 2: **정렬된** 고유 정수 배열과 타겟 정수가 주어졌을 때, 두 숫자의 합이 타겟이 되는 쌍이 존재하면 `true`, 그렇지 않다면 `false`를 반환하라.
이 문제는 Two Sum과 유사한데(차이점은 Two Sum의 입력 배열은 정렬되어 있지 않음), 예를 들어 `nums = [1, 2, 4, 6, 8, 9, 14, 15]`와 `target = 13`이 주어지면, `4 + 9 = 13`이므로 `true`를 반환해야 한다.
참고로 이 문제와 비슷한 버전은 LeetCode 167. Two Sum II - Input Array Is Sorted에서 찾을 수 있다.

The brute force solution would be to iterate over all pairs of integers. Each number in the array can be paired with another number, so this would result in a time complexity of `O(n^2)`, where `n` is the length of the array. Because the array is sorted, we can use two pointers to improve to an `O(n)` time complexity.
브루트 포스(brute force) 접근법은 모든 쌍을 확인하는 것입니다. 배열의 각 원소는 다른 모든 원소와 쌍을 이룰 수 있으므로, 이는 시간 복잡도가 `O(n^2)`이 됩니다(`n`은 배열의 길이). 하지만 배열이 정렬되어 있으므로, 투 포인터(two pointers)를 사용하여 시간 복잡도를 `O(n)`으로 줄일 수 있습니다.

Let's use the example input. With two pointers, we start by looking at the first and last numbers. Their sum is `1 + 15 = 16`. Because `16 > target`, we need to make our current sum smaller. Therefore, we should move the `right` pointer. Now, we have `1 + 14 = 15`. Again, move the right pointer because the sum is too large. Now, `1 + 9 = 10`. Since the sum is too small, we need to make it bigger, which can be done by moving the `left` pointer. `2 + 9 = 11 < target`, so move it again. Finally, `4 + 9 = 13 = target`.
예시 입력을 살펴봅시다. 투 포인터 기법으로, 우리는 배열의 첫 번째 원소(`nums[0]`)와 마지막 원소(`nums[nums.length - 1]`)부터 확인합니다. 이 둘의 합은 `1 + 15 = 16`입니다. `16`은 타겟(13)보다 크므로, 현재 합을 더 작게 만들어야 합니다. 따라서 `right` 포인터를 왼쪽으로 한 칸 움직입니다. 이제 `1 + 14 = 15`가 됩니다. 여전히 큰 값이므로 또다시 `right` 포인터를 움직입니다. 이제 `1 + 9 = 10`이 됩니다. 이번에는 합이 너무 작으므로, 합을 키우기 위해 `left` 포인터를 오른쪽으로 움직여야 합니다. `2 + 9 = 11 < target`이므로, 다시 한 번 `left`를 움직입니다. 결국, `4 + 9 = 13`이 되고, 이는 타겟과 일치합니다.

The reason this algorithm works: because the numbers are sorted, moving the left pointer permanently increases the value the left pointer points to (`nums[left] = x`). Similarly, moving the right pointer permanently decreases the value the right pointer points to (`nums[right] = y`). 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`. The same logic can be applied to `x` if `x + y < target`.
이 알고리즘이 동작하는 이유는 다음과 같습니다. 배열이 정렬되어 있으므로, `left` 포인터를 이동시키면 `nums[left]`의 값이 반드시 증가하고, `right` 포인터를 이동시키면 `nums[right]`의 값이 반드시 감소합니다. 예를 들어 `x + y > target`이라면, `x`는 더 커질 수밖에 없으므로, `y`가 위치한 `right` 포인터를 움직여 `y`를 줄이지 않으면 `x + y`가 타겟보다 작아질 기회가 없습니다. 반대로 `x + y < target`이면 `x`를 키워야 하므로 `left` 포인터를 이동시켜야 합니다.

Detailed explanation

Let's say we have `nums = [3, 6, 21, 23, 25]` and `target = 27`. We need to pick two numbers that sum to `target`. Using the two pointers technique, we start with the first and last numbers. Because the input is sorted, this is the smallest and largest number. We have `3 + 25 = 28`, which is greater than `target`.
`nums = [3, 6, 21, 23, 25]`와 `target = 27`이 있다고 해 봅시다. 두 수를 골라서 합이 `27`이 되어야 합니다. 투 포인터 기법을 적용할 때, 우리는 가장 작은 수(맨 앞)와 가장 큰 수(맨 뒤)부터 시작합니다. 즉, `3 + 25 = 28`인데, 이는 `target`(27)보다 큽니다.

Let's look at the `25`. We paired this number with the smallest number, and the sum was **still** too large. That implies that the `25` could never be part of the answer because if we chose any number other than the `3` to pair it with, the sum would be even larger. Since it can't be part of the answer, we move on to the next largest number, which is `23`.
여기서 `25`에 주목해 보겠습니다. 이 숫자를 가장 작은 수(3)와 쌍지었을 때조차 합이 여전히 너무 큽니다. 이는 `25`가 다른 수와 쌍을 이룬다면 합이 더 커질 테니, `25`는 정답의 일부가 될 수 없음을 의미합니다. 따라서 우리는 다음으로 큰 수(즉, `23`)를 고려합니다.

Now, we have `3 + 23 = 26`. This is smaller than `target`. In the previous step, we determined that the `25` could never be part of the answer. This makes the `23` the new "largest" number. Despite pairing the `3` with the largest number (that we haven't already ruled out), the sum is **still** too small. This implies that the `3` could never be part of the answer because if we chose any of the other remaining numbers (the `6` or `21`), the sum would be even smaller. Since it can't be part of the answer, we move on to the next smallest number, which is `6`.
이제 `3 + 23 = 26`이 되는데, 이는 `target`보다 작습니다. 앞선 단계에서 `25`는 정답이 될 수 없음을 확인했습니다. 이제 `23`이 "가장 큰 수"가 됩니다. 가장 작은 수(3)와 가장 큰 수(23)를 더했을 때도 합이 여전히 부족합니다. 이는 `3` 역시 정답의 일부가 될 수 없음을 의미합니다. 왜냐하면 `6`이나 `21` 같은 다른 수와 더하면 합이 더 작아질 뿐이기 때문입니다. 따라서 `3`을 제외하고 다음으로 작은 수(즉, `6`)로 이동합니다.

Now, we have `6 + 23 = 29`. Once again, our sum is too large. We apply the same logic as before - the `23` could never be part of the answer because we are already pairing it with the smallest number (that we haven't already ruled out), yet the sum is still too large. So we move to the next largest number, which is the `21`.
`6 + 23 = 29`가 되는데, 여전히 `target`을 초과합니다. 다시 같은 논리가 적용됩니다. 이미 가장 작은 수(6)와 쌍을 이루어도 너무 크다면, `23`도 정답이 될 수 없습니다. 그래서 더 작은 쪽인 `21`로 이동합니다.

Finally, we have `6 + 21 = 27`, and we have found our target.
마지막으로 `6 + 21 = 27`이 되어 `target`을 만족하는 쌍을 찾았습니다.

To implement this algorithm, we use a similar process as in the previous palindrome example. We use a while loop until the pointers meet each other. If at any point the sum is equal to the `target`, we can return true. If the pointers meet each other, it means we went through the entire input without finding `target`, so we return false.
이 알고리즘을 구현하는 방식은 앞선 팰린드롬 예시와 비슷합니다. 두 포인터가 서로 만날 때까지 `while` 루프를 돌립니다. 만약 어느 시점에서든 현재 합이 `target`과 같다면 `true`를 반환합니다. 두 포인터가 만난 뒤에도 `target`을 찾지 못했다면, 입력 전체를 확인했음에도 불가능한 경우이므로 `false`를 반환합니다.

Video

```
public boolean checkForTarget(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;

    while (left < right) {
        // curr is the current sum
        int curr = nums[left] + nums[right];
        if (curr == target) {
            return true;
        }

        if (curr > target) {
            right--;
        } else {
            left++;
        }
    }

    return false;
```

```
}
```

Like in the previous example, this algorithm uses `O(1)` space and has a time complexity of `O(n)`.
이전 예시와 마찬가지로, 이 알고리즘은 `O(1)` 공간을 사용하며 시간 복잡도는 `O(n)`입니다.

Another way to use two pointers

This method where we start the pointers at the first and last indices and move them towards each other is only one way to implement two pointers. Algorithms are beautiful because of how abstract they are – "two pointers" is just an idea, and it can be implemented in many different ways. Let's look at another method and some new examples. The following method is applicable when the problem has two iterables in the input, for example, two arrays.
이렇게 양 끝 인덱스에서 시작해 안쪽으로 이동시키는 방법은 투 포인터를 구현하는 여러 방식 중 하나일 뿐입니다. 알고리즘은 추상적이기에 “투 포인터”라는 아이디어는 다양한 방법으로 구현할 수 있습니다. 이제 다른 방식과 그 예시를 살펴봅시다. 다음에 소개할 방법은, 예를 들어 두 개의 배열처럼 **입력에 두 개의 순회 가능한 자료(Iterable)** 가 있을 때 적용할 수 있습니다.

> Move along both inputs simultaneously until all elements have been checked.
> 두 입력을 동시에 순회하면서 모든 원소를 확인할 때까지 포인터를 이동시킨다.

Converting this idea into instructions:

1. Create two pointers, one for each iterable. Each pointer should start at the first index.
2. Use a while loop until one of the pointers reaches the end of its iterable.
3. At each iteration of the loop, move the pointers forward. This means incrementing either one of the pointers or both of the pointers. Deciding which pointers to move will depend on the problem we are trying to solve.
4. Because our while loop will stop when one of the pointers reaches the end, the other pointer will not be at the end of its respective iterable when the loop finishes. Sometimes, we need to iterate through all elements – if this is the case, you will need to write extra code here to make sure both iterables are exhausted.

이 아이디어를 단계별로 정리해 보면:

1. 각각의 순회 가능한 자료에 대해 포인터를 하나씩 생성한다. 두 포인터는 모두 처음(인덱스 0)에서 시작한다.
2. 두 포인터 중 하나가 끝에 도달할 때까지 `while` 루프를 사용한다.

3. 반복문의 각 단계에서 포인터를 한 칸씩, 혹은 둘 다 이동시킨다. 어떤 포인터를 어떻게 움직일지는 문제 요구사항에 따라 달라진다.
4. `while` 루프가 종료될 때, 한 포인터는 끝에 도달했지만, 다른 포인터는 끝에 도달하지 않았을 수 있다. 문제 상황에 따라 모든 원소를 반드시 확인해야 한다면, 필요한 추가 작업을 통해 나머지 자료도 끝까지 순회하도록 처리해야 할 때가 있다.

Here's some pseudocode illustrating the concept:
다음은 이 개념을 보여주는 간단한 의사 코드 예시입니다:

```
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++
```

Similar to the first method we looked at, this method will have a linear time complexity of `O(n + m)` if the work inside the while loop is `O(1)`, where `n = arr1.length` and `m = arr2.length`. This is because at every iteration, we move at least one pointer forward, and the pointers cannot be moved forward more than `n + m` times without the arrays being exhausted. Let's look at some examples.
앞서 살펴본 첫 번째 방법과 마찬가지로, 이번 방법도 `while` 루프 내부의 작업이 `O(1)` 이라면 시간 복잡도는 `O(n + m)` 이 됩니다. 여기서 `n = arr1.length`, `m = arr2.length` 라고 할 때, 각 반복마다 최소 한 포인터가 이동하며, 배열(또는 리스트)이 소진되기 전에 한 포인터가 이동할 수 있는 횟수의 합은 최대 `n + m` 번이기 때문입니다. 이제 몇 가지 예시를 살펴봅시다.

> **Example 3**: Given two sorted integer arrays `arr1` and `arr2`, return a new array that combines both of them and is also sorted.
> **예시 3**: 정렬된 정수 배열 `arr1`과 `arr2`가 주어졌을 때, 두 배열의 원소를 모두 합쳐서 정렬된 새로운 배열을 반환하라.

The trivial approach would be to first combine both input arrays and then perform a sort. If we have `n = arr1.length + arr2.length`, then this gives a time complexity of `O(n · log n)` (the cost of sorting). This would be a good approach if the input arrays were not sorted, but because they are sorted, we can take advantage of the two pointers technique to improve to `O(n)`.
가장 단순한 방법은 두 배열을 먼저 하나로 합친 다음, 그 결과를 정렬하는 것입니다. 만약 `n = arr1.length + arr2.length`라고 정의한다면, 이는 정렬 비용 때문에 시간 복잡도가 `O(n · log n)`이 됩니다. 배열들이 정렬되어 있지 않은 경우라면 이는 합리적인 접근이겠지만, 이미 정렬되어 있으므로 투 포인터 기법을 이용하면 `O(n)` 시간 복잡도로 더 효율적인 해결이 가능합니다.

> In the previous example, we declared `n = arr1.length` and `m = arr2.length`. Here, we are saying `n = arr1.length + arr2.length`. Why have we changed the definition? Remember that when it comes to big O, **we are allowed to define the variables as we see fit**. We could certainly stick to using `n, m`. In that case, the time complexity of the sorting approach would be `O((n + m) · log (m + n))` and the time complexity of the approach we are about to cover would be `O(n + m)`. It makes no difference either way, but one justification we could give here is that since we are combining the arrays, the total length is a significant number, so it makes sense to represent it as `n`.
> Keeping the definition as `n = arr1.length` and `m = arr2.length` is fine as well.
> 이전 예시에서 우리는 `n = arr1.length`, `m = arr2.length`라고 했습니다. 여기서는 `n = arr1.length + arr2.length`라고 이야기하고 있죠. 왜 정의를 바꿨을까요? 빅 오 표기법에서는 **원하는 대로 변수를 정의할 수 있기 때문**입니다. 물론 `n`, `m`을 그대로 사용해도 되고, 그 경우 정렬 접근법의 시간 복잡도는 `O((n + m) · log(n + m))`, 새로 소개할 방법은 `O(n + m)`가 됩니다.
> 다만 여기서는 배열을 합치는 문제이므로, 전체 길이를 `n`으로 보는 것이 합리적일 수 있습니다. 배열을 합치는 상황에서 `n`이라는 값은 의미가 크니까요.
> 그렇다고 해서 `n = arr1.length`, `m = arr2.length`로 두는 것이 틀린 것은 전혀 아닙니다.

We can build the answer array `ans` one element at a time. Start two pointers at the first index of each array, and compare their elements. At each iteration, we have 2 values. Whichever value is lower needs to come first in the answer, so add it to the answer and move the respective pointer.
이제 결과 배열 `ans`를 한 번에 한 원소씩 만들어 봅시다. 두 포인터를 각각 `arr1`과 `arr2`의 첫 인덱스에서 시작하고, 두 배열의 원소를 비교합니다. 매 반복마다 두 값을 비교해 더 작은 값을 결과 배열에 넣고, 해당 포인터를 한 칸 이동시키면 됩니다.

Detailed explanation

Sorting an array of length `n` costs `O(n · log n)`. We can improve the time complexity by a factor of `log n` by taking advantage of the input arrays already being sorted.
길이가 `n`인 배열을 정렬하는 데 드는 시간 복잡도는 일반적으로 `O(n · log n)`입니다. 그런데 이미 정렬된 두 배열이 주어진 상황에서는, 이를 활용하여 `log n` 배 정도 더 빠른 시간복잡도로 문제를 풀 수 있습니다.

If we start with the smallest number from each array, then whichever one is smaller **must** be before the other one – so we add it to the answer and move to the next number in that array. If the values are equal, it doesn't matter which one we choose – we can arbitrarily choose either. This process can be repeated until one of the arrays runs out of numbers.
두 배열에서 각각 가장 작은 원소(현재 포인터가 가리키는 원소)를 비교했을 때, 더 작은 값은 어차피 상대적으로 앞에 와야 한다는 사실이 자명합니다. 따라서 더 작은 값을 결과 배열에 추가하고, 그 값을 포함했던 배열의 포인터를 한 칸 옮깁니다. 두 값이 같은 경우, 어느 쪽을 먼저 선택해도 상관없습니다. 이렇게 반복하다가 한 배열이 원소를 모두 소진하면 종료 조건에 도달합니다.

When this happens, we are still left with some numbers in the other array. These numbers are all larger than the largest number in the exhausted array. We should just append them to the answer.
한 배열이 소진된 시점에서는, 다른 배열에는 아직 원소들이 남아 있을 수 있습니다. 이 원소들은 소진된 배열에 있던 가장 큰 수보다 전부 크므로, 결과 배열 뒤에 그대로 이어붙이면 됩니다.

Video

```java
public List<Integer> combine(int[] arr1, int[] arr2) {
    // ans is the answer
    List<Integer> ans = new ArrayList<>();
    int i = 0;
    int j = 0;

    while (i < arr1.length && j < arr2.length) {
        if (arr1[i] < arr2[j]) {
            ans.add(arr1[i]);
            i++;
        } else {
            ans.add(arr2[j]);
            j++;
        }
```

```
    }

    while (i < arr1.length) {
        ans.add(arr1[i]);
        i++;
    }

    while (j < arr2.length) {
        ans.add(arr2[j]);
        j++;
    }

    return ans;
}
```

Like in the previous two examples, this algorithm has a time complexity of `O(n)` and uses `O(1)` space (if we don't count the output as extra space, which we usually don't).
앞서 본 두 예시와 마찬가지로, 이 알고리즘도 시간 복잡도는 `O(n)`이며, 사용되는 추가 공간은 `O(1)`입니다(일반적으로 결과 배열 자체는 추가 공간으로 세지 않습니다).

> **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.
> **예시 4**: 392. Is Subsequence
> 두 문자열 `s`와 `t`가 주어졌을 때, `s`가 `t`의 부분열(subsequence)이면 `true`, 아니면 `false`를 반환하라.
> 문자열의 부분열이란, 원본 문자열에서 일부(또는 전혀 제거하지 않아도 됨) 문자를 제거하여, 남은 문자들의 상대적 순서를 유지한 채 얻을 수 있는 문자의 나열을 말한다. 예를 들어, `"ace"`는 `"abcde"`의 부분열이지만, `"aec"`는 부분열이 아니다.

In this problem, we need to check if the characters of `s` appear in the same order in `t`, with gaps allowed. For example, `"ace"` is a subsequence of `"abcde"` because `"abcde"` contains the letters `"ace"` in that same order – the fact that they aren't consecutive doesn't matter.
이 문제에서, 우리는 `s`의 문자들이 `t` 안에서 같은 순서로 존재하는지(연속할 필요는 없음)를 확인해야 합니다. 예를 들어, `"abcde"`에는 `"ace"`라는 문자들이 순서대로 들어 있으므로, `"ace"`가 `"abcde"`의 부분열임을 알 수 있습

니다. 그 문자들이 연속되어 있지 않아도 순서만 유지된다면 상관없습니다.

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.
투 포인터를 사용하여 이 문제를 선형 시간에 해결할 수 있습니다. 만약 `s[i] == t[j]`가 참이라면, `s`에서 `i` 위치의 문자를 “찾았다”고 볼 수 있으므로 `i`를 1 증가시켜 다음 문자를 확인하면 됩니다. 어쨌든 매 반복마다 `j`는 증가해야 하며(이를 `for` 루프로 구현할 수도 있습니다), 모든 문자를 찾았다면, 즉 알고리즘 종료 시점에 `i == s.length`라면 `s`는 `t`의 부분열입니다.

Detailed explanation

For every character in `s`, we need to find a match in `t`. Let's say we have `s = "bc"` and `t = "abcd"`. Using the two pointers technique, we start by looking at the first character in both strings.
`s`의 각 문자에 대해, `t`에서 해당 문자를 찾아야 합니다. 예를 들어, `s = "bc"`, `t = "abcd"`라고 해 봅시다. 두 포인터 기법을 사용하여, 처음에는 두 문자열의 첫 글자부터 비교합니다.

We need to try and match the first character of `s`, which is `"b"`. The first character of `t` is `"a"`, which is not a match. As such, we will move to the next character in `t`. We don't move forward in `s` just yet, because we still need to match the `"b"`. The next character of `t` is `"b"`, and we have found a match. Now, we can move on to the next character in `s`, which is the `"c"`. A character in `t` can only be matched once, so we must also move forward in `t`. Now, we have another match since the next character in `t` is also `"c"`.
먼저 `s`의 첫 번째 문자 `"b"`를 `t`에서 찾으려 합니다. `t`의 첫 번째 문자는 `"a"`이므로 일치하지 않습니다. 따라서 아직 `"b"`를 찾지 못했으니 `s`는 그대로 두고, `t` 쪽 포인터만 이동하여 다음 문자를 확인합니다. 이제 `t`의 다음 문자는 `"b"`이고, 이는 `"b"`와 일치하므로, `s`의 다음 문자로 이동할 수 있게 됩니다. 즉, 이제 `"b"`를 찾았으니 `s`에서 다음 문자인 `"c"`를 확인합니다. 한편, 방금 `"b"`를 사용했으므로 `t`도 다음 문자로 이동해야 합니다. 이후 `t`의 현재 문자는 `"c"`인데, 이는 `s`의 `"c"`와 일치합니다.

We have managed to match all the characters in `s`, which means that `s` is a subsequence of `t`.
따라서 `s`에 있는 모든 문자를 순서대로 찾을 수 있었으므로, `"s"`는 `"t"`의 부분열임이 증명됩니다.

Video

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int i = 0;
        int j = 0;

        while (i < s.length() && j < t.length()) {
            if (s.charAt(i) == t.charAt(j)) {
                i++;
            }
            j++;
        }

        return i == s.length();
    }
}
```

Just like all the prior examples, this solution uses `O(1)` space. The time complexity is linear with the lengths of `s` and `t`.
이전 예시들과 마찬가지로, 이 솔루션은 `O(1)` 공간만을 사용합니다. 시간 복잡도는 `s`와 `t`의 길이에 비례하는 선형 시간입니다.

Closing notes

Remember that the methods laid out here are just guidelines. For example, in the first method, we started the pointers at the first and last index, but sometimes you might find a problem that involves starting the pointers at different indices. In the second method, we moved two pointers forward along two different inputs. Sometimes, there will only be one input array/string, but we still initialize both pointers at the first index and move both of them forward.
여기서 다룬 방법들은 일종의 가이드라인일 뿐이라는 점을 기억하세요. 예컨대, 첫 번째 방법에서는 포인터를 배열(또는 문자열)의 시작 인덱스와 끝 인덱스에서 시작했지만, 어떤 문제에서는 다른 인덱스에서 포인터를 시작해야 할 수도 있습니다. 두 번째 방법에서는 서로 다른 두 입력을 대상으로 포인터를 전진시켰습니다만, 경우에 따라서는 입력이 하나뿐임에도 불구하고 포인터 두 개를 모두 0번 인덱스에서 시작해서 함께 이동시키기도 합니다.

Two pointers just refers to using two integer variables to move along some iterables. The strategies we looked at in this article are the most common patterns, but always be on the lookout for a different way to approach a problem. There are even problems that make use of "three pointers".

결국 “투 포인터”란, 순회 가능한 자료(Iterable)를 따라 움직이는 두 개의 정수 변수를 사용하는 기법을 통칭하는 말입니다. 이 글에서 살펴본 전략들은 가장 일반적으로 쓰이는 패턴들이지만, 언제든 문제 해결을 위한 다른 접근법이 있을 수 있다는 점에 유의하세요. 실제로 “쓰리 포인터(Three Pointers)”를 활용하는 문제도 존재합니다.

> The chapters and articles in this course are ordered in a way that ideas learned in earlier chapters can be applied to later chapters. Two pointers certainly has a lot more applications than just what is in this article – don't worry, this won't be the last we'll be seeing of it.
> 이 강의는 앞선 장에서 배운 아이디어들을 이후 장에서 적용해볼 수 있도록 구성되어 있습니다. 여기서 살펴본 투 포인터 기법은 매우 다양한 활용 사례를 갖고 있으므로, 이 내용이 끝이 아닙니다. 이후 다른 문제들에서도 반복해서 등장할 것입니다.

Before we move on to the next pattern, try the upcoming practice problems to apply what was learnt here.
다음 패턴으로 넘어가기 전에, 본문에서 배운 내용을 직접 적용해볼 수 있도록 준비된 연습 문제를 풀어보길 권장합니다.

연습 문제

- 344. Reverse String
- 541. Reverse String II
- 977. Squares of a Sorted Array
- 88. Merged Sorted Array
- 167. Two Sum II - Input Array Is Sorted