2-5. More common patterns

#_area/coding_test/leetcode/2_arrays_and_strings

In this article, we'll briefly talk about a few more patterns and some common tricks that can be used in algorithm problems regarding arrays and strings.
이 글에서는 배열과 문자열과 관련된 알고리즘 문제에서 사용할 수 있는 몇 가지 추가 패턴과 자주 쓰이는 트릭들을 간단히 살펴보겠습니다.

O(n) string building

We mentioned earlier that in most languages, strings are immutable. This means concatenating a single character to a string is an *O(n)* operation. If you have a string that is 1 million characters long, and you want to add one more character, all 1 million characters need to be copied over to another string.
앞서 대부분의 언어에서 문자열은 불변(immutable)이라고 언급했습니다. 이는 문자열에 한 문자를 덧붙일 때마다 *O(n)*의 연산이 수행됨을 의미합니다. 예를 들어 길이가 백만 자에 달하는 문자열에 새 문자를 추가하려면, 백만 자 전부를 새 문자열로 복사해야 한다는 것입니다.

Many problems will ask you to return a string, and usually, this string will be built during the algorithm. Let's say the final string is of length `n` and we build it one character at a time with concatenation. What would the time complexity be? The operations needed at each step would be `1 + 2 + 3 + ... + n`. This is the partial sum of this series, which leads to *O(n^2)* operations.
많은 문제에서 결과를 문자열로 반환하도록 요구하며, 보통 이 문자열은 알고리즘을 수행하는 과정에서 만들어집니다. 최종 문자열의 길이가 `n`이라고 할 때, 각 문자를 하나씩 이어붙여서 문자열을 만든다면 시간 복잡도는 어떻게 될까요? 각 단계에서 필요한 연산 횟수는 `1 + 2 + 3 + ... + n`이 됩니다. 이는 해당 급수의 부분합이므로 결과적으로 *O(n^2)*의 연산을 필요로 하게 됩니다.

Simple concatenation will result in an *O(n^2)* time complexity if you are using a language where strings are immutable.
문자열이 불변인 언어에서 단순 연결을 통해 문자열을 구성하면 *O(n^2)*의 시간 복잡도가 발생합니다.

There are better ways to build strings in just *O(n)* time. This will vary between languages - here, we'll talk about Python and Java - if you're using another language, we recommend researching the best way to build strings in your language.
O(n) 시간으로 문자열을 만드는 더 효율적인 방법이 있습니다. 언어마다 방식은 조금씩 다르지만, 여기서는 Python과

Java를 예로 들겠습니다. 다른 언어를 사용 중이라면, 해당 언어에서 문자열을 효율적으로 만드는 방법을 조사해 보길 권장합니다.

Python

1. Declare a list
2. When building the string, add the characters to the list. This is *O(1)* per operation. Across `n` operations, it will cost *O(n)* in total.
3. Once finished, convert the list to a string using `"".join(list)`. This is *O(n)*.
4. In total, it cost us *O(n + n) = O(2n) = O(n)*.

—

1. 리스트를 선언합니다.
2. 문자열을 만드는 동안, 문자를 리스트에 추가합니다. 이 작업은 연산마다 *O(1)*에 해당하며, 총 `n`번 수행 시 전체 *O(n)*의 시간 복잡도를 가집니다.
3. 모든 문자를 추가한 후에는 `"".join(list)`를 사용해 리스트를 문자열로 변환합니다. 이 변환은 *O(n)*의 시간 복잡도를 가집니다.
4. 따라서 전체 과정은 *O(n + n) = O(2n) = O(n)* 시간 복잡도가 됩니다.

```
def build_string(s):
    arr = []
    for c in s:
        arr.append(c)

    return "".join(arr)
```

Java

1. Use the StringBuilder class.
2. When building the string, add the characters to the list. This is *O(1)* per operation. Across `n` operations, it will cost *O(n)* in total.
3. Once finished, convert the list to a string using `StringBuilder.toString()`. This is *O(n)*.
4. In total, it cost us *O(n + n) = O(2n) = O(n)*.

—

1. StringBuilder 클래스를 사용합니다.
2. 문자열을 만드는 과정에서, 문자를 StringBuilder에 추가합니다. 이 작업은 연산마다 *O(1)*에 해당하며, `n`번 수행 시 전체 *O(n)* 시간이 걸립니다.
3. 모든 문자를 추가한 후 `StringBuilder.toString()` 메서드로 문자열을 얻습니다. 이 과정은 *O(n)*입니다.
4. 전체적으로 *O(n + n) = O(2n) = O(n)* 시간 복잡도를 가집니다.

```
public string buildString(String s) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        sb.append(s.charAt(i));
    }

    return sb.toString();
}
```

> In C++ and JavaScript, simply using `+=` is fine when building strings.
> C++와 JavaScript에서는 문자열을 만들 때 `+=` 연산만 사용해도 무방합니다.

Subarrays/substrings, subsequences, and subsets

Let's quickly talk about the differences between these types and what to look out for when encountering them in problems.
이들의 차이점과 문제에서 이를 마주했을 때 주의해야 할 점들을 간단히 살펴보겠습니다.

Subarrays/substrings

> As a reminder, a subarray or substring is a contiguous section of an array or string.
> 서브어레이(subarray) 또는 서브스트링(substring)은 배열이나 문자열에서 연속된 부분을 의미합니다.

If a problem has explicit constraints such as:

- Sum greater than or less than `k`
- Limits on what is contained, such as the maximum of `k` unique elements or no duplicates allowed

만약 문제에 다음과 같은 명시적 제한이 있다면:

- 합이 `k`보다 크거나 작아야 한다
- `k`개의 고유 원소 이내여야 한다든지, 혹은 중복이 허용되지 않는다는 등의 제한

And/or asks for:

- Minimum or maximum length
- Number of subarrays/substrings
- Max or minimum sum

또는 다음과 같은 요청사항이 있다면::

- 최소 혹은 최대 길이
- 가능한 서브어레이/서브스트링의 개수
- 최대 혹은 최소 합

Think about a sliding window. Note that not all problems with these characteristics should be solved with a sliding window, and not all sliding window problems have these characteristics. These characteristics should only be used as a general guideline.
이런 경우 슬라이딩 윈도우(sliding window) 기법을 고려해볼 만합니다. 물론 이러한 특징을 가진 문제들이 모두 슬라이딩 윈도우로 해결되는 것은 아니며, 슬라이딩 윈도우 문제가 모두 이러한 특징을 갖는 것도 아닙니다. 이는 단지 일반적인 지침으로 활용하시기 바랍니다.

If a problem's input is an integer array and you find yourself needing to calculate multiple subarray sums, consider building a prefix sum.
만약 문제의 입력이 정수 배열이고 여러 서브어레이의 합을 자주 구해야 한다면, prefix sum(누적 합)을 고려해볼 수 있습니다.

The size of a subarray between `i` and `j` (inclusive) is `j - i + 1`. This is also the number of subarrays that end at `j`, starting from `i` or later.
`i`부터 `j`(포함)까지의 서브어레이 크기는 `j - i + 1`입니다. 이는 또한 `j`에서 끝나는 서브어레이의 수이기도 합니다. (시작점은 `i` 이상이 됩니다.)

Subsequences

> A subsequence is a set of elements of an array/string that keeps the same relative order but doesn't need to be contiguous.
> 서브시퀀스(subsequence)는 배열이나 문자열에서 원소들의 상대적 순서를 유지하면서, 연속적일 필요는 없는 부분 집합입니다.

For example, subsequences of `[1, 2, 3, 4]` include: `[1, 3]`, `[4]`, `[]`, `[2, 3]`, but not `[3, 2]`, `[5]`, `[4, 1]`.
예를 들어 `[1, 2, 3, 4]`의 서브시퀀스로는 `[1, 3]`, `[4]`, `[]`, `[2, 3]` 등이 있지만, `[3, 2]`, `[5]`, `[4,`

`1]` 등은 서브시퀀스가 아닙니다.

Typically, subsequence problems are more difficult. Because this is only the first chapter, it is difficult to talk about subsequence patterns now. Subsequences will come up again later in the course - for example, dynamic programming is used to solve a lot of subsequence problems.
보통 서브시퀀스 문제는 더 어렵습니다. 아직은 기초 단계를 다루고 있기 때문에, 서브시퀀스 패턴에 대해 자세히 논의하기는 이릅니다. 서브시퀀스는 이후 챕터에서 다시 다룰 것이며, 예를 들어 동적 프로그래밍(DP)을 통해 해결하는 서브시퀀스 문제가 많이 있습니다.

From the patterns we have learned so far, the most common one associated with subsequences is two pointers when two input arrays/strings are given (we did look at one problem in the two pointers articles involving subsequences). Because prefix sums and sliding windows represent subarrays/substrings, they are not applicable here.
지금까지 배운 패턴 중에서 서브시퀀스와 관련이 가장 깊은 것은, 두 개의 입력 배열(또는 문자열)이 주어졌을 때 사용하는 투 포인터(two pointers) 기법입니다. (이전에 투 포인터 챕터에서 서브시퀀스가 등장하는 예시를 살펴본 바 있습니다.) prefix sum과 슬라이딩 윈도우는 서브어레이/서브스트링에 대한 기법이므로, 서브시퀀스 문제에는 적용할 수 없습니다.

Subsets

> A subset is any set of elements from the original array or string. The order doesn't matter and neither do the elements being beside each other.
> 서브셋(subset)이란 원본 배열이나 문자열에서 임의의 원소들을 선택하여 만든 집합을 말합니다. 이때 순서는 중요하지 않으며, 원소들이 연속되어 있을 필요도 없습니다.

For example, given `[1, 2, 3, 4]`, all of these are subsets: `[3, 2]`, `[4, 1, 2]`, `[1]`. Note: subsets that contain the same elements are considered the same, so `[1, 2, 4]` is the same subset as `[4, 1, 2]`.
예를 들어 `[1, 2, 3, 4]`가 주어졌을 때, `[3, 2]`, `[4, 1, 2]`, `[1]` 등은 모두 서브셋입니다. 단, 동일한 원소들을 포함하고 있다면 순서가 달라도 같은 서브셋으로 간주하므로, `[1, 2, 4]`와 `[4, 1, 2]`는 동일한 서브셋입니다.

You may be thinking, what is the difference between subsequences and subsets if subsets with the same elements are considered the same? In subsequences, the order matters - let's say you had an array of integers and you needed to find a subsequence with 3 consecutive elements (like `1, 2, 3`).

This would be harder than finding a *subset* with 3 consecutive elements because, with a subset, the 3 elements simply need to exist. In a subsequence, the elements need to exist in the correct relative order.
서브셋에서 동일 원소를 담고 있으면 같은 경우로 친다면, 서브시퀀스와 서브셋의 차이는 무엇일까요? 서브시퀀스에서는 원소의 상대적 순서가 매우 중요합니다. 예를 들어, 어떤 정수 배열에서 `1, 2, 3`처럼 연속된 3개의 요소가 포함된 서브시퀀스를 찾아야 한다면, 이는 단순히 3개 원소가 존재하기만 하면 되는 서브셋을 찾는 것보다 훨씬 까다로운 문제가 됩니다. 서브시퀀스에서는 원소가 올바른 상대적 순서대로 배열에 존재해야 합니다.

Again, since we are only in the first chapter, it is hard to talk much about subsets. We will see subsets being used in the backtracking chapter.
여전히 첫 번째 챕터 단계이므로, 서브셋에 대해서도 깊이 다루기는 어렵습니다. 서브셋은 이후 백트래킹(backtracking) 챕터에서 다시 등장할 것입니다.

One thing to note is that if a problem involves subsequences, but the order of the subsequence doesn't actually matter (let's say it wants the sum of subsequences), then you can treat it the same as a subset. A useful thing that you can do when dealing with subsets that you can't do with subsequences is that you can sort the input, since the order doesn't matter.
한 가지 주의할 점은, 어떤 문제에서 서브시퀀스를 다루더라도 그 순서가 실제로는 중요하지 않은 경우(예를 들어, 서브시퀀스의 합만 요구하는 경우)에는 서브셋과 동일하게 다룰 수 있다는 것입니다. 서브시퀀스 문제와 달리, 서브셋 문제에서는 입력의 순서가 중요하지 않으므로 정렬을 수행해도 된다는 점이 유용합니다.

Video

Closing notes

That's all for the arrays and strings chapter. Because of the simplicity of the topic, it is difficult to delve into deeper problems at the moment. However, the structure of this course involves building on knowledge incrementally. For example, there are dozens of sliding window problems on LeetCode that we couldn't talk about here because we haven't talked about hash maps yet. Because almost all non tree/graph/linked list problems have an array or string in the input, this will definitely not be the last we see of the patterns learned in the chapter.
이것으로 배열과 문자열 챕터를 모두 마쳤습니다. 이 주제 자체가 비교적 간단하기 때문에, 현재로서는 더 심화된 문제를 자세히 다루기 어려운 점이 있습니다. 하지만 이 강의의 전체 구조는 단계별로 지식을 쌓아가는 형태입니다. 예를 들어 해시맵에 대해 아직 배우지 않았으므로, 여기서 다루지 못한 슬라이딩 윈도우 문제만 해도 LeetCode 상에 수십 개

가 있습니다. 트리, 그래프, 연결 리스트가 아닌 문제 대부분에서 배열과 문자열이 입력으로 주어지므로, 이번 챕터에서 배운 기법들은 앞으로도 계속 등장할 것입니다.