2-4. Prefix sum

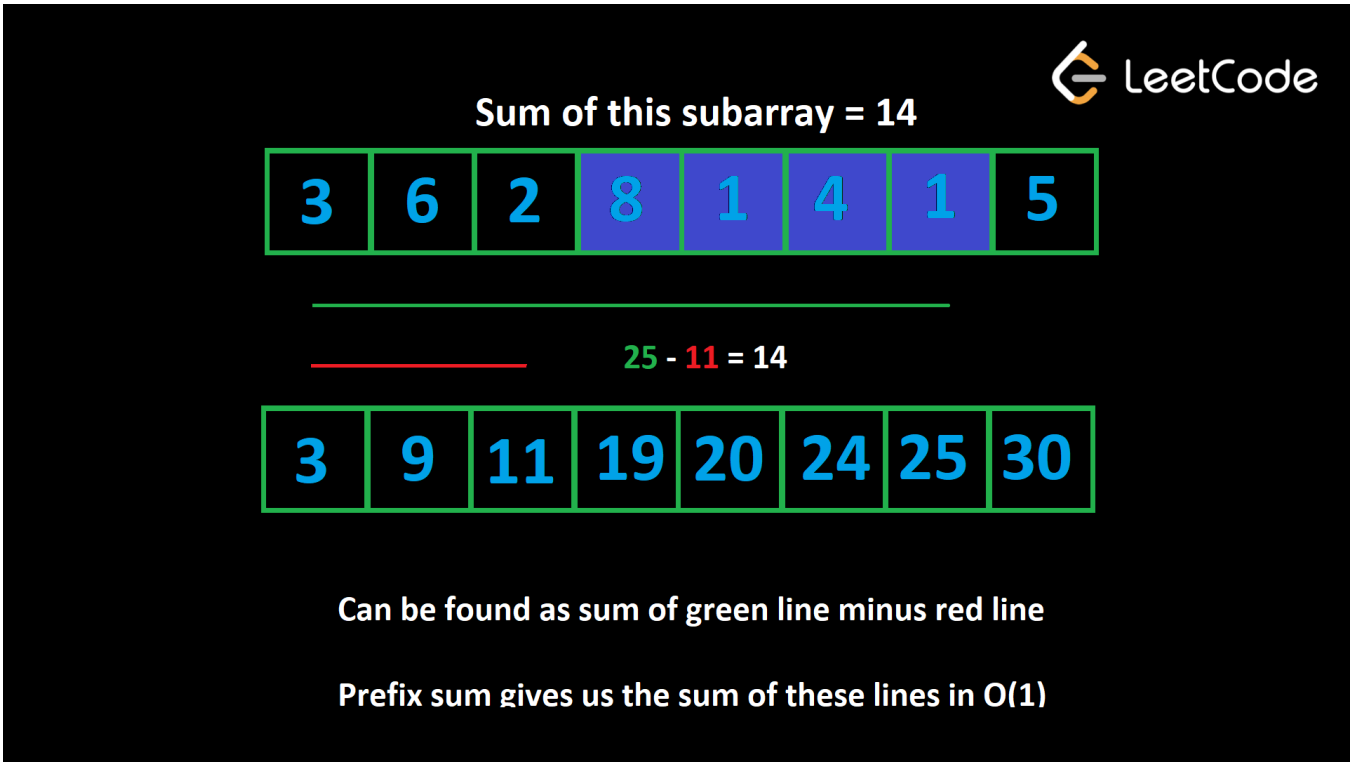
#_area/coding_test/leetcode/2_arrays_and_strings

Prefix sum is a technique that can be used on arrays (of numbers). The idea is to create an array `prefix` where `prefix[i]` is the sum of all elements up to the index `i` (inclusive). For example, given `nums = [5, 2, 1, 6, 3, 8]`, we would have `prefix = [5, 7, 8, 14, 17, 25]`.
프리픽스 합(Prefix Sum)은 숫자로 이루어진 배열에 사용할 수 있는 기법입니다. `prefix[i]`가 인덱스 `i`까지(포함) 모든 원소의 합을 나타내는 `prefix` 배열을 만드는 방식입니다. 예를 들어, `nums = [5, 2, 1, 6, 3, 8]`이라면, `prefix = [5, 7, 8, 14, 17, 25]`가 됩니다.

> When a subarray starts at index `0`, it is considered a "prefix" of the array. A prefix sum represents the sum of all prefixes.
> 부분 배열(subarray)이 인덱스 `0`에서 시작할 때, 이를 배열의 "프리픽스(prefix)"라고 합니다. 프리픽스 합은 이러한 모든 프리픽스의 합을 나타냅니다.

Prefix sums allow us to find the sum of any subarray in O(1). If we want the sum of the subarray from `i` to `j` (inclusive), then the answer is `prefix[j] - prefix[i - 1]`, or `prefix[j] - prefix[i] + nums[i]` if you don't want to deal with the out of bounds case when `i = 0`.
프리픽스 합을 사용하면, 임의의 부분 배열 합을 O(1)에 구할 수 있습니다. 인덱스 `i`부터 `j`(포함)까지의 부분 배열 합을 구하고 싶다면, `prefix[j] - prefix[i - 1]`을 사용하면 됩니다. 단, `i = 0`일 때의 인덱스 범위 문제를 피하려면 `prefix[j] - prefix[i] + nums[i]` 방식을 사용할 수도 있습니다.

This works because `prefix[i - 1]` is the sum of all elements before index `i`. When you subtract this from the sum of all elements up to index `j`, you are left with the sum of all elements starting at index `i` and ending at index `j`, which is exactly what we are looking for.
이 방식이 동작하는 이유는, `prefix[i - 1]`가 인덱스 `i` 이전까지의 모든 원소의 합을 나타내기 때문입니다. 인덱스 `j`까지의 합에서 이를 빼면, 인덱스 `i`부터 `j`까지의 합만 남게 됩니다. 이는 우리가 구하고자 하는 부분 배열의 합과 정확히 일치합니다.



> In the above image, we want to find the sum of the subarray highlighted in blue.
> If you take all the elements up until the end of the subarray (the green line) and subtract all the elements before it (the red line), you have the subarray.
> With a prefix sum, we can find the sum of the green line `25` and red line `11` in constant time and take their difference to find the sum of the subarray as `14`.
> 위 그림에서, 파란색으로 강조된 부분 배열의 합을 구하고자 합니다.
> 부분 배열이 끝나는 지점(녹색 선)까지의 모든 원소 합에서 그 전의 원소들(빨간색 선)의 합을 빼면, 해당 부분 배열이 구해집니다.
> 프리픽스 합을 사용하면, 녹색 선에 해당하는 `25`와 빨간색 선에 해당하는 `11`을 상수 시간에 구해 그 차이를 구할 수 있고, 이로써 부분 배열 합 `14`를 빠르게 얻을 수 있습니다.

Building a prefix sum is very simple. Here's some pseudocode:
프리픽스 합을 만드는 과정은 매우 간단합니다. 다음은 이를 보여주는 의사 코드입니다:

```
Given an array nums,

prefix = [nums[0]]
for (int i = 1; i < nums.length; i++)
    prefix.append(nums[i] + prefix[prefix.length - 1])
```

Initially, we start with just the first element. Then we iterate with `i` starting from index `1`. At any given point, the last element of `prefix` will represent the sum of all the elements in the input up to but not including index `i`. So we can add that value plus the current value to the end of `prefix` and continue to the next element.
처음에는 배열의 첫 원소만 사용해서 시작합니다. 그리고 `i`를 인덱스 `1`부터 시작하며 순회합니다. 이 때, `prefix`의 마지막 원소는 인덱스 `i` 이전까지의 모든 원소 합을 나타냅니다. 따라서 이 값에 현재 원소 값을 더한 뒤 `prefix`에 추가하고, 다음 인덱스로 넘어가면 됩니다.

A prefix sum is a great tool whenever a problem involves sums of a subarray. It only costs O(n) to build but allows all future subarray queries to be O(1), so it can usually improve an algorithm's time complexity by a factor of O(n), where n is the length of the array. Let's look at some examples.
프리픽스 합은 부분 배열의 합과 관련된 문제에서 매우 유용한 도구입니다. O(n) 시간에 미리 만들어두면, 이후 모든 부분 배열 합을 O(1)에 구할 수 있으므로, 일반적으로 알고리즘의 시간 복잡도를 O(n) 정도 줄이는 데 큰 도움이 됩니다 (여기서 n은 배열의 길이입니다). 예시를 살펴봅시다.

> Building a prefix sum is a form of **pre-processing**. Pre-processing is a useful strategy in a variety of problems where we store pre-computed data in a data structure before running the main logic of our algorithm. While it takes some time to pre-process, it's an investment that will save us a huge amount of time during the main parts of the algorithm.
> 프리픽스 합을 만드는 과정은 **전처리(Pre-processing)**의 한 형태입니다. 전처리는 알고리즘의 주요 로직을 실행하기 전에, 필요한 정보를 미리 계산해 데이터 구조에 저장해 두는 기법을 말합니다. 전처리 단계에 어느 정도 시간이 들긴 하지만, 이를 통해 알고리즘의 본 로직을 수행할 때 큰 시간을 절약할 수 있습니다.

> **Example 1**: Given an integer array `nums`, an array `queries` where `queries[i] = [x, y]` and an integer `limit`, return a boolean array that represents the answer to each query. A query is `true` if the sum of the subarray from `x` to `y` is less than `limit`, or `false` otherwise.
> For example, given `nums = [1, 6, 3, 2, 7, 2]`, `queries = [[0, 3], [2, 5], [2, 4]]`, and `limit = 13`, the answer is `[true, false, true]`. For each query, the subarray sums are `[12, 14, 12]`.
> **예제 1**: 정수 배열 `nums`와, `queries[i] = [x, y]` 형태를 갖는 쿼리 배열 `queries`, 그리고 정수 `limit`가 주어집니다. 각 쿼리에 대해 `x`부터 `y`까지의 부분 배열 합이 `limit`보다 작으면 `true`, 그렇지 않다면 `false`를 결과로 하는 불리언 배열을 반환해야 합니다.
> 예를 들어, `nums = [1, 6, 3, 2, 7, 2]`, `queries = [[0, 3], [2, 5], [2, 4]]`, `limit = 13`이 주어졌다면, 각 쿼리에 대한 부분 배열의 합은 `[12, 14, 12]`이고, 결과는 `[true, false, true]`가 됩니다.

Let's build a prefix sum and then use the method described above to answer each query in O(1).
프리픽스 합을 만들어둔 뒤, 위에서 설명한 방식을 사용하면 각 쿼리를 O(1) 시간에 처리할 수 있습니다.

```java
public boolean[] answerQueries(int[] nums, int[][] queries, int limit) {
    int[] prefix = new int[nums.length];
    prefix[0] = nums[0];

    for (int i = 1; i < nums.length; i++) {
        prefix[i] = prefix[i - 1] + nums[i];
    }

    boolean[] ans = new boolean[queries.length];
    for (int i = 0; i < queries.length; i++) {
        int x = queries[i][0], y = queries[i][1];
        int curr = prefix[y] - prefix[x] + nums[x];
        ans[i] = curr < limit;
    }

    return ans;
}
```

Without the prefix sum, answering each query would be O(n) in the worst case, where n is the length of `nums`. If `m = queries.length`, that would give a time complexity of O(n * m). With the prefix sum, it costs O(n) to build, but then answering each query is O(1). This gives a much better time complexity of O(n + m). We use O(n) space to build the prefix sum.
프리픽스 합 없이 각 쿼리를 처리하려면, 배열 `nums`의 길이를 n이라 할 때 최악의 경우 O(n)이 걸립니다. 쿼리 개수가 m이라면 전체 시간 복잡도는 O(n * m)이 됩니다. 반면, 프리픽스 합을 미리 구해두면 O(n) 시간에 준비를 마칠 수 있고, 이후 각 쿼리를 O(1)에 처리할 수 있으므로 최종 시간 복잡도는 O(n + m)이 됩니다. 또한, 프리픽스 합 배열을 저장하기 위해 O(n) 공간을 사용합니다.

> **Example 2**: 2270. Number of Ways to Split Array
> Given an integer array `nums`, find the number of ways to split the array into two parts so that the first section has a sum greater than or equal to the sum of the second section. The second section should have at least one number.

> **예제 2**: 2270. Number of Ways to Split Array
> 정수 배열 `nums`가 주어졌을 때, 배열을 두 부분으로 나누는 모든 방법의 수를 구하는 문제입니다. 이 때, 첫 번째 부분의 합이 두 번째 부분의 합보다 크거나 같아야 하고, 두 번째 부분에는 적어도 하나 이상의 원소가 포함되어야 합니다.

A brute force approach would be to iterate over each index `i` from `0` until `nums.length - 1`. For each index, iterate from `0` to `i` to find the sum of the left section, and then iterate from `i + 1` until the end of the array to find the sum of the right section. This algorithm would have a time complexity of O(n^2).
가장 단순한 방법으로는 인덱스 `0`부터 `nums.length - 1`까지를 순회하면서, 각 인덱스에 대해 왼쪽 부분의 합과 오른쪽 부분의 합을 각각 구하는 것입니다. 즉, 왼쪽 부분을 구하기 위해 `0`부터 `i`까지 순회하고, 오른쪽 부분을 구하기 위해 `i + 1`부터 배열 끝까지 순회합니다. 이 접근 방식의 시간 복잡도는 O(n^2)이 됩니다.

If we build a prefix sum first, then iterate over each index, we can calculate the sums of the left and right sections in O(1), which would improve the time complexity to O(n).
먼저 프리픽스 합을 만든 뒤 각 인덱스를 순회하면, 왼쪽 부분과 오른쪽 부분의 합을 O(1) 시간에 구할 수 있어, 시간 복잡도를 O(n)으로 개선할 수 있습니다.

Detailed explanation

When we split the array into two parts, we are left with two adjacent subarrays. We need to find the sums of these subarrays and compare them.
배열을 두 부분으로 나누면, 결국 두 인접한 부분 배열이 만들어집니다. 우리는 이 부분 배열들의 합을 구해 서로 비교해야 합니다.

There are n – 1 ways to split the array (the right section can't be empty). For each of these splits, it would cost O(n) to iterate over the two subarrays and find their sums.
배열을 나눌 수 있는 경우는 (오른쪽 부분이 비어 있을 수 없으므로) n – 1가지입니다. 각 경우마다 두 부분 배열의 합을 직접 구하려면 O(n) 시간이 소요됩니다.

Instead, we can spend O(n) once to build a prefix sum before trying any splits. Then we can use the prefix sum to perform each of the n – 1 splits in O(1) time. As we know, with a prefix sum we can calculate the sum of any subarray in O(1).
대신, 분할을 시도하기 전에 O(n) 시간을 들여 프리픽스 합을 한 번 계산해 두면, 이후 n – 1개의 분할을 각각 O(1) 시간에 처리할 수 있습니다. 이미 알고 있듯, 프리픽스 합을 사용하면 임의의 부분 배열 합을 O(1)에 구할 수 있기 때문입니다.

Let's say we are splitting at index `i`. The left section has all elements in the array up to index `i`, so it has a sum of `prefix[i]`. The right section begins at index `i + 1` and ends at the final index `n - 1`. This means it has a sum of `prefix[n - 1] - prefix[i]`.
예를 들어, 인덱스 `i`에서 배열을 나눈다고 합시다. 왼쪽 부분에는 인덱스 `i`까지의 모든 원소가 포함되므로 그 합은 `prefix[i]`가 됩니다. 오른쪽 부분은 인덱스 `i + 1`부터 마지막 인덱스인 `n - 1`까지 포함하므로, 합은 `prefix[n - 1] - prefix[i]`가 됩니다.

Video

```
class Solution {
    public int waysToSplitArray(int[] nums) {
        int n = nums.length;

        long[] prefix = new long[n];
        prefix[0] = nums[0];

        for (int i = 1; i < n; i++) {
            prefix[i] = nums[i] + prefix[i - 1];
        }

        int ans = 0;
        for (int i = 0; i < n - 1; i++) {
            long leftSection = prefix[i];
            long rightSection = prefix[n - 1] - prefix[i];
            if (leftSection >= rightSection) {
                ans++;
            }
        }

        return ans;
    }
}
```

Do we need the array?

In this problem, the order in which we need to access `prefix` is incremental: to find `leftSection`, we do `prefix[i]` as `i` increments by `1` each iteration.
이 문제에서는 `prefix` 배열에 접근하는 순서가 점진적(incremental)입니다. 예를 들어, `leftSection` 값을 구하기 위해서는 매 반복마다 `i`가 1씩 증가할 때 `prefix[i]`를 사용하게 됩니다.

As such, to calculate `leftSection` we don't actually need the array. We can just initialize `leftSection = 0` and then calculate it on the fly by adding the current element to it at each iteration.
따라서 `leftSection`을 구하기 위해 실제로는 배열이 필요하지 않습니다. `leftSection = 0`으로 초기화한 뒤, 매 반복마다 현재 원소를 더해 나가는 방식으로 즉석에서 값을 계산할 수 있습니다.

What about `rightSection`? By definition, the right section contains all the numbers in the array that aren't in the left section. Therefore, we can pre-compute the sum of the entire input as `total`, then calculate `rightSection` as `total - leftSection`.
그렇다면 `rightSection`은 어떻게 해야 할까요? 정의에 따르면 오른쪽 부분에는 왼쪽 부분에 포함되지 않은 배열의 모든 원소가 포함됩니다. 따라서 전체 배열 원소들의 합을 미리 `total`로 계산해 두면, `rightSection`은 `total - leftSection`으로 구할 수 있습니다.

We are still using the concept of a prefix sum as each value of `leftSection` represents the sum of a prefix. We have simply replicated the functionality using an integer instead of an array.
이 방식 역시 엄밀히 보면 프리픽스 합을 사용하는 것과 같습니다. `leftSection`이 매 순간 프리픽스(배열의 앞부분) 합을 나타내기 때문입니다. 단지, 배열 대신 하나의 정수 변수를 사용해 같은 기능을 구현했을 뿐입니다.

```
class Solution {
    public int waysToSplitArray(int[] nums) {
        int ans = 0;
        long leftSection = 0;
        long total = 0;

        for (int num: nums) {
            total += num;
        }

        for (int i = 0; i < nums.length - 1; i++) {
            leftSection += nums[i];
            long rightSection = total - leftSection;
            if (leftSection >= rightSection) {
                ans++;
            }
        }

        return ans;
    }
```

```
}
```

We have improved the space complexity to O(1), which is a great improvement.
이 방식으로 공간 복잡도를 O(1)로 개선할 수 있으며, 이는 큰 진전이라 할 수 있습니다.

Closing notes

This is the last major pattern we will be looking at for arrays and strings. In the next article, we'll look at a few more common tricks and patterns, then close the chapter with a quiz before moving on. Before that, try applying the concepts learned here in the next problem.
이로써 배열과 문자열을 다루는 핵심적인 기법들을 모두 살펴보았습니다. 다음 글에서는 몇 가지 추가적인 자주 쓰이는 팁과 패턴을 살펴본 뒤, 퀴즈로 마무리한 다음 다음 단계로 넘어갈 예정입니다. 그 전에, 여기서 배운 내용을 다음 문제에 직접 적용해 보시기 바랍니다.

연습 문제

- 1480. Running Sum of 1d Array
- 1413. Minimum Value to Get Positive Step by Step Sum
- 2090. K Radius Subarray Averages