Interview Question: Subarray Sums

This question is a hybrid of a coding problem and a math problem. If you're interested in giving someone a more mathematically interesting problem, give this one a try!

Problem Statement:

A ***subarray*** of an array is a consecutive sequence of zero or more values taken out of that array. For example, the array [1, 3, 7] has seven subarrays:

[ ] [1] [3] [7] [1, 3] [3, 7] [1, 3, 7]

Notice that [1, 7] is not a subarray of [1, 3, 7], because even though the values 1 and 7 appear in the array, they're not consecutive in the array. Similarly, the array [7, 3] isn't a subarray of the original array, because these values are in the wrong order.

The ***sum*** of an array is the sum of all the values in that array. Your task is to write a function that takes as input an array and outputs the sum of all of its subarrays. For example, given [1, 3, 7], you'd output 36, because

[ ] + [1] + [3] + [7] + [1, 3] + [3, 7] + [1, 3, 7] =

0 + 1 + 3 + 7 + 4 + 10 + 11 = 36

Possible Follow-Up Questions:

- Is the array necessarily sorted? **No, it's not necessarily sorted**.
- Can the array contain duplicate values? **Yes, it might**.
- Do we have to worry about integer overflow? **No, don't worry about that**.
- Are there any bounds on the number of elements in the array? **No, there isn't**.
- How do we handle the empty array? **The sum of zero numbers is zero.**

What to Watch For

- Watch out for off-by-one errors in the loop bounds – it's really easy to miss a subarray or walk off the end of the array with bad indices.
- Make sure they don't accidentally double-count subarrays or account for subarrays that don't exist.

Possible Solution Routes

Option 1: Brute force

One option would be to just list off all the subarrays and add all of them up. One way to do this is with a triple for loop: you can iterate over all pairs of (start, stop), then iterate over all the elements in the subarray defined by that range. This runs in time $\Theta(n^3)$, but only requires O(1) space. This is considered a "poor" solution because there are several observations you can use to speed this up considerably.

```
private int subarraySum(int[] arr) {
      int result = 0;
      for (int i = 0; i < arr.length; i++) {
            for (int j = i; j < arr.length; i++) {
                  for (int k = i; k <= j; k++) {
                        result += arr[k];
                  }
            }
      }
      return result;
}
```

Option 2: Optimized Subarray Enumeration

One observation you can use to speed up brute force is the following: if you know the sum of the subarray from index i to index j, then the sum of the subarray from index i to index j+1 can be formed by taking the sum of the original subarray, then adding arr[j+1] into the total. We can use this to improve the speed of the brute-force algorithm: rather than rescanning every subarray we see, we scan the subarrays in a way that lets us quickly determine the sum of each subarray from the newly-added value and the previous value. This drops the runtime down to $\Theta(n^2)$, which is much faster than before.

Sample solution:

```
private int subarraySum(int[] arr) {
      int result = 0;
      for (int i = 0; i < arr.length; i++) {
            int sum = 0;
            for (int j = i; j < arr.length; i++) {
                  sum += arr[j];
                  result += sum;
            }
      }
      return result;
}
```

Option 3: Regrouping the Sum

There's a third route you can take that runs in time O(n) but requires a bit of math. The basic idea behind the approach is to compute the sum, but not in the order intended. For example, take a look at the array [1, 2, 3, 4]. The subarrays are

[1] [2] [3] [4]

[1, 2] [2, 3] [3, 4]

[1, 2, 3] [2, 3, 4]

[1, 2, 3, 4]

Notice how many copies of each element there are. There are four 1's, six 2's, six 3's, and four 4's. If we could efficiently compute how many copies of each element there are across all the different subarrays, we could directly compute the sum by multiply each element in the array by the number of times it appears across all the subarrays and then adding them up.

You can find a bunch of interesting patterns with how many times each number shows up. Here's one useful one. We can count the number of subarrays that overlap a particular element at index i by counting those subarrays and focusing on the index at which those subarrays start. The first element of the array will appear in n different subarrays – each of them starts at the first position. The second element of the array will appear in n-1 subarrays that begin at its position, plus n-1 subarrays from the previous position (there are n total intervals, of which one has length one and therefore won't reach the second element). The third element of the array will appear in n-2 subarrays that begin in its position, plus n-2 subarrays beginning at the first element (all n arrays, minus the one of length two and the one of length one) and n-2 subarrays beginning at the second element (all n-1 of them except for the one of length one). More generally, the ith element will open $n - i$ new intervals (one for each length stretching out to the end) and, for each preceding element, will overlap $n - i$ of the intervals starting there. This means that the total number of intervals overlapping element i is given by

$$(n - i)i + (n - i) = (n - i)(i + 1)$$

Consequently, we can solve this problem by multiplying each element in the array by this scaling factor and taking the sum. This is shown here:

```
private int subarraySum(int[] arr) {
      int result = 0;
      for (int i = 0; i < arr.length; i++) {
            result += arr[i] * (i + 1) * (arr.length - i);
      }
      return result;
}
```

This runs in O(n) time and uses only O(1) space.