Complexity Practice

What is the complexity of the following function?

```
int sumOfCubes(int n) {
    int partialSum = 0;
    for (int i = 1; i <= n; ++i) {
        partialSum += i * i * i;
    }
    return partialSum;
}
```

Even though we're calculating a sum of cubes, this runs in linear time. Why? We

```
        for (int j = i; j < a.size(); ++j) {
            thisSum += a[j];
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

This is an improved version of algorithm 1, which eliminates some of the redundant calculation and eliminates one loop. With two nested loops, this will run in order $O(n^2)$ time.

```
/**
 * Algorithm 3 "Divide and conquer"
 */
int maxSumRec(const std::vector<int> &a, int left, int right) {
    // base case for recursion, when we have one element
    if (left == right) {
        if (a[left] > 0) {
            return a[left];
        } else {
            return 0;
        }
    }
    // divide into two halves
    int center = (left + right) / 2;
    // get the
```

Though this seems more complex, this is substantially more efficient than either of the two previous algorithm. It uses a recursive "divide and conquer" approach, and accordingly runs in $O(n \log(n))$ time. This algorithm is due to Shamos.

Remember what we learned about recursive functions at the beginning of the course, and notice that this function has a base case, where `left == right` — In the base case, we return the value of the one-element subsequence if it is greater than zero, or zero otherwise. This algorithm also has two recursive calls, so it is multiply recursive.

We call this "divide and conquer" because it divides the problem into two smaller instances, solves each, and then combines the result. This is where the recursive cases come in. The working portion of the sequence is divided into halves, and then each half becomes input to a recursive call.

The remaining code addresses the case where a subsequence may straddle the boundary between left and right halves. So the function takes the maximum of three results, left, right, and "middle" — so to speak.

This may seem complicated, but it is far more efficient than either of the two previous algorithms.

Because we divide recursively the problem in halves, then solve each half and combine the results, this runs in $O(n \log(n))$ time.

```
/**
 * Algorithm 4
 */
int kadanesAlgo(const std::vector<int> &a) {
    int bestSum = 0;
    int currentSum = 0;
    for (int i = 0; i < a.size(); ++i) {
        currentSum = std::max(0, currentSum + a-¡)Clrt+ (a
```

So we see our algorithms vary from cubic $O(n^3)$ to linear $O(n)$ time.

Does more code mean greater complexity? No. Shamos' div

ر

```
    vector<vector<int>> product;
    product.resize(nums1.size());
    for (int i = 0; i < nums1.size(); ++i) {
        product[i].resize(nums2.size());
        for (int j = 0; j < nums2.size(); ++j) {
                product[i][j] = nums1[i] * nums2[j];
        }
    }
    return product;
}
```

Let's think about what's going on here. We're taking two integer vectors as inputs, and we're calculating a 2D matrix of the element-wise products of the two vectors. So if we have *M* elements in `nums1` and *N* elements in `nums2` our result — product — will be an $M \times N$ matrix. Each element in the answer matrix will be the product of corresponding entries in the two inputs.

Here's an example: If `nums1 = {1, 2, 3, 4}` and `nums2 = {5, 0, 2}`, then `product` will look like this:

5	0	2
10	0	4
15	0	6
20	0	8

So time complexity will depend on the size of the two input vectors, that is $O(M \times N)$.

What is the auxiliary complexity? Well, we'll need to allocate an $M \times N$ matrix to hold the result, and integers `i` and `j` to control our loops, and something to hold the results of the size calculations. But apart from the $M \times N$ matrix, these other items are constant and do not vary with input. Accordingly, $M \times N$ dominates, and auxiliary complexity is $O(M \times N)$.

Space complexity also includes the input vectors, one of size *M*, and the other of size *N*, but these scale in a linear fashion and again, $M \times N$ dominates. Hence, the space complexity is $O(M \times N)$.

This concludes our discussion for now, but rest assured, we'll be spending plenty of time on complexity throughout the course.

An annotated transcript and source code to accompany this video have been posted on

Blackboard.