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? Well, simple arithmetic calculations run in constant time, and we have a single loop, to perform a simple calculation for all values from one to n. So this runs in linear time. If n is 10, this loop will perform 10 iterations.

What about performing a calculation on a 2-D array? Recall the pixel-averaging problem we discussed earlier? Let's look at a somewhat simpler problem, just taking the sum of elements in a 2-D array — that is, a matrix.

```
int twoDArraySum(std::vector<std::vector<int>> a) {
    int result = 0;
    for (int i = 0; i < a.size(); i++) {
        std::vector<int> row = a[i];
        for (int j = 0; j < a[i].size(); j++) {
            result = result + row[j];
        }
    }
    return result;
}
```

What's the complexity here? $O(n^2)$ because we have nested loops. Let's assume we have a square matrix. The outer loop executes n times — once for each row in the matrix. But for each time the outer loop executes, the inner loop executes n times, that is, once for each column within each row. That's $n \times n$ steps. Notice that it's the fact that the loops are *nested* that makes the complexity $O(n^2)$.

Same result, different complexity

Maximum Subsequence Sum Problem

Suppose you have a sequence of integers (in a vector) and you want to know the maximum sum of any contiguous subsequence. For example, the maximum subsequence sum of

`-2, 11, -4, 13, -5, -2`

is 20 (the 11, -4, 13 subsequence).

Here we present four different algorithms for solving this problem. Let's analyze the complexity of each.

```
/**
 * Algorithm 1 "Brute force"
 */
int maxSubSum1(const std::vector<int> &a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {  // left boundary
        for (int j = i; j < a.size(); ++j) {  // right boundary
            int thisSum = 0;
            for (int k = i; k <= j; ++k) {
                thisSum += a[k];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Algorithm 1 is a brute force approach, and with three loops the worst case is of order $O(n^3)$. For each i we iterate through all possible $j >= i$ and then all $i \leq k \leq j$. So subsequences are evaluated for all k between the left boundary, i, and the right boundary, j. While this is guaranteed to find the largest subsequence sum, it is very inefficient and duplicates many calculations. Because there are three nested loops, this runs in order $O(n^3)$ time.

```
/**
 * Algorithm 2
 */
int maxSubSum2(const std::vector<int> &a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {
        int thisSum = 0;
```

```
            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 max subsequence sum from each of the two halves
        int maxLeftSum = maxSumRec(a, left, center);  // recursive call
        int maxRightSum = maxSumRec(a, center + 1, right);  // recursive call

        // starting from the center and working toward left boundary
        int maxLeftBorderSum = 0, leftBorderSum = 0;
        for (int i = center; i >= left; --i) {
            leftBorderSum += a[i];
            if (leftBorderSum > maxLeftBorderSum) {
                maxLeftBorderSum = leftBorderSum;
            }
        }
        // starting from the center and working toward right boundary
        int maxRightBorderSum = 0, rightBorderSum = 0;
        for (int j = center + 1; j <= right; ++j) {
            rightBorderSum += a[j];
            if (rightBorderSum > maxRightBorderSum) {
                maxRightBorderSum = rightBorderSum;
            }
        }
        int combined = maxLeftBorderSum + maxRightBorderSum;

        // return the max of the three subsequence sums
        return std::max(maxLeftSum, std::max(maxRightSum, combined));
    }
```

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[i]);
        bestSum = std::max(bestSum, currentSum);
    }
    return bestSum;
}
```

Kadane's algorithm wins the prize. It is a very elegant solution that runs in linear $O(n)$ time. This is the fastest possible. Discussion of why this algorithm works is a little beyond the scope of this class, but feel free to puzzle over it. Notice that there is only *one* loop through the array, and thus its complexity is linear.

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

Does more code mean greater complexity? No. Shamos' divide and conquer algorithm is the longest -- in terms of code -- but is far, far more efficient than the brute force or improved brute force approaches which have fewer lines of code. So we cannot equate complexity with the number of lines of code.

Time Complexity

Now, in our discussions so far, we've been discussing time complexity.

- If we double the input, how much longer will the program take to run?
- If we double the input, how many more statements do we need to execute?

But there are other ways we can measure complexity.

Auxiliary complexity

One alternative measure of complexity is called "auxiliary complexity". Auxiliary complexity measures the amount of temporary memory needed by the algorithm. We may ask: If we double the input, how much more *temporary memory* does the algorithm allocate? Be aware that auxiliary complexity does not include the inputs to the algorithm.

Space Complexity

Another alternative measure is space complexity. With regard to space complexity, we ask:

- If we double the input, how much more memory does the algorithm use?
- If we double the input, how much more space does the algorithm allocate? Space complexity *does* include the input variables.

Let's look at some examples and analyze them in terms of time, space, and auxiliary complexity. What is the time complexity of the following code?Auxiliary complexity? Space complexity?

```
int sum(int a, int b, int c) {
    int sum = a + b + c;
    return sum;
}
```

The time complexity here is constant: *O*(1). What's the auxiliary complexity? Well, that excludes input, so what other memory must we allocate? We allocate space for an integer, `sum`, to hold the result. That's it. So auxiliary complexity is also constant, *O*(1). What's the space complexity? No matter what the inputs, this algorithm will always require space for four integers, so again, space complexity is constant, *O*(1).

What is the time complexity of the following code?Auxiliary complexity? Space complexity?

```
int sum(std::vector<int> nums) {
    int sum = 0;
    for (int i = 0; i < nums.size(); ++i){
        sum += nums[i];
    }
    return sum;
}
```

Here we take a vector of integers as an input and calculate the sum. The time complexity is linear, since we are making simple arithmetic calculations on each iteration of a one-dimensional vector.

How much auxiliary space is needed? Well, the algorithm must allocate space for the `sum` variable, and for the loop index, `i`. That's constant, regardless of the input. We could also include the space needed to hold the size of the vector as well, and whatever space is needed to determine the size of the vector, but again that's constant, and does not change the result.

How much space is needed overall? That's just the auxiliary space needed plus the size of the input. Since the size of the input dominates, the complexity is linear, *O*(*n*).

Complexities with Multiple Variables

Sometimes there are two independent variables at play that will affect the complexity of the algorithm. For example, having two vectors of different sizes.
In this case, you will need two variables to describe the complexity, e.g. $O(N + K)$, $O(N \times K)$, $O(NK)$, *etc*. If it is not clear what the variables represent, you will need to state it explicitly.

What is the time complexity of the following code?Auxiliary complexity? Space complexity?

```
vector<vector<int>> mult(vector<int> nums1, vector<int> nums2) {
```

```
    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.