CSC 20 Big-O worksheet

Recall the steps we learned in class for determining the Big-O representation for algorithm runtime efficiency.

1. Pick an operation that contributes to the dominant term of the formula representing the number of operations executed in the worst case.
2. Determine exactly how many times that operation takes place for a size n input.
3. Throw away low terms and coefficients, and put what's left inside O( ).

The resulting expression "scales" the same way that the runtime of the algorithm does (on large problem sizes). For example, O(n^2) means that if you double the problem size, you can expect runtime to roughly quadruple.

A `for` loop

```
for (int i=0; i<n; i++) {
    ...
}
```

is basically shorthand for

```
int i=0;
while (i<n) {
    ...
    i++;
}
```

So, when considering the operations of a `for` loop, consider the initialization, test, and increment statements separately.

Sample problems

The following methods each take an array with n elements as input. For each, put a box around any statement that contributes to the lead term of the work formula. Exactly how many times does each boxed statement get executed? What is the big-oh running time of each method?

1)

```
public static int foo1(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

2)

```
public static int foo2(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        for (int j=0; j<arr.length; j++) {
            sum += arr[i];
        }
    }
```

```
    return sum;
}
```

3)

```
public static int foo4(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        for (int j=0; j<10; j++) {
            sum += arr[i];
        }
    }
    return sum;
}
```

4)

```
public static int foo3(int[] arr) {
    int sum = 0;
    for (int i=0; i<arr.length; i++) {
        for (int j=i; j<arr.length; j++) {
            sum += arr[i];
        }
    }
    return sum;
}
```

5)

```
// pre: arr.length is a power of 2
public static int foo5(int[] arr) {
    int sum = 0;
    for (int i=1; i<=arr.length; i*=2) {
        sum += arr[i];
    }
    return sum;
}
```

Answers

In each of these, I'm going to focus on the most executed statement, but the way this worksheet is written, other statements contributing to the dominant term would be acceptable too.

1) `i<arr.length` gets executed the most, n+1 times (true n times, false 1 time). O(n).

2) The inner for-loop gets started n times, and each time the inner for-loop runs, its test `j<arr.length` gets executed n+1 times. So, in total, `j<arr.length` executes $n^2 + n$ times. So, O(n^2).

3) The inner for-loop gets started n times, and each time the inner for-loop runs, its test `j<10` gets executed 11 times. So, in total, `j<10` executes 11n times. So, O(n).

4) The inner for-loop gets started n times, but each time it gets started it loops one fewer time than the time before. The first time, the inner for-loop loops n times (and so the `j<arr.length` test occurs n+1 times). The second time the inner for-loop starts, it loops n-1 times (and so the `j<arr.length` test occurs n times). The last time the inner for-loop gets started, i=arr.length-1, and so the inner-loop loops just once (and so the `j<arr.length` test occurs 2 times). Summing the number of times the test `j<arr.length` occurs we get: (n+1) + (n) + (n-1) + ... + 2 = n(n+3)/2 = $0.5n^2 + 1.5n$ which is O(n^2).

5) Starting with 1, how many doublings does it take to get to n? Since we are told n is a power of 2, the answer is the base-2 logarithm of n. So, the test `i<=arr.length` occurs exactly log(n) + 1 times, which is O(log n).