Homework 1

Tyler Angert

Jan 22 2017

THIS CODE IS MY OWN WORK. IT WAS WRITTEN WITHOUT CONSULTING CODE WRITTEN BY OTHER STUDENTS OR MATERIALS OTHER THAN THIS SEMESTER'S COURSE MATERIALS. TYLER ANGERT.

1 ComplexCode 1

```java
    public void f(int N) {
    for (int i = 0; i < N; i++) {
        System.out.println("Hey");

        if (i == 5) {
            i = N;
        }
    }
}
```

1.1 Justification of O(1)

1. This code features a loop that starts at 0, and increments to N by 1 at each iteration.

2. When the index reaches the numerical value of 5, i = N. Since the loop ends when i < N and i is ALWAYS assigned the value of N when it reaches 5, then the loop always executes exactly 5 times.

3. Therefore it is O(1).

2 ComplexCode 4

```
  public void f(int N) {
for (int i = 1; i < N; i *= 2) {
  System.out.println("Hey");

  for (int j = 0; j < N; j += 2) {
    System.out.println("You");
  }
}
}
```

2.1 Justification of O(NlogN)

1. The inner loop has complexity of O(N/2) which is O(N). This is because on each iteration of the loop, the index increments by 2, so it covers the entire length of the loop in N/2 operations as opposed to N if it incremented by 1.

2. The outer loop is O(logN) since on every iteration of the loop, the index multiplies by a factor of 2. Therefore it covers double the distance of the total loop on each iteration. For example, if the input N = 32 and the index starts at 1, then the progression goes: 2, 4, 8, 16, 32, which is 5 steps. Lo and behold, log base 2 of 32 = 5.

3. Since the inner loop, which is O(N), occurs on each iteration of the outer loop, the total complexity is O(NlogN).

3 ComplexCode 9

```
public int f(int[] a, int N) {
if (N <= 0) {
  return a[0];
} else {
  return a[N-1] + f(a, N-1) + f(a, N-1);
}
}
```

3.1 Justification of $O(2^n)$

1. Each call of the function returns 2 calls to the same function, but with one subtracted iteration. Therefore any single call creates its own subtree of recursive function calls. Since each call to the function produces another 2 instances, each layer of the algorithm thus has a progressively increasing amount of function calls.

2. At step 1 there is the root call which produces 2 recursive calls. At each of those 2 recursive calls, there are another 2 recursive calls so now we have 2 * 2 = 4 recursive calls. At step 3, each child call has another 2 as well, so we have 2 * 2 * 2 calls. It keeps going down, adding another 2 calls at each node until the function returns for good.

3. Therefore the amount of function calls at any layer N can be represented by 2^n. By the time the function has completed for good, there are exactly 2^n operations on the bottom layer of the recursion tree.

4 ComplexCode 14

```
    public void f(int[] a, int N) {
    HashMap<Integer,Integer> x = new HashMap<Integer,Integer>();

    for (int i = 0; i < N; i++) {
       x.put(a[i], 2 * a[i]);
       x.put(2 * a[i], x.get(a[i]));
       System.out.println(x.get(2 * a[i]));
    }
}
```

4.1 Justification of O(N)

1. Insertion and retrival in hashmaps are (on average) O(1). However, this operation itself occurs N times in the loop, making it O(N) * O(1) = O(N).

5 ComplexCode 20

```
    public int f(int N) {
    if (N <= 0) {
```

```
        return 1;
    } else {
        return 2 * f(N / 2);
    }
}
```

5.1 Justification of O(logN)

1. The base case of this recursive code returns 1 when N <= 0.
2. Each other call of the function returns 2 times the function with HALF of the previous operations.
3. Therefore for every function call, a function is returned that only has to complete half as many operations. Therefore by the time we get to the base case, log(N) operations will have been completed.