CS 61B Spring 2022 — Recursive Algorithmic Analysis

Exam Prep Discussion 8: March 7, 2022

1 Asymptotics is Fun!

(a) Using the function g defined below, what is the runtime of the following function calls? Write each answer in terms of N.

```
1  void g(int N, int x) {
2      if (N == 0) {
3          return;
4      }
5      for (int i = 1; i <= x; i++) {
6          g(N - 1, i);
7      }
8  }
```

g(N, 1): Θ( )

g(N, 2): Θ( )

Solution:
g(N, 1): $\Theta(N)$
Explanation: When x is 1, the loop gets executed once and makes a single recursive call to g(N - 1). The recursion goes g(N), g(N - 1), g(N - 2), and so on. This is a total of N recursive calls, each doing constant work.

g(N, 2): $\Theta(N^2)$
Explanation: When x is 2, the loop gets executed twice. This means a call to g(N) makes 2 recursive calls to g(N - 1, 1) and g(N - 1, 2).
From the first part, we know g(..., 1) does linear work. Thus, this is a recursion tree with N levels, and the total work is $(N-1)+(N-2)+...+1 = \Theta(N^2)$ work.

(b) Suppose we change line 6 to g(N - 1, x) and change the stopping condition in the for loop to i <= f(x) where f returns a random number between 1 and x, inclusive. For the following function calls, find the tightest Ω and big O bounds.

```
1  void g(int N, int x) {
2      if (N == 0) {
3          return;
4      }
5      for (int i = 1; i <= f(x); i++) {
6          g(N - 1, x);
7      }
8  }
```

`g(N, 2)`: Ω( ), O( )
`g(N, N)`: Ω( ), O( )

Solution:
`g(N, 2)`: $\Omega(N)$, $O(2^N)$
`g(N, N)`: $\Omega(N)$, $O(N^N)$
Explanation: Suppose `f(x)` always returns 1. Then, this is the same as case 1 from (a), resulting in a linear runtime.
On the other hand, suppose `f(x)` always returns x. Then `g(N, x)` makes `x` recursive calls to `g(N - 1, x)`, each of which makes `x` recursive calls to `g(N - 2, x)`, and so on, so the recursion tree has 1, x, x^2 ... nodes per level. Outside of the recursion, the function `g` does `x` work per node. Thus, the overall work is $x * 1 + x * x + x * x^2 + ... + x * x^{N-1} = x(1 + x + x^2 + ... + x^{N-1})$.
Plug in `x = 2` to get $2(1 + 2 + 2^2 + ... + 2^{N-1}) = O(2^N)$ for our first upper bound. Plug in `x = N` to get $N(1 + N + N^2 + ... + N^{N-1}) = O(N^N)$ (ignoring lower-order terms).

2 Slightly Harder (Spring 2017, MT2)

Give the runtime of the following functions in Θ or O notation as requested. Your answer should be as simple as possible with no unnecessary leading constatns or lower order terms. For f5, your bound should be as tight as possible (so don't just put $O(N^{NM!})$ or similar for the second answer).

```
1 public static void f4(int N) {
2     if (N == 0) {return;}
3     f4(N / 2);
4     f4(N / 2);
5     f4(N / 2);
6     f4(N / 2);
7     g(N); // runs in Θ(N^2) time
8 }
```

Runtime: $\Theta($ $)$

Solution: Runtime: $\Theta(N^2 \log N)$

Explanation: We will try a sample input, N = 4.

```
                                        f4(4)

        f4(2)                f4(2)               f4(2)               f4(2)

   f4(1)   f4(1)    f4(1)   f4(1)     f4(1)   f4(1)    f4(1)   f4(1)

f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0)
```

For the first layer, the time needed is dominated by `g(N)`, which runs in $\Theta(N^2)$ time. Therefore, the time taken at this level is 4^2.

At the second level, each call is 2^2. But there are four calls to `f4(2)`, so the total time is $(4)(2)^2 = 4^2$.

In general, at the i-th level, the total time is $(4^i)(N/2^i)^2$, which is equal to exactly N^2.

Therefore:

```
                                f4(N)                                   N^2

      f4(N/2)            f4(N/2)           f4(N/2)           f4(N/2)        (4*N^2)/4

   f4(1)   f4(1)    f4(1)   f4(1)     f4(1)   f4(1)    f4(1)   f4(1)     (8*N^2)/8

f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) f4(0)f4(0) (16*N^2)/16
```

Each layer takes total time N^2, and the number of layers is $log_2 N$ (one layer when $N = 2$, three layers when $N = 8$, etc.). The total time is $\sum_{i=0}^{\log N} N^2 = \Theta(N^2 \log N)$.

```
1 public static void f5(int N, int M) {
2     if (N < 10) {return;}
```

```
    for (int i = 0; i <= N % 10; i++) {
        f5(N / 10, M / 10);
        System.out.println(M);
    }
}
```

Runtime: $O($ $)$

Solution:
Runtime: $O(N)$

Explanation:
Again, we can think of this as a tree. Each call to `f5` does `N\%10` work. We can consider this to be contant, since even as `N` gets massive, `N\%10` will always be between 0 and 10. So let's assume the worst case, which means we assume `N\%10 = 9` all the time. This means we make 9 calls to `f5` each time. Now, how many levels does our tree have before it ends? We are dividing `N` by 10 each call and we end when we hit `N < 10`. So, there are `logN` levels to our tree. We can now sum up the work done at each level:

$$1 + 9 + 9^2 + ... + 9^{\log N} = \sum_{i=0}^{N} 9^i = \frac{1 - 9^{\log N}}{1 - 9}$$

Remember that asymptotics don't care about constant coefficients! So we can get rid of all those to get:

$$9^{\log N} \approx O(N)$$

3 Flip Flop

Suppose we have the `flip` function as defined below. Assume the method `unknown` returns a random `integer` between 1 and `N`, exclusive, and runs in constant time. For each definition of the `flop` method below, give the best and worst case runtime of `flip` in $\Theta(.)$ notation as a function of `N`.

```
1  public static void flip(int N) {
2      if (N <= 100) {
3          return;
4      }
5      int stop = unknown(N);
6      for (int i = 1; i < N; i++) {
7          if (i == stop) {
8              flop(i, N);
9              return;
10         }
11     }
12 }
```

(a)
```
public static void flop(int i, int N) {
    flip(N - i);
}
```

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$
Explanation: Consider some arbitrary value of `stop`. When `stop = x`, we do `x` work inside of `flip` (the `for` loop) and recursively call `flip(N - x)` through `flop`. This results in a total of `N / x` calls before reaching our base case, and `x` work per call, for a total of $\Theta(N)$ work. Note that this holds for any value of `x`, so our best and worst case are the same.

(b)
```
public static void flop(int i, int N) {
    int minimum = Math.min(i, N - i);
    flip(minimum);
    flip(minimum);
}
```

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(1)$, Worst Case: $\Theta(N \log(N))$
Explanation: In the best case, `stop = 1`. This hits the base case immediately, so we make 2 calls to `flip` then stop for $\Theta(1)$ work.
In the worst case, `stop = N / 2`. This results in `flip` making 2 recursive calls to itself with the argument `N / 2`. Note the similarity of this recurrence and

mergesort; the runtime is the same $\Theta(N \log N)$.

(c)
```
public static void flop(int i, int N) {
    flip(i);
    flip(N - i);
}
```

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$
Explanation: In the best case, suppose `stop = 1`. Then `flip(N)` makes recursive calls to `flip(1)` and `flip(N - 1)`, the first of which terminates immediately in the base case. `flip(N - 1)` then calls `flip(1)` and `flip(N - 2)`. The pattern is a linear recursion: constant work per call, N calls total for $\Theta(N)$ work.
In the worst case, suppose `stop = N - 1`. Note that this case is symmetrical to the best case in terms of recursive calls; however we do work proportional to `N` inside of `flip` each time because of the `for` loop. The overall work is $(N-1) + (N-2) + (N-3) + ... + 2 + 1 = \Theta(N^2)$.