CS 61B Spring 2020 — Exam Prep 8: More Asymptotics

1 Flip Flop

For each problem, give the best and worst-case runtimes in $\Theta(.)$ notation as a function of `n`. Your answer should be simple with no unnecessary leading constants or summations.

```
public static void flip(int n) {
    if (n <= 100) {
        return;
    }
    for (int i = 1; i < n; i++) {
        // Assume g(i,n) will be equal to i for at least one i
        if (g(i, n) == i) {
            flop(i, n);
            return;
        }
    }
}
```

Given the method `flip` defined above, we will determine the best and worst case runtimes when `flop` is defined as:

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

Best Case: ___$\Theta(n)$___ Worst Case: ___$\Theta(n)$___

Simply executing the for loop to exhaustion and calling flop at the latest possible (when i equals n-1) takes up n time. The other extreme, exiting the for loop right away, calls flop(1, n) which calls flip(n - 1). Quanitfying this, we see flip(n) = 1 + flip(n - 1), so flip(n) takes N time (if we keep exiting the for loop right away after calling flip). Thus, flip(n) will always take n time!

(b)
```
public static void flop(int a, int b) {
    int low = Math.min(a, b - a);
    flip(low);
    flip(low);
}
```

Best Case: ___$\Theta(1)$___ Worst Case: ___$\Theta(nlogn)$___

We observe the best case behavior when g(i, n) always returns 1. This calls flop(1, n), which in turn calls flip(1) twice, both of which terminate immediately. We observe the worst case behavior when g(i, n) always returns n/2. This sets the minimum between i and n-i to n/2, producing two calls to flip(n / 2). Repeating this process gives us a runtime of nlogn (draw a recursive tree of height log(n) with n work performed per level).

(c)
```
public static void flop(int a, int b) {
    flip(a);
    flip(b - a);
}
```

Best Case: ___$\Theta(n)$___ Worst Case: ___$\Theta(n^2)$___

This one is very tricky! We will consider three cases, when g(i, n) returns 1, n*p (for some fraction p), or n-1. The reason that we only need to consider these three cases is because every other case produces the same runtime as one of these three!

When g(i, n) returns 1, or some constant much smaller than n, we see that we will have a spindly recursive tree where each of the n levels does constant work, producing a runtime of n. When g(i, n) returns some p*n, let's assume p is 1/2 since all fractions produce the same asymptotic runtime, we see that it is the same as part b: nlogn. Finally, when g(i, n) always returns n-1, we will again have a spindly recursive tree of n levels but the i^{th} level now does i work, giving us a runtime of n^2.

2 Some More Analysis

For each of the pieces of code below, give the **worst case** runtime in $\Theta(.)$ notation as a function of N. Your answer should be as simple as possible (i.e. avoid unnecessary constants, lower order terms, etc.). If the worst case is an infinite loop, write an infinity symbol in the blank. Assume there is no limit to the size of an int (otherwise technically they're all constant). (Sping 2015)

(a)
```
public static void f1(int N) {
    int sum = 0;
    for (int i = N; i > 0; i -= 1) {
        for (int j = 0; j < i; j += 1) {
            sum += 1;
        }
    }
}
```

Runtime: ___$\Theta(N^2)$___

The inner loop runs i number of times for each value of i, and each of those runs takes $\Theta(1)$ time. Because we decrement the value of i by 1 each time, i will take on every value from 1 to N inclusive. Thus, we end up with:

$$\Theta(1+2+...+N) = \Theta\left(\frac{N(N+1)}{2}\right) = \Theta(N^2)$$

(b)
```
public static void f2(int N) {
    int sum = 0;
    for (int i = 1; i <= N; i = i*2) {
        for (int j = 0; j < i; j += 1) {
            sum += 1;
        }
    }
}
```

Runtime: ___$\Theta(N)$___

The inner loop runs i number of times for each value of i, and each of those runs takes $Theta(1)$ time. In this case, i doubles with each iteration of the outer loop. Being a little sloppy with math, we have:

$$\Theta\left(\sum_{k=1}^{\lfloor \log_2 N \rfloor}\right) = \Theta\left(\frac{1-2^{\lfloor \log_2 N \rfloor}+1}{1-2} - 2^0\right) \approx \Theta\left(\frac{1-2N}{-1} - 1\right) = \Theta(2N-2) = \Theta(N)$$

(c)
```
public static void f3(int[] a) {
    if (a.length == 0) { return; }
    int N = a.length;
    int[] newA = new int[N-1];
    for (int i = 0; i < newA.length; i += 1) {
        newA[i] = a[i];
    }
    f3(newA);
}
```

Runtime: ___$\Theta(N^2)$___

The easiest way to explain this question is to use a diagram to show the amount of work that is being done at each level of recursion:

|——— N ———| Amount of Work at this level:

*** ..... *** N

.....

**** 4

*** 3

** 2

* 1

The amount of work that is being done at each level comes from needing to copy each element of the array a into newA. Thus, we observe the same idea as in f1:

$$\Theta(1+2+...+N) = \Theta\left(\frac{N(N+1)}{2}\right) = \Theta(N^2)$$

(d)
```
public static void f4(int N) {
    int x = N;
    while (x > 0) {
        x = x >>> 1;
    }
}
```

Runtime: ___$\Theta(logN)$___

Effectively, this function is right-shifting the bits until all of the bits become zero. Note that the problem asked you to assume that there is no limit to the size of an int. The number of iterations of the while loop is related to the number of bits b in the integer, and we know that $b = \lceil \log_2 N \rceil$.

(e)
```
public static void f5(int N) {
    f1(N);
    f2(N);
    f3(new int[N]);
    f4(N);
}
```

Runtime: ___$\Theta(N^2)$___

Since we are simply running each of the functions one after another, we just add the runtimes together. In asymptotic analysis, we drop all of the non-dominating terms, so we have:

$$\Theta(N^2 + N + N^2 + logN) = \Theta(N^2).$$