CS 61B Spring 2022

Bits & Algorithmic Analysis

Exam Prep Discussion 7: February 28, 2022

1 Asymptotics Introduction

Give the runtime of the following functions in Θ notation. Your answer should be as simple as possible with no unnecessary leading constants or lower order terms.

```
private void f1(int N) {
    for (int i = 1; i < N; i++) {
        for (int j = 1; j < i; j++) {
            System.out.println("hello tony");
        }
    }
}
Θ(___)
```

Solution: $\Theta(N^2)$

Explanation: The inner loop does up to i work each time, and the outer loop increments i each time. Summing over each loop, we get that $1+2+3+4+\ldots+N = \Theta(N^2)$.

```
private void f2(int N) {
    for (int i = 1; i < N; i *= 2) {
        for (int j = 1; j < i; j++) {
            System.out.println("hello hannah");
        }
    }
}
Θ(___)
```

Solution: $\Theta(N)$

Explanation: The inner loop does i work each time, and we double i each time until reaching N. $1 + 2 + 4 + 8 + \ldots + N = \Theta(N)$

Here is a video walkthrough of both parts.

2 Finish the Runtimes

Below we see the standard nested for loop, but with missing pieces!

```
for (int i = 1; i < ______; i = ______) {
    for (int j = 1; j < ______; j = ______) {
        System.out.println("We will miss you next semester Akshit :(");
    }
}
```

For each part below, **some** of the blanks will be filled in, and a desired runtime will be given. Fill in the remaining blanks to achieve the desired runtime! There may be more than one correct answer.

Hint: You may find `Math.pow` helpful.

(a) Desired runtime: $\Theta(N^2)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = ______) {
        System.out.println("This is one is low key hard");
    }
}
```

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = j + 1) {
        System.out.println("This is one is low key hard");
    }
}
```

Explanation: Remember the arithmetic series $1+2+3+4+\ldots+N = \Theta(N^2)$. We get this series by incrementing j by 1 per inner loop.

(b) Desired runtime: $\Theta(log(N))$

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < ______; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Any constant would work here, 2 was chosen arbitrarily.

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < 2; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Explanation: The outer loop already runs $\log n$ times, since i doubles each time. This means the inner loop must do constant work (so any constant `j < k` would work).

(c) Desired runtime: $\Theta(2^N)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < ______; j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < Math.pow(2, i); j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

Explanation: Remember the geometric series $1 + 2 + 4 + ... + 2^N = \Theta(2^N)$. We notice that i increments by 1 each time, so in order to achieve this 2^N runtime, we must run the inner loop 2^i times per outer loop iteration.

(d) Desired runtime: $\Theta(N^3)$

```
for (int i = 1; i < ______; i = i * 2) {
    for (int j = 1; j < N * N; j = ______) {
        System.out.println("yikes");
    }
}
```

```
for (int i = 1; i < Math.pow(2, N); i = i * 2) {
    for (int j = 1; j < N * N; j = j + 1) {
        System.out.println("yikes");
    }
}
```

Explanation: One way to get N^3 runtime is to have the outer loop run N times, and the inner loop run N^2 times per outer loop iteration. To make the outer loop run N times, we need stop after multiplying `i = i * 2` N times, giving us the condition `i < Math.pow(2, N)`. To make the inner loop run N^2 times, we can simply increment by 1 each time.

3 Asymptotic Expressions

(a) Which of the following expressions are true? Check all that apply. Equations between asymptotic expressions, such as $O(f) = O(g)$ simply mean that all functions that are $O(f)$ are also $O(g)$ and vice-versa. An expression such as $O(f) \subseteq O(g)$ means that all functions that are $O(f)$ are also $O(g)$.

- ☐ $\Theta(1000 * N^3 + N * \log(N)) = \Theta(N^3))$.
- ☐ For all $k \geq 0$, $O(N^k) \subseteq O(N^{k+1}))$.
- ☐ For all $k \geq 0$, $\Omega(N^k) \subseteq \Omega(N^{k+1}))$.
- ☐ For positive-valued functions f and g, if $f = \Omega(g)$ and $g = O(h)$, $f = \Omega(h)$.
- ☐ For positive-valued functions f and g, if $f = \Omega(g)$ and $h = O(g)$, $f = \Omega(h)$.

Solution:

- ■ $\Theta(1000 * N^3 + N * \log(N)) = \Theta(N^3))$.
  True, we ignore lower order terms.
- ■ For all $k \geq 0$, $O(N^k) \subseteq O(N^{k+1}))$.
  True, every function that is $O(N^k)$ is also $O(N^{k+1})$ since $O(N^{k+1})$ is a less tight bound.
- ☐ For all $k \geq 0$, $\Omega(N^k) \subseteq \Omega(N^{k+1}))$.
  False, a function that runs in $\Theta(N^k)$ runs in $\Omega(N^k)$ but not $\Omega(N^{k+1})$.
- ☐ For positive-valued functions f and g, if $f = \Omega(g)$ and $g = O(h)$, $f = \Omega(h)$.
  False, f and h are lower bounded by g, but we can't say anything their relation.
- ■ For positive-valued functions f and g, if $f = \Omega(g)$ and $h = O(g)$, $f = \Omega(h)$.
  True, f is lower bounded by g and g upper bounds h, so f is also lower bounded by h.

(b) For positive-valued functions $f_0 \ldots f_k$, where we define $f_i(n) = 1 + f_{n\%i}(n)$ for $i \geq 1$ and $f_0(n) = 1$, which of the following are true? Check all that apply. Assume that $n > k$.

- ☐ The evaluation of $f_k(n)$ may run forever.
- ☐ $f_k(n) = \Omega(log(k))$, with respect to k.
- ☐ $f_k(n) = O(k)$, with respect to k.
- ☐ $f_k(n) = \Theta(1)$, with respect to n.
- ☐ If $n = k! - 1$, $f_k(n) = \Theta(k)$, with respect to k.

Solution:

- ☐ The evaluation of $f_k(n)$ may run forever.
  False, notice that $n\%i$ is bounded between 0 and $i-1$, so $f_k(n)$ will recurse on some function $f_i(n)$ where $i < k$, and eventually the base case must be hit.

- ☐ $f_k(n) = \Omega(log(k))$, with respect to k.
  False, $f_k(n)$ could take constant time, e.g. when $n = 2 \times k$.

- ■ $f_k(n) = O(k)$, with respect to k.
  True, see the last part for the worst case behavior of $f_k(n)$

- ■ $f_k(n) = \Theta(1)$, with respect to n.
  True, since $f_k(n)$ recurses on $f_{n\%k}(n)$, the remainder operation bounds $n\%k$ between 0 and $k-1$, which is independent of n.

- ■ If $n = k! - 1$, $f_k(n) = \Theta(k)$, with respect to k.
  True, notice that $k!$ is divisible by every number between 1 and k, so when $k! - 1$ is divided by any i between 1 and k, it will have remainder $i - 1$. As such, $f_k(n)$ will recurse on $f_{k-1}(n)$, which will recurse on $f_{k-2}(n)$, and so on until $f_0(n)$ is hit, taking linear time with respect to k.

4 Prime Factors

Determine the best and worst case runtime of `prime_factors` in $\Theta(.)$ notation as a function of `N`.

```
1  int prime_factors(int N) {
2      int factor = 2;
3      int count = 0;
4      while (factor * factor <= N) {
5          while (N % factor == 0) {
6              System.out.println(factor);
7              count += 1;
8              N = N / factor;
9          }
10         factor += 1;
11     }
12     return count;
13 }
```

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

Solution:
Best Case: $\Theta(log(N))$, Worst Case: $\Theta(\sqrt{N})$
Explanation: In the best case, `N` is some power of 2. Then the inner while loop will halve `N` each time until it becomes 1. At this point, both the inner and outer while loop conditions will be false and the function will return. Halving `N` each time results in a $\Theta(\log N)$ runtime.
In the worst case, `N` will not be divisible by any value of `factor`. This means we increment `factor` by 1 each time until `factor * factor > N`. This is at most $\sqrt{N}$ loops.