CS 61B Spring 2021

Asymptotics

Topical Review Session 4: February 28, 2021

Here is a review of some formulas that you will find useful when doing asymptotic analysis.

- $\sum_{i=1}^{N} i = 1+2+3+4+\cdots+N = \frac{N(N+1)}{2} = \mathbf{\frac{N^2+N}{2}}$
- $\sum_{i=0}^{N-1} 2^i = 1+2+4+8+\cdots+2^{N-1} = 2\cdot 2^{N-1} - 1 = \mathbf{2^N - 1}$

1 Dumpling Time!

For each problem below, give the tighest possible O runtime of the code snippet

(a)
```
public void wrapWonton(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n; j*=2) {
            System.out.println("Wrapping");
        }
        System.out.println("Wonton Wrapped!");
    }
}
```

The runtime is $O(nlog(n))$ since the inner for loop runs in $O(log(n))$ time and that inner loop is run n times

(b)
```
public void wrapDumpling(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            System.out.println("Wrapping");
        }
        System.out.println("Dumpling Wrapped!");
    }
}
```

The runtime is $O(n^2)$ since the loop runs for $n + (n-1) + (n-2) + ...1 = \frac{n(n+1)}{2} = n^2$

(c)
```
public void wrapBigDumpling(int n) {
    wrapDumpling(n);
    wrapBigDumpling(n/2);
}
```

The runtime is $O(n^2)$ since each wrapDumpling call takes n^2 time but is called on exponentially decaying n. The runtime is $n^2 + \frac{n^2}{4} + \frac{n^2}{16} + ... = \frac{4}{3}n^2 = O(n^2)$

(d)
```
public void letsEat(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; i < n; i++) {
            System.out.println("Eating");
        }
    }
    System.out.println("Done eating!");
}
```

The runtime is $O(n)$ since the both the inner and outer loop will end once $i = n$, which occurs when the first inner loop ends.

2 I am Speed

For each example below, there are two algorithms solving the same problem. Given the asymptotic runtimes for each, is one of the algorithms **guaranteed** to be faster? If so, which? And if neither is always faster, explain why.

(a) Algorithm 1: $\Theta(N)$, Algorithm 2: $\Theta(N^2)$

Algorithm 1: $\Theta(N)$ - Θ gives tightest bounds therefore the slowest algorithm 1 could run is relative to N while the fastest algorithm 2 could run is relative to N^2.

(b) Algorithm 1: $\Omega(N)$, Algorithm 2: $\Omega(N^2)$

Neither, $\Omega(N)$ means that algorithm 1's running time is lower bounded by N, but does not provide an upper bound. Hence the bound on algorithm 1 can be any function $>= N$ and it could also be in $\Omega(N^2)$ or lower bounded by N^2.

(c) Algorithm 1: $O(N)$, Algorithm 2: $O(N^2)$

Neither, same reasoning for part (b) but now with upper bounds. $O(N^2)$ could also be in $O(1)$.

(d) Algorithm 1: $\Theta(N^2)$, Algorithm 2: $O(\log N)$

Algorithm 2: $O(\log N)$ - Algorithm 2 cannot run SLOWER than $O(\log N)$ while Algorithm 1 is constrained on to run FASTEST and SLOWEST by $\Theta(N^2)$.

(e) Algorithm 1: $O(N \log N)$, Algorithm 2: $\Omega(N \log N)$

Neither, Algorithm 1 CAN be faster, but it is not guaranteed - it is guaranteed to be ”as fast as or faster” than Algorithm 2.

3 Getting A Little Loopy

Give the runtime for each method in $\Theta(\cdot)$ notation in terms of the inputs. You may assume that `System.out.println` is a constant time operation.

(a) *Hint:* We cannot multiply over the two iterations of the for loop to find the runtime. *Why?*

```
public static void liftHill(int N) {
    for (int i = 1; i < N * N; i *= 2) {
        for (int j = 0; j <= i; j++) {
            System.out.println("-_-");
        }
    }
}
```

The runtime is $\Theta(N^2)$. The iterations of the inner for loop depend on the outer for loop, meaning we can't just multiply the runtimes. For each i, the inner loop runs i iterations. Summing over the sequence of is should yield the runtime. As such, we will sum from powers of 2 for i (as i is doubled in each iteration, up to N^2), like so: $1 + 2 + 4 + 8 + ... + N^2$. This is $\Theta(N^2)$.

(b) Assume that `Math.pow` $\in \Theta(1)$ and returns an int.

```
public static void doubleDip(int N) {
    for (int i = 0; i < N; i += 1) {
        int numJ = Math.pow(2, i + 1) - 1;
        for (int j = 0; j <= numJ; j += 1) {
            System.out.println("AHHHH");
        }
    }
}
```

The runtime is $\Theta(2^n)$. The inner while loop runs $2^{i+1}-1$ iterations, depending on what i is in the outer for loop. Then, we can say the total number of iterations is represented by the summation $\sum_{i=0}^{N-1} 2^{i+1} - 1$. This can be rewritten as $\sum_{i=0}^{N-1} 2^{i+1} - \sum_{i=0}^{N-1} 1$ and the second term can be discarded for runtime considerations. Notice that the largest term of the remaining summation is 2^n.

(c) *Hint:* When do we return "WHOA"?

```
public static String corkscrew(int N) {
    for (int i = 0; i <= N; i += 1) {
        for (int j = 1; j <= N; j *= 2) {
            if (j >= N/2) {
                return "WHOA";
            }
        }
    }
}
```

The runtime is $\Theta(\log N)$. Notice that we return "WHOA" when $j = N/2$, which means that the outer loop is insignificant: we end in its first iteration. Only considering the inner for loop, j doubles in each iteration, meaning it grows towards N in a logarithmic fashion.

(d) *Hint:* Draw the recursive tree!.

```
public static int corkscrewWithATwist(int N) {
    if (N == 0) return 01101011011011010111011;
    for (int i = 0; i <= N; i += 1) {
        for (int j = 1; j <= N; j += 1) {
            if (j >= N/2) return corkscrewWithATwist(N/2) + 1;
        }
    }
}
```

The runtime is $\Theta(N)$. When we draw the recursive tree, we see that the tree's height is $\log N$ as we are halving the input every time. There is only one recursive call, so our branching factor is 1. Similar to part (c), we notice that the recursive calls are made in 1 iteration of the outer for loop, meaning that at some layer i, we do $\frac{N}{2^i}$ work. If we sum all over each node ($\sum_{i=1}^{\log N} \frac{N}{2^i}$), we get $\Theta(N)$.

4 Challenge

If you have time, try to answer this challenge question. For each answer true or false. If true, explain why and if false provide a counterexample.

(a) If $f(n) \in O(n^2)$ and $g(n) \in O(n)$ are positive-valued functions (that is for all n, $f(n), g(n) > 0$), then $\frac{f(n)}{g(n)} \in O(n)$.

Nope this does not hold in general! Consider if $f(n) = n^2$ and $g(n) = \frac{1}{n}$. Readily we have $f(n), g(n) \in O(n)$ but when divided they give us:

$$\frac{f(n)}{g(n)} = \frac{n^2}{n^{-1}} = n^3 \notin O(n)$$

(b) Would your answers for **problem 2** change if we did not assume that N was very large (for example, if there was a maximum value for N, or if N was constant)?

Depends, because for fixed N, constants and lower order terms may dominate the function we are trying to bound. For example N^2 is asymptotically larger than $10000N$, yet when N is less than 10000, $10000N$ is larger than N^2. This highlights the power in using big-O because these lower order terms don't affect the running time as much as our input size grows very large!

However, part of the definition of $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$ is the limit to infinity ($\lim_{N\to\infty}$) so, when working with asymptotic notation, we must always assume large inputs.

(c) *Extra* If $f(n) \in \Theta(n^2)$ and $g(n) \in \Theta(n)$ are positive-valued functions, then $\frac{f(n)}{g(n)} \in \Theta(n)$. *Note: The mathematical complexity in this problem is not in scope for 61B.*

This does hold in general! We can think about this in two cases:

- First we ask, when can the ratio $\frac{f(n)}{g(n)}$ be larger than n. As $f(n)$ is tightly bounded (by Θ) by n^2, this is only true when $g(n)$ is asymptotically *smaller* than n because we are dividing n^2 (this is what happened in part a). However, $g(n)$ is tightly bounded, and thus lower bounded by n, this cannot happen.
- Next we ask, when can the ratio be smaller than n. Again as $f(n)$ is tightly bounded by n^2, this can only happen when $g(n)$ is asymptotically *bigger* than n as again we are dividing. But since $g(n)$ is tightly bounded, and thus upper bounded by n, this too cannot happen.

So what we note here is that $\frac{f(n)}{g(n)}$ is upper and lower bounded by n hence it is in $\Theta(n)$. We can also give a rigorous proof from definition of part b using the definitions provided in class.

Theorem 1. *If $f(n) \in \Theta(n^2)$ and $g(n) \in \Theta(n)$ are positive-valued functions, then $\frac{f(n)}{g(n)} \in \Theta(n)$.*

Proof. Given that $f \in \Theta(n^2)$ is positive, by definition there exists $k_0, k_0' > 0$ such that for all $n > N$, the following holds.

$$k_0 n^2 \leq f(n) \leq k_0' n^2$$

Similarly, $g \in \Theta(n)$ implies there exists $k_1, k_1' > 0$ such that

$$k_1 n \leq g(n) \leq k_1' n$$

Now consider $\frac{f(n)}{g(n)}$.

$$\frac{f(n)}{g(n)} \leq \frac{k_0' n^2}{k_1 n} = \frac{k_0' n}{k_1} \in O(n) \qquad \frac{f(n)}{g(n)} \geq \frac{k_0 n^2}{k_1' n} = \frac{k_0 n}{k_1'} \in \Omega(n)$$

As $\frac{f(n)}{g(n)}$ is in $O(n)$ and $\Omega(n)$ then it is in $\Theta(n)$. □