CS 61B

Spring 2018

More Asymptotic Analysis

Discussion 8: March 6, 2018

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

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

Intuition

For the following recursive functions, give the worst case and best case running time in the appropriate $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$ notation.

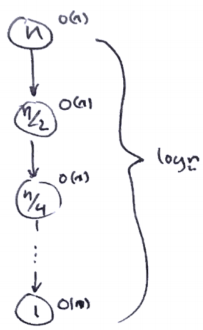
The meta-strat on this problem is to explore a rigourous framework to analyze running time for recursive procedures. Specifically, one can derive the running time by drawing the recursive tree and accounting for three pieces of information.

- The height of the tree.
- The branching factor of each node.
- The amount of work each node contributes relative to its input size.

1.1 Give the running time in terms of N.

```
public void andslam(int N) {
    if (N > 0) {
        for (int i = 0; i < N; i += 1) {
            System.out.println("datboi.jpg");
        }
        andslam(N / 2);
    }
}
```

andslam(N) runs in time $\Theta(N)$ worst and best case. One potentially tricky portion is that the $\sum_{i=0}^{\log n} 2^{-i}$ is at most 2 because the geometric sum as it goes to infinity is bounded by 2! (2 *exclamation mark* not "2 factorial")



[1] Height of tree

→ how many times can you divide n by 2 until you get n = 1. Let h be height.

$\frac{n}{2^h} = 1 \rightarrow n = 2^h \rightarrow h = \log_2 n$

[2] Branching Factor

→ Note each time andslam is called, it makes 1 recursive call on n/2.

→ # nodes per layer = 1.

[3] Amount of work each node does.

→ linear relative to input size. so O(n).

Now running time ~~can be~~ of entire recursive procedure can be calcula by summing over entire recursive tree.

running time = (sum over) # layers · $\left(\frac{\text{\# nodes}}{\text{\# layer}}\right) \cdot \left(\frac{\text{amount work}}{\text{1 node}}\right)$

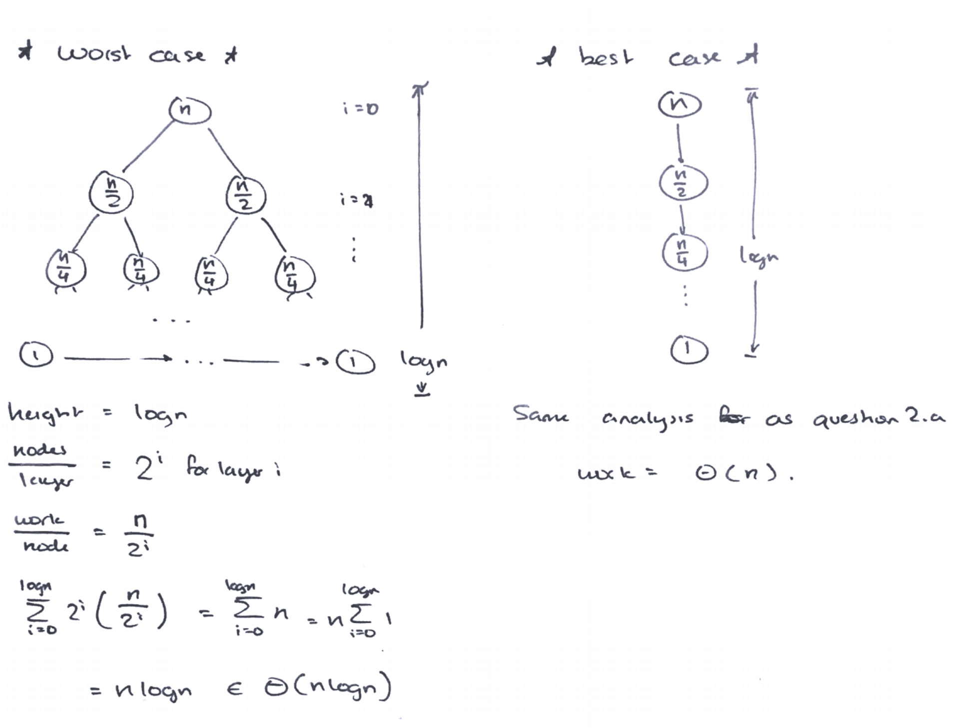
$= \sum_{i=0}^{\log n} (1) \cdot \left(\frac{n}{2^i}\right)$ (layers; # nodes / layer; work / node)

$= \sum_{i=0}^{\log n} \frac{n}{2^i} = n \sum_{i=0}^{\log n} \frac{1}{2^i} \leq 2n \in \Theta(n)$

1.2 Give the running time for `andwelcome(arr, 0, N)` where N is the length of the input array `arr`.

```java
public static void andwelcome(int[] arr, int low, int high) {
    System.out.print("[ ");
    for (int i = low; i < high; i += 1) {
         System.out.print("loyal  ");
    }
    System.out.println("]");
    if (high - low > 0) {
        double coin = Math.random();
        if (coin > 0.5) {
            andwelcome(arr, low, low + (high - low) / 2);
        } else {
            andwelcome(arr, low, low + (high - low) / 2);
            andwelcome(arr, low + (high - low) / 2, high);
        }
    }
}
```

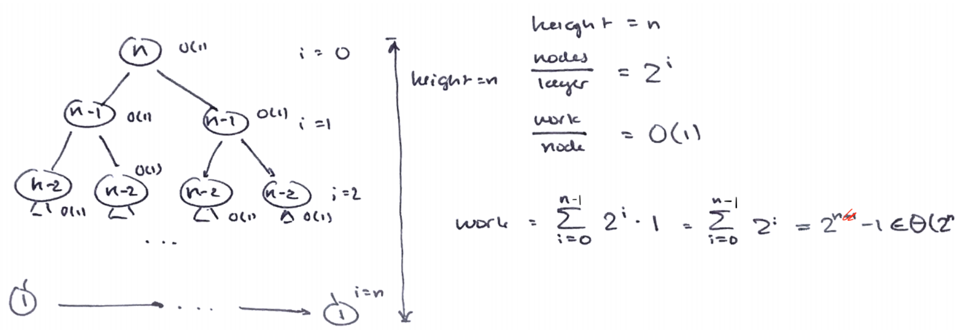
`andwelcome(arr, 0, N)` runs in time $\Theta(N \log N)$ worst case and $\Theta(N)$ best case. The recurrence relation is different for each case. In the worst case you always flip the wrong side of the coin resulting in a branching factor of 2. Because there is a branching factor of 2, there are 2^i nodes in the i-th layer. Meanwhile, the work you do per node is linear with respect to the size of the input. Hence in the i-th layer, the work done is about $\frac{n}{2^i}$. In the best case you always flip the right side of the coin giving a branching factor of 1. The analysis is then the same as the previous problem!



1.3 Give the running time in terms of N.

```
public int tothe(int N) {
    if (N <= 1) {
        return N;
    }
    return tothe(N - 1) + tothe(N - 1);
}
```

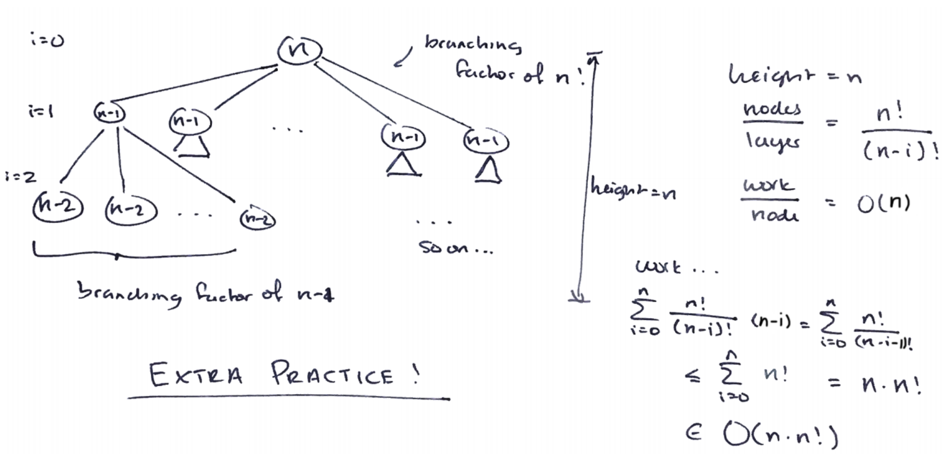
For `tothe(N)` the worst and best case are $\Theta(2^N)$. Notice that at the i-th layer, there are 2^i nodes. Each node does constant amount of work so with the fact that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$, we can derive the following.



1.4 Give the running time in terms of N.

```java
public static void spacejam(int N) {
    if (N <= 1) {
        return;
    }
    for (int i = 0; i < N; i += 1) {
        spacejam(N - 1);
    }
}
```

For `spacejam(N)` the worst and best case is $O(N \cdot N!)$. Now for the i-th layer, the number of nodes is $n \cdot (n-1) \cdot \ldots \cdot (n-i)$ since the branching factor starts at n and decrements by 1 each layer. Actually calculating the sum is a bit tricky because there is a pesky $(n-i)!$ term in the denominator. We can upper bound the sum by just removing the denominator, but in the strictest sense we would now have a big-O bound instead of big-Θ.



Hey you watchu gon do

2.1 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 could not be tight 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.

Would your answers above 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)?

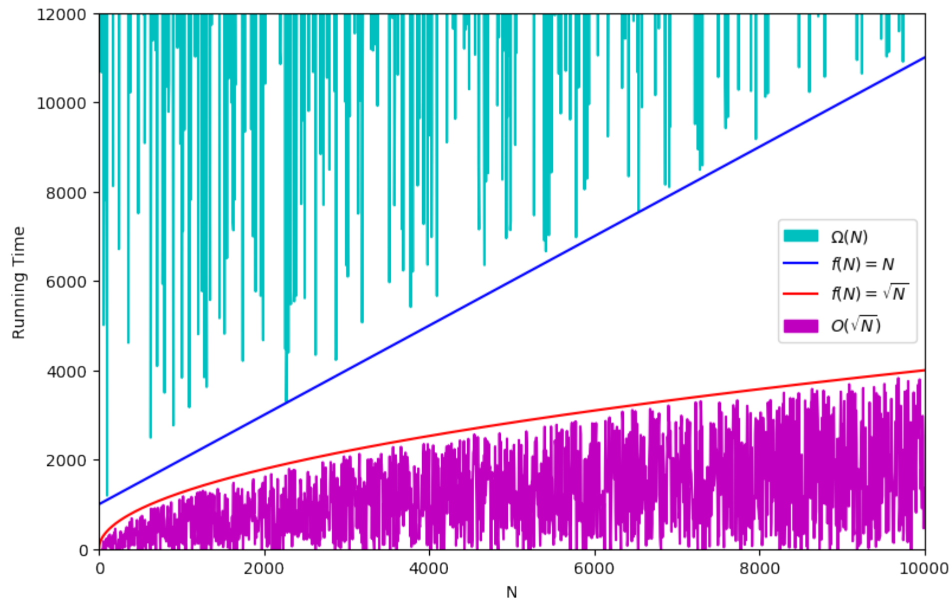
Depends, because for fixed N, constants and lower order terms may dominate the function we are trying to bound. For example N^2 is asymptotically larger than $10000N$, yet when N is less than 10000, $10000N$ is larger than N^2. This highlights the power in using big-O because these lower order terms 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.

Asymptotic Notation

3.1 Draw the running time graph of an algorithm that is $O(\sqrt{N})$ in the best case and $\Omega(N)$ in the worst case. Assume that the algorithm is also trivially $\Omega(1)$ in the best case and $O(\infty)$ in the worst case.

Below we have the graph an example solution. Assume that the cyan region extends arbitrarily towards infinity. Ignoring constants, the algorithm running time takes at most $\sqrt{N}$ time (and trivially at least constant time) in the worst case. The algorithm also takes at least N time (and trivially at most infinite time) in the worst case.



Extra: Following is a question from last week, now that you have properly learned about $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$.

3.2 Are the statements in the right column true or false? If false, correct the asymptotic notation ($\Omega(\cdot)$, $\Theta(\cdot)$, $O(\cdot)$). Be sure to give the tightest bound. $\Omega(\cdot)$ is the opposite of $O(\cdot)$, i.e. $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$.

$f(n) = 20501$	$g(n) = 1$	$f(n) \in O(g(n))$
$f(n) = n^2 + n$	$g(n) = 0.000001n^3$	$f(n) \in \Omega(g(n))$
$f(n) = 2^{2n} + 1000$	$g(n) = 4^n + n^{100}$	$f(n) \in O(g(n))$
$f(n) = \log(n^{100})$	$g(n) = n \log n$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n + 3^n + n$	$g(n) = n^2 + n + \log n$	$f(n) \in \Omega(g(n))$
$f(n) = n \log n + n^2$	$g(n) = \log n + n^2$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n$	$g(n) = (\log n)^2$	$f(n) \in O(g(n))$

- True, although $\Theta(\cdot)$ is a better bound.
- False, $O(\cdot)$. Even though n^3 is strictly worse than n^2, n^2 is still in $O(n^3)$ because n^2 is always as good as or better than n^3 and can never be worse.
- True, although $\Theta(\cdot)$ is a better bound.
- False, $O(\cdot)$.
- True.
- True.
- False, $\Omega(\cdot)$.

Fall 2015 *Extra*

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

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 $\Theta(n)$. □