

---

## 1 More Running Time

---

Give the worst case and best case running time in  $\Theta(\cdot)$  notation in terms of  $M$  and  $N$ .

(a) Assume that `comeon()`  $\in \Theta(1)$  and returns a boolean.

```
1 for (int i = 0; i < N; i += 1) {
2     for (int j = 1; j <= M; ) {
3         if (comeon()) j += 1;
4         else          j *= 2;
5     }
6 }
```

(a) For `comeon()` the worst case is  $\Theta(NM)$  and the best case is  $\Theta(N \log M)$ . To see this, note that in the best case `comeon()` always returns false. Hence  $j$  multiplies by 2 each iteration. The inner loop would execute relative to  $\log M$  and the outer loop iterates  $N$  times. In the worst case, `comeon()` always returns true, thus the inner loop iterates  $M$  times.

## 2 Recursive Running Time

---

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 rigorous 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.

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

(a) Give the running time in terms of  $N$ .

```
1 public void andslam(int N) {
2     if (N > 0) {
3         for (int i = 0; i < N; i += 1) {
4             System.out.println("datboi.jpg");
5         }
6         andslam(N / 2);
7     }
8 }
```

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

```
1 public static void andwelcome(int[] arr, int low, int high) {
2     System.out.print("[ ");
3     for (int i = low; i < high; i += 1) {
4         System.out.print("loyal ");
5     }
6     System.out.println("]");
7     if (high - low > 0) {
8         double coin = Math.random();
9         if (coin > 0.5) {
10            andwelcome(arr, low, low + (high - low) / 2);
11        } else {
12            andwelcome(arr, low, low + (high - low) / 2);
13            andwelcome(arr, low + (high - low) / 2, high);
14        }
15    }
16 }
```

(c) Give the running time in terms of  $N$ .

```
1 public int tothe(int N) {
2     if (N <= 1) {
3         return N;
4     }
5     return tothe(N - 1) + tothe(N - 1);
6 }
```

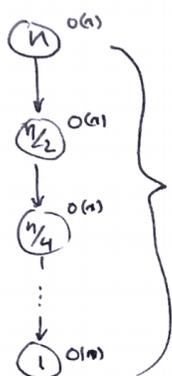
(d) *Extra Hard!* Give the running time in terms of  $N$

```

1 public static void spacejam(int N) {
2     if (N == 1) {
3         return;
4     }
5     for (int i = 0; i < N; i += 1) {
6         spacejam(N - 1);
7     }
8 }

```

(a) `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!



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.

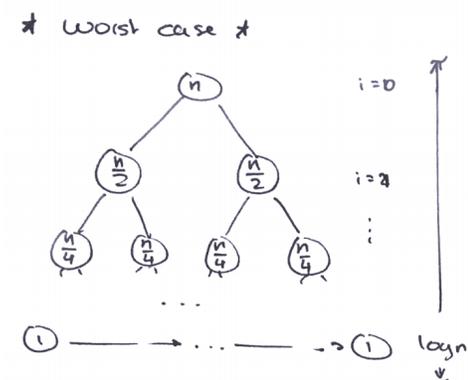
3) Amount of work each node does.

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

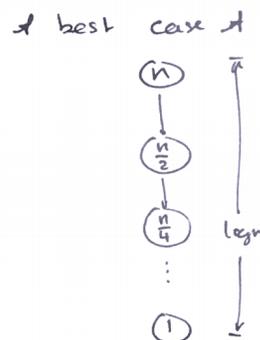
Now running time ~~can be~~ of entire recursive procedure can be calculated by summing over entire recursive tree.

$$\begin{aligned}
 \text{running time} &= \sum_{\text{# layers}} \left( \frac{\text{\# nodes}}{\text{\# layers}} \right) \cdot \left( \frac{\text{amount work}}{\text{\# 1 node}} \right) \\
 &= \sum_{i=0}^{\log n} \underbrace{(1)}_{\text{layers}} \cdot \underbrace{\left( \frac{n}{2^i} \right)}_{\text{work/node}} \\
 &= \sum_{i=0}^{\log n} \frac{n}{2^i} = n \sum_{i=0}^{\log n} \frac{1}{2^i} \leq 2n \in \Theta(n)
 \end{aligned}$$

- (b) `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!

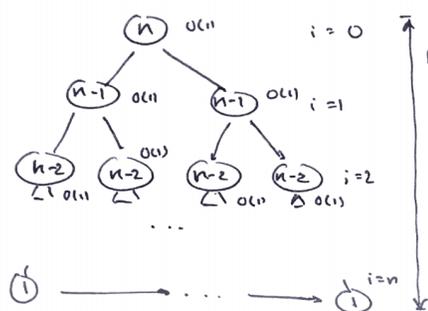


height =  $\log n$   
 $\frac{\text{nodes}}{\text{layer}} = 2^i$  for layer  $i$   
 $\frac{\text{work}}{\text{node}} = \frac{n}{2^i}$   
 $\sum_{i=0}^{\log n} 2^i \left( \frac{n}{2^i} \right) = \sum_{i=0}^{\log n} n = n \sum_{i=0}^{\log n} 1$   
 $= n \log n \in \Theta(n \log n)$



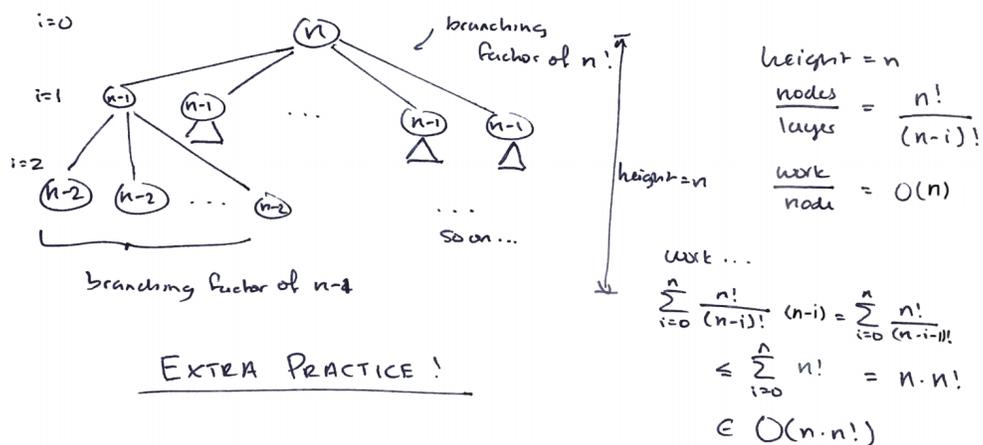
Same analysis as question 2.a  
work =  $\Theta(n)$ .

- (c) 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.



height =  $n$   
 $\frac{\text{nodes}}{\text{layer}} = 2^i$   
 $\frac{\text{work}}{\text{node}} = O(1)$   
work =  $\sum_{i=0}^{n-1} 2^i \cdot 1 = \sum_{i=0}^{n-1} 2^i = 2^{n+1} - 1 \in \Theta(2^n)$

(d) For spacejam(N) the worst and best case is  $O(N \cdot N!)$ . Now for the  $i$ -th layer, the number of nodes is  $n \cdot (n-1) \cdot \dots \cdot (n-i)$  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- $\Theta$ .



### 3 Hey you watchu gon do

---

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. Assume the algorithms have very large input (so  $N$  is very large).

- (a) Algorithm 1:  $\Theta(N)$ , Algorithm 2:  $\Theta(N^2)$
- (b) Algorithm 1:  $\Omega(N)$ , Algorithm 2:  $\Omega(N^2)$
- (c) Algorithm 1:  $O(N)$ , Algorithm 2:  $O(N^2)$
- (d) Algorithm 1:  $\Theta(N^2)$ , Algorithm 2:  $O(\log N)$
- (e) Algorithm 1:  $O(N \log N)$ , Algorithm 2:  $\Omega(N \log N)$

- (a) Algorithm 1:  $\Theta(N)$  -  $\Theta$  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) 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) Neither, same reasoning for part (b) but now with upper bounds.  $O(N^2)$  could also be in  $O(1)$ .
- (d) 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) 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?

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!

## 4 More Extra Problems [Final FA15]

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  $\Theta$ ) 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 4.1.** 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  $\Theta(n)$ . □