
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 `slam()` $\in \Theta(1)$ and returns a boolean.

```
1 public void comeon() {
2     int j = 0;
3     for (int i = 0; i < N; i += 1) {
4         for (; j < M; j += 1) {
5             if (slam(i, j))
6                 break;
7         }
8     }
9
10    for (int k = 0; k < 1000 * N; k += 1) {
11        System.out.println("space jam");
12    }
13 }
```

(a) For `comeon()` the worst case is $\Theta(M + N)$ and the best case is $\Theta(N)$. To see this, note that `j` is declared outside of the for loops. In the best case, `slam(i, j)` always returns true. However, in the worst case where it's always false, `j` can only increment to at most M .

2 Recursive Running Time

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

This exercise targets understanding of how branching and input size affect the running time of analyzing recursive functions. I would explain each problem by writing the recurrence relation, draw the tree out, and ask (1) what's the branching factor, (2) how much work does each node contribute, (3) what's the height of the tree.

(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("bigballer.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) + tothe(N - 1);
6 }
```

(d) 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. The recurrence relation is $T(N) = T(\frac{N}{2}) + O(N)$.
- (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. The recurrence relation is $T(N) = 2T(\frac{N}{2}) + O(N)$. In the best case you always flip the right side of the coin giving a branching factor of 1. The recurrence relation is $T(N) = T(\frac{N}{2}) + O(N)$.
- (c) For `tothe(N)` the worst and best case are $\Theta(3^N)$. The recurrence relation is $T(n) = 3T(n-1) + O(1)$. Draw the call tree out and for each node notice that there is constant work associated with it. The running time scales w.r.t. the number of nodes and we know that the number of nodes for a complete ternary tree is of order 3^N .
- (d) For `spacejam(N)` the worst and best case is $\Theta(N!)$. The recurrence relation is $T(n) = nT(n-1) + O(1)$. To explain, I would again draw the tree out and make an argument based on the number of nodes.

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: $\Theta(N)$ - straight forward, Θ gives tightest bounds
 - (b) Neither, something in $\Omega(N)$ could also be in $\Omega(N^2)$
 - (c) Neither, something in $O(N^2)$ 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 best and worst case by $\Theta(N^2)$.
 - (e) Neither, Algorithm 1 CAN be faster, but is not guaranteed - it is guaranteed to be "as fast as or faster" than Algorithm 2.

Why did we need to assume that N was large? Asymptotic bounds often only make sense as N gets large, because constant factors may result in a function with a smaller order of growth growing faster than a faster one. For example, take the functions $1000n$ and n^2 . n^2 is asymptotically larger than $1000n$, but for small n , it will seem that $1000n$ is larger than n^2 .

4 Big Ballin' Bounds

1. Prove the following bounds by finding some constant $M > 0$ and input $N > 0$ for $M \in \mathbb{R}, N \in \mathbb{N}$ such that f and g satisfy the relationship.
 - (a) $f \in O(g)$ for $f = 2n, g = n^2$
 - (b) $f \in \Omega(g)$ for $f = 0.1n, g = 40$
 - (c) $f \in \Theta(g)$ for $f = \log(n), g = \log(n^a)$, for $a > 0$.

- (a) Let $M = 2, N = 1$. Then we see for all $n \geq N, |f(n)| \leq |Mg(n)|$.
- (b) Let $M = \frac{1}{400}, N = 1$. Then we see for all $n \geq N, f(n) \geq |Mg(n)|$. Alternatively, $M = 1$ and $N = 400$ also satisfies the bound.

(c) For $M = \frac{1}{a}, N = 2$, we have $|\log(n)| = |\frac{1}{a} \times \log(n^a)|$. It should be noted that this is an equality, which proves that these functions are asymptotically the same.

2. Answer the following claims with true or false. If false, provide a counterexample.

(a) If $f(n) \in O(g(n))$, then $500f(n) \in O(g(n))$.

(b) If $f(n) \in \Theta(g(n))$, then $2^{f(n)} \in \Theta(2^{g(n)})$.

(a) True.

(b) False, consider $f(n) = n, g(n) = 2n$. Then $2^{f(n)} = 2^n$, but $2^{g(n)} = 2^{2n} = (2^2)^n = 4^n$.