Section 02: Solutions

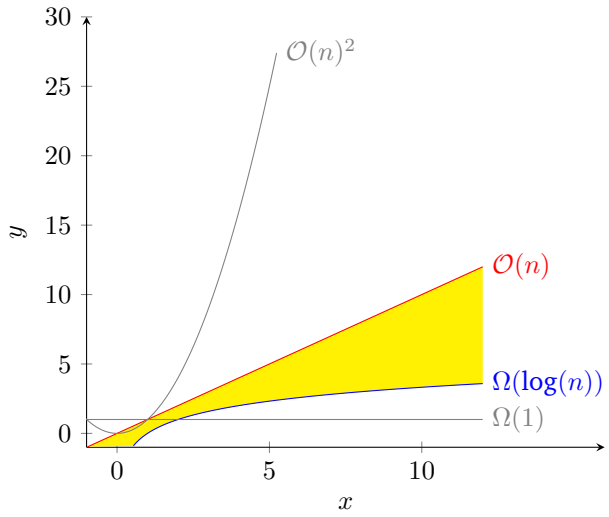
General Concepts

1. Simple Definitions

(a) What are $\mathcal{O}$ Θ Ω? What are the differences between these three?
Solution:

- $\mathcal{O}$ is informally the upper bound of asymptotic complexity.
- Ω is informally the lower bound of asymptotic complexity.
- $\mathcal{O}$ is the asymptotic complexity of the function if and only if its $\mathcal{O}$ is the same as its Θ.

(b) What is the meaning of tight bounds? Give two lower bounds, two upper bounds and theta bound for BSTs. If part of the question is not applicable, why?
Solution:



Tight bounds are the bounds that is on the edge of the range. The answers are shown in the graph above. BST doesn't have a Θbound since its tight $\mathcal{O}$bound and tight Ωbound are not the same. Thus, the runtime of BST could fall between those two, which is not a concret bound.

(c) In the worst case, is finding an element in a sorted array using binary search $\mathcal{O}(n)$? In the worst case, is finding an element in a sorted array using binary search $\Omega(n)$?
Solution:

Note that binary search takes $\log(n)$ time to complete. $\log(n)$ is upper-bounded by n, so $\log(n) \in \mathcal{O}(n)$. However, $\log(n)$ is not lower-bounded by n, which means $\log(n) \in \Omega(n)$ is false.

2. Comparing growth rates

(a) Order each of the following functions from fastest to slowest in terms of asymptotic growth. (By "fastest", we mean which function increases the most rapidly as n increases.)

- $\log_4(n) + \log_2(n)$
- $\frac{n}{2} + 4$
- $2^n + 3$
- $750,000,000$
- $8n + 4n^2$

Solution:

- $2^n + 3$
- $8n + 4n^2$
- $\frac{n}{2} + 4$
- $\log_4(n) + \log_2(n)$
- $750,000,000$

(b) For each of the above expressions, state the simplified tight $\mathcal{O}$ bound in terms of n.

Solution:

- $\mathcal{O}(\log(n))$
- $\mathcal{O}(n)$
- $\mathcal{O}(2^n)$
- $\mathcal{O}(1)$
- $\mathcal{O}(n^2)$

(c) Order each of these more esoteric functions from fastest to slowest in terms of asymptotic growth. (By "fastest", we mean which function increases the most rapidly as n increases.) Also state a simplified tight $\mathcal{O}$ bound for each.

- $2^{n/2}$
- 3^n
- 2^n

Solution:

- 3^n, which is in $\mathcal{O}(3^n)$
- 2^n, which is in $\mathcal{O}(2^n)$
- $2^{n/2}$, which is in $\mathcal{O}\left(\sqrt{2}^n\right)$ (or $\mathcal{O}(2^{n/2})$).

Constant multipliers don't matter in big-O notation, but a constant factor in the exponent **does** matter, since it corresponds to multiplying by some constant to the n^{th} power. Saying $2^{n/2}$ is in $\mathcal{O}(2^n)$ would be true, but it would not be a tight bound.

3. True or false?

(a) If a function is in $\Omega(n)$, then it could also be in $\mathcal{O}(n^2)$.

(b) If a function is in $\Theta(n)$, then it could also be in $\mathcal{O}(n^2)$.

(c) If a function is in $\Omega(n)$, then it is always in $\mathcal{O}(n)$.

Solution:

(a) True

(b) True

(c) False

As a reminder, we can think about $\mathcal{O}$ informally as an upper bound. If a function $f(n)$ is in $\mathcal{O}(g(n))$, then $g(n)$ is a function that *dominates* $f(n)$, and this domination can be really overshooting the mark. Every (correct) piece of code we write in this class will have a running time that is $\mathcal{O}\left(n!^{n!^{n!}}\right)$. Conversely, we can think about Ω informally as a lower bound. If a function $f(n)$ is in $\Omega(g(n))$, then $f(n)$ is a function that *dominates* $g(n)$, and this domination can be really overshooting the mark also. The running time of any piece of code is always in $\Omega(1)$. And finally, Θ is a much stricter definition. $f(n)$ is in $\Theta(g(n))$ (if and only if) $f(n)$ is in $\mathcal{O}(g(n))$ and in $\Omega(g(n))$. Usually when people say $\mathcal{O}$, they mean Θ.

Algorithm Analysis without Recurrence

4. Modeling code

For each of the following code blocks, give a summation that represents the worst-case runtime in terms of n, then evaluate the summation to give a tight big-O bound of it.

(a)

```
int x = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        x++;
    }
}
```

Solution:

One possible solution is

$$\begin{aligned} T(n) &= 1 + \sum_{i=0}^{n-1} \sum_{k=0}^{i-1} 1 \\ &= 1 + \sum_{i=0}^{n-1} i \\ &= 1 + \frac{(0+n-1)n}{2} \\ &= 1 + \frac{n(n-1)}{2} \\ &= 1 + \frac{n^2-n}{2} \\ &= O(n^2) \end{aligned}$$

(b)

```
int x = 0;
for (int i = n; i >= 1; i /= 2) {
    x += i;
}
```

Solution:

One possible solution is

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{\log(n)} 1 \\ &= 1 + log(n) \\ &= O(log(n)) \end{aligned}$$

(c)
```
int x = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n * n / 3; j++) {
        x += j;
    }
}
```

Solution:

One possible answer is $T(n) = \frac{n^3}{3}$. The inner loop performs approximately $\frac{n^2}{3}$ iterations; the outer loop repeats that n times, and each inner iteration does a constant amount of work.

So the tight worst-case runtime is $\mathcal{O}\left(n^3\right)$.

The exact constant you get doesn't matter here, since we'll ignore the constant when we put it into $\mathcal{O}$ notation anyway. For example, saying we do 3 operations per inner-loop iteration (checking the loop condition, updating `j`, and updating `x`) and getting n^3 instead of $n^3/3$ is also completely reasonable.

(d)
```
int x = 0;
for (int i = n; i >= 0; i -= 1) {
    if (i % 3 == 0) {
        break;
    } else {
        x += n;
    }
}
```

Solution:

The tightest possible big-$\mathcal{O}$ bound is $\mathcal{O}\left(1\right)$ because exactly one of n, $n-1$, or $n-2$ will be divisible by three for all possible values of n. So, the loop runs at most 3 times.

(e)
```
int x = 0;
for (int i = 0; i < n; i++) {
    if (i % 5 == 0) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                x += i * j;
            }
        }
    }
}
```

Solution:

While the inner-most if statement executes only once per loop, we must check if `i == j` is true once per each iteration. This will take some non-zero constant amount of time, so the inner-most loop will perform approximately n work (setting the constant factors equal to 1, is conventional, since constant factors can depend on things like system architecture, what else the computer is doing, the temperature of the room, etc.).

The outer-most loop and if statement will perform n work during only 1/5th of the iterations and will perform a constant amount of work the remaining 4/5ths of the time. So, the total amount work done is approximately $\frac{n}{5} \cdot n + \frac{4n}{5} \cdot 1$. If we simplify, this means we can ultimately model the runtime as

approximately $T(n) = \frac{n^2}{5} + \frac{4n}{5}$.

Therefore, the tightest worst-case asymptotic runtime will be $\mathcal{O}\left(n^2\right)$.

(f)
```
int x = 0;
for (int i = 0; i < n; i++) {
    if (n < 100000) {
        for (int j = 0; j < n; j++) {
            x += 1;
        }
    } else {
        x += 1;
    }
}
```

Solution:

Recall that when computing the asymptotic complexity, we only care about the behavior for large inputs. Once n is large enough, we will only execute the second branch of the if statement, which means the runtime of the code can be modeled as just $T(n) = n$. So, the tightest worst-case runtime is $\mathcal{O}(n)$.

(g)
```
int x = 0;
if (n % 2 == 0) {
    for (int i = 0; i < n * n * n * n; i++) {
        x++;
    }
} else {
    for (int i = 0; i < n * n * n; i++) {
        x++;
    }
}
```

Solution:

We can model the runtime of this function in the **general** case as:

$$T_g(n) = \begin{cases} n^4 & \text{when } n \text{ is even} \\ n^3 & \text{when } n \text{ is odd} \end{cases}$$

However, the prompt was asking you to prove a model for the **worst** possible case – that is, when n is even. If we assume n is even, we can produce the following model:

$$T_w(n) = n^4$$

The tightest worst-case asymptotic runtime is then $\mathcal{O}\left(n^4\right)$ in this case.

Something interesting to note is that the **general** model has differing tight big-$\mathcal{O}$ and big-Ω bounds and so therefore has no big-Θ bound.

That is, the best big-$\mathcal{O}$ bound we can give for $T_g(n)$ is $T_g(n) \in \mathcal{O}\left(n^4\right)$; the best big-$\Omega$ bound we can give is $T_g(n) \in \Omega\left(n^3\right)$. These two bounds ($n^4$ and n^3) are different so there is no big-Θ for T_g. Importantly however, there is a big-Θ for our simpler model, T_w. That is, $T_w(n) \in \Theta\left(n^4\right)$.

Algorithm Analysis with Recurrence

5. Practicing Recurrence Equations

With recursive or divide-and-conquer algorithms, it is often helpful to express the runtime recursively with a recurrence equation.

Identify recurrence equations for the following programs, then unroll your recurrence equations using the expansion or tree method to find a big-O bound.

(a) Consider the function f

```
f(n) {
    if (n <= 0) {
        return 1;
    }
    return 2 * f(n - 1) + 1;
```

Solution:

The following recurrence models the worst case time complexity of f(n)

$$T(n) = \begin{cases} c_0 & \texttt{if } n <= 0 \\ T(n-1) + c_1 & \texttt{otherwise} \end{cases}$$

Unrolling the recurrence, we get $T(n) = \underbrace{c_1 + c_1 + \cdots + c_1}_{n \text{ times}} + c_0 = c_1 n + c_0 \in \mathcal{O}(n)$.

(b) Consider the function g

```
g(n) {
    if (n <= 1) {
        return 1000
    }
    if (g(n/3) > 5) {
        for (int i = 0; i < n; i++) {
            println("Yay!")
        }
        return 5 * g(n/3)
    }
    else {
        for (int i = 0; i < n * n; i++) {
            println("Yay!")
        }
        return 4 * g(n/3)
    }
}
```

Solution:

The following recurrent models the worst-case time complexity of $g(n)$.

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{3}\right) + c_1 n & \text{otherwise} \end{cases}$$

To unroll the recurrence, we can use the tree method. The recursion tree has height $\log_3(n)$, each non-leaf

level i has work $\frac{c_1 n 2^i}{3^i}$, and the leaf level has work $c_0 2^{\log_3(n)}$. Putting this together, we have:

$$\begin{aligned}
\sum_{i=0}^{\log_3(n)-1} \left(\frac{c_1 n 2^i}{3^i}\right) + c_0 2^{\log_3(n)} &= c_1 n \sum_{i=0}^{\log_3(n)-1} \left(\frac{2}{3}\right)^i + c_0 n^{\log_3(2)} \\
&= c_1 n \left(\frac{1 - \left(\frac{2}{3}\right)^{\log_3(n)}}{1 - \frac{2}{3}}\right) + c_0 n^{\log_3(2)} \qquad \text{By finite geometric series} \\
&= 3c_1 n \left(1 - \left(\frac{2}{3}\right)^{\log_3(n)}\right) + c_0 n^{\log_3(2)} \\
&= 3c_1 n \left(1 - \frac{n^{\log_3(2)}}{n}\right) + c_0 n^{\log_3(2)} \\
&= 3c_1 n - 3c_1 n^{\log_3(2)} + c_0 n^{\log_3(2)}
\end{aligned}$$

6. Unrolling Recurrences

Unroll the following recurrence equations. For these problems, you should show work and find a closed form, not a big-O bound.

(a) Unroll the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

Solution:

We can easily solve this recurrence with the expansion method.

$$T(n) = T(n-1) + 1 = (T(n-2) + 1) + 1 = (T(n-3) + 1) + 2 = T(n-k) + k$$

The above recurrence will be closed when $n - k = 1$, or when $k = n - 1$. Plugging this in we get

$$T(n) = T(1) + (n-1) = 1 + n - 1 = n$$

(b) Unroll the following recurrence

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + c_1 & \text{otherwise} \end{cases}$$

Solution:

The recursion tree has height $\lg(n)$, each non-leaf level i has has work $c_1 2^i$, and the leaf level has work $c_0 2^{\lg(n)}$. Putting this together, we have:

$$\left(\sum_{i=0}^{\lg n-1} c_1 2^i\right) + c_0 2^{\lg(n)} = c_1 \left(\sum_{i=0}^{\lg n-1} 2^i\right) + c_0 n = c_1 \frac{1-2^{\lg n-1+1}}{1-2} + c_0 n$$
$$= c_1 2^{\lg n} - c_1 + c_0 n$$
$$= c_1(n-1) + c_0 n$$
$$= (c_0 + c_1)n - c_1$$

(c) Unroll the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T(n/3) + 1 & \text{otherwise} \end{cases}$$

Solution:

We can solve this recurrence with the tree method. The recursion tree has a height of $log_3(n)$ since that is how many times we would expand $T(n/3)$. The number of nodes at a given height i is 2^i since at each level we multiply $T(n/3)$ by 2. The work done at a non-leaf node is 1. This gives us the sum

$$\sum_{i=0}^{log_3(n)-1} (2^i * 1) + 2^{log_3(n)} * 1$$

Where the left term describes the total work done at non-leaf nodes and the right determines work done at leaves. When we simplify, using the finite geometric series sum formula ($\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$)

$$= \frac{1-2^{log_3(n)}}{1-2} + 2^{log_3(n)}$$
$$= -1 * (1-2^{log_3(n)}) + 2^{log_3(n)}$$
$$= 2 * 2^{log_3(n)} - 1$$

Using the log rule $a^{log_b c} = c^{log_b a}$ we get

$$= 2 * n^{log_3(2)} - 1$$

7. Applying definitions

For each of the following, choose a c and n_0 which show $f(n) \in \mathcal{O}(g(n))$. Explain why your values of c and n_0 work.

Solution:

These solutions are divided into "scratch work" which is algebra you have to do before you start writing the proof and the "proof" itself. The scratch work technically doesn't belong in a final answer, but the proofs are difficult to understand without them.

For these, the proof will just be the scratch work algebra, possibly done in a different order, with some connecting words.

(a) $f(n) = 3n + 4, g(n) = 5n^2$

Solution:

scratch work: Our goal is to bound f by a function with n^2 terms so comparing to g is easier.

$$3n \leq 3n^2 = \frac{3}{5} \cdot 5n^2 \qquad \text{if } n \geq 1$$
$$4 \leq 4n^2 = \frac{4}{5} \cdot 5n^2 \qquad \text{if } n \geq 1$$

We add together the inequalities to get:

$$f(n) = 3n + 4 \leq \left(\frac{3}{5} + \frac{4}{5}\right) 5n^2 = \frac{7}{5}g(n)$$

proof: One possible solution is $c = \frac{7}{5}$ and $n_0 = 1$.

We note that $3n \leq 3n^2$ and $4 \leq 4n^2$ as long as $n \geq n_0$. Adding these two inequalities, we have $f(n) = 3n + 4 \leq 7n^2 = \frac{7}{5}g(n)$ is true for all $n \geq 1$.

Therefore, we know that $3n + 4 \leq c \cdot 5n^2$ is true for our chosen values of c and for all $n \geq n_0$.

(b) $f(n) = 33n^3 + \sqrt{n} - 6, g(n) = 17n^4$

Solution:

scratch work: Since g's dominating term is n^4, we will try to bound f by a function with only n^4 terms. Going term by term of f:

$33n^3 \leq 33n^4$ as long as muliplying by n increases the function (i.e. as long as $n \geq 1$).
$\sqrt{n} \leq n^4$ as long as $n \geq 1$.
$-6 \leq 0n^4$ (always).

Combining these we want to get: $33n^3 + \sqrt{n} - 6 \leq 33n^4 + n^4 \leq 34n^4 \leq c \cdot 17n^4$ c being 2 is enough.

proof: One possible solution is $c = 2$ and $n_0 = 1$.

We note that $33n^3 \leq 33n^4$, $\sqrt{n} \leq n^4$, and -6 $leq0n^4$ all hold for $n \geq n_0 = 1$.

Next, note that $34n^4 \leq c \cdot 17n^4$ is true for all values of n and when $c = 2$.

Therefore, we know that $33n^3 + \sqrt{n} - 6 \leq c \cdot 17n^4$ is true for our chosen value of c and for all $n \geq n_0$.

(c) $f(n) = 17\log(n), g(n) = 32n + 2n\log(n)$

Solution:

scratch work: There are a lot of ways to do this one. Normally we would compare to the highest order term in g, but because the constant is larger on the n term, it will be easier to compare to that.

$17\log(n) \leq 17n$ as long as $\log(n) < n$. $\log(n) < n$ whenever $n > 2$. Then we can compare immediately to g: $17\log(n) \leq 17n \leq 32n \leq 32n + 2n\log(n) \leq c(32n + 2n\log(n))$ where it's good enough to set c to 1

proof: One possible solution is $c = 1$ and $n_0 = 2$.

We can convince ourselves this is true by examining our inequalities: $17\log(n) \leq 17n \leq 1 \cdot 32n$ for $n \geq n_0$. Since $2n\log(n)$ is always positve, we have So, since $17\log(n) \leq c \cdot (32 + 2n\log(n))$ is true for our chosen values of c and n_0, we know that $f(n) \in \mathcal{O}(2n\log(n))$.

Binary Search Tree

8. Binary Search Tree (BST) Practice Problems

Consider the definition for a Binary Tree Node as below:

```
public class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    TreeNode(int n) {
        value = n;
    }
}
```

(a) Search in BST: Given the root node of a BST and a value, write an efficient method $searchBST$ to find the node in the BST that the node's value is the same as the given value. Return true if such node exist, else false. Lastly, provide the complexity analysis of your solution (Time complexity, Space complexity).

```
public boolean searchBST(TreeNode root, int val) {
    // begin your code
}
```

Solution:

Possible Approach: Iterative Solution

```
public boolean searchBST(TreeNode root, int val) {
    while (root != null && val != root.val) {
        root = val < root.val ? root.left : root.right;
    }
    return root != null;
}
```

Complexity Analysis:

Time Complexity: $O(logN)$ in the average case, $O(N)$ in the worst case, where n is number of nodes.

Space Complexity: $O(1)$ since it's a constant space solution.

(b) Validate BST: Given a binary tree, write an efficient method $isValidBST$ to determine if it is a valid binary search tree (BST). Lastly, provide the complexity analysis of your solution (Time complexity, Space complexity).

```
public boolean isValidBST(TreeNode root) {
    // begin your code
}
```

Solution:

Possible Approach: Recrusive Solution

```
public boolean isValidBST(TreeNode root) {
    return helper(root, null, null);
}

public boolean helper(TreeNode node, Integer lower, Integer upper) {
```

```
        if (node == null) {
            return true;
        }
        int val = node.val;
        if (lower != null && val <= lower) {
            return false;
        } else if (upper != null && val >= upper) {
            return false;
        } else if (! helper(node.right, val, upper)) {
            return false;
        } else if (! helper(node.left, lower, val)) {
            return false;
        } else {
            return true;
        }
    }
```

Complexity Analysis:

Time Complexity: $O(N)$ since we visit each node exactly once, where n is number of nodes.

Space Complexity: $O(N)$ since we keep up to the entire tree, where n is number of nodes.

(c) Inorder Traversal in BST: Given a binary tree, write an efficient method $inorderTraversalBST$ to return the inorder traversal of its nodes' values. Lastly, provide the complexity analysis of your solution (Time complexity, Space complexity).

```
public List<Integer> inorderTraversalBST(TreeNode root) {
    // begin your code
}
```

Solution:

Possible Approach: Recrusive Solution

```
public List <Integer> inorderTraversalBST(TreeNode root) {
    List < Integer > res = new ArrayList < > ();
    helper(root, res);
    return res;
}

public void helper(TreeNode root, List < Integer > res) {
    if (root != null) {
        if (root.left != null) {
            helper(root.left, res);
        }
        res.add(root.val);
        if (root.right != null) {
            helper(root.right, res);
        }
    }
}
```

Complexity Analysis:

Time Complexity: $O(N)$, where n is number of nodes. Note that the recursive function is $T(N) = 2 * T(N/2) + 1$.

Space Complexity: $O(N)$ in the worst case, and $O(logN)$ in average case, where n is number of nodes.