Section 03: Solutions

1. Analysis

For each of the following code blocks, what is the worst-case runtime? Give a big-$\Theta$ bound.

(a) $\text{public IList<String> repeat(DoubleLinkedList<String> list, int n) \{}$
    
    $\text{IList<String> result = new DoubleLinkedList<String>();}$
    
    $\text{for(String str : list) \{}$
    
    $\text{\quad for(int i = 0; i < n; i++) \{}$
    
    $\text{\quad \quad result.add(str);}$
    
    $\text{\quad \}}$  
    
    $\text{\quad return result; \}}$  

$\text{\textbf{Solution:}}$

The runtime is $\Theta(nm)$, where $m$ is the length of the input list and $n$ is equal to the int $n$ parameter.

One thing to note here is that unlike many of the methods we've analyzed before, we can't quite describe the runtime of this algorithm using just a single variable: we need two, one for each loop.

The other thing to remember is that in Java, foreach loops are converted into a while loop using iterators, which will influence the final runtime of our algorithm.

(b) $\text{public void foo(int n) \{}$
    
    $\text{\quad for (int i = 0; i < n; i++) \{}$
    
    $\text{\quad \quad for (int j = 5; j < i; j++) \{}$
    
    $\text{\quad \quad \quad System.out.println("Hello!");}$
    
    $\text{\quad \quad \}}$  
    
    $\text{\quad \quad for (int j = i; j >= 0; j -= 2) \{}$
    
    $\text{\quad \quad \quad System.out.println("Hello!");}$
    
    $\text{\quad \quad \}}$  
    
    $\text{\quad \}}$  

$\text{\textbf{Solution:}}$

$\Theta(n^2)$.

(c) $\text{public int num(int n)\{}$
    
    $\text{\quad if (n < 10) \{}$
    
    $\text{\quad \quad return n; \}}$  
    
    $\text{\quad else if (n < 1000) \{}$
    
    $\text{\quad \quad return num(n - 2); \}}$  
    
    $\text{\quad else \{}$
    
    $\text{\quad \quad return num(n / 2); \}}$  
    
$\text{\}}$  

$\text{\textbf{Solution:}}$
The answer is $\Theta(\log(n))$.

One thing to note is that the second case effectively has no impact on the runtime. That second case occurs only for $n < 1000$ – when discussing asymptotic analysis, we only care what happens with the runtime as $n$ grows large.

(d) \[
\text{public int foo(int n) \{}
\hspace{1cm} \text{if (n <= 0) \{}
\hspace{1cm} \hspace{1cm} \text{return 3;}
\hspace{1cm} \} \hspace{1cm}
\hspace{1cm} \text{int x = foo(n - 1);}
\hspace{1cm} \text{System.out.println("hello");}
\hspace{1cm} \text{x += foo(n - 1);}
\hspace{1cm} \text{return x;}
\hspace{1cm} \} \]

Solution:

The answer is $\Theta(2^n)$.

In order to determine that this is exponential, let's start by considering the following recurrence:

$$T(n) = \begin{cases} 
1 & \text{if } n = 0 \\
2T(n-1) + 1 & \text{Otherwise}
\end{cases}$$

While we could unfold this to get an exact closed form, we can approximate the final asymptotic behavior by taking a step back and thinking on a higher level what this is doing.

Basically, what happens is we take the work done by $T(n-1)$ and multiply it by 2. If we ignore the +1 constant work done in the recursive case, the net effect is that we multiply 2 approximately $n$ times. This simplifies to $2^n$.

2. Recurrences

For each of the following recurrences, use the unfolding method to first convert the recurrence into a summation. Then, find a big-$\Theta$ bound on the function in terms of $n$. Assume all division operations are integer division.

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

Solution:

The summation is $1 + \sum_{i=2}^{\log(n)+1} 3$. The big-$\Theta$ bound is $\Theta(\log(n))$.

Something you may notice is that depending on what exactly $n$ is, the expression $\log(n) + 1$ may not evaluate to an integer. In that case, what does it mean to have $\log(n) + 1$ as the upper limit of a summation?

What exactly this mean differs based on convention, but for the purposes of this class, we'll assume that $i$ varies starting at 2 up to the largest possible integer that is $\leq \log(n) + 1$. We could write this more explicitly using floors: $1 + \sum_{i=2}^{\lfloor \log(n)+1 \rfloor} 3$. 

2
(b) \( T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + 2 & \text{otherwise} \end{cases} \)

**Solution:**

The summation is \( 1 + \sum_{i=1}^{n} 2 \). The big-\( \Theta \) bound is \( \Theta (n) \).

(c) \( T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n/3) + 4 & \text{otherwise} \end{cases} \)

**Solution:**

The summation is \( 1 + \sum_{i=1}^{\log_3(n)+1} 4 \). The big-\( \Theta \) bound is \( \Theta (n) \).

(d) \( T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2T(n/3) + n & \text{otherwise} \end{cases} \)

**Solution:**

In order to determine what this expression looks like as a summation, it helps to first partially unroll it:

\[
T(n) = n + 2T\left(\frac{n}{3}\right)
\]
\[
= n + 2\left( \frac{n}{3} + 2T\left( \frac{n}{9} \right) \right)
\]
\[
= n + 2\left( \frac{n}{3} + 2 \left( \frac{n}{9} + 2T\left( \frac{n}{27} \right) \right) \right)
\]
\[
= n + 2\left( \frac{n}{3} + 2 \left( \frac{n}{9} + 2 \left( \frac{n}{27} + 2T\left( \frac{n}{81} \right) \right) \right) \right)
\]

We then multiply in the 2 on the outside:

\[
T(n) = n + 2\left( \frac{n}{3} + 2 \left( \frac{n}{9} + 2 \left( \frac{n}{27} + 2T\left( \frac{n}{81} \right) \right) \right) \right)
\]
\[
= n + 2\left( \frac{n}{3} + 2^2 \left( \frac{n}{9} + 2 \left( \frac{n}{27} + 2T\left( \frac{n}{81} \right) \right) \right) \right)
\]
\[
= n + 2\left( \frac{n}{3} + 2^2 \frac{n}{9} + 2^3 \left( \frac{n}{27} + 2T\left( \frac{n}{81} \right) \right) \right)
\]
\[
= n + 2\left( \frac{n}{3} + 2^2 \frac{n}{9} + 2^3 \frac{n}{27} + 2^4 T\left( \frac{n}{81} \right) \right)
\]

We can start to see the pattern now: our summation is roughly of the form \( \sum_{i=0}^{?} 2^i \frac{n}{3^i} \).

What about the base case? It’s not just 1, we need to multiply it by some power of 2 to account for the accumulating multiples.

We put all the pieces together and finish: \( 1 \cdot 2^{\log_3(n)+1} + \sum_{i=0}^{\log_3(n)} \frac{2^i}{3^i} n \)

To compute the \( \Theta \) bound, we observe that the large constant, despite being large, is still ultimately a
constant. We can also simplify the summation by pulling out the $n$ (since it doesn’t vary on $i$). The remaining summation must simplify to some integer. So, we conclude $\Theta (n)$.

\[(e)\quad T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n - 1) + 1 & \text{otherwise} 
\end{cases} \]

Solution:

Using a similar process, we get the following expression: $2^{n-1} + \sum_{i=2}^{n} 2^{i-2}$.

This ends up being in $\Theta (2^n)$.

\[(f)\quad T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + 100 & \text{otherwise} 
\end{cases} \]

Solution:

We first get $2^{\log(n)}$ + $\sum_{i=2}^{\log(n)+1} 100 \cdot 2^{i-2}$.

Therefore, we have $\Theta (\log(n))$.

3. Modeling recursive functions

Consider the following method.

```java
public static int f(int n) {
    if (n == 0) {
        return 0;
    }

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

    return 5 * f(n / 2) + 3 * result + 2 * f(n / 2);
}
```

(a) Find a recurrence $T(n)$ modeling the worst-case runtime of $f(n)$.

Solution:

$T(n) = \begin{cases} 
1 & \text{When } n = 0 \\
\frac{n(n-1)}{2} + 2T(n/2) & \text{Otherwise} 
\end{cases} \quad \text{When } n = 0 \quad \text{Otherwise}$

(b) Find a recurrence $W(n)$ modeling the integer output of $f(n)$. 


4. Modeling recursive functions 2

```java
public static int g(n) {
    if (n <= 1) {
        return 1000;
    }
    if (g(n / 3) > 5) {
        for (int i = 0; i < n; i++) {
            System.out.println("Hello");
        }
        return 5 * g(n / 3);
    } else {
        for (int i = 0; i < n * n; i++) {
            System.out.println("World");
        }
        return 4 * g(n / 3);
    }
}
```

(a) Find a recurrence \( S(n) \) modeling the worst-case runtime of \( g(n) \).

**Solution:**

\[
S(n) = \begin{cases} 
1 & \text{When } n \leq 1 \\
2S(n/3) + n & \text{Otherwise}
\end{cases}
\]

Important: note that the if statement contains a recursive call that must be evaluated for \( n > 1 \).

(b) Find a recurrence \( X(n) \) modeling the integer output of \( g(n) \).

**Solution:**

\[
X(n) = \begin{cases} 
1000 & \text{When } n \leq 1 \\
5T(n/3) & \text{Otherwise}
\end{cases}
\]
5. Modeling recursive functions 3

Consider the following set of recursive methods.

```java
public int test(int n) {
    IDictionary<Integer, Integer> dict = new AvlDictionary<>();
    populate(n, dict);
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += dict.get(i);
    }
    return counter;
}

private void populate(int k, IDictionary<Integer, Integer> dict) {
    if (k == 0) {
        dict.put(0, k);
    } else {
        for (int i = 0; i < k; i++) {
            dict.put(i, i);
        }
        populate(k / 2, dict);
    }
}
```

(a) Write a mathematical function representing the worst-case runtime of `test`.

You should write two functions, one for the runtime of `test` and one for the runtime of `populate`.

**Solution:**

The runtime of the `populate` method is:

\[
P(k) = \begin{cases} 
\log(N) & \text{When } k = 0 \\
 k \log(k) + P(k/2) & \text{Otherwise} 
\end{cases}
\]

Here, \( N \) is the maximum possible value of \( n \)

The runtime of the `test` method is then \( R(n) = P(n) + n \).

(b) Write a mathematical function representing the integer output of `test`.

**Solution:**

\[
Y(n) = \frac{n(n-1)}{2}
\]
6. **AVL Trees**

(a) Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

\{13, 17, 14, 19, 22, 18, 11, 10, 21\}

**Solution:**

(b) Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

\{1, 2, 3, 4, 5, 6\}

**Solution:**

7. **More AVL Trees**

(a) Is this a valid AVL tree? Explain your answer.

**Solution:**
(b) Is this a valid AVL tree? Explain your answer.

Solution:

No, does not meet the balance property.

(c) Is this a valid AVL tree? Explain your answer.

Solution:

Yes, it satisfies the balance and BST properties.
8. Algorithm Design

(a) Given a binary search tree, describe how you could convert it into an AVL tree with worst-case time $O(n \log(n))$. What is the best case runtime of your algorithm?

Solution:

Since we already have a BST, we can do an in-order traversal on the tree to get a sorted array of nodes. We could now simply insert all of these nodes back into an AVL tree using rotations which would give us an $O(n \log(n))$ runtime.

(b) Given an AVL tree, describe how would you do a level order tree traversal. What is the worst-case runtime of your algorithm?

Solution:

Since an AVL tree is just a balanced BST, we can use a queue to add each node we visit. As we dequeue each node, we will add it’s children to the queue. We would get an $O(n)$ runtime.