Section 03: Solutions

Section Problems

1. Binary Search Trees

(a) Write a method `validate` to validate a BST containing no duplicates. Although the basic algorithm can be converted to any data structure and work in any format, if it helps, you may write this method for the `IntTree` class:

```
public class IntTree {
    private IntTreeNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;

        // constructors omitted for clarity
    }
}
```

Solution:

```
public boolean validate() {
    return validate(overallRoot, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean validate(IntTreeNode root, int min, int max) {
    if (root == null) {
        return true;
    } else if (root.data > max || root.data < min) {
        return false;
    } else {
        return validate(root.left, min, root.data) &&
            validate (root.right, root.data, max);
    }
}
```

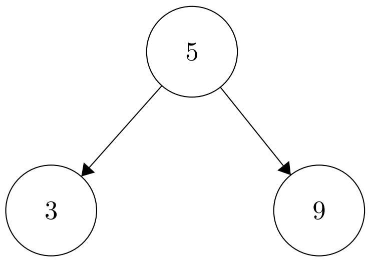
(b) Suppose we want to implement a method `findNode(V value)` that searches a binary search tree with n unique nodes for a given value.

(i) We want to analyze the runtime of our `findNode` method in the best possible case and the worst possible case. What does our tree look like in the best possible case? In the worst possible case? *Draw* two examples of binary search trees with up to 5 nodes each that would result in the best-case and worst-case runtimes of `findNode`.

Solution:

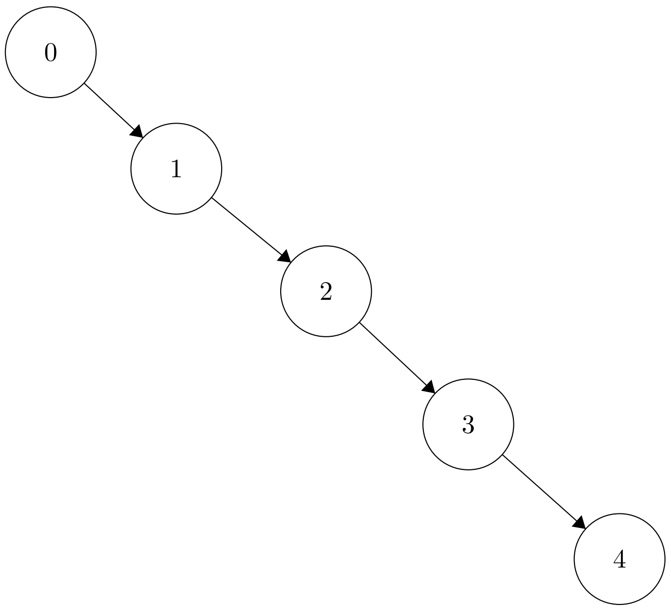
Our best-case input would be a tree with the value we're looking for at the root.

Example: `findNode(5)`:



Our worst-case input would be a completely unbalanced tree with the value we're looking for at the very bottom.

Example `findNode(4)`:



(ii) What is the worst case big-Θ runtime for `findNode`?

Solution:

Our worst-case tree is completely unbalanced (resembling a linked list) with the value we're looking for at the bottom. Since `findNode` would have to search through every node, the answer is $\Theta(n)$.

(iii) What is the best case big-Θ runtime for `findNode`?

Solution:

The answer is $\Theta(1)$, because the node we're searching for could be at the root.

2. TreeMap implemented as a Binary Search Tree

Consider the following method, which is a part of a Binary Search Tree implementation of a TreeMap class.

```java
public V find(K key) {
    return find(this.root, key);
}

private V find(Node<K, V> current, K key) {
    if (current == null) {
        return null;
    }
    if (current.key.equals(key)) {
        return current.value;
    }
    if (current.key.compareTo(key) > 0) {
        return find(current.left, key);
    } else {
        return find(current.right, key);
    }
}
```

(a) We want to analyze the runtime of our `find(x)` method in the best possible case and the worst possible case. What does our tree look like in the best possible case? In the worst possible case?

Solution:

In the best possible case, our tree will be completely balanced. In the worst possible case, it will be completely unbalanced, resembling a linked list.

(b) Write a recurrence to represent the worst-case runtime for `find(x)` in terms of n, the number of elements contained within our tree. Then, provide a Θ bound.

Solution:

The recurrence representing the worst-case runtime of `find(x)` is:

$$T_w(n) = \begin{cases} 1 & \text{when } n = 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases}$$

That is, every time we recurse, we are able to eliminate only one node from the span of possibilities we must consider. This is possible in case the tree is absolutely unbalanced (think of a tree that looks like a linked list).

This recurrence is in $\Theta(n)$.

(c) Assuming we have an optimally structured tree, write a recurrence for the runtime of `find(x)` (again in terms of n). Then, provide a Θ bound.

Solution:

The recurrence representing the best-case runtime of `find(x)` is:

$$T_w(n) = \begin{cases} 1 & \text{when } n = 0 \\ 1 + T(n/2) & \text{otherwise} \end{cases}$$

That is, every time we recurse, we are able to eliminate about half of the nodes we must consider.

This recurrence is in $\Theta(\log(n))$.

3. Code Analysis

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

(a)
```
public List<String> repeat(List<String> list, int n) {
    List<String> result = new LinkedList<String>();
    for(String str : list) {
        for(int i = 0; i < n; i++) {
            result.add(str);
        }
    }
    return result;
}
```

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.

(b)
```
public int num(int n){
    if (n < 10) {
        return n;
    } else if (n < 1000) {
        return num(n - 2);
    } else {
        return num(n / 2);
    }
}
```

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.

(c)
```java
public int foo(int n) {
    if (n <= 0) {
        return 3;
    }
    int x = foo(n - 1);
    System.out.println("hello");
    x += foo(n - 1);
    return x;
}
```

Solution:

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

If we visualized a tree that counted the number of calls to foo, we would see that exactly $2^{n+1} - 1$ calls are made. This is very similar to the example we saw in lecture 5 (https://courses.cs.washington.edu/courses/cse373/19au/lectures/05/)

(d)
```java
public boolean isPrime(int n) {
    int toTest = 2;
    while (toTest < n) {
        if (n % toTest == 0) {
            return false;
        } else {
            toTest++;
        }
     }
     return true;
}
```

Solution:

There is no big-Θ bound for this function. Note that as n grows very large, this function will occasionally run a single iteration. Remember that we only care about the worst case **as n grows larger and larger**.

Another way to think about this: if we thought about this as a graph, with n on the x axis and the total number of operations on the y axis, we would see that as n grows larger the y-axis value would be fluctuating. If we tried to come up with a function that lower-bounded this function as n goes to infinity, the only lower bound we would be able to find would be $\Omega(1)$. Similarly, the tightest upper bound we could find would be $\mathcal{O}(n)$. Because the big-Ω and big-O bounds do not match up in the worst case, there is no big-Θ bound.

4. Tree method walk-through

Consider the following recurrence: $A(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 3A(n/6) + n & \text{otherwise} \end{cases}$

We want to find an *exact* closed form of this equation by using the tree method. Suppose we draw out the total work done by this method as a tree, as discussed in lecture. Let n be the initial input to A.

(a) What is the size of the input at level i (as in class, call the root level 0)?

Solution:

We divide by 6 at each level, so the input size is $n/6^i$.

(b) What is the number of nodes at level i?

Note: let $i = 0$ indicate the level corresponding to the root node. So, when $i = 0$, your expression should be equal to 1.

Solution:

Each (non-base-case) node produces 3 more nodes, so at level i we have 3^i nodes.

(c) What is the total work at the i^{th} **recursive** level?

Solution:

Combining our last two parts: $3^i \cdot \frac{n}{6^i} = \frac{n}{2^i}$

(d) What is the last level of the tree?

Solution:

We hit our base case when $n/6^i = 1$, which is at level $i = \log_6(n)$.

(e) What is the work done in the base case?

Solution:

From previous parts, there are $3^{\log_6(n)}$ nodes at that level, from the recurrence each does 1 unit of work, so we get $1 \cdot 3^{\log_6(n)}$ work.

(f) Combine your answers from previous parts to get an expression for the total work.

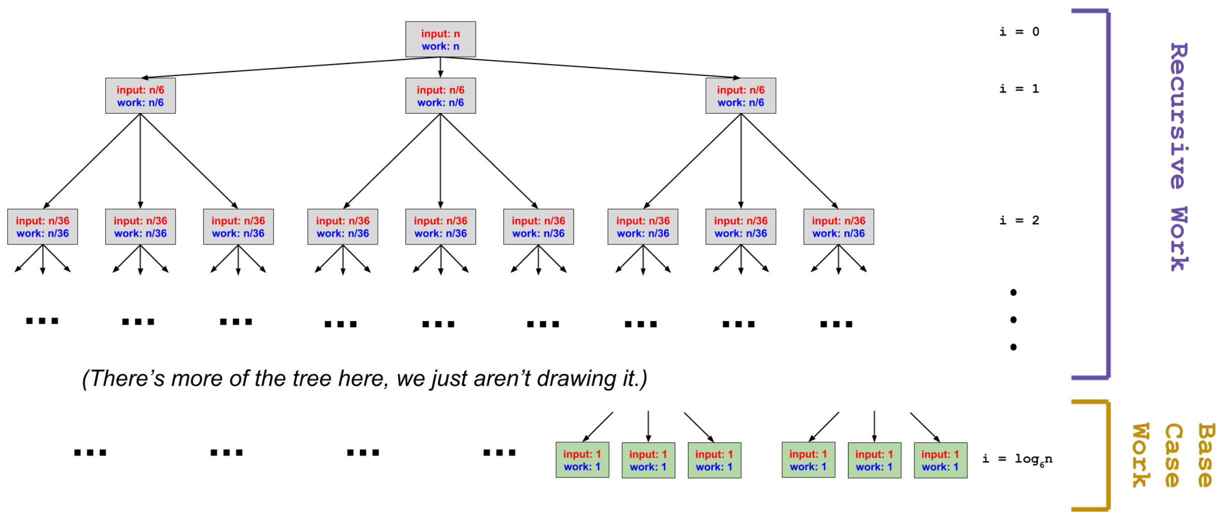
Solution:

We know the expression for the work done at each recursive level and the base case level. Combining these we have:

$$\sum_{i=0}^{\log_6(n)-1} \frac{n}{2^i} + 3^{\log_6(n)}$$

Side note: Yes, it says $\log_6(n) - 1$ and not $\log_6(n)$. This is because the last level ($\log_6(n)$) is the base case level and has a different formula.

Here's a drawing of of the tree. Note that we distinguish the **input** to a node (red) and the **work** done by a node (blue) since these can be different when we're applying the tree method.



(g) Simplify to a closed form.

Note: you do not need to simplify your answer, once you found the closed form. Hint: You should use the finite geometric series identity somewhere while finding a closed form.

Solution:

We combine all the pieces and simplify:

$$\begin{aligned} A(n) &= \sum_{i=0}^{\log_6(n)-1} \frac{n}{2^i} + 3^{\log_6(n)} \\ &= n \sum_{i=0}^{\log_6(n)-1} \left(\frac{1}{2}\right)^i + 3^{\log_6(n)} \end{aligned}$$

We'll apply two identities: to the summation we apply the finite geometric series identity; and to the base-case work, we apply the power of a log identity. We get:

$$A(n) = n \cdot \frac{\left(\frac{1}{2}\right)^{\log_6(n)} - 1}{\frac{1}{2} - 1} + n^{\log_6(3)}$$

You don't have to simplify further, but if you were to simplify, you would get:

$$\begin{aligned} A(n) &= n \cdot \frac{\left(\frac{1}{2}\right)^{\log_6(n)} - 1}{1/2 - 1} + n^{\log_6(3)} \\ &= -2n \left(n^{\log_6(1/2)} - 1\right) + n^{\log_6(3)} \\ &= -2n \left(n^{\log_6(3/6)} - 1\right) + n^{\log_6(3)} \\ &= -2n \left(n^{\log_6(3)-\log_6(6)} - 1\right) + n^{\log_6(3)} \\ &= -2n \left(\frac{n^{\log_6(3)}}{n} - 1\right) + n^{\log_6(3)} \\ &= -2n^{\log_6(3)} + 2n + n^{\log_6(3)} \\ &= 2n - n^{\log_6(3)} \end{aligned}$$

5. More tree method recurrences

For each of the following recurrences, find their closed form using the tree method. It may be a useful guide to use the steps from section 4 of this handout to help you with all the parts of solving a recurrence problem fully.

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

Solution:

Given this recurrence, we know...

- Input size at level i: $n/2^i$
- Number of nodes on level i: $1^i = 1$
- Total work done per level: $1 \cdot 3 = 3$
- Last level of the tree: $\log_2(n)$
- Total work done in base case: $1 \cdot 1^{\log_3(n)} = 1$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 3\right) + 1$$

Using the summation of a constant identity, we get:

$$3\log_2(n) + 1$$

This simplifies to $T(n) \in \Theta(\log(n))$.

(b) $S(q) = \begin{cases} 1 & \text{if } q = 1 \\ 2S(q-1) + 1 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Size of input at level i is $q - i$
- Number of nodes on level i: 2^i
- Total work done at (recursive) level i: $2^i \cdot 1$
- Last level of the tree: $q - 1$
- Total work done in base case: $1 \cdot 2^{q-1}$

Note that these expressions look a little different from the ones we've seen up above. This is because we aren't *dividing* our terms by some constant factor – instead, we're *subtracting* them.

So we get the expression:

$$\left(\sum_{i=0}^{q-1-1} 2^i\right) + 2^{q-1}$$

We apply the finite geometric series to get:

$$\frac{2^{q-1} - 1}{2 - 1} + 2^{q-1}$$

If we wanted to simplify, we'd get:

$$2^q - 1$$

This simplifies to $S(q) \in \Theta\left(2^q\right)$.

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + 4n^2 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Size of input at level i is $n/2^i$
- Number of nodes on level i: 8^i
- Total work done per (recursive) level: $8^i \cdot 4\left(\frac{n}{2^i}\right)^2 = 8^i \cdot 4 \cdot \frac{n^2}{4^i}$
- Last level of the tree: When $n/2^i = 1$, i.e. $\log_2(n)$
- Total work done in base case: $1 \cdot 8^{\log_2(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 8^i \cdot 4 \cdot \frac{n^2}{4^i}\right) + 8^{\log_2(n)}$$

We can simplify by pulling the $4n^2$ out of the summation:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} \frac{8^i}{4^i}\right) + 8^{\log_2(n)}$$

This further simplifies to:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} 2^i\right) + 8^{\log_2(n)}$$

After applying the finite geometric series identity, we get:

$$4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2(n)}$$

This is a closed form so we could stop, but if we want a tidy solution, we can continue simplifying:

$$\begin{aligned} T(n) &= 4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2(n)} \\ &= 4n^2 \cdot \left(2^{\log_2(n)} - 1\right) + 8^{\log_2(n)} \\ &= 4n^2 \cdot \left(n^{\log_2(2)} - 1\right) + n^{\log_2(8)} \\ &= 4n^2 \cdot (n - 1) + n^3 \\ &= 5n^3 - 4n^2 \end{aligned}$$

This simplifies to $T(n) \in \Theta\left(n^3\right)$.

Useful summation identities

Splitting a sum

$$\sum_{i=a}^{b}(x+y) = \sum_{i=a}^{b} x + \sum_{i=a}^{b} y$$

Adjusting summation bounds

$$\sum_{i=a}^{b} f(x) = \sum_{i=0}^{b} f(x) - \sum_{i=0}^{a-1} f(x)$$

Factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

Summation of a constant

$$\sum_{i=0}^{n-1} c = \underbrace{c + c + \ldots + c}_{n \text{ times}} = cn$$

Note: this rule is a special case of the rule on the left

Gauss's identity

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

Sum of squares

$$\sum_{i=0}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$$

Finite geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Infinite geometric series

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

Note: applicable only when $-1 < x < 1$

Useful logarithm identities

Note: we assume here that in all cases, variables are non-zero.

Log of a product

$$\log_b(x \cdot y) = \log_b(x) + \log_b(y)$$

Log of a fraction

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$$

Log of a power

$$\log_b(x^y) = y \cdot \log_b(x)$$

Power of a log

$$x^{\log_b(y)} = y^{\log_b(x)}$$

Change of base

$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$$