CS 61B Spring 2019

Asymptotics II, Search Trees

Discussion 7: March 4, 2019

Here's a review of some formulas you will find useful when doing asymptotic analysis.

- $\sum_{i=1}^{N} i = 1 + 2 + 3 + 4 + \cdots + N = \frac{N(N+1)}{2} = \frac{N^2+N}{2} \in \mathbf{\Theta(N^2)}$
- $\sum_{i=0}^{N} 2^i = 1 + 2 + 4 + 8 + \cdots + 2^N = 2 \cdot 2^N - 1 \in \mathbf{\Theta(2^N)}$
- $N + \frac{N}{2} + \ldots + 2 + 1 = 2N - 1 \in \mathbf{\Theta(N)}$

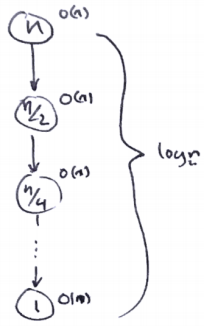
1 Space Jam 2

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

1.1 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  }
```

andslam(N) runs in time $\Theta(N)$ worst and best case. In both cases the runtime is $\sum_{i=0}^{\log(n)} \frac{N}{2^i} \leq N \sum_{i=0}^{\log(\infty)} \frac{1}{2^i} = 2N$. One potentially tricky portion is that $\sum_{i=0}^{\log(n)} \frac{1}{2^i}$ 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.

$$\text{running time} = \underbrace{\#\text{ layers}}_{\text{sum over}} \cdot \left(\frac{\#\text{ nodes}}{\#\text{ layers}}\right) \cdot \left(\frac{\text{amount work}}{\$1 \text{ node}}\right)$$

$$= \sum_{i=0}^{\log n} (1) \cdot \left(\frac{n}{2^i}\right)$$

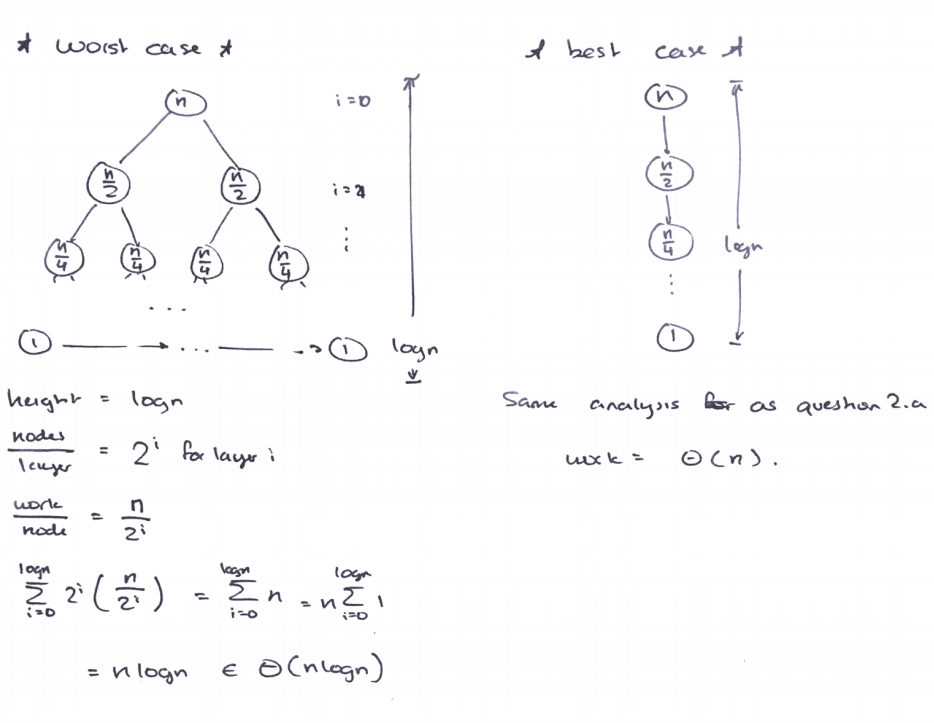
($i=0$ to $\log n$: layers; (1): $\#\text{nodes}/\text{layer}$; $\left(\frac{n}{2^i}\right)$: work / node)

$$= \sum_{i=0}^{\log n} \frac{n}{2^i} = n \sum_{i=0}^{\log n} \frac{1}{2^i} \leq 2n \in \Theta(n)$$

1.2 Give the running time for `andwelcome(0, N)` in terms of N.

```
1  public static void andwelcome(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(low, low + (high - low) / 2);
11         } else {
12             andwelcome(low, low + (high - low) / 2);
13             andwelcome(low + (high - low) / 2, high);
14         }
15     }
16 }
```

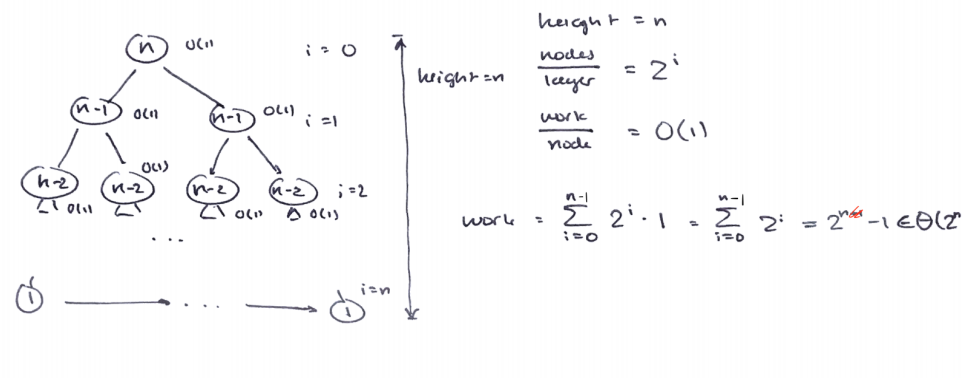
andwelcome(0, N) runs in time $\Theta(NlogN)$ worst case and $\Theta(N)$ best case. The recurrence relation is different for each case. In the worst case $coin > 0.5$ every time, resulting in a branching factor of 2. Because there is a branching factor of 2, there are 2^i nodes in the i-th layer. Meanwhile, 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



1.3 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  }
```

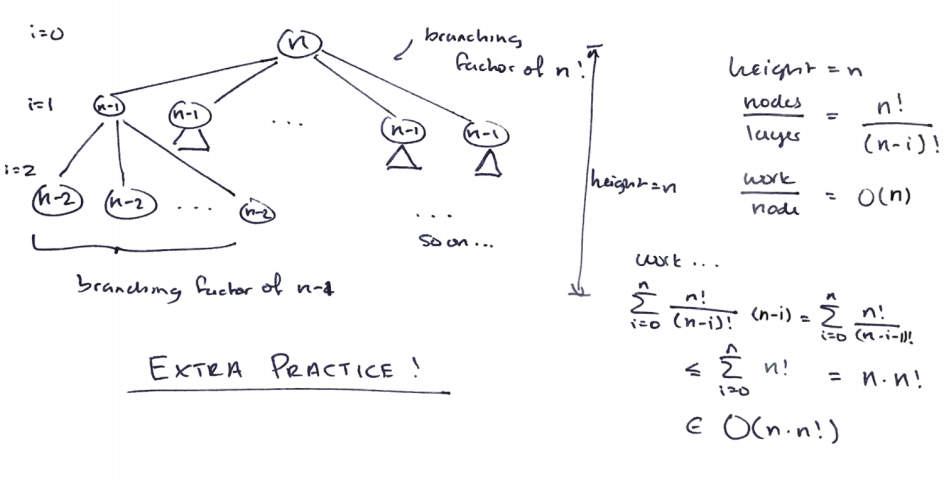
For `tothe(N)`, the worst and best case are $\Theta(2^N)$. Notice that at the i-th layer, there are 2^i nodes. Each node does a constant amount of work so with the fact that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$, we can derive the following.



1.4 *Extra:* Give the running time in terms of N. An $\mathcal{O}$-bound is sufficient.

```
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  }
```

For `spacejam(N)` the worst and best case is $\mathcal{O}(N \cdot N!)$. Now for the i-th layer, the number of nodes is $n \cdot (n-1) \cdot \ldots \cdot (n-i)$ 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-$\mathcal{O}$ bound instead of big-Θ.



2 Is This a BST?

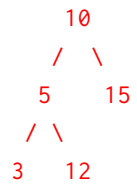
The following method `buggyIsBST` is supposed check if a given binary tree is a BST, though for some binary trees, it is returning the wrong answer. Think about an example of a binary tree for which `buggyIsBST` fails. Then, write `isBST` so that it returns the correct answer for any binary tree. The `TreeNode` class is defined as follows:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
}
```

Hint: You will find `Integer.MIN_VALUE` and `Integer.MAX_VALUE` helpful when writing `isBST`.

```
public static boolean buggyIsBST(TreeNode T) {
    if (T == null) {
        return true;
    } else if (T.left != null && T.left.val > T.val) {
        return false;
    } else if (T.right != null && T.right.val < T.val) {
        return false;
    } else {
        return buggyIsBST(T.left) && buggyIsBST(T.right);
    }
}

public static boolean isBST(TreeNode T) {
    return isBSTHelper(                                          );
}

public static boolean isBSTHelper(                                    ) {









}
```

An example of a binary tree for which the method fails:



The method fails for some binary trees that are not BSTs since it only checks that the value at a node is greater than its left child and less than its right child, not that its value is greater than every node in the left subtree and less than every node in the right subtree. Above is one example of a tree for which it fails.

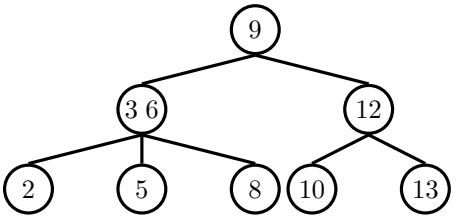
By the way, the method does return true for every binary tree that actually is a BST.

Below is the correct code:

```
public static boolean isBST(TreeNode T) {
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(TreeNode T, int min, int max) {
    if (T == null) {
        return true;
    } else if (T.val < min || T.val > max) {
        return false;
    } else {
        return isBSTHelper(T.left, min, T.val) && isBSTHelper(T.right, T.val, max);
    }
}
```

3 … as all Trees Should be

3.1 Consider the 2-3 tree below. What order should we insert these numbers so that we get the tree shown? There may be multiple correct answers.



2,3,5,9,10,12,13,6,8 is one such ordering. Noticing this tree is *almost* a BST is useful. Our last insertions should be 6,8, preceded by the insertions that yielded the BST before these insertions. This can be done by working backwards, and recursively "unsplitting" each level to find a candidate leaf-level insertion that cause the tree to split.

What is the **minimum** number of insertions that one can make to the above tree to cause the root to split? Assume we insert no duplicate items.

We can cause the root to split by inserting 0,1,14,15,16,17

Extra: What is the **maximum** number of insertions one can make to the above tree **without** splitting the root? Assume we insert no duplicate items.

We actually have room here to entirely stuff the height 3 tree (which has at most 26 elements), meaning we can insert 26 - 9 = 17 more elements. In fact, if we insert the elements 1,4,7,11,18,19,21,22,15,16,14,17,20,24,26,23,25 we get the maximally-stuffed height 3 tree. That looks like this: