CS 61B
Fall 2025

Asymptotics II and BSTs

Exam-Level Discussion 06: October 06, 2025

1 Re-cursed with Asymptotics

To help visualize the solutions better, a video walkthrough of this problem is linked here!

(a) What is the runtime of the code below in terms of `n`?

```
public static int curse(int n) {
    if (n <= 0) {
        return 0;
    } else {
        return n + curse(n - 1);
    }
}
```

$\Theta(n)$. On each recursive call, we do a constant amount of work. We make n recursive calls, because we go from n to 1. Then n recursive layers with 1 work at each layer is overall $\Theta(n)$ work.

(b) Can you find a runtime bound for the code below? We can assume the `System.arraycopy` method takes $\Theta(N)$ time, where N is the number of elements copied. The official signature is `System.arrayCopy(Object sourceArr, int srcPos, Object dest, int destPos, int length)`. Here, `srcPos` and `destPos` are the starting points in the source and destination arrays to start copying and pasting in, respectively, and `length` is the number of elements copied.

```
public static void silly(int[] arr) {
    if (arr.length <= 1) {
        System.out.println("You won!");
        return;
    }

    int newLen = arr.length / 2;
    int[] firstHalf = new int[newLen];
    int[] secondHalf = new int[newLen];

    System.arraycopy(arr, 0, firstHalf, 0, newLen);
    System.arraycopy(arr, newLen, secondHalf, 0, newLen);

    silly(firstHalf);
    silly(secondHalf);
}
```

Solution: $\Theta(N \log(N))$
At each level, we do N work, because the call to `System.arraycopy`. You can see that at the top level, this is N work. At the next level, we make two calls that each operate on arrays of length N/2, but that total work sums up to N. On the level after that, in four separate recursive function frames we'll call `System.arraycopy` on arrays of length N/4, which again sums up to N for that whole layer of recursive calls.

Now we look for the height of our recursive tree. Each time, we halve the length of N, which means that the length of the array N on recursive level k is roughly $N * \left(\frac{1}{2}\right)^k$. Then we will finally reach our base case $N \leq 1$ when we have $N * \left(\frac{1}{2}\right)^k = 1$. Doing some math, we see this can be transformed into $N = 2^k$, which means $k = \log_2(N)$. In other words, the number of layers in our recursive tree is $\log_2(N)$. If we have $\log_2(N)$ layers with $\Theta(N)$ work on each layer, we must have $\Theta(N \log(N))$ runtime.

(c) Given that `exponentialWork` runs in $\Theta(3^N)$ time with respect to input N, what is the runtime of `amy`?

```
public void ronnie(int N) {
    if (N <= 1) {
        return;
    }
    amy(N - 2);
    amy(N - 2);
    amy(N - 2);
    exponentialWork(N); // Runs in $Theta(3^N)$ time
}
```

$\Theta(3^N)$. Drawing out the recursive tree, the first level has $\Theta(3^N)$ work, the next level has $\Theta(3^{N-1})$ work, and so on until the last level which has approximately $\Theta\left(3^{\frac{N}{2}}\right)$ work. This gives the sum $\Theta(3^N) + \Theta(3^{N-1}) + ... + \Theta\left(3^{\frac{N}{2}}\right) = \Theta(3^N)$.

2 Asymptotics is Fun!

(a) Using the function `g` defined below, what is the runtime of the following function calls? Write each answer in terms of `N`. Feel free to draw out the recursion tree if it helps.

```
public static void g(int N, int x) {
    if (N == 0) {
        return;
    }
    for (int i = 1; i <= x; i++) {
        g(N - 1, i);
    }
}
```

`g(N, 1)`: $\Theta($`____`$)$

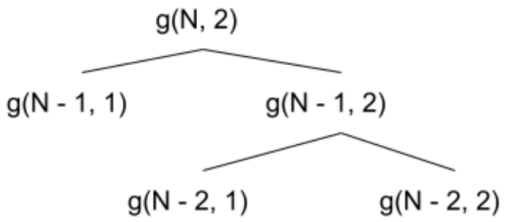
`g(N, 1)`: $\Theta(N)$

Explanation: When `x` is 1, the loop gets executed once and makes a single recursive call to `g(N - 1)`. The recursion goes `g(N)`, `g(N - 1)`, `g(N - 2)`, and so on. This is a total of `N` recursive calls, each doing constant work.

`g(N, 2)`: $\Theta($`____`$)$

`g(N, 2)`: $\Theta(N^2)$

Explanation: When `x` is 2, the loop gets executed twice. This means a call to `g(N)` makes 2 recursive calls to `g(N - 1, 1)` and `g(N - 1, 2)`. The recursion tree looks like this:



From the first part, we know `g(..., 1)` does linear work. Thus, this is a recursion tree with N levels, and the total work is $(N-1)+(N-2)+...+1 = \Theta(N^2)$ work.

(b) Suppose we change line 6 to `g(N - 1, x)` and change the stopping condition in the for loop to `i <= f(x)` where `f` returns a random number between 1 and `x`, inclusive. For the following function calls, find the tightest Ω and big O bounds. Feel free to draw out the recursion tree if it helps.

```
public static void g(int N, int x) {
    if (N == 0) {
        return;
    }
    for (int i = 1; i <= f(x); i++) {
        g(N - 1, x);
    }
}
```

`g(N, 2)`: $\Omega($`____`$)$, $O($`____`$)$

`g(N, N)`: $\Omega($`____`$)$, $O($`____`$)$

`g(N, 2)`: $\Omega(N)$, $O(2^N)$

`g(N, N)`: $\Omega(N)$, $O(N^N)$

Explanation: Suppose `f(x)` always returns 1. Then, this is the same as case 1 from (a), resulting in a linear runtime.
On the other hand, suppose `f(x)` always returns x. Then `g(N, x)` makes `x` recursive calls to `g(N - 1, x)`, each of which makes `x` recursive calls to `g(N - 2, x)`, and so on, so the recursion tree has 1, x, x^2 … nodes per level. Outside of the recursion, the function `g` does `x` work per node. Thus, the overall work is $x * 1 + x * x + x * x^2 + ... + x * x^{N-1} = x(1 + x + x^2 + ... + x^{N-1})$.
Plug in `x = 2` to get $2(1 + 2 + 2^2 + ... + 2^{N-1}) = O(2^N)$ for our first upper bound. Plug in `x = N` to get $N(1 + N + N^2 + ... + N^{N-1}) = O(N^N)$ (ignoring lower-order terms).

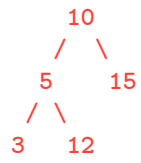
3 Is this a BST?

In this setup, assume a **BST** (Binary Search Tree) has a **key** (the value of the tree root represented as an **int**) and pointers to two other child BSTs, **left** and **right**. Additionally, assume that **key** is between **Integer.MIN_VALUE** and **Integer.MAX_VALUE** non-inclusive.

(a) The following code should check if a given binary tree is a BST. However, for some trees, it returns the wrong answer. Give an example of a binary tree for which **brokenIsBST** fails.

```
public static boolean brokenIsBST(BST tree) {
    if (tree == null) {
        return true;
    } else if (tree.left != null && tree.left.key >= tree.key) {
        return false;
    } else if (tree.right != null && tree.right.key <= tree.key) {
        return false;
    } else {
        return brokenIsBST(tree.left) && brokenIsBST(tree.right);
    }
}
```

Here is an example of a binary tree for which **brokenIsBST** fails:



The method fails for some binary trees that are not BSTs because 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 an example of a tree for which it fails.

It is important to note that the method does indeed return true for every binary tree that actually is a BST (it correctly identifies proper BSTs).

(b) Now, write **isBST** that fixes the error encountered in part (a).

Hint: You will find **Integer.MIN_VALUE** and **Integer.MAX_VALUE** helpful.

Hint 2: You want to somehow store information about the keys from previous layers, not just the direct parent and children. How do you use the parameters given to do this?

```
public static boolean isBST(BST T) {
    return isBSTHelper(_____________________________________________);
}

public static boolean isBSTHelper(BST T, int min, int max) {

    if (_____________________________________) {

        ------------------------------------------------

    } else if (_____________________________________________) {

        ------------------------------------------------

    } else {

        ------------------------------------------------

    }
}
```

Solution:

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

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

Explanation:

A BST is a naturally recursive structure, so it makes sense to use a recursive helper to go through the BST and ensure it is valid. Specifically, our recursive helper will traverse the BST while tracking the minimum and maximum valid values for subsequent nodes along our current path. We can get these minimum and maximum values by remembering the key property of BSTs: nodes to the left of our current node are always less than the current value, and nodes to the right of our current node are always greater than our current value. So for example, if we encounter a node with value 5, anything to the left must be < 5.

In our base case, an empty BST is always valid. Otherwise, we can check the current node. If it doesn't fall within our precomputed min/max, we know this is invalid, and return immediately.

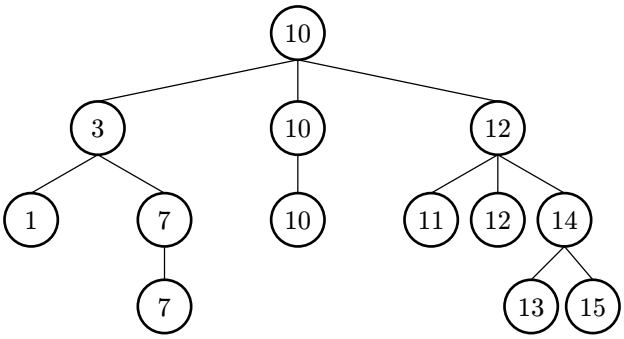
Otherwise, we use the properties of BSTs to bound our subsequent min and max values. If we traverse to the left, everything must be less than or equal to the current value, so the value of our current node becomes the new texttt{max} for the tree at `T.left`. Similar logic applies to the right.

4 Extra: Tri-nary Search Tree

We'd like a data structure that acts like a BST (Binary Search Tree) in terms of operation runtimes but allows duplicate values. Therefore, we decide to create a new data structure called a TST (Trinary Search Tree), which can have up to three children, which we'll refer to as `left`, `middle`, and `right`. In this setup, we have the following invariants, which are very similar to the BST invariants:

1. Each node in a TST is a root of a smaller TST
2. Every node to the `left` of a root has a value "lesser than" that of the root
3. Every node to the `right` of a root has a value "greater than" that of the root
4. **Every node to the `middle` of a root has a value equal to that of the root**

Below is an example TST to help with visualization.



Describe an algorithm that will print the elements in a TST in **descending** order. *(Hint: recall that an in-order traversal for a BST gives elements in increasing order.)*

Solution:

Inorder traversal on a BST yields the sorted elements in the BST in ascending order. Therefore, the core of the algorithm we'd like here is going to be quite similar to inorder traversal, but reversed (visit the right child before the left child) and with the added caveat that we also must traverse through the middle children.

In essence, given the root of some TST, we reverse onto the right child subtree, then print the root's value, then reverse onto the middle child subtree, then finally reverse onto the left subtree. The print root value and reverse onto the middle child steps can be swapped, because overall the order of the printed values should be the same.

Pseudocode:

```
reverse(tst):
    if tst is null:
        return
    reverse(tst.right)
    print(tst.value)
    reverse(tst.middle)
    reverse(tst.left)
```