CS 61B Spring 2024

ADTs, Asymptotics II, BSTs

Exam-Level 06: February 26, 2024

1 Finish the Runtimes

Below we see the standard nested for loop, but with missing pieces!

```
for (int i = 1; i < ______; i = ______) {
    for (int j = 1; j < ______; j = ______) {
        System.out.println("Circle is the best TA");
    }
}
```

For each part below, **some** of the blanks will be filled in, and a desired runtime will be given. Fill in the remaining blanks to achieve the desired runtime! There may be more than one correct answer.
Hint: You may find `Math.pow` helpful.

(a) Desired runtime: $\Theta(N^2)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = ______) {
        System.out.println("This is one is low key hard");
    }
}
```

Solution:

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = j + 1) {
        System.out.println("This is one is low key hard");
    }
}
```

Explanation: Remember the arithmetic series $1+2+3+4+\ldots+N = \Theta(N^2)$. We get this series by incrementing j by 1 per inner loop.

(b) Desired runtime: $\Theta(\log(N))$

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < ______; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Solution: Any constant would work here, 2 was chosen arbitrarily.

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < 2; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

Explanation: The outer loop already runs $\log n$ times, since i doubles each time. This means the inner loop must do constant work (so any constant `j < k` would work).

(c) Desired runtime: $\Theta(2^N)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < ______; j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

Solution:

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < Math.pow(2, i); j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

Explanation: Remember the geometric series $1+2+4+...+2^N = \Theta(2^N)$. We notice that i increments by 1 each time, so in order to achieve this 2^N runtime, we must run the inner loop 2^i times per outer loop iteration.

(d) Desired runtime: $\Theta(N^3)$

```
for (int i = 1; i < ______; i = i * 2) {
    for (int j = 1; j < N * N; j = ______) {
        System.out.println("yikes");
    }
}
```

Solution:

```
for (int i = 1; i < Math.pow(2, N); i = i * 2) {
    for (int j = 1; j < N * N; j = j + 1) {
        System.out.println("yikes");
    }
}
```

Explanation: One way to get N^3 runtime is to have the outer loop run N times, and the inner loop run N^2 times per outer loop iteration. To make the outer loop run N times, we need stop after multiplying `i = i * 2` N times, giving us the condition `i < Math.pow(2, N)`. To make the inner loop run N^2 times, we can simply increment by 1 each time.

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.

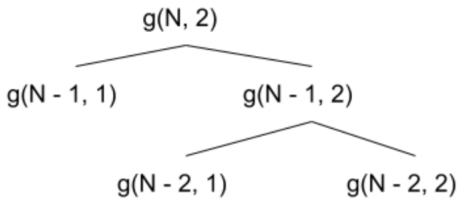
```
1  void g(int N, int x) {
2      if (N == 0) {
3          return;
4      }
5      for (int i = 1; i <= x; i++) {
6          g(N - 1, i);
7      }
8  }
```

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

Solution:
`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)`: $\Theta($ $)$

Solution: `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.

```
1  void g(int N, int x) {
2      if (N == 0) {
3          return;
4      }
5      for (int i = 1; i <= f(x); i++) {
6          g(N - 1, x);
7      }
```

```
}
```

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

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

Solution:
`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).

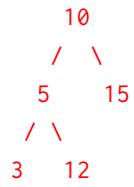
3 Is This a BST?

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`.

(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.

```
1  public static boolean brokenIsBST(BST tree) {
2      if (tree == null) {
3          return true;
4      } else if (tree.left != null && tree.left.key > tree.key) {
5          return false;
6      } else if (tree.right != null && tree.right.key < tree.key) {
7          return false;
8      } else {
9          return brokenIsBST(tree.left) && brokenIsBST(tree.right);
10     }
11 }
```

Solution: 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:

```
1  public static boolean isBST(BST T) {
2      return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
3  }
4
5  public static boolean isBSTHelper(BST T, int min, int max) {
6      if (T == null) {
7          return true;
8      } else if (T.key < min || T.key > max) {
9          return false;
10     } else {
11         return isBSTHelper(T.left, min, T.key)
12                && isBSTHelper(T.right, T.key, max);
13     }
14 }
```

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 or equal to the current value, and nodes to the right of our current node are always greater than or equal to 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 `max` for the tree at `T.left`. Similar logic applies to the right.