Section 10

Monday, June 3, 2019 3:08 PM

section10_short

(d) What is the simplified tight oh bound for the runtime of each of the loops above? Ignore the context of the if conditions for now. Finally, give the overall simplified runtime of method4.

```
public void method4(int n) {
        for (int i = 0; i < n; i++) {
                if (i == 0) {
                        for (int j = 0; j < n; j++) {
                                System.out.println("lol");
                        }
                } else if (n < 1000) {
                        for (int j = 0; j < n * n; j++) {
                                System.out.println("rofl");
                        }
                } else {
                        for (int j = 0; j < 10000; j++) {
                                System.out.println("haha");
                        }
                }
        }
}
```

4. Code Modeling 2: Electric Boogaloo

For each scenario, provide a Θ bound for both the **best** and **worst** case runtimes for the function. Make sure your bounds are as tight/simplified as possible. Give your answers in terms of n, the number of strings in `input`.

You may assume that all strings are of constant length, but can make no other assumptions about the input. You should also assume that all iterators are efficient.

(a) Consider the following snippet of Java code:

```
public static IDictionary<String, Integer> countStrings(ISet<String> input) {
    IDictionary<String, Integer> dict = new BSTDictionary<String, Integer>();

    for (String curString : input)
    {
        dict.put(curString, 1 + dict.getOrDefault(curString, 0));
    }

    return dict;
}
```

(i) Describe the runtime of this function in terms of n, the number of strings contained in `input`. Provide bounds for both the **best** and **worst** case runtimes. Note that `BSTDictionary` uses a BST to implement a dictionary.

(ii) Do the same thing, but assume now that `dict` is of type `AVLDictionary`. This dictionary is implemented using an AVL Tree, just like in the experiments for HW5.

(iii) Do the same thing, but assume now that `dict` is of type `ArrayDictionary`. This is the data structure you implemented in HW2. You can ignore the time it takes to resize.

(iv) Do the same thing, but assume now that `dict` is of type `ChainedHashDictionary`. This is the data structure you implemented in HW4. You can ignore the time it takes to resize. You can also assume that `.hashCode()` takes constant time.

(b) For the same four `IDictionary` implementations, provide best and worst case runtime bounds for **this** code snippet:

```
public static IDictionary<String, Integer> countStrings(IList<String> input) {
    IDictionary<String, Integer> dict = new BSTDictionary<String, Integer>();

    for (String curString : input)
    {
        dict.put(curString, 1 + dict.getOrDefault(curString, 0));
    }

    return dict;
}
```

Hint 1: You should be able to do this problem very quickly by looking at your answers for the previous snippet.
Hint 2: What's different about this function?

(i)

(ii)

(iii)

(iv)

5. Heaps

Consider the following list of numbers:

`[1, 5, 2, -6, 7, 12, 3, -5, 6, 2, 1, 6, 7, 2, 1, -2]`

(a) Insert these numbers into a min 2-heap (into a min-heap with up to two children per node). Show both the final tree and the array representation.

(b) Insert these numbers into a max 3-heap (a max heap with up to three children per node). Show both the final tree and the array representation.

(c) Insert these numbers into a min 4-heap using Floyd's `buildHeap` algorithm. Show both the final tree and the array representation.

(d) Insert these numbers into a max 2-heap using Floyd's `buildHeap` algorithm. Show both the final tree and the array representation.

(e) Suppose we modify Floyd's `buildHeap` algorithm so we start from the front of the array, iterate forward, and call `percolateDown(...)` on each element. Why is this a bad idea?