CSM 61B
Spring 2017

Asymptotic Analysis

Week 8: March 6, 2017

1 Something Fishy

Give a tight asymptotic runtime bound for each of the following functions. Assume array is an $M \times N$ matrix (*rows* $\times$ *cols*).

1.1
```
public static int redHerring(int[][] array) {
    if (array.length < 1 || array[0].length <= 4) {
        return 0;
    }
    for (int i = 0; i < array.length; i++) {
         for (int j = 0; j < array[i].length; j++) {
            if (j == 4) {
                return -1;
            }
        }
    }
    return 1;
}
```

$\Theta(1)$

1.2
```
public static int crimsonTuna(int[][] array) {
    if (array.length < 4) {
        return 0;
    }
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
              if (i == 4) {
                return -1;
            }
        }
    }
    return 1;
}
```

$O(N)$

1.3
```
public static int pinkTrout(int a) {
    if (a % 7 == 0) {
        return 1;
    } else {
        return pinkTrout(a - 1) + 1;
    }
}
```

$\Theta(1)$

1.4
```
public static boolean scarletKoi(int[] sortedArray, int x) {
    int N = sortedArray.length;
    return scarletKoi(sortedArray, x, 0, N);
}

private static boolean scarletKoi(int[] sortedArray, int x, int start, int end) {
    if (start == end || start == end - 1) {
        return sortedArray[start] == x;
    }
    int mid = end + ((start - end) / 2);
    return sortedArray[mid] == x ||
           scarletKoi(sortedArray, x, start, mid) ||
           scarletKoi(sortedArray, x, mid, end);
}
```

$O(N)$

This method is a trap, as it seems like a binary search. But in the recursive case, we make recursive calls on both the left and right sides, *without taking advantage of the sorted array*. We can craft an input that requires exploring the entire array in linear time.

2 Amortized Analysis

2.1 Mallory is designing a resizing `ArrayList` implementation. She needs to decide the amount to resize by. Help her figure out which option provides the best runtime.

Assuming Mallory resizes her `ArrayList` when it's full, what is the average runtime of adding an element to the `ArrayList`?

(a) When full, increase the size of array by 10,000 elements.

This would still be in $\Theta(N)$ since the constant, 10,000, does not scale with the size of the array. We're resizing the array every 10,000 inputs, regardless of the size of the array which could be much larger than 10,000.

(b) When full, double the size of the array.

This would be in $\Theta(1)$ since at any given length, N, we can insert $N/2$ elements before having to resize.

3 Triple Trouble

3.1 Given a linked list of length N, provide the runtime bound for each operation. Recall that `IntList` is the naive linked list implementation, `SLList` is an encapsulated singly-linked list with a front sentinel, and `LinkedList` is Java's encapsulated doubly-linked list implementation with pointers to the first and last node.

Operation	`IntList`	`SLList`	`LinkedList`
`size()`	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$
`get(int index)`	$O(N)$	$O(N)$	$O(N)$
`addFirst(E e)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`addLast(E e)`	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
`addBefore(E e, Node n)`	$O(N)$	$O(N)$	$\Theta(1)$
`remove(int index)`	$O(N)$	$O(N)$	$O(N)$
`remove(Node n)`	$O(N)$	$O(N)$	$\Theta(1)$
`reverse()`	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$

(a) Give the runtime of `addAll(Collection<E> c)` assuming an empty linked list and `c` of size N. Assume `addAll` is implemented by calling `addLast` repeatedly.

`IntList`: $\Theta(N^2)$
`SLList`: $\Theta(N^2)$
`LinkedList`: $\Theta(N)$

(b) How can we do better?

Instead of calling `addLast`, keep track of the last node and append each new element in constant time to the end of the list.

4 Nearest Duplicate *Extra for Experts*

4.1 Define a procedure, `nearestDuplicate`, that accepts a `String[] array` and returns the string closest to its duplicate. For example, if given the following input:

```
{ "all", "work", "and", "no", "play", "makes", "for", "no", "work", "no", "fun", "and", "no", "results" }
```

`nearestDuplicate` would return `no` because the second and third `no`'s are the closest. This function should run in $O(N)$ time, where N is the size of the array.

```java
public static String nearestDuplicate(String[] array) {
    int min = Integer.MAX_VALUE;
    String duplicate = null;
    Map<String,Integer> nearestIndices = new HashMap<>();
    for (int i = 0; i < array.length; i++) {
        String s = array[i];
        if (nearestIndices.containsKey(s) && i - nearestIndices.get(s) < min) {
            min = i - nearestIndices.get(s);
            duplicate = s;
        }
        nearestIndices.put(s, i);
    }
    return duplicate;
}
```