CSM 61B — Asymptotic Analysis

Fall 2017 — Mentoring 6: October 2, 2017

1 Analysis of Algorithms

The **running time** of a program can be modeled by the number of instructions executed by the computer. To simplify things, suppose arithmetic operators (`+`, `-`, `*`, `/`), logical operators (`&&`, `||`, `!`), comparison (`==`, `<`, `>`), assignment, field access, array indexing, and so forth take 1 unit of time. `(6 + 3 * 8) / 3` would take 3 units of time, one for each arithmetic operator.

While this measure is fine for simple operations, many problems in computer science depend on the size of the input: `fib(3)` executes almost instantly, but `fib(10000)` will take much longer to compute.

Asymptotic analysis is a method of describing the run-time of an algorithm *with respect* to the size of its input. We can now say,

> The run-time of `fib` is, at most, within a factor of 2^N where N is the size of the input number.

Or, in formal notation, `fib(n)` $\in O(2^N)$.

1.1 Define, in your own words, each of the following asymptotic notation.

(a) O

(b) Ω

(c) Θ

1.2 Give a tight asymptotic runtime bound for `containsZero` as a function of N, the size of the input array in the *best case, worst case, and overall.*

```java
public static boolean containsZero(int[] array) {
    for (int value : array) {
        if (value == 0) {
            return true;
        }
    }
    return false;
}
```

2 Something Fishy

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

2.1
```java
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;
}
```

2.2
```java
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;
}
```

2.3
```java
public static int pinkTrout(int a) {
    if (a % 7 == 0) {
        return 1;
    } else {
        return pinkTrout(a - 1) + 1;
    }
}
```

2.4 (a) Give a $O(\cdot)$ runtime bound as a function of N, `sortedArray.length`.

```java
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);
}
```

(b) Why can we only give a $O(\cdot)$ runtime and not a $\Theta(\cdot)$ runtime?

3 Linky Listy *Extra Practice*

3.1 Given a linked list of length N, give a tight asymptotic runtime bound for each operation. Recall that `IntList` is a naive linked list, `SLList` is an encapsulated singly-linked list with a front sentinel, and `DLList` is an encapsulated doubly-linked list with front and back pointers.

Operation	`IntList`	`SLList`	`DLList`
`size()`			
`get(int index)`			
`addFirst(E e)`			
`addLast(E e)`			
`addBefore(E e, Node n)`			
`remove(int index)`			
`remove(Node n)`			

```java
class IntList {
    int first;
    IntList rest;
}

class SLList {
    static class IntNode {
        int item;
        IntNode next;
    }

    IntNode sentinel;
    int size;
}

class DLList {
    static class IntNode {
        int item;
        IntNode next;
        IntNode prev;
    }

    IntNode head;
    IntNode tail;
    int size;
}
```

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

(b) How can we do better?