CSM 61B Fall 2017

Asymptotic Analysis

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

'Big-O' notation gives an *upper* bound on the runtime of a function as the size of the input approaches infinity.

(b) Ω

'Big-Omega' notation gives a *lower* bound on the runtime of a function as the size of the input approaches infinity.

(c) Θ

'Big-Theta' notation gives a *tight* bound on the runtime of a function as the size of the input approaches infinity.

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.*

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

$\Theta(1)$ in the best case, $\Theta(N)$ in the worst case, and $\Omega(1), O(N)$ overall.

Asymptotic analysis is always concerned with what happens as our input grows to infinity. In this case, we'd like to describe the order of growth of `containsZero` as a function of the *size* of the input array, but that doesn't say anything about the contents of the input array.

We could find a zero at the beginning of the array: this is the best case as we can terminate in $\Theta(1)$. Or there could be no zeroes in the array at all: this is the worst case, which terminates $\Theta(N)$. We can then describe the overall runtime of the function taking into account all possible cases from spanning from best to worst: hence, the overall runtime is $\Omega(1), O(N)$.

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
```
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;
}
```

$\Theta(1)$. The function will return once it reaches the fourth element of the first row of the `array` matrix, so this function always takes constant time. (And if the array doesn't have any rows, or if the first row has fewer than 4 entries, the function returns immediately.)

2.2
```
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;
}
```

$O(N)$. This function returns after it reaches the fourth row (and if there are less than four rows it returns immediately). The number of elements it's able to reach in four rows is dependent on the number of columns, which in this case is N.

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

$\Theta(1)$. If a is a multiple of 7, this function will return immediately. If not, it will call `pinkTrout(a - 1)` and add one to its result. We know that, at most, it will take 6 calls until `pinkTrout` is called on a multiple of 7, so thus the function takes constant time.

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

```
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.

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

In the best case, the middle element will be equal to `x`, in which case the runtime will be $\Theta(1)$. Thus, overall, the runtime of the function is in $\Omega(1), O(N)$ and there's no $\Theta(\cdot)$ runtime because the $\Omega(\cdot)$ and $O(\cdot)$ runtimes don't match.

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()`	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$
`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)`	$O(N)$	$O(N)$	$\Theta(1)$

`size()`: `SLList` and `DLList` both save size as a field, whereas `IntList` has to iterate through the whole list to find the size.

`addBefore(E e, Node n)`: `DLList` has a pointer from each node to the next node and to the previous node, so adding before is constant time. `IntList` and `SLList` have to iterate through the list from the start to find the node before the current node.

`remove(Node n)`: Because `DLList` has a pointer to the previous node and next node, it can easily remove the node and point the previous node to the next node (and point the next node point to the previous node). However, `IntList` and `SLList` have to iterate through the list to find the node before the current one.

(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.

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

`IntList` and `SLList` both take $\Theta(N)$ time to `addLast(E e)`, where `N` is number of items currently in the list. The trick here is to notice that we

```
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;
}
```

start adding to an empty list and add until we fill the list to size `N`. So every addition takes variable time since the size of the list is different at each addition. We can calculate the size of the list before each addition as follows:

$$0 + 1 + 2 + \cdots + N - 1 = \frac{N(N-1)}{2}$$

Therefore, it takes $\Theta(N^2)$ time.

`DLList`: $\Theta(N)$

(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.