CSM 61B Fall 2020

Asymptotics

Mentoring 6: September 28 - October 2, 2020

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*) and that M and N are both large.

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;
}
```

$\Theta(N)$. This problem assumes that M and N are large numbers, which means that the input matrix has a large number of rows and columns.
Best Case- Although an input that has less than 4 rows would make this function return immediately, we can not consider it to be a 'best case' input, because that would mean we are imposing a limit on the number of rows. A valid input matrix would therefore have to be one that is arbitrarily large, but satisfies the if (i == 4) condition almost immediately (ex- any matrix that has at least 4 rows). The work that this function has to do in order to reach the 4th row is dependent on how many elements it visits, which is equal to the number of elements in the previous 3 rows, or 3N. $\Theta(N)$ in the best case.

Worst Case- No matter what the values of an input matrix are, the function will always hit the if (i == 4) condition at the start of the 4th row, which means that it will always visit 3N elements before it will return out. $\Theta(N)$ in the worst case.

Therefore, the this function runs in $\Theta(N)$ overall.

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?

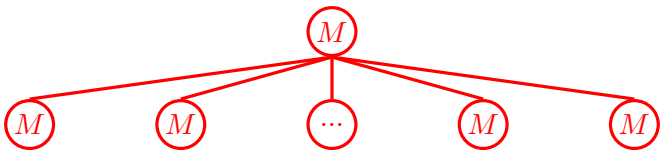
In the best case, the middle element will be equal to `x`, in which case the runtime will be $\Theta(1)$. Thus, overall, the runtime of the function is in $\Omega(1), O(N)$ and there's no $\Theta(\cdot)$ runtime because the $\Omega(\cdot)$ and $O(\cdot)$ runtimes don't match.

3 Shoutout Recursion

3.1 Give a tight asymptotic bound for `many_N` as a function of N and M. If possible, give a $\Theta(\cdot)$ bound for the overall runtime. Otherwise, provide a $\Theta(\cdot)$ bound for both the best case and worst case runtime.

```
int many_N(int N, int M) {
    if (N == 0) { return 1;}
    int k = 0;
    for (int i = 0; i <= M; i++) {
        k += many_N(N-1, M)
    }
    return k;
}
```

$\Theta(M^N)$



We only draw the first two levels of the recursive tree above. The value inside each node represents its corresponding function call's runtime. Since we loop M times, we know that the runtime of the first function call is M. This also means we make M recursive calls, meaning that there will be M child nodes branching from the root nodes. An important observation to make is that the value of M never changes in our recursive calls. This suggests that each of the child nodes will also have a runtime of M which implies that their branching factor will also be M. Now that we have ascertained the branching factor and runtime of each node, all that remains is finding the height of the tree. We note that since N is decremented by 1 on each recursive call, the height of the tree will be N.

The information we have so far is the following

- Runtime of each node is M
- Branching factor of each node is M
- Height of tree is N

We can use this information to calculate the total work being done at each level of the tree. The first level is doing M work. The second level is doing M^2 work. The third level is doing M^3 work. We can make the following summation $M + M^2 + M^3 + ... + M^N \approx M^N$

3.2 Give a tight asymptotic bound for `skip` as a function of N. If possible, give a $\Theta(\cdot)$ bound for the overall runtime. Otherwise, provide a $\Theta(\cdot)$ bound for both the best case and worst case runtime.

```
void skip(int N) {
    if (N <= 1) { return;}
    else {
        skip(N/N);
        skip(N/2);
    }
}
```

$\Theta(log(N)))$

Every function call (node) in the tree above is given a constant runtime since regardless of what the value of N, we are always processing less than 5 lines of code. The branching factor is 1 for every node since the first recursive call skip(N/N) returns immediately. This means we simply need to get the height of the tree to solve for the overall runtime. Since N is being divided by 2 on each function call, the height of the tree is $logN$ which implies the runtime is always $logN$

4 Disjoint Sets Intro

4.1 Suppose we have a `WeightedQuickUnionUF` disjoint set with path compression. Show the tree structure in the union-find algorithm as the following sequence of commands is executed.

```
connect(1, 2);
connect(3, 4);
connect(5, 6);
connect(1, 6);
connect(3, 6);
```