DSC 40B - Homework 03

Due: Monday, October 23

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 p.m.

Problem 1.

Determine the worst case time complexity of each of the recursive algorithms below. In each case, state the recurrence relation describing the runtime. Solve the recurrence relation, either by unrolling it or showing that it is the same as a recurrence we have encountered in lecture.

a)

```
import math
def summation(numbers):
    """Given a list, returns the sum of the numbers in the list."""
    n = len(numbers)
    if n == 0:
        return 0
    if n == 1:
        return numbers[0]
    middle = math.floor(n / 2)
    return summation(numbers[:middle]) + summation(numbers[middle:])
```

Hint: remember that slicing causes a copy to happen!

> **Solution:** Note that the slicing in `numbers[:middle]` takes linear time in the length of the array, so the recurrence relation is:
>
> $$T(n) = \Theta(n) + 2T(n/2)$$
>
> This is the same recurrence relation as for mergesort, and the solution is $\Theta(n \log n)$.

b)

```
import math
def summation_2(numbers, start, stop):
    """Computes the sum of numbers[start:stop]"""
    if stop <= start:
        return 0
    if stop - start == 1:
        return numbers[start]
    middle = math.floor((start + stop) / 2)
    left_sum = summation_2(numbers, start, middle)
    right_sum = summation_2(numbers, middle, stop)
    return left_sum + right_sum
```

Hint: recall the formula for a geometric sum: $\sum_{p=0}^{N} b^p = \frac{1-b^{N+1}}{1-b}$

> **Solution:** No slicing is being done, so a constant amount of time is required outside of the recursive calls. Therefore the recurrence relation is:
>
> $$T(n) = 2T(n/2) + \Theta(1)$$

We have not seen this recurrence before, so we must solve it by unrolling:

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2\left[2T(n/4) + 1\right] + 1 \\ &= 4T(n/4) + 2 + 1 \\ &= 4\left[2T(n/8) + 1\right] + 2 + 1 \\ &= 8T(n/8) + 4 + 2 + 1 \end{aligned}$$

At this point, a pattern has emerged, and we can guess that on unroll k we'll have:

$$T(n) = 2^k \cdot T(n/2^k) + 1 + 2 + 4 + \ldots + 2^{k-1}$$

We will reach $T(1)$ when $k = \log_2 n$. Making this substitution, we have:

$$\begin{aligned} T(n) &= 2^{\log_2 n} \cdot T(1) + 1 + 2 + 4 + \ldots + 2^{(\log_2 n)-1} \\ &= n + \left(1 + 2 + 4 + \ldots + 2^{(\log_2 n)-1}\right) \end{aligned}$$

The summation is a geometric sum, and, using the hint with $b = 2$, we have:

$$\begin{aligned} &= n + \frac{1 - 2^{\log_2 n}}{1 - 2} \\ &= n + \frac{1 - n}{1 - 2} \\ &= \Theta(n) \end{aligned}$$

c) In this subproblem, you may assume that `arr` is sorted.

```
import math
def ternary_search(arr, t, start, stop):
    """Return the index of t in arr[start:stop]"""
    if stop - start <= 0:
        return None
    n = stop - start
    left_ix = math.floor(start + (stop - start) / 3)
    right_ix = math.floor(start + 2 * (stop - start) / 3)
    if arr[left_ix] == t:
        return left_ix
    if arr[right_ix] == t:
        return right_ix
    if t < arr[left_ix]:
        return ternary_search(arr, t, start, left_ix)
    elif t > arr[right_ix]:
        return ternary_search(arr, t, right_ix + 1, stop)
    else:
        return ternary_search(arr, t, left_ix + 1, right_ix)
```

Solution: This is like binary search, but the recursive calls are on arrays of size $n/3$. Hence the recurrence is:

$$T(n) = T(n/3) + \Theta(1)$$

The solution will be the same as the solution for binary search, except all instances of $\log_2 n$ will be replaced by $\log_3 n$, so the overall time complexity is the same: $\Theta(\log n)$.

Problem 2.

While on campus, you notice someone standing next to a very tall ladder getting ready to replace a lightbulb. As they are about to get started, though, they are distracted by a raccoon and accidentally drop the bulb to the ground. Surprisingly, though, the bulb doesn't break!

You wonder to yourself: how high up could the bulb be dropped without it breaking? The person goes away for a moment, leaving their ladder, two bulbs, and an opportunity for you to find out the answer to your question. They'll be back soon, though, so you must hurry. You decide to test the bulbs by dropping them and seeing when they break.

More formally, the ladder has n rungs (that is, n steps). You want to find out the maximum step, $n_{\max}$, that you can drop a bulb without it breaking. You'll assume that the two bulbs are both equally-strong, and will break on the same step. Since time is limited, you want to find out the answer to your question in as few bulb-drops as you can. You're allowed to break both bulbs.

One approach is essentially linear search. Here, you stand on step 1, drop the bulb, and see if it breaks. If it doesn't, move to step 2, and so on. If the bulb breaks on step k, then you know the highest you can drop the bulb from is step $k-1$. While this strategy is guaranteed to find you the answer and breaks only one bulb, it is time consuming: in the worst case, you'll need to drop the bulb $\Theta(n)$ times.

But you've taken DSC 40B, so you're clever – what about binary search? In this approach, you'll start on step $n/2$, and drop the first bulb – if it doesn't break, you'll go higher. This seems more efficient, but there's a big problem with this: what if you break the bulb on the first drop? Then you only have one bulb remaining, and you have to be careful. You'll need to go back to using linear search, dropping the remaining bulb from step 1, step 2, and so forth until it breaks. In the worst case this linear search will do around $n/2 = \Theta(n)$ drops.

However, there is a strategy (many strategies, actually) for finding out $n_{\max}$ by breaking at most two bulbs and in the worst case making a number of drops $f(n)$ that is asymptotically much smaller than n. That is, if $f(n)$ is the number of drops needed by your method in the worst case, it should be true that $\lim_{n\to\infty} f(n)/n = 0$.

Give such a strategy and state the number of drops it needs in the worst case, $f(n)$, using asymptotic notation. There are many strategies that satisfy the conditions – yours does not need to be the most efficient.

Solution: Go up to step $\sqrt{n}$ and drop the first bulb. If it doesn't break, go up to step $2\sqrt{n}$ and drop. Repeat until the bulb breaks. Say it breaks at step $k\sqrt{n}$. Then you only need to check steps $(k-1)\sqrt{n}$ through $k\sqrt{n}$, which you can do with a linear search in less than $\sqrt{n}$ steps.

In the worst case, $n_{\max} = n-1$. In this case, you'll drop at $\sqrt{n}$, then $2\sqrt{n}$, then $3\sqrt{n}$ and so on, all the way to the top. It takes $n/\sqrt{n} = \sqrt{n}$ drops to get to the top. Then the bulb finally breaks, and you have to do a linear search to find $n_{\max}$ exactly – it could be anywhere between the top rung, n, and the location of the previous drop, $n - \sqrt{n}$. This linear search takes about $\sqrt{n}$ drops. The total number of drops is $\sqrt{n}$.

A concrete example might make this clearer. Say $n = 100$. Then start on step 10 and drop the bulb. Say it doesn't break. Move to step 20 and drop the first bulb again. Still doesn't break. Now drop from 30 – supposed it breaks. $n_{\max}$ is between 20 and 30, so drop the remaining bulb from 21, 22, 23, etc. until it breaks. In the worst case, you drop the bulb from $10, 20, 30, \ldots, 90, 100$, and it breaks at 100, so you then drop from $91, 92, \ldots, 99$.

Note that $\sqrt{n}$ is (close) to the optimal solution, but there are infinitely many others. For example, you

could drop from $\log n$, then $2\log n$, etc. Here the worst case would be $\Theta(n/\log n)$, which is still better than $\Theta(n)$. Similarly, you could drop at $n^{1/4}$, etc.

The most efficient solution is determined by the number of bulbs you have. If you have $\Theta(\log n)$ bulbs, the most efficient solution is binary search.

Programming Problem 1.

Commercial flights must balance the weight of passengers between the two sides of the plane in order to ensure a safe takeoff. Suppose you are piloting a full flight and must choose a passenger from the left and a passenger from the right to switch places so that the total weight of the passengers on the left side is as close as possible to the total weight of the passengers on the right.

In a file named `balance.py`, design an algorithm named `balance(left, right)` which takes in two *sorted* arrays, `left` and `right`, containing the weights of the passengers on the left side of the plane and the right side of the plane, respectively. Your function should return a pair `(left_ix, right_ix)` containing the index of the person on the left and the index of the person on the right who should be swapped to bring the weights to as close as balanced as possible. If the left and right sides are already balanced, return `None`. If there are multiple optimal solutions (that is, multiple ways of swapping passengers that result in the same optimal balance), return just one of them.

For example, suppose `left = [4, 8, 12]` and `right = [6, 15]`. When `balance(left, right)` is called, your function should return `(1, 0)` because the passenger with weight 8 on the left should swap with the passenger with weight 6 on the right. This would result in the left side having a total weight of 4 + 6 + 12 = 24 and the right side having a total weight of 8 + 15 = 23, which is as close as possible.

To receive full credit, your algorithm must be optimal in the sense that its worst case time complexity is as good as theoretically possible.

Solution:

```
def balance(left, right):
    left_sum = sum(left)
    right_sum = sum(right)

    best_swap = (None, None)
    best_difference = float('inf')


    left_ix = 0
    right_ix = 0

    while left_ix < len(left) and right_ix < len(right):
        updated_left = left_sum + right[right_ix] - left[left_ix]
        updated_right = right_sum + left[left_ix] - right[right_ix]

        difference = updated_left - updated_right
        if abs(difference) < best_difference:
            best_swap = left_ix, right_ix
            best_difference = abs(difference)

        # the right side is too small
        if difference == 0:
            return (left_ix, right_ix)
        elif difference > 0:
```

```
            # if the right side is too small, then swapping the current left_ix
            # with anything to the right of the current right_ix would just make
            # things worse. This means that we can rule out left_ix and increment it.
            left_ix += 1
        else:
            right_ix += 1

    return best_swap


def brute_force(left, right):
    left_sum = sum(left)
    right_sum = sum(right)

    best_swap = (None, None)
    best_difference = float('inf')

    for left_ix in range(len(left)):
        for right_ix in range(len(right)):
            updated_left = left_sum + right[right_ix] - left[left_ix]
            updated_right = right_sum + left[left_ix] - right[right_ix]

            difference = updated_left - updated_right
            if abs(difference) < best_difference:
                best_swap = left_ix, right_ix
                best_difference = abs(difference)

            if difference == 0:
                return (left_ix, right_ix)

    return best_swap
```