DSC 40B - Discussion 03

Problem 1.

a) State (but do not solve) the recurrence relation describing this function's run time.

```
import random
def foo(n):
    if n <= 2:
    return
for i in range(n):
    for j in range(i,n):
        print(i)
return foo(n//2) + foo(n//2)
```

> **Solution:** $T(n) = 2T(n/2) + \Theta(n^2)$

b) Suppose a binary search is performed on the following array using the implementation of *binary_search* from lecture. What is the worst case number of equality comparisons that would be made to search for an element in the array? That is, what is the greatest number of times that `arr[middle] == target` can be run?

```
[1, 4, 7, 8, 8, 10, 15, 51, 60, 65, 71, 72, 101]
```

> **Solution:** 4.
>
> The worst case number of comparisons occurs when what we're looking for is not in the array. Assume that the target is -999 so that every recursive call looks left (it actually matters whether we always look left or look right; if we look right, the number of comparisons will be one fewer).
>
> On the first call, `start` is 0 and the `stop` is 13. One comparison is made during this call to check if the middle element is the target.
>
> On the second call, `start` is 0 and the `stop` is 6. One comparison is made during this call to check if the middle element is the target.
>
> On the third call, `start` is 0 and the `stop` is 3. One comparison is made during this call to check if the middle element is the target.
>
> On the fourth call, `start` is 0 and the `stop` is 1. One comparison is made during this call to check if the middle element is the target.
>
> On the fifth call, `start` is 0 and the `stop` is 0. However, no comparisons between the target and other numbers are made during this call since the base case (any empty array) has been reached.

Problem 2.

a) Suppose you are sorting an array using selection sort. At some point during sorting, a snapshot of the array is recorded:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 2]
```

In this snapshot of the array, what is the maximum number outer loop iterations ran?

```
def selection_sort(arr):
    """In-place selection sort."""
    n = len(arr)
```

```
if n <= 1:
    return
for barrier_ix in range(n-1):
        # find index of min in arr[start:]
    min_ix = find_minimum(arr, start=barrier_ix) #swap
    arr[barrier_ix], arr[min_ix] = (
        arr[min_ix], arr[barrier_ix]
    )
```

Solution: 2.

Recall that selection sort works by having a pointer that points to the beginning of the array, then it from that index, it looks forward to find the minimum element and swap. The pointer then moves up to the next index (through the for loop) and the process repeats.

Therefore, at the end of the αth iteration of the loop, the first α elements of the array are sorted. Additionally, the first α elements of the array are the smallest α elements in the array.

In the given array, it may seem like the first 11 elemenets are in sorted order, so the outer loop must have ran a maximum of 11 times. However, this is not true because remember that selection sort finds the minimum element to swap with at each iteration. So by iteration 3, the 2 at the end of the array would have been swapped with the 3 at the index 2.

Since this is not the case, the outer loop must have ran a maximum of 2 times: the first two elements are sorted, and they are the smallest 2 elements in the whole array.

b) State the recurrence relation of the following code.

```
import math
def foo(arr):
    if len(arr) == 0:
        return 0

    if len(arr) == 1:
        return arr[0]

    one_third = math.floor(len(arr) / 3)
    left = arr[:one_third]
    middle = arr[one_third: one_third * 2]
    right = arr[one_third * 2:]

    for i in range(len(arr)):
        if i < one_third:
            arr[i] = 2 * arr[i] - sum(left)/len(left)
        if i >= one_third and i < one_third * 2:
            arr[i] = 2 * arr[i] - sum(middle)/len(middle)
        if i >= one_third * 2:
            arr[i] = 2 * arr[i] - sum(right)/len(right)

    total = sum(left) + sum(middle) + sum(right)

    return foo(left) + foo(middle) + foo(right) + total
```

Solution:

$$T(n) = \begin{cases} 3T(n/3) + \Theta(n^2) & n \geq 2 \\ \Theta(1) & n = 1 \text{ or } n = 0 \end{cases}$$

c) Solve the recurrence relation describing the above function's run time

Solution:

$$\begin{aligned} T(n) &= 3T(n/3) + n^2 && \text{(unrolling } k = 1\text{)} \\ &= 3[3T(n/9) + (n/3)^2] + n^2 = 9T(n/9) + (n^2/3) + n^2 && \text{(unrolling } k = 2\text{)} \\ &= 9[3T(n/27) + (n/9)^2] + (n^2/3) + n^2 = 27T(n/27) + n^2/9 + n^2/3 + n^2 \\ & && \text{(unrolling } k = 3\text{)} \end{aligned}$$

We can then infer that on the k'th unroll, $T(n) = 3^k T(n/3^k) + \sum_{i=0}^{k-1} \frac{n^2}{3^i}$
To get to the base case of $T(1)$, let $k = log_3(n)$, plugging in, we get:

$$T(n) = T(1) + \sum_{i=0}^{\log_3 n - 1} \frac{n^2}{3^i}$$

What about the sum on the right? We can rewrite it as:

$$\begin{aligned} \sum_{i=0}^{\log_3 n - 1} \frac{n^2}{3^i} &= n^2 \sum_{i=0}^{\log_3 n - 1} \frac{1}{3^i} \\ &= n^2 \sum_{i=0}^{\log_3 n - 1} (\frac{1}{3})^i \end{aligned}$$

This is a geometric series with $a = 1$, $r = \frac{1}{3}$, $m = 0$, and $n = \log_3 n - 1$. The sum of a geometric series is given by:

$$\sum_{i=m}^{n} ar^k = \frac{a(r^m - r^{n+1})}{1 - r}$$

Using this formula, we can find that the sum of the series is $\Theta(n^2)$.

$$\begin{aligned} n^2 \sum_{i=0}^{\log_3 n - 1} (\frac{1}{3})^i &= n^2 \frac{(1/3)^0 - (\frac{1}{3})^{\log_3 n}}{1 - 1/3} \\ &= n^2 \frac{1 - \frac{1}{n}}{2/3} \\ &= \frac{3}{2} n^2 (1 - \frac{1}{n}) \\ &= \frac{3}{2} n^2 - \frac{3}{2} n \\ &= \Theta(n^2) \end{aligned}$$

Problem 3.

We're given two sorted lists in ascending order, `A` and `B` and a target `t`, and our goal is to find an element a of `A` and an element b of `B` such that $a + b = t$.

Solution:

```
def target_sum(A,B,t):
    """
        A and B are list of sorted numbers
        t is the target to be found
    """

    # since the two arrays are sorted. we start with
    # initializing one pointer with the first index of any one array
    # and the other pointer with the last index of the other array
    A_i, B_i = 0, len(B) - 1

    while A_i < len(A) and B_i >= 0:

        current_sum = A[A_i] + B[B_i]

        # found target then return the values a and b
        if current_sum == t:
            return (A[A_i], B[B_i])


        elif current_sum < t:
            # current_sum was smaller! since we are incrementing the values in A
            # and decrementing the values in B
            # Increasing A_i will mean that the element in A that we get next will
            # increase our current sum, which is what is expected to find the target
            A_i += 1
        else:
            # current sum was larger!
            # Decreasing B_i will mean that the element in B that we get next will
            # decrease our current sum, which is what is expected to find the target
            B_i -= 1

    return None
```