DSC 40B - Discussion 03

Problem 1.

We'll consider an array of `True`s and `False`s to be sorted if all of the `False`s come before any of the `True`s, like in `[False, False, True, True, True]`.

The function `find_first_true(arr, start, stop)` below is a generalization of the binary search we saw in lecture. It accepts a sorted array of `True`s and `False`s and returns the index of the first `True` (if there is one) within `arr[start:stop]`; if the array contains all `False`s, it returns `stop`.

```
def find_first_true(arr, start, stop):
    if stop - start == 1:
        if arr[start]:
            return start
        else:
            return stop

    middle = math.floor((start + stop) / 2)
    if arr[middle]:
        return find_first_true(arr, start, middle)
    else:
        return find_first_true(arr, middle, stop)
```

Modify this function so that it takes in a sorted array of floats and a number a and returns the index of the first element that is $\geq a$.

Solution:

```
def find_first_geq(arr, x, *, start=0, stop=None):
    """Find first index that is >= x in arr[start:stop], where arr is sorted"""
    if stop is None:
        stop = len(arr)

    if stop - start == 1:
        if arr[start] >= x:
            return start
        else:
            return stop

    middle = math.floor((start + stop) / 2)
    if arr[middle] >= x:
        return find_first_geq(arr, x, start=start, stop=middle)
    else:
        return find_first_geq(arr, x, start=middle, stop=stop)
```

Problem 2.

a) Recursive code is not necessarily more efficient than iterative code. In fact, it is easy to write bad recursive code that is very slow if we aren't careful. Here's an example:

```
def summation(numbers):
    if len(numbers) == 0:
        return 0
    return numbers[0] + summation(numbers[1:])
```

This code might look fine, but it's actually needlessly slow. The reason is that `numbers[1:]` creates a *copy* of the list, and a copy takes time linear in the size of the new list (since that much memory has to be allocated).

Write down a recurrence relation for `summation` and solve it to determine the function's asymptotic time complexity.

Solution: The copy takes time $\Theta(n)$, and the recursive call is on an array of size $n-1$. Therefore our recurrence is:

$$T(n) = \begin{cases} T(n-1) + \Theta(n), & n =\geq 1 \\ \Theta(1), n = 0 \end{cases}$$

We'll replace the Θ notation to solve the recurrence: $T(n) = T(n-1)+n$. Unrolling a few times, we find:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \end{aligned}$$

In general, it looks like $T(n) = T(n-k)+n+(n-1)+\ldots+(n-k+1)$. The base case is reached when $k = n$, and on that iteration we have:

$$T(n) = T(n-n) + n + (n-1) + \ldots + (n-n+1) = \Theta(n^2)$$

So this algorithm takes quadratic time complexity!

b) Taking a long time to run is not the only way that an algorithm can be inefficient; it can also take a lot of memory. Just as we used time complexity to measure the speed of an algorithm, we can use *space complexity* to measure how much memory a piece of code requires. The space complexity of an algorithm is the amount of *auxiliary* (or *extra*) memory it requires to execute.

It turns out that `summation` is not only slow; it is wasteful of memory, too. When `numbers[1:]` is executed in the root call, a copy is made, resulting in the creation of a new list containing $n-1$ elements. This list is kept in memory while all subsequent recursive calls are made.

At the moment the base case is reached, what is the total size of all lists created by copying in all recursive calls made so far? State your answer exactly (and not in asymptotic notation), simplifying it as much as possible.

Solution: The root-level call on an array of size n creates a list with $n-1$ elements. The first recursive call, which is on an array of size $n-1$, creates a list with $n-2$ elements. So the kth call (where $k = 1$ is the root call) is on an array of size $n-k+1$, and creates a list of size $n-k$.

The base case occurs when the input list is empty; that is, when $n-k+1 = 0$. Solving for k, we find $k = n+1$. But no copy is made on this call (it returns before a copy is made). So the total size of all created lists is:

$$\sum_{k=1}^{n}(n-k) = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

Problem 3.

a) Consider the recurrence

$$T(n) = T\big(n/2\big) + n^2.$$

Solve the recurrence and give the asymptotic runtime using $\Theta(\cdot)$ notation. Provide a short justification.

Solution: Claim: $T(n) = \Theta(n^2)$.
Proof (unrolling). Unroll the recurrence k times:

$$\begin{aligned} T(n) &= T(n/2) + n^2 \\ &= T(n/2^2) + (n/2)^2 + n^2 \\ &= T(n/2^3) + (n/4)^2 + (n/2)^2 + n^2 \\ &= T(n/2^k) + n^2 \sum_{i=0}^{k-1} \frac{1}{4^i}. \end{aligned}$$

Stop when $n/2^k = 1$, i.e. $k = \log_2 n$. The geometric series $\sum_{i=0}^{\infty} 1/4^i = 1/(1-1/4) = 4/3$ is a constant, so the summed work is $n^2 \cdot O(1)$. The base term $T(1)$ is constant, hence

$$T(n) = T(1) + n^2 \cdot O(1) = \Theta(n^2).$$

b) Consider the recurrence

$$T(n) = T(n-1) + n^2.$$

Solve the recurrence and give the asymptotic runtime. Provide a short justification.

Solution: Claim: $T(n) = \Theta(n^3)$.
Proof (telescoping / summation). Expand down to the base case:

$$T(n) = T(1) + \sum_{k=2}^{n} k^2.$$

It is standard that $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$. Therefore

$$T(n) = \Theta\big(\sum_{k=1}^{n} k^2\big) = \Theta(n^3).$$

c) Consider the recurrence

$$T(n) = 4\,T\big(n/2\big) + n.$$

Solve the recurrence and compare its growth with the previous two recurrences. Give a short justification.

Solution: Claim: $T(n) = \Theta(n^2)$.
Proof (Master theorem and level-wise sum). Use the Master theorem: here $a = 4$, $b = 2$, and $f(n) = n$. Compute

$$n^{\log_b a} = n^{\log_2 4} = n^2.$$

Since $f(n) = n = O\big(n^{2-\epsilon}\big)$ for $\epsilon = 1$, we are in Master case 1, hence

$$T(n) = \Theta\big(n^{\log_b a}\big) = \Theta(n^2).$$

(Alternative level-wise justification.) At level i of the recursion tree there are 4^i nodes, each of

size $n/2^i$ and each contributing $f(n/2^i) = n/2^i$ work. So the total work at level i is

$$4^i \cdot \frac{n}{2^i} = n \cdot 2^i.$$

The tree has $\log_2 n$ levels; summing over $i = 0, \ldots, \log_2 n$ gives

$$\sum_{i=0}^{\log_2 n} n \cdot 2^i = n \cdot (2^{\log_2 n + 1} - 1) = n \cdot (2n - 1) = \Theta(n^2).$$

Comparison summary:

- $T(n) = T(n/2) + n^2$: top-level cost n^2 dominates; total $\Theta(n^2)$.
- $T(n) = T(n-1) + n^2$: many levels ($\Theta(n)$ depth) each contributing i^2; sum is $\Theta(n^3)$.
- $T(n) = 4T(n/2) + n$: branching increases number of subproblems, but $f(n) = n$ is smaller than the critical n^2, so the recursion tree sums to $\Theta(n^2)$.