DSC 40B - Homework 03

Due: Tuesday, January 26

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 on Tuesday at 11:59 p.m.

Problem 1. *(Eligible for Redemption)*

Determine whether each piece of code is correct or incorrect by thinking inductively. If it is correct, say so (you do not need to show work). But if it is incorrect, say what will go wrong (e.g., infinite recursion, incorrect answer), give an example of an input that will demonstrate the error, and briefly explain *why* the code does not work correctly. A description of what the code *should* do is contained in the docstring of the function; if the code doesn't do this, it should be considered incorrect.

Hint: use the tips for analyzing recursive arguments from lecture.

a) In this part, adopt the convention that the sum of the numbers in an empty list is zero.

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

b) In this part, you may assume that the input list is not empty.

```python
def product(numbers):
    """Should return the product of all elements in numbers."""
    if len(numbers) == 1:
        return numbers[0]
    return numbers[0] * product(numbers[1:])
```

c) You may assume that `start` and `stop` are valid indices.

```python
def inplace_reverse(l, start, stop):
    """After running this, l[start:stop] should be reversed."""
    if stop - start <= 1:
        return
    l[start], l[stop-1] = l[stop-1], l[start] # swap first and last
    inplace_reverse(l, start, stop-1)
```

d) You may assume that `arr` is non-empty.

```python
def find_mode(arr):
"""Should return the mode of arr, along with its frequency."""
    if len(arr) == 1:
        return arr[0], 1
    middle = math.floor(len(arr) / 2)
    left_mode, left_freq = find_mode(arr[:middle])
    right_mode, right_freq = find_mode(arr[middle:])
```

```python
    if left_freq > right_freq:
        return left_mode, left_freq
    else:
        return right_mode, right_freq
```

Problem 2. *(Eligible for Redemption)*

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)
```python
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:])
```

b)
```python
import math
def summation_2(numbers, start, stop):
    """Returns the sum of numbers[start:stop]"""
    if stop <= start:
        return 0
    if stop - start == 1:
        return numbers[start]
    left_ix = math.floor(start + (stop - start) / 3)
    right_ix = math.floor(start + 2 * (stop - start) / 3)
    left_sum = summation_2(numbers, start, left_ix)
    middle_sum = summation_2(numbers, left_ix, right_ix)
    right_sum = summation_2(numbers, right_ix, stop)
    return left_sum + right_sum
```

c) In this problem, remember that // performs *flooring division*, so the result is always an integer. For example, 1//2 is zero. `random.randint(a,b)` returns a random integer in $[a, b)$ in constant time.

```python
import random
def foo(n):
    """This doesn't do anything meaningful."""
    if n == 0:
        return 1

    # generate n random integers in the range [0, n)
    numbers = []
    for i in range(n):
        number = random.randint(1, n)
        numbers.append(number)

    x = sum(numbers)
    return (foo(n//3) + foo(n//3) + foo(n//3)) / x**.5
```

Problem 3.

In this problem, we will show that the average case time complexity of binary search is $\Theta(\log n)$.

a) The number of calls to `binary_search` required to find a target depends on the target's position within the array. For instance, if the target is in the exact middle of the array, only one call to binary search is necessary. If the target is elsewhere, more calls are required.

The graphic below shows 15 blanks, one for each element of a 15-element array. In each blank, write the number of calls needed to `binary_search` in order to find the target if the target were located at that position in the array. Several of the blanks have been filled in for you.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
___	___	___	2	___	___	___	1	___	___	___	2	___	___	___

b) Given an array of size n, what is the greatest number of calls to `binary_search` necessary to find a target in the array? Your answer should be a formula involving n. You should assume that the target is in the array, and that $n = 2^k - 1$ for some $k \in \{1, 2, 3, \ldots\}$; that is, $n \in \{1, 3, 7, 15, \ldots\}$.

c) Given an array of size n, where $n = 2^k - 1$ for some k, let $f_n(k)$ be the number of elements of the array which require exactly k calls to `binary_search` before they are found. Derive a formula for $f_n(k)$.

d) Define

$$S(n) = \sum_{k=1}^{\log_2(n+1)} k \cdot 2^{k-1}.$$

Show that $S(n) = O(n \log n)$.

Hint 1: show that $S(n)$ is smaller than something that is $\Theta(n \log n)$. The following fact might help:

$$\sum_{k=1}^{\log(n+1)} k \cdot 2^{k-1} \leq \sum_{k=1}^{\log(n+1)} \log(n+1) \cdot 2^{k-1}$$

Hint 2: You may need to remember the formula for the sum of a geometric progression from calculus:

$$\sum_{k=0}^{K} x^k = \frac{1 - x^{K+1}}{1 - x}$$

e) Define

$$S(n) = \sum_{k=1}^{\log(n+1)} k \cdot 2^{k-1}.$$

Show that $S(n) = \Omega(n \log n)$.

f) What is the average case time complexity of binary search? Assume that the target is in the array exactly once and that the probability of it being in any given position is simply $1/n$. shuffled randomly before the first call. You may also assume that the size of the array is $2^k - 1$ for some $k \in 1, 2, 3, \ldots$.

Hint: since the time taken during any call to `binary_search`, excluding time spent in recursive calls, is constant, the total time taken by binary search is proportional to the number of calls made.

Programming Problem 1.

You are performing a massive study of Twitter data and have recorded over 1 billion tweets. In your analysis, you frequently need to count the number of tweets posted after time a but before time b. A linear time algorithm is not fast enough for your purposes.

In a file named `logcount.py`, write a function named `logcount(arr, a, b)` which takes in a *sorted* array `arr` and returns the number of elements of the array which are in the closed interval `[a, b]`. You may assume that the elements of the array are numbers, but you should not assume that the elements are distinct (there may be duplicates). To receive credit, your algorithm must have a worst case time complexity of $\Theta(\log n)$, where n is the size of the array.

Note: This is our first *programming problem* of the quarter. Programming problems will appear separate from the "main" homework on Gradescope and are (mostly) autograded. Upon submitting your code, make sure that the autograder passes *all* of the tests – it is checking to make sure that your file is named correctly, that the function runs, etc. However, passing all of these tests **does not** guarantee that you will receive full credit! After the homework due date, we will upload a more sophisticated autograder that will thoroughly check your code. A human grader will also quickly check your code to make sure that it adheres to any constraints imposed in the problem statement. Your code is allowed to import any of the built-in Python modules (such as `math`), as well as numpy (if you should need it).