

---

## 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.

```
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:])
```

**Solution:** This is missing a base case for when the array is of size 1. If an input of size one is given to this function, it will make a recursive call on an empty array and an array of size one. This recursive call will make the same recursive calls, *ad infinitum*. In other words, this will recurse infinitely.

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

```
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:])
```

**Solution:** This is correct.

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

```
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)
```

**Solution:** The arguments to the recursive call are wrong: they should sort the middle part of the list (excluding the first and last element), but they include the first element. This causes incorrect output. For instance, `inplace_reverse([4,5,2,3], 0, 4)` results in [5,2,3,4].

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

```
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:])
    if left_freq > right_freq:
        return left_mode, left_freq
    else:
        return right_mode, right_freq
```

**Solution:** This will return the incorrect answer. Assuming that recursive calls work correctly, the algorithm may still not produce the right result. For instance, suppose our input is: [1, 1, 1, 2, 2, 3, 3, 3, 2, 2]. The mode is clearly 2, but the recursive calls will find 1 and 3 for the mode in each half.

In short, the overall mode is not necessarily the mode of the left half or the right half.

### 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) `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:])
```

**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):
    """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 + middle_sum + right_sum

```

**Solution:** No slicing is being done, so a constant amount of time is required outside of the recursive calls. Therefore the recurrence relation is:

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

We have not seen this recurrence before, so we must solve it by unrolling. It isn't so different from  $T(n) = 2T(n/2) + 1$ , which you may have seen in lecture or in discussion. Both have  $\Theta(n)$  as their answer.

- 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.

```

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

```

**Solution:** The work outside of the recursive calls takes linear time. There are three recursive calls, each on problems of size (roughly)  $n/3$ . Therefore the recurrence is:

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

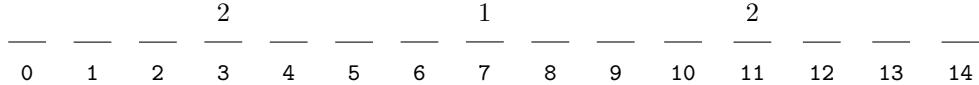
We can solve this recurrence by unrolling it. We'll find that the solution is  $\Theta(n \log n)$ . Note the similarity between this recurrence and that for mergesort, and that both have the same solution.

### 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.



- 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  $k \in \{1, 2, 3, \dots\}$ ; that is,  $n \in \{1, 3, 7, 15, \dots\}$ .

**Solution:** If  $n = 1$ , only one call is necessary.

If  $n = 3$ , two calls might be necessary. Note that this is  $\log_2(3 + 1)$ .

If  $n = 7$ , three calls might be necessary. Note that this is  $\log_2(7 + 1)$ .

Apparently,  $\log_2(n + 1)$  calls are necessary.

- 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  $f_n(k)$ .

**Solution:** Only one element requires just one call: the middle element. So  $f_n(1) = 1$ .

Two elements can be found with exactly two calls, so  $f_n(2) = 2$ .

Four elements can be found with exactly three calls, so  $f_n(3) = 4$ .

It looks like the number of elements is doubling with each step. Here's a table capturing the pattern.

$k$	$f_n(k)$
1	1
2	2
3	4
4	8
5	16
$\vdots$	$\vdots$

So it looks like  $f_n(k) = 2^{k-1}$ .

- 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:

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

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

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

**Solution:**

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

We'll try to get an upper bound using hint 1:

$$\begin{aligned} &\leq \sum_{k=1}^{\log_2(n+1)} \log_2(n+1) \cdot 2^{k-1} \\ &= \log_2(n+1) \sum_{k=1}^{\log_2(n+1)} 2^{k-1} \end{aligned}$$

We recognize the sum of a geometric progression and want to use the formula in hint 2 above, but this sum starts at  $k = 1$  instead of  $k = 0$ . We can start the sum at  $k = 0$  by reindexing:

$$= \log_2(n+1) \sum_{k=0}^{\log_2(n+1)-1} 2^k$$

Using the formula for the sum of a geometric progression:

$$\begin{aligned} &= \log_2(n+1) \left[ \frac{1 - 2^{\log_2(n+1)}}{1 - 2} \right] \\ &= \log_2(n+1) (2^{\log_2(n+1)} - 1) \end{aligned}$$

You can convince yourself that  $\log_2(n+1) = \Theta(\log_2 n)$  and  $2^{\log_2(n+1)} = \Theta(2^{\log_2 n}) = \Theta(n)$  using any of the techniques that we saw when learning about asymptotic notation, such as limits.

$$\begin{aligned} &= \Theta(\log n) \cdot \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

e) Define

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

Show that  $S(n) = \Omega(n \log_2 n)$ .

**Solution:** If we drop all but the last term from the sum, we make it smaller. This gives us the lower bound we need:

$$\begin{aligned} S(n) &= \sum_{k=1}^{\log_2(n+1)} k \cdot 2^{k-1}. \\ &\geq \log_2(n+1) \cdot 2^{\log_2(n+1)} \\ &= \Theta(\log n) \cdot \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

- 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  $k \in 1, 2, 3, \dots$

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.

**Solution:** The average case time complexity can be found by computing the average number of calls to `binary_search` needed, assuming that the target is placed uniformly at random within the array.

The first case is that only one call is necessary. There is only  $f_n(1) = 1$  place where the target can be for this to happen, and this occurs with probability  $1/n$ .

The second case is that exactly two calls are necessary. There are  $f_n(2) = 2$  places where the target can be for this to happen, so this occurs with probability  $2/n$ .

The third case is that exactly three calls are necessary. There are  $f_n(3) = 4$  places where the target can be for this to happen, so this occurs with probability  $4/n$ .

In general, case  $k$  is when exactly  $k$  calls are necessary. There are  $f_n(k) = 2^{k-1}$  places where the target can be for this to happen, so this occurs with probability  $2^{k-1}/n$ .

In total, there are  $\log_2(n + 1)$  cases.

The average number of calls needed is therefore:

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

From the previous parts, we know that the summation is both  $O(n \log n)$  and  $\Omega(n \log n)$ , and so it is  $\Theta(n \log n)$ :

$$\begin{aligned} &= \frac{1}{n} \cdot \Theta(n \log n) \\ &= \Theta(\log n) \end{aligned}$$

This shows that the average case time complexity is  $\Theta(\log n)$ .

### 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). If the array is empty, you should return zero. If the interval  $[a, b]$  is empty (as in the case when  $a > b$ , you should also return 0. To receive credit, your algorithm must have a worst case time complexity of  $\Theta(\log n)$ , where  $n$  is the size of the array.

Example: if `arr = [2, 3, 5, 5, 6, 9, 12]`, `a = 3.7`, and `b = 6.1`, your function should return 3 since there are three elements within the closed interval  $[3.7, 6.1]$ ; namely: 5, 5, 6.

**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)).