
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!

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: $\sum_{p=0}^N b^p = \frac{1-b^{N+1}}{1-b}$

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)

```

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 $\lim_{n \rightarrow \infty} f(n)/n = 0$.

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.

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.