DSC 40B - Homework 03

Due: Tuesday, October 22

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

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.

One of the most used tools in the data scientist's toolkit is the *histogram*. We often use it to get a sense of the distribution of a dataset. You'll remember them from DSC 10, where we used them quite a bit. In this problem, you'll be asked to design and implement an efficient algorithm that computes a histogram from sorted data.

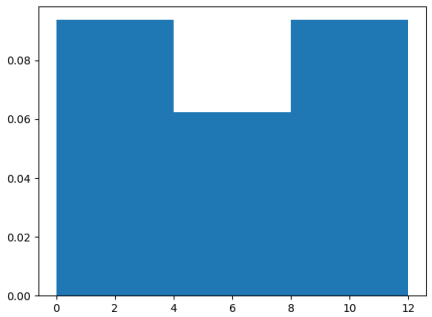
Computing a histogram. We saw how to compute a density histogram in DSC 10, but here's a quick reminder. Suppose you're given a list of n points, along with k bins of the form $[a, b)$, where a is the start of the bin and b is the end (we assume that a is included in the bin, but b is not). The density within a given

bin is calculated using the formula:

$$\frac{\text{\# of points in the bin}}{(\text{total \# of points}) \times (\text{bin width})}$$

For example, suppose we are given 8 data points: $(1, 2, 3, 6, 7, 9, 10, 11)$ and 3 bins: $[0, 4), [4, 8)$, and $[8, 12)$. Three of the eight points fall in the first bin, whose width is 4. The density in the first bin is therefore $\frac{3}{8\times 4} = 0.09375$. On the other hand, only two points fall into the second bin, so the density there is $\frac{2}{8\times 4} = 0.0625$. Three points fall into the last bin, so the density there is also $\frac{3}{8\times 4} = 0.09375$.

We can visualize these densities using a plot like the one shown below:



The problem. In a file named `histogram.py`, write a function called `histogram(points, bins)` which accepts two arguments:

- `points`: a **sorted** Python list of n numbers, in order from smallest to largest
- `bins`: a Python list of k tuples, each of the form (a, b), where a and b are the start and end of a bin, respectively. You can assume that the bins are also sorted from left to right, and the endpoint of one bin is the start of the next. For example, the list `[(0, 4), (4, 8), (8, 12)]` denotes bins of $[0, 4), [4, 8)$, and $[8, 12)$. Each bin includes its start, but not its end.

The output of your function should be a Python list which contains the histogram's density within each bin. So if `bins` has k elements, your function should return a list of k numbers.

For example, here's what your code should do on a simple input:

```
>>> points = [1, 2, 3, 6, 7, 9, 10, 11]
>>> bins = [(0, 4), (4, 8), (8, 12)]
>>> histogram(points, bins)
[0.09375, 0.0625, 0.09375]
```

Your code will be tested not only for correctness, but also for efficiency. To get full credit, your function should take $\Theta(n)$ time, where n is the number of data points. It should take this time regardless of the number of bins! Note that there is a simple, inefficient solution that will take $\Theta(n \times k)$ time (for instance, if the number of bins is $n/10$, it would take $\Theta(n^2)$ time!). Your code should be much faster than that, and it should only need to loop through the data points *once*. *Hint*: make good use of the fact that the data and the bins are given to your function in **sorted** order.

For this problem, you may not use `numpy` or any other imports. You can do this in pure Python! You may assume that the number of bins is between 1 and n, and that each of the data points falls into one of the bins.

About the autograder. This is the first "Programming Problem" of the quarter. You'll submit your code to Gradescope in a separate assignment named "Homework 03 - Programming Problem 01" with the same due date as the written homework. If you want to use a slip day on this homework, you only need to use one for both the written and programming problems.

Programming problems are mostly autograded, which means that your code will need to pass several tests to receive full credit. Some of these tests are *public*, meaning that you can see the result before the deadline. They usually check to make sure that your code has the correct filename and works on the simplest of examples. The private tests check your code on more complex examples, and usually will make sure that it has the correct time complexity. You can submit your code as many times as you'd like, and we suggest submitting it early and often! After uploading your code, be sure to stick around and see if it passes the public tests. If it doesn't, you might lose points.