DSC 40B - Homework 03

Due: Wednesday, April 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 find_max(numbers):
    """Given a list, returns the largest number in the list.

    Remember: slicing a list performs a copy, and so takes linear time.
    """
    n = len(numbers)
    if n == 0:
        return 0
    if n == 1:
        return numbers[0]
    mid_left = math.floor(n / 3)
    mid_right = math.floor(2 * n / 3)

    return max(
        find_max(numbers[:mid_left]),
        find_max(numbers[mid_left:mid_right]),
        find_max(numbers[mid_right:])
    )
```

Solution: Note that the slicing in `numbers[:mid_left]` (and so on) takes linear time in the length of the array, so the recurrence relation is:

$$T(n) = \begin{cases} \Theta(n) + 3T(n/3) & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

This is the same as the recurrence relation as for 3-way mergesort, and the solution is $\Theta(n \log n)$.

b)

```
import math
def find_max_again(numbers, start, stop):
    """Returns the max of numbers[start:stop]"""
    if stop <= start:
        return 0
    if stop - start == 1:
        return numbers[start]

    middle = math.floor((start + stop) / 2)
```

```
    left_max = find_max_again(numbers, start, middle)
    right_max = find_max_again(numbers, middle, stop)

    return max(left_max, right_max)
```

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

$$T(n) = \begin{cases} \Theta(1) & \text{stop} \leq \text{start or stop} - \text{start} = 1 \\ 2T(n/2) + \Theta(1) & \text{otherwise} \end{cases}$$

We have not seen this recurrence before, so we must solve it by unrolling. For simplicity, assume $T(n) = 2T(n/2) + 1$. If we perform unrolling k times, we will get the following:

$$T(n) = 2^k T(n/2^k) + 1 + 2 + 4 + \cdots + 2^{k-1}.$$

This unrolling stops when we have $n/2^k = 1$, meaning $2^k = n$ and $k = \log_2 n$. Furthermore, note that $1 + 2 + 4 + 8 + \cdots + 2^{k-1} = 2^k - 1$ (geometric sum). We thus have

$$T(n) = nT(1) + 2^k - 1 = n \cdot \Theta(1) + n - 1 = \Theta(n).$$

It will have $\Theta(n)$ as its solution.

Notice that this recurrence is similar to the recurrence for binary search, which was $T(n/2)+\Theta(1)$, and whose solution was $\Theta(\log n)$. The difference between the two is the factor of 2 on the $T(n/2)$; binary search does not have this because it only makes one recursive call. This difference has a big effect: it reduces the time complexity from $\Theta(n)$ all the way down to $\Theta(\log n)$.

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. Assume `isEven()` takes in an integer and returns whether it is even in constant time. Note that you are asked to determine the **worst-case** time complexity of the following algorithm.

```
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)
    if isEven(x):
        return foo(n//2) / x**.5
    else:
        return foo(n//2) * x
```

Solution: The work outside of the recursive calls takes linear time. (Note that while there are random numbers generated, the time complexity for outside the recursive calls are deterministic.)

Depending on whether x is even or not, the algorithm will make **one** recursive call on problem of size $n/2$. Therefore the recurrence is:

$$T(n) = \begin{cases} T(n/2) + \Theta(n) & n > 0 \\ \Theta(1) & n = 0 \end{cases}$$

We can solve this by unrolling it. In particular, for simplicity, assume $T(n) = T(n/2) + cn$. Applying unrolling k times, we have have the following:

$$T(n) = T(\frac{n}{2^k}) + c\frac{n}{2^{k-1}} + c\frac{n}{2^{k-2}} + \cdots + c\frac{n}{2} + cn.$$

The unrolling stops when we have $n/2^k = 1$, meaning that $2^k = n$ and $k = \log_2 n$. Furthermore,

$$n + n/2 + \cdots n/2^{k-1} = n(1 + 1/2 + 1/4 + \cdots + 1/2^{k-1}) = n \cdot \Theta(1) = \Theta(n).$$

It then follows that for $k = \log_2 n$, we have that

$$T(n) = T(\frac{n}{2^k}) + c\frac{n}{2^{k-1}} + c\frac{n}{2^{k-2}} + \cdots + c\frac{n}{2} + cn = \Theta(1) + \Theta(n) = \Theta(n).$$

Therefore $T(n) = \Theta(n)$.

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.

Solution: Go up to step $\sqrt{n}$ and drop the first bulb. If it doesn't break, go up to step $2\sqrt{n}$ and drop. Repeat until the bulb breaks. Say it breaks at step $k\sqrt{n}$. Then you only need to check steps $(k-1)\sqrt{n}$ through $k\sqrt{n}$, which you can do with a linear search in less than $\sqrt{n}$ steps.

In the worst case, $n_{\max} = n - 1$. In this case, you'll drop at $\sqrt{n}$, then $2\sqrt{n}$, then $3\sqrt{n}$ and so on, all the way to the top. It takes $n/\sqrt{n} = \sqrt{n}$ drops to get to the top. Then the bulb finally breaks, and you have to do a linear search to find $n_{\max}$ exactly – it could be anywhere between the top rung, n, and the location of the previous drop, $n - \sqrt{n}$. This linear search takes about $\sqrt{n}$ drops. The total number of drops is $\sqrt{n}$.

A concrete example might make this clearer. Say $n = 100$. Then start on step 10 and drop the bulb. Say it doesn't break. Move to step 20 and drop the first bulb again. Still doesn't break. Now drop from 30 – supposed it breaks. $n_{\max}$ is between 20 and 30, so drop the remaining bulb from 21, 22, 23, etc. until it breaks. In the worst case, you drop the bulb from $10, 20, 30, \ldots, 90, 100$, and it breaks at 100, so you then drop from $91, 92, \ldots, 99$.

Note that $\sqrt{n}$ is (close) to the optimal solution, but there are infinitely many others. For example, you could drop from $\log n$, then $2 \log n$, etc. Here the worst case would be $\Theta(n/\log n)$, which is still better than $\Theta(n)$. Similarly, you could drop at $n^{1/4}$, etc.

The most efficient solution is determined by the number of bulbs you have. If you have $\Theta(\log n)$ bulbs, the most efficient solution is binary search.

Programming Problem 1.

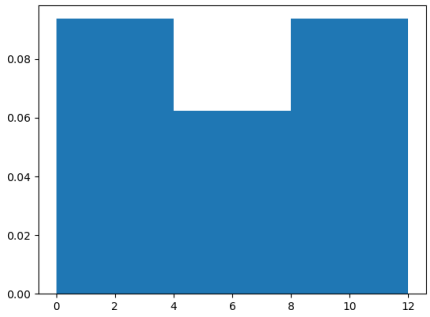
One of the most used tools in the data scientist's toolkit is the *histogram*. We often use it to get a sense of the distribution of a dataset. You'll remember them from DSC 10, where we used them quite a bit. In this problem, you'll be asked to design and implement an efficient algorithm that computes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

Solution: First, let's look at the simplest solution. It will return the right densities, but it is inefficient.

It looks like this:

```
def slow_histogram(points, bins):
    """This is a solution that has the incorrect time complexity."""
    result = []
    n = len(points)

    for bin_start, bin_end in bins:
        bin_count = 0
        for point in points:
            if bin_start <= point < bin_end:
                bin_count += 1
        result.append(bin_count / (n * (bin_end - bin_start)))

    return result
```

That solution has a nested loop which, for each bin, loops through the data points, counting the number of points in the bin. This means that we loop through the data once for every bin, or a total of $n \times k$ times. For example, if the number of bins is $n/10$, the time complexity of this code is $\Theta(n^2)$. This is not efficient, and we can do better!

The above solution does not take advantage of the fact that the data and the bins are sorted. Here's a solution that does:

```
def histogram(points, bins):
    """Efficiently computes a histogram.

    Assumes that both `points` and `bins` are sorted in ascending order to
    avoid looping through all bins for each point.

    """
    result = []
    n = len(points)

    point_ix = 0

    for bin_start, bin_end in bins:
        bin_count = 0
        while point_ix < len(points) and points[point_ix] < bin_end:
            point_ix += 1
            bin_count += 1
        result.append(bin_count / (n * (bin_end - bin_start)))

    return result
```

This code also has a nested loop, but there's a key difference. For each bin, we start iterating through the data points, but stop when we see a data point past the end of the bin. The key observation is that, for the next bin, we can start where we left off previously, with the first data point that was past the end of the previous bin. This means that we only loop through the data once, and the time complexity is $\Theta(n)$, as required, no matter how many bins there are.

Programming Problem 2.

In a file named `swap_sum.py`, write a function named `swap_sum(A, B)` which, given two **sorted** integer arrays `A` and `B`, returns a pair of indices `(A_i, B_i)` – one from A and one from B – such that after swapping these indices, `sum(B) == sum(A) + 10`. If more than one pair is found, return any one of them.

If such a pair does not exist, return `None`.

For example, suppose `A = [1, 6, 50]` and `B = [4, 24, 35]`. Swapping 6 and 4 results in arrays $(1, 4, 50)$ and $(6, 24, 35)$; the elements of each list sum to 55 and 65. Thus, you must return `(1, 0)` as you are expected to return the indices.

Your algorithm should run in time $\Theta(n)$, where n is the size of the larger of the two lists. Your code should not modify `A` and `B`.

Hint: This is similar to the movie problem from lecture, but also to a problem covered in discussion. In the discussion problem, we're given two lists, `A` and `B` and a target `t`, and our goal is to find an element a of `A` and an element b of `B` such that $a + b = t$. This problem is similar, in that we want (sum of A after swap) − (sum of B after swap) $= -10$. Note that here we are subtracting, where in discussion we were adding! How does that change things?

This is a coding problem, and you'll submit your `swap_sum.py` file to the Gradescope autograder assignment named "Homework 03 - Programming Problem 01". The public autograder will test to make sure your code runs without error on a simple test, so be sure to check its output! After the deadline a more thorough set of tests will be used to grade your submission.

Solution:

```
def swap_sum(A, B):
    # what we want sum(A) - sum(B) to equal
    target_difference = -10

    A_i, B_i = 0, 0

    # we compute these once to avoid having to recompute them again and again
    # later -- that would take linear time per call to `sum`, but the sum isn't
    # changing...
    sum_A, sum_B = sum(A), sum(B)

    while A_i < len(A) and B_i < len(B):
        sum_A_after_swap = sum_A - A[A_i] + B[B_i]
        sum_B_after_swap = sum_B + A[A_i] - B[B_i]
        array_diff = sum_A_after_swap - sum_B_after_swap
        if array_diff == target_difference:
            return (A_i, B_i)
        elif array_diff < target_difference:
            # sum(A) - sum(B) was too small! we have a choice: increase A_i or
            # increase B_i. Increasing A_i will mean that the element of A we
            # swap into B will be larger than before, so B will get bigger and
            # A will get smaller. This would mean *decreasing* the difference
            # in sums. So instead we increase B_i
            B_i += 1
        else:
            # The difference was too big! Increasing A_i will decrease the
            # difference, so is the right choice.
            A_i += 1
    return None
```