DSC 40B - Homework 03

Due: Wednesday, April 24

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(2n / 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:

$$T(n) = \begin{cases} \Theta(n) + 3T(n/3) & n \geq 2 \\ \Theta(1) & n = 1 \text{ or } n = 0 \end{cases}$$

Recurrence Relation:

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

$$
\begin{aligned}
T(n) &= 3T\left(\frac{n}{3}\right) + n && \text{(first unrolling } k = 1\text{)} \\
&= 3\left[3T\left(\frac{n}{9}\right) + n/3\right] + n \\
&= 9T\left(\frac{n}{9}\right) + 2n && \text{(second unrolling } k = 2\text{)} \\
&= 9\left[3T\left(\frac{n}{27} + \frac{n}{9}\right)\right] + 2n \\
&= 27T\left(\frac{n}{27}\right) + 3n && \text{(third unrolling } k = 3\text{)} \\
&\vdots \\
&= 3^k T\left(\frac{n}{3^k}\right) + kn && (k\text{-th unrolling})
\end{aligned}
$$

$T\left(\frac{n}{3^k}\right)$ becomes $T(1)$ when $\frac{n}{3^k} = 1$ which implies $k = \log_3(n)$. Therefore,

$$
\begin{aligned}
T(n) &= 3^{\log_3(n)} T(1) + n\log_3(n) \\
&= nT(1) + n\log_3(n) = \Theta(n \log n)
\end{aligned}
$$

b)

```python
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:

$$T(n) = 2T(n/2) + \Theta(1)$$

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. 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 = []
    i = 0
    while i < n:
        i = i + 2
        number = random.randint(1, n)
        numbers.append(number)

    x = sum(numbers)

    if x is even:
        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 after the `while` loop the length of `numbers` is about $n/2$, hence `sum(numbers)` take $\Theta(n)$ time. Moreover, 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:

$$T(n) = T(n/2) + \Theta(n)$$

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.

A *rotated sorted array* is an array that is the result of taking a sorted array and moving a contiguous section from the front of the array to the back of the array. For example, the array `[5,6,7,1,2,3,4]` is a rotated sorted array: it is the result of taking the sorted array `[1,2,3,4,5,6,7]` and moving the first 4 elements, `[1,2,3,4]`, to the back of the array. Sorted arrays are also rotated sorted arrays, technically speaking, since you can think of a sorted array as the result of taking the sorted array and moving the first 0 elements to the back.

For example, all rotated version of `[1,2,3,4,5]` is the following:

- `[1,2,3,4,5]`
- `[5,1,2,3,4]`
- `[4,5,1,2,3]`
- `[3,4,5,1,2]`
- `[2,3,4,5,1]`

The function below attempts to find the value of the minimum element in a rotated sorted array. It is given the array `arr` and the indices `start` and `stop` which indicate the range of the array that should be searched. You may assume the numbers in `arr` are **unique**.
Fill in the blanks to make the function work correctly. Your function should have time complexity $\Theta(\log n)$.

```
import math
def find_min(arr, start, stop):
    """Searches arr[start:stop] for the samllest element.

    Assumes arr is a rotated sorted array.
    """
    if ------: # write doum appropriate base case;
        return -----

    # you may include more than one base case if needed

    mid = math.floor((stop + start) / 2)

    if ----------------:

        return arr[---------]

    elif ------------------ :

        return find_min(----------------)

    else:

        return find_min(----------------)
```

Solution:

```
import math
def find_min(arr, start, stop):
    """Finds the minimum element in a rotated sorted arr[start:stop] using recursion.
```

```
    You can assume that the array contains no duplicates and that start < stop.

    Args:
        arr (list): rotated array of size containing no duplicates
        start (int): first index of the subarray (inclusive)
        stop (int): last index of the subarray (exclusive)

    Returns:
        int: minimum element in the subarray in arr[start:stop]
    """

    # If the list has one element, then the minimum element is the only element
    if stop - start == 1:
        return arr[start]

    mid = math.floor((start + stop) / 2)

    # Edge case, such as arr = [2, 1]
    # Note that if the element before mid is > mid, then it means that
    # mid has to be the smallest element. This is essentially the "inflection point"
    if arr[mid - 1] > arr[mid]:
        return arr[mid]

    # If the middle element is > than the last element, ar
    # then the minimum must be in the right half, since the "inflection point"
    # is to the right
    elif arr[mid] > arr[stop - 1]:
        return find_min(arr, mid + 1, stop)

    # If the middle element is < than than the last element, then the right subarray
    # must be in sorted ascending order. Thus, the minimum must reside in arr[start:mid+1]
    else:
        return find_min(arr, start, mid)
```

Programming Problem 1.

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`.

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:

```python
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
```