SI 335 Spring 2015: Problem Set 3

Due: Monday, April 6

Your scheduled presentation time:

Group member:

Group member:

Group member:

Group member:

Instructions: Review the course honor policy: you may not use any human sources outside your group, and must document anything you used that's not on the course webpage.

This cover sheet must be the front page of what you hand in. Use separate paper for the your written solutions outline and make sure they are neatly done and in order. Staple the entire packet together.

Comments or suggestions about this problem set:

Comments or suggestions about the course so far:

Citations (be specific about websites):

Grading rubric:

A: Solution meets the stated requirements and is completely correct. Presentation is clear, confident, and concise.
B: The main idea of the solution is correct, and the presentation was fairly clear. There may be a few small mistakes in the solution, or some faltering or missteps in the explanation.
C: The solution is acceptable, but there are significant flaws or differences from the stated requirements. Group members have difficulty explaining or analyzing their proposed solution.
D: Group members fail to present a solution that correctly solves the problem. However, there is clear evidence of significant work and progess towards a solution.
F: Little to no evidence of progress towards understanding the problem or producing a correct solution.

Problem	Final assessment
1	
2	
3	
4	

1 FrankenSort

Behold, your professor's most beautiful creation:

```
def frankenSort(A):
    n = len(A)
    m = ceil(n/4)

    if n <= 5: # base case
        selectionSort(A)

    else:
        A[0 : 2*m] = frankenSort(A[0 : 2*m]) # sort first half
        A[2*m : n] = frankenSort(A[2*m : n]) # sort second half

        for i in range(m, 2*m): # swap middle parts
            swap(A, i, i+m)

        A[0 : 2*m] = frankenSort(A[0 : 2*m]) # sort first half again
        A[2*m : n] = frankenSort(A[2*m : n]) # sort second half again
        A[m : 3*m] = frankenSort(A[m : 3*m]) # sort the middle half

    return A
```

It's a sorting algorithm that makes multiple recursive calls on different sub-arrays of the original array.

a) Explain how and why this algorithm works. You should go through a reasonable-sized example on paper. Think about where every element in the array *belongs* in the sorted order, and then how do you know the algorithm will get it there.

b) Determine the worst-case running time of this algorithm. You should write a recurrence relation and then solve it using MMA.

c) I want to know **exactly** how many swaps are performed by this algorithm. There are three places where you have to count swaps: in the call to `selectionSort` in the base case, inside the for loop on line 13, and in the recursive calls to `frankenSort`.

Fortunately the number of swaps performed by `selectionSort` on a size-n input is a simple function of n - look back to the solutions to the last problem on Problem Set 1 if you forget.

The number of swaps performed by `frankenSort` depends only on the size of the input, n. So we can define a function $s(n)$ to be the total number of swaps - including all recursive calls - performed on step 13 for an input of length n.

Here is a table for the first few values of $s(n)$:

n	$s(n)$
1	0
2	1
3	2
4	3
5	4
6	13
7	15
8	17
9	46
10	48

n	$s(n)$
50	1863
100	9340

Come up with an algorithm to determine $s(n)$, given the input size n. Try to make your algorithm as fast as possible.

As a starting point, you could imagine just using the original `frankenSort` algorithm and adding a line inside the `swap` helper function that increments a global variable called `count`. That will work, but have the same running time as the algorithm itself. You should be able to come up with something (much) faster.

d) Analyze your algorithm from part (c) and determine a big-O bound on the worst-case running time.

e) Use your algorithm to compute a few values of $s(n)$ for me. You can do it by hand or write an implementation of your algorithm; I don't care either way. Try and compute:

- $s(500)$
- $s(1000)$
- $s(12345)$
- s(each of your alpha number(s))
- s(each of your alpha number(s) cubed)

(The last one is a fairly large number. You can just write down the last 5 digits if you like.)

2 Road Trip Planning

2.1 Scenario

You are planning a road trip that will follow a certain route and stop in a bunch of towns along the way. You have determined how many days you'd like to spread the trip over, the exact sequence of towns (each of which has a place to sleep up for the night), and the distances between each consecutive pair of towns.

You want to break up the days of your road trip so that they are as balanced as possible. Now you know that the *average* distance per day will be the same regardless, since the total distance is already decided. Statistically, having a good balance means minimizing the *variance* of the distances per day, which is the same as minimizing the sum of the squares of the distances travelled in each day. If you've heard of something called a "least squares fit", that's essentially what we're doing.

Therefore you are going to find which way of splitting up the days will minimize the sum of the squares of the distances travelled each day. For example, if your journey has 4 legs of 2, 5, 3, and 3 miles (in that order), and you have only 2 days for the trip, you have the following options:

```
Option A: 0 miles day 1, (2+5+3+3) miles day 2
  Sum of squares: 0^2 + 13^2 = 169

Option B: 2 miles day 1, (5+3+3) miles day 2
  Sum of squares: 2^2 + 11^2 = 125

Option C: (2+5) miles day 1, (3+3) miles day 2
  Sum of squares: 7^2 + 6^2 = 85

Option D: (2+5+3) miles day 1, 3 miles day 2
  Sum of squares: 10^2 + 3^2 = 109

Option E: (2+5+3+3) miles day 1, 0 miles day 2
  Sum of squares: 13^2 + 0^2 = 169
```

In this example, the best choice would be to take the first two legs of the journey on day 1, and the last two legs on day 2.

2.2 Specifics

Your road trip itinerary has n legs, each represented by a numeric distance in miles. You have d days to complete the road trip.

So the input to your algorithm is a list of numbers of length n: $[x_1, x_2, \ldots, x_n]$

and the output is a d stopping points (each is an integer from 0 up to n): $[s_1, s_2, \ldots, s_d]$

such that the sum of squares

$$\left(x_1 + \cdots + x_{s_1}\right)^2 \quad + \quad \left(x_{s_1+1} + \cdots + s_{s_2}\right)^2 \quad + \quad \cdots \quad + \quad \left(x_{s_{d-1}+1} + \cdots + x_n\right)^2$$

is as small as possible. In the example above, we had $n = 4$, $d = 2$, and the distances $[x_1, x_2, x_3, x_4] = [2, 5, 3, 3]$ as input. The output optimal stopping points are $[s_1, s_2] = [2, 4]$.

2.3 Tasks

a) Devise an algorithm to determine the optimal road trip. Describe your algorithm in words, or using pseudocode, or both. It is *your job* to describe your algorithm as simply and as clearly as possible.

b) Analyze your algorithm and determine a big-Theta bound for the worst-case cost in terms of n and d.

2.4 Hints and Comments

- The most important thing is that your algorithm is CORRECT and clearly explained. Correctness in this case means that your algorithm *always* finds the optimal solution. This means that, if you think of a way of speeding up the algorithm, but you're not sure if it will still give the absolutely best solution every time, *don't do it*!
- Of course, more points will be awarded to faster algorithms. You should try to make your algorithm as fast as possible in every case, particularly the worst case.
- Planning this trip is sort of like putting parentheses around the distances, grouping together each day. Try to solve this problem in a similar way to what we did for the matrix chain multiplication problem in class.
- Lao-tzu said, "The journey of a thousand miles begins with a single step". Similarly, the road trip of n legs begins with a single day. You might want to structure your algorithm recursively by (at the top level) figuring out the optimal number of legs to take on day 1.
- I am not sure what the asymptotically optimal algorithm for this problem is. However, I assure you that it can be done in polynomial-time (*not* exponential) in terms of n and d. Do your best!

3 Party Planning version 1

This is the first of two problems related to planning a fantastic party, subject to two different kinds of constraints. **One of these problems has an greedy algorithm that always gives the optimal solution, and the other does not.** I'm not going to tell you which one is which.

Hint: Sometimes it's not about who you invite to the party, but who you *don't* invite to the party.

(I'm also not going to tell you which variant that hint applies to.)

For either problem that you do (or both), you need to come up with a greedy algorithm that is as good as possible, in terms of coming up as close as it can to getting the optimal solution. Then do the following:

a) Describe your greedy algorithm clearly and concicely, in words or in pseudocode.
b) Demonstrate how your algorithm works with a small but meaningful example.
c) Analyze the worst-case big-O running time of your greedy algorithm, in terms of the number of friends n and the number of relationships m. For full credit, your algorithm just needs to be polynomial-time. However, there will be a **tangible prize** for any idividual or group that gets the correct greedy algorithm with the best possible running time.
d) **Either** come up with a counterexample to demonstrate some input where your algorithm does not produce the optimal solution, **or** explain how you know that your algorithm always produces the optimal solution for any possible input.

3.1 Variant 1 description

Dave has n friends and he wants to invite as many as possible to his party. But some of his friends absolutely hate each other and would not have fun if the other person were invited.

Given the list of n friends, and then m pairs of friends that hate each other, Dave needs to find the largest number of friends to invite such that none of them hate each other.

4 Party Planning version 2

The instructions are the same as in the first variant. Remember, only one of these two variants has a greedy algorithm that always gives the optimal solution. Either way, you need to come up with a greedy algorithm that is as good as possible.

4.1 Variant 2 description

Eva has n friends and she wants them to have a great time at her party. Some of her friends know each other already, and some of them do not. (But none of them hate each other; they are nice people.) Eva has decided that everyone she invites should know at least 2 other people, and should also *not know* at least one other person at the party.

Given the list of n friends, and then m pairs of friends that already know each other, Eva needs to find the largest number of friends to invite so that all of them know at least 2 other invitees, and all of them do not know at least 1 other invitee.