SI 335 Spring 2015: Problem Set 1

Due: Wednesday, January 28

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 Sums of Squares

A "perfect square" is an integer multiplied by itself; the first few are 0, 1, 4, 9, 16, etc. Some integers can be written as the sum of two perfect squares. For example, $10 = 1^2 + 3^2$. But some cannot: for example, 7. And some can be written as the sum of 2 squares in more than one way: for example, $50 = 1^2 + 7^2 = 5^2 + 5^2$.

> (Caution: mathematical enrichment irrelevant to the assignment.) Integers that can be written as the sum of two squares have interested mathematicians for a long time. For example, Fermat showed that a prime number p is the sum of two perfect squares if and only if $p - 1$ is divisible by 4.

The following algorithm takes a given integer n and determines whether it can be written as the sum of two perfect squares. If so, it returns a and b such that $n = a^2 + b^2$. Otherwise, it returns "NO".

```
def sumsq(n):
    a, b = 0, n
    s = a*a + b*b
    while a <= b and s != n:
        if s < n:
            a = a + 1
        else:
            b = b - 1
        s = a*a + b*b
    if s == n:
        return (a, b)
    else:
        return 'NO'
```

a) Use a loop invariant to show that this algorithm is correct. State the invariant, then go through the three steps from class to show correctness.

b) Determine the worst-case running time of the algorithm. Give a Θ bound on the number of primitive operations, in terms of the input integer n, and simplify as much as possible. Show your work.

c) Develop an improved algorithm that solves the same problem. Present your new algorithm, briefly explain why it is correct (you do not have to do a formal proof with a loop invariant), and state the worst-case running time.

2 Meet and Greet

There are n strangers in a room. Everyone wants to get acquainted and meet everyone else. Assume that:

- It takes exactly one minute for a pair of people to meet each other.
- There is plenty of space in the room, and people can move instantly from one stranger to another.
- Each individual can only meet one other person at a time, but multiple meetings can be happening simultaneously in the room.

For example, if $n = 4$, if we call the people A, B, C, D, then you could have everyone meet everyone else in the following algorithm:

1) A and B meet, and C and D meet
2) A and C meet, and B and D meet
3) A and D meet, and B and C meet

At this point everyone has met, and the total time taken was 3 minutes.

Your task is to develop an efficient algorithm if there are any number n of people, not just 4. It doesn't have to match with my algorithm for $n = 4$; that's just an example to see how it could be done. Answer the following questions to get you on the right track:

a) When $n = 4$ above, there are a total of 6 meetings that take place. How many meetings are required for any n? Your answer should be a formula in terms of n.

b) Only two people can meet at once, so for example when $n = 4$ only 2 meetings can take place at once, and therefore the 3-minute solution above is the best possible for $n = 4$.

   Using this same logic, what would be the least possible number of minutes to meet for any value of n? (Again, your answer should be a formula in terms of n.)

c) Imagine you have split everyone up into two equal-sized groups G_1 and G_2. Everyone within G_1 has met everyone else in G_1, and the same goes for G_2, but no one from G_1 has met anyone from G_2 yet.

   Describe an algorithm for everyone in G_1 to meet everyone in G_2. If there are $n/2$ people in each group, your algorithm should take $n/2$ minutes.

   (Hint: look up how "Speed Dating" works.)

d) Come up with an efficient algorithm for everyone in the whole group of n people to meet each other. You might be able to use your algorithm from part (c) as a useful subroutine, but that's not required. (There are many efficient ways to solve this problem.)

   Try to get an exact formula in terms of n for how many minutes your algorithm will take. Ideally, your formula should match your lower bound from part (b), or at least it should have the same big-oh.

3 Coastal Search

You are on a ship and have lost your bearings. You have no means of navigation and no charts to follow. Fortunately, you can see land to the west, and you know that *somewhere* along this coastline is a friendly port. Unfortunately, you have no idea how far away the port is, or which direction up or down the coastline it is.

Your Captain's plan is to sail parallel to the shore until the port is found. The only question is how far to go in one direction before turning around, and then how far to go before turning around again, etc. Your ship is low on supplies so the Captain wants to find the port within a minimum total distance travelled. Since he knows you are an expert in algorithmic problem solving, the Captain asks you to devise the plan of how to search (how far to sail north, then south, then north, etc.).

For simplicity, assume that the shoreline is perfectly flat and extends forever in both directions (north and south). Also assume that you can only see one point on the shore, directly to the west of your ship, and you will know instantly when you see the port. Finally, the ship always turns around in-place and instantly. (In general, any exploits of my made-up scenario will not receive credit.)

In your algorithm, besides the usual pseudocode operations (variables, adding/subtracting/multiplying integers, loops, etc.), you can also call the following subroutines:

- `moveNorth(n)`: Sail north `n` miles
- `moveSouth(n)`: Sail south `n` miles
- `foundPort()`: Returns true if you have seen a port and can therefore quit.

a) Present an algorithm for the coastal search problem.

b) Analyze the worst-case cost of your algorithm. The *measure of difficulty* will be d, the distance (in miles) that the port city is from your starting point. The *cost measure* will be the total number of miles sailed. Give a big-O bound on the worst-case number of miles sailed, in terms of d. The best solutions will have worst-case cost $O(d)$.

   Important: this difficulty measure is not typical! Since there is no *input* to this algorithm, the measure of difficulty cannot be the input size. And the cost measure is also unusual - we're not counting primitive operations, but instead distance sailed.

c) Asymptotics are great and all, but in this case the "hidden constants" in the big-O really matter! If your algorithm means sailing half distance of mine to find the same city, then we should definitely use yours! Refine your analysis from (b) to give an exact upper bound on the number of miles sailed (*not* a big-O bound). There will be a **tangible prize** for the group who gets the smallest coefficient in front of d.

4 Swap counting

For this problem, you will study three sorting algorithms you should already be familiar with: InsertionSort, SelectionSort, and HeapSort. The first two of these are in the class notes for Unit 2, and here is the version of HeapSort that you can use for the purposes of this problem:

```
def heapSort(A):
    # this loop goes from i=n-1 down to i=0
    # This does the heapify operation
    for i in reversed(range(len(A))):
        bubbleDown(A, i, len(A))
    # This loop does the heap removals.
    for i in reversed(range(len(A))):
        swap(A, 0, i)
        bubbleDown(A, 0, i)
```

And here is a bubbleDown algorithm that you should use:

```
# Note: max-heap! The largest thing is at A[0]
def bubbleDown(A, start, end):
    while 2*start + 1 < end:
        child = 2*start + 1
        if child + 1 < end and A[child+1] > A[child]:
            child = child + 1
        if A[start] < A[child]:
            swap(A, start, child)
            start = child
        else:
            break
```

a) Tell me the worst-case running time of InsertionSort, SelectionSort, and HeapSort. You should know this already.

b) Analyze the number of *swaps* performed by a call to InsertionSort, in the worst case. To do this, you first have to describe a worst-case family of examples, i.e., what ordering of a list, for *any* size, makes InsertionSort do the maximum number of swaps.

Second, you need to analyze the number of swaps that InsertionSort does on your worst case example. This should be an exact function of n, like $3n^3 - 6$ or something like that. Try to be as precise as possible.

Finally, turn your formula into a Θ-bound on the number of swaps.

c) Repeat (b) for SelectionSort.

d) Repeat (b) for HeapSort. It will be a little more difficult to get the exact analysis of the number of swaps, so for this one you just need to give a formula for an *upper bound* on the number of swaps. But still, try to make your formula as precise as possible.

e) Show that *any* sorting algorithm that only uses swaps to move elements in the input array must always perform at least $\lceil n/2 \rceil$ swaps, in the worst case. Which of the three algorithms above comes closest to this lower bound?