SI 335 Spring 2013: Problem Set 1

Your name:

Due: Wednesday, January 23

Instructions: Review the course honor policy: informal *discussions* are permitted, but no *written collaboration* of any kind..

This cover sheet must be the front page of what you hand in. Fill out the left column in the table to the right before you hand it in.

Use separate paper for your solutions and make sure they are submitted in order — please!.

Comments or suggestions about this problem set:

Comments or suggestions about the course so far:

I had discussions with:

Citations (be specific about websites):

Grading rubric:

- **5**: Solution is completely correct, concisely presented, and neatly written.
- **4**: The solution is mostly correct, but one or two minor details were missed, or the presentation could be more concise.
- **3**: The main idea is correct, but there are some significant mistakes. The presentation is somewhat sloppy or confused.
- **2**: A complete effort was made, but the result is mostly incorrect. There may be some basic misunderstandings of the topic or the problem.
- **1**: The beginning of an attempt was made, but the work is clearly incomplete.
- **0**: Not submitted.

Problem	Self-assessment	Final assessment
1		
2		
3		

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: this paragraph may be 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.

   You are encouraged to think up and develop a useful loop invariant on your own. If you get stuck, Dr. Roche will give you a hint - just ask!

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 Average Air Temperatures

There is a weather station which reads the air temperature continuously and reports it at regular intervals, say k times every hour. These temperatures are stored in an array, and the meteorologists want a computer program to report all the hourly average temperatures.

Specifically, we have the following problem:

Input: An array A of n numbers, and an integer k
Output: $n-k+1$ numbers representing the hourly averages: $(A[i]+A[i+1]+\cdots+A[i+k-1])/k$ for $i = 0, 1, \ldots, n-k$.

a. Consider the following algorithm for this problem:

```
def avgtemps(A, k):
    for i in range(0, len(A)-k+1):
        s = 0
        for j in range(i, i+k):
            s = s + A[j]
        print(s/k)
```

Analyze the running time of this algorithm, in terms of n and k. Give a Θ bound on the worst-case running time.

b. Devise another algorithm for this problem with a better worst-case running time. Present your algorithm in pseudocode, and then give a worst-case running time analysis with a Θ-bound.

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 submission with the smallest coefficient in front of d.