SI 335 Spring 2012: Project 1

This project is due before the beginning of class on **Friday, January 27**. Late submissions will not be accepted, in accordance with the class policy. Students with excused absences on that date may email their assignment by the deadline *in addition to handing in a paper copy at the earliest opportunity.*

Guidelines:

- All work must be typed or neatly written
- Multiple-page submissions must be stapled or otherwise bound securely into a single packet.
- **Be concise**. Most problems have at least one simple and relatively short correct answer. Solutions will be graded not only on their correctness but also on how clearly that solution is presented.
- In general, there is a preference for **correct** algorithms over efficient ones. If the problem asks for a $O(n)$ algorithm, a *correct* $O(n \log n)$ algorithm is usually preferable over an incorrect one that claims $O(n)$ running time.
- **Do not try to pass off a solution you know is incorrect**. A statement to the effect of "I know this doesn't work because of blah" will be effective in garnering partial credit points.

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".

```
a := 0
b := n
s := a*a + b*b
while a <= b and s != n
  if s < n then
    a := a + 1
  else
    b := b - 1
  end if
  s := a*a + b*b
end while
if s = n then
  return (a, b)
else
  return "NO"
end if
```

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, go to this page and you can see the invariant I used.

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:

```
for i from 0 to (n-k) do
  sum := 0
  for j from i to (i+k-1) do
    sum := sum + A[j]
  end for
  print (sum / k)
end for
```

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 pseudo-code, 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.)

Present your algorithm for the coastal search problem, then analyze its worst case performance. The *difficulty measure* should be the distance n, in miles, from your starting point to the port. (Remember that this is unknown at the beginning of the algorithm!) The *cost function* will be the total distance sailed, both north and south, in miles, before the port at distance n is reached.

The best solutions achieve a worst-case cost of $O(n)$. But in this case the "hidden constants" really matter! For the most possible credit, give an *exact* analysis of the worst-case cost of your algorithm. There will be a class-wide competition for the algorithm with the lowest "hidden constant" in front of the n term.

4 Dominance

A *dominant* element in an array is one that occurs more than half of the time. Given an array A of size n, this algorithm determines what the dominant element is, or if none exists.

```
if n = 1 then
  return A[0]
else
  middle = floor(n/2)
  x := dominantElement(A[0..middle-1])
  y := dominantElement(A[middle..n-1])
  if x != "NONE" and count(A, x) > n/2 then return x
  else if y != "NONE" and count(A, y) > n/2 then return y
  else return "NONE"
  end if
end if
```

a) A crucial subroutine to the dominantElement algorithm above is the count algorithm, which counts how many times a given element occurs in the array.

Write a pseudocode description for the count algorithm that uses a loop. Then show that your algorithm is correct by using a loop invariant. Finally, show that your algorithm has worst-case cost $\Theta(n)$.

(Note: this seems like a lot to do but it's a really simple algorithm so everything should be very short.)

b) Since you know the count algorithm is $\Theta(n)$ time, determine the total worst-case cost of the dominantElement algorithm above. Express your answer as a big-O bound on the worst-case cost, simplified as much as possible.

c) What programming design paradigm does the dominantElement algorithm utilize? Explain your answer briefly.

d) Describe a different algorithm that solves this problem in worst-case $\Theta(n \log n)$ time by first sorting the array using MergeSort.

e) (**Warning**: this is a tough one. Don't spend all your time working on just this part. And be sure to acknowledge any sources.) Present an algorithm for this problem that has worst-case running time $\Theta(n)$.