Homework 2: CSCI 6212 Algorithms, due: Beginning of class, October 4, 2019

Collaboration Policies: This is not a group homework, you are required to do this homework yourself. However, you are *allowed* to use or search any passive online resource for information about how to solve the problem, and discuss the problems with classmates. You are *not allowed* to use active web resources (like Stack Overflow) where you post questions and ask for responses, post these questions to a "work for hire" site where someone else does them for you, or take *any* written notes from discussions with classmates or others. Problems:

1. Some convex hull algorithms have the property that their running time depends on the number of points on the convex hull as well as the total number of points, so that they run faster on inputs with a small convex hull than they do on inputs with a large convex hull. Explain why Graham's scan does not have this property.

   [5 points] Answer: The first step of Graham's scan is that it sorts all the points. This step does not depend on the number of points that are on the convex hull, so the runtime always is at least O(n log n).

2. In class, we discussed:

   (a) that the lower bound on the run time for sorting is $O(nlogn)$,

   (b) that you can reduce the sorting problem to the convex hull problem (so that you can use convex hull to sort a collection of numnbers), and

   (c) Chan's algorithm has a runtime that is O(n log h), which is better than O(n log n) when there are not many points on the convex hull.

   In a few sentences, explain why this isn't a contradiction.

   *[5 points] The reduction we talked about in class, takes a sorting problem and constructs a set of points. The way the construction worked, *all* the points end up on the convex hull. So, for the points we created, n = h, so O(n log h) = O(n log n), and there is no contradiction.*

3. We talked about several Convex Hull algorithms in class, but two of the simplest were Grahams Scan, which had a run time of $O(nlogn)$, and Gift-Wrapping, which has a run time of $O(nh)$. Suppose you know that you will be running your convex hull algorithms on inputs that have about 10,000 points in them and where about 200 will be on the convex hull. In three or four sentences, Which algorithm would you choose, and why? *[5 points] In general, O(nh) might be larger than O(n log n). In this case we are told* $h = 200$. *However,* $log_2(10000)$ *is about* 13*, so O(nh) is indeed larger than O(n log n) in this case. This suggests that the Graham's Scan algorithm would be best for this case.*

4. We are going to try a variant of the "which building blocks your view" problem given in class. In this case we are worried about the effects of massiving flooding on a 1-D city. If water comes into the city, how much gets trapped? specifically, you are given n non-negative integers that show the local elevation map. Compute how much water this terrain will trap if there is first a flood covering everything and then all the water than can flows away. Example picture:

   Please give an algorithm to compute the amount of water that is being trapped, a run-time analysis of that algorithm, and an argument that your algorithm is correct. For full credit,

Figure 1: The picture shows an elevation map that is represented by the array [0,1,0,2,1,0,1,3,2,1,2,1]. The correct final answer in this case is 6, because 6 units of water, shown in blue, are being trapped

the algorithm must run in O(n) time. Partial credit will be given for correct algorithms and proofs with a slower run-time.

[5 points — 2 points for argument, 2 points for pseudocode, 1 point for time complexity]
Idea: We will write an algorithm that computes, for each location the heighest building to its left and right. The min of these is the final water level at this location, this location will trap water starting at it's own level up to that "final water level"

```
Input: City [1..n] of heights at each location

 % initialize arrays to record max height on each side of each location

 TotalFlood(City)
    maxLeft[0...n]    = 0
    MaxRight[1...n+1] = 0

    For i = 1...n              % Loop 1, find highest location left
        maxLeft = max(City[i], maxLeft[i - 1]);

    For i = n...1              % Loop 2, find highest location right
        maxRight = max(City[i], maxRight[i + 1]);

    totalFlood = 0;
    For i = 1...n
        FloodLevel = min(maxLeft,maxRight)
        totalFlood = totalFlood + FloodLevel - City[i]

    return totalFlood
}
```

Argument for correctness: At each location, the water height, at the end, depends on what water is trapped at that location, and that water level is the min of the highest point on the left and right of each location. We explicitly compute the highest point to the left and right of each location, and then sum the amount of water flooding each location.