CS458 Assignment 2   
Tip: If you are using Word, symbols and equations are under the Insert tab.   
   
1. Let f(n) = aknk + ak-1nk-1 + ... + a1n + a0 be a polynomial where ak > 0.   
Prove that f(n) ε Θ(nk). (10 Points)

To prove this, first p(n) = O(nk) needs to be proven

Let B1 = ak + ak-1 + … + a1 + a0

Then for all n: p(n) = aknk + ak-1nk-1 + … + a1n + a0 <= aknk + ak-1nk + .. + a1nk + a0nk = (ak + a-1 + … + a1 + a0)nk = B1nk

So, p(n) = O(nk)

Now p(n) = Ω(nk) will be shown

P(n) = aknk + ak-1nk-1 + … a1n + a0 >= aknk = B2nk, where B2 = ak

So, p(n) = Ω(nk)

Therefore, p(n) = O(nk) and p(n) = Ω(nk), then p(n) = Θ(nk).  
   
2. Let G be a digraph. Let S be a vertex in G. Design an algorithm that computes the # shortest path from s to every other vertex of G. Show an example that # shortest path from S to some t ε G is exponential of |G|. (10 Points)

Create a dictionary, unvisited and put all the vertices in it. Create a dictionary, distance, and do the same, with the distance to s being 0 and all other nodes being infinity.

First, look at the neighbors of s and update their distances. Then, remove s from unvisited (or mark as visited).

Next, look at the closest vertex to s (min distance), t. Look at t’s neighbors and update their distances if you find a path closer to s ( min( d(u), d(t) + d(tu)). Mark t as visited.

Repeat the previous step until all vertices are visited.

The time complexity for this algorithm is O(V^2) where V = number of vertices.

Time to visit all vertices = O(V), time to process one vertex = O(V), so time required for visiting and processing all vertices = O(V) \* O(V) = O(V^2)

3. In each of the following compute T(n) or find an upper bound. T(1) is constant.   
(20 Points)   
   
a. T(n) = 4T(n/3) + log n!

T(n) = Θ(n logn)

b. T(n) = 9T(∛𝑛) + log2n

T(n) = Θ(n logn)

c. T(n) = T(n-2) + log n

T(n) = O(n^2)

d. T(n) = 2T(n-1) + 1

T(n) = O(4n)

4. We are given n points in a 2-D plane (x, y coordinates are given). Design an algorithm that finds a pair of points that have maximum distance from each other. Can it be done faster than O(n2)? (10 Points)

A simple way to find the maximum distance between a pair of points is to brute force it and check every possible distance between points. This has a time complexity of O(n^2) which isn’t the most efficient.

A more efficient way to find the maximum distance is a good deal more complex. This involves the use of graham scan’s convex hull, then further implementing an algorithm to find the maximum distance.

First, find the point with the smallest y-value. If multiple points share this, use the right-most one (the largest x value). Create a list of the output hull and put this point, P0, first.

Next, sort the remaining points by their polar angle in a counterclockwise fashion around P0. If two points have the same angle, put the nearest one first.

After the points have been sorted, check if two or more points have the same angle, if so, remove all points but the furthest from P0. Let m be the size of the new array.

Create an empty stack, S, and push P0, P1, and P2 onto it. Process the remaining m-3 points one at a time. For every point:

While the orientation of the: point at the top of the stack, point next to the top of the stack, and points[i] is not counterclockwise, keep removing points from the stack.

Push the next point onto S. When this is over, we have completed our convex hull.

Now, we can use our convex hull to implement a distance algorithm.

We have N points. Start from P1 and include points from a set of given points, such that the area of the region increases as more points are included.

Starting from P1, let K = 2. Increment K while area(PN, P1, PK) is increasing. Stop before it starts to decrease. The current point, PK may be the antipodal for P1.

Find the antipodal for P2 by area(P1, P2, PK) and incrementing K from where we previously stopped. We can do this continuously and keep track of the largest distance between any point and its antipodal. Once we are through the points, we will have our maximum distance, using an algorithm with a complexity of O(n\*log(n)).