CSCI 3104: Algorithms **Instructor Buxton**
Problem Set 6 (70 points) **Summer 2019, CU-Boulder**

1. (15 points) Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$. Compute its asymptotic running time.

(a):

To compute the running time for the transitive closure as an adjacency matrix. We know our worst case is that everything is connected to every other node, hence search traverse every edge and it must be V and E so the complexity is: O(VE)

2. (15 points) Grog –master of pictures– needs your help to compute the in- and out-degrees of all vertices in a directed multigraph G. However, he is not sure how to represent the graph so that the calculation is most efficient. For each of the three possible representations, express your answers in asymptotic notation (the only notation Grog understands), in terms of V and E, and justify your claim.

   (a) An *adjacency matrix* representation. Assume the size of the matrix is known.

   (b) An *edge list* representation. Assume vertices have arbitrary labels.

   (c) An *adjacency list* representation. Assume the vector's length is known.

(a):

This will take $O(v^2)$ because in an adjacency matrix you have a row of 1 vertices that correspond to a column of j vertices and to scan through each i and j of the matrix and that corresponds to scanning through V twice thus giving an asymptotic running time of $O(v^2)$.

(b):

This will take O(E) because with an edge list you scan through the edges between the vertice labels with the list to calculate the degree and so your calculation just depends on E. If this costs "E" find or insertion operations which is O(lg V) (lookup cost). Each lookup is calculation just depends on E. If this cost O(E) of them, two for each edge. Thus the total running time will take O(E lg V).

(c):

This will take O(V+E) because in an adjacency list, you have to initialize an array of V, where each index of the array can be called i which denotes a vertex from set V. Then for each i with the array, it points to a linked list containing all the vertices j where an edge exists between i an j. So for an adjacency list, it scans through the array of i vertice (v) then scans through the linked list corresponding to that index i until it finds and edge connecting i to j (from set E), giving a time of O(V+E).

3. (40 points) Consider a valleyed array $A[1, 2, \ldots, n]$ with the property that the subarray $A[1 \ldots i]$ has the property that $A[j] > A[j+1]$ for $1 \leq j < i$, and the subarray $A[i \ldots n]$ has the property that $A[j] < A[j + 1]$ for $i \leq j < n$. For example,
$A = [16, 15, 10, 9, 7, 3, 6, 8, 17, 23]$ is a valleyed array.

   (a) Write a recursive algorithm that takes asymptotically sub-linear time to find the minimum element of A.

   (b) Prove that your algorithm is correct. (Hint: prove that your algorithm's correctness follows from the correctness of another correct algorithm we already know.)

   (c) Now consider the multi-valleyed generalization, in which the array contains k valleys, i.e., it contains k subarrays, each of which is itself a valleyed array. Let $k = 2$ and prove that your algorithm can fail on such an input.

   (d) Suppose that $k = 2$ and we can guarantee that neither valley is closer than $n = 4$ positions to the middle of the array, and that the "joining point" of the two singly valleyed subarrays lays in the middle half of the array. Now write an algorithm that returns the minimum element of A in sublinear time. Prove that your algorithm is correct, give a recurrence relation for its running time, and solve for its asymptotic behavior.

(a):

Algorithm:

```
Given: Valley array
A[1,...,n]
Assumed: Count = 0
minArr(A[], count):
    if := A[count] < A[count+1] do
        print A[count]
            return
    else
    minArr(A[], count+1)
            end
```

Time Complexity:

If there are no identical elements in the array, it will take n time, so it takes asymptotically sub linear time T(n)=O(n)

(b):

A valleyed array has the property that its values are first decreasing and increasing. We can see we are finding the first element of the array. In the algorithm, the minArr() function takes the array and the count variable, otherwise it will move to next element by increasing the count. This algorithm will search for the minimum element until the required element is fount, which can take n-count steps. Therefore, the time complexity of the algorithm is O(n), or asymptotically sub-linear.

(c):

S'pose A= [100,50,10,25,30,60,40,20,80,120] We will have a return of 25 instead of 20. The algorithm output will fail under the two "waves" of A (d):

```
minElement(A, 1, n):
    Mem=1+n/2
    x=minElement(A, 1, mem)
    y=minElement(A, mem, n)
    if x < y
        return x;
    else
        return y;
        end
```