CS 3510: Design and Analysis of Algorithms
Section A, Spring 2017

Homework 1
Due: Monday 11:55PM, Jan 30th, 2017

Name: Alan Chiang

Notes:

1. Please don't write a program when you describe an algorithm. You should give an idea of what the algorithm does and then discuss the details of how your algorithm does specific tasks. If you give pseudo-code, you should explain your code.

2. For every algorithm that you describe, please explain its correctness and its runtime behavior.

3. You can assume that basic operations on integers which are not described to have a variable length (addition, multiplatication, subtraction, floor, ceiling, absolute value, comparison of less than, greater than, or equals, and anything you can form out of a constant number of these) can be done in constant time. You can also assume that, given i, you can read a_i from a list $a_1, ...a_n$ in constant time.

1. (10 points) Suppose you are given the following recursive program:

```
NoIdea(n) = if n > 1 -> print 'A';
                        NoIdea(n/2);
                        NoIdea(n/2)
            fi
```

Describe the recurrence relation in terms of $T(n)$, and express the number of times that this algorithm prints 'A', in terms of n using the big-O notation. You may assume that n is a power of 2.

ANSWER:

The recurrence relation is $T(n) = 1+T(n/2)+T(n/2)$, with 1 for the constant-time operation of printing A, and $T(n/2)$ for each of the two recursive calls on $n/2$.

The algorithm prints 'A' $O(n)$ times. $O(n)$ is reached by simplifying $O(2^{\log n})$, because each NoIdea(n) call prints A once, and NoIdea is called once on n, then twice on $n/2$, then four times on $n/4$.....and so on, doubling each time, until the base case is reached at $n = 1$.

This forms a geometric series $\sum_{i=n/2}^{\log n} 2^i T(n/2)$, which sums to $2^{\log n}$ because for geometric series with $r > 1$, the sum of a series is its last element. The exponent and logarithm cancel out. Thus $O(n)$.

2. (10 points) What does the algorithm *astatistic* below compute from the input list A of n integers? What is the running time of the algorithm *astatistic*? Why?

- Step 1: FOR $1 \leq i \leq n$ DO:
  - $Noccurences[i] = 0$.
  - FOR $1 \leq j \leq n$ DO:
    - IF $A[i] = A[j]$ THEN increment $Noccurences[i]$ by 1.
- Step 2: Find i such that $Noccurences[i]$ is a maximum and output $A[i]$.

ANSWER:

1. It finds the mode, the most frequently-occuring integer in input list A. It does this by selecting a number at index i, $A[i]$, and iterating through the elements of the input list. Every time $A[i]$ = the selected element, denoted $A[j]$, a counter for that integer is incremented. Then the maximum counter is found, and the integer which occurs most is returned.

2. Runtime is $O(n^2)$, because the algorithm traverses the entire list, size n, once for each of the n integers within the list. $n * n = n^2$.

3. (10 points) Describe a linear-time $O(|V| + |E|)$ algorithm that, given a graph $G = (V, E)$ and an edge $e \in E$, decides if G has a cycle containing e.

   ANSWER:

   We begin with the edge e, connecting vertices u and v. If the input is an edgelist, we convert the edgelist input into an adjacency list, but omitting u from the adjacency list of v and v from the neighbor list of u. If the input is an adjacency list, we simply remove u from the neighbor list of v and vice versa.

   We then perform a standard DFS on our graph (with edge e effectively omitted) using vertex u as the root. Each time a node is visited, we add it to an array (as well as marking it visited). When DFS is completed, we check the the array index corresponding to v to see if the array contains v. If so, there original graph has a cycle containing e and we return true. If the array does not contain v, then the graph does not have a cycle containing e and we return false..

   This solution runs in time $O(|V| + |E|)$, because running DFS takes linear time as described in class. If necessary, converting an edgelist to an adjacency list is also linear time. Array lookup is is $O(1)$.

4. (10 points) Describe a linear-time $O(|V|+|E|)$ algorithm that, given a directed acyclic graph $G = (V, E)$, decides if G contains a path which includes every vertex.

ANSWER:

We begin by identifying the source and sink nodes, of which there will always be at least one. The source node(s) will not be a neighbor to any other node, while sink node(s) will have no neighbors of their own. This will require iterating over all the nodes in our graph, which will also allow us to convert our edgelist input into an adjacency list, if necessary.

If the graph contains more than one source or more than one sink, we return false because a path can only originate from one source and end at one sink. If there are multiple of either, at least one vertex (the extra source or sink) will be excluded from the path, and thus the path will not include every vertex.

We will then perform depth-first search using the source node as our root. Every time a node is visited for the first time, we will increment a counter variable (and mark it as visited). The algorithm can then be broken into several cases:

1. If this node is not a sink node, we will continue searching as we would in ordinary DFS.

2. If it is a sink node, we will check if we have visited every node in the graph by comparing our counter variable == graph size.

2.a. If it does, we are finished and return true.

2.b. If it does not, we decrement the counter and continue our DFS until we hit step 2.a. or DFS terminates, at which point we return false.

This solution runs in time $O(|V|+|E|)$, because running DFS takes linear time as described in class. Our initial traversal to identify sources and sinks will also take linear time. Marking nodes as source/sink is an $O(1)$ operation, as is accessing an integer instance variable.