HOMEWORK 4
DATHRIKA SAICHARITHA
02010404.

1. 17.2.3. (Hint: Keep a pointer to the high-order 1.)
    - Let there be an array ‘a’
    - Create a variable A(max) to hold the higher order (1) index.
    - The lower order bit is at index 0 and A(max) is set to 1 because there aren’t any 1’s in A at first.
    - Depending on the value of the counter, whether it is increased or decreased, the value of A(max) is changed respectively.
    - To reset the counter, we can limit the value of A and the cost of reset is kept under control by ensuring that only a certain portion of credit from prior increments is used. So, reset only changes bits up to A(max).
    - To slightly turn the counter in the book costs 1’ and the cost of updating A(max) is expected to be ‘1’ too.
    - At first there aren’t any 1’s in A so one bit will cost ‘1’ to SET index to 1.
    - Same as this ‘1’ will be credited on every bit that has been SET to 1 and this credit on each bit will pay to RESET the bit during incrementing
    - Also, cost ‘1’ will update A(max)
    - If A(max) increases another cost’1’ will be added to the new higher order (1).
    - If A(max) does not increase cost’1’ will no longer be required.
    - Reset only changes bits at locations up to A(max) so every bit viewed by RESET has a credit of cost ‘1’
    - So, the credit saved on the bits can offset the bits of A becoming zeroes.
    - Cost of resetting the maximum value is only ‘1’ cost.
    - For the series of n INCREMENT and RESET operations it takes O(n) time because the cost of incrementing is “4’ and RESETTING is ‘1’.

2. OUT-DEGREE:
   Let the adjacency list of a directed graph be ‘adj’ and one vertex be x.
   Then the outdegree of this one vertex x is the length of adj[x].
   So, the time taken to compute the out-degree of one vertex is $\theta(|adj(v)|)$
   For all vertices, we need the sum of the lengths of all the adjacency lists in adj that is $|E|$.
   So, the time taken to compute the out-degree of every vertex is $\theta(V + E)$ where E is no. of edges and V is no. of vertices.

   IN-DEGREE:
   The in-degree of vertex v is the total number of times the vertex appears in all the lists in ‘adj’.
   The time to compute the in-degree of all vertices, if we search all the lists for each vertex is $\theta(VE)$.
   The other way of doing this is to scan through adj and increment a counter when x appears in the lists. The value in T will be in-degree of all the vertices.
   So, the time taken to compute the in-degree of every vertex be $\theta(V + E)$.

3. White vertices cannot convert directly into black. This is because for a vertex to convert into black, it must be allocated to $\boldsymbol{u}$ in the BFS procedure. This implies that each vertex must be enqueued in the queue at a certain point. This can happen only in line 17. If line 17 is executing on a vertex, then line 14 must also be executing on that vertex before executing line 17 to give it the GRAY color. By eliminating line 18, the algorithm does not reflect any change in the solution because line 13 doesn't concern if a vertex is BLACK or GRAY. So, since we never assign BLACK, we can represent the vertex color with a single bit indicating whether it is WHITE or GRAY. Overall, the result will be the same if line 18 is removed and we can use a single bit to store vertex color.

4. If we represent the input graph of BFS by an adjacency matrix the run time would be as follows:
   For the initializations from lines 1 to 9 the time complexity will be $\theta(V)$. where V=number of vertices.
   Time complexity for queue operation is $\theta(V)$.
   There is for loop inside the while loop,
   We need to modify the below for loop to handle the given form of input.
   12 $for\ each\ v \in G.adj(u)$
   13. $if\ v.color == WHITE$

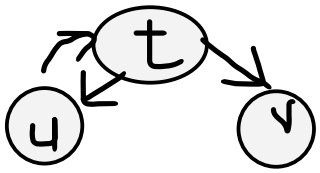
   Change the for loop to:

   12 $for\ each\ v = 1\ to\ |V|$
   13. $if\ v.color == WHITE\ \&\&\ adj(u)(v) == 1$
   This for loop executes two times so the run time is $\theta(V^2)$.

   Total run time for the modified BFS will be $\theta(V^2)$.

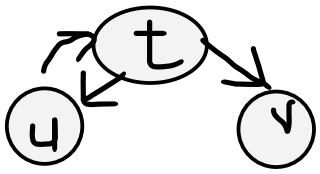
5. We can use depth-first search algorithm.
   Each edge in DFS is marked differently the first and second time it is traversed. The edge that is marked when second-time traversed should not be considered again.
   every time you see such an edge, just traverse it in both directions.
   To ensure that edges are taken in both directions, we simply need to backtrack until we find an unexplored edge.
   So, during the backtrack phase, the edges are explored in the opposite direction. But we need to be sure that we should not double-count edges at both ancestor and the descendent.
   The run time for this algorithm is O ($V + E$).
   Given a large supply of pennies, we place a penny on the edge that is traversed
   And when an edge has two pennies, we do not need to traverse it.

6. A DFS algorithm traverses a tree or graph in a single path from the parent vertex to its children's and grandchildren's vertices until it reaches a dead end.
   When a path has no more vertices to visit, the DFS algorithm will backtrack to a point where it can choose another path. The process will be repeated until all vertices have been visited.
   COUNTER EXAMPLE:



Consider 3 vertices u, v, and t of a graph and let there edges be (u,t), (t,u) (t,v).
When we first explore t and the adjacency list of t has u before v. so after exploring t we discover u. but u gets finished because the only vertex adjacent to u is w.
In this case, v can never be a descendant of u, because u is already finished, and v is not yet a descendant of u.

7. Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.\ d \leq u.\ f$

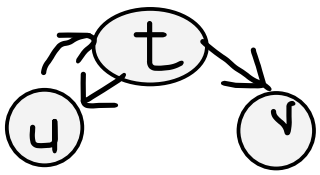


Let vertices be t, u, v and edges are (u, t), (t, u) (t, v).
Starting at vertex t (t. d = 1) and we discover v (v. d=3) and we discover no other vertex from v (v. f=4)
We now discover u (u.d=6) and there is no edge from u because t is already discovered so we finish exploring edge u (u. f=5) and t is finished so (t.f=8).
Edge list for given example:
(t. d =1) <= (v. f =4)
(t. d =1) <= (u. f =5)
(u. d =6) <= (t. f =8)
So (v. d =3) <= (u. f =5) is the counter-example.

8. Two things can happen if a new edge is added.
   1. If a new edge is added as a self-loop of a node or a new edge connects two vertices that are strongly connected, then the number of strongly connected components will not change.
   2. Instead, if the edge is in a reverse direction and this edge connects two strongly connected components then it may decrease the number of the strongly connected component from 1 to n., we can see that adding an edge cannot remove any path that existed before.

   We can see that it can either stay the same or decrease.

9. This simpler algorithm does not always produce correct results.
   Consider DFS will begin from vertex t, the finish time order will result as u, t, v.



If we use original given graph in the second DFS and scan the vertices in increasing finish time order, this will result in strongly connected component (t, u, v)
But there are two strongly connected component (t, u) and (v).
Clearly, we can see that the components will not be the same if we calculate with normal method but after using the simpler algorithm as stated by Professor Bacon, it gives us completely different set of strongly connected components as a result.