Name: ______________________________ **Student ID:** ______________________

Final Exam -- SOLUTIONS
Algorithms

Part I: Short Answer

1. [2 pts] Does there exist a simple graph having 10 vertices and 50 edges? Explain your answer.

   No. The maximum number of edges in a graph with 10 vertices is C(10,2) = (10)(10 – 1) / 2 = 45.

2. [2 pts] Does there exist a red-black tree having height 10 and exactly 30 (internal) nodes? Explain your answer.

   No. Recall the facts established in the proof of logarithmic height of red-black trees:

   $$n(T_r) \geq 2^{\mathrm{bh}(r)} - 1 \geq 2^{\frac{h}{2}} - 1$$

   This inequality fails to hold when n = 30 and h = 10.

   You can also arrive at this conclusion using the result that

   $$h \leq 2\log(n+1)$$

   since 10 is greater than 2log 31.

3. [2 pts] Is it possible for a binary search tree with 10 nodes to have height 10? Explain.

   No. Even a completely imbalanced tree having 10 nodes only has height 9. Moreover, every BST can be viewed as a tree in the sense of graph theory, and whenever |E| ≥ |V|, the graph must contain a cycle – so in this case, such a graph could not be a BST.

4. [2 pts] At a Las Vegas casino, there is a gambling machine called PRIMEBALL that gives information about prime numbers. You play PRIMEBALL by putting in some money, typing in a b-bit integer (where b must be quite big), and pulling a lever. The machine will, in O(b) time, output one of two values: "prime" or "not prime." If the machine displays "prime," there is a 25% chance that your b-bit number really is prime and a 75% chance that it is in fact composite. If the machine displays "not prime," then your b-bit number really is not prime.

   You decide to use this machine to create an algorithm for determining whether a certain b-bit number that you have in mind is prime. Your algorithm is to play PRIMEBALL k times (k is a positive integer), using the same b-bit integer as input every time. If the output is ever "not prime" you conclude that your integer is composite. If, in each of the k tries, you always see "prime" displayed, you conclude your integer is prime.

   Is this a good algorithm for determining if a number is prime? How big does k have to be for you to have 99% certainty that your algorithm has given a correct result? Explain. (Hint #1: $0.01 > 2^{-7}$. Hint #2: $2 – \log 3 > 2/5$.)

   Yes. For $1 \leq i \leq k$, let E_i be the event: PRIMEBALL is incorrect. Then $\Pr(E_i) = ¾$. Using the hint, we seek k such that

$$\left(\frac{3}{4}\right)^k < \frac{1}{2^7}$$

   In other words,

$$3^k 2^7 < 4^k$$

   This holds if and only if each of the following holds:

$$\log(3^k 2^7) < \log(4^k)$$
$$\log(3^k) + \log(2^7) < 2k$$
$$2k - k\log(3) > 7$$
$$k(2 - \log(3)) > 7$$
$$k > \frac{7}{2 - \log(3)}$$

   Since $2 - \log(3) > \frac{2}{5}$,

$$\frac{1}{2 - \log(3)} < \frac{1}{\frac{2}{5}}$$

   We let $k = 18 > \frac{7}{\frac{2}{5}}$.

   Then

$$k > \frac{7}{2 - \log(3)}$$

   Therefore, at least (approx) 18 tries are needed to guarantee 99% certainty.

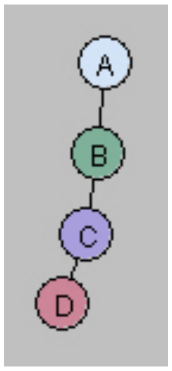
5. [2 pts] Explain what it means to say that a decision problem belongs to the class *NP*.

Good enough to say that a correct solution can be checked in polynomial time.

6. [2 pts] Suppose a graph G has m edges and n vertices. Suppose that G is connected. Is it possible that

$$m \leq (1/2)(n-1)(n-2)?$$

If so, give an example, with $n \leq 4$. If not, explain why not.



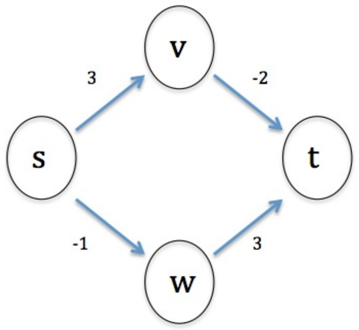
Yes, this is an example: graph is connected, $m = 3$, $n = 4$, and inequality holds.

7. [2 pts] Suppose you are given a graph G and a list $C_0, C_1, C_2, \ldots, C_k$ of its connected components (including the number of vertices and edges in each component). Explain how you can use this information to determine, by a single scan of the components (and performing O(1) steps on each component) whether G contains a cycle.

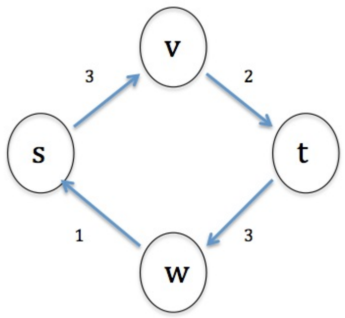
Let $n_i = n(C_i)$ and $m_i = m(C_i)$. Then G has a cycle if and only if, for some i, $m_i \neq n_i - 1$.

8. [2 points each] Four graphs are given below, each with a starting vertex S. Determine which shortest path algorithm (if any) can be used to determine the shortest path length from s to t. Select only one answer in each case. "SP" stands for "shortest path."

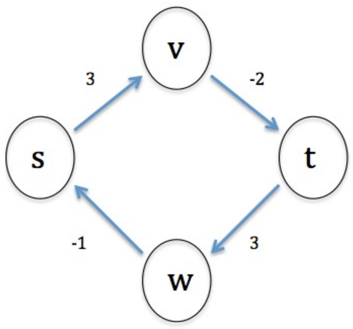
I. __B___ A. Dijkstra's algorithm (negative edge weight)
B. Dynamic programming SP algorithm
C. Neither A nor B, but another SP algorithm can be used
D. There is no known SP algorithm for this



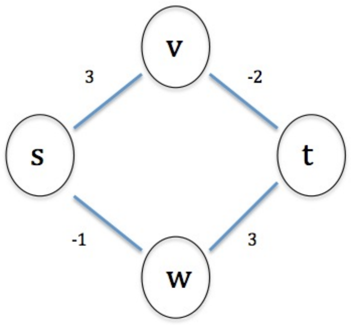
II. __A___ A. Dijkstra's algorithm
B. Dynamic programming SP algorithm (not a DAG)
C. Neither A nor B, but another SP algorithm can be used
D. There is no known SP algorithm for this



III. __C__ A. Dijkstra's algorithm (neg edge weights)
B. Dynamic programming SP algorithm (not a DAG)
C. Neither A nor B, but another SP algorithm can be used
D. There is no known SP algorithm for this

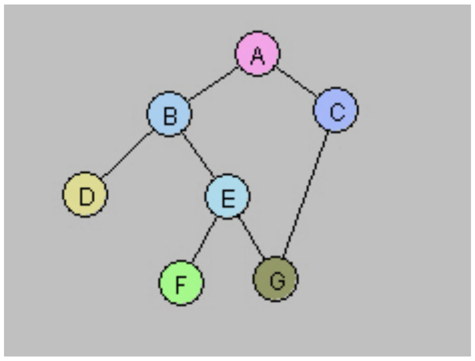


IV. __D___ A. Dijkstra's algorithm
B. Dynamic programming SP algorithm
C. Neither A nor B, but another SP algorithm can be used
D. There is no known SP algorithm for this (undirected, neg edge weight)



Part II. Skill Problems. There are six problems listed below. **Pick four problems to answer.** You must indicate which problems you are doing my circling the problem numbers. If you do more than four problems, I will randomly pick four to grade.

1. [20 points] The graph displayed has an odd cycle which we call K.



a. Describe informally the variant of the BFS algorithm used in class to detect whether a graph contains an odd cycle.

Pick a starting vertex and build the BFS rooted tree, keeping track of levels. As the tree is being built, each time a new vertex is considered, check if there is another vertex at the same level to which it is adjacent. If this happens, return "has an odd cycle"; if not, return "no odd cycle."

b. What are the vertices of K? A, B, E, G, C

c. Show the steps of the BFS algorithm with starting vertex A (if there are multiple vertices adjacent to a current vertex, traverse them in alphabetical order). Be sure to indicate the extra steps needed to detect an odd cycle.

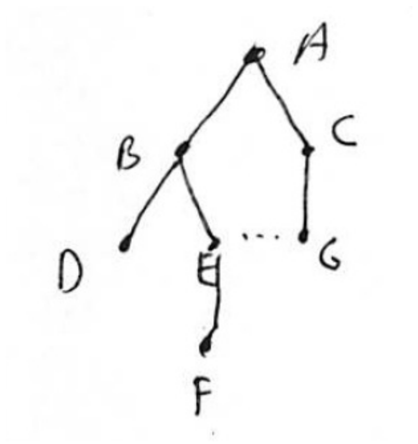
A is starting vertex, marked, put in Q
Set level(A) = 0
Remove A from Q
Insert verts adjacent to A into Q, mark, set level 1
  Insert B, C into Q, mark
  level(B) = 1, level(C) = 1
Remove B from Q
Insert verts adjacent to B into Q, mark, set lev 2
  Insert D, E into Q, mark
  level(D) = 2 = level(E)
Remove C from Q
Insert verts adjacent to C into Q, mark, set lev 2
  Insert G into Q
  level(G) = 2
  E and G are adjacent at same level – return "odd cycle"

Algorithm: Breadth First Search (BFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.

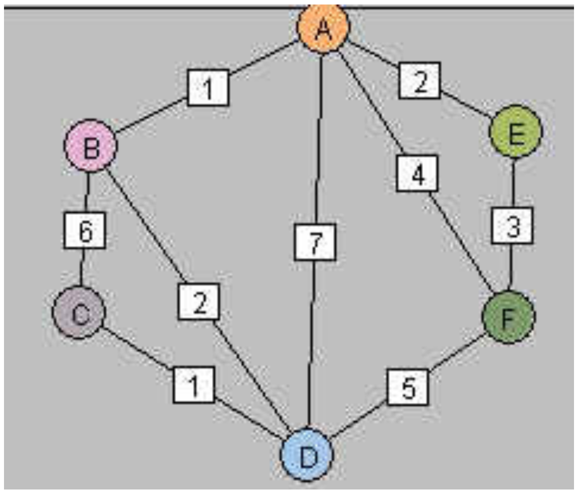
```
Initialize a queue Q
Pick a starting vertex s and mark s as visited
Q.add(s)
while Q $\neq \varnothing$ do
    v $\leftarrow$ Q.dequeue()   //adds v to X, the "pool" of explored vertices
    for each unvisited w adjacent to v do
        mark w
        Q.add(w)
```

d. At what level in the BFS rooted tree are two adjacent vertices discovered in the algorithm? (Remember that the root occupies Level 0.)

Level 2



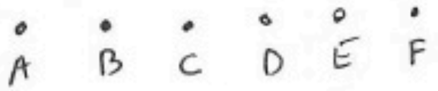
2. [20 points] Use Kruskal's algorithm to obtain a minimum spanning tree for the weighted graph given below. To manage clusters, use a tree-based implementation of the DisjointSets data structure. Show how trees evolve after each edge is explored. You must display your final set T of edges, representing a minimal spanning tree for the graph.



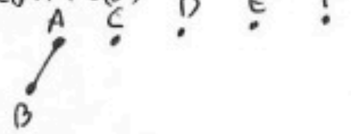
Sorted edges: AB, CD, BD, AE, EF, AF, DF, BC, AD

T = {AB, CD, BD, AE, EF }

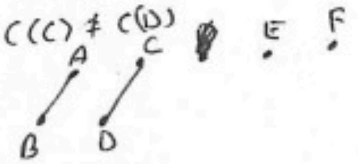
Starting clusters:



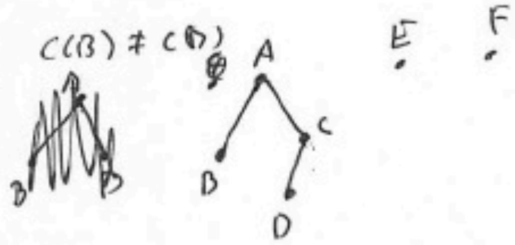
AB $C(A) \neq C(B)$



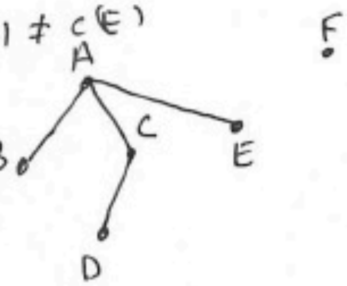
CD $C(C) \neq C(D)$



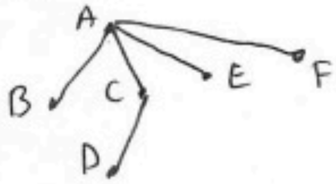
BD $C(B) \neq C(D)$



AE $C(A) \neq C(E)$



EF $C(E) \neq C(F)$



3. [20 points] Carry out the steps of HeapSort for input array [5, 8, 6, 3, 4]. For Phase I of the algorithm, use BottomUpHeap instead of the usual procedure for heapifying the input array. Then perform Phase II in the usual way. You must show all steps of BottomUpHeap and also of Phase II.

HeapSort ([5, 8, 6, 3, 4])

Phase I using BUH

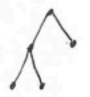
BUH (5, 8, 6, 3, 4)

k = 5

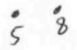
Not of the form $2^h - 1$.

We use iterative version

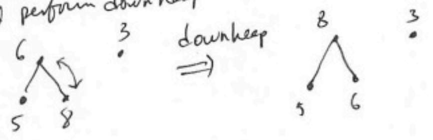
Heap must be like this



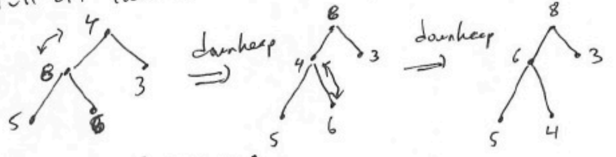
Begin with 5, 8 as separate heaps.



Pull off two more for next level and perform downheap



Pull off last and do downheap



As array: [8, 6, 3, 5, 4]

Phase II

$\swarrow$ $\textcircled{8}$ 6 3 5 4

8   4 6 3 5 _

$\overset{\text{downheap}}{\Longrightarrow}$

6 4 3 5 _

6 4 3 5 | 8

$\textcircled{6}$ $\swarrow$ $\textcircled{6}$ 4 3 5 | 8

5 4 3 _ | 8

$\overset{\text{downheap not}}{\Longrightarrow}$ needed

5 4 3 | 6 8

$\textcircled{5}$ $\swarrow$ $\textcircled{5}$ 4 3 | 6 8

3 4 _ | 6 8

$\overset{\text{downheap}}{\Longrightarrow}$

4 3 _ | 6 8

4 3 | 5 6 8

---

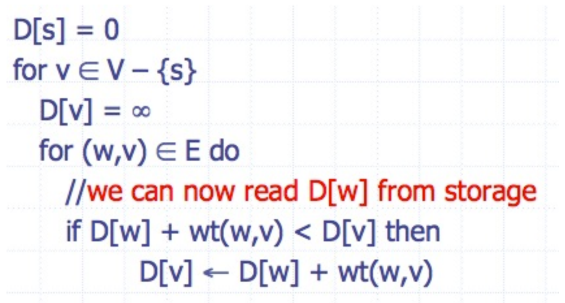
4   3 _ | 5 6 8

3 | 4 5 6 8

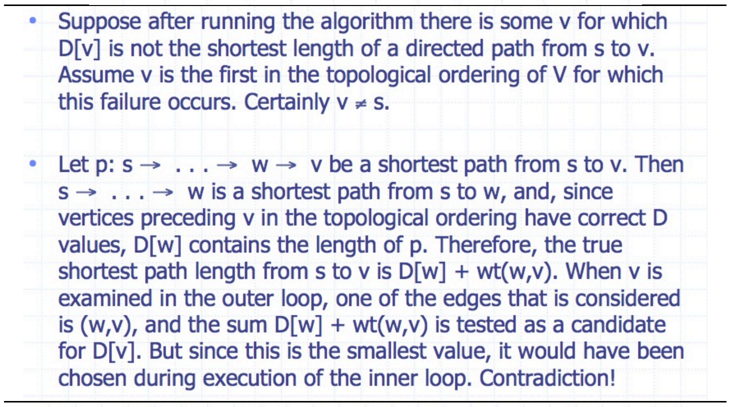
Output: [3 4 5 6 8]

4. [20 points] Provide pseudocode for the dynamic programming solution to the shortest path problem for directed acyclic graphs. Then do the following:

   a. Provide an analysis of the correctness of this algorithm.
   b. Derive the asymptotic running time for the algorithm.

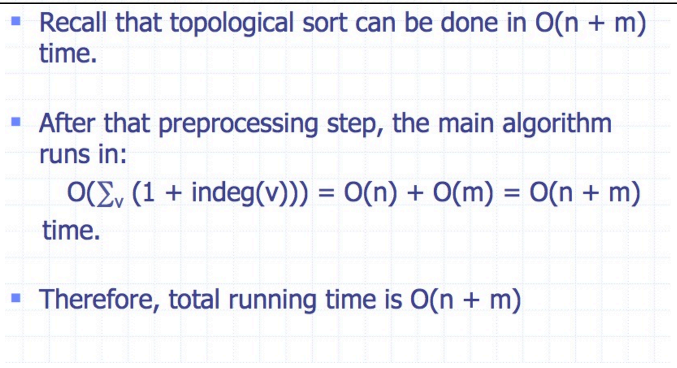


```
D[s] = 0
for $v \in V - \{s\}$
    D[v] = $\infty$
    for $(w,v) \in E$ do
        //we can now read D[w] from storage
        if D[w] + wt(w,v) < D[v] then
                D[v] $\leftarrow$ D[w] + wt(w,v)
```



- Suppose after running the algorithm there is some v for which D[v] is not the shortest length of a directed path from s to v. Assume v is the first in the topological ordering of V for which this failure occurs. Certainly $v \neq s$.

- Let p: $s \to \ldots \to w \to v$ be a shortest path from s to v. Then $s \to \ldots \to w$ is a shortest path from s to w, and, since vertices preceding v in the topological ordering have correct D values, D[w] contains the length of p. Therefore, the true shortest path length from s to v is D[w] + wt(w,v). When v is examined in the outer loop, one of the edges that is considered is (w,v), and the sum D[w] + wt(w,v) is tested as a candidate for D[v]. But since this is the smallest value, it would have been chosen during execution of the inner loop. Contradiction!



- Recall that topological sort can be done in O(n + m) time.

- After that preprocessing step, the main algorithm runs in:

  $O(\sum_v (1 + \mathrm{indeg}(v))) = O(n) + O(m) = O(n + m)$

  time.

- Therefore, total running time is O(n + m)

5. [20 points] The SetCover decision problem is as follows. You are given a positive integer *k,* a set *S*, and a collection of *n* sets $\{S_1, S_2, \dots, S_n\}$, where

$$S = \bigcup_{i=1}^{n} S_i$$

The decision problem asks whether there is a subcollection of $\{S_1, S_2, \dots, S_n\}$, consisting of at most *k* sets, whose union is also equal to *S*. In symbols, does there exist a subcollection $\{S_{i_1}, S_{i_2}, \dots, S_{i_k}\}$ of the original collection so that

$$S = \bigcup_{j=1}^{k} S_{i_j}$$

(Such a subcollection is called a *cover of size k.*)

Example: Let S = {1,2,3,4}, k = 2, and suppose the given collection of sets is {{1,2,3}, {2,3,4}, {1,2,4}}.
Notice that the subcollection {{1,2,3}, {1,2,4}} is a cover of size 2, so the return value for this instance of the problem is "Yes".

For this problem, prove that SetCover belongs to NP.

Suppose we are given a positive integer *k,* a set *S*, and a collection of *n* sets $\{S_1, S_2, \dots, S_n\}$, where

$$S = \bigcup_{i=1}^{n} S_i$$

and $|S| = m$. If we are given a collection of sets $\{T_1, T_2, \dots, T_1\}$, we take these steps to verify it is a solution: We will show that verification can be done in time bounded by a polynomial in $n + m$ (notice input size is $O(n+m)$ – in particular, k is $O(n)$).

A. We verify that $\ell \le k$ – i.e. that the subcollection has size at most *k* (one loop of length at most *k*, or less, depending on the implementation)

B. Verify each T_i belongs to the original collection: For $1 \le i \le k$, for $1 \le j \le k$, check if $T_i = S_j$**.** This step is done, for each *i, j*, by comparing sizes of T_i and S_j; then storing elements of S_j in a hashtable, and then performing a single scan of elements of T_i to make sure they all also belong to S_j**.** This innermost loop requires O(m+m) (since these sets are potentially as large as *S*); in the context of the two outer loops, running time is $O(mk^2)$ which is $O(mn^2)$

C. Verify that

$$S = \bigcup_{i=1}^{k} T_i$$

This requires three nested loops, one of size *m*, another other of size *k*.and a third possibly as big as *m*: For each *x* in *S*, for each *i* with $1 \le i \le k$, we scan T_i in search of *x*. This takes $O(km^2)$, which is $O(n\, m^2)$.

Total running time is therefore $O(n(n^2 + m^2))$. It follows that SetCover belongs to NP.

6. [20 points] Show that the decision problem SubsetSum is polynomial reducible to the Knapsack problem. (For this one, state the SubsetSum problem, state the Knapsack problem, explain how a SubsetSum instance can be converted into a Knapsack instance, explain how a solution to one automatically provides a solution to the other, and explain why the running time of the conversion algorithm is polynomial bounded.)

Given an instance I_{SS} of SubsetSum of size n (that is, a set $S = \{s_0, s_1, \dots, s_{n-1}\}$ together with some positive integer k), we show how to obtain an instance I_K of Knapsack also of size n so that I_{SS} has a solution if and only if I_K does. Moreover, we show that the general process of conversion runs in polynomial time.

I_K consists of the same set $S = \{s_0, s_1, \dots, s_{n-1}\}$, weights $w[] = \{w_0, w_1, \dots, w_{n-1}\}$ and values $v[] = \{v_0, v_1, \dots, v_{n-1}\}$, and max weight W and min value V, defined by

$$W = k,\ V = k, \text{ and for } 0 \le i \le n-1, w_i = s_i = v_i.$$

The effort to build I_K requires nothing more than assignment statements; we therefore take our polynomial witness to be cn for some big enough constant c.

We prove that a solution to I_K yields a solution to I_{SS} : A solution to I_K is a subset $T \subseteq \{0, 1, \dots, n-1\}$ satisfying $\sum_{i \in T} w_i \le W$ and $\sum_{i \in T} v_i \ge V$. Since for $0 \le i \le n-1, w_i = s_i = v_i$ and $W = k$, $V = k$, it follows that $\sum_{i \in T} s_i = k$.

We prove that a solution to I_{SS} yields a solution to I_K : A solution to I_{SS} is a subset $T \subseteq \{0, 1, \dots, n-1\}$ satisfying $\sum_{i \in T} s_i = k$. Since for $0 \le i \le n-1, w_i = s_i = v_i$ and $W = k$, $V = k$, it follows that $\sum_{i \in T} w_i = W$ and $\sum_{i \in T} v_i = V$.

Part III. SCI. Describe how *two* principles of SCI find expression in the principles underlying the study of algorithms

Part IV. (Extra Credit – 8 points). A directed graph is *strongly connected* if, for any two vertices v, w in the graph, there is a directed path from v to w in the graph. Give an O(m + n) algorithm that determines whether an input digraph is strongly connected. You must do the following:

1. Give an English description of how your algorithm works
2. Write up your algorithm in pseudo-code
3. Give a fairly rigorous explanation for why the algorithm has an O(m + n) running time.

Pick a starting vertex s. Perform DFS for digraphs and check that the number of marked vertices at the end is equal to number of vertices of the graph; if not, return "not strongly connected." Perform DFS for digraphs again, starting at s, but this time, reverse direction of all edges. This can be done by treating the in-list (the adjacency list that matches to each vertex *w* the in-vertices for *w*; namely, those vertices *v* for which (*v*,*w*) is an edge) as the out-list (recall that the outlist is what DFS uses as it traverses the graph). Check that number of marked vertices is same as size of V; if not, return "not strongly connected"; if so, return "strongly connected."

Correctness: Suppose v, w are vertices. There is a forward path from s to v:

$$s = v_0 \to v_1 \to \cdots \to v_m = v.$$

There is a backward path from s to w:

$$s = w_0 \leftarrow w_1 \leftarrow \cdots \leftarrow w_k = w.$$

Now consider the path

$$w = w_k \to \cdots \to w_1 \to w_0 = s = v_0 \to v_1 \to \cdots \to v_m = v.$$

Running Time: DFS is run twice, with O(1) extra work. Therefore, running time is O(m + n).