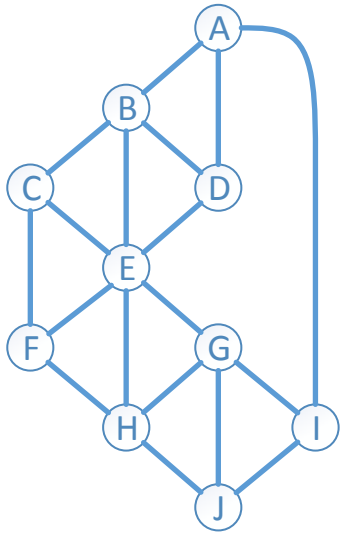
Problem #1

Perform Depth-First Search (DFS) on the graph below starting from vertex A. Trace the execution of DFS by writing the discovery and finish time for every node in the graph.

Homework #7 – Problem #1

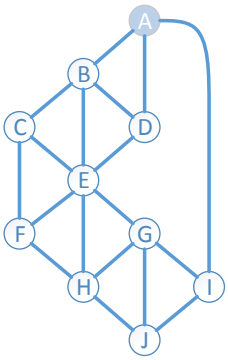


Below is a table of the discovery and finish times for the ten vertices.

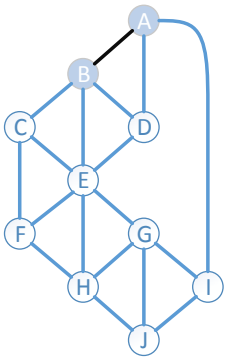
Vertex ID	A	B	C	D	E	F	G	H	I	J
Discovery Time	1	2	3	5	4	7	9	8	10	11
Finish Time	20	19	18	6	17	16	14	15	13	12

The following is a trace of the execution of DFS through the graph. Once an edge has been discovered, it is turned gray. When it is finished, it is turned black. Unclassified edges are light blue; tree edges are in black, and back edges are green. The graph has no forward or cross edges, since those types of edges only appear in directed graphs.

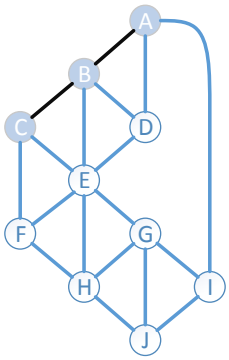
Homework #7 – Problem #1
Step #1



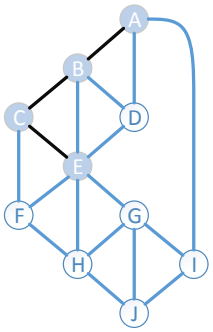
Homework #7 – Problem #1
Step #2



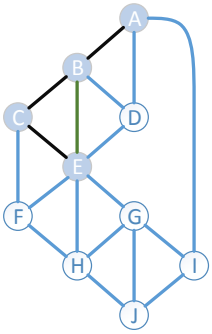
Homework #7 – Problem #1
Step #3



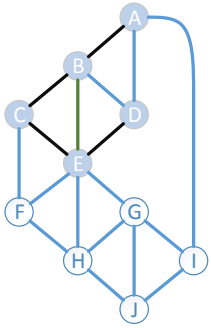
Homework #7 – Problem #1
Step #4



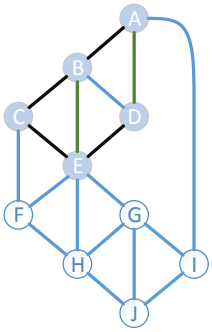
Homework #7 – Problem #1
Step #5.i



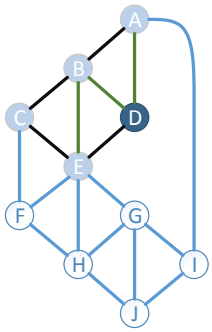
Homework #7 – Problem #1
Step #5.ii



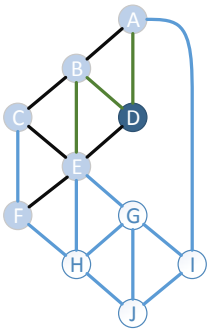
Homework #7 – Problem #1
Step #6.i



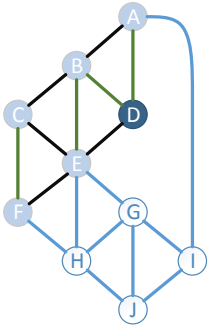
Homework #7 – Problem #1
Step #6.ii



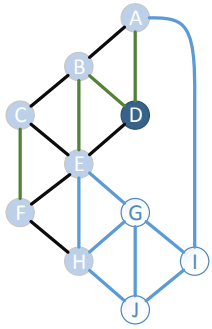
Homework #7 – Problem #1
Step #7



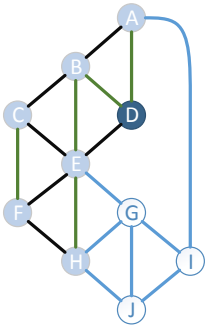
Homework #7 – Problem #1
Step #8.i



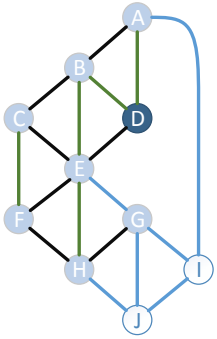
Homework #7 – Problem #1
Step #8.ii



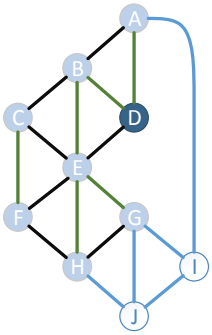
Homework #7 – Problem #1
Step #9.i



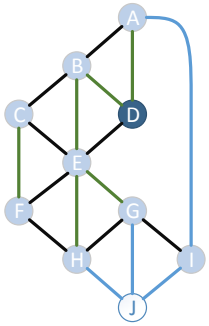
Homework #7 – Problem #1
Step #9.ii



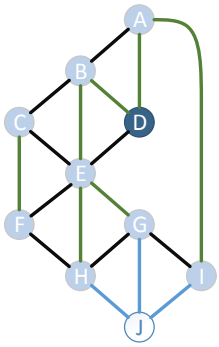
Homework #7 – Problem #1
Step #10.i



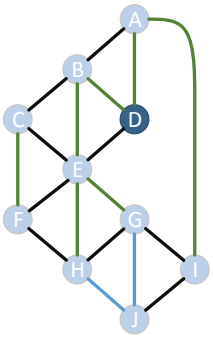
Homework #7 – Problem #1
Step #10.ii



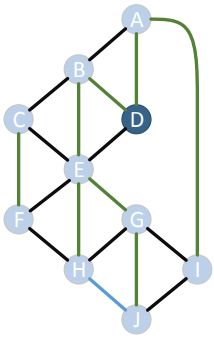
Homework #7 – Problem #1
Step #11.i



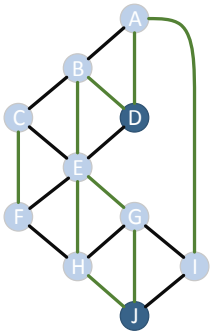
Homework #7 – Problem #1
Step #11.ii



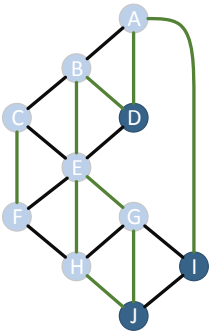
Homework #7 – Problem #1
Step #12.i



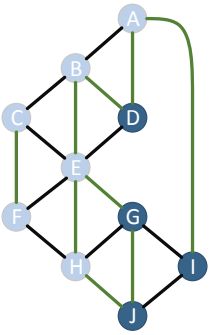
Homework #7 – Problem #1
Step #12.ii



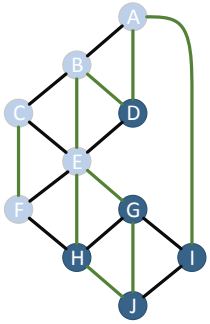
Homework #7 – Problem #1
Step #13



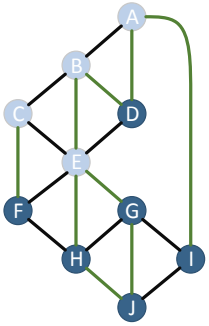
Homework #7 – Problem #1
Step #14



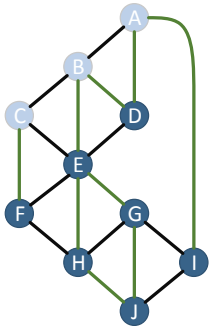
Homework #7 – Problem #1
Step #15



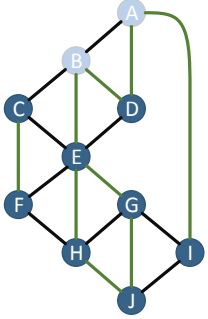
Homework #7 – Problem #1
Step #16



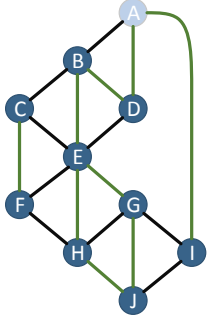
Homework #7 – Problem #1
Step #17



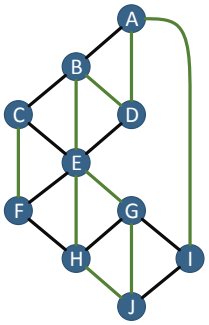
Homework #7 – Problem #1
Step #18



Homework #7 – Problem #1
Step #19



Homework #7 – Problem #1
Step #20



Problem #2

In the textbook, the running time of depth first search is specified as:

$$\theta(|V| + |E|)$$

This running time applies when the edge data is stored in the form of an adjacency list. For each vertex, v, the time required to check its edges is equal to $|Adj[v]|$ (i.e. the size of vertex *v*'s adjacency list). Using aggregate analysis, it can be shown that the total running time to check all edges is:

$$\sum_{v \in V} |Adj[v]| = \theta(E)$$

Since each vertex needed to be checked, the total running time was:

$$\theta(|V|) + \theta(|E|) = \theta(|V| + |E|)$$

However, in an adjacency matrix, this aggregate analysis is not possible. For all vertices, the equivalent size of a matching adjacency list (i.e. $|Adj[v]|$) would equal to the number of vertices in the graph (i.e. $|V|$). As such, the running time to check all vertices is:

$$\sum_{v \in V} |V| = |V| \cdot |V| = \theta(|V|^2)$$

That makes the total running time for depth-first search (DFS) with an adjacency matrix:

$$\boxed{\theta(|V|^2)}$$

Problem #3

Any directed acyclic graph (DAG) simplifies to a directed tree or a collection of directed trees (i.e. a directed forest). I will only discuss the case of the directed tree in detail. The case of the directed forest is an extension of the directed tree case; this proof can be extended to directed forests by treating the trees in the forest individually or by merging the individual directed trees into a single tree and using that single, merged tree as the basis for discussion.

Define a source vertex as any vertex in a directed tree that has no incoming edges. In a rooted, directed tree, there is a single source vertex. In a polytree, there may be multiple source vertices. I will discuss the case of the single source vertex because given any source vertex, a polytree can be simplified to a standard directed tree by excluding all unreachable vertices or by adding safe edges to the graph that preserve the tree structure while creating a single source (root) vertex.

In a rooted tree by definition, there are no cycles. Moreover, by definition, the source/root vertex is the only vertex in the tree that has no incoming edges. All other vertices are reachable from that source vertex. If an edge originating from any other vertex in the graph (including itself) going to the source/root vertex is added to the graph, then the tree will necessarily have a cycle. That is because all other vertices in the tree (including itself) are already reachable from the source/root vertex, and by adding this edge, a cycle is closed from the source vertex to the already reachable vertex where this new edge originates. Hence, by allowing all vertices to have at least one incoming edge, a cycle is necessarily formed making the graph no longer acyclic. The cases of the polytree and the directed forest are extensions of the individual tree case as explained above.

Note: The requirement that a DAG has at least one vertex with no incoming edges is only a necessary condition; it is not a sufficient condition. Figure 1 is an example of a graph with a vertex (D) that has no incoming edges, but the graph has a cycle (A → C → B → A).

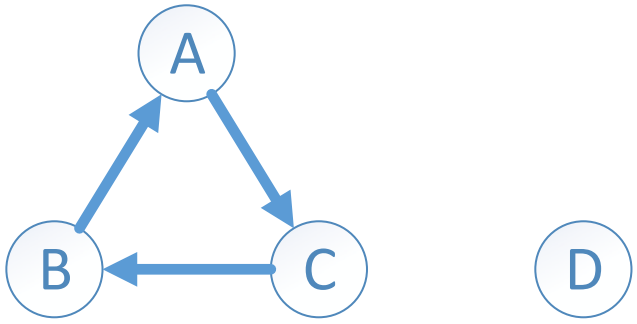


Figure 1 – Directed Graph with a Cycle and a Vertex (D) with No Incoming Edges

Problem #4

Given a graph, G, with a set of vertices, *V*, and a set of edges, *E*, below is an algorithm to perform topological sort in linear time. An assumption used below is that the edges are defined as adjacency lists and not as an adjacency matrix.

Step #1: Initialize an array of $|V|$ counters. Call this array of vertices `InDeg`. The running time of this step is:

$$\theta(1)$$

Step #2: Determine the in-degree of each vertex, $v \in V$; store the in-degree for each vertex, *v*, in the $|V|$ counters (i.e. array `InDeg`) created in step #1. The procedure to determine all in-degrees entails iterating over the set of edges, *E*, in two nested `for` loops. The outer `for` loop iterates over the set of vertices while the inner `for` loop iterates over each vertex's adjacency list. For each edge *e* (where $e \in E$), the algorithm determines the edge's head[1] and increments the associated counter for that head vertex in array `InDeg` (see step #1). The running time of this step is:

$$\sum_{v \in V} 1 + |Adj[v]| = \theta(|V| + |E|)$$

Step #3: Initialize an empty, standard, FIFO queue, *Q*. The running time of this step is:

$$\theta(1)$$

Step #4: Iterate through the $|V|$ counters (i.e. array `InDeg`) created in step #1 and loaded in step #2. If the counter for a vertex, *v*, is 0, then add it to the queue (*Q*) initialized in step #3. The running time of this step is:

$$\theta(|V|)$$

Step #5: Initialize a counter that represents the number of completed vertices. Call this counter "`Numb_Completed_Vertices`", and set it to 0. The running time of this step is:

$$\theta(1)$$

Step #6: Initialize an empty list. This list will store the result of the topological sort, and is in the form of a linked list of vertices. Call this list `Topoloigcal_Order`. This list will be loaded from the head to the tail as explained in step #7 below. The running time of this step is:

$$\theta(1)$$

Step #7: Create a while loop that runs until the queue, *Q*, (created in step #3) is empty (i.e. $|Q| == 0$).

```
while(|Q| > 0 ){          //---- Execute the loop until the queue is empty
    v = DEQUEUE(Q)        //---- Pop off the next vertex with no incoming edges
    Numb_Completed_Vertices = Numb_Completed_Vertices + 1 //--- Increment number of completed vertices

    Add vertex v to the end (tail) of the list Topological_Order

    //---- Iterate through each edge in the adjacency list of the dequeued vertex, v
    for each e in Adj[v]{
        InDeg[e.head] = InDeg[e.head] – 1 //-- Decrement numb InDeg for head vertex
        If (InDeg[e.head] == 0) Then
            ENQUEUE(Q, e.head)
```

[1] Note for a directed edge (i.e. arc) from vertex *u* to vertex *v* where $u, v \in V$, vertex *v* is the head of the edge, and *u* is the tail.

```
        }
    }
```

This algorithm iterates through the queue, *Q*, which contains all vertices with zero incoming edges. At each iteration of the loop, the head element of the queue (i.e. vertex *v*) is dequeued. Once a vertex is dequeued, that vertex is added to the list of the topologically sorted vertices. Unlike the textbook algorithm that does topological sorting using Depth-First Search and builds the topologically sorted list from least precedence to highest precedence, this algorithm builds the list from highest precedence to lowest precedence. As such, as each new vertex is added to the list of vertices, it is added at the end (tail) of the list and ***not*** the head. This is required because those vertices with lowest in-degree (i.e. *InDeg*[*v*] == 0) in this algorithm have the highest precedence. In addition to mark that vertex, *v*, has been processed, a counter, "`Numb_Completed_Vertices`", is incremented. This will be used to determine if the graph has a cycle. Both of these steps are require a total of $\theta(1)$ time.

For each outgoing edge from dequeued vertex *v*, the head of the edge (i.e. the destination vertex) has its associated counter of incoming edges decremented by 1. There are then two possible scenarios:

Case #1: The In-Degree of the Edge's Head is not Zero – The loop would skip the `if` statement and continue executing. This is done because the vertex at the edge's head is still dependent on one or more other vertices.

Case #2: The In-Degree of the Edge's Head is Zero – The `if` block would be executed and the head vertex of the edge is added to the queue, *Q*; this is done because for the reduced graph with vertex, *v*, and edge, *e*, removed, the head vertex no longer has any more parents. Otherwise,

The tight bound on the running time of this step requires aggregate analysis. First, each vertex can only be enqueued (and in turn dequeued) once since it is only enqueued when it no longer depends on any other edge; hence, the `while` loop can be run a maximum of $|V|$ times. **If the graph has a cycle, the while loop will exit before all vertices have been processed (i.e. enqueued). This possibility will be checked for in step #8 below.**

Second, for each vertex, *v*, the inner `for` loop is run $|Adj[v]|$ times. Hence, the combined running time of each iteration of the `while` loop is:

$$\theta(1 + |Adj[v]|)$$

Since the `while` loop is executed a maximum of $|V|$ times, its total running time is:

$$\sum_{v \in V} \theta(1 + |Adj[v]|) = O(|V| + |E|)$$

Step #8: Check if the counter, "`Numb_Completed_Vertices`", is equal to the number of vertices (i.e. $|V|$). **If not, then the graph has a cycle, and an error would be returned by the algorithm.** If they are equal, then the graph is acyclic, and the topologically sorted list of vertices, `Topological_Order`, is returned by the algorithm. The running time of this step is:

$$\theta(1)$$

The total running time of this algorithm is:

$$\theta(Total) = \sum_{i=1}^{8} \theta(Step_i)$$

$$\theta(Total) = \theta(1) + \theta(|V| + |E|) + \theta(1) + \theta(|V|) + \theta(1) + \theta(1) + O(|V| + |E|) + \theta(1)$$

$$\boxed{\theta(Total) = \theta(|V| + |E|)}$$

Problem #5

Given a graph, *G*, with weight function, *W*, the simplest way to find the maximum spanning tree is to transform the graph into a modified form and then to run one of the minimum spanning tree algorithms on it. There are two ways to transform the graph; the first way would work on the majority of graphs that are normally encountered in real world applications, while the second one is guaranteed to work on all graphs.

Option #1: For all edges, *e* (where $e \in E$), and associated edge weights w_e, (where $w_e \in W$), define a modified weight, w'_e, as:

$$w'_e = \frac{1}{w_e}$$

Note that option 1 only works for weight functions where are all edge weights are greater than 0.

Option #2: For all edges, *e* (where $e \in E$) , and associated edge weight, w_e (where $w_e \in W$), define a modified weight, w'_e, as:

$$w'_e = -1 \cdot w_e$$

Define W' as the set of all modified edge weights w'_e.

Depending on how the weights are stored, the running time of the weight function conversion may vary. If the weights are stored in essentially a long array with constant time look-up, then the running time would be:

$$\sum_{e \in E} \theta(1) = \theta(|E|)$$

If the weights are stored in a structure similar to the adjacency list, then the running time would be:

$$\sum_{v \in V} \theta(1 + |Adj[v]|) = \theta(|V| + |E|)$$

The running time to transform the weight function is independent of whether option #1 or option #2 are used.

By using either of the two transformation methods described above, the edges that previously had the maximum weight in *W* have the minimum weight in *W'*. Similarly, those edges that previously had the minimum weight in *W* have the maximum edge weight in *W'*.

Given this transformation, you can now use either Prim's or Kruskal's algorithms to find the minimum spanning tree on *G* and *W'*; this equates to the maximum spanning tree on *G* and *W*. Both Prim's and Kruskal's algorithms have a running time of:

$$O(|E| \cdot \lg(|V|))$$

Using the worst case of the two running times to convert the weight function, the total running time would be:

$$\theta(Step\#1) + \theta(Step\#2) = \theta(|V| + |E|) + O(|E| \cdot \lg(|V|))$$

Since in an undirected, connected graph, the number of edges $|E|$ is:

$$|E| \geq |V| - 1$$

Then the number total running time simplifies to:

$$\boxed{\theta(|V| + |E|) + O(|E| \cdot \lg(|V|)) = O(|E| \cdot \lg(|V|))}$$

Note this is the same asymptotic runtime as the standard minimum spanning tree algorithms.

Problem #6

Depending on the location of the maximum edge within the graph, it may or may not be included in the minimum spanning tree. If a graph is undirected and connected with the number of edges equal to the number of vertices, then the graph has a simple cycle[2]. If the maximum weighted edge is part of the simple cycle, then it would not be in the minimum spanning tree. However, if the edge is not part of a simple cycle, then it would be included in the minimum spanning tree. Below are two example graphs.

Figure 2 is a graph with three vertices and three edges. The minimum spanning tree for this graph contains two edges: (A,B) and (B,C). Since the maximum weight edge (A,C) is part of a simple cycle, it is not included in the minimum spanning tree.

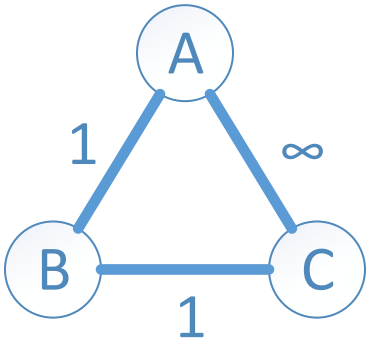


Figure 2 - Graph with Three Vertices where the Maximum Weighted Edge (A,C) is Not Part of the Minimum Spanning Tree

Figure 3 is a graph with four vertices and four edges. The minimum spanning tree for this graph is not unique. It can be three different trees. The first is { (A,B), (B,C), (C,D) }; the second is { (A,C), (B,C), (C,D) }, and the third is { (A,B), (A,C), (C,D) }. Note that all three minimum spanning trees contain the edge (C,D), which has an extremely expensive weight (e.g. ∞) that should never be included in a minimum spanning tree if otherwise avoidable. The reason this maximum weighted edge must be in this minimum spanning tree is because it is the only way to reach vertex D. In Figure 2, there was an alternate route to connect A and C since they were part of a simple cycle.

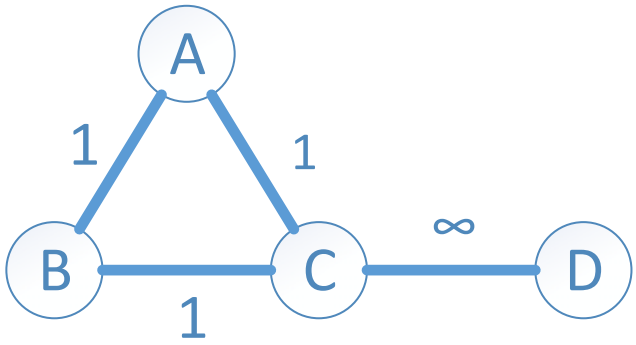


Figure 3 - Graph with Four Vertices where the Maximum Weighted Edge (C,D) is Part of the Minimum Spanning Tree

[2] A simple cycle is a closed walk in a graph with no repetition of edges or vertices other than the starting/final vertex.

Problem #7

Set Cover Problem – Given a set *U* of *n* elements, a connection of subsets (e.g. $S_1, S_2, \ldots S_m$), and an integer *k*, determine if there exists a collection of at most *k* of these subsets whose union is *U*.

Part A: Given an algorithm that solves the set cover problem, describe an algorithm to solve the corresponding optimization problem of finding the fewest number of subsets whose union is *U*.

Assign the function name, "SET-COVER-EXISTS" to the algorithm that solves whether a set-cover of size *k* exists; it takes three arguments, *U* (the set of n elements), *S* (the collection of all subsets), and *k* (the maximum set cover size). It returns a Boolean result, "TRUE" if a set cover of at most size *k* exists, and "FALSE" otherwise. Define *m* as the number of subsets.

The algorithm to return the minimum set cover size is straightforward. The algorithm, MIN-SET-COVER, is shown below. It returns -1 if no set cover exists.

```
MIN-SET-COVER(U, S){

    m = |S|                     //---- m is the number of subsets.
    k = 0                       //---- Set cover size
    SET-COVER-FOUND = FALSE     //--- Stores whether a set cover has been found

    //--- Continue looping until k ≤ m (note increment in the loop) or a set cover found
    //--- First time, set cover is found, then that is the minimum size since starting from 1
    while(k < m and SET-COVER-FOUND == FALSE){
        k = k + 1
        SET-COVER-FOUND = SET-COVER-EXISTS(U, S, k)
    }

    //---- Check whether a set cover was found and if so return k
    //---- Otherwise return -1 as error to indicate no set cover exists in collection S.
    If (SET-COVER-FOUND) Then
        return k
    Else
        return -1

}
```

The algorithm begins by checking if a collection of subsets of size 1 exists in *S* that covers *U*. If it does, then the while loop exits, and the function returns *k*. If not, then the loop continues to execute until a minimum set cover size is found or until $k > m$, which means no set cover exists in the collection of subsets, *S*.

Note this approach is not the asymptotically fastest method. You could do a binary search on the range 1 to |S|. Such an algorithm is an extension of the linear search above, with the above method sufficient to convey the concept.

Part B: Prove that the Set Cover Problem is NP-Complete.

To prove that a problem is NP-complete, two separate properties must be individually proven.

i. The language, *L*, (i.e. set of solutions) is in the set of non-deterministic polynomial problems (i.e. $L \in NP$). This is done by showing that a given solution to the problem can be verified in polynomial time.

ii. The language *L* can be formed by polynomial-time reducing all other languages in the family of NP problems. Hence:

$$\forall L'(L' \in NP \rightarrow L' \leq_P L)$$

Property ii entails proving the problem is NP Hard. By Lemma 34.8, if for a given language *L* it is shown that $L' \leq_P L$

for some $L' \in NPC$, then *L* is NP Hard. This is because the polynomial-time reduction property is transitive. As such, if a single *L'* can be polynomial-time reduced to *L*, then all other NP complete languages can be as well. Hence, when proving NP hard in this example, only one polynomial-time reduction (e.g. from vertex cover) is required.

Part B.i: Prove that the set cover problem is in the set NP by showing that a polynomial time verification algorithm exists for the problem.

Below is pseudocode for this verification algorithm called "VERIFY-SET-COVER". It takes the set of *n* elements, *U*, as the first input. The second input (i.e. the certificate) is a collection of subsets that requires verification whether it covers the set, *U*; call this collection *S'*. The function returns 1 if the collection, *S'*, covers the set and 0 otherwise.

```
VERIFY-SET-COVER(U, S'){

    n = |U|                         //---- n is the number of elements in the set U
    m = |S'|                            //---- m is the number of subsets.
    NumbCoveredElements = 0         //---- Counter the number of covered elements

    Let ElementCovered[1 to n] Be an Array of Type Boolean  //-- Array stores whether the jth element has been covered

    //---- Set all elements as uncovered
    For j = 1 to n
        ElementCovered[j] = FALSE

    //---- Iterate through each element in each subset to verify whether U is covered
    For i = 1 to m
        For j = 1 to |S'[i]|
            //--- Check whether the jth element in the ith subset has yet been covered
            If( ElementCovered[ S'[i][j] ] == FALSE )
                ElementCovered[ S'[i][j] ] = TRUE              //--- Mark element covered so only counted once
                NumbCoveredElements = NumbCoveredElements + 1  //--- Increment covered element count

    //---- Return whether all elements covered
    If(NumbCoveredElements == n)
        return 1
    else
        return 0

}
```

The algorithm "VERIFY-SET-COVER" iterates through each element in each of the *m* subsets. In the nested `If` statement, it checks whether that j^{th} element in the i^{th} subset has yet been covered. If it has, then the loop continues. If it has not yet been covered, it marks the element as covered and increments the number of covered elements. Since an element can only be covered once, if the number of elements covered by algorithm equals the total number of elements, *n*, in set *U*, then the algorithm returns 1 (i.e. binary representation of "YES"). Otherwise, it returns 0 (i.e. binary representation of "NO"). The running time of this algorithm is:

$$\theta(\text{VERIFY-SET-COVER}) = \theta(Setup) + \theta(InitArray) + \theta(SetCoverCheck) + \theta(Return)$$

$$\theta(\text{VERIFY-SET-COVER}) = \theta(1) + \theta(n) + O(n \cdot m) + \theta(1)$$

$$\theta(\text{VERIFY-SET-COVER}) = O(n \cdot m)$$

Each subset can contain a maximum of *n* elements (i.e. it cannot contain more elements than are in the set *U*), the running time of two the nested for loops is bounded by the number of subsets multiplied by the number of elements in *U*. The overall runtime of the algorithm could be improved by checking at each step in the `for` loops whether the number of covered elements ("`NumbCoveredElements`") is equal to *n*, but the simplified algorithm described above is already polynomial, and such a step would only be an optimization.

Part B.ii: Prove the set cover problem is NP Hard.

Given an undirected graph, *G*, defined as, $G = (V, E)$, where *V* is the set of vertices and *E* is the set of edges, a vertex cover is a subset of vertices *V'* ($V' \subset V$) such that for every edge $(u, v) \in E$, either $u \in V'$ and/or $v \in V'$. The vertex cover problem, which is NP complete, determines whether a graph has a vertex cover of a given size *k*.

Set cover requires that all elements of a set *U* are included in a collection of subsets. For the polynomial-time reduction, the set of edges *E* in the graph *G* map one to one to the set of elements *U*. Hence, if there are *n* elements in *E*, then there are *n* elements in *U* (i.e. $|E| = |U|$). As such, no direct conversion on the set of edges is required. If these edges were stored in an adjacency list, the running time to check all edges would be:

$$\theta(|V| + |E|)$$

which is linear.

The set of vertices, *V*, in the vertex cover problem maps one-to-one to the collection of subsets *S* in the set cover problem. Each edge is incident on either one (in a loop although vertex cover is formally on an undirected graph so a loop is generally not included) or two vertices. Store the incidence relation for each vertex in an adjacency list.[3] Each vertex's adjacency list comprises the set of elements in *U* that it contains.

k in the vertex cover problem was the maximum number of vertices required to cover all the edges. In the set cover problem, *k* maps to the number of subsets (i.e. vertices) required to cover all elements (i.e. edges) in the set *U*. No conversion on this variable is required.

Using this mapping, the vertex cover problem can be reduced to the set cover problem. Hence, if a polynomial time algorithm exists to solve the set cover problem, then that algorithm could also be used to solved the polynomial-time reduced vertex cover problem.

As shown above, very little modification of the vertex cover problem was required to reduce it to the set cover problem. The only major reduction would be to cover an adjacency matrix to an adjacency list and while polynomial, is more for notational convenience than as a hard requirement. Hence, this mapping shows the set cover problem is NP Hard.

Conclusion: Since the set cover problem was shown to be in the set of NP problems and that it is NP hard, it is NP Complete.

[3] If the edges are stored in an adjacency list, no conversion is required. In contrast, if the edges are stored in an adjacency matrix, then that matrix should be converted to an adjacency list in $\theta(|V|^2)$ time, which is done by traversing the |V| by |V| array.