Recitation 11 (HW10)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 461: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Problem 1

Problem 1 (25 points)

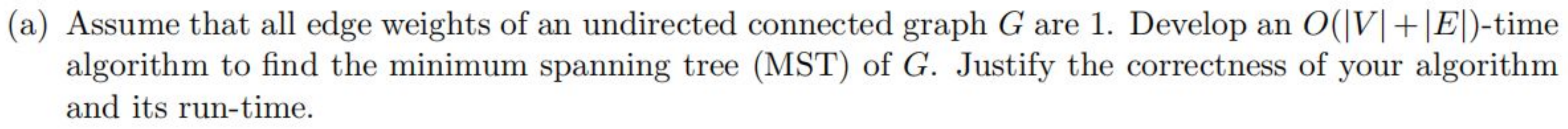
(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V|+|E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

Problem 1

(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V|+|E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

Problem 1

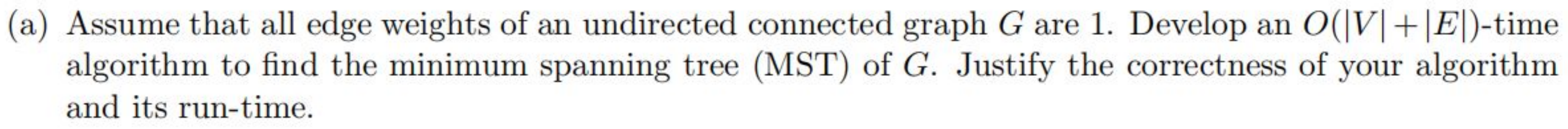


(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V|+|E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

Since the running time requirement is **O(V+E)**, the Prim / Kruskal don’t work.

Hint: What algorithms you have learnt is **O(V+E)** running time?

Problem 1

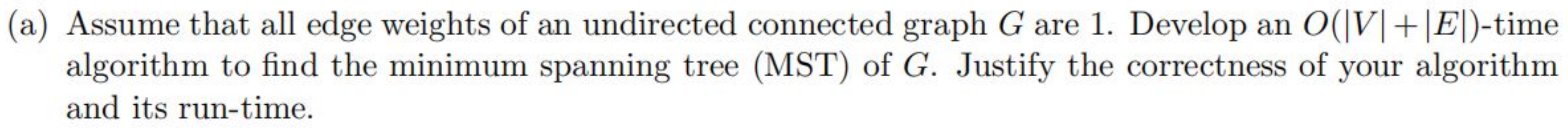


(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V|+|E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

Recall:

In any undirected connected graph, the spanning tree must be **n** nodes connected by **n-1** edges.

Problem 1

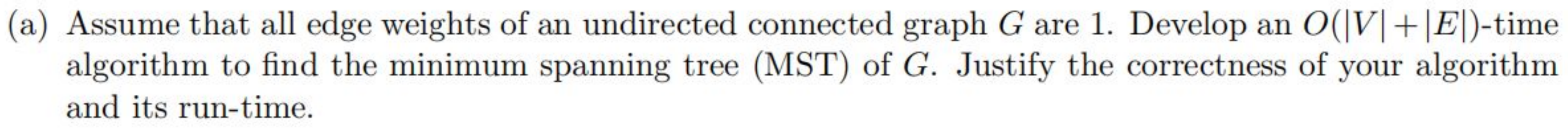

(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V|+|E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

Recall:

In any undirected connected graph, the spanning tree must be **n** nodes connected by **n-1** edges.

Since all edge weights are 1, the weight of **ANY** spanning tree must be **n-1**.

Problem 1



(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V| + |E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

Recall:

In any undirected connected graph, the spanning tree must be **n** nodes connected by **n-1** edges.

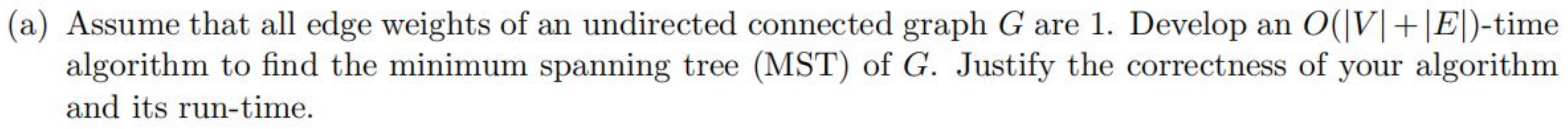
Since all edge weights are 1, the weight of **ANY** spanning tree must be **n-1**.

Thus, only need to find a spanning tree.

How?

In other words, select **n-1** edges to connect **n** nodes.

Problem 1


(a) Assume that all edge weights of an undirected connected graph G are 1. Develop an $O(|V|+|E|)$-time algorithm to find the minimum spanning tree (MST) of G. Justify the correctness of your algorithm and its run-time.

How to select **n-1** edges to connect **n** nodes?

Select all TREE edges in a DFS.

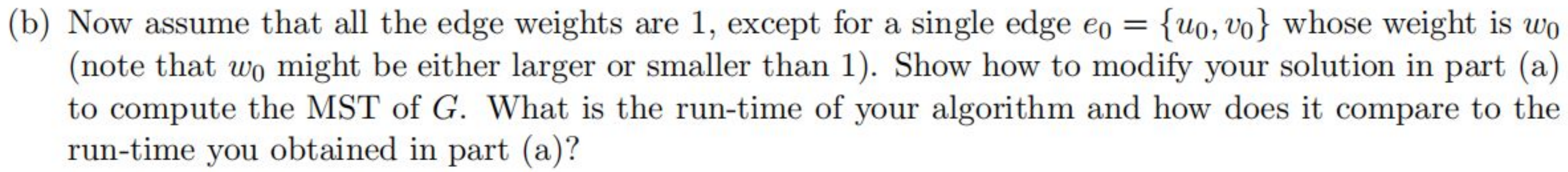
DFS(G, s) (recursive implementation)

```
O(1)  { time += 1        }
time  { s.d = time       } optional for applications
      { s.l = active     }

      for u in V[s]
O(1)      { if u.l == undiscovered
time      {     u.parent = s        ← Select edge (s,u) into MST
          {     DFS(G, u)

O(1)  { s.l = seen
time  { time += 1        }
      { s.f = time       } optional for applications
```

Problem 1

(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

Problem 1



(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

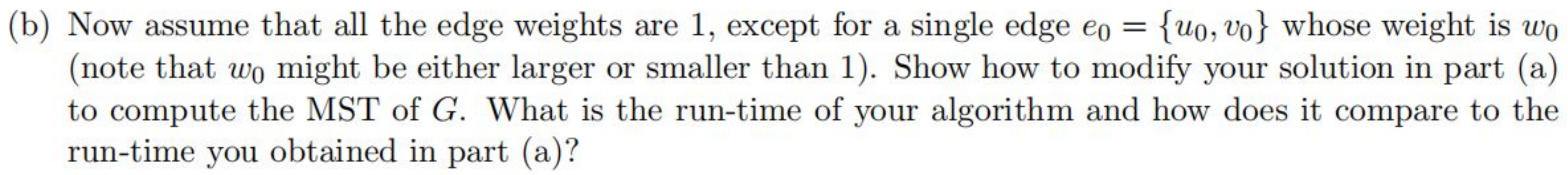
Three conditions:

W0 < 1

W0 = 1

W0 > 1

Problem 1



(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

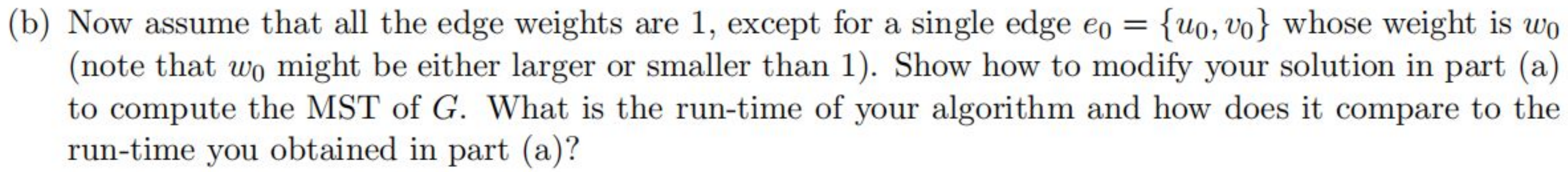
Three conditions:

$W0 < 1$

$W0 = 1$: could be covered by solution in (a)

$W0 > 1$

Problem 1


(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

W0 < 1:

This edge weight is **smaller** than all other edge weights.

Want to find MST (**minimum** weight)

Problem 1

(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

W0 < 1:

This edge weight is **smaller** than all other edge weights.

Want to find MST (**minimum** weight)

Select this edge firstly before selecting any other edge.

DFS(G, s) (recursive implementation)

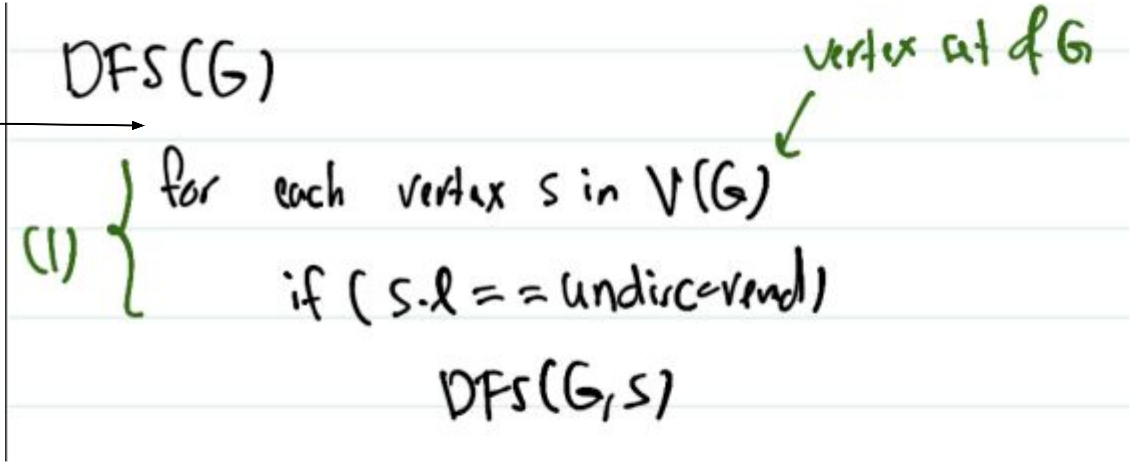
```
O(1)  {  time += 1          }
time  {  s.d = time         }  optional for applications
      {  s.l = active       }

         for u in V[s]
O(1)   {     if u.l == undiscovered
time   {         u.parent = s        <-- Select edge (s,u) into MST
       {         DFS(G, u)

O(1)  {  s.l = seen
time  {  time += 1          }  optional for applications
      {  s.f = time         }
```

How to modify?
You could also modify invocation call

```
DFS(G, s)   (recursive implementation)
O(1)  {  time += 1          }
time  {  s.d = time         }  optional for applications
      {  s.l = active       }
                                   If s == u0:
                                        v0.parent = u0
                                        Select edge (u0,v0) into MST
                                        DFS(G,v0)
    for u in V[s]
O(1)  {  if u.l == undiscovered
time  {      u.parent = s          Select edge (s,u) into MST
      {      DFS(G, u)

O(1)  {  s.l = seen
time  {  time += 1          }  optional for applications
      {  s.f = time         }
```

Problem 1

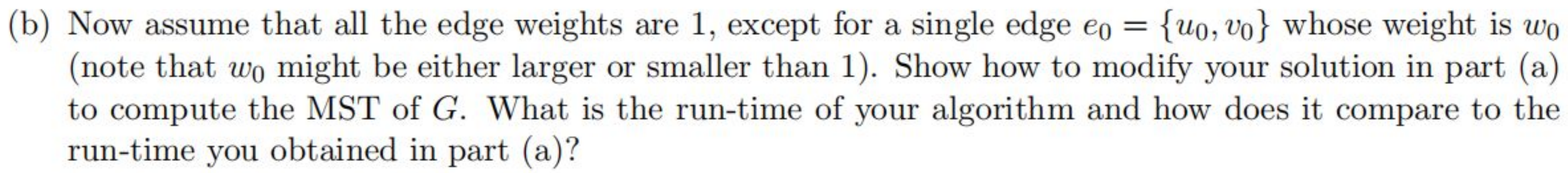
DFS(G, u0)



vertex set of G

```
DFS(G)
    for each vertex s in V(G)
(1)     if (s.l == undiscovered)
            DFS(G, s)
```

Problem 1



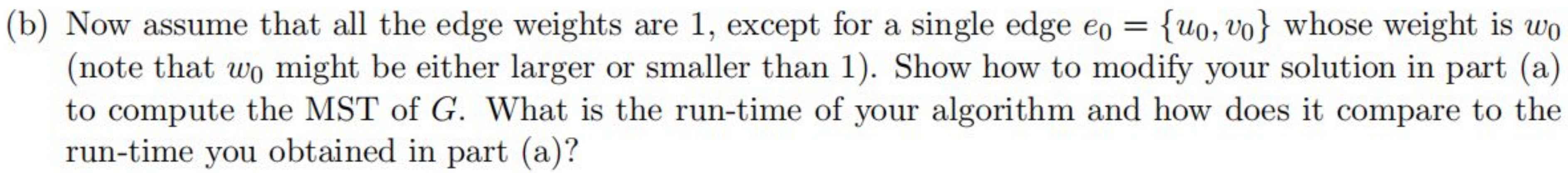
(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

W0 > 1:

This edge weight is **larger** than all other edge weights.

Want to find MST (**minimum** weight)

Problem 1



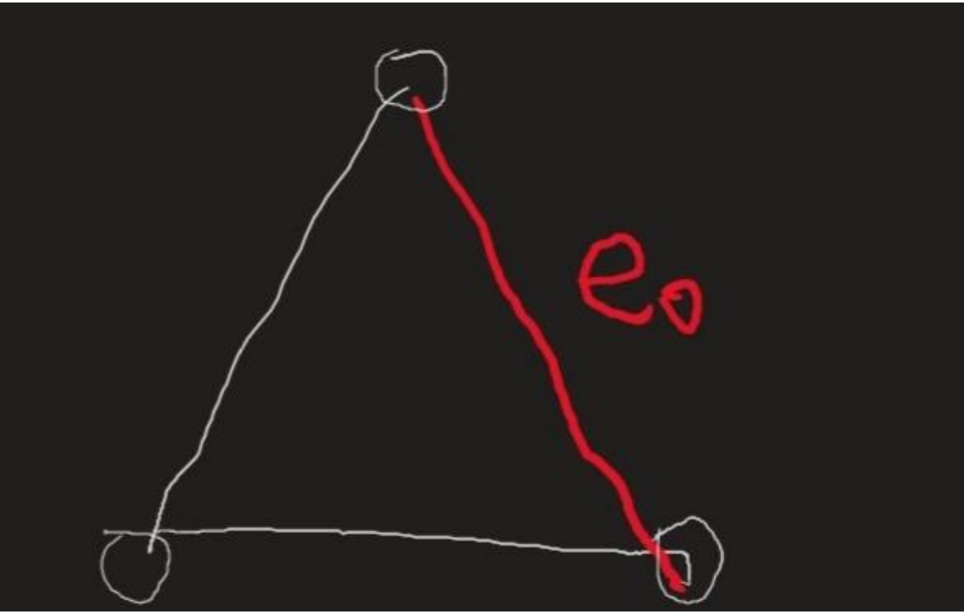
(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

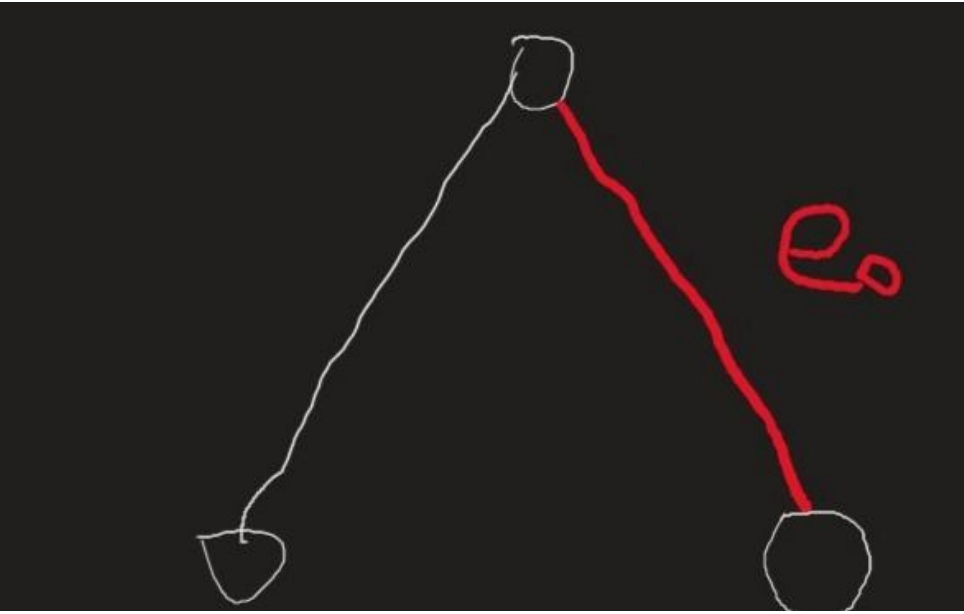
W0 > 1:

This edge weight is **larger** than all other edge weights.

Want to find MST (**minimum** weight)

Avoid selecting this edge if possible

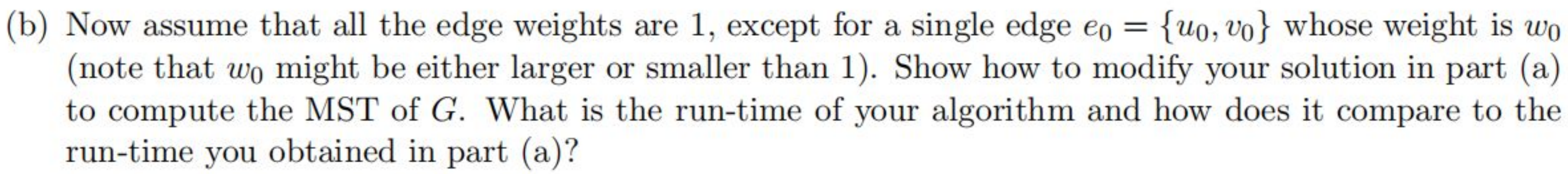




Could avoid select e0
The graph is still connected without e0

Have to select e0
Otherwise the graph is unconnected

Problem 1

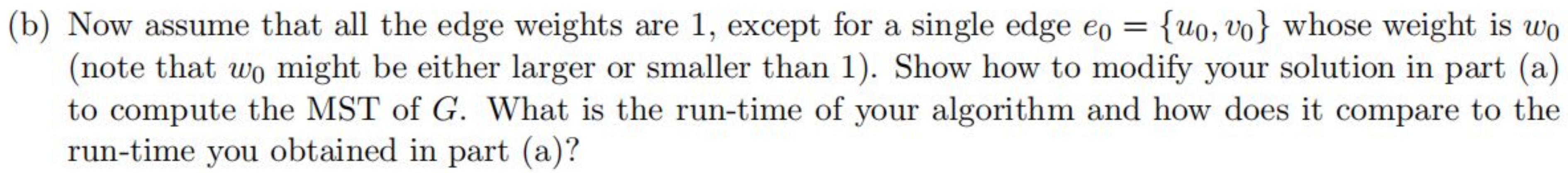


(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

W0 > 1:

1. Temporarily remove edge e0, and run DFS-modified in part (a)

Problem 1

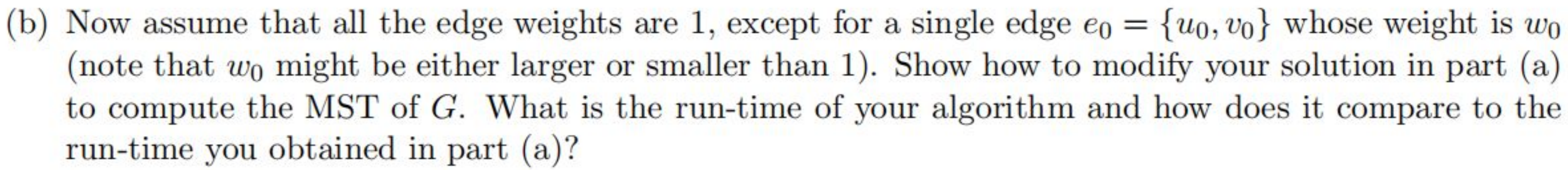


(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

W0 > 1:

1. Temporarily remove edge e0, and run DFS-modified in part (a)
2. If the graph is still connected (checked by DFS), output the edges

Problem 1

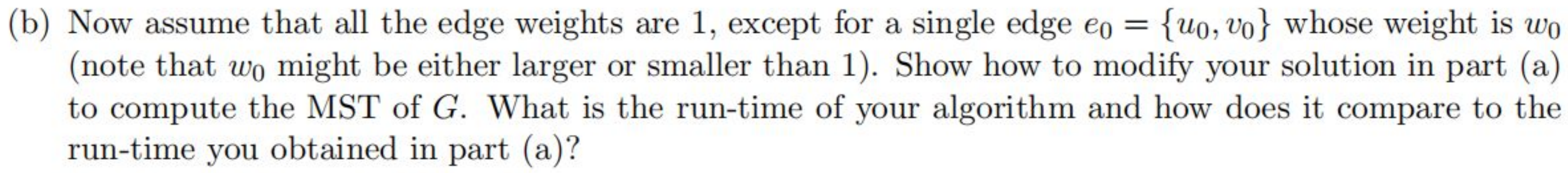


(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

W0 > 1:

1. Temporarily remove edge e0, and run DFS-modified in part (a)
2. If the graph is still connected (checked by DFS), output the edges
3. Otherwise, (have to) select e0 firstly, run the same DFS as when W0<1

Problem 1



(b) Now assume that all the edge weights are 1, except for a single edge $e_0 = \{u_0, v_0\}$ whose weight is w_0 (note that w_0 might be either larger or smaller than 1). Show how to modify your solution in part (a) to compute the MST of G. What is the run-time of your algorithm and how does it compare to the run-time you obtained in part (a)?

Running Time:

W0 < 1: One DFS with **O(1)** running time of modification, **O(V+E)**

W0 = 1: One DFS, **O(V+E)**

W0 > 1: At most two DFS, still **O(V+E)**

Problem 2

Problem 2 (25 points)

Suppose we are given an undirected graph G with weighted edges and a minimum spanning tree T of G. In each of the following cases, we change the weight of one edge e of G, and obtain the new undirected weighted graph G'. Justify why T remains an MST of the new graph G' in each of these cases.

(a) The weight of one edge $e \in T$ is decreased.

(b) The weight of one edge $e \notin T$ is increased.

Problem 2

(a) The weight of one edge $e \in T$ is decreased.

$e \in T$ and T is the minimum spanning tree of G. The weight of e decreased and e became a new edge e’.

Let $G' = G - \{e\} + \{e'\}$, $T' = T - \{e\} + \{e'\}$.

T’ has the same number of edges as T. Then T’ remains a spanning tree of G’.

$\text{Weight}(T') < \text{Weight}(T)$. Then T’ becomes the new minimum spanning tree of G’.

Problem 2

(b) The weight of one edge e ∉ T is increased.

Assume there is a spanning tree with edge e, say Ts, weight(Ts) >= weight(T).

If the weight of e was increased, then for the new spanning tree Ts’, weight(Ts’) > weight(Ts) >= weight(T).

T remains the minimum weight, making it remain the minimum spanning tree.

Problem 3

Problem 3 (25 points)

Provide a counterexample to the following false claim. Fully justify your answer.
If a directed graph G contains cycles (i.e., is not acyclic), then topological sort produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced. More precisely, a bad edge is an edge going from a vertex later in the ordering to an earlier vertex.

Problem 3

$$G = (V, E)$$

$$V(G) = \{c, a, b, d\}$$

$$\begin{tikzcd}
 & b \arrow[dr] & \\
a \arrow[ur] \arrow[dr] & & c \arrow[ll] \\
 & d \arrow[ur] &
\end{tikzcd}$$

Problem 3

$$\begin{tikzcd}
 & \overset{2,5}{b} \arrow[dr] & \\
1,8 \ a \arrow[ur] \arrow[dr] & & c \ 3,4 \arrow[ll] \\
 & \underset{6,7}{d} \arrow[ur] &
\end{tikzcd}$$

① DFS(G, a)

② topological sort :

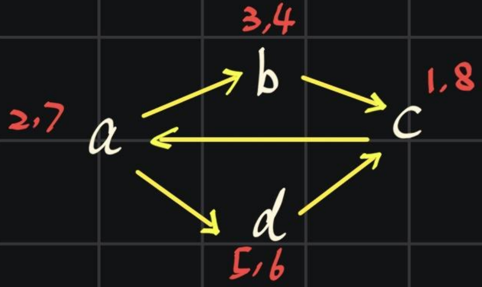
(decreasing finish time)

$$\begin{cases} a - 8 \\ d - 7 \\ b - 5 \\ c - 4 \end{cases}$$

③ bad edges: $\begin{cases} c \rightarrow a \end{cases}$

Problem 3

counterexample:



① DFS(G, c)

② topological sort:
(decreasing finish time)

$$\begin{cases} c - 8 \\ a - 7 \\ d - 6 \\ b - 4 \end{cases}$$

③ bad edges: $\begin{cases} b \to c \\ d \to c \end{cases}$

Problem 4

Problem 4 (25 points)

A directed graph G is called *semi-connected* if, for all pairs of vertices u and v in G, we have a path from u to v **or** a path from v to u (or both).

Note: Recall that in a connected graph, we must have a path from u to v **and** a path from v to u.

(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

(b) Justify the correctness and run-time of your algorithm.

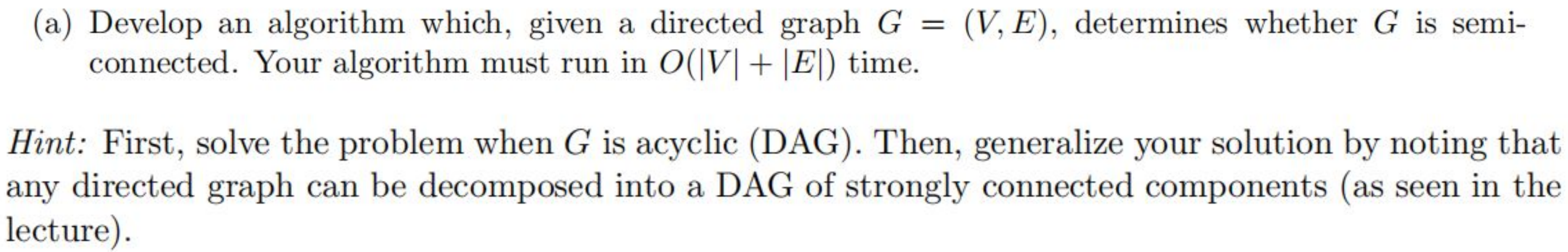
Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

Problem 4

(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

Problem 4



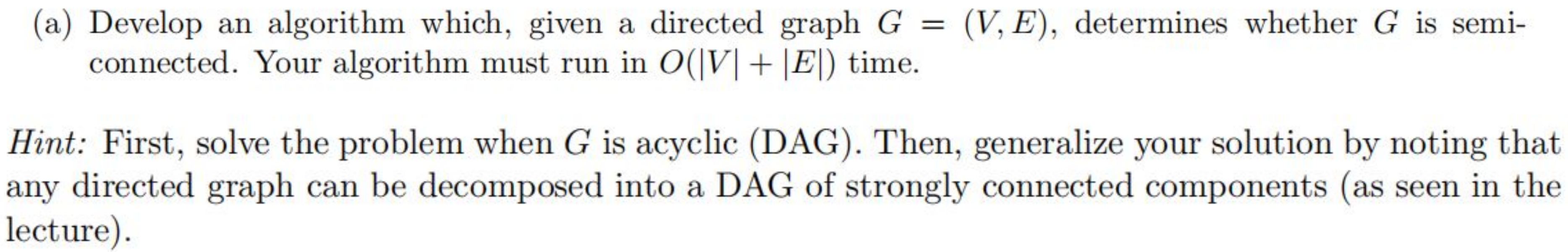
(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

Consider when G is a DAG:

There couldn't both exist a path from u to v **AND** a path from v to u.

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

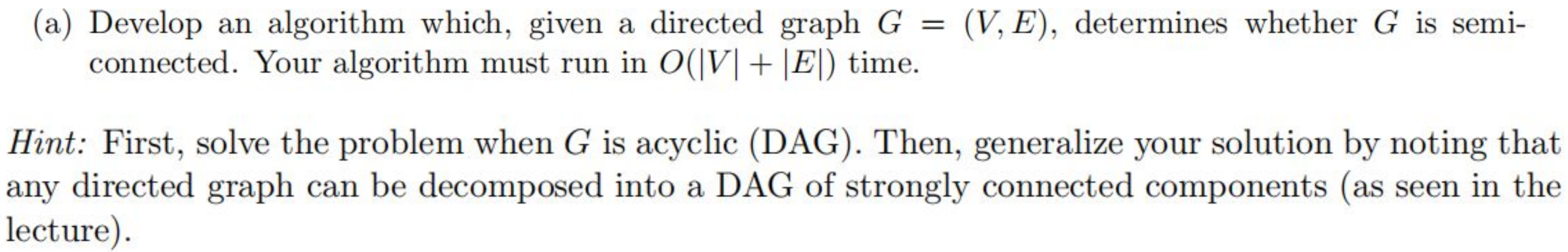
Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

Consider when G is a DAG:

There couldn't both exist a path from u to v **AND** a path from v to u.

If G is semi-connected, either a path from u to v **OR** a path from v to u for all pairs (u,v)

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

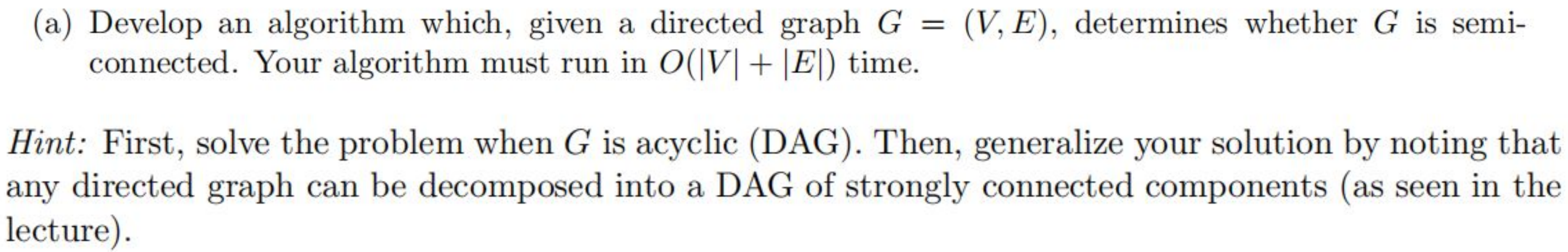
Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

If G is semi-connected, either a path from u to v **OR** a path from v to u for **all** pairs (u,v)

The order of Topological Sort: A path from u to v is considered as U > V

What can you infer about the comparison for any (U,V)?

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

If G is semi-connected, either a path from u to v **OR** a path from v to u for **all** pairs (u,v)

The order of Topological Sort: A path from u to v is considered as U > V

What can you infer about the comparison for any (U,V)? **Either U>V or V>U**

What can you infer about the order of all vertices?

$$\begin{tikzcd}
 & A \arrow[dl] & \\
B & & C \arrow[ll]
\end{tikzcd}$$

Not Semi-Connected

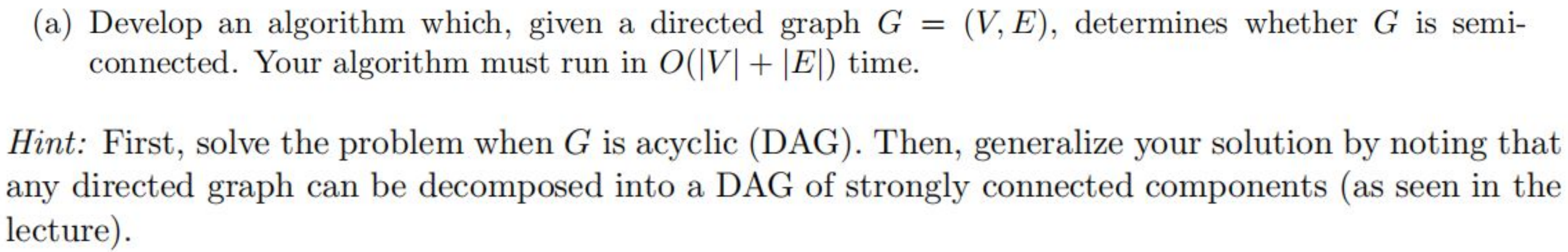
Topological Sort order could be either A C B / C A B

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B & & C \arrow[ll]
\end{tikzcd}$$

Semi-Connected

UNIQUE Topological Sort order: A C B

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

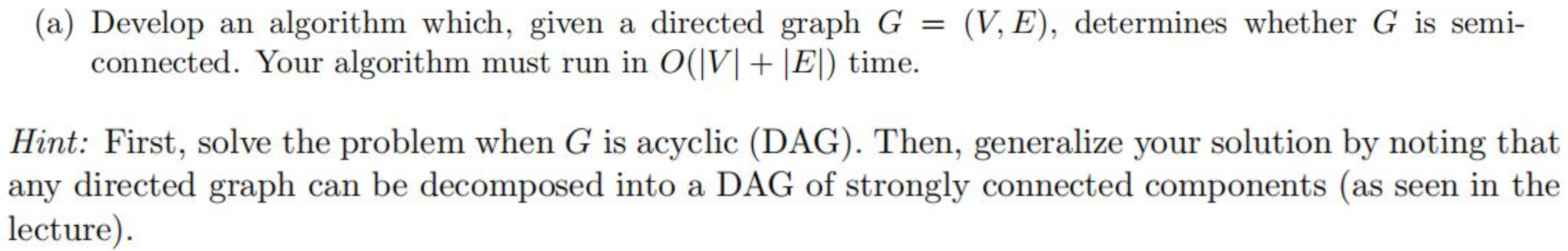
If G is semi-connected, either a path from u to v **OR** a path from v to u for **all** pairs (u,v)

The order of Topological Sort: A path from u to v is considered as U > V

What can you infer for any (U,V)? **For any pair (u,v), either U>V or V>U**

What can you infer about the order of all vertices? **UNIQUE Topological Sort order**

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

How to check whether the order of Topological Sort is UNIQUE?

$$\begin{tikzcd}
& A \arrow[dl] & \\
B & & C \arrow[ll]
\end{tikzcd}$$

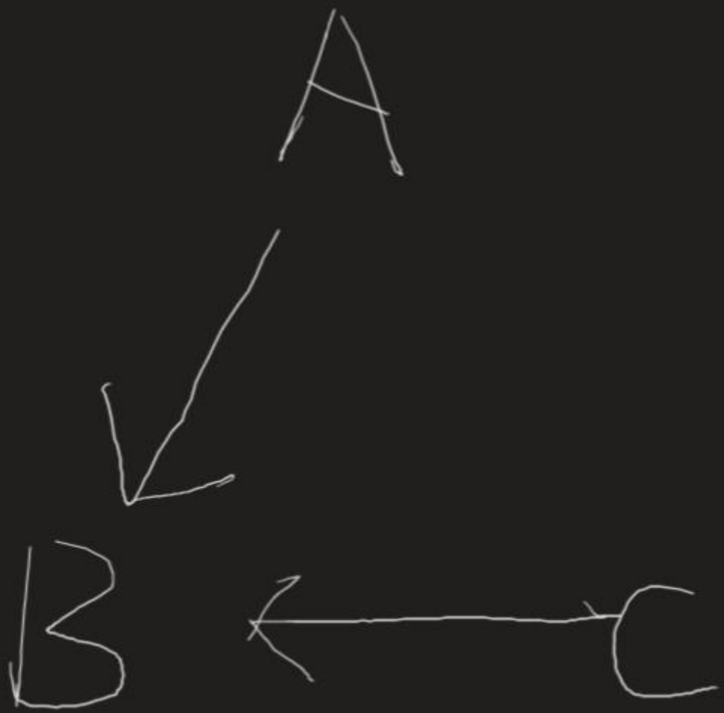
non-UNIQUE Topological Sort order

No edge between A and C

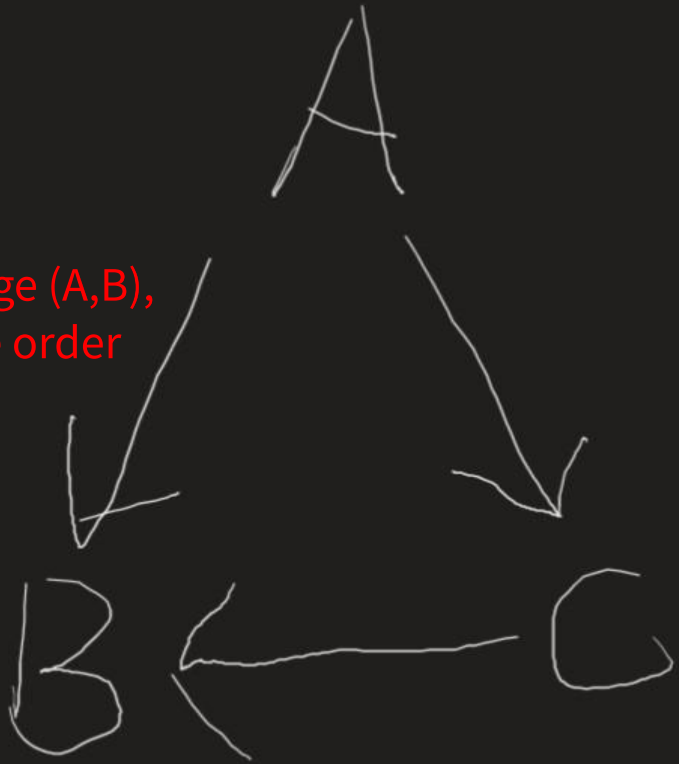
$$\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & \\
B & & C \arrow[ll]
\end{tikzcd}$$

UNIQUE Topological Sort order

A directed edge from A to C



Without edge (A,B),
Still Unique order



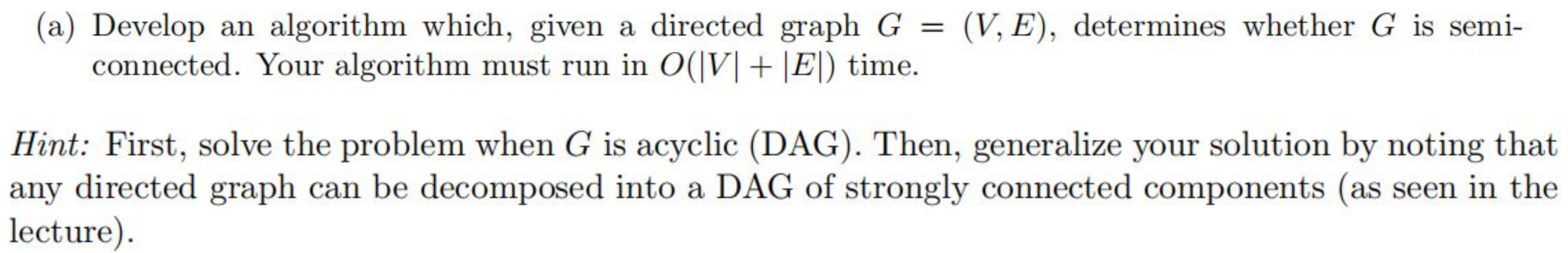
non-UNIQUE Topological Sort order

No edge between A and C

UNIQUE Topological Sort order

A directed edge from A to C

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

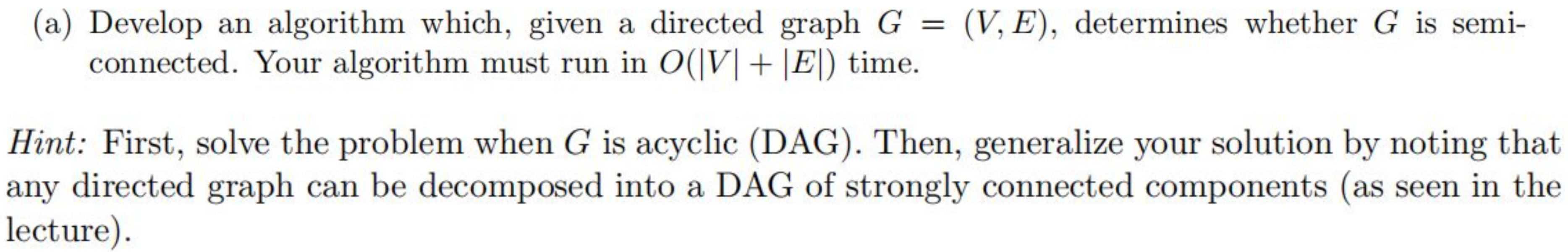
Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

How to check whether the order of Topological Sort is UNIQUE?

For a Topological Sort order **A, B, C, D, ...,** check whether there exists directed edges **(A,B) (B,C) (C,D) ...**

If any of these edge doesn't exist, the order is non-UNIQUE, thus not semi-connected

Problem 4

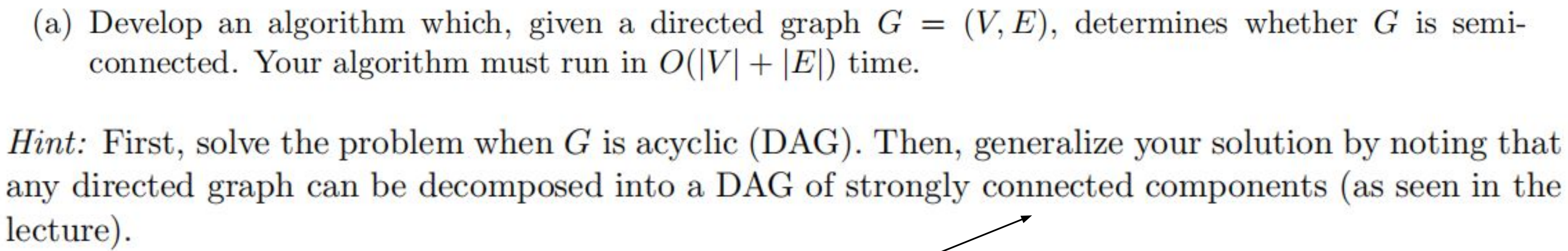


(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

What if the G is not DAG?

Problem 4



(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

What if the G is not DAG?

Decompose G into a DAG of SCC.

Problem 4

(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

What if the G is not DAG?

Decompose G into a DAG of SCC.

Since nodes within one SCC are connected to each other, if SCC_1 and SCC_2 are semi-connected, then all nodes within SCC_1 are semi-connected to those within SCC_2

Problem 4

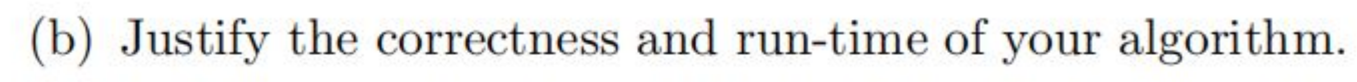
(a) Develop an algorithm which, given a directed graph $G = (V, E)$, determines whether G is semi-connected. Your algorithm must run in $O(|V| + |E|)$ time.

Hint: First, solve the problem when G is acyclic (DAG). Then, generalize your solution by noting that any directed graph can be decomposed into a DAG of strongly connected components (as seen in the lecture).

Overall Solutions:

1. Decompose G into a DAG of SCC, call it G’
2. Compute the topological sort order of G’
3. Check whether the order is UNIQUE

Problem 4

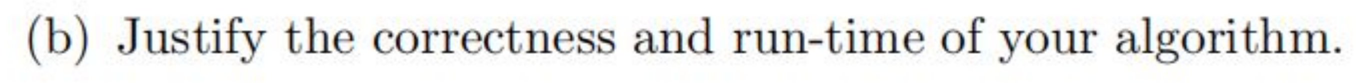

(b) Justify the correctness and run-time of your algorithm.

Running time:

Overall Solutions:

1.	Decompose G into a DAG of SCC, call it G’	O(V+E)
2.	Compute the topological sort order of G’	O(V+E)
3.	Check whether the order is UNIQUE	O(V+E)

Problem 4


(b) Justify the correctness and run-time of your algorithm.

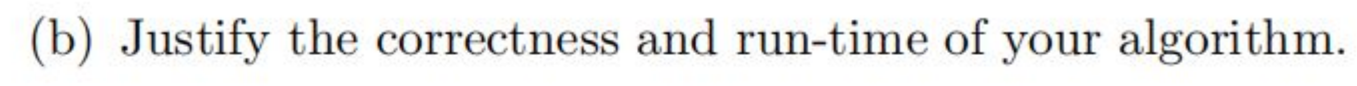
Correctness:

If a DAG has unique topo sort order, we can see it is semi-connected graph. Why?

Consider a DAG has two topo sort orders: [A B C D] and [A C B D]

We then can infer that B and C are at the same level of topo sort. Since the graph is a DAG, there cannot be any path between B and C, which means the graph is not a semi-connected graph.

Problem 4


(b) Justify the correctness and run-time of your algorithm.

Correctness:

For a DAG, assume the topological sort order is **A, B, C, D…**

If there exists edges (A,B) (B,C) (C,D) …, all nodes are semi-connected with each other.

Otherwise, assume no edge (A,B), and as definition of DAG, there couldn't be edge (B,A).

Thus, A and B are not semi-connected.   (And it applies to all other conditions)

Problem 4

To be concluded:

Given a directed graph G, a DFS traversal on the vertex set of G can be interpreted as a DFS traversal on the component graph of G, i.e., G^{SCC}.

Thus, a topological sorting of the vertices of G can be interpreted as a topological sorting of the component vertices of G^{SCC}, and since G^{SCC} is a DAG, this gives a sorting of component vertices in such a way that all the edges in G^{SCC} are drawn from left ro right (all this has been already discussed in the lecture).

Problem 4

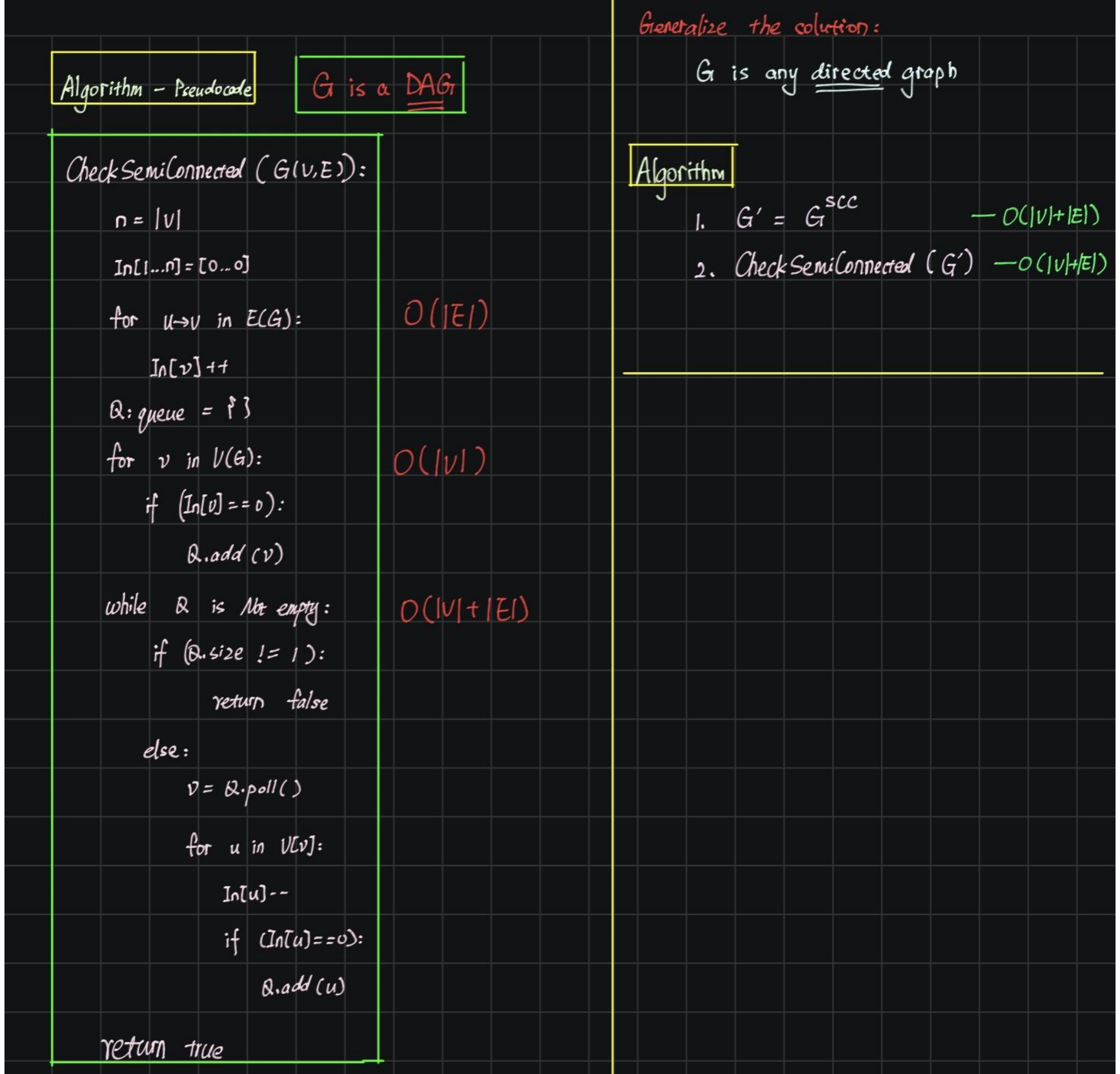
Now given this topological sorting of the component vertices of G^{SCC}, one should note that G is semi-connected iff there is an edge between any two consecutive component vertices in this topologically sorted order.

The reason is that if X and Y are two consecutive component vertices in the order, then to make sure that the vertices in X and Y are semi-connected, there must be an edge from X to Y. To see this, note that all the edges in the component graph G^{SCC} are drawn from left to right, so there must be always an edge from the leftmost component vertex to the next component vertex. This argument can be iterated and extended for any two consecutive component vertices.

Problem 4

For your reference -

Another approach to solve this problem:



Algorithm - Pseudocode

G is a DAG

```
CheckSemiConnected ( G(V,E) ):
    n = |V|
    In[1...n] = [0..0]
    for u→v in E(G):                  O(|E|)
        In[v]++
    Q: queue = {}
    for v in V(G):                    O(|V|)
        if (In[v] == 0):
            Q.add(v)
    while Q is Not empty:             O(|V|+|E|)
        if (Q.size != 1):
            return false
        else:
            v = Q.poll()
            for u in V[v]:
                In[u]--
                if (In[u]==0):
                    Q.add(u)
    return true
```

Generalize the solution:

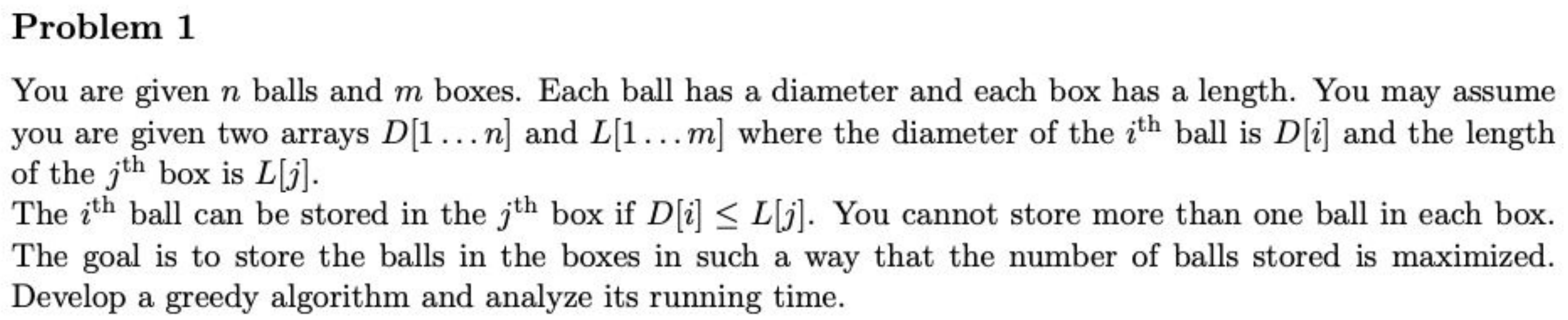
G is any directed graph

Algorithm

1. $G' = G^{SCC}$ — $O(|V|+|E|)$
2. CheckSemiConnected (G') — $O(|V|+|E|)$

Additional Practice Problems

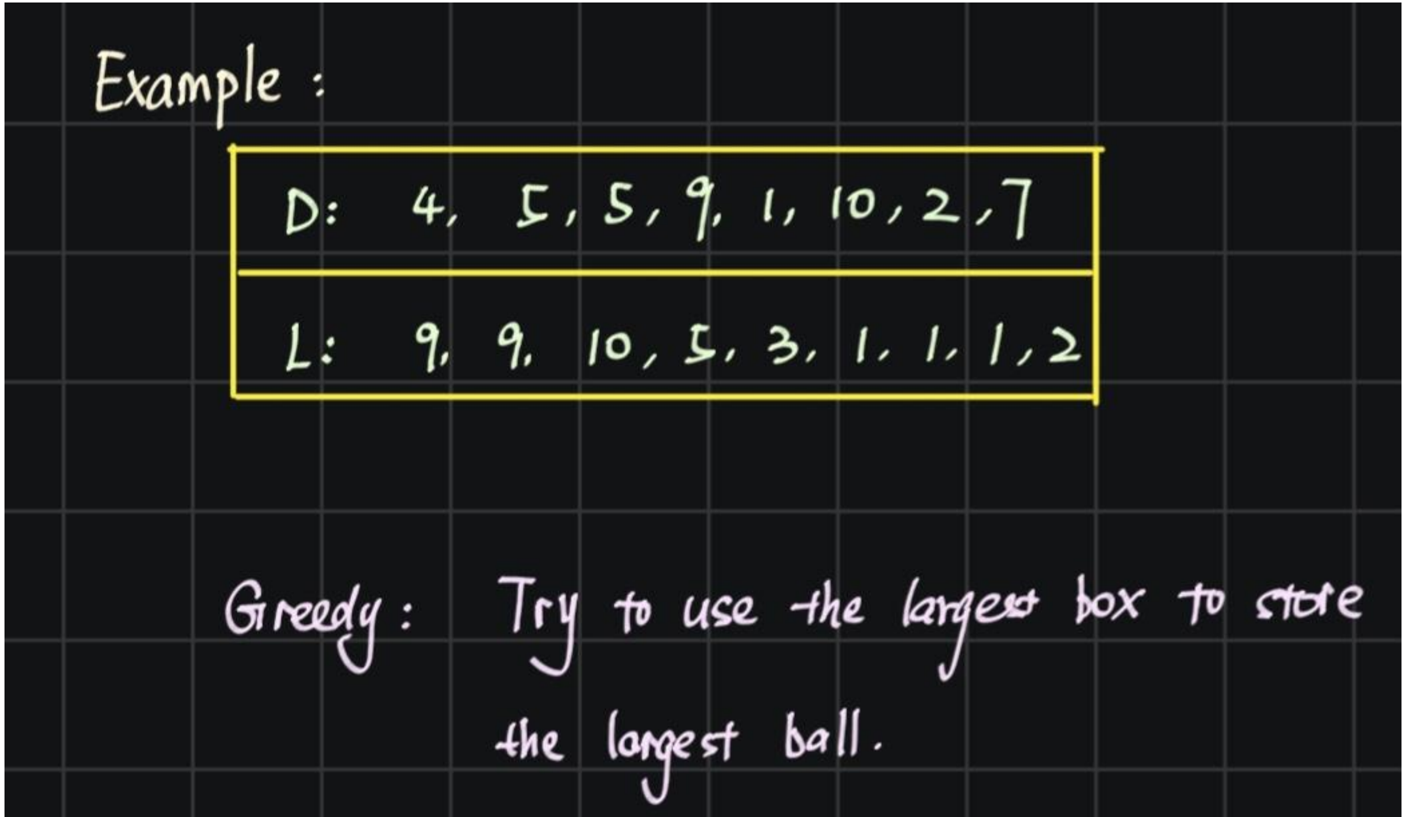
Problem 1



Problem 1

You are given n balls and m boxes. Each ball has a diameter and each box has a length. You may assume you are given two arrays $D[1\ldots n]$ and $L[1\ldots m]$ where the diameter of the i^{th} ball is $D[i]$ and the length of the j^{th} box is $L[j]$.
The i^{th} ball can be stored in the j^{th} box if $D[i] \leq L[j]$. You cannot store more than one ball in each box. The goal is to store the balls in the boxes in such a way that the number of balls stored is maximized. Develop a greedy algorithm and analyze its running time.

Problem 1



Example :

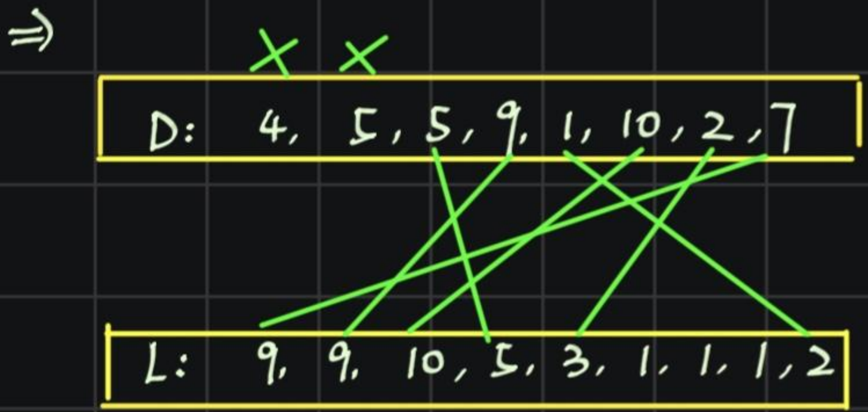
D:	4,	5,	5,	9,	1,	10,	2,	7	
L:	9,	9,	10,	5,	3,	1,	1,	1,	2

Greedy : Try to use the largest box to store the largest ball.

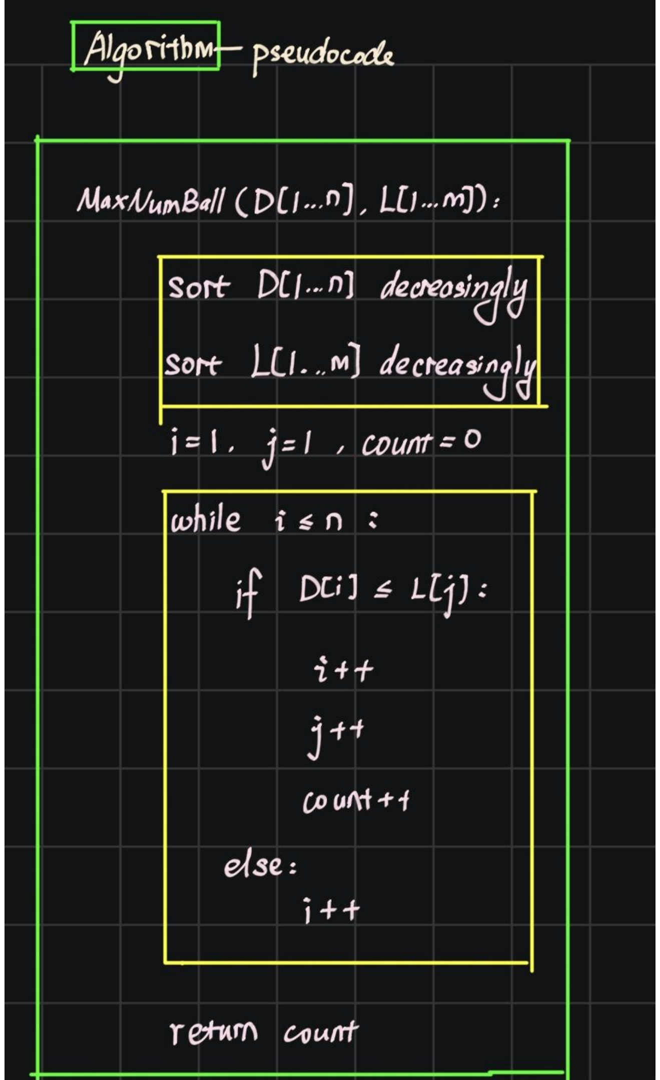
Problem 1

Greedy: Try to use the largest box to store the largest ball.

⇒



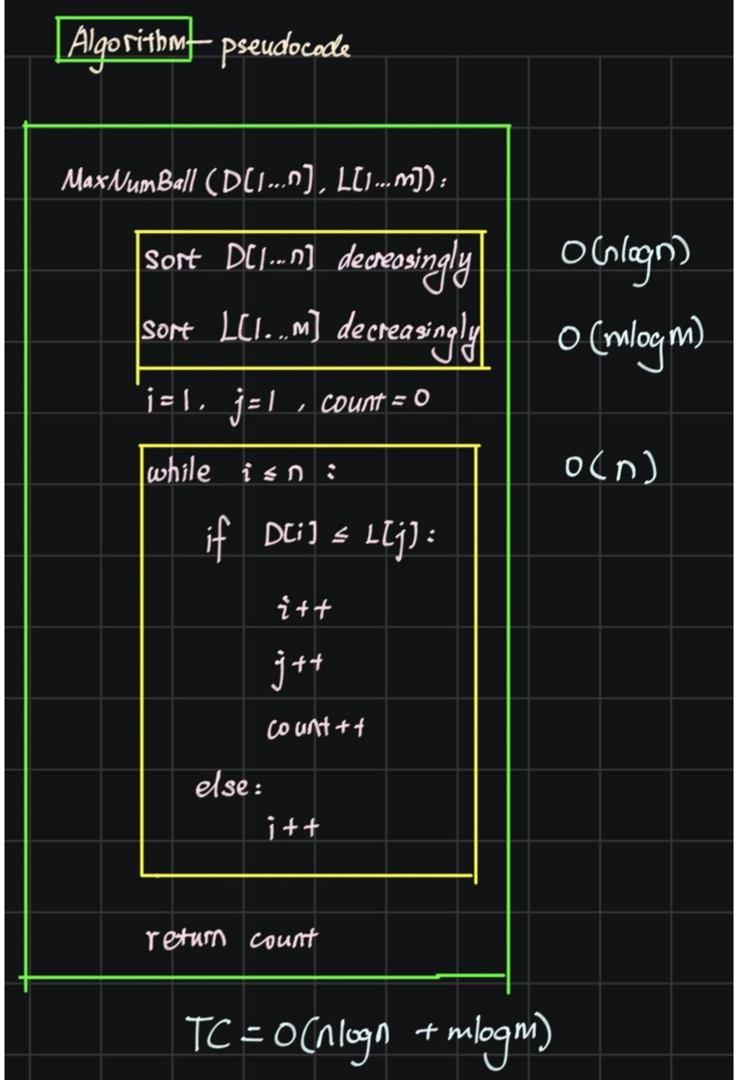
Problem 1



Algorithm – pseudocode

```
MaxNumBall (D[1...n], L[1...m]):
    sort D[1...n] decreasingly
    sort L[1...m] decreasingly
    i = 1, j = 1, count = 0
    while i ≤ n :
        if D[i] ≤ L[j] :
            i++
            j++
            count++
        else:
            i++
    return count
```

Problem 1

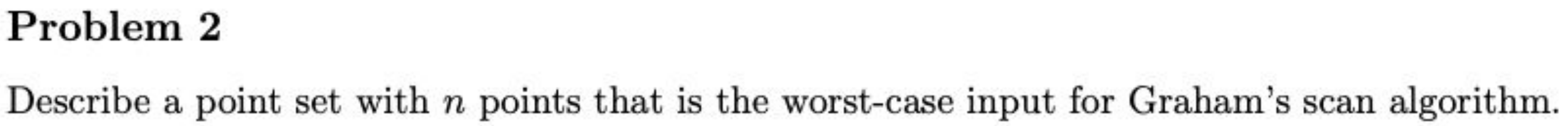


Algorithm — pseudocode

```
MaxNumBall (D[1...n], L[1...m]):
    sort D[1...n] decreasingly          O(nlogn)
    sort L[1...m] decreasingly          O(mlogm)
    i=1, j=1, count=0
    while i ≤ n :                       O(n)
        if D[i] ≤ L[j]:
            i++
            j++
            count++
        else:
            i++
    return count
```

$TC = O(n\log n + m\log m)$

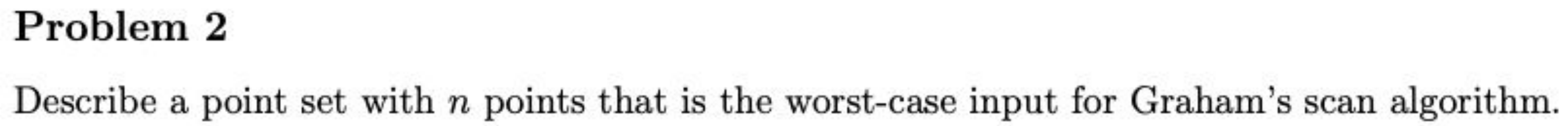
Problem 2



Problem 2

Describe a point set with n points that is the worst-case input for Graham's scan algorithm.

Problem 2



Problem 2

Describe a point set with n points that is the worst-case input for Graham's scan algorithm.

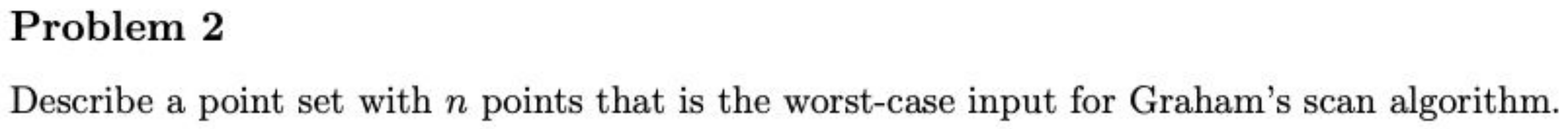
In the Graham's scan, there are different types of operations:

1- operations needed to sort out n-1 points

2- adding each point once to the stack

3- removing some of the points at most once from the stack

Problem 2



Problem 2

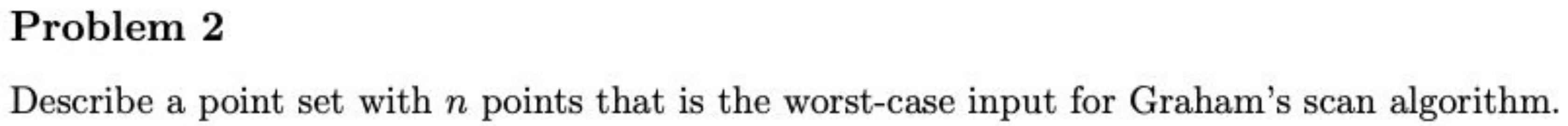
Describe a point set with n points that is the worst-case input for Graham's scan algorithm.

The number of operations for (1) becomes maximal depending on how the points are given as an input.

More precisely, if the input points are given in the reverse order as they appear in the sorted order in step (1).

This part only depends on how the input points are given to us.

Problem 2



Problem 2

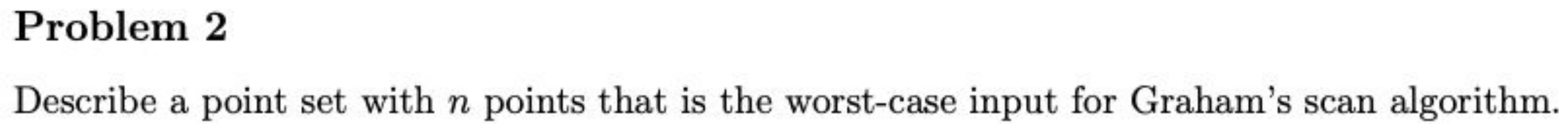
Describe a point set with n points that is the worst-case input for Graham's scan algorithm.

For part (2), we have exactly n operations which cannot be increased/decreased.

For part (3), the number of removals becomes maximized when almost all the points are removed.

Note that we need at least 3 points to remain in the stack since there are always at least three points on the boundary of any convex hull.

Problem 2



Problem 2

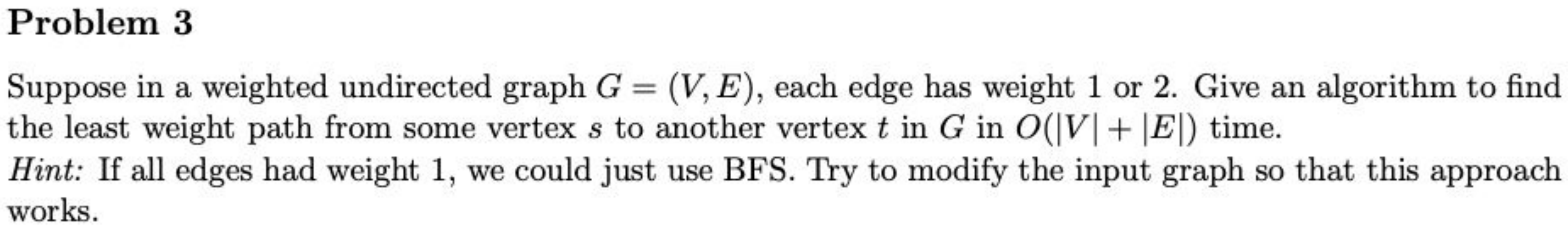
Describe a point set with n points that is the worst-case input for Graham's scan algorithm.

Thus, an example of a point set giving the worst-case for Graham's scan is:

A set of n points whose convex hull only consists of 3 points.

Any such point set gives the worst running time.

Problem 3



Problem 3

Suppose in a weighted undirected graph $G = (V, E)$, each edge has weight 1 or 2. Give an algorithm to find the least weight path from some vertex s to another vertex t in G in $O(|V| + |E|)$ time.
Hint: If all edges had weight 1, we could just use BFS. Try to modify the input graph so that this approach works.

Problem 3

If all edges had weight 1, we could just use BFS.

Why?

One important observation about BFS: the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path.

Running time: O(|V|+|E|)

Problem 3

For this problem, all edges are of weight either 1 or 2.

Can we transfer this graph to a new graph with all edges of weight 1? How?

G -> G’

Problem 3

We can modify the graph and split all edges of weight 2 into two edges of weight 1 each.

In the modified graph G’, we can use BFS(G’) to find the shortest path. Running time: O(|V|+|E|)

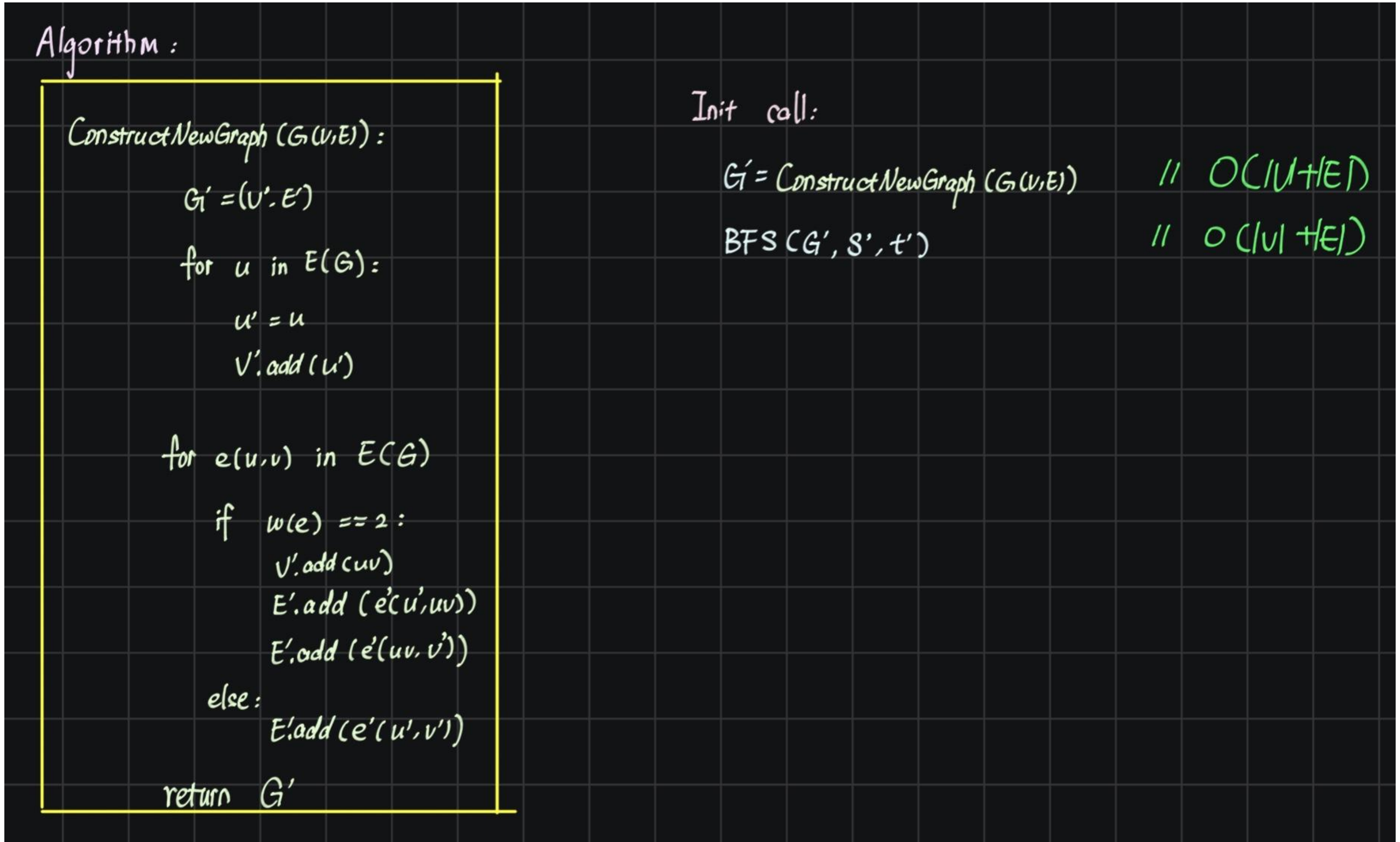
How to construct G’?

For each pair (u,v) whose edge(u,v) = 2:

Add intermediate vertex, uv;

Add one edge weighted 1 for (u, uv); add one edge weighted 1 for (uv, v).

Problem 3



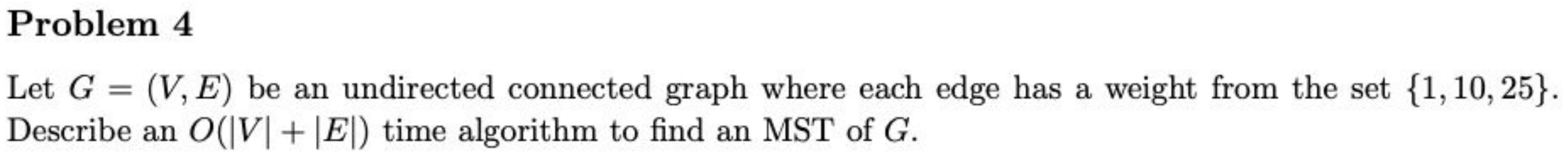
Algorithm :

```
ConstructNewGraph (G(V,E)) :
        G' = (V', E')
        for u in E(G) :
            u' = u
            V'.add (u')

        for e(u,v) in E(G)
            if w(e) == 2 :
                V'.add (uv)
                E'.add (e'(u', uv))
                E'.add (e'(uv, v'))
            else :
                E'.add (e'(u', v'))
        return G'
```

Init call:

```
G' = ConstructNewGraph (G(V,E))     // O(|V|+|E|)
BFS (G', S', t')                    // O(|V|+|E|)
```

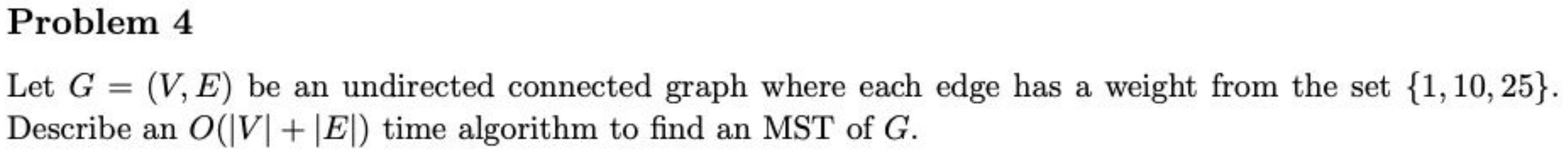
Problem 4



Problem 4

Let $G = (V, E)$ be an undirected connected graph where each edge has a weight from the set $\{1, 10, 25\}$. Describe an $O(|V| + |E|)$ time algorithm to find an MST of G.

Problem 4



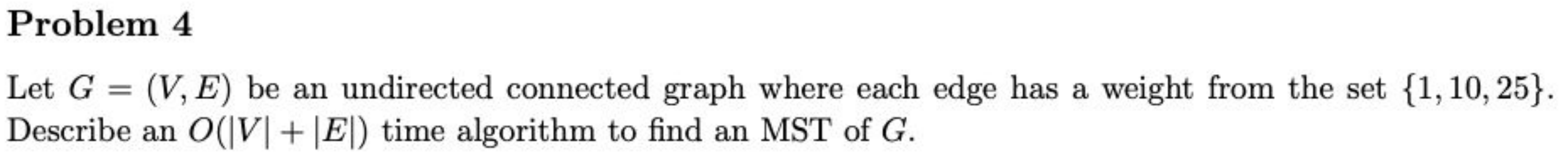
Problem 4

Let $G = (V, E)$ be an undirected connected graph where each edge has a weight from the set $\{1, 10, 25\}$. Describe an $O(|V| + |E|)$ time algorithm to find an MST of G.

The general algorithm: Kruskal algorithm, running time **O(E log E)**.

This is because we have to sort all edges at first.

Problem 4



Problem 4

Let $G = (V, E)$ be an undirected connected graph where each edge has a weight from the set $\{1, 10, 25\}$. Describe an $O(|V| + |E|)$ time algorithm to find an MST of G.

The general algorithm: Kruskal algorithm, running time **O(E log E)**.

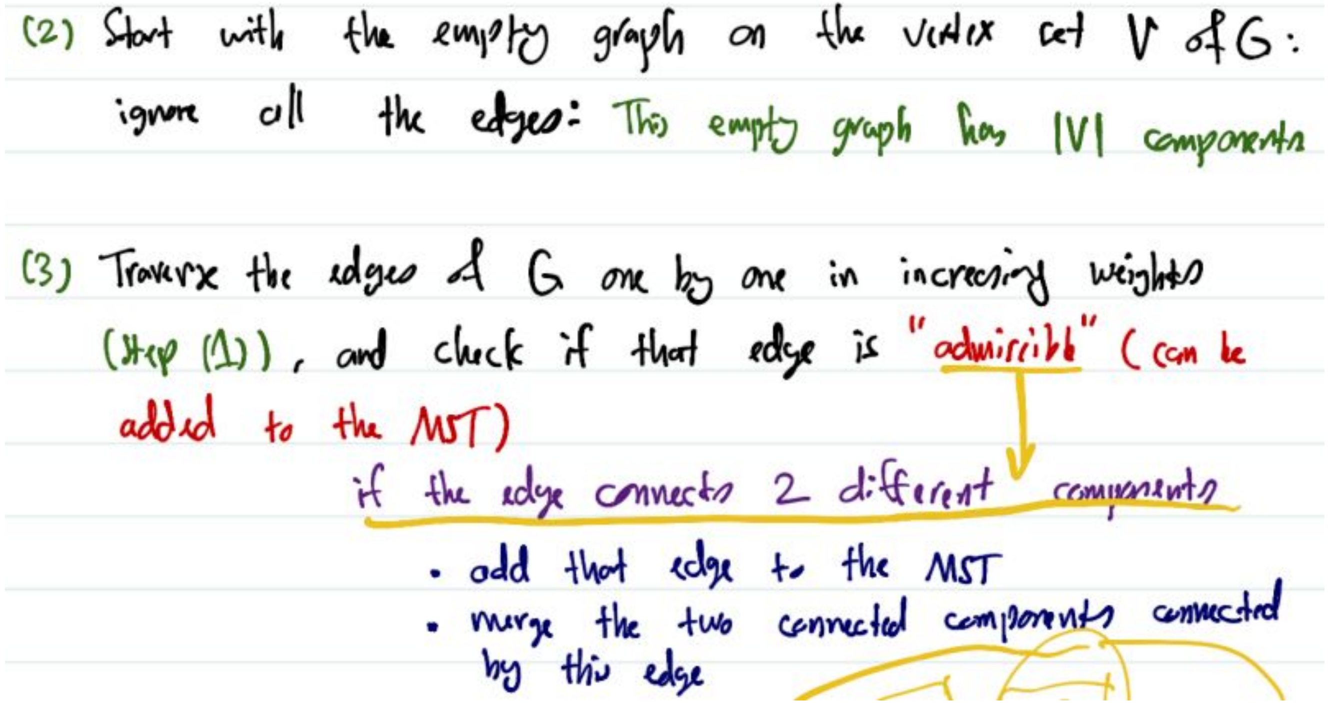
This is because we have to sort all edges at first.

In this problem, the weight could only be 1/10/25.

Thus, instead of sorting, we divide edges into 1-set, 10-set and 25-set. Time: **O(E)**

Problem 4

After that, repeat the following steps three times (as lectures notes):



(2) Start with the empty graph on the vertex set V of G:
ignore all the edges: This empty graph has $|V|$ components

(3) Traverse the edges of G one by one in increasing weights
(step (1)), and check if that edge is "admissible" (can be
added to the MST)

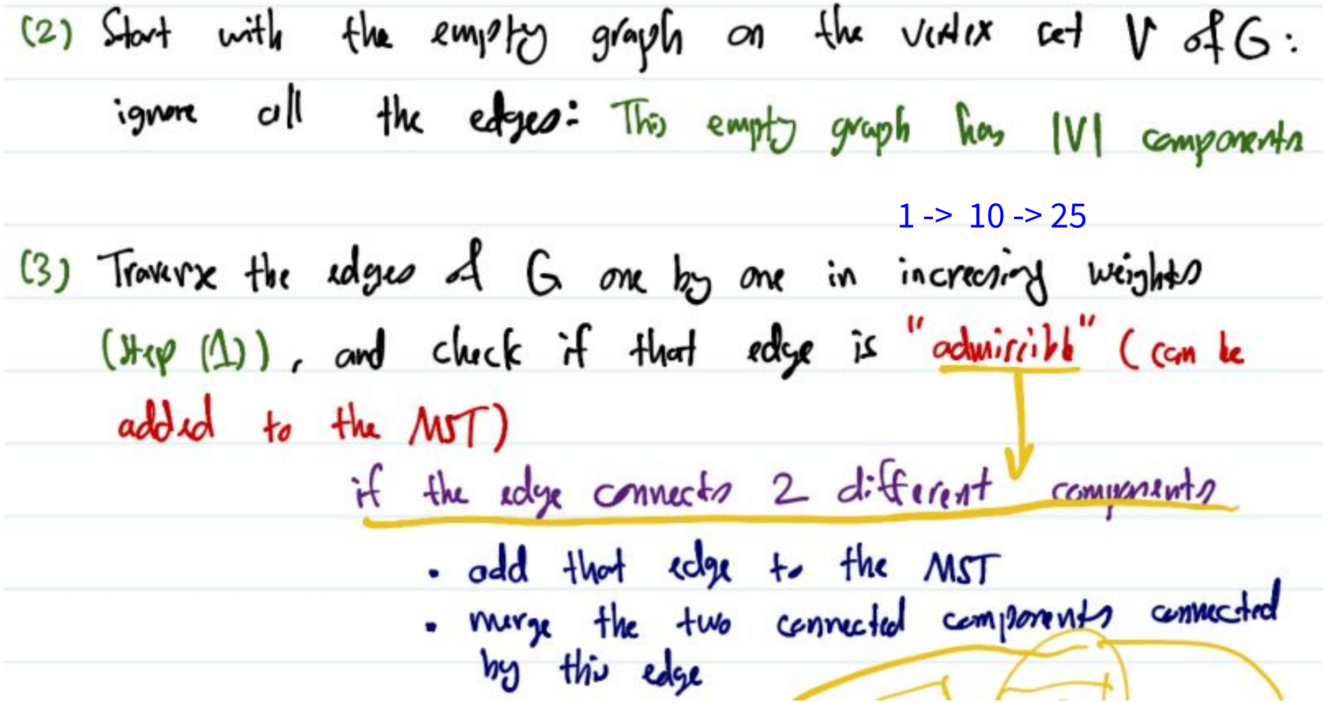
if the edge connects 2 different components

- add that edge to the MST
- merge the two connected components connected by this edge

Problem 4

After that, repeat the following steps three times (as lectures notes):

1. Consider all edges in 1-set
2. Then Consider all edges in 10-set (while keeping the result of previous one)
3. Consider all edges in 25-set (same as above)



(2) Start with the empty graph on the vertex set V of G:

ignore all the edges: This empty graph has $|V|$ components

1 -> 10 -> 25

(3) Traverse the edges of G one by one in increasing weights (step (1)), and check if that edge is "admissible" (can be added to the MST)

if the edge connects 2 different components

- add that edge to the MST
- merge the two connected components connected by this edge

Q & A

Thank you