CSCI 232:
Data Structures and Algorithms

Minimum Spanning Tree (MST) Part 2

Reese Pearsall

Spring 2024

Announcements

Lab 9 due **tomorrow**

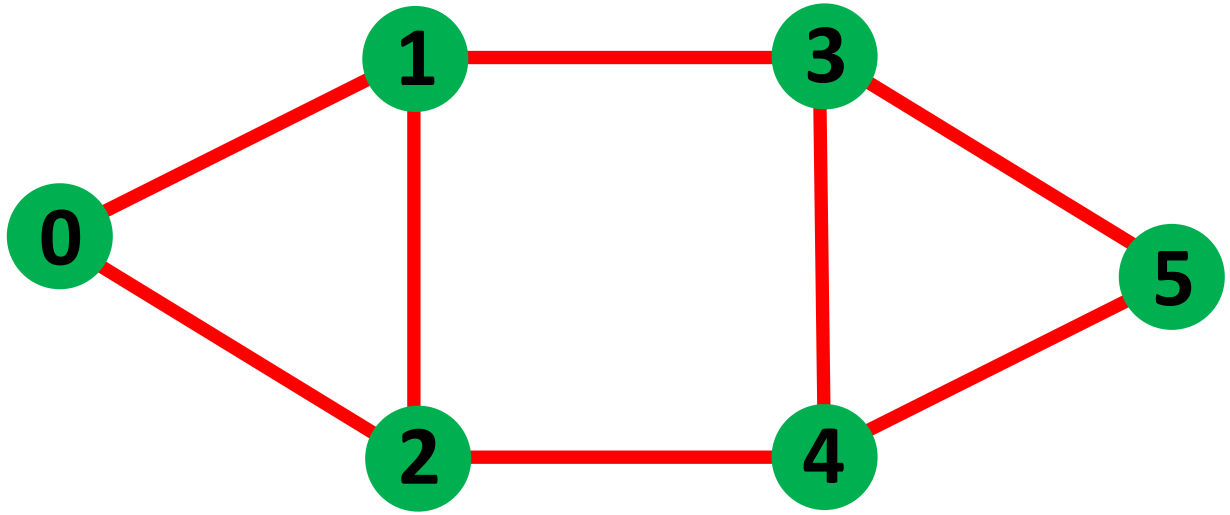
Fill out survey if you haven't already for lab 8

All of program 3 has been posted

Lab 9

Graphs

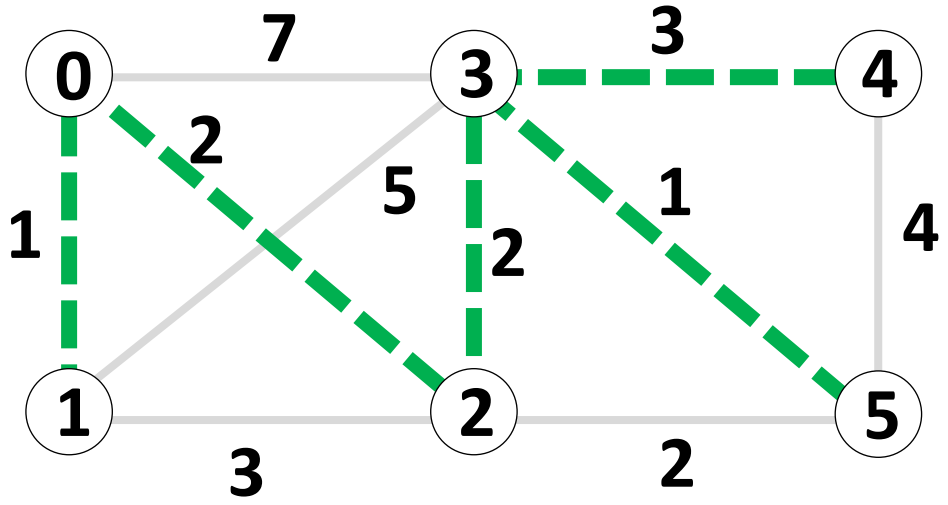
$$G = (V, E)$$



Adjacency List

0	→	{1,2}
1	→	{0,2,3}
2	→	{0,1,4}
3	→	{1,4,5}
4	→	{2,3,5}
5	→	{3,4}

Minimum Spanning Tree

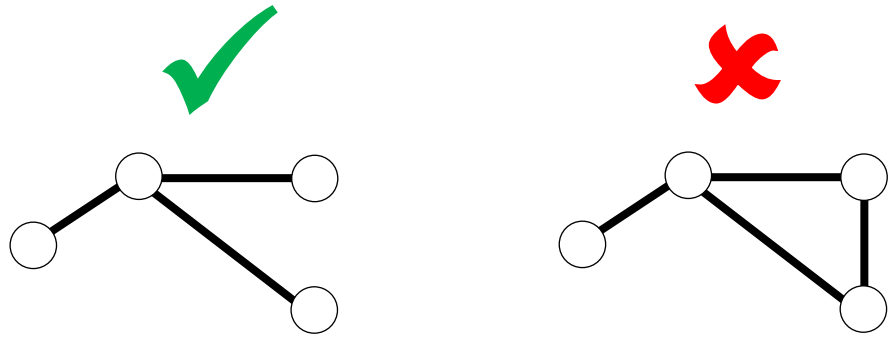


Edge-weighted graph: A graph where each edge has a weight (cost).

MST Goal: Connect all vertices to each other with a minimum weight subset of edges.

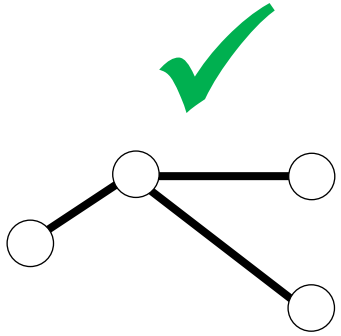
Minimum Spanning Tree

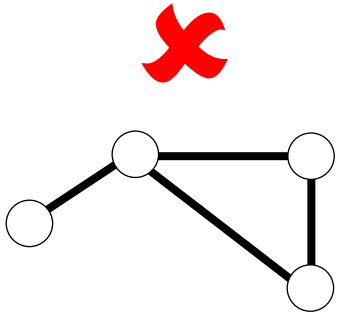
Tree – connected graph with no loops.



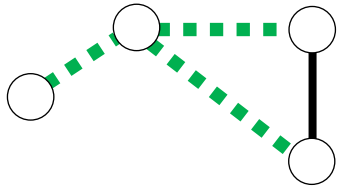
Minimum Spanning Tree

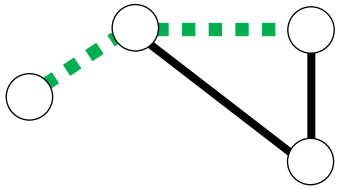
Tree – connected graph with no loops.





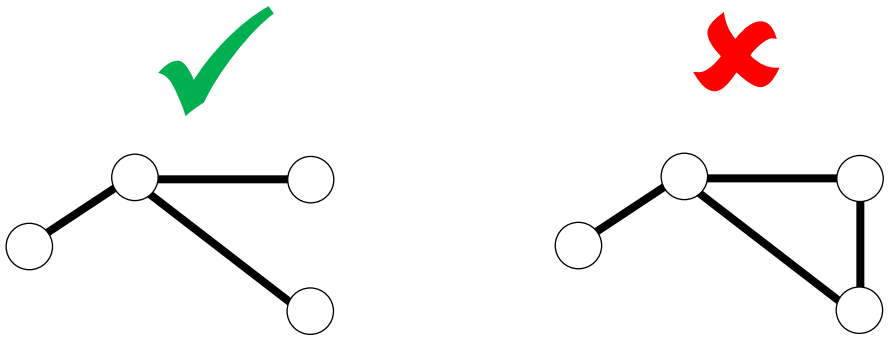
Spanning tree – tree that includes all vertices in a graph.





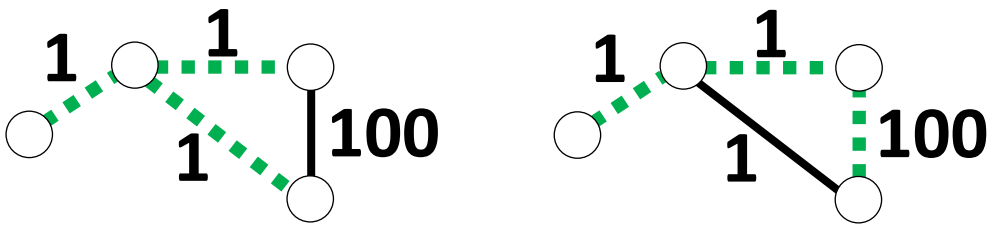
Minimum Spanning Tree

Tree – connected graph with no loops.

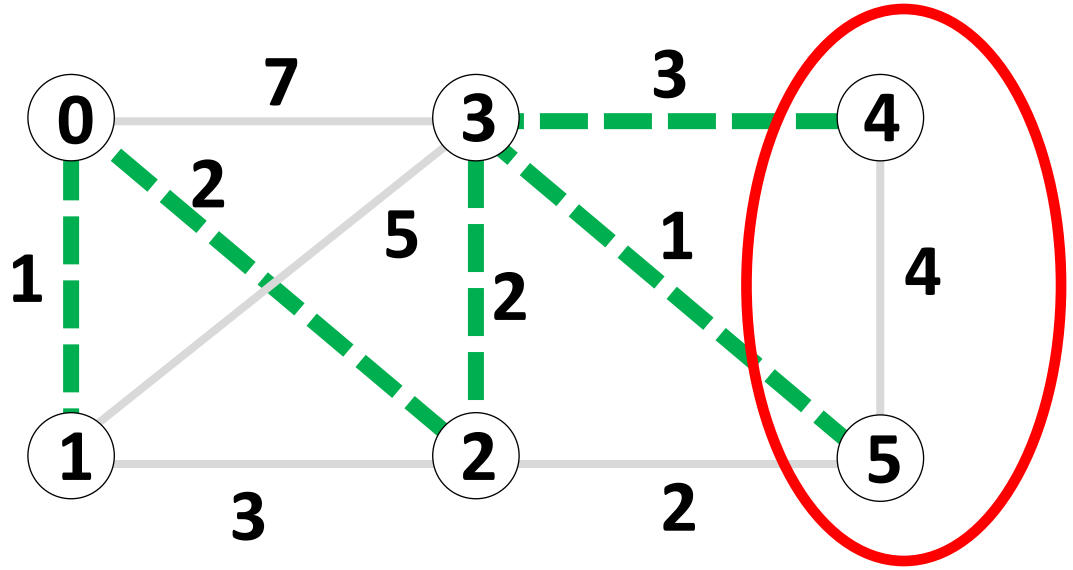


Spanning tree – tree that includes all vertices in a graph.

Minimum spanning tree – spanning tree whose sum of edge costs is the minimum possible value.



Minimum Spanning Tree



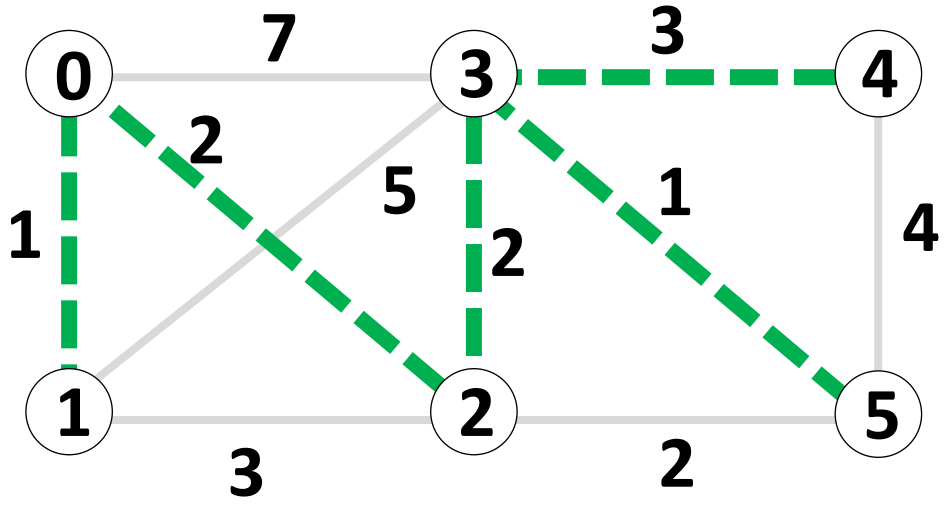
Does it ever make sense to have a cycle?

No!

Must be a tree!

MST Goal: Connect all vertices to each other with a minimum weight subset of edges.

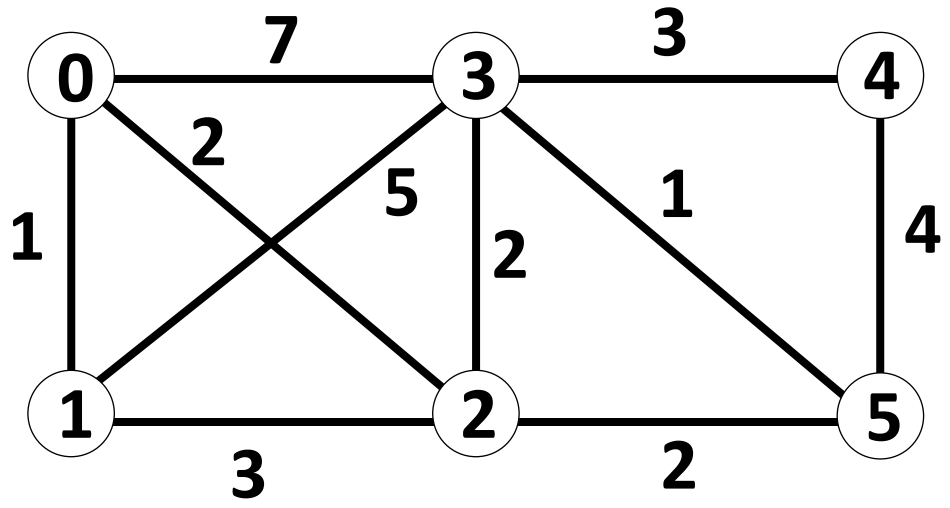
Minimum Spanning Tree



How to find MSTs?

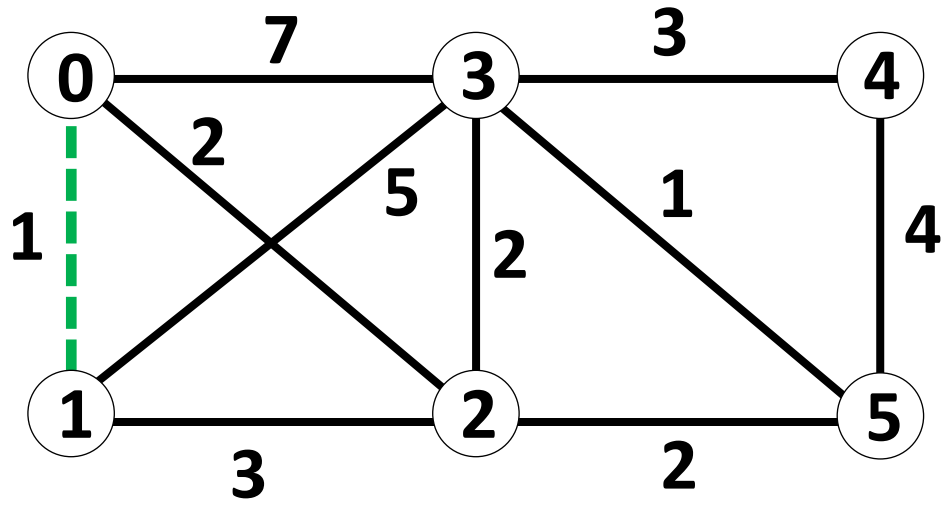
Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



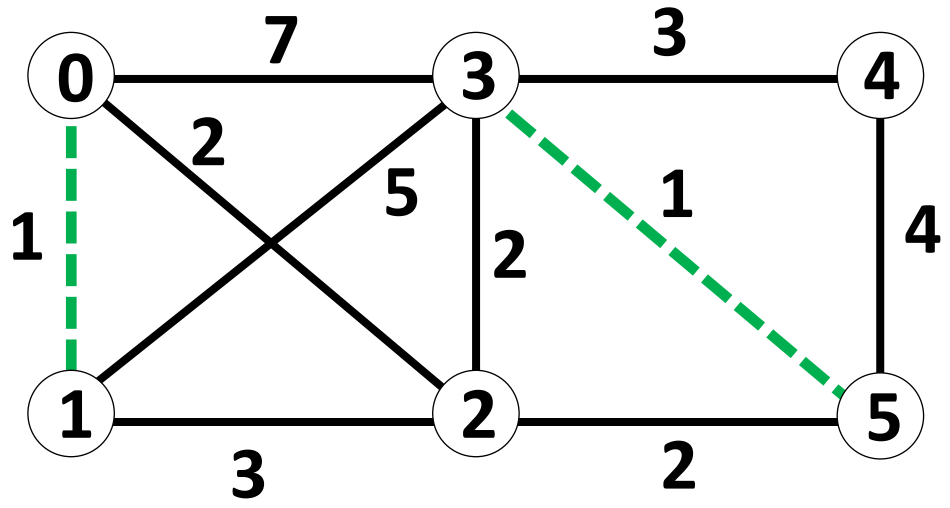
Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



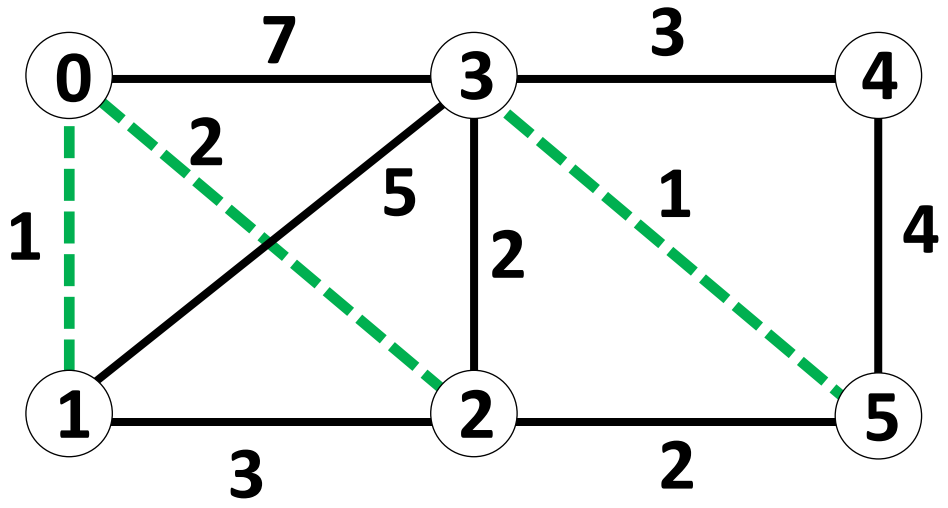
Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



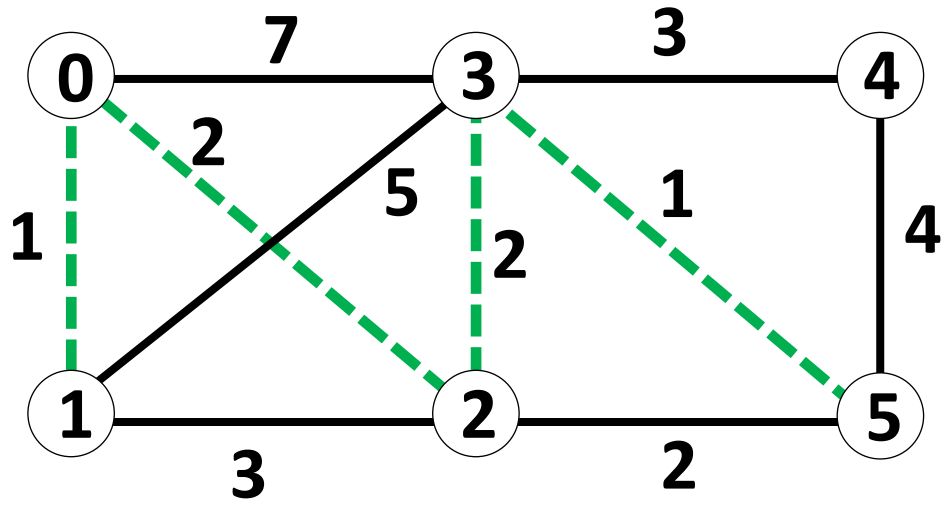
Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



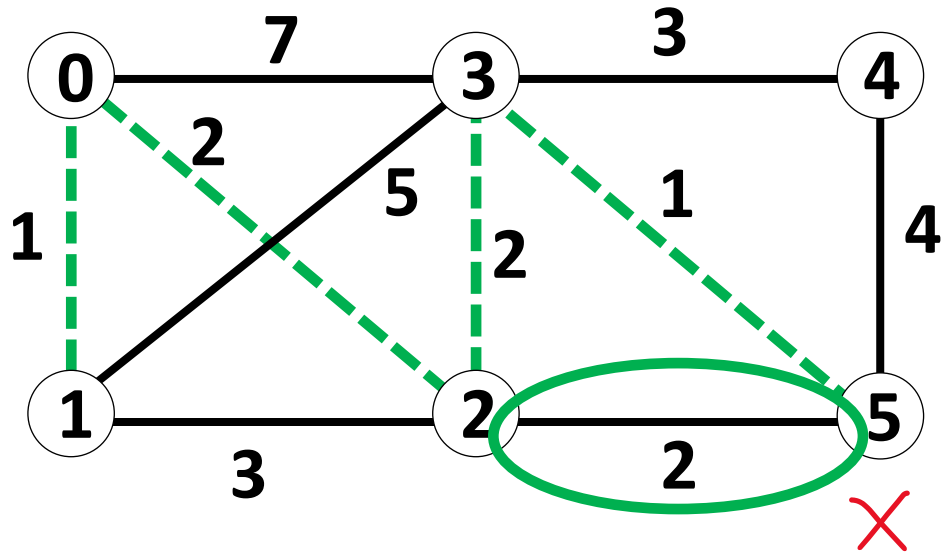
Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



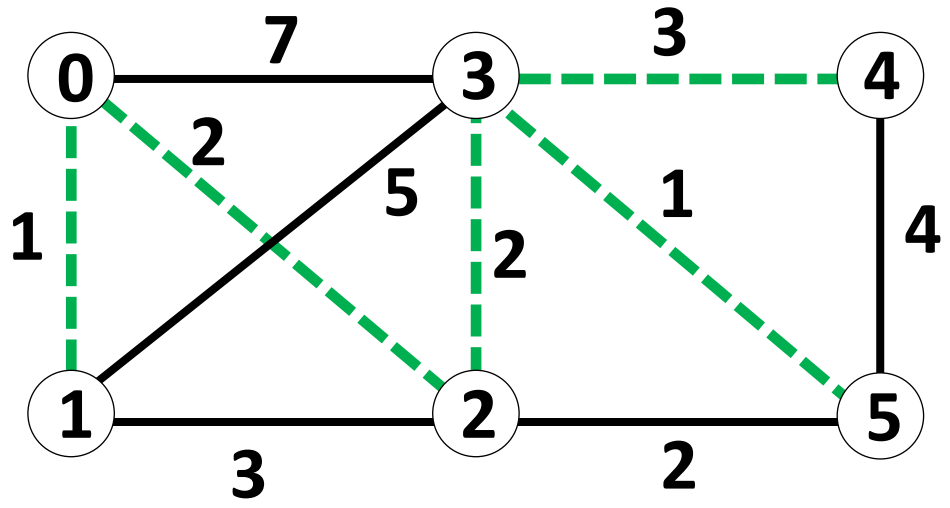
Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



Kruskal's MST Algorithm

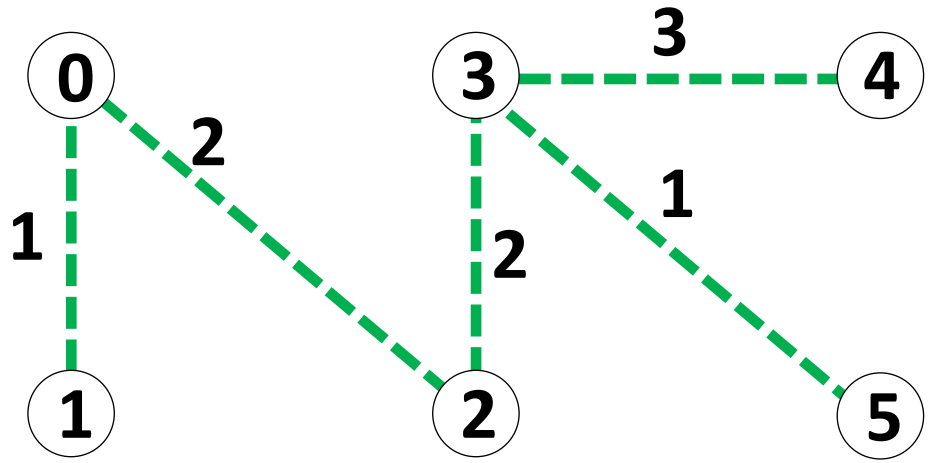
At each iteration, add the edge with smallest weight, that does not create a cycle.



Kruskal's MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.

MST = [0, 1], [0, 2], [2,3], [3,5], [3,4] **Total Cost = 9**



```
public kruskalsAlgorithm(WeightedGraph graph) {




}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();



    // Get the set of edges.









}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();



    // Get the set of edges.
    HashSet<Edge> Edges = graph.getEdges();









}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();



    // Get the set of edges, in order of increasing weight.
    HashSet<Edge> Edges = graph.getEdges();




}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();



    // Get the set of edges, in order of increasing weight.
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();









}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();



    // Get the set of edges, in order of increasing weight.
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }









}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();



    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    // Run Kruskal's algorithm.




}
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {









}   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();









}   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        mst.add(edge);









}   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        ~~mst.add(edge);~~
```

Need to check if adding edge adds a loop!

```
}   }
```

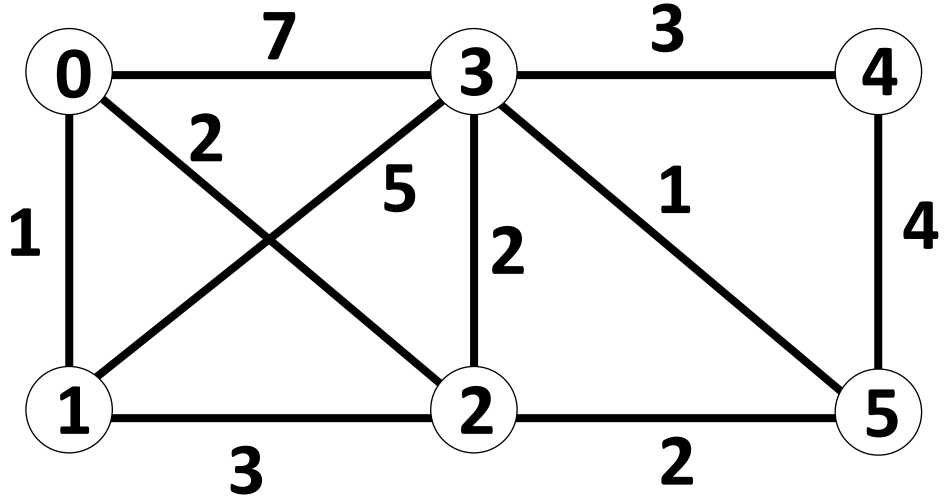
```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        mst.add(edge);
```

Need to check if adding edge adds a loop!

How?

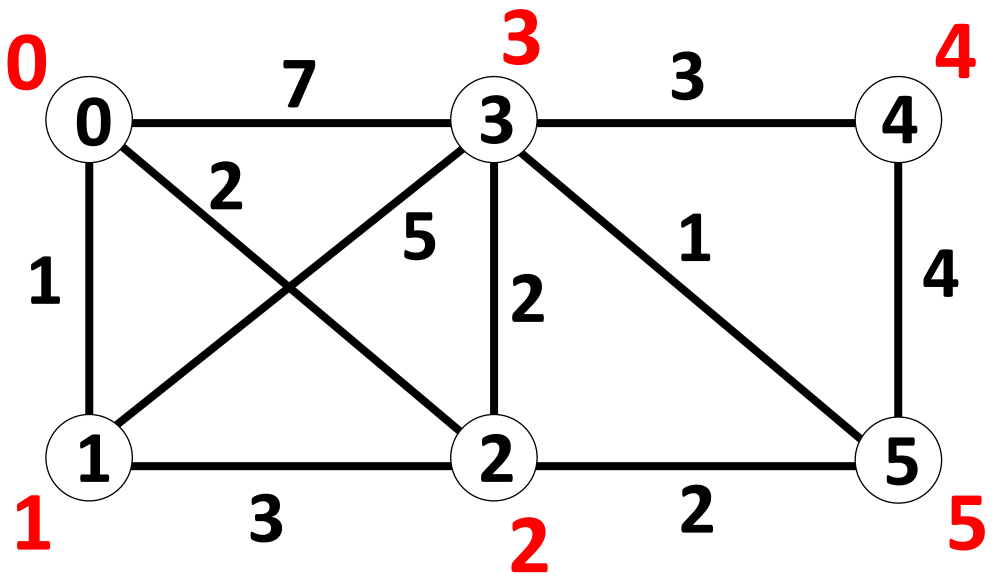
```
}   }
```

Cycle Finding



How can we determine if adding an edge puts in a cycle?

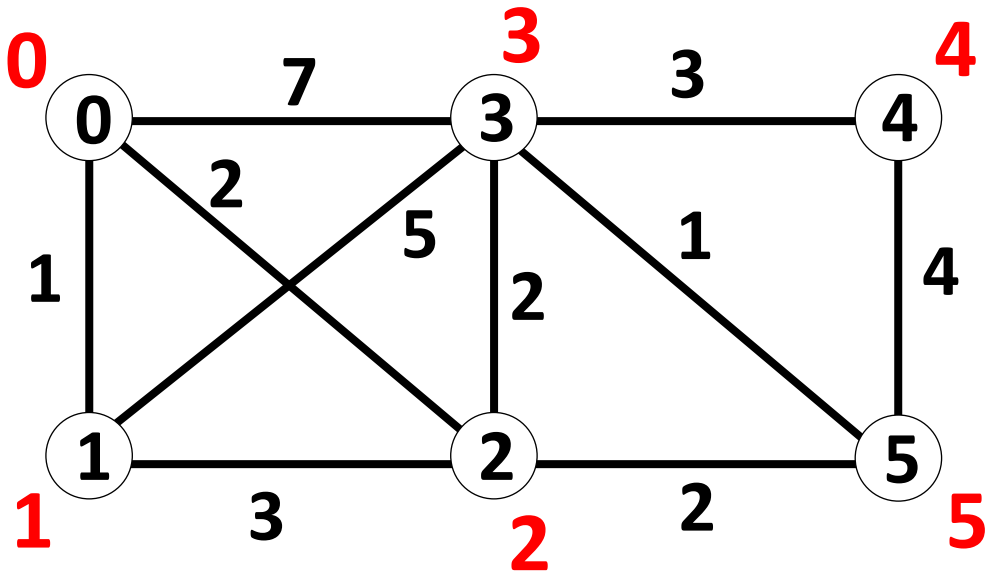
Cycle Finding



Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

Cycle Finding

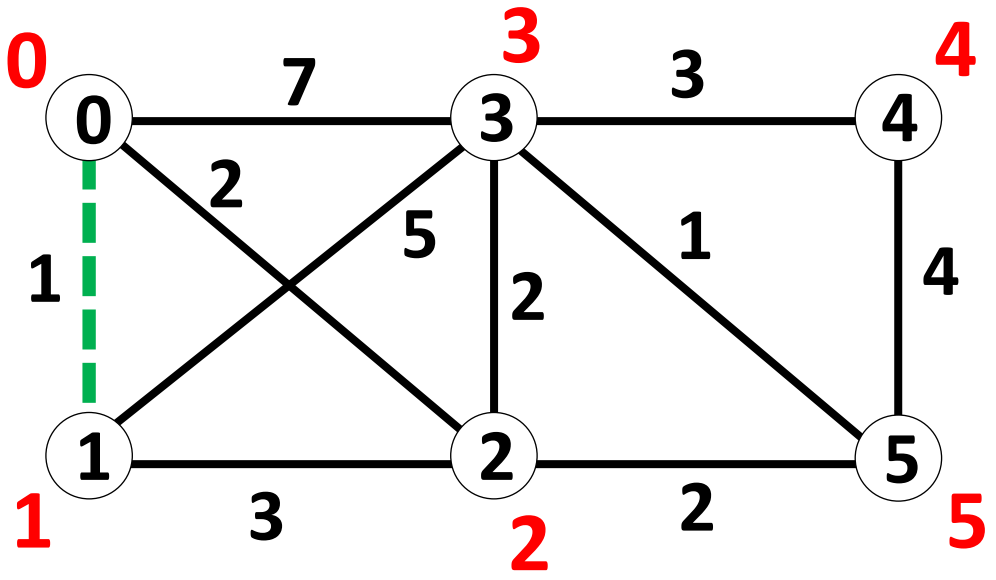


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

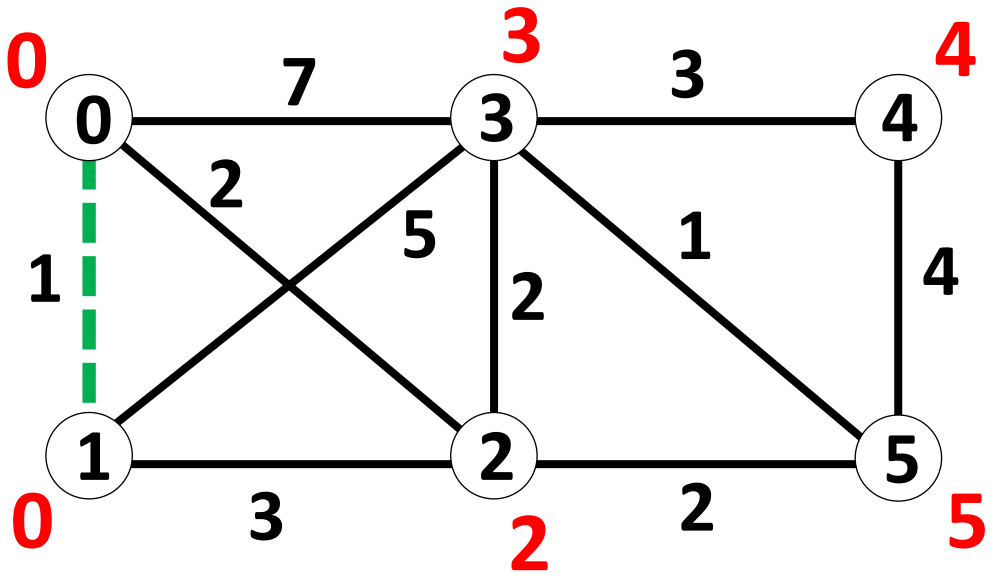


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

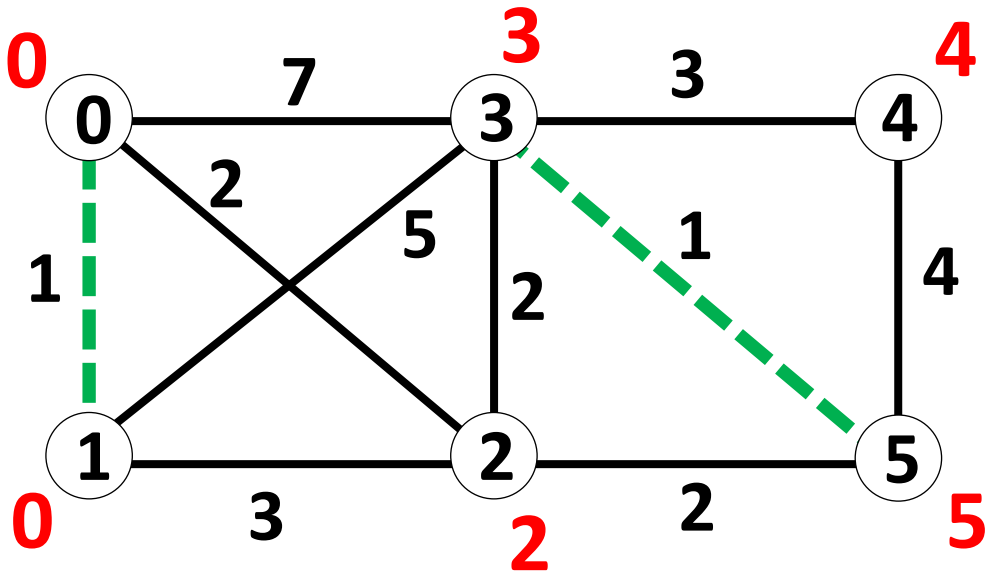


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

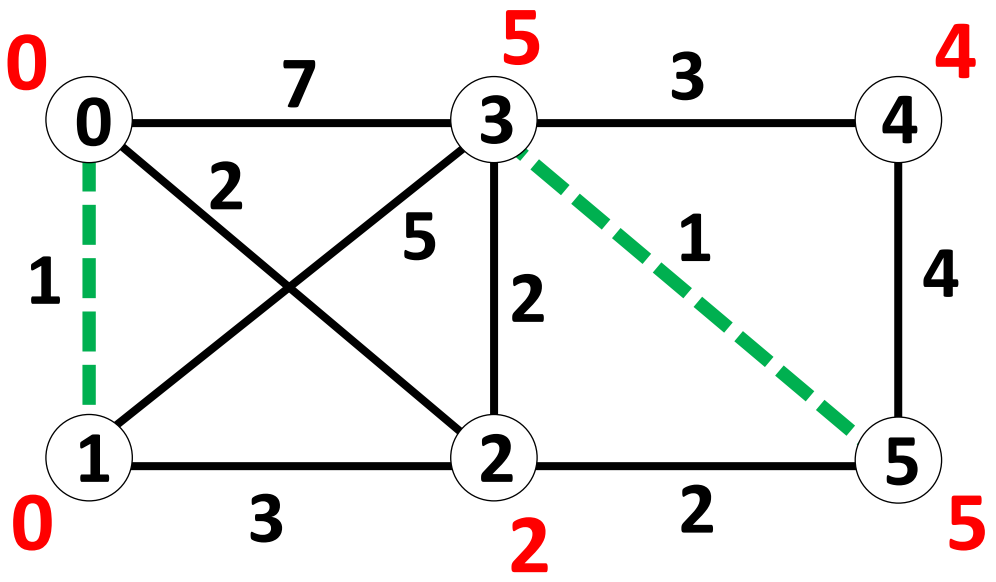


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

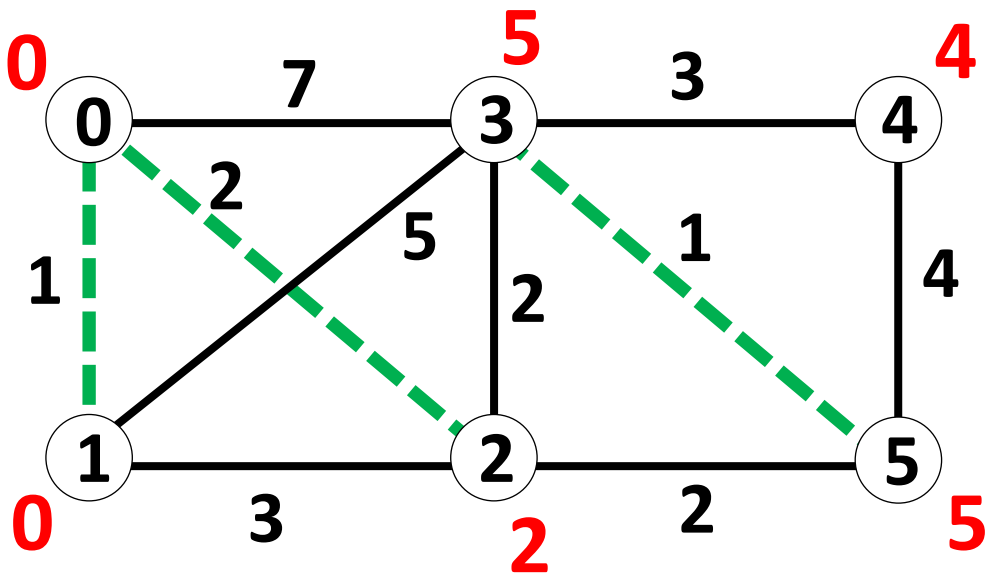


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

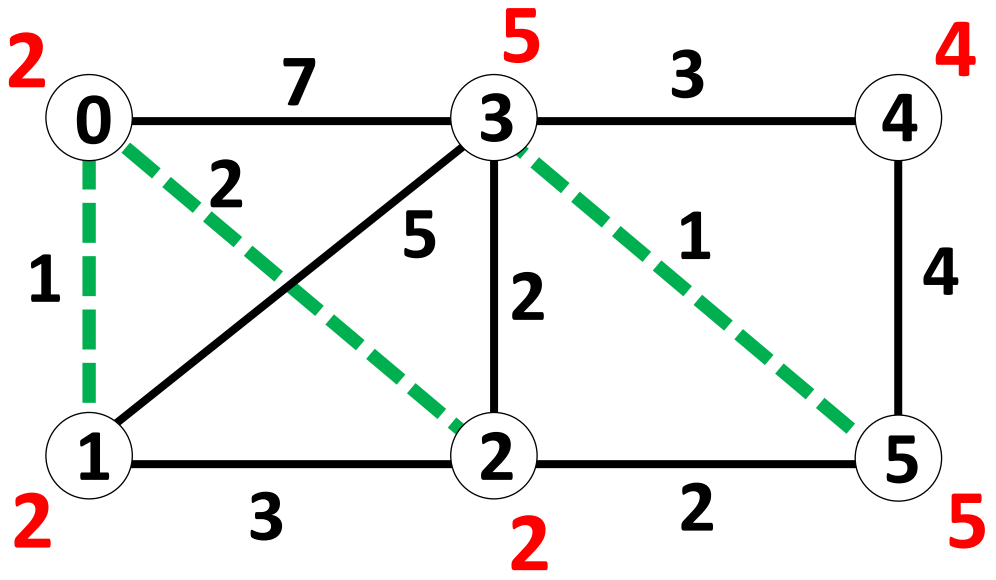


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

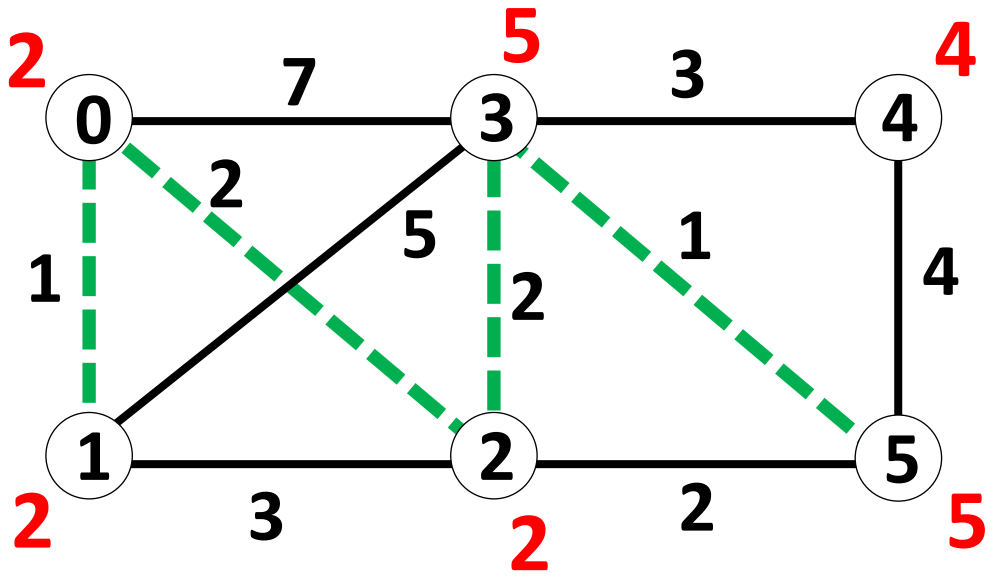


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

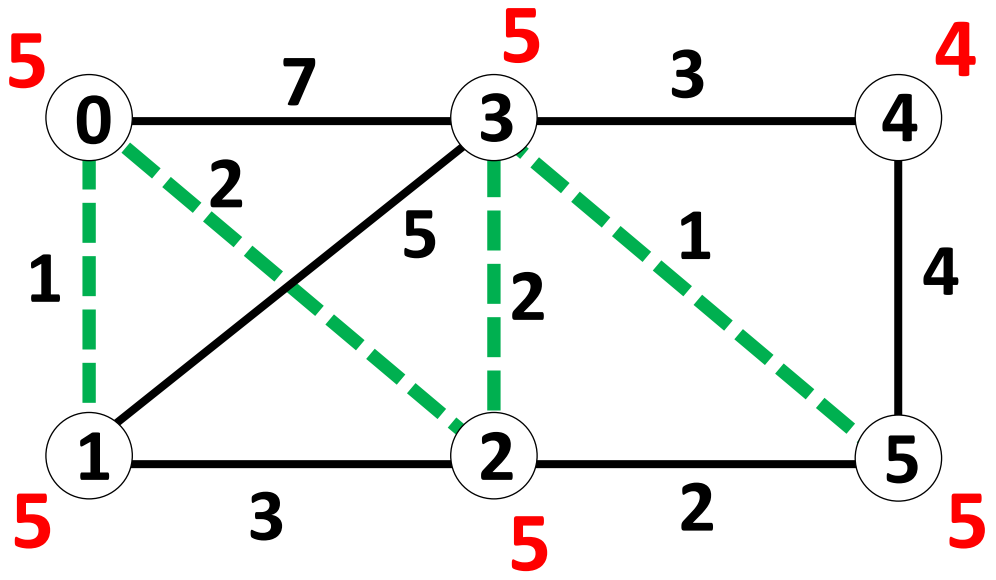


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

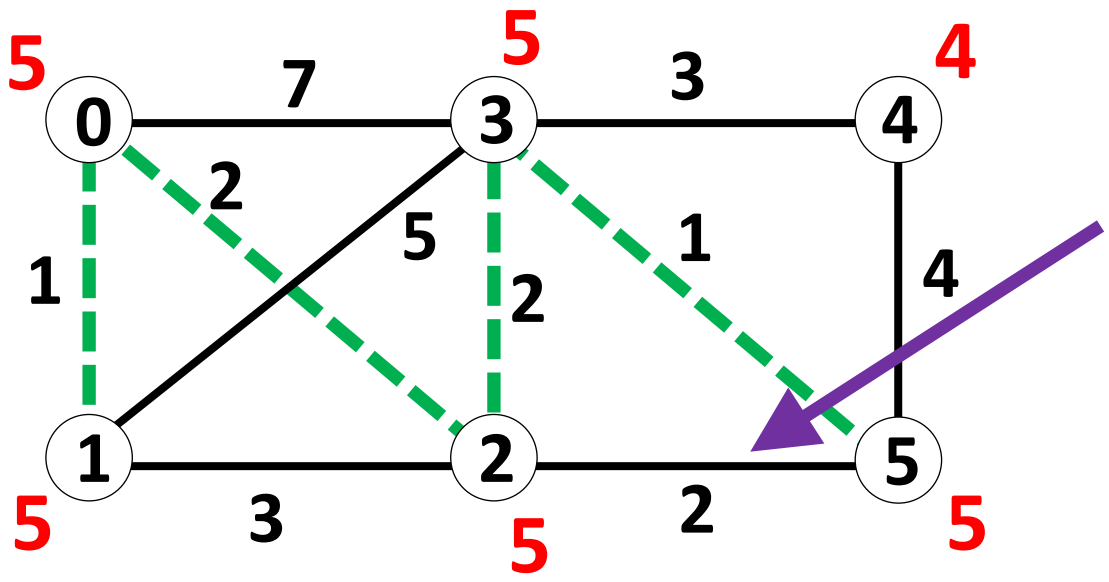


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding



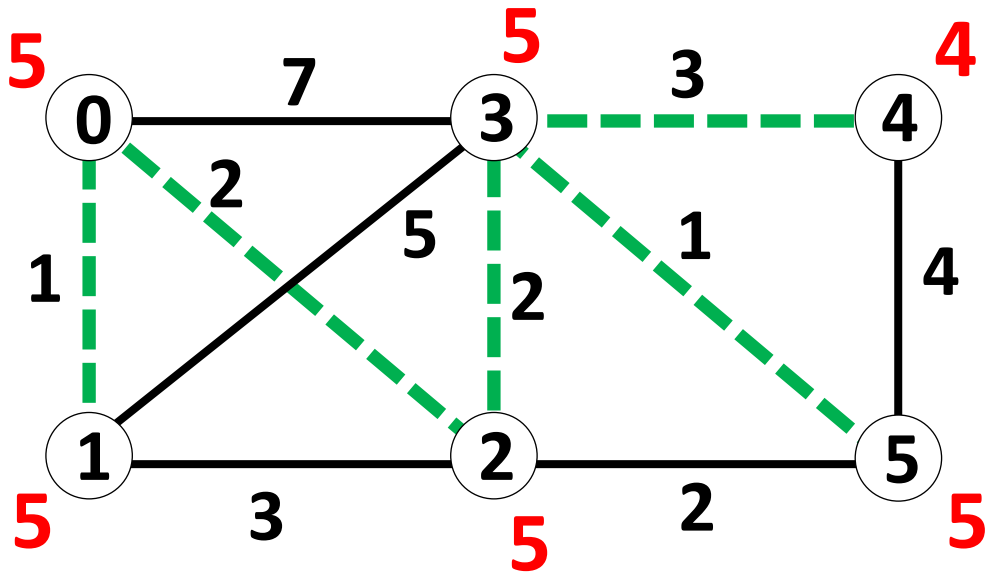
Why is adding this edge a bad idea?

Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

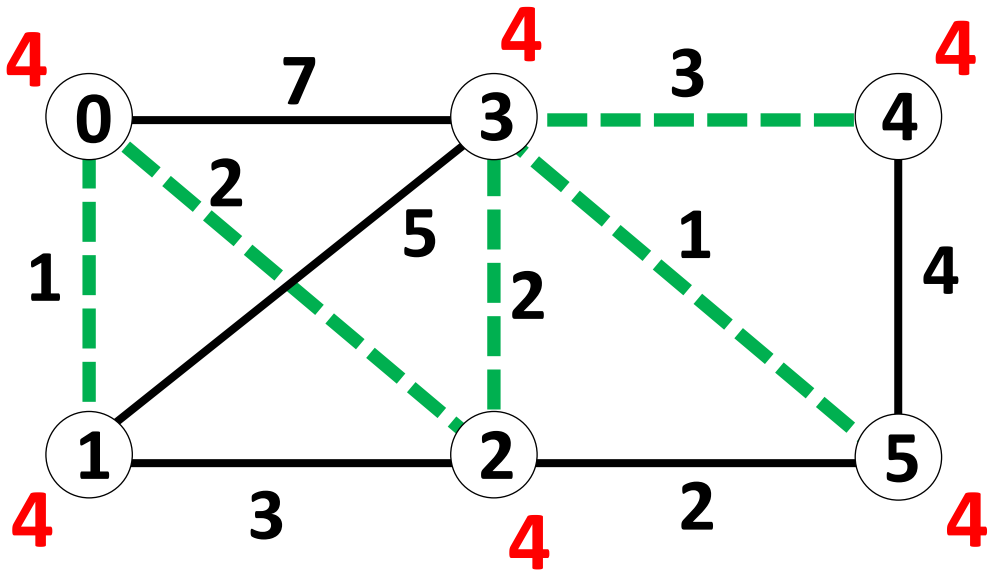


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding



Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        ~~mst.add(edge);~~
```

Need to check if adding edge adds a loop!

```
}   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();

        // Need to check if adding edge adds a loop.









}   }
```

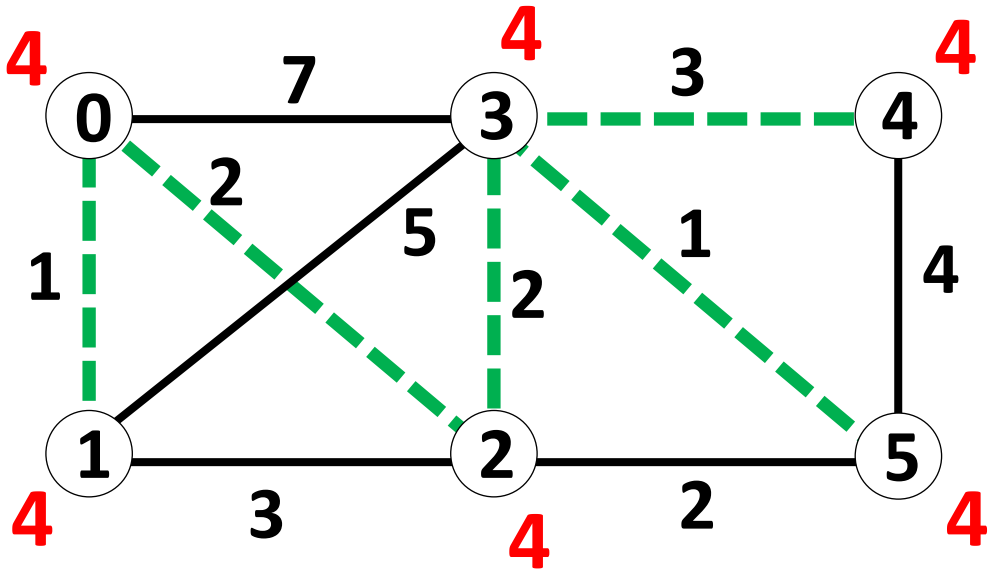
```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];



    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();

        // Need to check if adding edge adds a loop.









}   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();

        // Need to check if adding edge adds a loop.









}   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {









}   }   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                    != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);









}   }   }
```

Cycle Finding



Rules: Only add edge if vertices have different connected component markers.

Connected component (in the tree) marker.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                    != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);









}   }   }
```

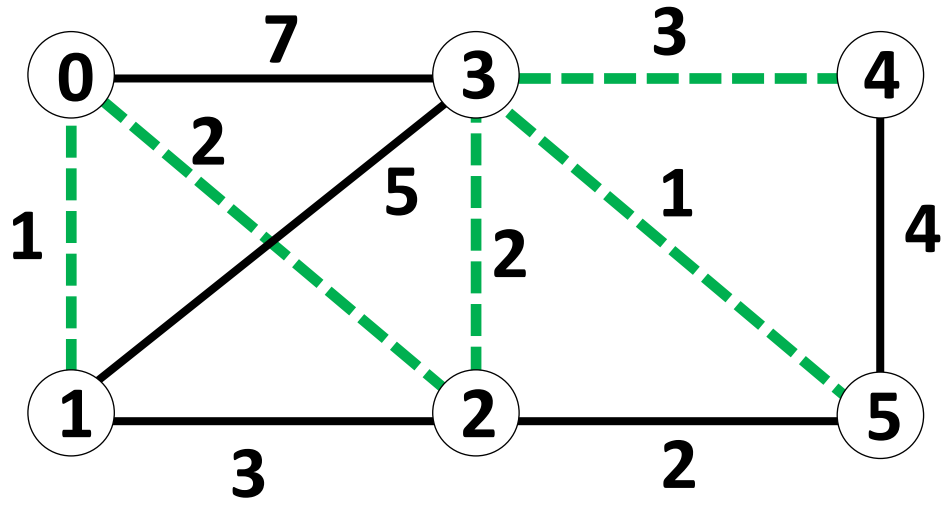
```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];



}   }   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {



}   }   }   }
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
}   }   }   }   }
```

Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.



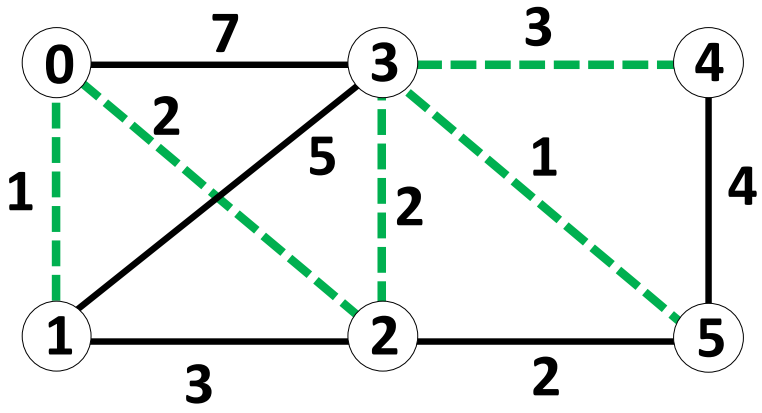
Is the solution valid (i.e., is the output a spanning tree)?

Is the solution optimal (i.e., is the output a minimum spanning tree)?

Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

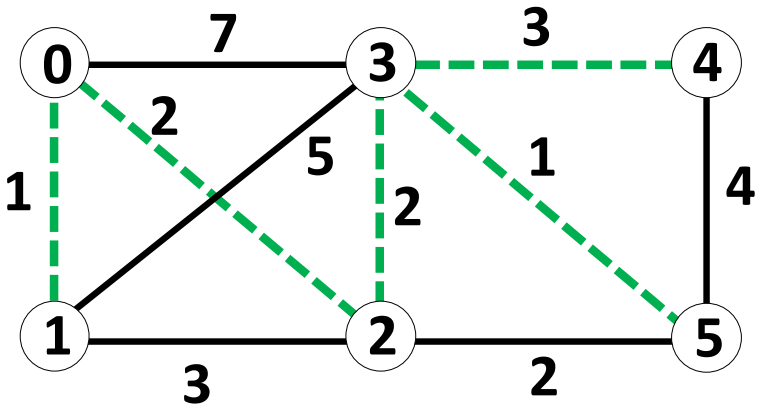


Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

T is a tree because…?

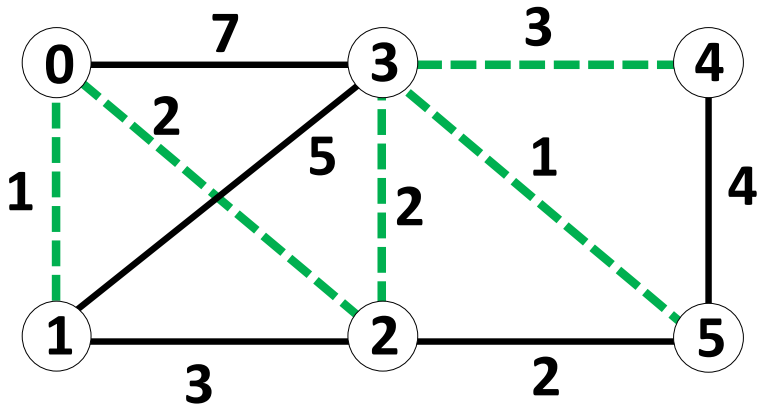


Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.



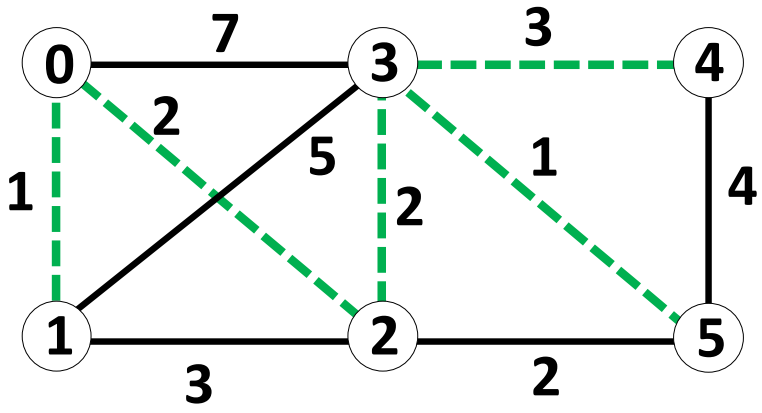
Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.

T spans G because…?



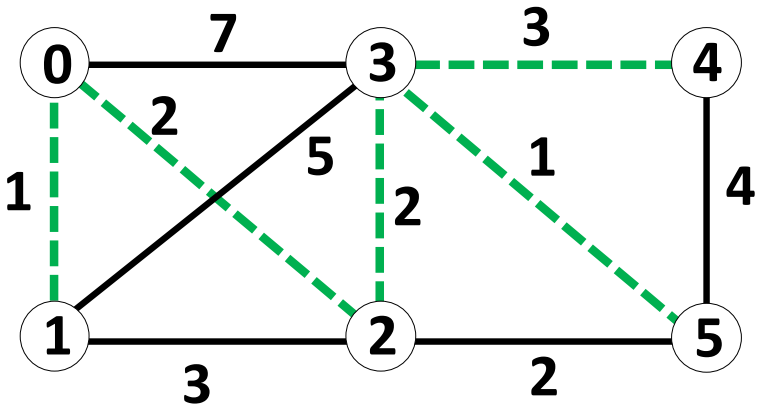
Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.

T spans G because if it did not, we could have added more edges to connected unreached nodes without creating cycles.



Kruskal's MST Algorithm

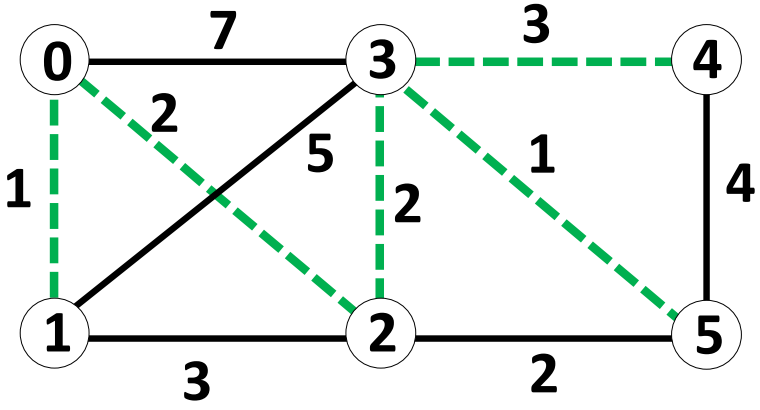
At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.

T spans G because if it did not, we could have added more edges to connected unreached nodes without creating cycles.

$\therefore$ T is a spanning tree of G



Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

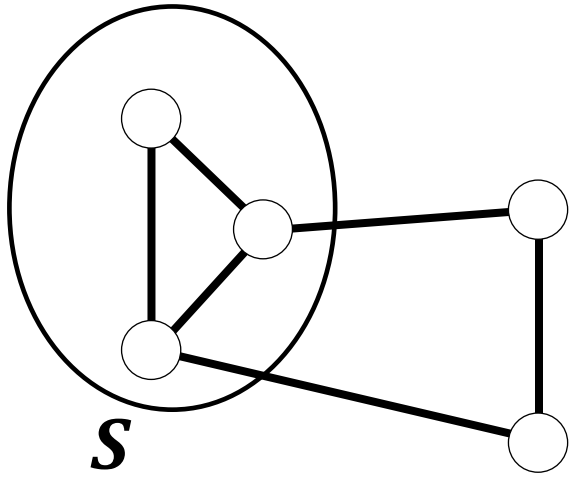
Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

T is an MST because???

MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

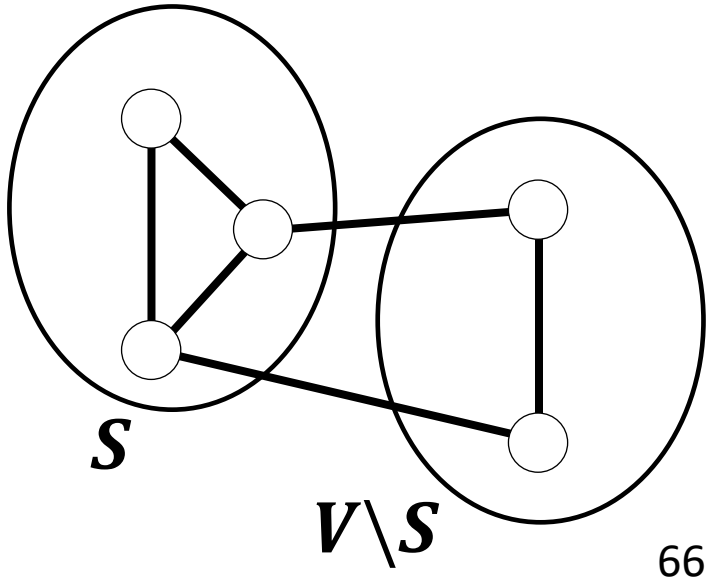
Proof:



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

Proof:



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

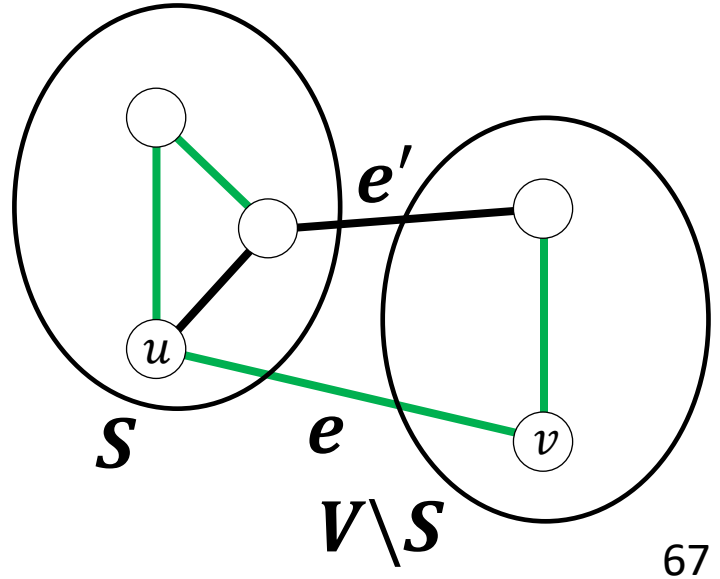
Let e be the cheapest edge between S and $V \backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V \backslash S$.

Remove e' to form $T' = T \cup \{e\} \backslash \{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)
T' spans V (same number of edges as ST T)
weight(T') = weight(T) + weight(e) – weight(e').
⇒ weight(T') < weight(T), since weight(e) < weight(e').
⇒ T' is a cheaper ST.

So, e is part of every MST.

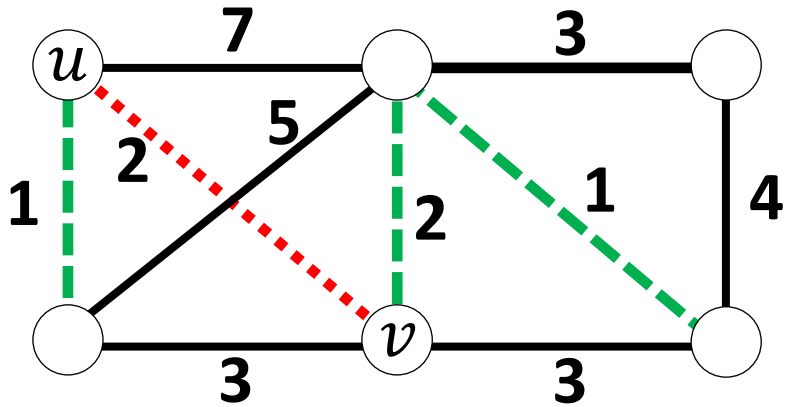


Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

Consider the iteration that edge $e = (u, v)$ is added by Kruskal's algorithm. Let S be the set of v and nodes connected to v. Clearly $v \in S$ and $u \in V \backslash S$ (otherwise adding e would have created a cycle). We are picking the cheapest such edge (otherwise the cheaper edge would have been selected since it would not have created a cycle either). By the cut property lemma, this edge must be part of the MST.



Lemma: The cheapest edge between $S \subseteq V$ and $V \backslash S$ is part of every MST.

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

O(1)

O(V)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

O(1)

O(V)

O(V)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

O(1)*

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

O(1)*

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

O(1)*

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

O(1)*

O(1)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

O(1)*

O(1)

O(1)

O(1)

O(V)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E)

O(E)

O(1)*

O(E * V)

O(1)

O(1)

O(1)

O(V)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

E = # of edges

O(1)

O(V)

O(V)

O(1)

O(E log E))

O(E log E)

O(E)

O(E * V)

O(1)*

O(1)

O(1)

O(1)

O(V)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

E = # of edges

O(1)

O(V)

O(V)

O(V)

O(1)

O(E log E))

O(E log E)

O(E)

O(E * V)

O(1)*

O(1)

O(1)

O(1)

O(V)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();
    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }
    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

E = # of edges

O(1)

O(V)

O(V)

O(V)

O(1)

O(E log E))

O(E log E)

O(E)

O(E * V)

O(1)*

O(1)

O(1)

O(1)

O(V)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
```

V = # of vertices

E = # of edges

O(1)

O(V)

O(V)

O(V)

O(1)

O(E log E))

O(E log E)

O(E)

O(E * V)

O(1)*

O(1)

O(1)

O(1)

O(V)

O(1)

O(1)

```
public kruskalsAlgorithm(WeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();
    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for ( : graph.getEdges()) {
        (edge);
    }
    (!edge pty()) {
    Edge edge = ue.poll();
    if (connectedComponentMarker[edge.getVertices()[0]]
            != connectedComponentMarker[edge.getVertices()[1]]) {
        mst.add(edge);
        int newMarker = connectedComponentMarker[edge.getVertices()[0]];
        int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
        for (int i = 0; i < connectedComponentMarker.length; i++) {
            if (connectedComponentMarker[i] == oldMarker) {
                connectedComponentMarker[i] = newMarker;
```

V = # of vertices

E = # of edges

O(1)

O(V)

O(V)

O(V)

O(1)

O(E log E))

O(E log E)

O(E)

O(1)*

O(E * V)

O(1)

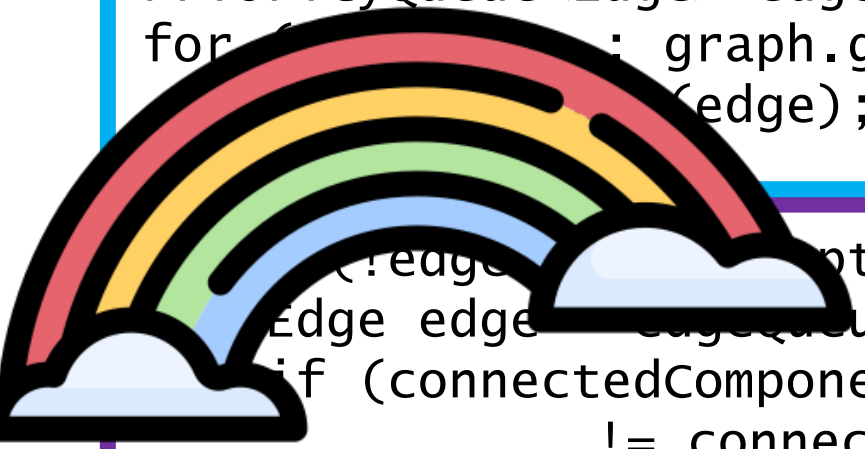
O(1)

O(1)

O(V)

O(1)

O(1)



$O(E * V) + O(E \log E)) + O(V) + O(V) + O(1)$

$$O(E * V) + O(E \log E)) + O(V) + O(V) + O(1)$$

Dominant Factor

O(E * V)

Dominant Factor

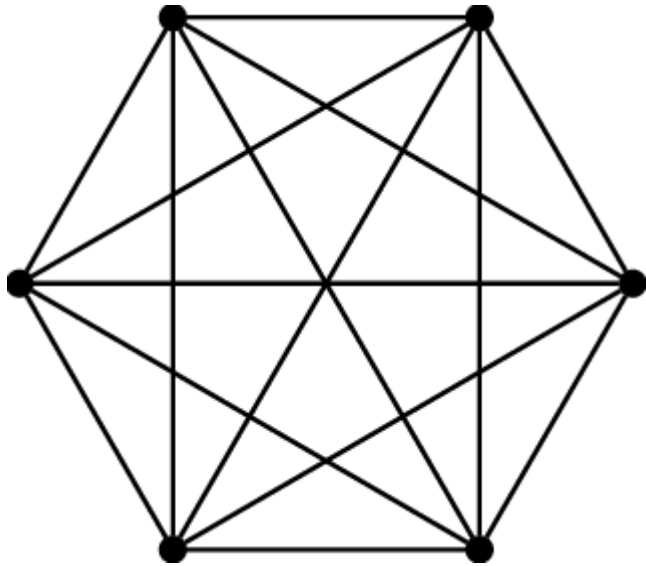
At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

O(E * V)

Dominant Factor

At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph



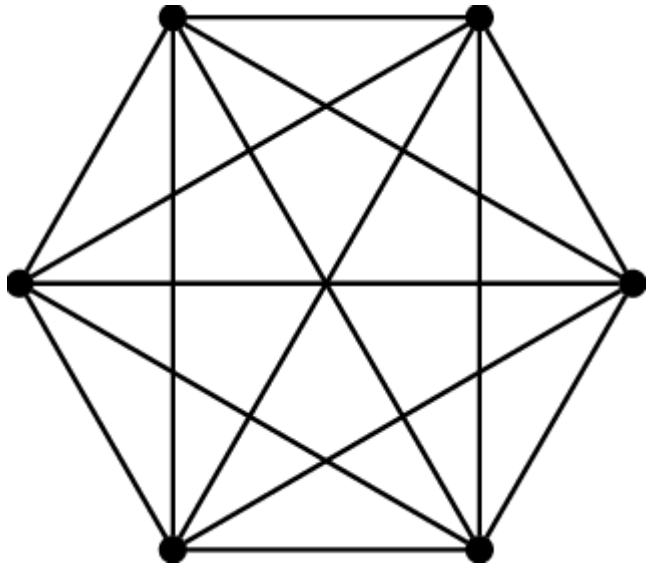
O(E * V)

Dominant Factor

At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?



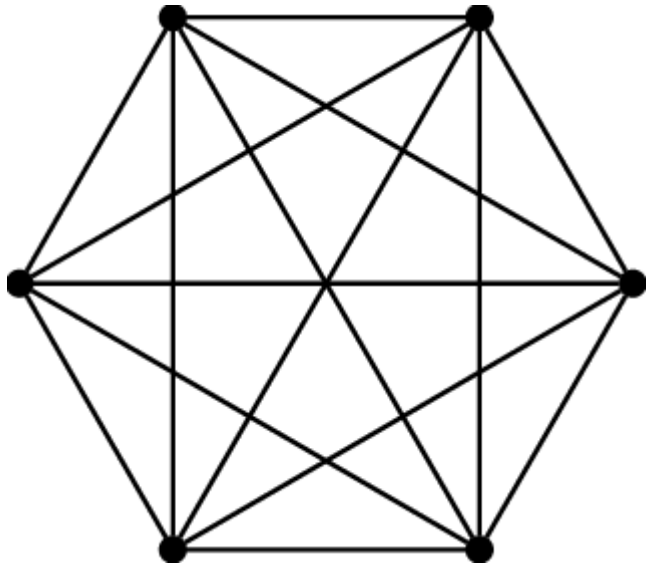
O(E * V)

Dominant Factor

At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?
How many edges leave each vertex? n-1



O(E * V)

Dominant Factor

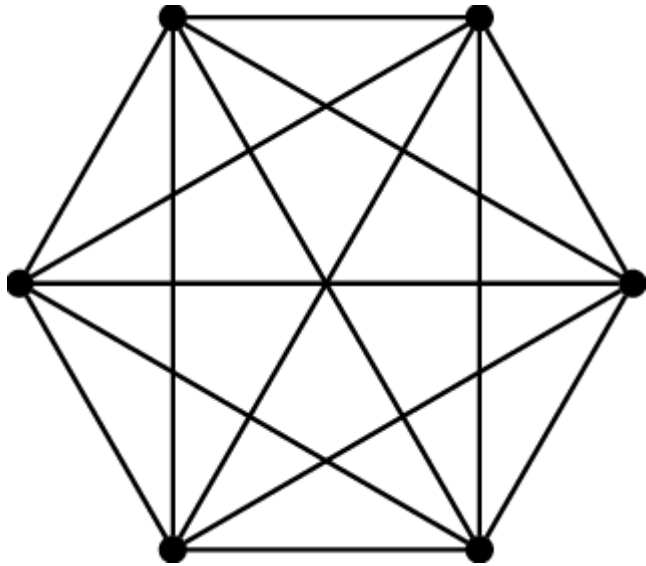
At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?

How many edges leave each vertex? n-1

How much does that all add up to? n * (n-1)



O(E * V)

Dominant Factor

At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

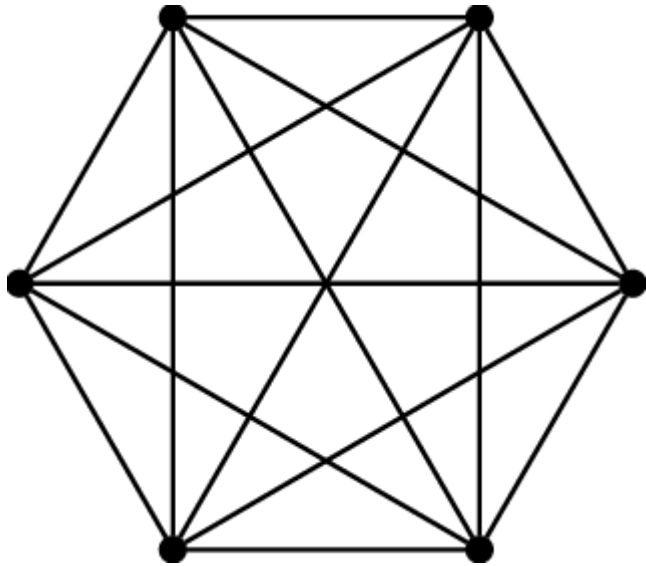
A complete graph

How many edges does a complete graph with **n** vertices have?

How many edges leave each vertex? n-1

How much does that all add up to? n * (n-1)

Did we double count any edges? Yes



O(E * V)

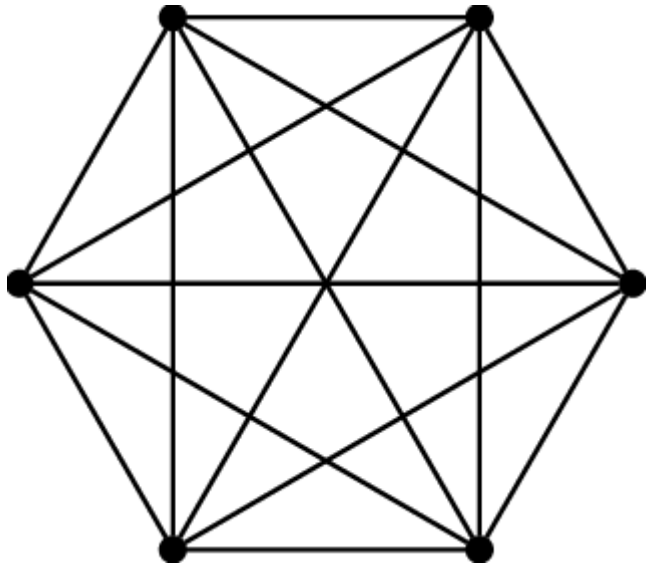
Dominant Factor

At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?

- How many edges leave each vertex? n-1
- How much does that all add up to? n * (n-1)
- Did we double count any edges? Yes
- So how many edges are there?



O(E * V)

Dominant Factor

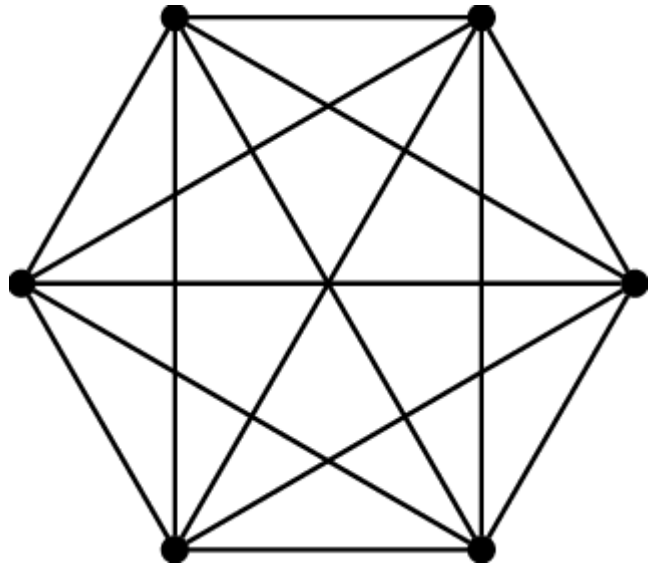
At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?

How many edges leave each vertex? n-1
How much does that all add up to? n * (n-1)
Did we double count any edges? Yes
So how many edges are there?

$$\frac{n(n-1)}{2}$$



O(E * V)

Dominant Factor

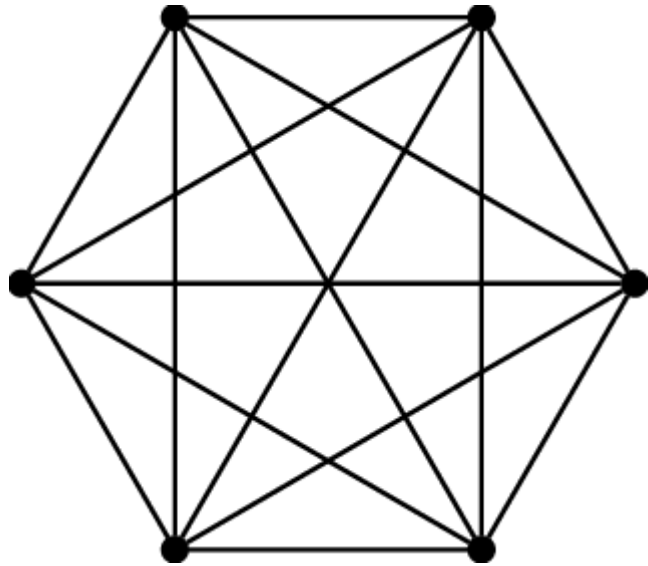
At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?

How many edges leave each vertex? n-1
How much does that all add up to? n * (n-1)
Did we double count any edges? Yes
So how many edges are there?

$$\frac{n(n-1)}{2} \rightarrow \frac{1}{2}(n^2 - n) \in O(n^2)$$

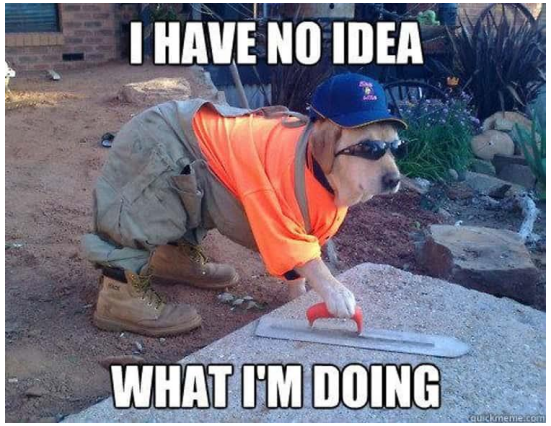


$O(E * V)$ $O(n^2 * n)$

Running time of **our** Kruskal's algorithm

Dominant Factor

$O(n^3)$



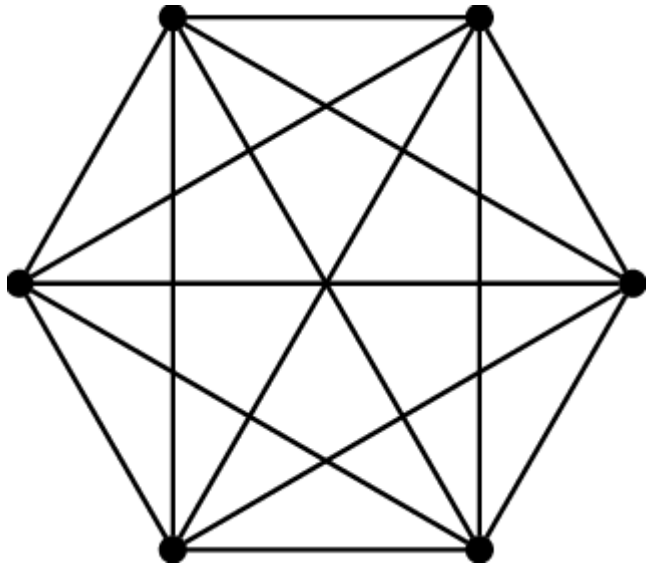
At most, how many edges are in an undirected graph with **n** vertices?
What graph has the most number of edges?

A complete graph

How many edges does a complete graph with **n** vertices have?

How many edges leave each vertex? n-1
How much does that all add up to? n * (n-1)
Did we double count any edges? Yes
So how many edges are there?

$$\frac{n(n-1)}{2} \rightarrow \frac{1}{2}(n^2 - n) \in O(n^2)$$

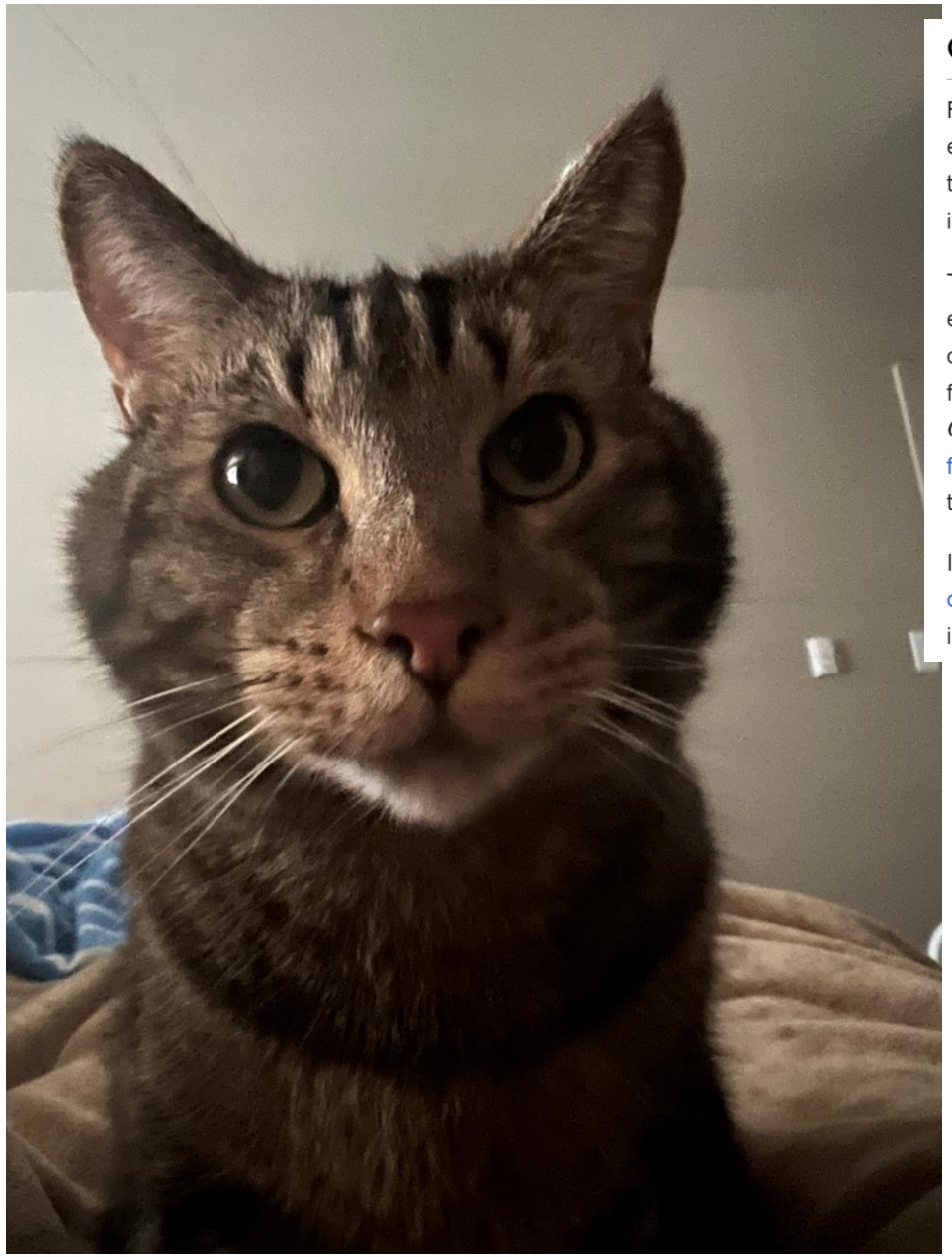


There are many ways to write Kruskal's algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅                                                        O(1)
2  for each vertex v ∈ G.V                                      O(V)
3      MAKE-SET(v)                                              O(V)
4  sort the edges of G.E into nondecreasing order by weight w   O(E logE)
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)                             O(V logV)
7          A = A ∪ {(u, v)}                                     O(V logV)
8          UNION(u, v)                                          O(V logV)
9  return A
```

E > V in a complete graph (worst case), so we can get Kruskal's algorithm to run in **O(E log E)** time

Running time of finding an MST: **O(E log E)**



Complexity [edit]

For a graph with E edges and V vertices, Kruskal's algorithm can be shown to run in time $O(E \log E)$ time, with simple data structures. Here, O expresses the time in big O notation, and $\log$ is a logarithm to any base (since inside O-notation logarithms to all bases are equivalent, because they are the same up to a constant factor). This time bound is often written instead as $O(E \log V)$, which is equivalent for graphs with no isolated vertices, because for these graphs $V/2 \leq E < V^2$ and the logarithms of V and E are again within a constant factor of each other.

To achieve this bound, first sort the edges by weight using a comparison sort in $O(E \log E)$ time. Once sorted, it is possible to loop through the edges in sorted order in constant time per edge. Next, use a disjoint-set data structure, with a set of vertices for each component, to keep track of which vertices are in which components. Creating this structure, with a separate set for each vertex, takes V operations and $O(V)$ time. The final iteration through all edges performs two find operations and possibly one union operation per edge. These operations take amortized time $O(\alpha(V))$ time per operation, giving worst-case total time $O(E\ \alpha(V))$ for this loop, where α is the extremely slowly growing inverse Ackermann function. This part of the time bound is much smaller than the time for the sorting step, so the total time for the algorithm can be simplified to the time for the sorting step.

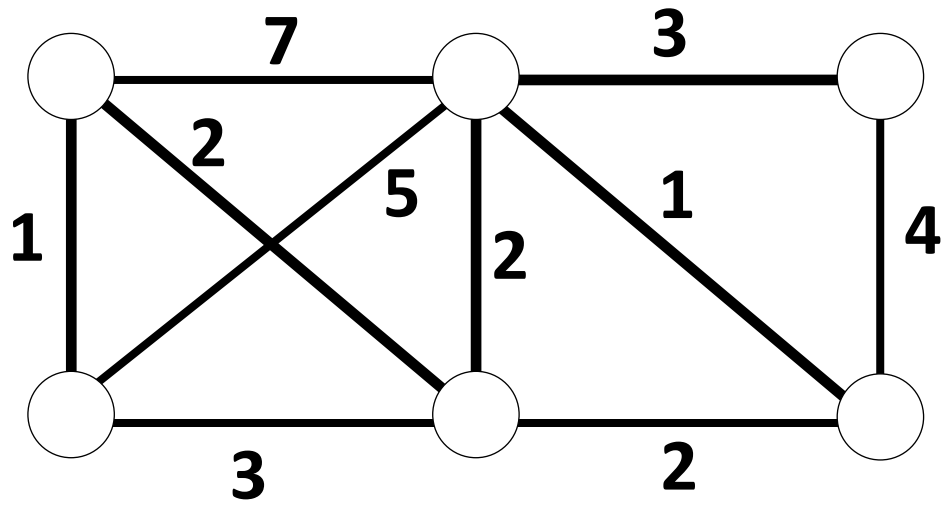
In cases where the edges are already sorted, or where they have small enough integer weight to allow integer sorting algorithms such as counting sort or radix sort to sort them in linear time, the disjoint set operations are the slowest remaining part of the algorithm and the total time is $O(E\ \alpha(V))$.

Ok, take a deep breath, the analysis is over

There is another prominent algorithm that can also compute an MST

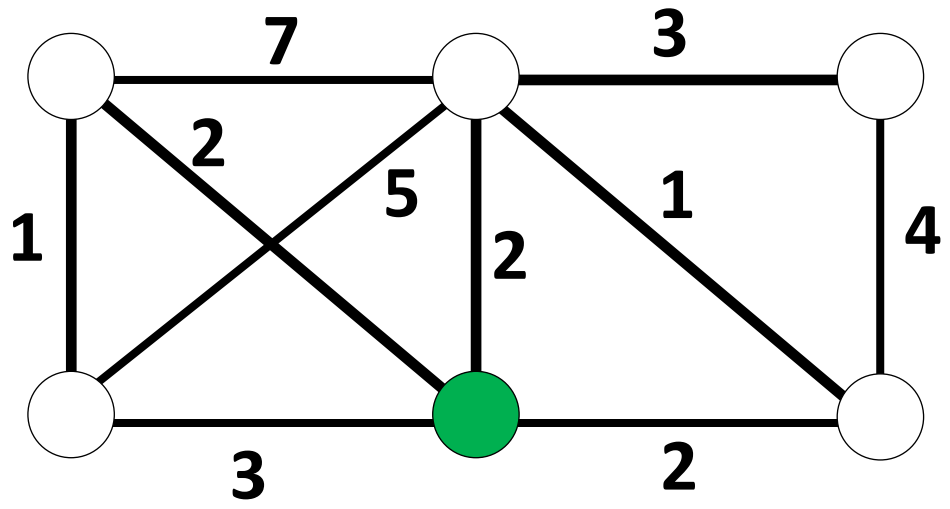
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



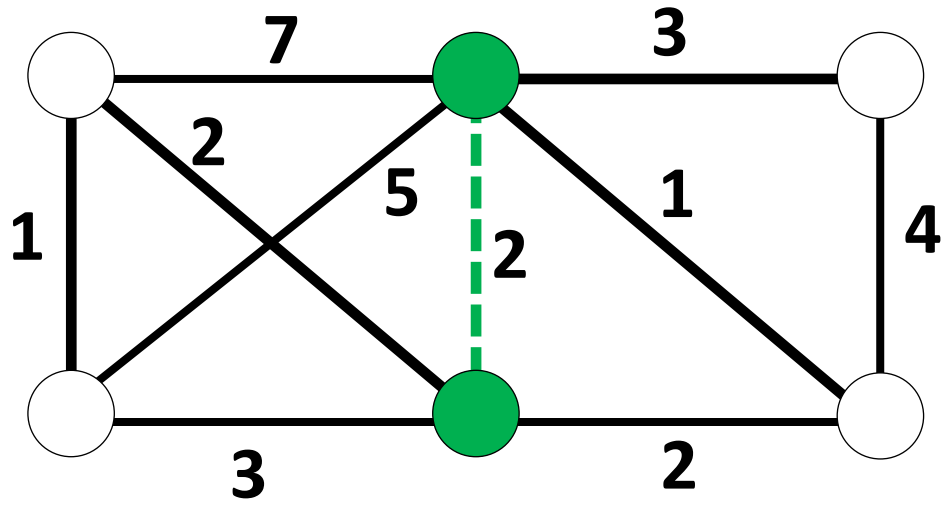
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



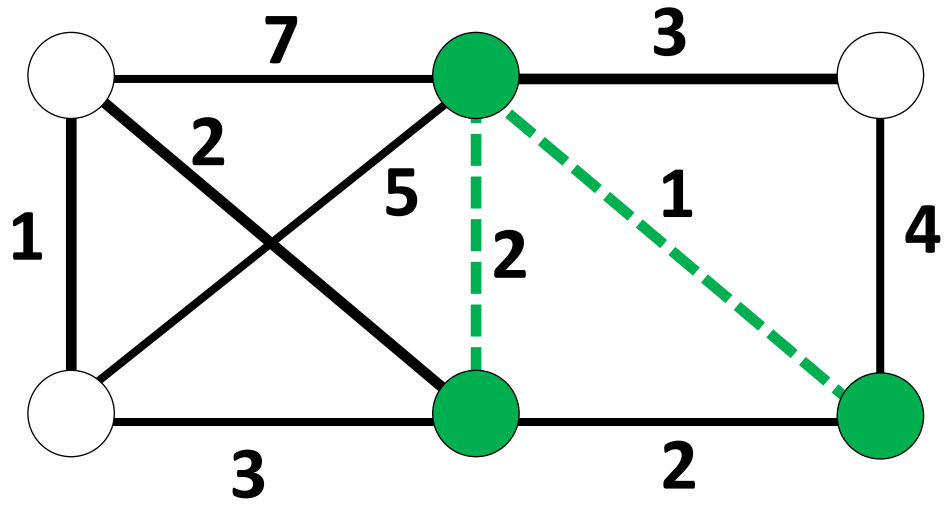
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



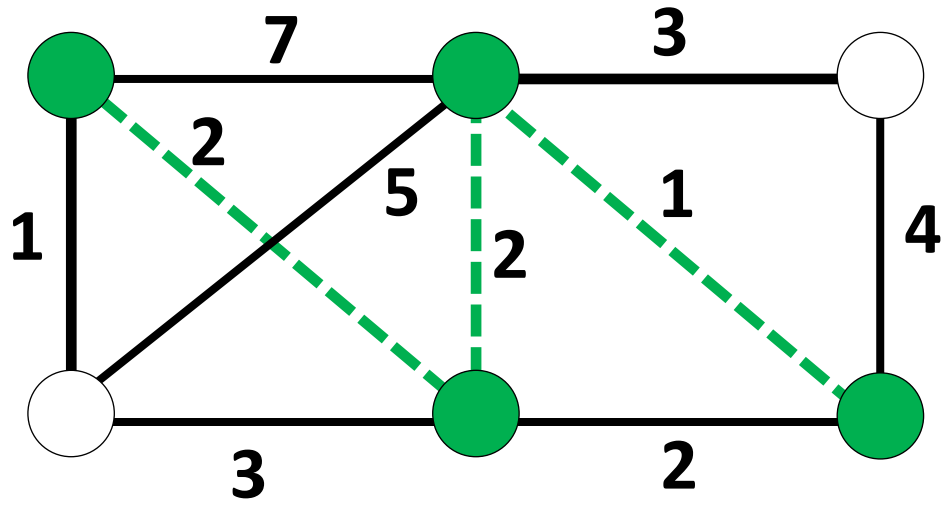
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



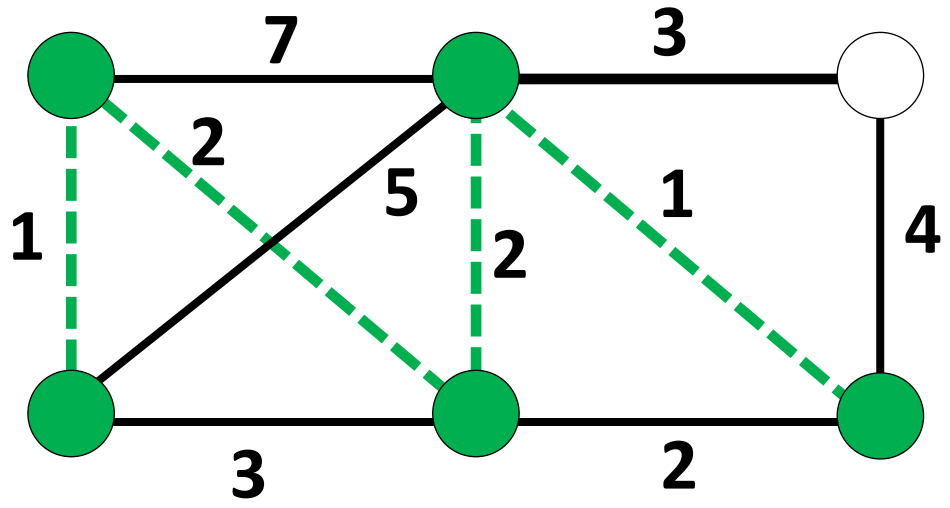
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



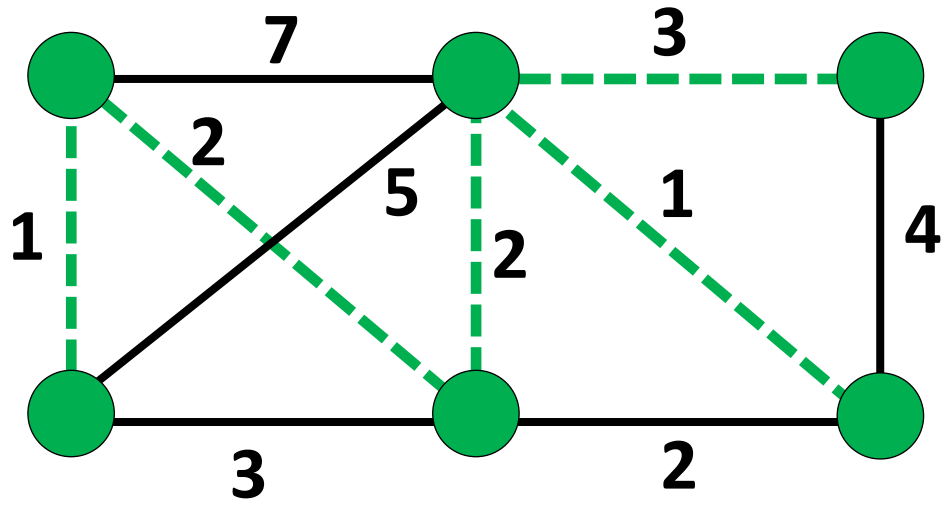
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



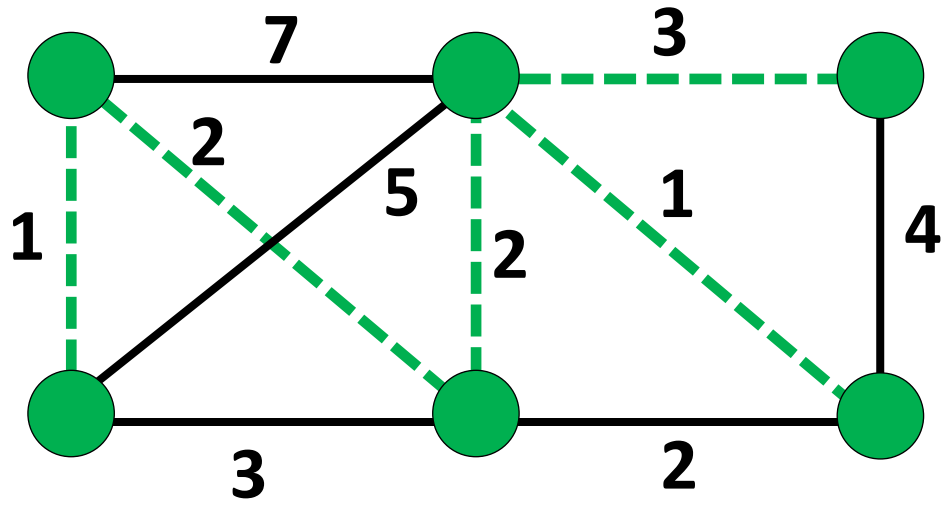
Prim's MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



Prim's MST Algorithm

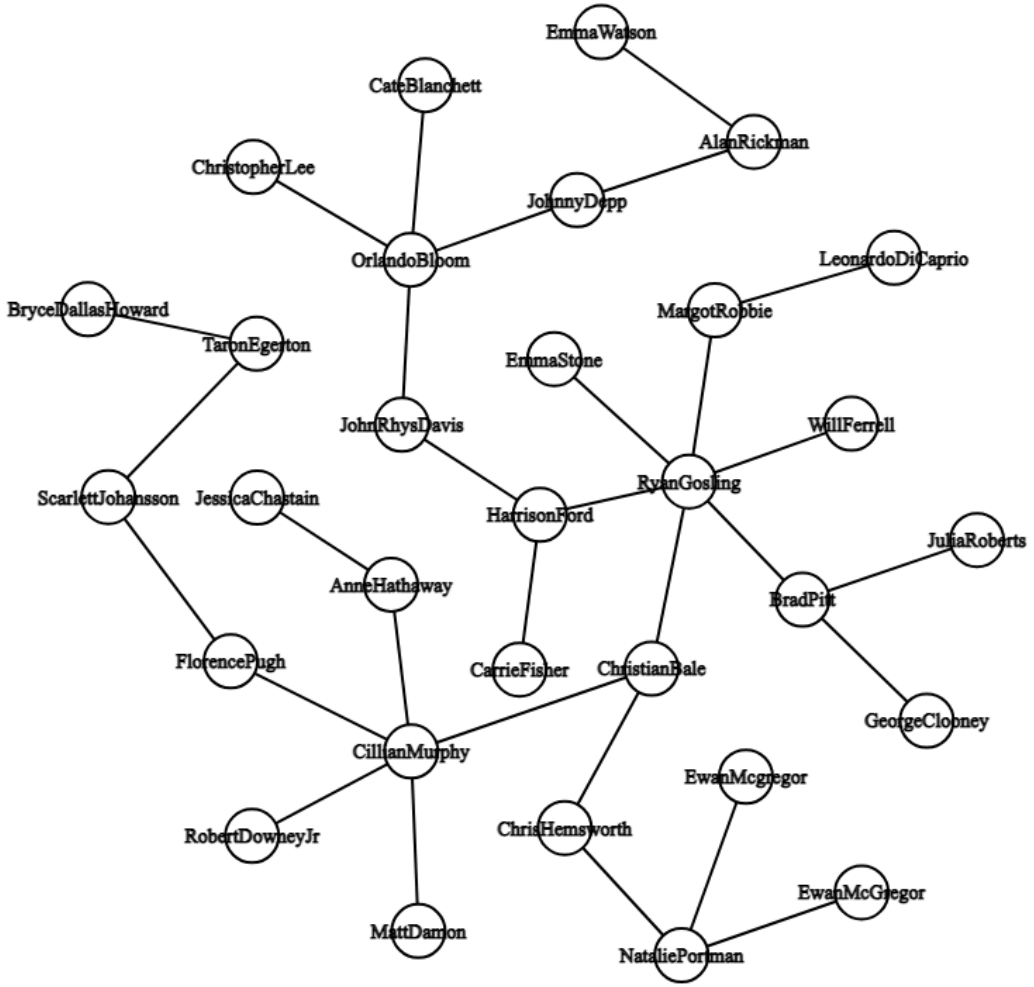
Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



We will use Kruskal's algorithm in this class, but you should at least be aware of what Prim's is

Prim's Algorithm has the same running time

Program 3



(Edge weights for program 3)