PRIM'S ALGORITHM!

WHAT WE'VE COVERED THIS SEMESTER

Prior to Spring Break we covered: Linear and Binary Search. Bubble, Selection, Insertion, Merge and Quick Sort. Since Spring Break we have covered Breadth-First Search, Depth-First Search, Hashing, Dijkstras, Dynamic Programming, Union Find, and Kruskal's Algorithm!

WHAT WE ARE COVERING TODAY

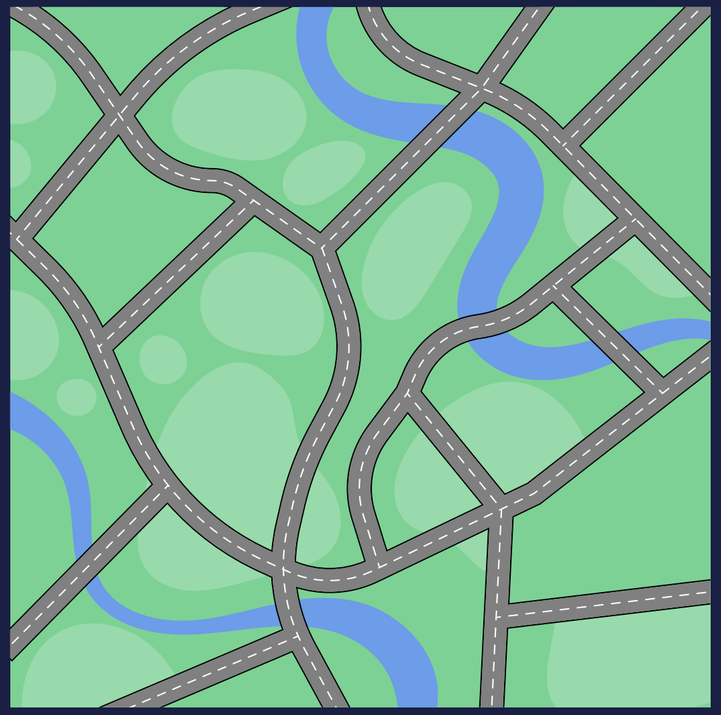
The ***LAST*** algorithm we are covering this semester is Prim's Algorithm.

WHAT IS PRIM'S?

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) for a connected, weighted graph.

It connects all the vertices together with the smallest total edge weight without creating any cycles.

WHAT?



Think of it like there is a bunch of cities and you want to build roads to connect them.

You want the cheapest overall cost without making loops.

WAIT! ISN'T THIS KRUSKAL'S?

Prim's and Kruskal's Algorithms are ***almost*** identical! They share a lot of the same concepts and can probably be used interchangeably.

SO WHAT'S THE DIFFERENCE?

Well, Kruskal's finds the path by treating all nodes as separate trees and adds the cheapest edge that connects two distinct trees along the way.

Prim's starts with a single tree and connects to it.

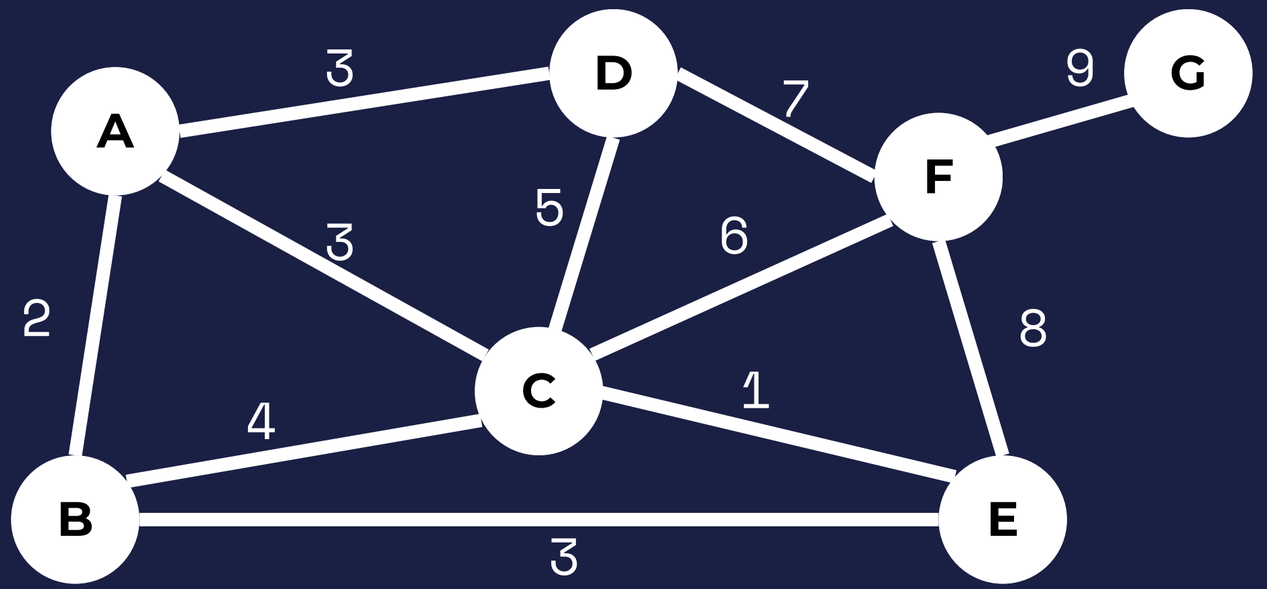
Lets consider a graph of values:

```
graph = {
  'A': {'B': 2, 'C': 3, 'D': 3},
  'B': {'A': 2, 'C': 4, 'E': 3},
  'C': {'A': 3, 'B': 4, 'D': 5, 'E': 1, 'F': 6},
  'D': {'A': 2, 'C': 5, 'F': 7},
  'E': {'B': 3, 'C': 1, 'F': 8},
  'F': {'D': 7, 'E': 8, 'G': 9},
  'G': {'F': 9}
}
```

But lets make it look like a graph:

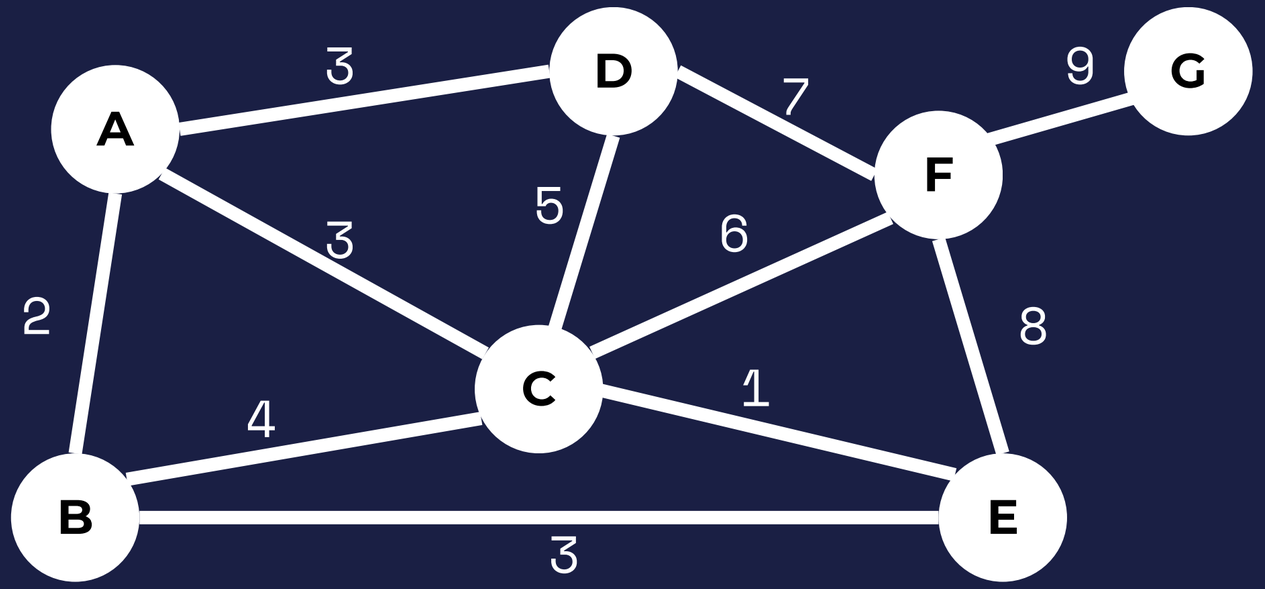
```
graph = {
  'A': {'B': 2, 'C': 3, 'D': 3},
  'B': {'A': 2, 'C': 4, 'E': 3},
  'C': {'A': 3, 'B': 4, 'D': 5, 'E': 1, 'F': 6},
  'D': {'A': 2, 'C': 5, 'F': 7},
  'E': {'B': 3, 'C': 1, 'F': 8},
  'F': {'D': 7, 'E': 8, 'G': 9},
  'G': {'F': 9}
}
```

But lets make it look like a graph



With Prims we create an empty array called “visited” to hold the nodes we’ve added to the tree.

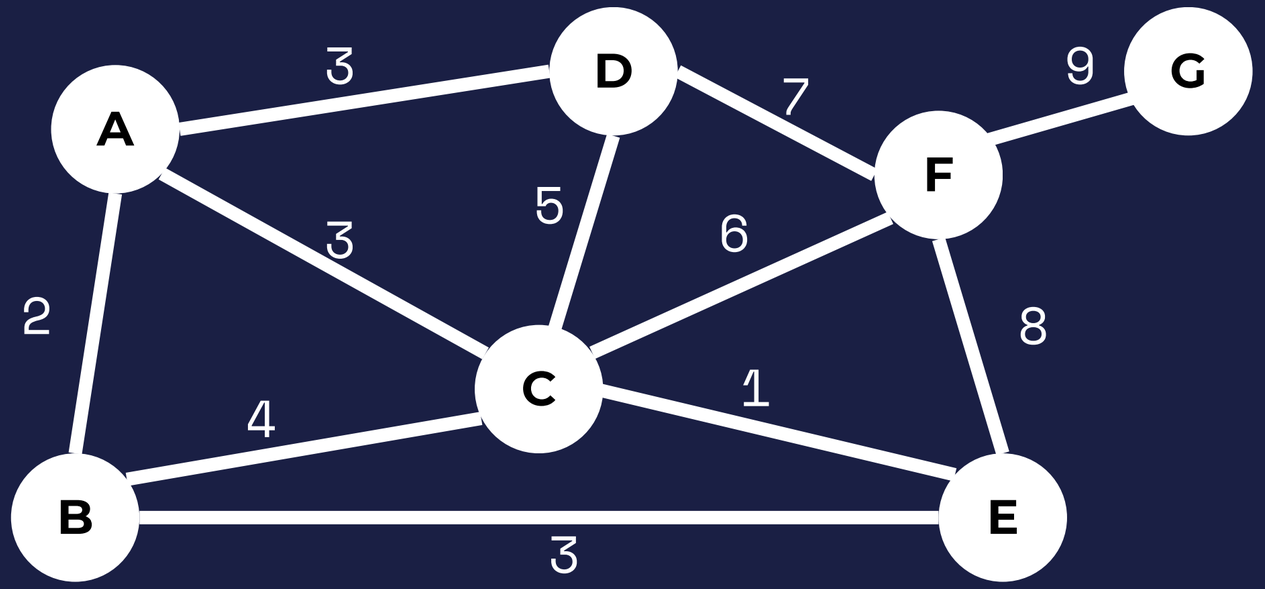
Spanning Tree



With Prims we create an empty array called “visited” to hold the nodes we’ve added to the tree.

Spanning Tree

```
visited = {

}
```

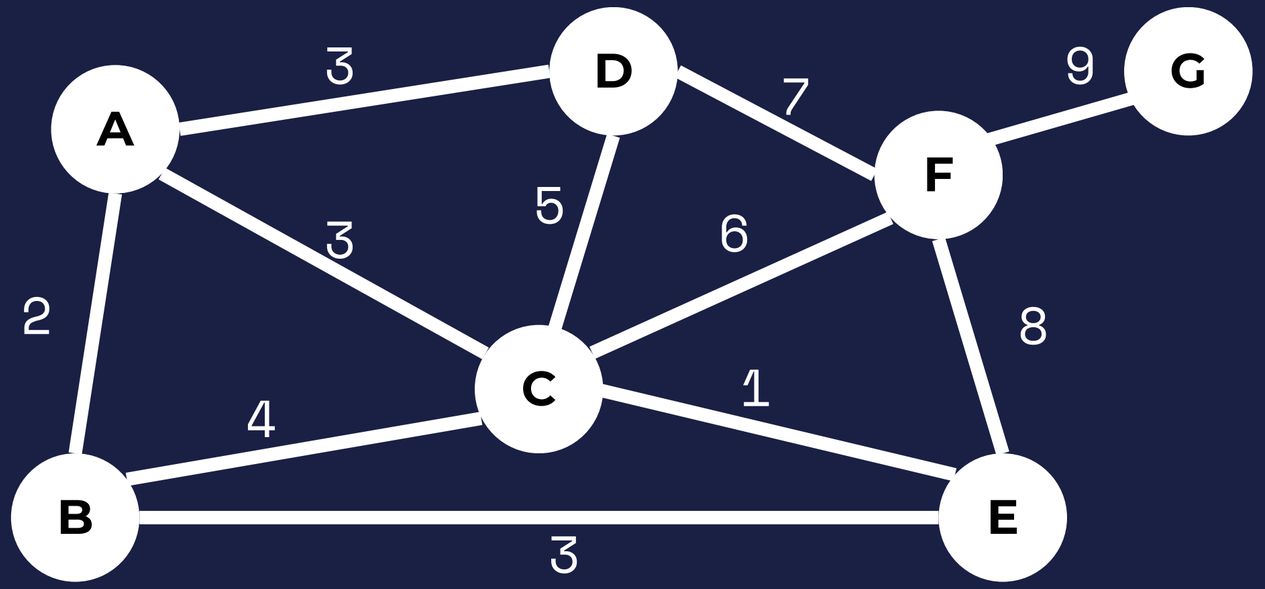


Next we need pick a node to start from, it can be any node, but lets pick “A”

Spanning Tree

visited = {

}

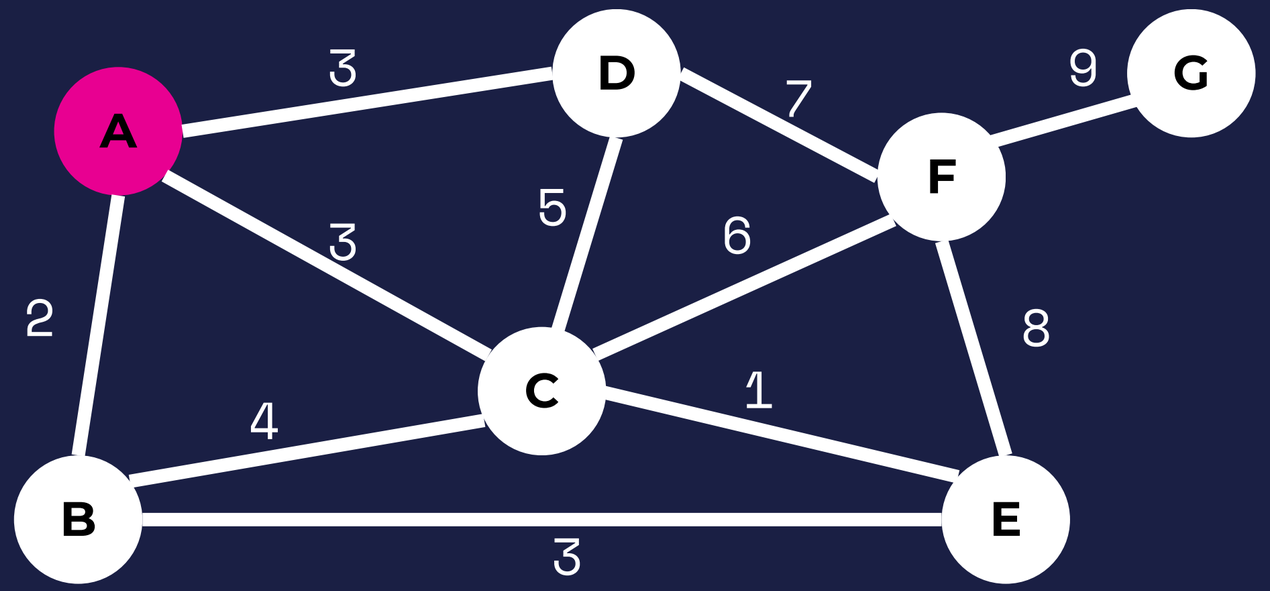


Next we need pick a node to start from, it can be any node, but lets pick “A”

Spanning Tree

visited = {

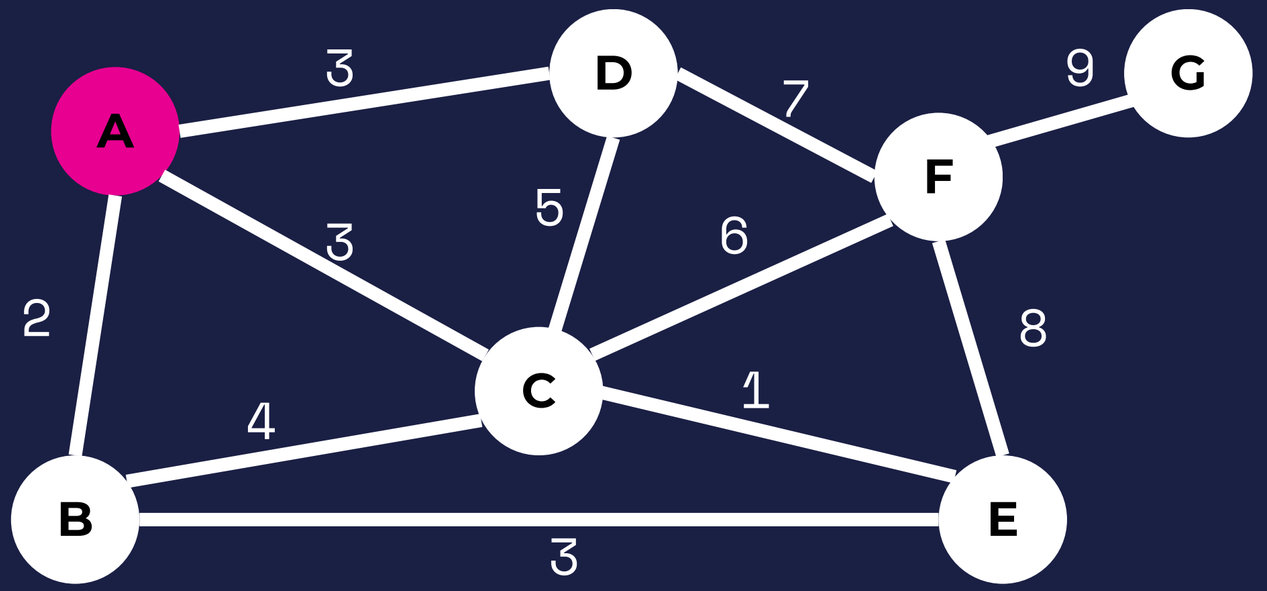
}



And we'll add "A" to the visited list

Spanning Tree

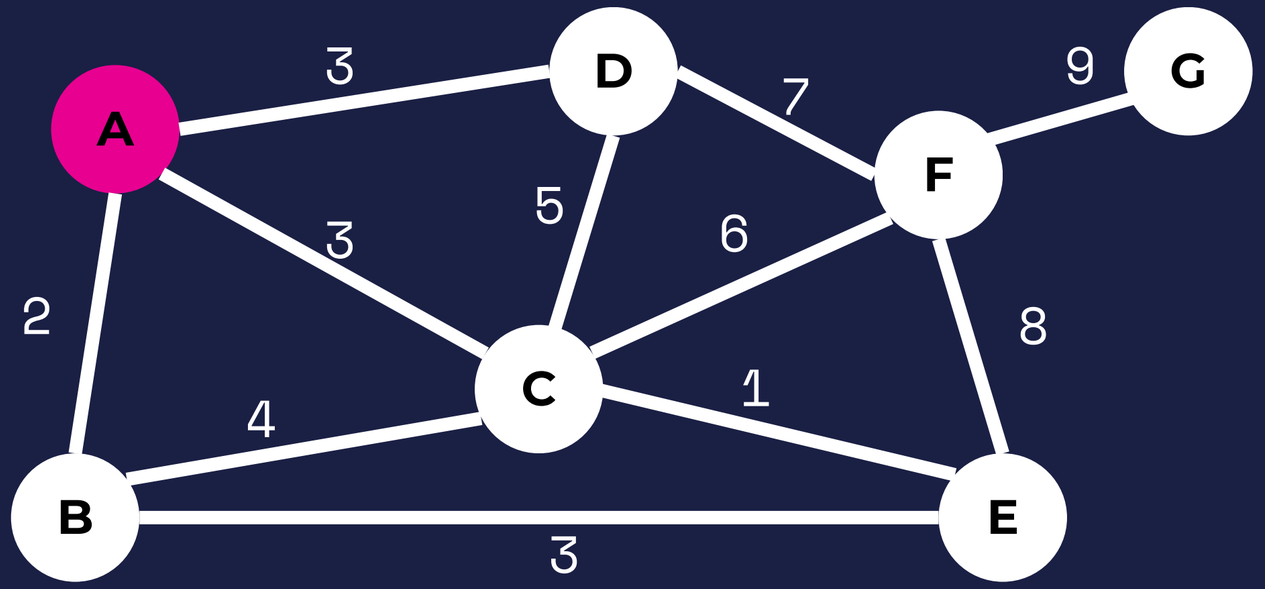
```
visited = {
    'A',

}
```



And we'll look at all the vertices that are connected to A. Which are Nodes B, C, and D.

Spanning Tree

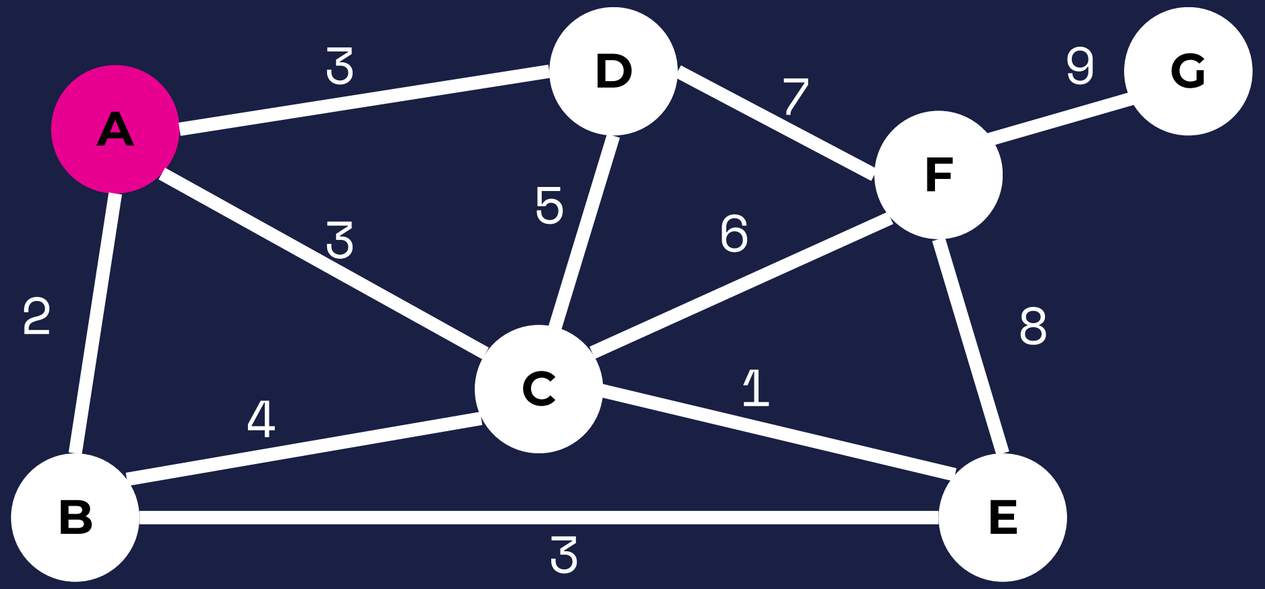
```
visited = {
    'A',

}
```



We want to choose the smallest possible weight to a nearby node. In this case, it would be “B” with a weight of 2.

Spanning Tree

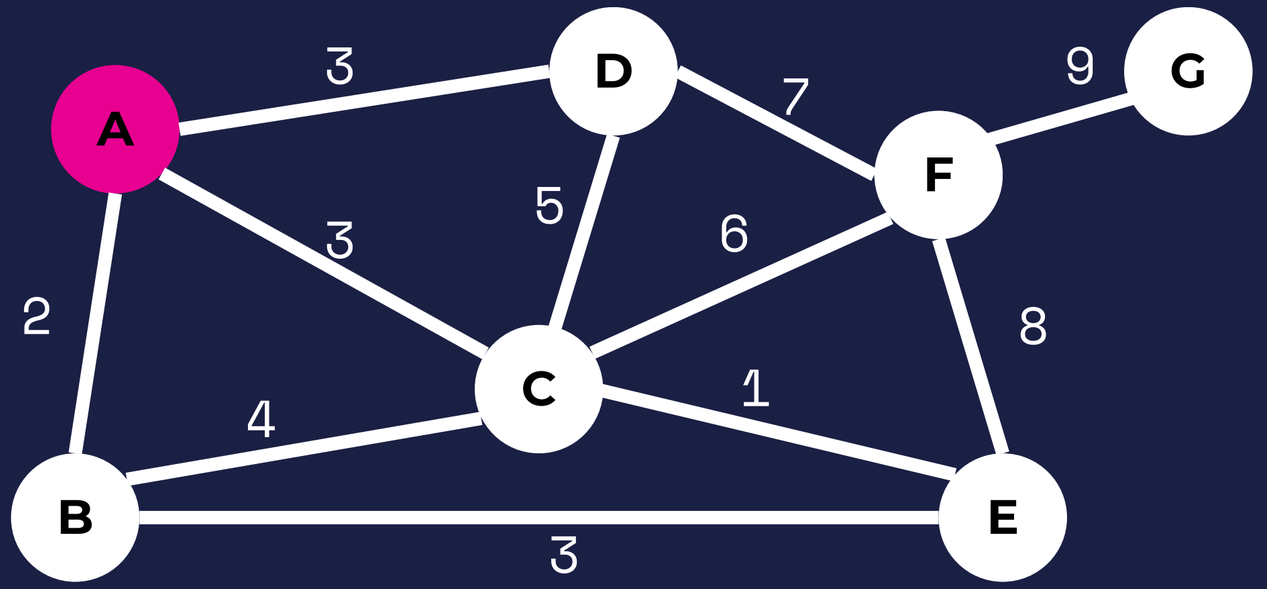
```
visited = {
    'A',

}
```



We'll mark "B" as visited, making it a part of our spanning tree.

Spanning Tree

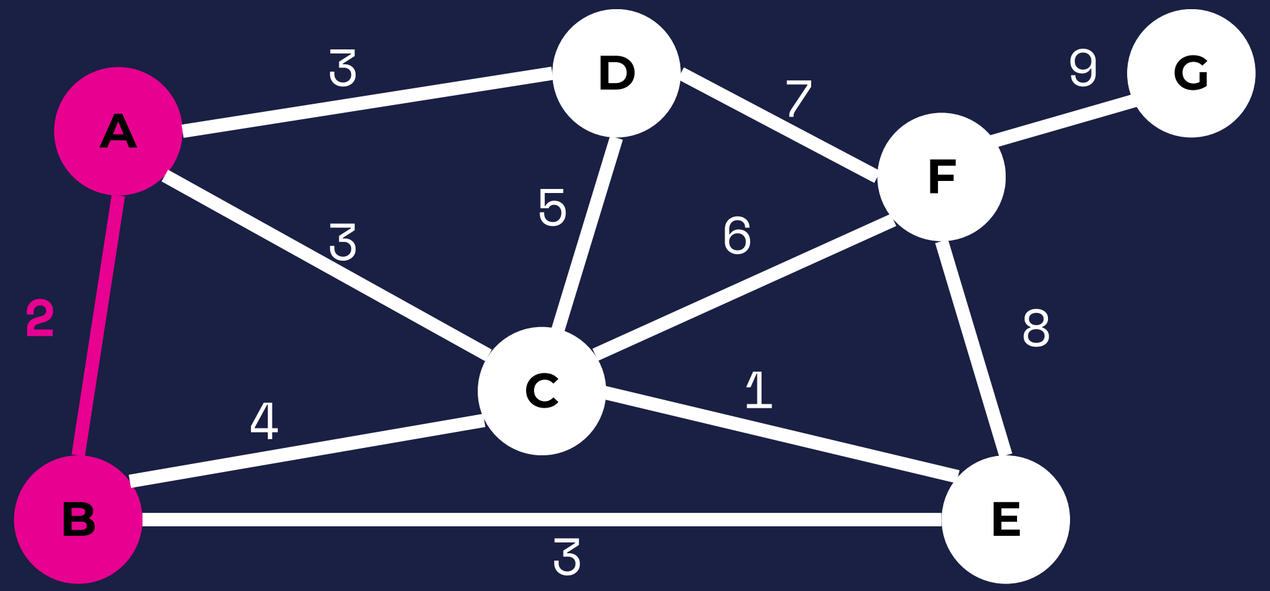
```
visited = {
    'A',

}
```



We'll mark "B" as visited, making it a part of our spanning tree.

Spanning Tree

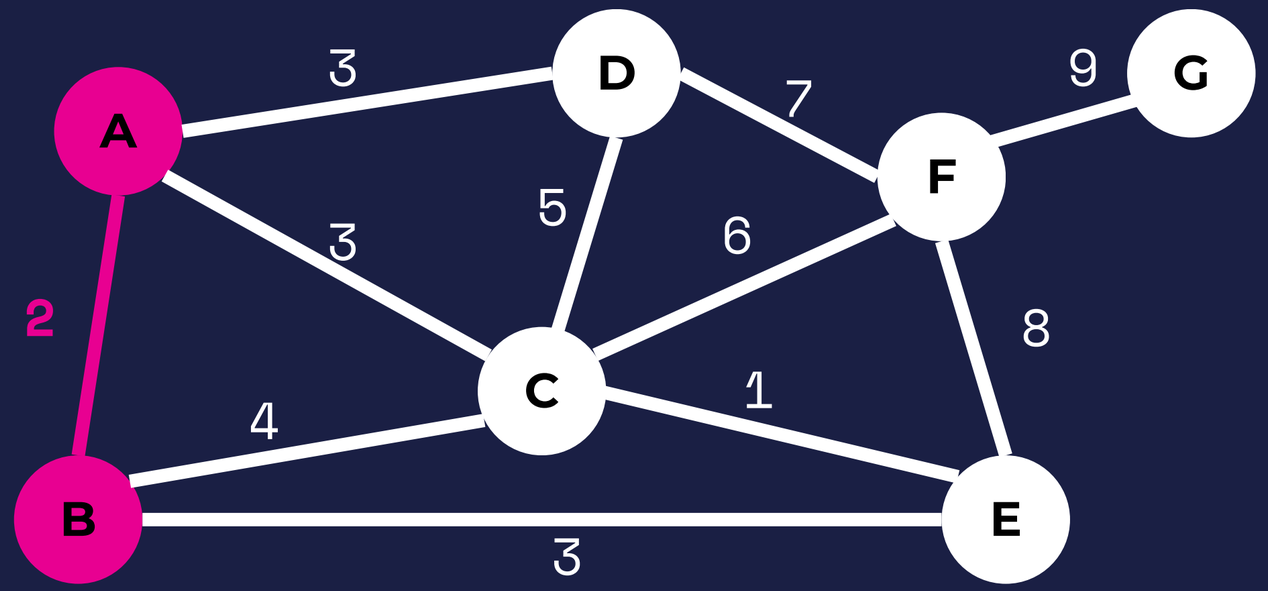
```
visited = {
    'A',
    'B',

}
```



Now we look at all nodes that are reachable by both A and B, which are C, D, and E and choose the smallest weight

Spanning Tree

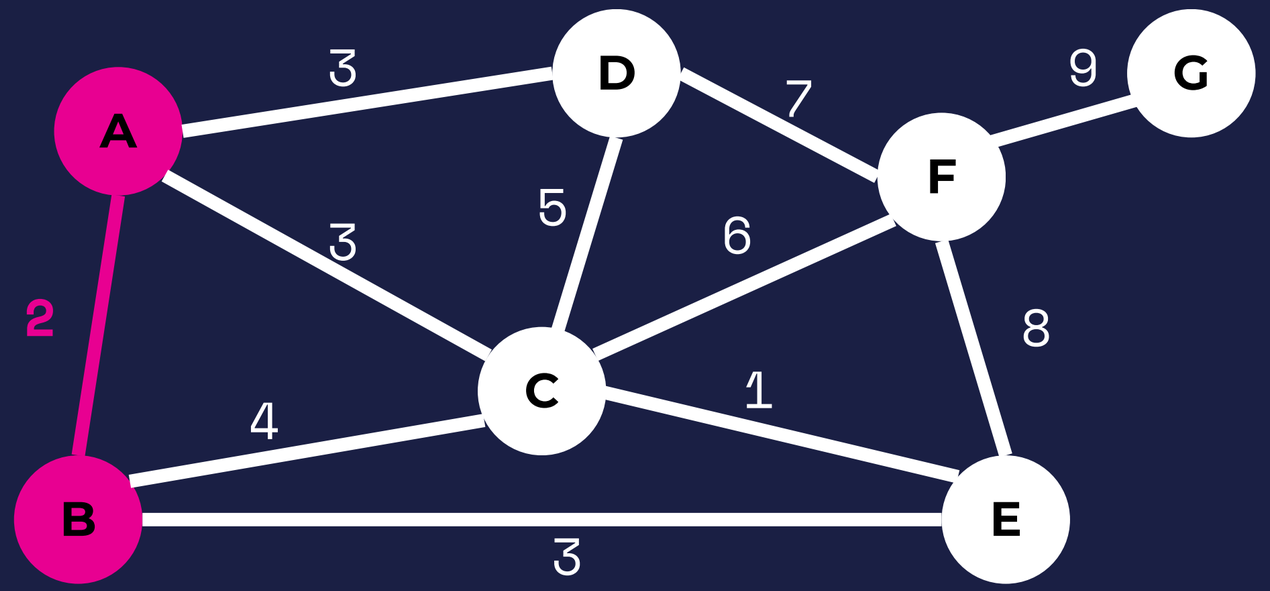
```
visited = {
    'A',
    'B',

}
```



A to D, A to C, and B to E all have a weight of 3.
We can choose any of them to start so, lets go with A to C.

Spanning Tree

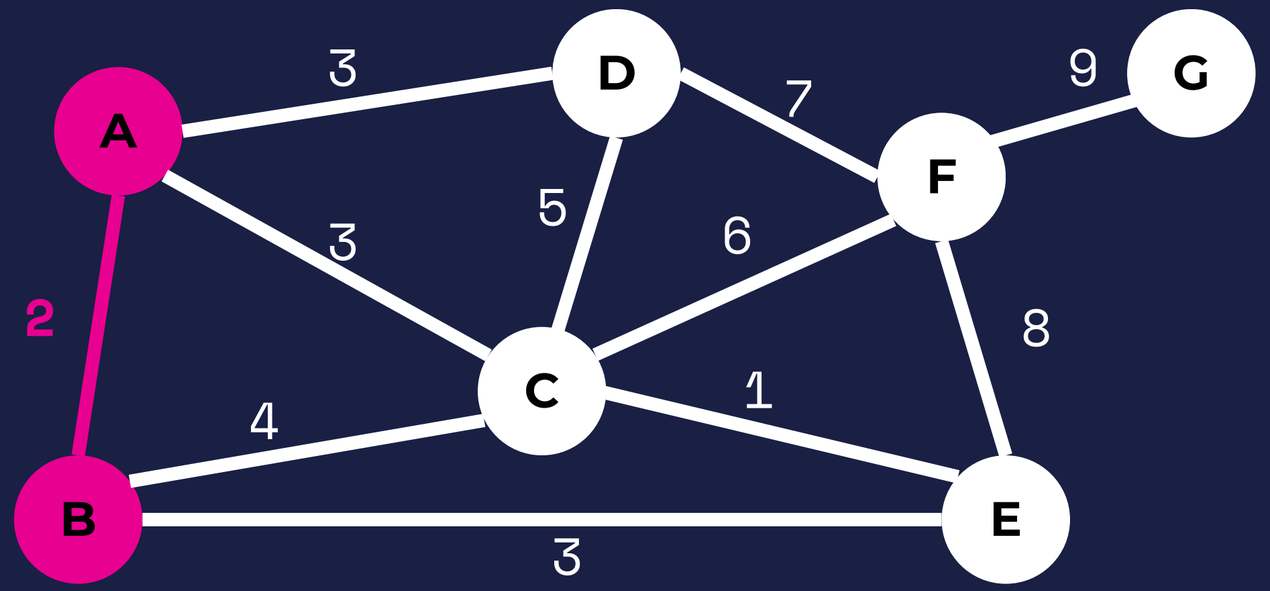
```
visited = {
    'A',
    'B',

}
```



We'll mark C as visited as we add it to our spanning tree.

Spanning Tree

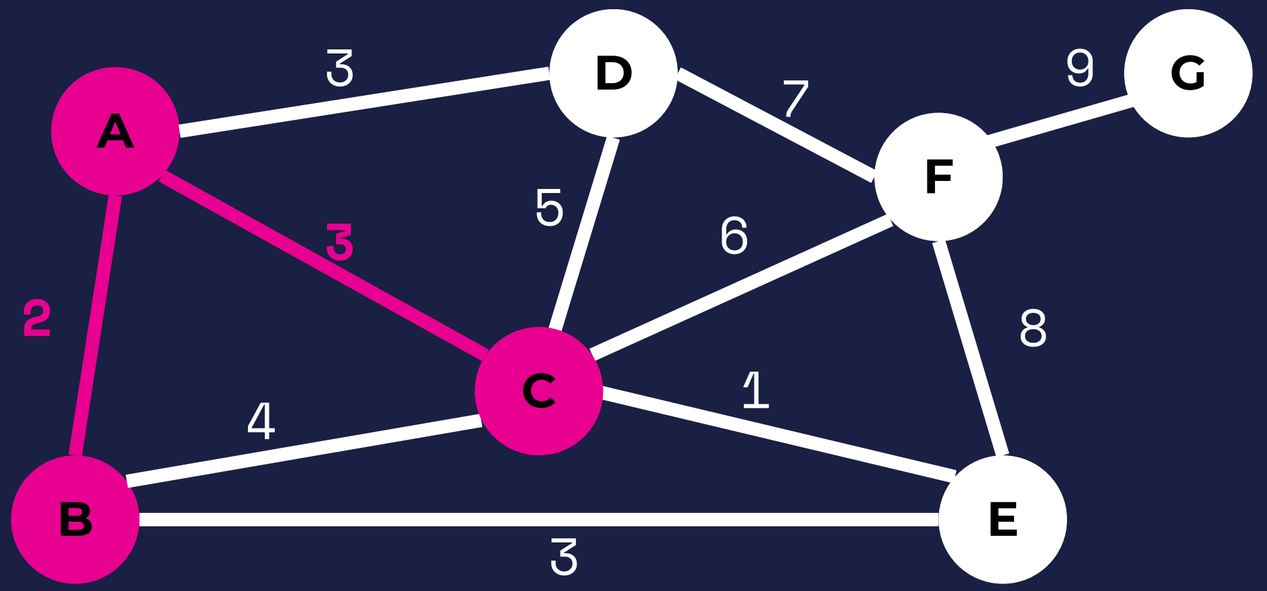
```
visited = {
    'A',
    'B',

}
```



We'll mark C as visited as we add it to our spanning tree.

Spanning Tree

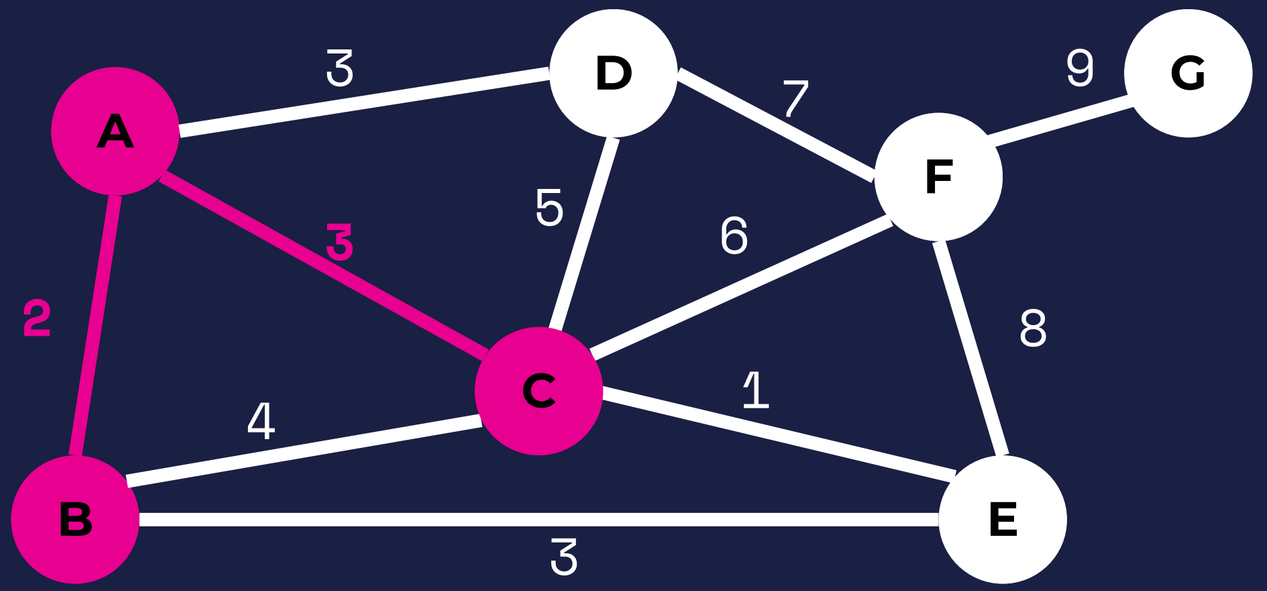
```
visited = {
    'A',
    'B',
    'C',

}
```



Now we look at all reachable nodes from A, B, and C. Which is D, E, and F.

Spanning Tree

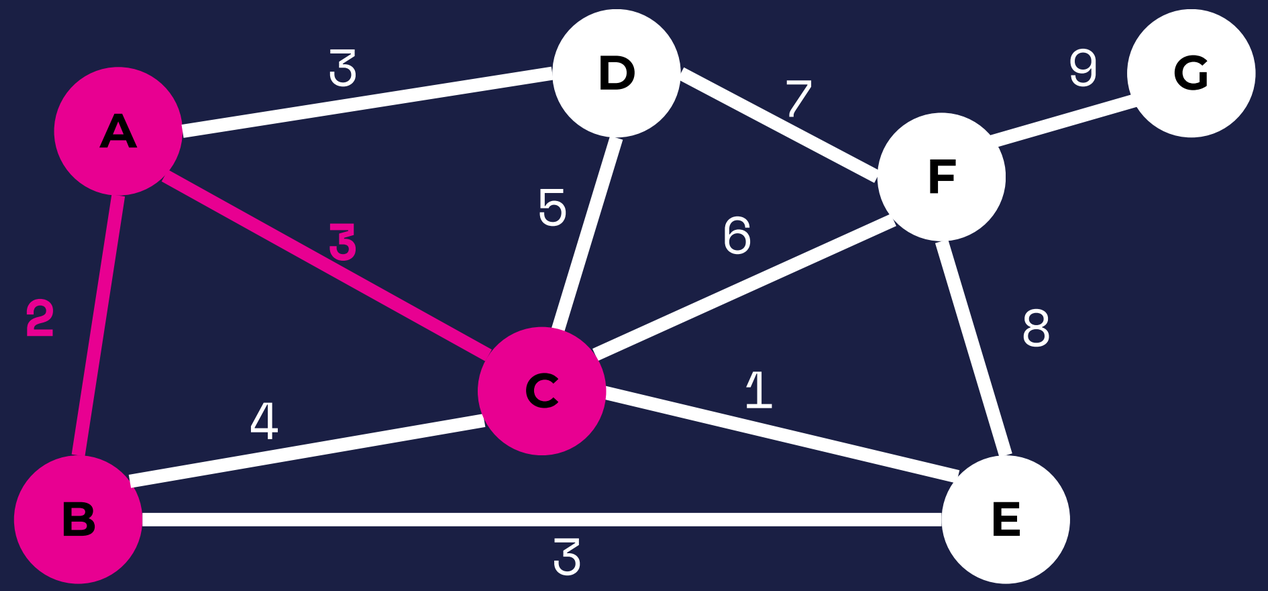
```
visited = {
    'A',
    'B',
    'C',

}
```



We choose the smallest weight, which in this case is C to E for 1. Lets add E to the visited nodes list and the spanning tree.

Spanning Tree

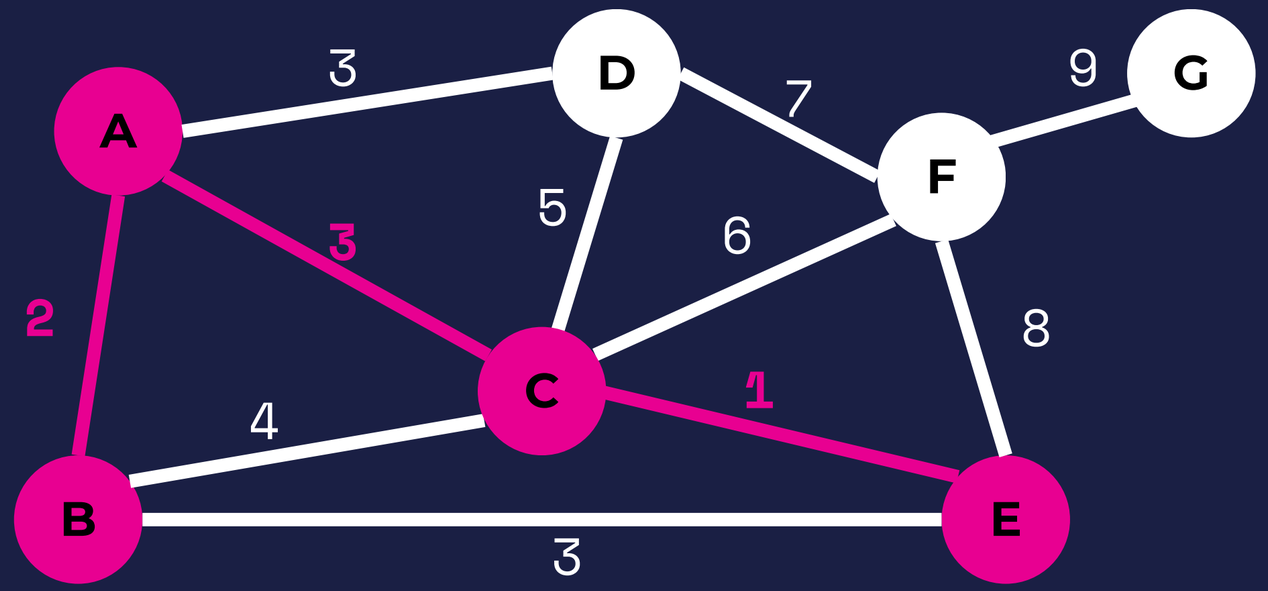
```
visited = {
    'A',
    'B',
    'C',

}
```



We choose the smallest weight, which in this case is C to E for 1. Lets add E to the visited nodes list and the spanning tree.

Spanning Tree

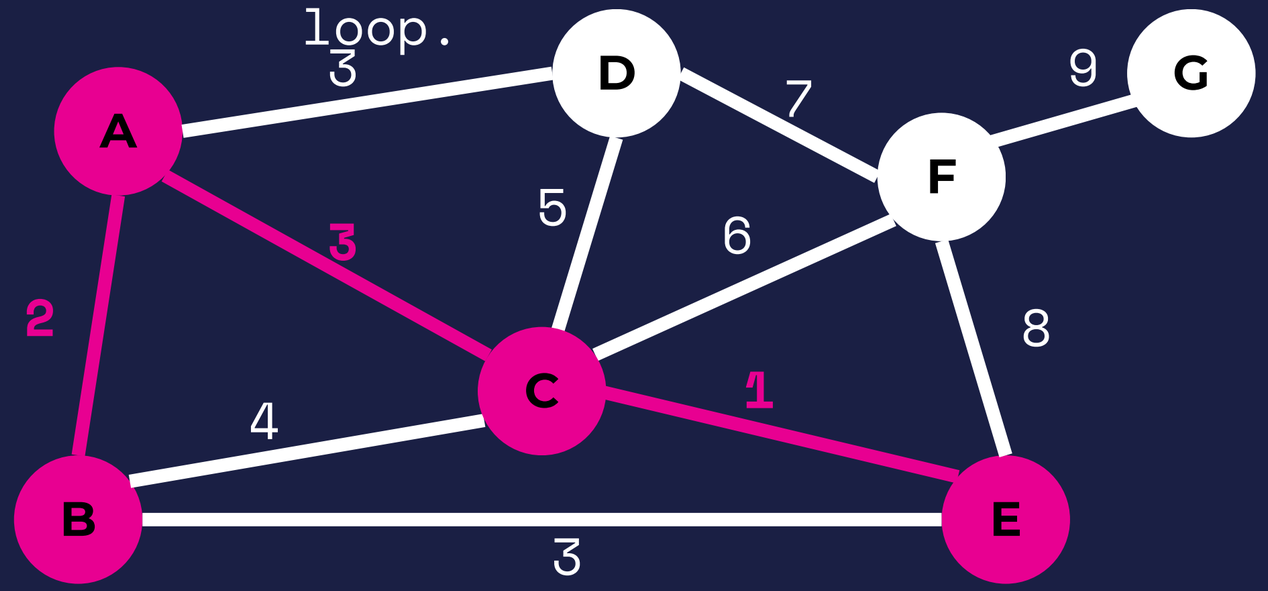
```
visited = {
    'A',
    'B',
    'C',
    'E',

}
```



Even though 3 is the next smallest and there are multiple 3s, we know we won't use the one between B and E because both are visited and it would cause a loop.

Spanning Tree

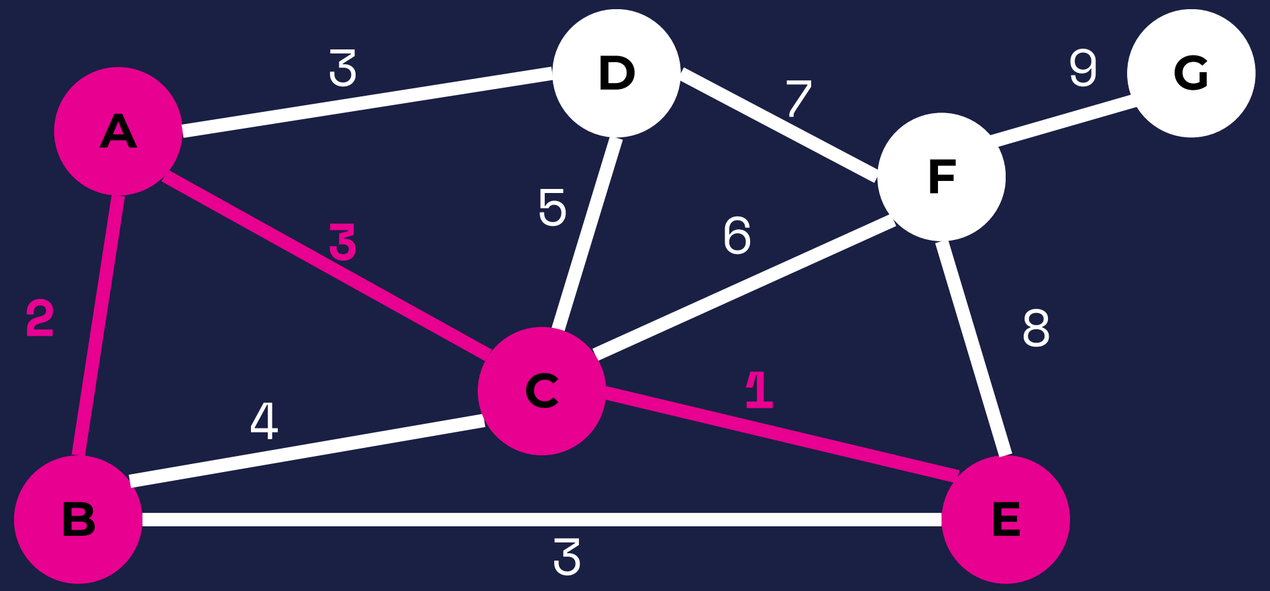
```
visited = {
    'A',
    'B',
    'C',
    'E',

}
```



So, we'll go with D! Lets add D to the spanning tree and visited list!

Spanning Tree

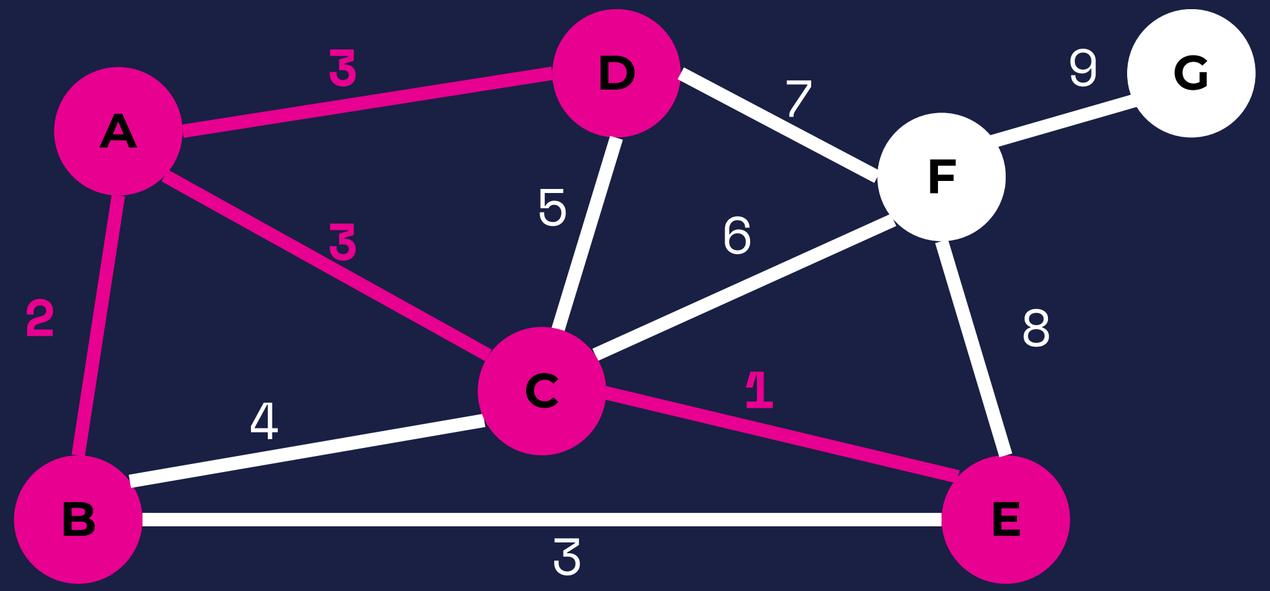
```
visited = {
    'A',
    'B',
    'C',
    'E',

}
```



So, we'll go with D! Lets add D to the spanning tree and visited list!

Spanning Tree

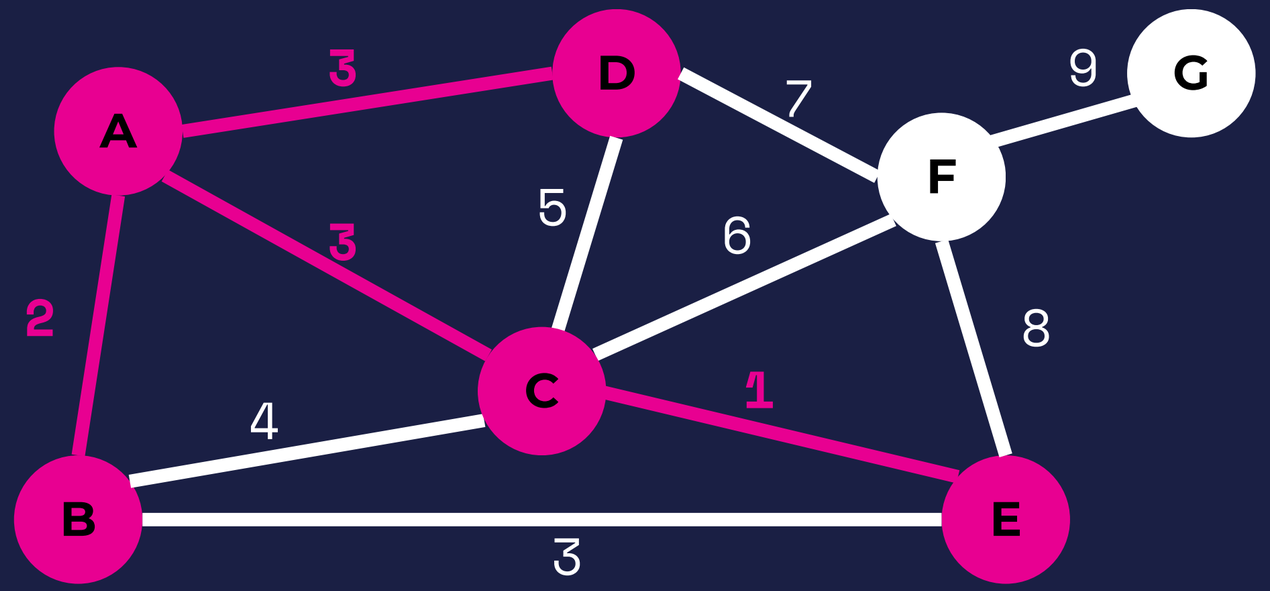
```
visited = {
    'A',
    'B',
    'C',
    'E',
    'D',

}
```



The next smallest weight is 4, which we won't use because both B and C are connected already

Spanning Tree

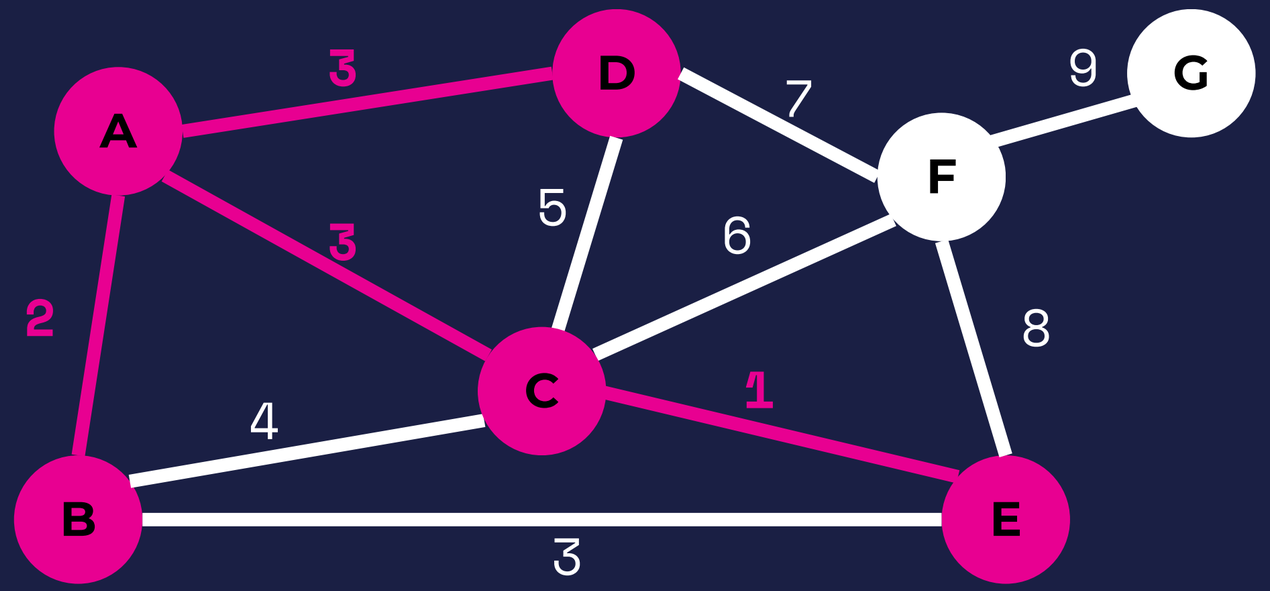
```
visited = {
    'A',
    'B',
    'C',
    'E',
    'D',

}
```



The next weight is 5, which we won't use because both C and D are already connected.

Spanning Tree

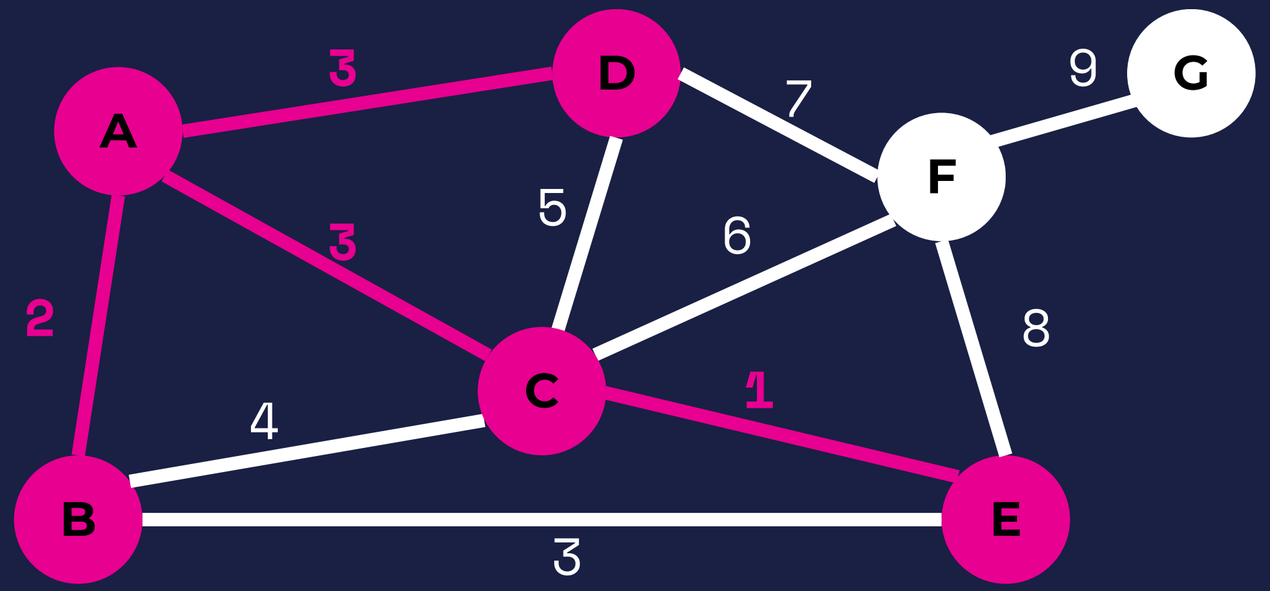
```
visited = {
    'A',
    'B',
    'C',
    'E',
    'D',

}
```



So now we will look at 6, and connect F!

Spanning Tree

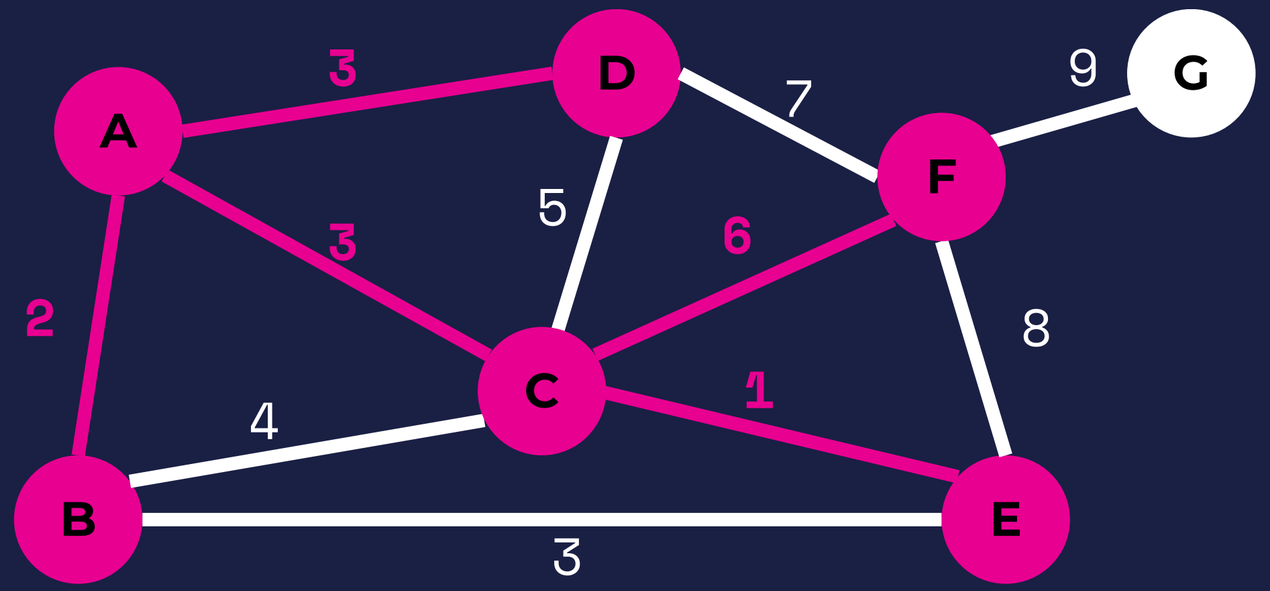
```
visited = {
    'A',
    'B',
    'C',
    'E',
    'D',

}
```



So now we will look at 6, and connect F!

Spanning Tree

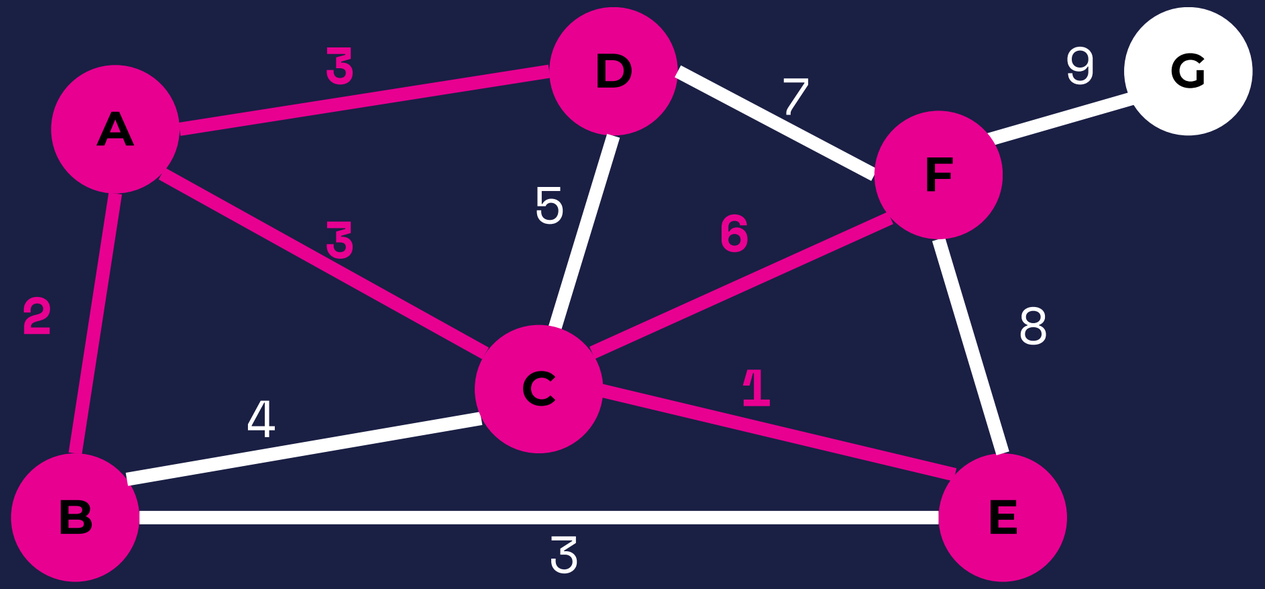
```
visited = {
    'A',
    'B',
    'C',
    'E',
    'D',
    'F',

}
```



We won’t use 7 or 8 because all of the nodes are already visited. We’ll connect G with 9!

Spanning Tree

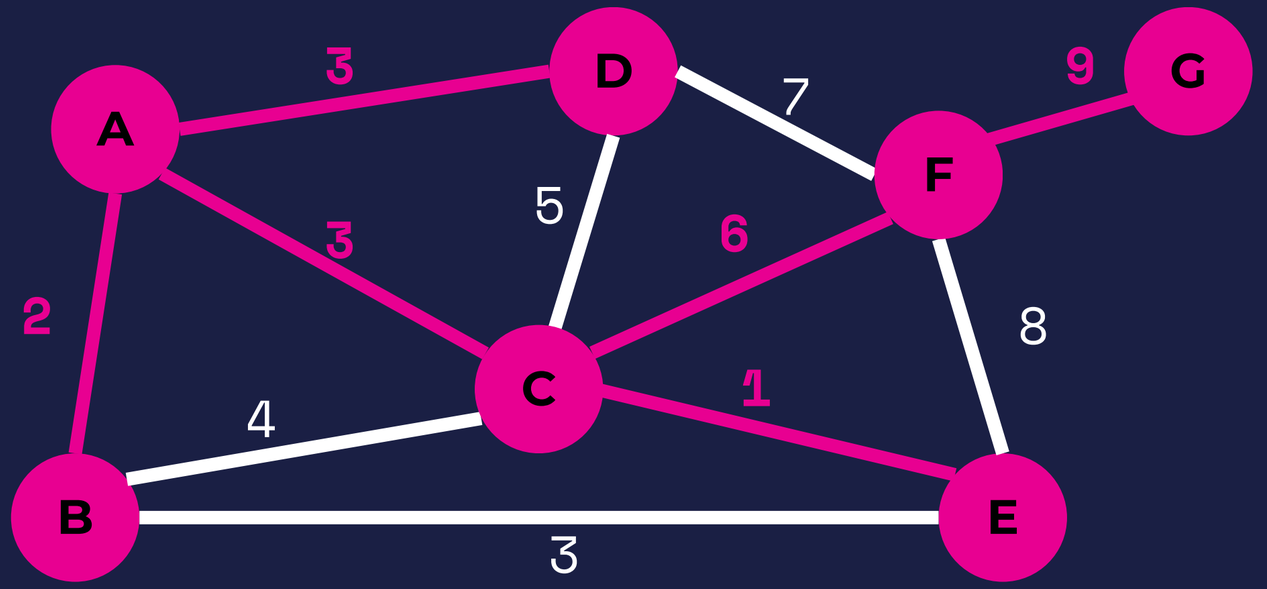
```
visited = {
    ‘A’,
    ‘B’,
    ‘C’,
    ‘E’,
    ‘D’,
    ‘F’,

}
```



We won’t use 7 or 8 because all of the nodes are already visited. We’ll connect G with 9!

Spanning Tree

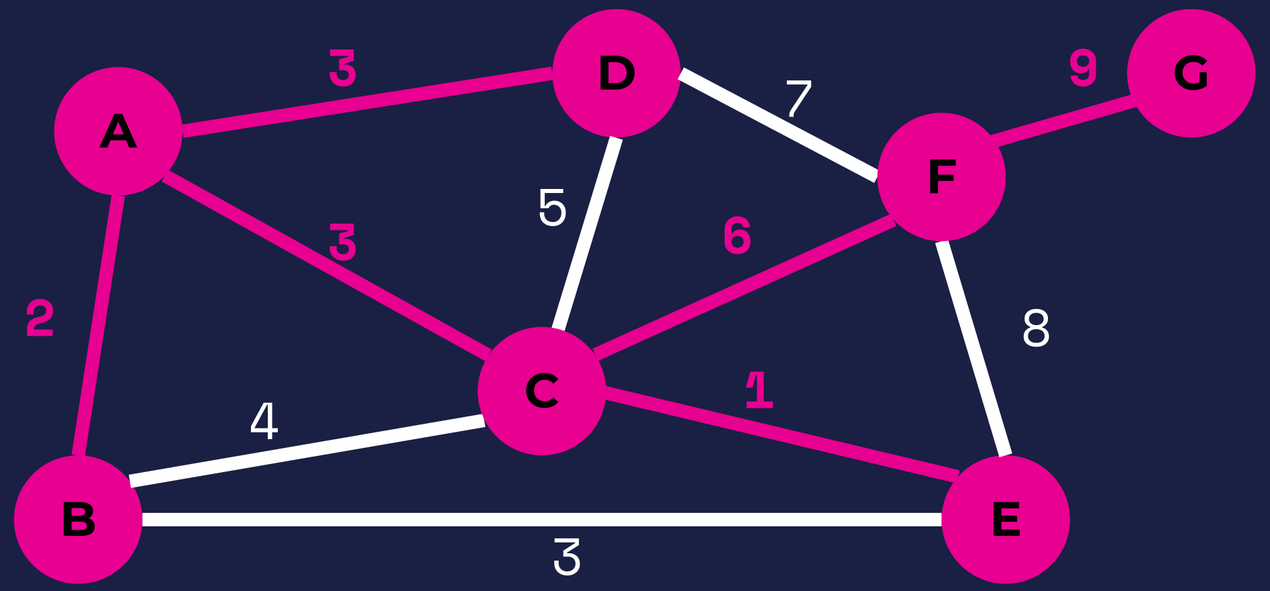
```
visited = {
    ‘A’,
    ‘B’,
    ‘C’,
    ‘E’,
    ‘D’,
    ‘F’,
    ‘G’
}
```



Since there shouldn’t be any loops, we should be able to remove all the white lines and see a spanning tree

Spanning Tree

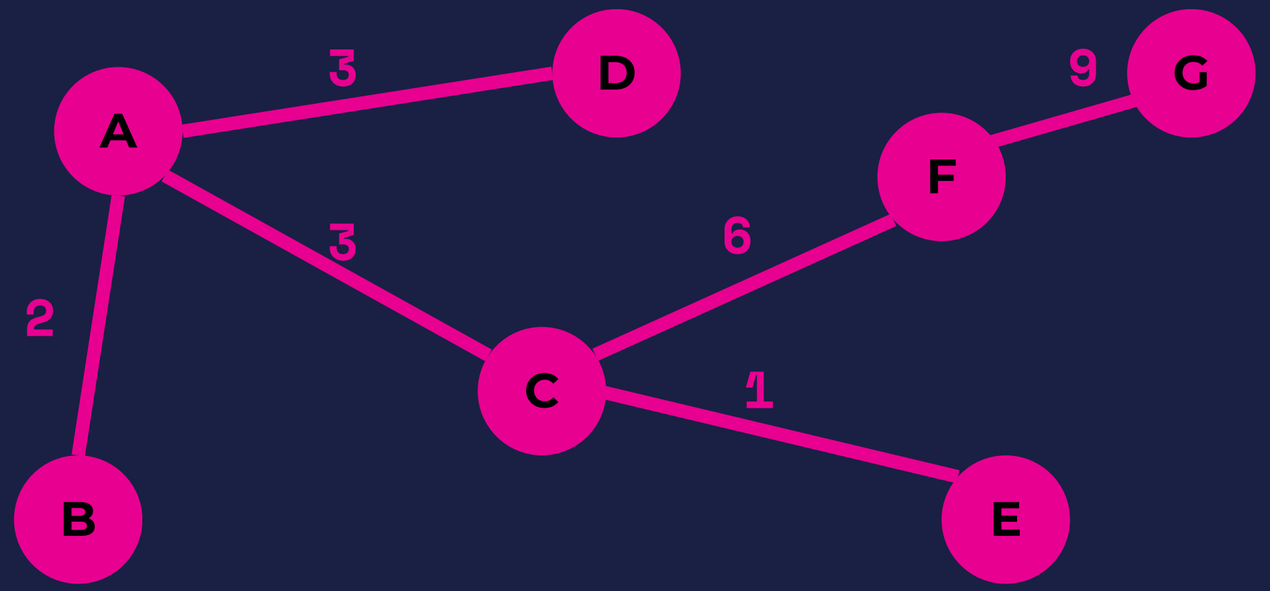
```
visited = {
    ‘A’,
    ‘B’,
    ‘C’,
    ‘E’,
    ‘D’,
    ‘F’,
    ‘G’
}
```



Since there shouldn’t be any loops, we should be able to remove all the white lines and see a spanning tree

Spanning Tree

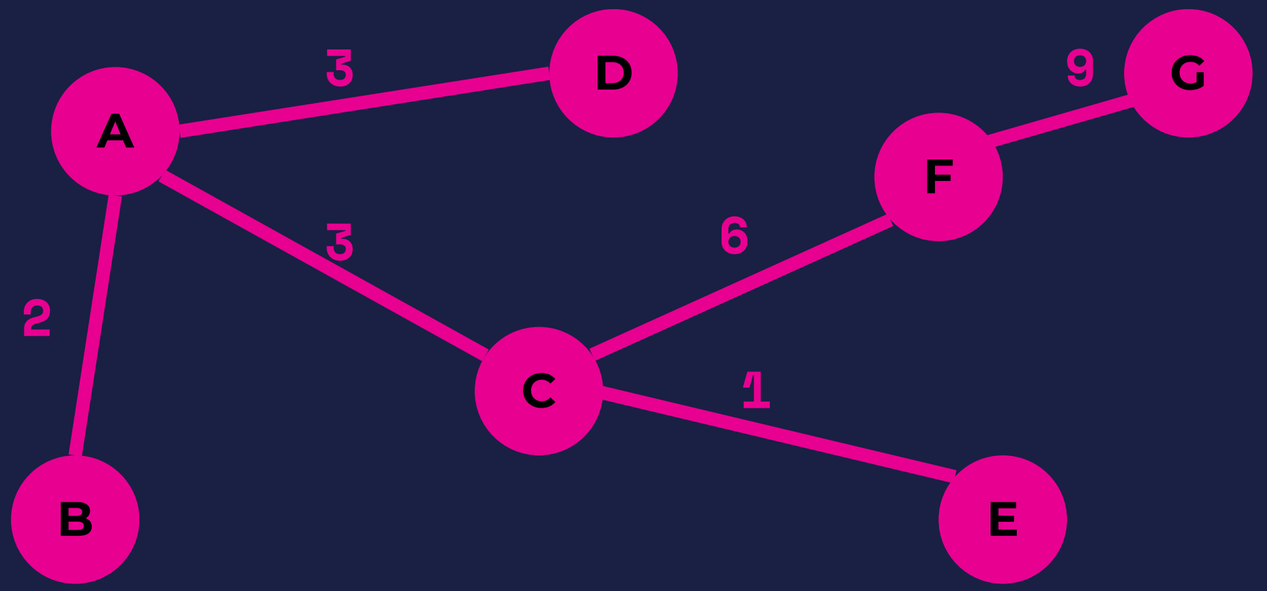
```
visited = {
    ‘A’,
    ‘B’,
    ‘C’,
    ‘E’,
    ‘D’,
    ‘F’,
    ‘G’
}
```



Tada!

Spanning Tree

```
visited = {
    'A',
    'B',
    'C',
    'E',
    'D',
    'F',
    'G'
}
```



THE PSEUDOCODE

```
function PRIMS(graph, start):
    min_spanning_tree = []
    visited = []
    edges = min-heap //(binary tree that finds highest/lowest value in a graph))

    WHILE edges is not empty:
        (weight, node, parent) ← remove smallest item from edges

        IF node is NOT in visited:
            visited.add(node)

            IF parent is NOT none then:
                min_spanning_tree.add(parent)

            FOR each (neighbor, cost) in graph[node]:
                IF neighbor NOT in visited:
                    edges.add(cost, neighbor, node)

    return min_spanning_tree
```

EXAMPLES REPLIT AND GITHUB! PLEASE GO TO:

HTTPS://REPLIT.COM/@RIKKIEHRHART/GRABABYTE

HTTPS://GITHUB.COM/RIKKITOMIKOEHRHART/GRABABYTE

UP NEXT

That’s it! We learned 15 Algorithms this semester!

Join us next semester where our topic will be: GitHub!

Questions? - rikki.ehrhart@g.ausitncc.edu

If you’d like the opportunity to run a Grab a Byte workshop, please let me know!