Data Structures

MST 2

CS 225
Brad Solomon

November 6, 2026





UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Review the minimum spanning tree (with weights)

Review Kruskal's / Prim's MST Algorithms

Focus on determining Big O of complex pseudocode

Compare implementations under different conditions

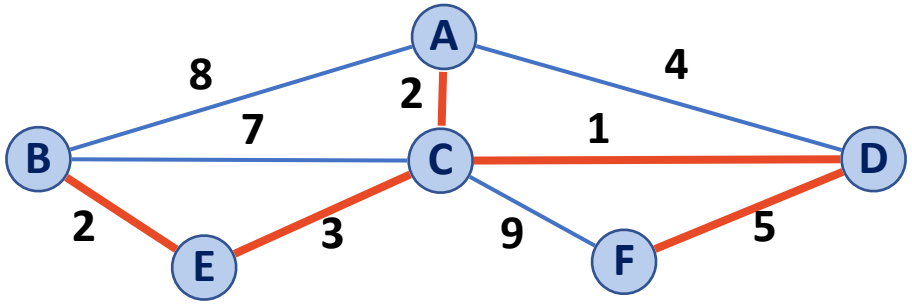
Minimum Spanning Tree Algorithms

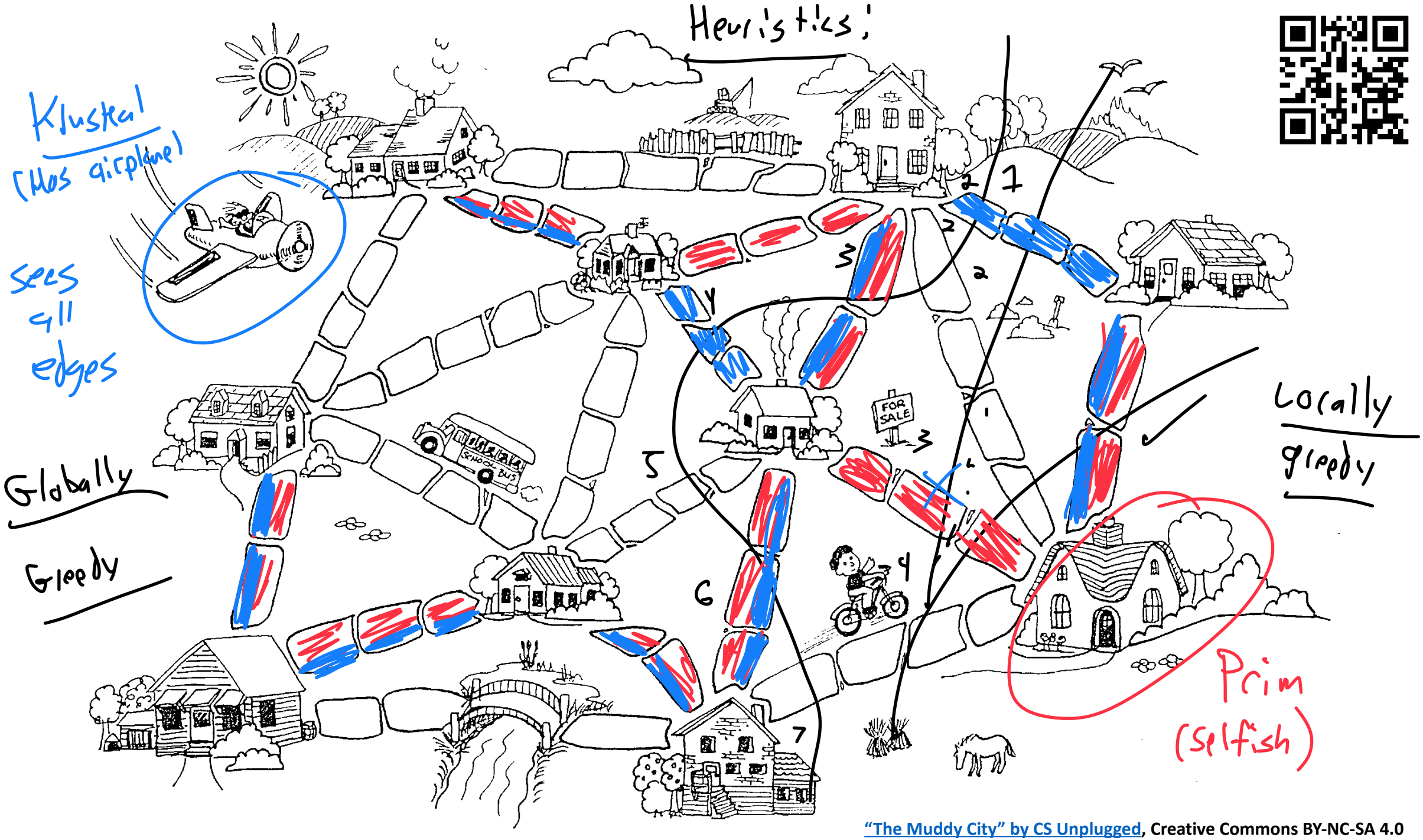
Input: Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

Output: A graph G' with the following properties:

- G' is a spanning graph of G
- G' is a tree (connected, acyclic)
- G' has a minimal total weight among all spanning trees

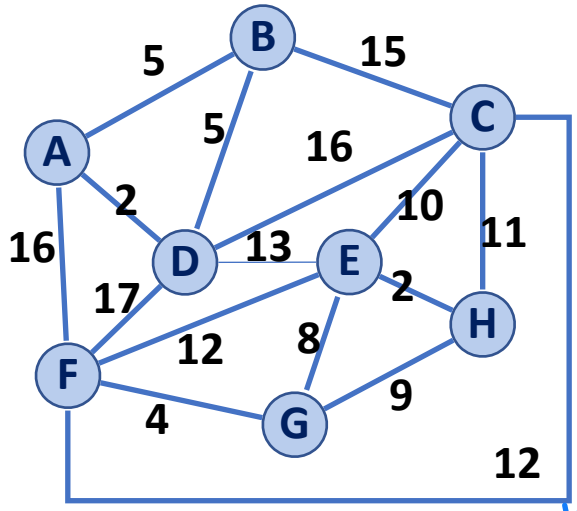
N - 1 (for N vertices)





"The Muddy City" by CS Unplugged, Creative Commons BY-NC-SA 4.0

Kruskal's Algorithm *(A graph algorithm for the MST problem)*



What information do I need to get efficiently?

1) The global minimum edge weight 75%

2) Edge connections for a given vertex 50%

↳ Never need this information

3) If two vertices are already connected 60%

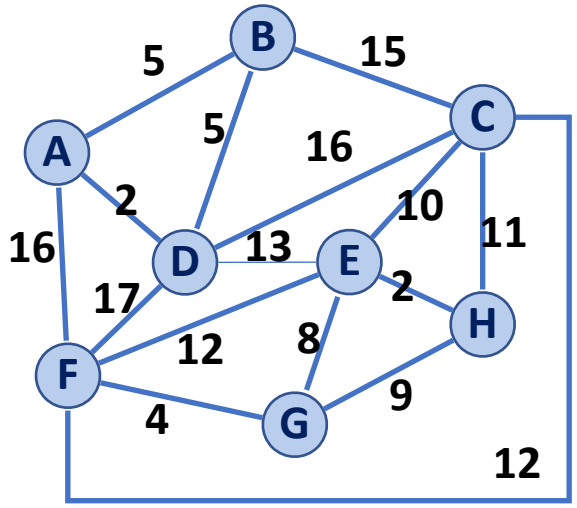
4) A visited list of vertices 60%

↳ We stop when N-1 edges in tree

what data structure does (1)? Priority Queue / min heap O(log n) or Sorted List O(1)

data structure does (3)? Disjoint set

Kruskal's Algorithm *(A graph algorithm for the MST problem)*



What information do I need to get efficiently?

1) The global minimum edge weight

3) If two vertices are already connected

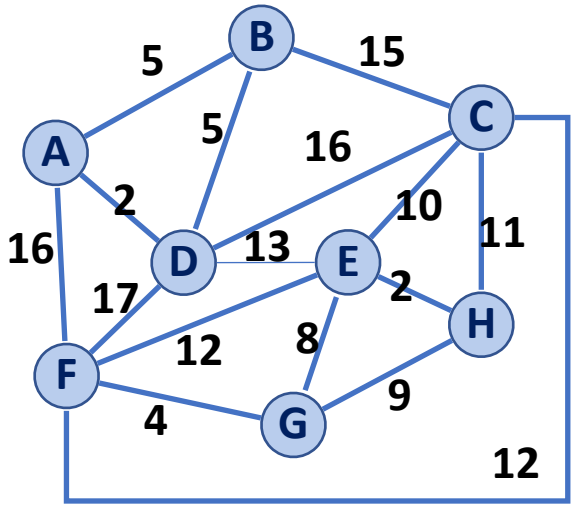
What does this tell me about my implementation choices?

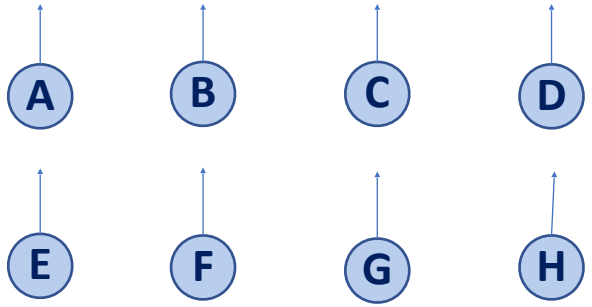
1) We need a **priority queue** of edges (sorted by weight)

2) We need a **disjoint set** of vertices

Kruskal's Algorithm

(A, D)
(E, H)
(F, G)
(A, B)
(B, D)
(G, E)
(G, H)
(E, C)
(C, H)
(E, F)
(F, C)
(D, E)
(B, C)
(C, D)
(A, F)
(D, F)





1) Build a **priority queue** on edges

↳ minheap

or

Sorted array

2) Build a **disjoint set** on vertices

Repeatedly:

↳ Get next min edge by weight

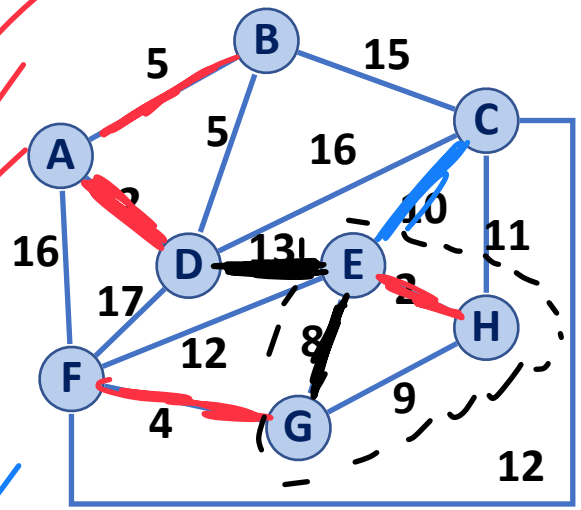
↳ If connects two unconnected vertices

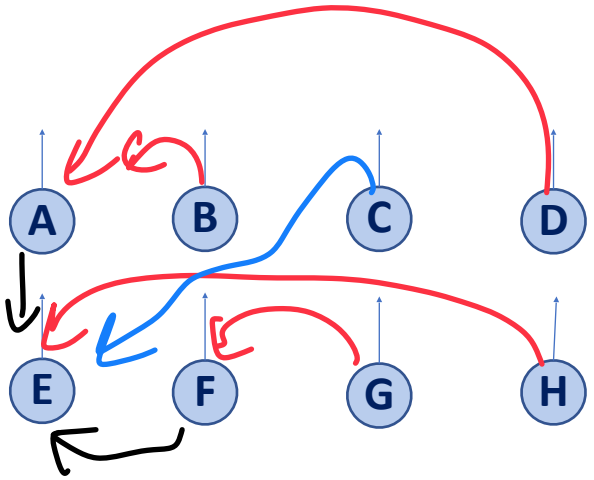
- Union Set
- Add edge to output

Stop when N-1 edges added

Kruskal's Algorithm

(A, D)
(E, H)
(F, G)
(A, B)
(B, D)
(G, E)
(G, H)
(E, C)
(C, H)
(E, F)
(F, C)
(D, E)
(B, C)
(C, D)
(A, F)
(D, F)





1) Build a **priority queue** on edges

A minheap

or

A sorted array

2) Build a **disjoint set** on vertices

All vertices start as their own set

3) Loop through min edges

If edge connects two disjoint sets

Union sets and record edge in MST

4) Stop when:

N-1 edges recorded

Only a single disjoint set remains

Kruskal's Algorithm

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

(2)

(1)

(n)

1) Build a **priority queue** on edges

A minheap

or

A sorted array

2) Build a **disjoint set** on vertices

All vertices start as their own set

3) Loop through min edges

If edge connects two disjoint sets

Union sets and record edge in MST

4) Stop when:

N-1 edges recorded

Only a single disjoint set remains

Kruskal's Algorithm

$|V| = n, |E| = m$

What is the Big O?



```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q      // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

O(n)

O(m)

O(log m)

O(n + m log m)

30 10

30 seconds

O(n)
O(m)
O(n log n)

O(m log m)

n + n log m / m + n log n

Kruskal's Algorithm

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

$|V| = n, |E| = m$

What is the Big O?

2 — 4: O(n)

6 — 7:
- Heap: O(m)
- Sorted List: O(m log m)

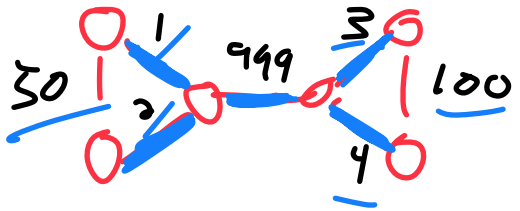
11: m x <12-17>

12—17:
- Heap: O(log m)
- Sorted List: O(1)

Tradeoff

Disjoint set we treat as O(1) b/c path compression w/ smart union

Kruskal's Algorithm



n = 6 vertices
m = 7 edges

"pie process"

Priority Queue:	Heap	Sorted Array
Building :7	O(m)	O(m log m)
Each removeMin :12	O(log m)	O(1)

m x

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

this is not looping n-1 times

m x

when would you prefer heap?

↳ construction is fast/query is slow

↳ If dynamic data (weight changes)

when would you prefer sorted array?

↳ It produces a sorted array at the end!

↳ Sorted array can be reused!

Kruskal's Algorithm

Priority Queue:	Heap	Sorted Array
Building :7	O(m)	O(m log m)
Each removeMin :12	O(log m)	O(1)

Both result in n + m log m

Why is heap good?

If edge weights can change!

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

Why is sorted array good?

Sorted array not destroyed and can be useful in other algorithms!

Kruskal's Algorithm



Priority Queue:	Total Running Time
Heap	
Sorted Array	

```
 1  KruskalMST(G):
 2    DisjointSets forest
 3    foreach (Vertex v : G.vertices()):
 4      forest.makeSet(v)
 5
 6    PriorityQueue Q     // min edge weight
 7    Q.buildFromGraph(G.edges())
 8
 9    Graph T = (V, {})
10
11    while |T.edges()| < n-1:
12      Vertex (u, v) = Q.removeMin()
13      if forest.find(u) != forest.find(v):
14         T.addEdge(u, v)
15         forest.union( forest.find(u),
16                       forest.find(v) )
17
18    return T
19
```

Kruskal's Algorithm

|V| = n
|E| = m

log m = log n² = 2 log n
drop the pow of log
dont care about const



Priority Queue:	Total Running Time
Heap	O(n)+O(m) + O(m log m)
Sorted Array	O(n) + O(m log m) + O(m)

O(n + m log m) or O(n + m log n)

Neat trick: O(log m) = O(log n)

Bounds of n & m relationship?

$$n - 1 \leq m \leq n^2$$

Connected graph *Complete graph*

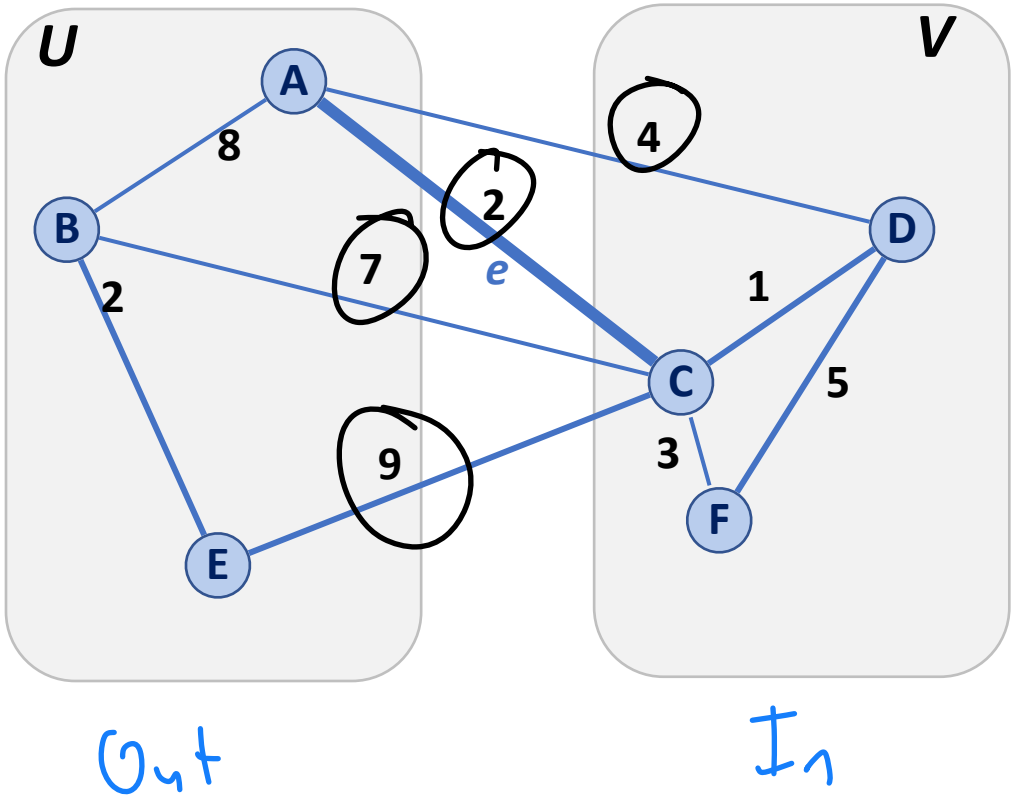
```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q     // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

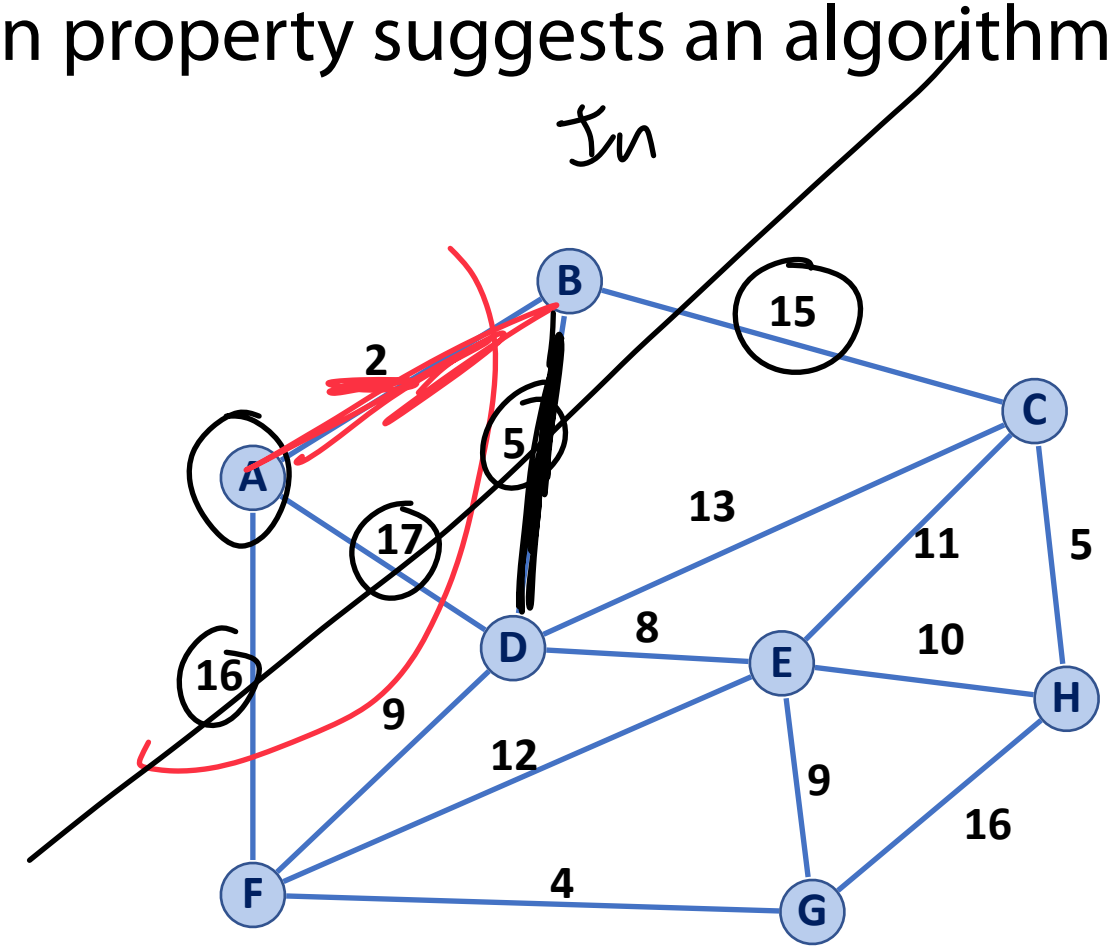
Let **e** be an edge of minimum weight across the partition.

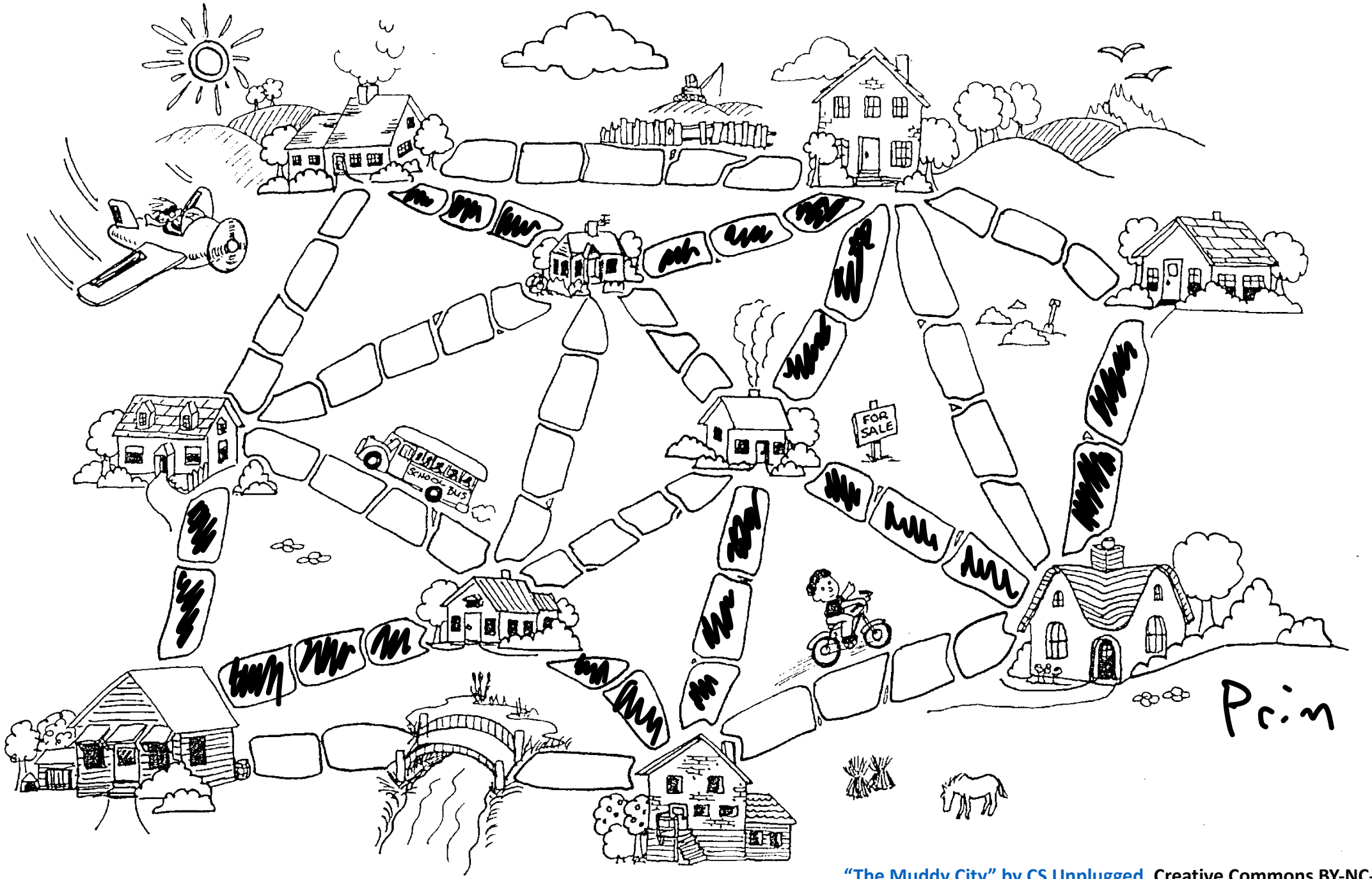
Then **e** is part of some minimum spanning tree.



Partition Property

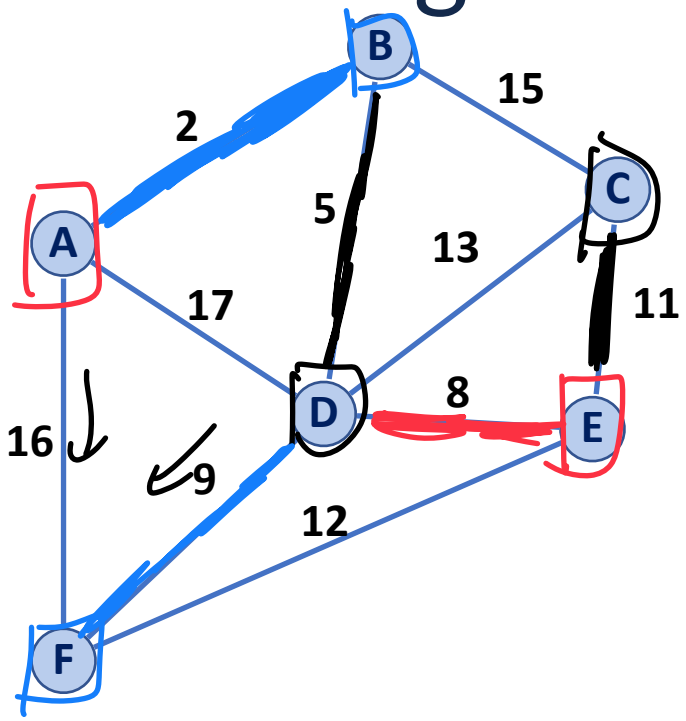
The partition property suggests an algorithm:





"The Muddy City" by CS Unplugged, Creative Commons BY-NC-SA 4.0

Prim's Algorithm



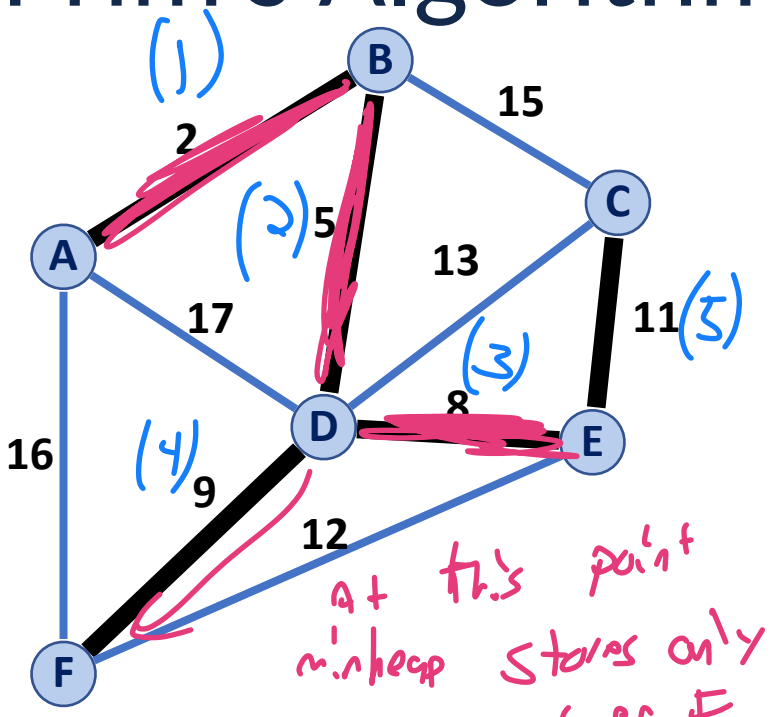
A	B	C	D	E	F
0, -	∞	∞	∞	∞	∞
	2, A	15, B	17, A	8, D	16, A
		13, D	5, B		9, D
		11, E			

```
1   PrimMST(G, s):
2     Input: G, Graph;
3            s, vertex in G, starting vertex
4     Output: T, a minimum spanning tree (MST) of G
5
6     foreach (Vertex v : G.vertices()):
7       d[v] = +inf
8       p[v] = NULL
9     d[s] = 0
10
11    PriorityQueue Q   // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T           // "labeled set"
14
15    repeat n times:
16      Vertex m = Q.removeMin()
17      T.add(m)
18      foreach (Vertex v : neighbors of m not in T):
19        if cost(v, m) < d[v]:
20          d[v] = cost(v, m)
21          p[v] = m
22
23    return T
```

all edges in m

if it's lower than q

Prim's Algorithm



at this point minheap stores only C or F

A	B	C	D	E	F
0, —	2, A	11, E	5, B	8, D	9, D

```
1  PrimMST(G, s):
2    Input: G, Graph;
3           s, vertex in G, starting vertex
4    Output: T, a minimum spanning tree (MST) of G
5
6    foreach (Vertex v : G.vertices()):
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q   // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T           // "labeled set"
14
15   repeat n times:
16     Vertex m = Q.removeMin()
17     T.add(m)
18     foreach (Vertex v : neighbors of m not in T):
19       if cost(v, m) < d[v]:
20         d[v] = cost(v, m)
21         p[v] = m
22
23   return T
```

Is this really O(1)?

Prim's Big O

$|V| = n, |E| = m$

Init

complex!

```
 6  PrimMST(G, s):
 7    foreach (Vertex v : G.vertices()):
 8      d[v] = +inf
 9      p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

Prim's Big O

|V|= n, |E|= m

7 — 9: O(n)

12—14:

MinHeap: O(n)

Unsorted Array: O(1)

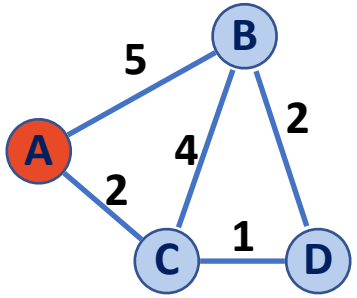
16—22: Complicated!

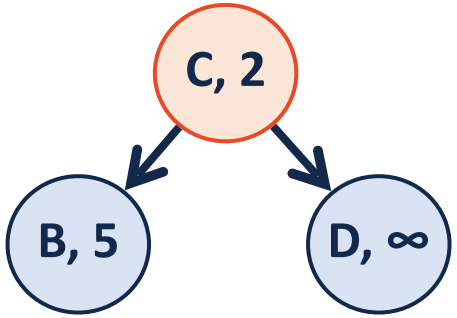
```
 6  PrimMST(G, s):
 7    foreach (Vertex v : G.vertices()):
 8      d[v] = +inf
 9      p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

Depends on choice of **PriorityQueue** (MinHeap vs Unsorted Array)

Depends on choice of **Graph** (Adjacency Matrix vs Adjacency List)

A	B	C	D
0	5	2	∞

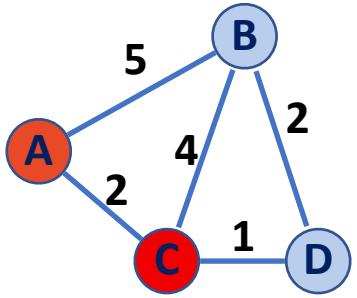


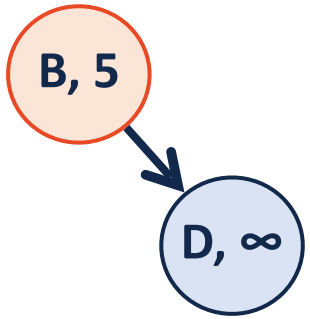



```
 6  PrimMST(G, s):
 7    foreach (Vertex v : G.vertices()):
 8      d[v] = +inf
 9      p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

	Adj. Matrix	Adj. List
Heap	O(n) + _____ + O(n^2) + _____	O(n) + _____ + O(m) + _____

A	B	C	D
0	5	2, A	∞

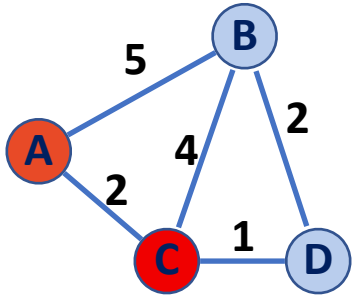


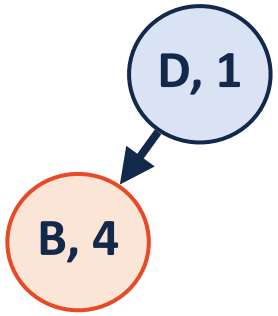



```
6   PrimMST(G, s):
7     foreach (Vertex v : G.vertices()):
8       d[v] = +inf
9       p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T           // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

	Adj. Matrix	Adj. List
Heap	O(n) + O(n log n) + O(n^2) + _____	O(n) + O(n log n) + O(m) + _____

A	B	C	D
0	4	2, A	1





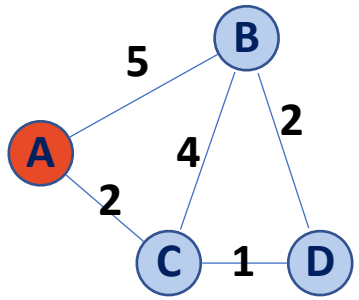

```
6   PrimMST(G, s):
7     foreach (Vertex v : G.vertices()):
8       d[v] = +inf
9       p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

1) Change minheap value

2) HeapifyUp()

	Adj. Matrix	Adj. List
Heap	O(n) + O(n log n) + O(n^2) + O(m log n)	O(n) + O(n log n) + O(m) + O(m log n)

(A, 0)
(D, ∞)
(C, 2)
(B, 5)



What is the size of array?

```
 6  PrimMST(G, s):
 7    foreach (Vertex v : G.vertices()):
 8      d[v] = +inf
 9      p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array		

Prim's Algorithm

Sparse Graph:

Dense Graph:





```
 6  PrimMST(G, s):
 7    foreach (Vertex v : G.vertices()):
 8      d[v] = +inf
 9      p[v] = NULL
10    d[s] = 0
11
12    PriorityQueue Q // min distance, defined by d[v]
13    Q.buildHeap(G.vertices())
14    Graph T          // "labeled set"
15
16    repeat n times:
17      Vertex m = Q.removeMin()
18      T.add(m)
19      foreach (Vertex v : neighbors of m not in T):
20        if cost(v, m) < d[v]:
21          d[v] = cost(v, m)
22          p[v] = m
23
```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array	$O(n^2)$	$O(n^2)$

MST Algorithm Runtime:

Kruskal's Algorithm:
O(n + m log (n))

Prim's Algorithm:
O(n log(n) + m log (n))

Sparse Graph:

Dense Graph:

Suppose I have a new heap:

	Binary Heap	Fibonacci Heap
Remove Min	O(lg(n))	O(lg(n))
Decrease Key	O(lg(n))	O(1)*

What's Prim's updated running time?

```
    PrimMST(G, s):
 6    foreach (Vertex v : G.vertices()):
 7      d[v] = +inf
 8      p[v] = NULL
 9    d[s] = 0
10
11    PriorityQueue Q // min distance, defined by d[v]
12    Q.buildHeap(G.vertices())
13    Graph T          // "labeled set"
14
15    repeat n times:
16      Vertex m = Q.removeMin()
17      T.add(m)
18      foreach (Vertex v : neighbors of m not in T):
19        if cost(v, m) < d[v]:
20          d[v] = cost(v, m)
21          p[v] = m
```

Shortest Path

