1/23/2018

1) Disjoint Set
2) Minimum Spanning Tree (Kruskal's / Prim's)
3) Experimental Run Time Analysis

{1} {2} {3} {4} ... {n}

find(v) , union(v1, v2)

v: 1 # (1-n)

finds the "leader" of the group(set) that contains v.

if v1 and v2 are already in the same set, false is returned no action taken. The 2 sets of v1 and v2 are merged into one + one of the 2 leaders is chosen as the leader of the merged sets, and true is returned.

We can make find, union run really fast. O(lg n) (initial)
If you use path compression close to O(1).

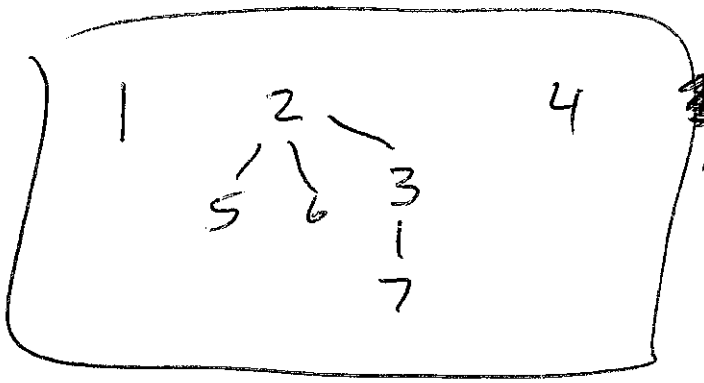
	Set View	Tree View	Array/ Code View
	{1} {2} {3} {4} {5} {6} {7}	1 2 3 4 5 6 7	1 2 3 4 5 6 7 1 2 3 4 5 6 7
Union (2,6)	{1} {2,6} {3} {4} {5} {7}	1 2 3 4 5 7 6	1 2 3 4 5 2 7 1 2 3 4 5 6 7
Union (3,7)	{1} {2,6} {3,7} {4} {5}	1 2 3 4 5 6 7	1 2 3 4 5 2 3
Union (5,6)	{1} {2,5,6} {3,7} {4}	1 2 3 4 5 6 7	1 2 3 4 2 2 3 1 2 3 4 5 6 7
Union (6,3)	{1} {2,3,5,6,7} {4}	1 2 4 5 6 3 7	1 2 2 4 2 2 3 1 2 3 4 5 6 7
Union (1,4)	{1,4} {2,3,5,6,7}	1 2 4 5 6 3 7	1 2 2 1 2 2 3 1 2 3 4 5 6 7
Union (4,7)	2 1 5 6 3 4 7 {1,2,3,4,5,6,7}		2 2 2 1 2 2 3 1 2 3 4 5 6 7

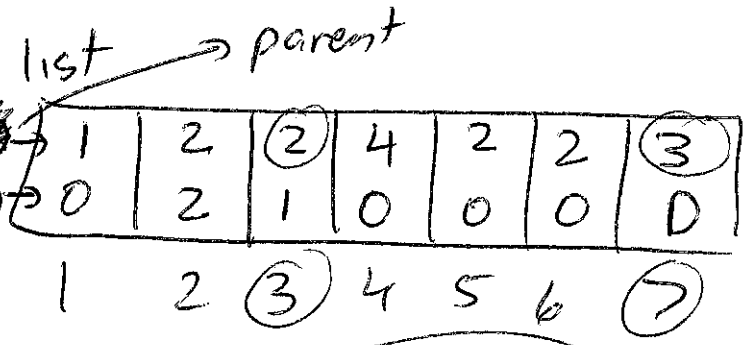
1/23/18 (2)

Code online sketch

pair ID (which # in DJ)
height (how far from the bottom of the tree)

Array of pair





// O(height) tree

```
int find(int v) {
    if (list[v].parent == v)
        return v;
    int res = find(list[v].parent);
    return res;
}
```

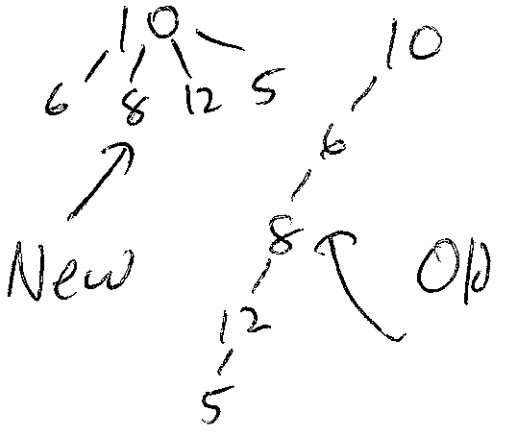
PATH COMP: `list[v].parent = res;`

```
boolean union(int v1, int v2) {
    int a = find(v1);
    int b = find(v2);
    if (a == b) return false;
    if (list[a].height > list[b].height)
        list[b].parent = a;
    else if (list[b].height > list[a].height)
        list[a].parent = b;
```

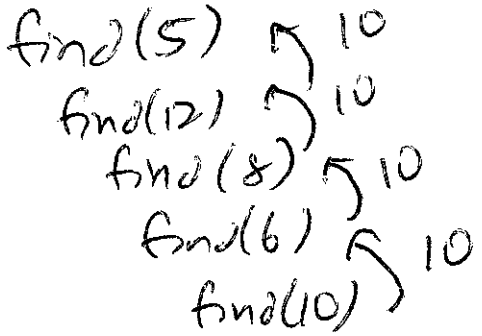
```
else {
    list[b].parent = a;
    list[a].height++; // tree of a
}                     "grows"
```

Path Compression

Wouldn't it be nice if we didn't keep track of heights but still had short trees?



Consider



Implementation IS EASY!

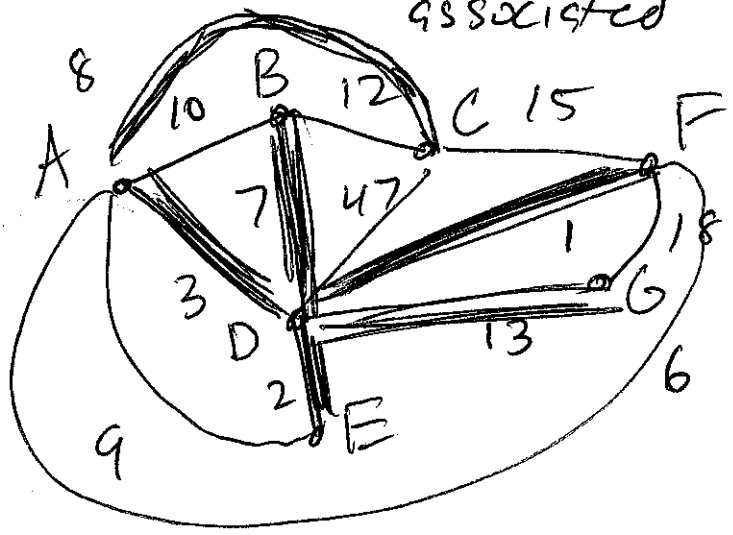
Kruskal's Algorithm to find a minimum spanning tree

sum of cost edges is minimized — covers whole graph — set connected vertices with no cycle.

Graph is a set of vertices (locations) connected by edges (roads).

✓ Undirected - roads are 2 way

✓ Weighted - each road has a "cost" associated with it.



edge 1 ✓
edge 2 ✓
edge 3 ✓
edge 6 ✗ (causes cycle)
edge 7 ✓
edge 8 ✓
edge 9 ✗ (causes cycle)
~~edge 12 ✗~~
edge 10 ✗ (causes cycle)
edge 12 ✗ (=)
edge 13 ✓
DONE

$S = \emptyset$ (set of edge in MST)

int i=0 sort (edge)

while (|S| < n-1) { (n = # vertices)

→ add edge [i] to S if it doesn't cause a cycle

i++;

}

How do I detect a cycle 1/23/18 (6)

Disjoint Set

When you add an edge btw V1, V2
Do djset.union(V1, V2).
If this returns false, V1, V2 were already in the same connected component.