Kruskal’s MST Algorithm:

Kruskal’s algorithm takes a different approach from Prim’s algorithm. Where Prim builds the MST by choosing edges that always form a connected tree and expanding it until it is a spanning tree, Kruskal chooses edges from different parts of the graph and eventually joins them together to create the MST.

```python
def Kruskal(G):
    chosen_edges = {}

    while |chosen_edges| < n-1:
        Let (u,v) be a minimum-weight edge in G
        such that chosen_edges + (u,v) contains no cycle
        chosen_edges += (u,v)
```

It is not difficult to prove that Kruskal’s algorithm chooses edges that form an MST, but as with Prim’s Algorithm we will defer the proof until CISC-365.

For choosing the next edge, there are two obvious methods. We could sort the entire list of edges in $O(m\times \log n)$ time, then test the edges in order. Since we might need to examine every edge in the sorted list before we find an MST, the while loop may execute $m$ times. On each iteration, the difficult task is testing the candidate edge to see if it can be added to chosen_edges without creating a cycle. If we use "test_time" to represent how long all this testing takes, then the time for the while loop is simply $O(test_time)$. Thus the whole algorithm runs in $O(m\times \log n + test_time)$.
As an alternative we could place all the edges in a min-heap (since we don't necessarily need to sort the whole list of edges - if we are lucky, the MST will be found quickly and most edges will never be considered). We can build the heap in $O(m)$ time, which is less than the $O(m \times \log n)$ required to sort the whole set of edges. The heap would have height in $O(\log m)$ which is the same as $O(\log n)$. Since we might need to access the heap $O(m)$ times, this also works out to $O(m \times \log n)$ to access the edges in the appropriate order and fix the heap after each access. Once again, we need to test each candidate edge to see if it creates a cycle, so again we get $O(\text{test_time})$ for this, and the whole algorithm runs in $O(m \times \log n + \text{test_time})$ ... the same complexity as sorting the set of edges. The actual running time of this heap-based version may be less than the “sort-all-the-edges” version, particularly if the MST is found without looking at too many edges.

Now we need to consider that "test_time". Suppose the edge being considered is $(u, v)$. We need a fast method to check to see if $u$ and $v$ are already connected by some of the edges already chosen (in which case we cannot use this edge). If $u$ and $v$ are not already connected then we choose the edge $(u, v)$ and add it to the chosen_edges set. But then we need to update the “connected-to” information: we need a method to record that all the vertices that were already connected to $u$ are now also connected to all the vertices that were already connected to $v$, and vice versa.

We will look at two methods (there are others). Both are based on the idea of identifying groups of vertices that are currently joined by edges or paths in the set $A$ of chosen edges.

**Solution 1:**

We will identify each group of connected vertices by an unique integer, stored in an array $P[1..n]$: $P[x]$ holds the group_id for vertex $x$. When we want to check an edge $(u, v)$ to see if it creates a cycle, we just need to compare the group_id of $u$ to the group_id of $v$. This takes constant time. If they are the same, $u$ and $v$ are already connected and we cannot use the edge. If they are different, the edge is safe. However, adding the edge $(u, v)$ to $A$ means that all the vertices in the two groups are now connected. We need to find all the vertices $x$ such that $P[x] = P[u]$, and change them all to $P[v]$

In pseudo-code it looks something like this:
def Kruskal(G):
    chosen_edges = {}
    for i = 1 to n:
        P[i] = i  # each vertex is in its own group
    while |chosen_edges| < n-1:
        Let (u,v) be a minimum-weight edge in G such that P[u] != P[v]
        chosen_edges += (u,v)
        temp = P[u]
        for i = 1 to n:  # update the P values
            if P[i] == temp:
                P[i] = P[v]

The for loop that updates the P values is obviously in O(n). It executes every time we choose an edge, which happens n-1 times. Thus the total time involved in testing and fixing the P values is O(n^2)

Using this method, Kruskal’s MST algorithm runs in O(m * \log n + n^2)

Solution 2:
Instead of an array that stores the group_id for each vertex, we will use special trees which we will call **disjoint-set trees** to store this information. A **disjoint-set tree** consists of a number of vertices that represent set elements, with one designated as the root vertex. All edges in the set union tree point upwards (from the leaves to their parents, from those vertices to their parents, and so on up to the root).

It is important to note that these set union trees are not formed from edges of G - they simply represent the groups of vertices that are connected by the edges that we choose during Kruskal’s algorithm.

At the start of the algorithm, each vertex is the sole leaf (and also the root!) of a separate **disjoint-set tree**. As the algorithm progresses, the **disjoint-set trees** will be combined. To test an edge (u,v) to see if it can be used, we trace upwards in u’s **disjoint-set tree** to the root, and then trace upwards in v’s **disjoint-set tree** to the root. We will see that this can be done in O(log n) time. If the two roots are the same, then u and v are in the same **disjoint-set tree** -
which means they are already connected by edges in chosen_edges, so the edge \((u,v)\) cannot be added. If the two roots are not the same, then the edge is added to chosen_edges and the two disjoint-set tree must be combined. We will see that this can be done in constant time.

In pseudo-code the whole algorithm looks something like this:

```python
def Kruskal(G):
    chosen_edges = {}
    for i = 1 to n:
        P[i] = new R(i)
        # R(i) is a disjoint_set_tree_vertex object - it contains
        # an attribute id which is initialized to i, and an
        # attribute parent which is initialized to None, and an
        # attribute rank which is initialized to 1
    while |chosen_edges| < n-1:
        let (u,v) be the next candidate edge
        trace up from P[u] until we find a disjoint_set_tree_vertex object with no parent. This is the root of \(u\)’s disjoint set tree. Call this \(R_u\)
        trace up from P[v] until we find a disjoint_set_tree_vertex object with no parent. This is the root of \(v\)’s disjoint set tree. Call this \(R_v\)
        if \(R_u \neq R_v\):
            chosen_edges += {(u,v)}  # add the edge
            # combine the disjoint set trees
            if (Ru.rank >= Rv.rank) :
                Rv.parent = Ru
                Ru.rank = max(Ru.rank, Rv.rank+1)
            else
                Ru.parent = Rv
                Rv.rank = max(Rv.rank, Ru.rank+1)
```

As promised, combining the two sets into one takes constant time. However, we need to think carefully about which way we do the combination. Clearly we could do either "Ru.parent = Rv" or "Rv.parent = Ru" to combine the two sets. We want to minimize the height of the combined tree, since this will keep the "trace up" steps as efficient as possible. If the two disjoint set trees being combined have the same rank, it doesn't matter which one
becomes the parent of the other. But if (for example) Ru.rank is greater than Rv.rank, then making Ru the parent of Rv creates a combined disjoint set tree in which the maximum trace up time is less than if we make Rv the parent of Ru.

With some careful counting that I'm not going to go into here, we can show that if we always make the root of the larger set union tree the parent of the root of the smaller set union tree, then the set union trees will always have height in \( O(\log n) \). This means the trace time is in \( O(\log n) \).

But we can do even better! Every time we search for the root of the disjoint set tree for a vertex \( u \), we can make \( u \) and all the vertices “above it” in the disjoint set tree point directly to the root. The algorithm for this looks like

\[
\text{find_root}(x):
\begin{align*}
\text{if } P[x] & = \text{null}: \\
& \quad \text{return } x \\
\text{else}: \\
& \quad P[x] = \text{find_root}(P[x]) \\
& \quad \text{return } P[x]
\end{align*}
\]

This makes the disjoint set trees very short – most elements in each disjoint set tree point directly to the root of the tree, which means the trace time is effectively constant!

Since we may have to do the tracing up operation for every edge of the graph, test_time is effectively in \( O(m) \).

Thus, thanks to the choice of a good data structure, Kruskal’s MST algorithm can be implemented in effectively \( O(m \times \log n + m) \) time, which reduces to effectively \( O(m \times \log n) \) time.

Note that if \( m \) is in \( O(n) \) (ie G is sparse) then this algorithm runs in effectively \( O(n \times \log n) \) time. However if \( m \) is in \( \Omega(n^2) \) (ie G is dense) then this algorithm runs in effectively \( O(n^2 \times \log n) \) time.

(A note about this effective constant running time for finding the roots of the disjoint set trees:
it’s not quite constant. Its actual growth rate is given by the inverse Ackermann function \( \alpha(n) \), which grows very very very slowly … so slowly that \( \alpha(n) \leq 5 \quad \forall n \leq 2^{2^{2^{16}}} \). Thus for any plausible graph size, finding the roots of the disjoint set trees will take constant time.)

**Prim vs. Kruskal**

Now that we have seen two MST algorithms, it is natural to ask which is better. The answer of course is that it depends on the situation. Some authors suggest that for dense graphs Prim is faster (recall the \( O(n^2) \) implementation) but that for "typical graphs" - whatever that means - Kruskal is faster. I have not seen any convincing evidence to support this claim but if "typical" means "sparse" then it makes sense.

**Last Words**

There are other, more sophisticated data structures that reduce the complexity of finding an MST (particularly Prim’s Algorithm) even further. I encourage you to research them.