**From abstract pseudocode to less-abstract pseudocode:**

Kruskal’s, Prim’s, and Borůvka’s algorithms

In the following examples, we will see that different types of pseudocode meet different needs. Abstract pseudocode should be very easy to understand, but it might be hard to see how long the resulting algorithm will take. In fact, the analysis of the run-time will likely depend on the data structures that are used. In less-abstract pseudocode, it should be clear exactly what data structures are to be used (beyond simple arrays) in an implementation. As a result we should be able to analyze the running time.

The input to an MST algorithm is a graph \( G = (V, E) \) that has non-negative weights \( w \) on the edges. We will assume that no two edges have the same weight. As we saw in class, we know that:

Suppose \( T = (V, E_T) \) is the minimum spanning tree. Let \( F = (V, E_F) \) be a subgraph of \( T \). Define:

- **useless edge** both endpoints in same component of \( F \)
- **safe edge** min weight edge with exactly one endpoint in a given connected component of \( F \)

Then \( T \) contains every safe edge and no useless edge.

This implies that the following very abstract algorithm is correct:

\[
\text{MST-Very-Abstract}(G = (V, E), w) \\
1 \text{ initialize } F = (V, \emptyset) \\
2 \text{ while there is a safe edge with respect to } F: \\
3 \quad \text{ add one or more safe edges with respect to } F. \\
4 \text{ return } F
\]

However, it is not clear what the running time of this algorithm is. It is not even clear how to find the safe edges.
Kruskal’s minimum spanning tree

Kruskal’s algorithm builds a minimum spanning tree (MST) by adding edges to a set in order of increasing length as long as doing so does not introduce a cycle: as long as the endpoints of the considered edge are in different components of the tree built so far.

We use a data structure for disjoint sets (the union-find data structure, below). Each set will correspond to a component in the tree built so far. Since we must consider the edges in increasing order of weight, we may as well sort the edges to start.

\[
\text{Kruskal-Abstract}(G = (V, E), w)
\]

1. initialize \( F = (V, \emptyset) \)
2. for edges \( uv \) in increasing order of length \( w \)
3. \hspace{1em} if \( u \) and \( v \) are in different components of \( F \)
4. \hspace{2em} add \( uv \) to \( F \)
5. return \( F \)

Each edge is considered once. For each edge, we need to determine if its endpoints are in the same component of \( F \) or not. It should be clear that every edge added is a safe edge, and so this correctly finds the MST of \( G \).

\[
\text{Kruskal}(G = (V, E), w)
\]

1. disjoint sets \( D = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\} \)
2. initialize \( F = (V, \emptyset) \)
3. sort \( E \) by increasing \( w \)
4. for each edge \( uv \) in this order
5. \hspace{1em} if find\( (D, u) \neq \text{find}(D, v) \)
6. \hspace{2em} \( F = F \cup \{uv\} \)
7. \hspace{2em} \text{union}(D, u, v) \)
8. return \( F \)

Sorting takes \( O(m \log m) \) time. There are \( 2m \) calls to find and \( n - 1 \) calls to union for a total of \( O((m + n) \log n) \) time. Can you see why this is also \( O((m + n) \log n) \) time?

The union-find data structure maintains a set of disjoint sets and performs union and find operations. find\( (D, u) \) returns which set (in the data structure \( D \)) \( u \) is in and union\( (D, u, v) \) merges the sets in \( D \) that \( u \) and \( v \) are in. Both operations take \( O(\log n) \) time for sets over \( n \) elements.
Borůvka’s minimum spanning tree

Borůvka’s minimum spanning tree algorithm simply adds all the safe edges at once, so it should be clear that it correctly finds the MST.

\[
\text{Borůvka-abstract}(G = (V, E), w)
\]

\begin{verbatim}
1     initialize \( F = (V, \emptyset) \)
2     while there are safe edges:
3         add all the safe edges to \( F \)
4         delete all the useless edges from \( G \)
5     return \( F \)
\end{verbatim}

\[
\text{Borůvka}(G = (V, E), w)
\]

global edge list

\[
\text{Borůvka-R}(G, w)
\]

return all marked edges

\[
\text{Borůvka-R}(G, w)
\]

if \(|V| \neq 0\)

for each vertex \( v \):
    mark the lowest weight edge adjacent to \( v \)
create \( G' \) by:
    contracting the marked edges
deleting the self-loops
\[
\text{Borůvka-R}(G', w)
\] % this graph is smaller
Primer’s minimum spanning tree

Primer’s minimum spanning tree algorithm grows a tree as a single connected component in a manner similar to Dijkstra’s algorithm. Since $uv$ is a safe edge, the algorithm is correct.

Can you write the pseudocode for Primer that makes clear what the running time is using a priority queue data structure? What is the running time of the algorithm?

**Primer-abstract**($G = (V, E), w$)

1. pick an arbitrary vertex $s$
2. initialize $F = (V, \emptyset)$
3. while the component of $F$ containing $s$ does not span $V$:
   4. let $uv$ be the cheapest edge with one endpoint in the component of $F$ that contains $s$
   5. add $uv$ to $F$
4. return $F$