6.7.1 Implemetations of Prim's Algorithm

The following is a straightforward implementation of Prim's algorithm.

```
Input: Connected undirected weighted graph G as adjacency list
Output: the edges of a minimum spanning tree of G

Init:
  X <- {s}  // s is an arbitrarily chosen vertex.
  T <- {}   // chosen edges; initially empty.

Main loop:
  for n-1 iterations
    Among edges (u,v) with u in X and v not in X
      Take (u*,v*) with minimum weight
      Add v* into X
      Add (u*,v*) into T
  return T
```

What is the running time of this implementation? Answer: $O(nm)$. At each iteration, we need to do the validity check for all the edges, that is whether, for (u, v), $u \in X$ and $v \notin X$.

우선 큐

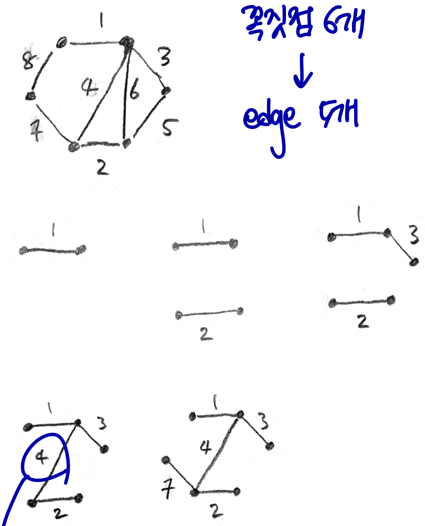
Improvement using "heap-based priority queue" We can improve using a heap-based priority queue. Together with the usual `pop()` and `push()`, we use `update key` operation that runs, again, in $O(\log n)$ time. [[Revise the code in the style of Dijkstra!!!]]

```
Init:
  X <- {s}; T <- {}
  Q: priority queue (of vertices with keys)

  for each v != s
    if there is edge (s,v)
       key(v) <- w(s,v);  e(v) <- (s,v)
    else
       key(v) <- infty;  e(v) <- NULL
    push(v)

Main loop: n-1번.
  while Q is non-empty
    v* <- pop()          // the closest vertex chosen.
    Add v* into X
    Add e(v*) into T
    // update keys
    for each edge (v*,y) with y not in X  // we're using adj list!
      if w(v*,y) < key(y)
        update key(y) <- w(v*,y); e(y) <- (v*,y)
  return T
```

The loop in the initialization (lines 5–10) performs n heap operations. Therefore, it takes $O(n \log n)$ time. The main loop has $n-1$ iterations, in which we have a heap operation and constant-time operations (lines 14–16) and a for loop (lines 18–20). Except for the for loop, it takes $O(n \log n)$ time. Now, the inner for loop itself is hard to estimate, but during the overall lifetime of the main loop, $O(m)$ edges are checked for key updates, and so, it takes $O(m \log n)$. Therefore, in total, the running time is $O((n+m)\log n)$.

6.7.2 Implementation of Kruskal's algorithm

While Prim's algorithm grows a single tree that eventually becomes the minimum spanning tree, Kruskal's algorithm maintains a subgraph that can have more than one component, which grows to become a minimum spanning tree. For example, see the following illustration:



꼭짓점 6개 → edge 5개

→ 이래해도 위쪽은 cycle 안생기니까.

Figure 13: An illustration of Kruskal's algorithm.

Edges are added in the order of weights, and note that the edge of weight 6 could not be added because it makes a cycle. Now, let us see the following straightforward implementation of Kruskal's algorithm:

```
Input: Connected undirected weighted graph G as adjacency list
Output: the edges of a minimum spanning tree of G

Preprocessing:
  S <- {}   // chosen edges; initially empty. S for subgraph :)

  Sort edges by weights 따라서 정렬 n개 → O(nlogn)

Main loop:
  for each edge e, in increasing order of weight
    if S and e does not make a cycle
```