Análisis básico de los costes de los algoritmos de Prim y Kruskal para el peor caso, según el planteo hecho en clase (queda como ejercicio calcular los costes de las funciones Set, Unión y Find de Kruskal con distintas estructuras, y plantear implementaciones alternativas para ambos algoritmos (por ejemplo, incorporando el uso de heaps o montículos de forma adecuada), evaluando su coste)

Algoritmo de Prim:

Enlace interesante: http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Prim1957.pdf

Tal como hemos propuesto, el esquema del algoritmo indica:

```
AristasR = {} // conjunto de aristas del árbol recubridor
VerticesR = {un nodo del Grafo G} // Conjunto de vértices del árbol recubridor
//recordar que se elige el primer nodo de G=(V, A) de forma arbitraria
mientras (VerticesR != V)
    {
    seleccionar arista mínima (u,v) tal que u ε VerticesR y v en V-VerticesR
    AristasR ::= AristasR U {(u,v)}
    VerticesR ::= VerticesR U {v}
    }
retornar G'= (VerticesR, AristasR)
```

Consideremos que implementamos el grafo con una con matriz de adyacencia y que (como propusimos en clase) usamos un arreglo dist_min para establecer cuál es la distancia mínima de cada nodo no incluido en el árbol que se está generando a alguno de los nodos del árbol, el cual se va actualizando en cada etapa. También hay que registrar a qué vértice – o al menos registrar uno de ellos, si hay varios con esa condición – la distancia es mínima. Eso lo hacernos con el array vert_mas_cerc.

Entonces resulta, detallando el esquema:

```
//Inicialización:
AristasR = {} // conjunto de aristas del árbol recubridor
VerticesR = {1} //considerando los vértices numerados 1..n y eligiendo el vértice 1 como arranque
                                                                                    } O(1)

Para i desde 2 hasta n ------------------------------------------------------------> O(n)
  {
   dist_min [i] = coste(1, i) //se toma de la lista de adyacencia del vértice1; si no hay conexión, va ∞ ó equivalente
   vert_mas_cerc [i] = 1                                                            } O(1)
  }

mientras (#(VerticesR) < #(V))  //esto equivale a mientras (VerticesR != V) ------> O(n) iteraciones
 {
   Elegir v / (u,v) arista con u ε VerticesR y v en V-VerticesR / dist_min [v]>0 y que minimice dist_min [v] ---> O(n)
   min = dist_min [v]  // coste O(n), se ejecuta n-1 veces
   AristasR ::= AristasR U {(u,v)} // coste O(n), se ejecuta n-1 veces
   VerticesR ::= VerticesR U {v} // coste O(n), se ejecuta n-1 veces
   dist_min [v]=-1  // coste O(n), se ejecuta n-1 veces                             } Cada sentencia tiene coste O(n)

   para j desde 2 hasta n ------------------------------------------------------> O(n)
    {
     Si coste (j, v) < dist_min [j]  // coste O(1), se ejecuta n-1 veces para cada iteración del ciclo externo
        {
         dist_min [j] = coste (j, v) // coste O(1), se ejecuta n-1 veces para cada iteración del ciclo externo

         vert_mas_cerc [j] = v // coste O(1), se ejecuta n-1 veces para cada iteración del ciclo externo
        }                                                                           } Cada sentencia tiene coste O(1)
    }
 }
```

Algoritmo de Kruskal

Enlace interesante: http://www.cmat.edu.uy/~marclan/TAG/Sellanes/Kruskal.pdf

El esquema del algoritmo indica:

```
Ordenar aristas de forma creciente
AristasR = {}  // conjunto de aristas del árbol recubrido
VerticesR = {} //inicializar  conjunto de vertices del arbor recubridor
Mientras # (VerticesR) < n-1
  {
    Elegir arista (u,v) de peso mínimo que no forme ciclo
   AristasR = AristasR U {(u,v)}
   VerticesR = VerticesR U { verticenuevo} //puede ser el vértice u , o el v
  }
```

Si la arista seleccionada es (u,v), recordar que habitualmente se utiliza la función Find para establecer si los vértices y v están o no en el mismo conjunto (es decir, en el mismo árbol parcial) y la función Union para realizar la unión de los conjuntos a que pertenecen los vértices u y v respectivamente. Inicialmente cada vértice es el único en su propio conjunto, lo cual realizamos a través de la function MakeSet.
El coste dependerá del coste de dichas funciones; de todos modos, analizaremos los costes de las etapas del algoritmo para el peor caso, considerando un ordenamiento rápido para las aristas del grafo.
Al cálculo habrá que agregarle los costes de Union y Find, los cuales dependen de las estructuras que se hayan usado para la implementación.
Proponemos un grafo con n vértices y a aristas

```
Ordenar A de forma creciente en lista L ------------------------> O(a * log a)
Para cada vértice v
  {
   MakeSet(v)                                  } O(n) * coste de MakeSet
  }

AristasR = {} ------------------------------------------------------> O(1)

mientras (#(AristasR) , n-1) ---------------------------------------> O(a) iteraciones
  {
    (u,v) = primera arista de L ---------------------------------------> O(1)
    Eliminar (u,v) de L -----------------------------------------------> O(1)

 Si (Find(u) !=  Find(v))
        {   Union (u,v) }                      } Depende de coste de Find y de Union

  }
```

¿Cuándo será más conveniente usar Prim y cuándo Kruskal? ¿Qué elementos deberán analizarse? Responder y justificar. Mostrar casos ejemplificadores