Respuestas

1a) Mantenemos un arreglo, indexado por los números de los vértices, con las distancias de cada vértice a s, todas inicialmente *infinito*, excepto la del propio s (que es 0). Cada vez que agregamos un vértice v a T, actualizamos, via la operación *reduce*, las distancias a s de los vecinos de v —cada actualización toma tiempo O(1) y a lo más hay $|V|-1$ actualizaciones— y luego buscamos el vértice w que quedó más cerca de s —hay que hacer $|V|-1$ comparaciones—; w hará las veces de v en la próxima iteración. Es decir, en cada iteración hacemos O(V) operaciones; y en total hacemos $|V|-1$ iteraciones.

1b) Basta con dar un (contra)ejemplo. Lo primero que hace el algoritmo es mirar a todos los vecinos de s y agregar a T el más cercano de estos. Supongamos que los vecinos son dos, u y v, y que los costos de las aristas son $w(s, u) = 2$ y $w(s, v) = 3$. Entonces el algoritmo agrega u a T, con distancia 2. Sin embargo, si hay una arista (u, v) con costo $w(u, v) = -1$, significa que la distancia de s a v es realmente 1 (= 2 + –1, yendo a través de u) y no 3 (yendo directo), y por lo tanto v está más cerca de s que u y debería haber sido agregado a T antes que u. O bien, si hay una arista (v, u) con costo $w(v, u) = -2$, entonces la distancia de s a u es 1 (= 3 + –2, yendo a través de v) y no 2 (yendo directo).

2a) Primero, ordenar las aristas de $G \rightarrow E \log E$; luego, aplicar DFS en un grafo con a lo más $|V|$ aristas (el número de aristas que finalmente va a tener el árbol T) $\rightarrow |V|$; como esto último hay que repetirlo en las $|E|$ iteraciones $\rightarrow |E|\,|V|$; así, en total $\rightarrow E \log E + EV = O(EV)$.

2b) Las estructuras que representan los conjuntos disjuntos (ya sea arreglos, listas ligadas o árboles con raíz) son las mismas componentes —es decir, tienen los mismos vértices— que los árboles que constituyen las componentes conectadas de T.

2c) *Propuesto*.

3) Sea A un subconjunto de E, tal que las aristas de A forman parte de algún MST de G; y sea $(S, V - S)$ un corte de G, tal que ninguna arista de A cruza el corte (es decir, todas las aristas de A están al mismo lado del corte). Sea (u, v) una arista que cruza el corte y que tiene el menor costo entre todas las aristas que cruzan el corte. Queremos demostrar que podemos agregar (u, v) a A, de modo que A siga formando parte de algún MST de G; es decir, (u, v) es la próxima arista que agregamos al MST de G que estamos construyendo.

Sea T un MST de G y supongamos que T no incluye a (u, v): (u, v) forma un ciclo con las aristas que están en el camino (simple) de u a v en T. Ya que (u, v) cruza el corte, u y v están en lados opuestos del corte, y por lo tanto al menos una arista de T de las que están en el camino de u a v también cruza el corte: sea (x, y) tal arista, la cual no está en A, ya que todas las aristas de A están al mismo lado del corte. Como el camino de u a v en T es único, si sacamos (x, y) de T, partimos T en dos componentes. Si ahora agregamos (u, v), entonces reconectamos las dos componentes formando un nuevo árbol de cobertura, T'.

Ahora falta demostrar que T' es un MST. Como (u, v) es una arista de menor costo entre las aristas que cruzan el corte, y como (x, y) también cruza el corte, entonces $w(u, v) \leq w(x, y)$. Por lo tanto, costo(T') = costo(T) – $w(x, y)$ + $w(u, v) \leq$ costo(T). Como partimos del hecho que T es un MST, costo(T) $\leq$ costo (T'). Por lo tanto, T' debe ser también un MST.

4a)

implementación	*union*	*find*
arreglo simple	**O(*n*)**	**O(1)**
lista ligada lineal	**O(1)**	**O(*n*)**
árboles con raíz (representados mediante arreglos)	**O(1)**	**O(*n*)**
árboles con raíz, en que el árbol más pequeño apunta al más grande	**O(1)**	**O(log*n*)**

4b) Cualquiera de las estructuras de datos para conjuntos disjuntos sirve para resolver el problema; lo importante es entender bien lo que representan los conjuntos y el rol de las operaciones *find* y *union*. Inicialmente, dado que no hay dos celdas conectadas entre ellas, cada celda está en un conjunto por sí misma. Entonces, repetidamente, elegimos una muralla de manera aleatoria —representada por las dos celdas adyacentes a la muralla— y ejecutamos dos operaciones *find* para determinar a qué conjunto pertenece cada celda. Si las celdas pertenecen al mismo conjunto, significa que ya están conectadas entre ellas y por lo tanto no conviene botar esa muralla. Por el contrario, si las celdas pertenecen a conjuntos distintos, significa que no están conectadas entre ellas, por lo que botamos la muralla, es decir, realizamos una operación *union* entre los conjuntos. Podemos detener el algoritmo cuando la celda de la entrada y la celda de la salida queden en el mismo conjunto, o, mejor, cuando solo quede un conjunto (todas las celdas están conectadas entre ellas).

5) Después del primer paso:

	¿sol?	*dist*	*padre*
a	***T***	0	–
b	*F*	**2**	***a***
c	*F*	**4**	***a***
d	*F*	**1**	***a***
e	*F*	∞	–
f	*F*	∞	–
g	*F*	∞	–

…

Después del último paso:

	¿sol?	*dist*	*padre*
a	***T***	0	–
b	***T***	**2**	***a***
c	***T***	**2**	***d***
d	***T***	**1**	***a***
e	***T***	**6**	***g***
f	***T***	**1**	***g***
g	***T***	**4**	***d***