Estructuras de datos y algoritmos I-2012

Interrogación n° 3

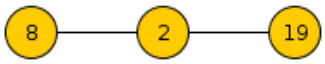
Instrucciones generales

- Ingresa tu nombre en todas las hojas de respuesta
- Entrega 1 hoja por pregunta, independiente si no respondiste la pregunta
- Lee cuidadosamente cada pregunta
- Dispones de 120 minutos

Pregunta 1 – Grafos (20 pts)

a) (8 pts) Escribe una función que permita ir de un vértice A a un vértice B pasando por el menor número de vértices como intermediarios. Tu función debe recibir como parámetros el grafo, el vértice A y el vértice B – de acuerdo a lo visto en clases, puedes elegir como deseas recibir el grafo y los vértices. Al principio, tu función debe imprimir “Comenzando en el vértice i” donde i es el identificador del vértice A. Luego, cada vez que tu función visite un vértice camino al vértice B, imprimir el identificador del vértice que está visitando. Finalmente, cuando llega al vértice B, imprimir “Llegando al vértice j”, donde j es el identificador del vértice B.

Por ejemplo, en el grafo:

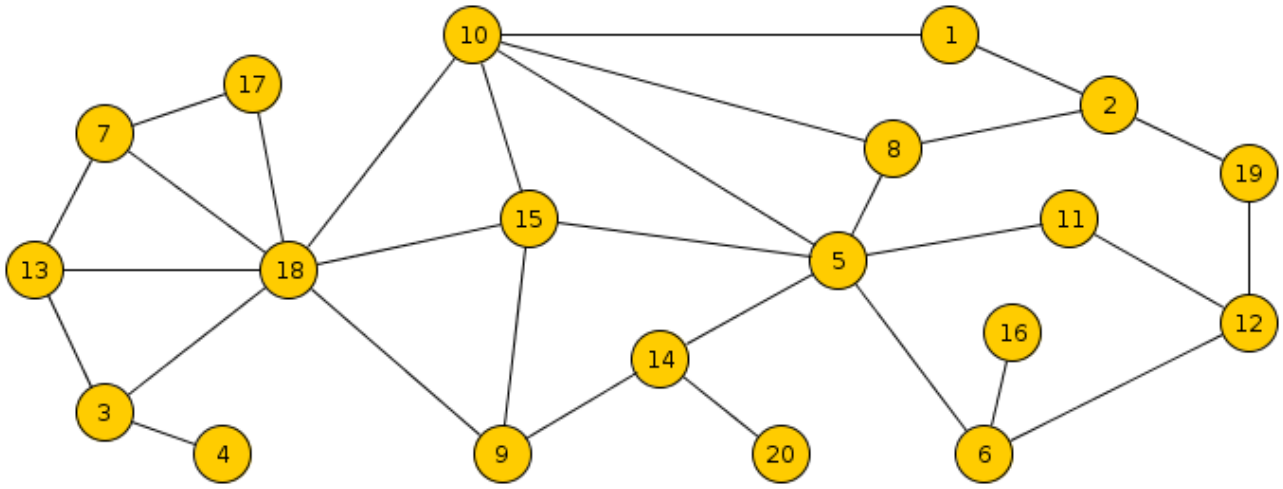


La función, al ir del vértice 8 al vértice 19 debiera imprimir:

```
Comenzando en el vértice 8
2
Llegando al vértice 19
```

Este ejercicio se puede resolver aplicando BFS para encontrar el camino más corto entre A y B. Sin embargo, esto tiene una complicación adicional y es que no basta con recorrer el grafo usando BFS, sino que además hay que guardar el camino (lo que se puede conseguir guardando para cada vértice el vértice padre del árbol resultante). Si se utiliza una matriz de adyacencia, y se considera que un vértice i NO está conectado al vértice j si y sólo si matriz[i][j] == 0 (o, dicho de otro modo, matriz[i][j] != 0 significa que hay una arista ij), es posible guardar en la misma matriz los ids de los vértices padres del camino BFS, cambiando el valor [i][j] (que debiera ser 1) por -1 si en el recorrido BFS se llegó al vértice j desde i. Luego, para imprimir el camino es cuestión de comenzar del vértice A y seguir al vértice j tal que matriz[A][j] == -1, y así hasta llegar a matriz[k][B] con misma condición.

b) (6 pts) Escribe un algoritmo que recorra en profundidad (DFS) un grafo. Luego, considera el siguiente grafo y utilizando tu algoritmo, recórrelo desde el vértice 1. Escribe la secuencia de nodos que visitas.



Este ejercicio consiste en efectuar un DFS clásico y aplicarlo al grafo. Sólo para cambiar un poco, voy a usar una lista de adyacencia, representada como una lista de listas.

(dado que podía ser en seudo código, me voy a dar el gusto de escribir mi solución en Java :D)

```
void dfs(List<List<Integer>> graph, int source) {
        boolean[] visited = new boolean[graph.size()];
        // en Java los boolean se inicializan en false por defecto
        rdfs(graph, visited, source);
}
void rdfs(List<List<Integer>> graph, boolean[] visited, int source) {
        // he visitado source!
        visited[source] = true;
        List<Integer> adjVertices = graph.get(source);
        for(int adj : adjVertices) {
                // adj es un vertice adyacente
                if (!visited[adj]) {
                        // aún no he visitado adj. A por ello.
                        rdfs(graph, visited, adj);
                }
                // si hubo algun vertice adyacente que falto por visitar, el for se encargará.
        }
}
```

Ahora, la ejecución de mi dfs dependerá del orden en que estén los vértices en la lista. Supondré que la lista de nodos adyacentes siempre está ordenada del nodo menor al mayor.

Así, la ejecución será:

1 2 8 5 6 12 11 19 16 10 15 9 14 20 18 3 4 13 7 17

c) (6 pts) Imagina que debes escribir una función simple, que reciba un grafo, dos vértices y retorne true o false si están directamente conectados (si hay una arista entre los vértices). Sin embargo, considera que el grafo tendrá entre 50.000 y 100.000 vértices y cada vértice tentrá como máximo un grado de 4. ¿Cómo representarías el grafo para resolver este problema y por qué? También debes escribir la función.

Acá lo importante que debes notar es que:

- *No puedes usar una matriz de adyacencia porque, si suponemos que un vértice ocupa n bytes, la matriz ocuparía entre 2.500.000.000n y 10.000.000.000n bytes (si suponemos que un vértice es un int de 32 bits, es decir, ocupa 4 bytes, entonces la matriz usa entre 10GB y 40GB de memoria)*
- *Si hay hasta 100.000 vértices de hasta grado 4, entonces habrá hasta 200.000 aristas, por lo que una matriz de incidencia tendría hasta 100.000 x 200.000 datos, lo que es aún menos viable que la matriz de adyacencia.*

Por lo tanto, por el número de vértices hay que usar o bien una lista de incidencia (que tendrá hasta 200.000 x 2 datos, es decir, hasta 400.000 datos, lo cual es un número bien manejable: 1.6 MB de memoria con int de 32 bits), o bien una lista de adyacencia (que tendrá hasta 100.000 x 4 datos, es decir, también hasta 400.000 datos).

Dado que se desea saber si dos vértices están conectados, si usamos la lista de incidencia cada vez habría que buscar entre todas las aristas, mientras que con la lista de adyacencia la lista será, en el peor de los casos, buscar entre 4 elementos. Por lo tanto representaría el grafo con una lista de adyacencias.

```
boolean isConnected(int[][] adjList, int i, int j) {
        for (int k = 0; k < adjList[i].length; k++) {
                if (adjList[i][k]  == j) return true;
        }
        return false;
}
```

Pregunta 2 – Técnicas de programación (20 pts)

A continuación se presentan 4 problemas, cada uno con una solución propuesta. Para cada caso, escribe qué tipo de solución es y por qué, entre: Dividir y conquistar (DyC), Algoritmo codicioso, Programación Dinámica o ninguno de los anteriores. Nota que las soluciones propuestas podrían ser incorrectas, pero este aspecto no es relevante para la evaluación que debes hacer.

a) (5 pts) El gato enojado >(

Tienes un gatito que está durmiendo plácidamente en medio de la sala. Tu quieres cruzar la sala, pasando frente al gatito. Pero ¡cuidado! Si despiertas al gato, este estará muy enojado, así que debes ser muy silencioso. Para mala suerte, el piso de la sala rechina y cada vez que das un paso el gato despierta ligeramente. Pero si esperas un poco, el gato vuelve a dormir profundamente.

Para ser exactos, para cruzar la sala necesitas dar a pasos ($0 < a < 100$). El gato duerme inicialmente con profundidad 1 y si en algún momento la profundidad de sueño es inferior a w ($0 < w < 1$), el gato despertará. Dependiendo de donde estés, cada vez que das un paso, la profundidad del sueño decae en s_i ($0 < s_i < 1$), para s_0, s_1, … s_{a-1}. Finalmente, por cada tiempo k ($0 < k < 10$) que esperes, la profundidad del sueño subirá en 0.1 .

¿Cuál es el menor tiempo de espera en que puedes cruzar la sala sin despertar al gatito?

El input consiste en una primera línea con los valores a y w. Luego una línea con los a valores s (s_0, s_1, … s_{a-1}). Por último hay una línea con el valor k. Los valores a y k son números enteros, mientras los demás números son decimales con 1 dígito de precisión.

El output consiste en un número entero con el menor tiempo de espera o -1 si no es posible cruzar la sala sin despertar al gato.

Ejemplo:

Input

```
10 0.3
0.2 0.4 0.3 0.5 0.6 0.4 0.3 0.1 0.2 0.1
1
```

Output

```
24
```

Solución propuesta

En una variable vamos a mantener el nivel de sueño del gato (p). Luego, para cada paso entre 0 y a -1 primero evaluamos si con el nivel de sueño actual podemos dar el paso sin despertar al gato, damos el paso y actualizamos el nivel de sueño como $p' = p - s_i$. La evaluación de si podemos dar el gato en el fondo es evaluar si $p - s_i >= w$. Si no es posible, entonces esperaremos el mínimo posible para poder dar el paso, es decir j*k unidades de

tiempo con el menor j tal que p + (0.1*j*k) – s_i >= w. Esta condición de mínimo se cumple en la igualdad, es decir, esperaremos j*k = (w + s_i – p)/0.1. Lo que si, antes es necesario un chequeo adicional: si para poder dar el paso hubiera que hacer que la profundidad de sueño sea superior a 1 (es decir, si p + (0.1*j*k) > 1), entonces no podemos dar el paso y la respuesta será un -1. Si logramos dar el paso, agregar j*k a la suma acumulada de tiempos esperados.

Finalmente, si logramos llegar hasta el final, entregar la suma acumulada de tiempos esperados.

Este algoritmo es greedy que supone que encontrará el óptimo al tratar siempre de esperar lo menos posible, justo lo suficiente para dar el siguiente paso. Si uno compara con el método general, considera que la solución al problema consiste en el conjunto de tiempos de espera, la selección de cada turno es el tiempo mínimo de espera, la decisión de factibilidad es ver si no se supera el nivel de sueño "1" para poder dar el paso y agregarlo a la solución es la suma (este último análisis de comparación con el método general no es necesario, pero da una evidencia irrefutable de que es greedy).

b) (5 pts) Elegir las frutas más pesadas del canasto*

Se tiene un canasto de m frutas con n tipos distintos (m > n). Calcula la suma de los pesos de las frutas, al elegir una fruta de cada tipo (siempre la más pesada de su tipo).

El input consiste en n (tipos de frutas), luego m (total de frutas), luego una lista de m pesos de las frutas y finalmente una lista con los m tipos de cada fruta (números entre 0 y n-1).

Ejemplo.

Input:

```
3 5
1 8 5 20 2
0 1 0 1 2
```

Output:

```
27
```

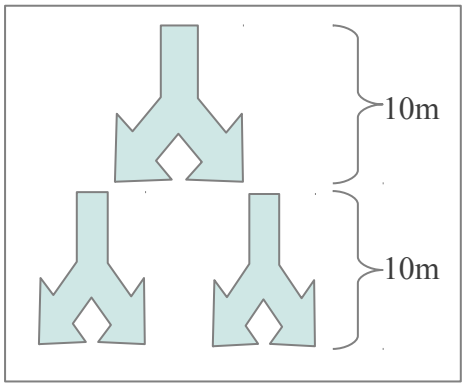
Solución propuesta

Se divide la lista de frutas en n listas, una por cada tipo. Luego en cada lista se elige la fruta de mayor peso y con las soluciones parciales para cada tipo, se suman los pesos máximos de cada tipo, obteniendo el peso total.

Este algoritmo parece ser dividir y conquistar porque al principio se dividen las frutas en n listas, pero en realidad no lo es. Aquí alguien podría confundirse y pensar que si ordena en la segunda etapa con mergesort (que si es DyC) o con quicksort, entonces eso hace que todo el algoritmo sea DyC, pero se equivoca. DyC requiere que el problema se reduzca a instancias más pequeñas del mismo problema, lo cual no ocurre en este caso.

El algoritmo en realidad es greedy, de momento que en cada paso (1 paso por cada tipo de fruta) se selecciona la fruta más pesada y con esta se va construyendo la solución completa.

c) (5 pts) Acantilado submarino



En el fondo del mar hay una cueva que se divide en 4 cuevas más profundas, las cuales a su vez se dividen en 4 cuevas aún más profundas. Por fortuna, las cuevas están numeradas (para evitar que los turistas se pierdan) y siempre se dividen cada 10 metros. Encuentra la cueva más profunda.

Input: Primero el número de bifurcaciones, luego una línea por bifurcación con 5 números i j k l m, indicando que la cueva i se divide en las cuevas j, k, l y m; un valor igual a 0 indica que no hay una cueva abierta para ese lado. Tanto i, j, k, l y m son valores entre 1 y 100.

Ouput: El número de la cueva más profunda y su profundidad desde el lecho marino.

Ejemplo:

Input

```
12
31 3 0 0 0
15 31 0 0 13
13 89 0 8 0
89 0 12 0 0
14 15 11 1 37
11 0 10 0 0
10 0 0 0 27
1 0 7 41 0
7 0 0 0 9
41 0 0 0 17
37 0 0 4 0
4 47 42 48 43
```

Output

```
12 50
```

Solución propuesta

Representar las cuevas como un árbol; a su vez, el árbol lo representamos como un grafo dirigido en una matriz de adyacencia. A medida que leemos los datos, se van anotando las aristas de modo que la matriz [i][j] == 1 solo si desde j se puede ir a i. Una vez leido los datos, la cueva inicial es aquella que no posee aristas incidentes (no llegan aristas, sólo salen aristas).

Con el árbol construido, representado con un grafo, partiendo desde el vértice raiz aplicar recursivamente: encontrar profundidad máxima y vertice respectivo para cada subárbol (hijos de la raiz actual) y quedarse con la profundidad máxima entre éstas, con el vértice respectivo, sumar 10 y retornar; el caso base es un vértice que no tiene hijos, en cuyo caso se retorna la raiz con profundidad 10.

Este algoritmo es absolutamente dividir y conquistar: el problema de encontrar la cueva más profunda se divide en 4: encontrar la cueva "hija" más profunta y luego, en base a las soluciones parciales se construye la solución global (eligiendo la más profunda y sumando 10).

d) (5 pts) La suma máxima en un toroide*

Una grilla que se envuelve tanto horizontal como verticalmente es llamada un toroide. Dado un toroide donde cada celda contiene un entero, determina el subrectángulo con la suma más grande. La suma del subrectángulo es la suma de todos los elementos en el rectángulo. La grilla a continuación muestra un toroide en que el subrectángulo máximo ha sido sombreado.

1	-1	0	0	-4
2	3	-2	-3	2
4	1	-1	5	0
3	-2	1	-3	2
-3	2	4	1	-4

Input

La primera línea contiene el número de casos (hasta 18). Cada caso comienza con un entero N (1 <= N <= 75) especificando el tamaño del toroide (siempre un cuadrado). Luego siguen N lineas describiendo el toroide, cada línea contiene N enteros entre -100 y 100 inclusive.

Output

Para cada caso entregar una línea que contiene un simple entero: la suma máxima del subrectángulo en el toroide.

Input de ejemplo

```
2

5

1 -1 0 0 -4

2 3 -2 -3 2

4 1 -1 5 0

3 -2 1 -3 2

-3 2 4 1 -4

3

1 2 3

4 5 6

7 8 9
```

Output de ejemplo

```
15
```

45

Solución propuesta

En cada caso, resolver así:

Crear una matriz de 2N x 2N y cuadruplicar los datos del toroide, de manera que tendremos 4 copias de la matriz dada. La gracia de hacer esto es que cualquier cuadrado de NxN es cómo si estuvieramos viendo una “cara” del toroide, y podremos emular la envoltura del toroide. Dado que todas las “caras” del toroide están en esta matriz, en particular estará el subrectángulo de suma máxima.

int mejor, mejor_i, mejor_j, mejor_k, mejor_m = 0

para i desde 0 hasta N-1 inclusive

para j desde 0 hasta N-1 inclusive

Suponer que [i][j] es la esquina superior izquierda de la solución. Ahora, para k desde i hasta i + N - 1 y para m desde j hasta j + N - 1 suponer que [k][m] es la esquina inferior derecha. Sumar todos los valores delimitados por esta submatriz. Si la suma es mejor que “mejor”, guardar este valor, así como i, j, k y m en la respectiva variable “mejor”.

Finalmente mejor contendrá la mejor suma (y los valores mejor_ contendran las coordenadas que delimitan el subrectángulo) Imprimr la mejor suma.

Nuevamente aquí hay una trampa: el algoritmo podría parecer de programación dinámica, de momento que prueba todas las posibilidades. Sin embargo, no es así dado que lo único que hace es probar todas las posibilidades, sin ningún tipo de optimización. Este tipo de solución es un buen ejemplo de resolver un problema a fuerza bruta. Por lo tanto, no es ninguna de las técnicas vistas.

Preguntas 3 y 4 – Resolución de problemas

A continuación se presentan 2 problemas. Resuélvelos usando las técnicas vistas en clases, señalando qué técnica estás usando.

Pregunta 3 – Equilibrio perfecto (20 pts)

El asombroso Equilibrini, un famoso malavarista, está preparando su nuevo espectáculo llamado “Equilibrio perfecto”. En este espectáculo, hay n elementos a su alrededor, cada uno con pesos p_0, p_1, … p_{n-1}. Mientras el asombroso Equilibrini se mantiene en su monociclo, le pide al público que elija algún elemento y uno de sus asistentes lo toma y lo ubica en una de sus manos (o sobre el elemento que tenga en esa mano). De esta forma, va formando dos torres en perfecto equilibrio sobre el pequeño monociclo.

Bueno, no todo es tan mágico como parece: en realidad es Equilibrini quien elige los elementos, pues el público normalmente pide casi todo lo que está a la vista, y él simplemente remarca el elemento que previamente había decidido. Esto lo hace por una buena razón, Equilibrini puede mantener las dos torres sin problema siempre y cuando, la diferencia entre el peso de ambas no sea mayor a 4 en ningún momento. Además, desea maximizar el asombro del público, el cual se puede medir como la cantidad de elementos que Equilibrini llega a equilibrar.

Ayuda al asombroso Equilibrini, creando un algoritmo que determine qué elementos debe seleccionar y en qué orden, de modo de maximizar lo asombroso del espectáculo.

El input consiste en una línea con n (el número de pesos) y luego una línea con los pesos p_0, p_1, … p_{n-1}. El output consiste en el nivel de asombro alcanzado.

Ejemplo Input	Output:
10 2 5 9 4 4 1 3 2 7 4	8

Este problema posiblemente se puede resolver con algún algoritmo greedy, pero convertir esta posibilidad en una afirmación requeriría acompañar el algoritmo de una demostración de correctitud.

Como ya lo he dicho en múltiples ocasiones, prefiero un algoritmo de programación dinámica que sería así:

Considerar el siguiente método:

int probar(int elementos, bool* agregados, int n, int* pesosManos, int elementosManos);*

el cual recibe la lista con los elementos que hay que agregar (peso y si está agregado o no), y los pesos en las mano derecha e izquierda (puntero de largo 2).

a) Entonces, si sólo falta 1 elemento por agregar ver si es posible agregarlo a la mano derecha, sino, a la izquierda, sino descartar el agregar este elemento y actualizar el pesosManos. Retornar el nuevo número de elementos (elementosManos +1 o +0, según el caso).

b) Si faltan más elementos por agregar, entonces tomar el primer elemento, y si es posible agregarlo a la mano izquierda (actualizando las variables respectivas) y llamar a "probar" sin este elemento. Repetir de manera análoga agregando en la mano derecha. Luego elegir el caso de probar mayor y retornarlo, actualizando las variables respectivas.

Para enmazcarar el algoritmo recursivo, crear un método "fachada" que reciba los elementos, inicialice el arreglo de agregados y el arreglo de pesos manos, llame a probar con estas variables y elementosManos=0, y una vez obtenido el máximo, elimine la memoria solicitada y retorne el máximo.

Hasta aquí, el algoritmo garantizará la optimalidad al probar cada caso. Para optimizarlo y evitar recalcular cada subproblema, guardar un caché en forma de mapa de boolean a int; si el mapa contiene un valor para boolean*, retornarlo de inmediato. Sino, efectuar las pruebas respectivas descritas en a) y b) y almacenar el resultado en el mapa antes de retornarlo.*

Pregunta 4 – Bloqueo anti fuerza bruta* (20 pts)

Recientemente ha habido un serio problema con las cajas fuertes Panda: ¡varias cajas fuertes han sido robadas! Estas cajas fuertes están usando el viejo sistema de cerraduras de combinaciones de 4 dígitos (sólo necesitas girar el dígito, ya sea para arriba o para abajo hasta que los cuatro coinciden con la clave). Cada dígito está hecho para girar de 0 a 9. Girar hacia arriba en el 9 hace que el dígito se convierta en 0, y girar hacia abajo el 0 hace que el dígito se convierta en 0. De momento que sólo hay 10.000 claves posibles, desde 0000 hasta 9999, cualquiera puede probar todas las combinaciones hasta que la caja se haya desbloqueado.

Lo hecho, hecho está. Pero con el fin de retrasar futuros ataques de ladrones, la Agencia de Seguridad Panda (ASP) ha diseñado una cerradura más segura con múltiples claves. En lugar de usar sólo una combinación como clave, la cerradura ahora tiene hasta N claves las cuales deben ser todas desbloqueadas antes que la caja fuerte puede abrirse. Esta cerradura funciona así:

- Inicialmente los dígitos están en 0000
- Las claves pueden ser desbloqueadas en cualquier orden, poniendo los dígitos en la cerradura para coincidir con la clave deseada y luego presionando el botón DESBLOQUEAR.
- Un botón mágico, SALTAR, puede convertir los dígitos en cualquier clave ya desbloqueada sin girar ningún dígito.
- La caja fuerte se desbloqueará si y sólo si todas las claves son desbloqueadas en un total mínimo de giros de dígitos, excluyendo cualquier cambio hecho con el botón SALTAR (si, esta característica es una de las más geniales).

- Si el número de giros de dígitos es excedido, los dígitos serán regresados a 0000 y todas las claves se bloquearán nuevamente. En otras palabras, el estado de la cerradura se reiniciará si el desbloqueo falla.

ASP está bien confiado en que este nuevo sistema retrasará un intento de violación por fuerza bruta, dándoles suficiente tiempo como para identificar y capturar los ladrones. Con el fin de determinar el número mínimo de giros de dígitos requeridos, ASP quiere que escribas un programa. Dadas todas las claves, calcula el mínimo número de giros necesarios para desbloquear la caja fuerte.

Input

La primera línea de input contiene un entero T, el número de casos que siguen. Cada caso comienza con un entero N (1 <= N <= 500), el número de claves. La siguiente línea contiene N números de exactamente 4 dígitos (con ceros a la izquierda) representando las claves del desbloqueo.

Output

Para cada caso, imprimir en una única línea el mínimo número de giros requeridos para desbloquear todas las claves.

Input de ejemplo

```
4
2 1155 2211
3 1111 1155 5511
3 1234 5678 9090
4 2145 0213 9113 8113
```

Output de ejemplo

```
16
20
26
17
```

Explicación del segundo caso

- Convierte `0000` en `1111`, giros: 4
- Convierte `1111` en `1155`, giros: 8
- Saltar `1155` en `1111`, podemos hacer esto porque `1111` había sido desbloqueado previamente.
- Convertir `1111` en `5511`, giros: 8

Giros totales = 4 + 8 + 8 = 20

Primero hay que representar el problema como un grafo, en que cada vértice es una de las claves y la distancia entre vértices está dada por el número de giros necesarios.

Luego, hay que calcular el número de giros entre el "0000" y las demás claves, y anexar "0000" como un vértice más, que está únicamente conectado a la clave de distancia mínima.

Entonces basta con encontrar un árbol de cobertura mínima (o MST) y sumar los tamaños de las aristas de este árbol. Para esto podemos usar el algoritmo de Prim, el cual es greedy, y veremos que la solución es greedy.

*: Estos problemas no son originales del profesor. Créditos pendientes.