Algoritmos y Estructuras de Datos (VJ1215) - Universitat Jaume I
Examen final - 2022/2023 - Segunda parte

8 de junio de 2023

Nombre:

Esta prueba consta de dos partes con una duración de dos horas cada una. Este es el enunciado de la segunda parte, que suma 5 puntos. La prueba es individual. No puedes consultar libros, apuntes ni dispositivos electrónicos. Al finalizar entrega tu solución junto con el enunciado (no es necesario que entregues todas las hojas). Pon tu nombre en cada hoja que entregues.

En los ejercicios 7, 8 y 9, escribe tus soluciones empleando el lenguaje C++.

En los ejercicios 7 y 8 debes decidir qué algoritmo utilizar para resolver eficientemente el problema planteado, e implementarlo añadiendo el método que se pide a la que corresponda de las siguientes dos clases utilizadas para representar grafos en los ejercicios de la asignatura. No puedes añadir ahí otros atributos. Puedes hacer uso de los Tipos Abstractos de Datos *Pila*, *Cola* y *Cola de Prioridad*, con las operaciones que necesites, sin tener que implementarlas (puedes poner sus nombres en inglés o en castellano).

```
class GrafoDirigido {
   struct Arco {
      int vecino;
      float peso;
      Arco * siguiente;
      Arco(int, float, Arco *);
   };
   struct Vertice {
      Arco * primerArcoDeEntrada;
      Arco * primerArcoDeSalida;
      int gradoDeEntrada;
      int gradoDeSalida;
      Vertice();
   };
   vector<Vertice> vertices;
 public:
   ...
};
```

```
class GrafoNoDirigido {
   struct Arco {
      int vecino;
      float peso;
      Arco * siguiente;
      Arco(int, float, Arco *);
   };
   struct Vertice {
      Arco * primerArco;
      int grado;
      Vertice();
   };
   vector<Vertice> vertices;
 public:
   ...
};
```

Ejercicio 6 — 0,5 puntos

Tenemos un montículo binario de máximos (*max-heap*) en un vector de talla n. Explica cómo podemos conseguir eficientemente, a partir de ahí, que el mismo vector contenga los datos ordenados de menor a mayor, sin utilizar ningún otro vector ni estructura de datos auxiliar. Ilustra tu solución dibujando el árbol y el vector que se obtienen tras cada paso si partimos del siguiente vector:

100	50	80	40	20	30	60

hasta llegar a tener:

20	30	40	50	60	80	100

Indica en esta hoja cuál es el coste temporal en el peor caso de tu solución en función de n, sin incluir el coste de construir el montículo (puesto que nos lo dan ya construido):

Coste temporal en el peor caso	$O($ $)$

Ejercicio 7 — 1,5 puntos

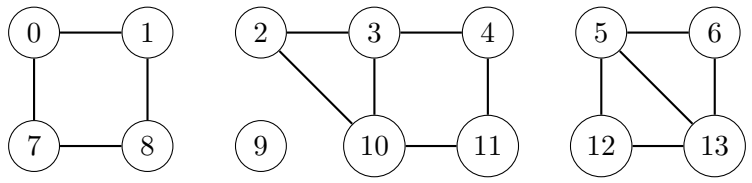
Sea $G = (V, E)$ un grafo no dirigido que representa posibles desplazamientos en un juego. En el juego participan n jugadores. Cada jugador ya está situado en un vértice diferente. Queremos elegir un subconjunto de esos jugadores de modo tal que se cumplan dos condiciones: (i) todo jugador sea alcanzable desde todos los demás jugadores, y (ii) la cantidad de jugadores elegidos sea la máxima posible.

Añade a la clase `GrafoNoDirigido` un método

```
vector<int> elegirJugadoresConectados(const vector<bool> & jugadores) const
```

que recibe un vector de talla $|V|$ en el que la posición i vale *true* cuando en el vértice i hay algún jugador. El método debe crear y devolver un vector que contenga los vértices en los que se encuentran los jugadores elegidos.

Por ejemplo, con el siguiente grafo



y con el siguiente vector *jugadores*

0	1	2	3	4	5	6	7	8	9	10	11	12	13
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*	*true*	*false*	*false*	*false*

un resultado válido sería $[1, 7, 8]$. No importa, dentro del resultado, en qué orden aparecen los vertices elegidos.

Se valorará que la solución no siga recorriendo el grafo si, con lo visitado hasta ese momento, ya se puede determinar el resultado.

Ejercicio 8 — 1,5 puntos

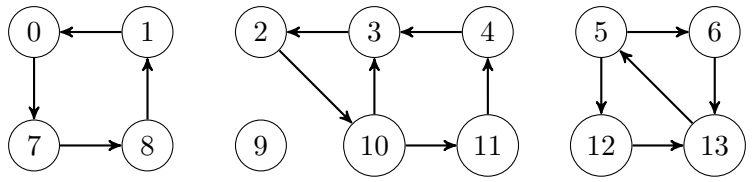
Sea $G = (V, E)$ un grafo dirigido que representa posibles desplazamientos en un juego. En el juego participan n jugadores. Cada jugador ya está situado en un vértice diferente. Queremos elegir un subconjunto de esos jugadores de modo tal que se cumplan dos condiciones: (i) todo jugador sea alcanzable desde todos los demás jugadores, y (ii) la cantidad de jugadores elegidos sea la máxima posible.

Añade a la clase `GrafoDirigido` un método

```
vector<int> elegirJugadoresConectados(const vector<bool> & jugadores) const
```

que recibe un vector de talla $|V|$ en el que la posición i vale *true* cuando en el vértice i hay algún jugador. El método debe crear y devolver un vector que contenga los vértices en los que se encuentran los jugadores elegidos.

Por ejemplo, con el siguiente grafo



y con el siguiente vector *jugadores*

0	1	2	3	4	5	6	7	8	9	10	11	12	13
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*	*true*	*false*	*false*	*false*

un resultado válido sería $[1, 7, 8]$. No importa, dentro del resultado, en qué orden aparecen los vertices elegidos.

Se valorará que la solución no siga recorriendo el grafo si, con lo visitado hasta ese momento, ya se puede determinar el resultado.

Ejercicio 9 — 1,5 puntos

Los n escalones de una escalera están numerados consecutivamente desde 0, que es el escalón inferior, hasta $n-1$, que es el escalón superior. Un jugador debe subir desde el escalón inferior hasta el escalón superior. Para avanzar, desde cualquier escalón i puede saltar hacia arriba hasta cualquier escalón $i+s$ siempre que s no supere una distancia máxima de salto que denominaremos d.

La energía que consume el jugador en su recorrido depende de los escalones que pisa y de los saltos que realiza:

1. Si en su recorrido hace uso del escalón i, consume *energíaPisar*$[i]$, para cualquier i desde 0 hasta $n-1$.
2. Si salta desde el escalón i hasta el escalón $i+s$, consume *energíaSaltar*$[s][i]$, para cualquier i desde 0 hasta $n-1$ y para cualquier s desde 1 hasta d.

Necesitamos una función que calcule la energía mínima necesaria para llegar desde el escalón inferior hasta el superior:

```
float minimaEnergia(const vector<float> & energiaPisar, const vector< vector<float> > & energiaSaltar)
```

Por ejemplo, con $n = 8$ escalones, distancia máxima de salto $d = 3$, este vector *energíaPisar*:

0	1	2	3	4	5	6	7
50	70	30	20	90	10	80	60

y esta matriz *energíaSaltar*:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	50	70	30	20	90	10	80	0
2	20	30	80	70	90	40	0	0
3	10	80	20	70	50	0	0	0

el resultado sería 230, que es la suma de 50 por pisar el escalón 0, 20 por saltar del 0 al 2, 30 por pisar el 2, 20 por saltar del 2 al 5, 10 por pisar el 5, 40 por saltar del 5 al 7 y 60 por pisar el 7:

0	1	2	3	4	5	6	7
50	70	30	20	90	10	80	60

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	50	70	30	20	90	10	80	0
2	20	30	80	70	90	40	0	0
3	10	80	20	70	50	0	0	0

Empleando Programación Dinámica, diseña e implementa un algoritmo eficiente para resolver el problema a) de forma recursiva y b) de forma no recursiva.

Indica el coste temporal y espacial de cada una de tus dos soluciones en el peor caso, en función de n y d.

	Recursiva	**No recursiva**
Coste temporal en el peor caso	$O($ $)$	$O($ $)$
Coste espacial en el peor caso	$O($ $)$	$O($ $)$