Algoritmos y Estructuras de Datos (VJ1215) - Universitat Jaume I
Examen final - 2023/2024 - Segunda parte

6 de junio de 2024

Nombre:

El examen final 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, 10 y 11, 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 siguiente clase utilizada para representar grafos dirigidos 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). No puedes hacer uso de otros métodos o funciones si no los implementas.

```
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:
   ...
};
```

Ejercicio 7 — 1,4 puntos

Consideremos un grafo dirigido $G = (V, E)$ que representa el escenario de un juego en el que participan $n \geq 2$ jugadores. Cada jugador debe alcanzar un objetivo situado en un vértice t. Inicialmente, para cada j desde 0 hasta $n-1$, el jugador j-ésimo se encuentra en el vértice $posiciones[j]$. Es posible que varios jugadores se encuentren en el mismo vértice.

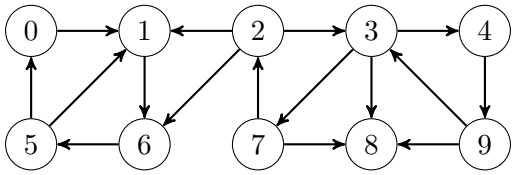
La distancia de cada jugador al objetivo es la mínima cantidad de arcos que debe recorrer para llegar a él. Para ajustar la dificultad del juego, es necesario calcular la diferencia entre la distancia a la que está el jugador más cercano al objetivo y la distancia a la que está el jugador más alejado del objetivo.

Implementa un método

```
int calcularBrechaDeDistancia(int t, const vector<int> & posiciones) const
```

que resuelva este problema. En caso de que exista algún jugador que no pueda alcanzar el objetivo, el método debe lanzar una excepción para indicarlo.

Por ejemplo, con el siguiente grafo:



si tuviéramos $t = 6$ y *posiciones* $= [9,\ 4,\ 5,\ 3]$ el jugador más cercano al objetivo sería el del vértice 5, que estaría a una distancia de 2; el más alejado sería el del vértice 4, que estaría a una distancia de 5. El resultado que nos piden sería 3, que es la diferencia entre ambas distancias.

Para que tu solución sea válida, debe ser eficiente. Además, se valorará que el algoritmo detenga el recorrido del grafo tan pronto como sea posible determinar el resultado.

Indica cuál es el siguiente coste de tu solución en función de n, $|V|$ y $|E|$. No es necesario que lo justifiques.

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

Ejercicio 8 — 1,4 puntos

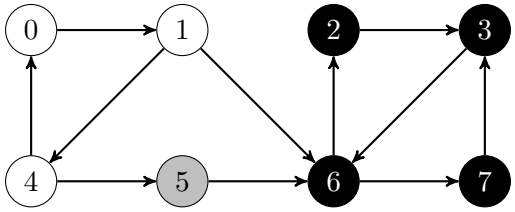
Consideremos un grafo dirigido $G = (V, E)$ que representa el escenario de un juego en el que participan varios equipos de jugadores. Inicialmente, cada vértice está asignado a un equipo y solo uno. Sea *equipo* un vector que en la posición v contiene el equipo asignado al vertice v, para cada v desde 0 hasta $|V| - 1$.

Implementa un método

```
bool mutuamenteAlcanzablesSonAmigos(const vector<int> & equipo) const
```

que devuelva *true* si, y solamente si, se cumple la siguiente condición: para todo par de vértices u y v, si v es alcanzable desde u y u es alcanzable desde v, entonces u y v deben pertenecer al mismo equipo. Recuerda que un vértice v se considera alcanzable desde otro vértice u si existe al menos un camino dirigido que parte de u y llega a v. Si v es alcanzable desde u y u es alcanzable desde v, entonces se dice que u y v son mutuamente alcanzables.

Por ejemplo, si tenemos el siguiente grafo:



con los equipos representados por el vector *equipo* $= [404, 404, 313, 313, 404, 666, 313, 313]$ el método debería devolver *true*, porque todos los vértices mutuamente alcanzables pertenecen al mismo equipo. De manera similar, con el vector *equipo* $= [64, 64, 64, 64, 64, 64, 64, 64]$, el resultado también sería *true*, por el mismo motivo: no existe ningún par de vértices mutuamente alcanzables que pertenezcan a equipos diferentes. Sin embargo, con el vector *equipo* $= [1, 1, 1, 1, 0, 1, 1, 1]$, el resultado sería *false* porque, por ejemplo, los vértices 0 y 4 son mutuamente alcanzables y no pertenecen al mismo equipo.

Para que tu solución sea válida, debe ser eficiente. Además, se valorará que el algoritmo detenga el recorrido del grafo tan pronto como sea posible determinar la existencia de dos vértices mutuamente alcanzables que pertenecen a equipos diferentes.

Indica cuál es el siguiente coste de tu solución en función de $|V|$ y $|E|$. No es necesario que lo justifiques.

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

Ejercicio 9 — 0,4 puntos

El siguiente algoritmo recursivo cuenta cuántas veces aparece un dato en un vector:

```
int contar(const vector<int> & v, int dato) {
   if (v.size() == 0) return 0;
   if (v.size() == 1)
      if (v[0] == dato) return 1;
      else return 0;
   int mitad = v.size() / 2;
   vector<int> izquierda(mitad);
   for (int i = 0; i < mitad; i++)
      izquierda[i] = v[i];
   vector<int> derecha(v.size() - mitad);
   for (int i = mitad, j = 0; i < v.size(); i++, j++)
      derecha[j] = v[i];
   return contar(izquierda, dato) + contar(derecha, dato);
}
```

Analiza los costes que se piden a continuación en función de n, siendo n la talla del vector que se le pasa en la primera llamada. Hazlo sin utilizar el Teorema Maestro. En el apartado d se pide determinar el coste si el vector se pasase siempre por valor, cambiando `const vector<int> & v` por `vector<int> v`, sin realizar ningún otro cambio.

Justifica tus respuestas proporcionando una explicación clara de los cálculos realizados y por qué son relevantes, incluyendo cualquier sumatorio que pueda surgir al evaluar el coste.

a) Coste temporal en el peor caso	$O($ $)$
b) Coste espacial en el peor caso	$O($ $)$
c) Coste espacial en el peor caso sin contar los vectores	$O($ $)$
d) Coste temporal en el peor caso pasando v por valor	$O($ $)$

Ejercicio 10 — 1,4 puntos

Imagina una competición que dura $n > 0$ días, en la que cada día se plantea un reto que otorga una cierta cantidad de puntos. Un jugador tiene la libertad de elegir a cuáles retos enfrentarse, siempre y cuando los puntos obtenidos formen una secuencia creciente. Es decir, la puntuación de cada reto en el que participe debe ser mayor que la del anterior reto superado. No es necesario que los retos elegidos sean consecutivos.

Para cada día i, numerado desde 0 hasta $n-1$, la cantidad de puntos que se puede obtener ese día es $premio[i]$. Se requiere una función que, a partir de estos datos, calcule la máxima suma de puntos que puede obtener el jugador:

```
int maximoPremioAcumulado(const vector<int> & premio)
```

Por ejemplo, supongamos que en una competición de $n = 7$ días los puntos otorgados son los siguientes:

0	1	2	3	4	5	6
70	40	90	30	60	80	50

En este caso, la máxima suma de puntos que puede obtener el jugador siguiendo la regla de formar una secuencia creciente sería $40 + 60 + 80 = 180$.

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

Di cuáles son los siguientes costes de cada una de tus dos soluciones y justifica brevemente por qué:

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

Ejercicio 11 — 0,4 puntos

A continuación se muestra una implementación incorrecta del algoritmo que, dados $N \geq 2$ puntos 2D, calcula la distancia entre los dos puntos más próximos empleando Divide y Vencerás: falta algo importante en la función `distanciaAlCuadradoMinima`.

Escribe en C++ los cambios necesarios para corregir esa implementación, indicando claramente su posición (puedes hacer referencia a los números de línea).

```
1  typedef pair<float, float> Punto;
2
3  float distanciaAlCuadrado(const Punto & p1, const Punto & p2) {
4     float a = p2.first  - p1.first;
5     float b = p2.second - p1.second;
6     return a * a + b * b;
7  }
8
9  bool compararY(const Punto & p1, const Punto & p2) {
10    return p1.second < p2.second;
11 }
12
13 float distanciaAlCuadradoMinima(const vector<Punto> & puntosOrdenX, const vector<Punto> & puntosOrdenY) {
14    int talla = puntosOrdenX.size();
15    if (talla == 2)
16       return distanciaAlCuadrado(puntosOrdenX[0], puntosOrdenX[1]);
17    if (talla == 3)
18       return min({distanciaAlCuadrado(puntosOrdenX[0], puntosOrdenX[1]),
19                   distanciaAlCuadrado(puntosOrdenX[0], puntosOrdenX[2]),
20                   distanciaAlCuadrado(puntosOrdenX[1], puntosOrdenX[2])});
21    int tallaIzquierda = talla / 2, tallaDerecha = talla - tallaIzquierda;
22    vector<Punto> izquierdaX(tallaIzquierda), derechaX(tallaDerecha),
23                  izquierdaY(tallaIzquierda), derechaY(tallaDerecha);
24    for (int i = 0; i < tallaIzquierda; i++)
25       izquierdaX[i] = puntosOrdenX[i];
26    for (int i = 0; i < tallaDerecha; i++)
27       derechaX[i] = puntosOrdenX[i + tallaIzquierda];
28    for (int i = 0, j = 0, k = 0; i < talla; i++)
29       if (puntosOrdenY[i] < derechaX[0])
30          izquierdaY[j++] = puntosOrdenY[i];
31       else
32          derechaY[k++] = puntosOrdenY[i];
33    float minima = min(distanciaAlCuadradoMinima(izquierdaX, izquierdaY),
34                       distanciaAlCuadradoMinima(derechaX, derechaY));
35    for (int i = 0; i < puntosOrdenY.size() - 1; i++)
36       for (int j = i + 1;
37            j < puntosOrdenY.size() && puntosOrdenY[j].second - puntosOrdenY[i].second < minima;
38            j++)
39          minima = min(minima, distanciaAlCuadrado(puntosOrdenY[i], puntosOrdenY[j]));
40    return minima;
41 }
42
43 float distanciaMinima(const vector<Punto> & puntos) {
44    vector<Punto> puntosOrdenX(puntos), puntosOrdenY(puntos);
45    sort(puntosOrdenX.begin(), puntosOrdenX.end());
46    for (int i = 0; i < puntosOrdenX.size() - 1; i++)
47       if (puntosOrdenX[i] == puntosOrdenX[i + 1])
48          return 0;
49    sort(puntosOrdenY.begin(), puntosOrdenY.end(), compararY);
50    return sqrt(distanciaAlCuadradoMinima(puntosOrdenX, puntosOrdenY));
51 }
```