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

23 de enero 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 añadir 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.

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

```
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

Sea $G = (V, E)$ un grafo ponderado, no dirigido y conexo que representa el mapa de una isla: cada vértice representa una aldea, y cada arco representa un camino de tierra que une dos aldeas. El peso asociado a cada arco es el coste económico de convertir ese camino en una carretera asfaltada de doble dirección. La isla no tiene ninguna carretera asfaltada actualmente. El gobierno de la isla quiere conseguir que desde cualquier aldea se pueda llegar hasta cualquier otra aldea empleando únicamente carreteras asfaltadas, y también quiere que el coste total de asfaltar todos los caminos elegidos para ello sea el mínimo posible.

Indica qué algoritmo utilizarías para resolver eficientemente ese problema, e impleméntalo en un método:

```
float minimoCosteAsfaltar() const
```

que devuelva como resultado cuál sería ese coste.

En este ejercicio 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).

Ejercicio 8 — 1,4 puntos

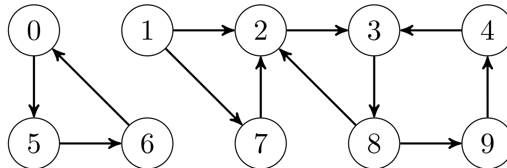
Sea $G = (V, E)$ un grafo dirigido que representa posibles desplazamientos en un juego. En el juego participan $n \geq 2$ guerreros. Cada guerrero está situado en un vértice. Puede haber varios guerreros en el mismo vértice. Sea *posiciones* un vector que contiene los vértices en los que se encuentran los guerreros.

Necesitamos un método

```
bool todosFuertementeConectados(const vector<int> & posiciones) const
```

que devuelva *true* si, y solo si, para todo par de guerreros a y b, o bien a y b están en el mismo vértice, o bien desde a se puede llegar por algún camino hasta b y desde b se puede llegar por algún camino hasta a.

Por ejemplo, con el siguiente grafo:



si tuviésemos *posiciones* $=$ [3, 9, 2, 9] significaría que hay 4 guerreros (uno en el vértice 3, otro en el vértice 2 y dos en el vértice 9) y el resultado que buscamos sería *true*. Con *posiciones* $=$ [7, 9, 2, 9], el resultado sería *false*.

Supón que no puedes usar los Tipos Abstractos de Datos *Pila*, *Cola* ni *Cola de Prioridad* (tampoco puedes implementarlos ni utilizar, para hacer ese papel, vectores u otras estructuras de datos). Teniendo en cuenta eso, diseña e implementa un algoritmo que resuelva eficientemente el problema y que no siga recorriendo el grafo si, con lo visto hasta ese momento, ya se puede 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 9 — 0,4 puntos

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

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

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. En los apartados c y d se pide cuáles serían esos costes si el vector se pasase siempre por valor cambiando `const vector<int> & v` por `vector<int> v`, sin cambiar nada más.

Justifica tus respuestas diciendo, para cada coste, qué cálculo realizas para obtener el resultado y por qué. En particular, si interviene algún sumatorio en el cálculo, indica de cuál se trata.

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

Un jugador debe superar n combates cuerpo a cuerpo, siendo $n > 0$. En cada combate puede elegir uno entre k contrincantes posibles. Se le permite elegir al mismo contrincante varias veces, consecutivas o no. En cada combate puede acumular puntos por dos motivos:

1. Si vence al contrincante j, obtiene $puntosVencer[j]$, para cualquier j desde 0 hasta $k-1$.
2. Si vence al contrincante j y en el combate inmediatamente anterior venció al contrincante i, consigue $puntosEvolución[i][j]$, para cualquier $i = 0, \ldots, k-1$ y para cualquier $j = 0, \ldots, k-1$. Esto se aplica en todos los combates excepto en el primero.

Por ejemplo, con $n = 3$ combates, si vence, en este orden, a los contrincantes 7, 4 y 6, la cantidad de puntos total que obtiene es $puntosVencer[7] + puntosEvolución[7][4] + puntosVencer[4] + puntosEvolución[4][6] + puntosVencer[6]$.

Necesitamos una función que calcule la máxima cantidad de puntos que puede acumular el jugador si elige y vence a los contrincantes adecuados para ello:

```
float maxPuntos(int n,
                const vector<float> & puntosVencer,
                const vector<vector<float> > & puntosEvolucion)
```

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

Indica cuáles son los siguientes costes de cada una de tus dos soluciones en función de n y k. No es necesario que lo justifiques.

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

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 }
```