Examen de Programación II
(Ingeniería Informática)

Junio 2008

1) Lenguaje C

1.a) (1 punto) Escribir una función C que reciba dos strings como parámetros y retorne un nuevo string formado por los caracteres comunes a ambos strings.

Para facilitar la implementación pueden utilizarse alguna de las funciones descritas a continuación:

```
NOMBRE
       strchr, strrchr - localiza un carácter en un string

SINOPSIS
       #include <string.h>

       char *strchr(const char *s, int c);

       char *strrchr(const char *s, int c);

DESCRIPCIÓN
  La función strchr() retorna un puntero a la primera ocurrencia
  del carácter c en el string s.

  La función strrchr() retorna un puntero a la última ocurrencia
  del carácter c en el string s.

VALOR DE RETORNO
  Las funciones strchr() y strrchr() retornan un puntero al
  carácter buscado o NULL si el carácter no existe en el string.
```

Solución:

```
#include <malloc.h>
#include <string.h>

char * caracteres_comunes(const char *s1, const char * s2) {
  char *comunes; // puntero al array de caracteres comunes

  // reserva espacio para un string del tamaño del menor de los
  // strings de entrada, ya que el número de caracteres comunes
  // nunca puede ser mayor que la longitud del más pequeño de los
  // strings de entrada
  if (strlen(s1) <= strlen(s2))
    comunes = (char *) malloc(strlen(s1)+1);
  else
    comunes = (char *) malloc(strlen(s2)+1);

  // comunes empieza siendo un string vacío
  comunes[0]='\0';
```

```
  int i; // índice de los caracteres de s1
  int iCom=0; // índice de los caracteres de comunes
  // para cada carácter de s1
  for(i=0; i<strlen(s1); i++) {
    // si el carácter i-ésimo de s1 está en s2 y todavía no está en
    // comunes le incluyo
    if ((strchr(s2,s1[i])!=NULL) && (strchr(comunes,s1[i])==NULL)){
      comunes[iCom]=s1[i];
      comunes[iCom+1]='\0'; // acaba después del último carácter
      iCom++;
    }
  }

  return comunes;
}
```

1.b) (1 punto) Escribir una función C que calcule el inverso del módulo de un número complejo. La función recibirá el número complejo como parámetro de entrada y devolverá en un parámetro de salida el valor del inverso del módulo.

La función retornará -1 para indicar que el módulo del complejo es 0 (y por tanto no es posible calcular su inverso) y 0 en otro caso para indicar que no ha habido ningún error.

Los números complejos son variables del tipo `complejo_t` que está definido como se indica a continuación:

```
  typedef struct {
    double real;
    double imag;
  } complejo_t;
```

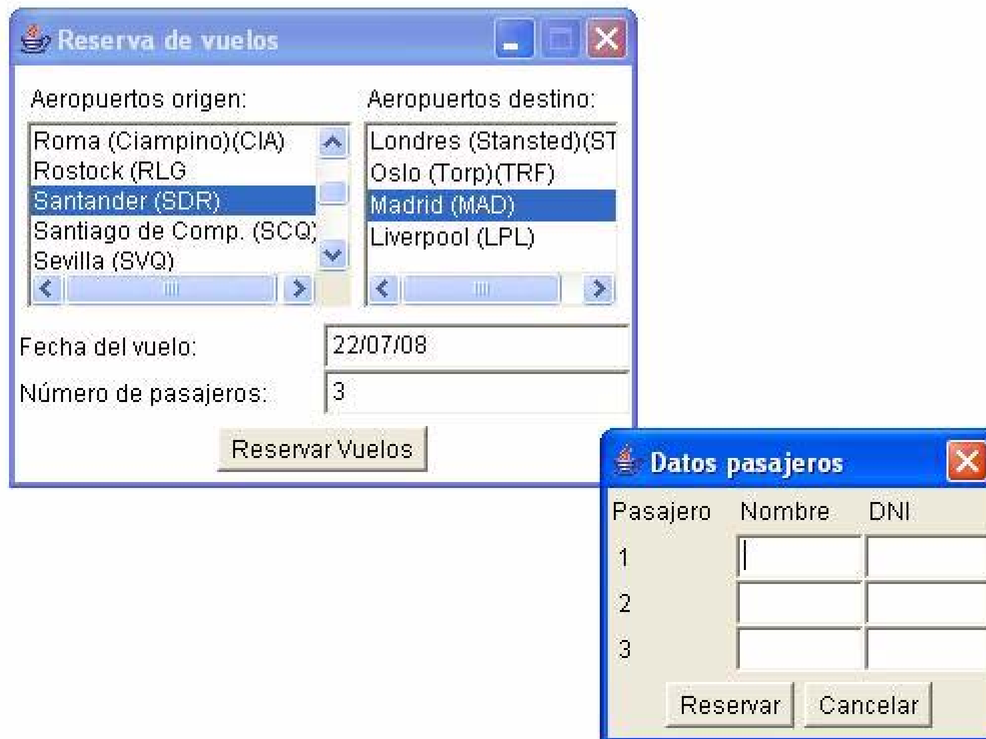
El módulo de un complejo se define como la raíz cuadrada de la suma de los cuadrados de sus componentes real e imaginaria. Para calcular la raíz cuadrada en C existe la función:

```
  double sqrt(double x); // definida en <math.h>
```

Solución:

```
#include <math.h>

int inverso_del_modulo(complejo_t c, double *inverso) {
  // si el módulo es 0 retorna error
  if (c.real == 0 && c.imag == 0)
    return -1;

  // calcula el inverso del módulo
  *inverso = 1 / sqrt(c.real*c.real + c.imag*c.imag);
  return 0;
}
```

2) (2.5 puntos) Se pretende realizar una GUI que permita gestionar las reservas de vuelos de una compañía aérea. La GUI se compondrá de dos ventanas: `VentanaVuelos` (`Frame`) y `DiálogoReserva` (`Dialog`). La apariencia de ambas ventanas es la mostrada a continuación:



Funcionalidad deseada (`VentanaVuelos`):

- Al crearse la ventana la lista de aeropuertos de origen deberá rellenarse con los aeropuertos de origen existentes. La lista de destinos estará vacía.
- Al seleccionar un aeropuerto de origen debe rellenarse la lista de destinos con sus aeropuertos de destino.
- Si se pulsa el botón "Reservar Vuelos" o se presiona "enter" sobre uno de los cuadros de texto y están seleccionados un aeropuerto de origen y otro de destino y el campo de texto "Número de pasajeros" contiene un número, aparece el `DiálogoReserva`.
- Si al tratar de reservar los billetes de los pasajeros introducidos con el `DiálogoReserva` se detecta que no es posible se escribe el mensaje "No hay billetes disponibles" en el cuadro de texto "Número de pasajeros".

Funcionalidad deseada (`DiálogoReserva`):

- Hasta que no se cierre este diálogo no deberá ser posible realizar ninguna acción sobre `VentanaVuelos`.
- Este diálogo permite introducir los datos de tantos pasajeros como se haya indicado en el campo de texto "Número de pasajeros" de `VentanaVuelos`.
- Al pulsar el botón "Reservar", y si se han introducido los datos de todos los pasajeros, se cierra (o desaparece) el diálogo y se tratan de reservar los billetes.

Se dispone de las clases ya realizadas `GestiónVuelos` y `Pasajero`, cuya interfaz se muestra a continuación:

```
Class GestiónVuelos
   Gestión de los vuelos de la compañía aérea

GestiónVuelos
public GestiónVuelos()
   Construye un objeto de gestión de los vuelos con los datos de
   aeropuertos en los que opera la compañía

destinos
public java.util.LinkedList<java.lang.String>
                              destinos(java.lang.String Origen)
     Lista de los aeropuertos de destino desde un aeropuerto dado
     Parameters:
         Origen - aeropuerto de origen
     Returns:
         lista de los aeropuertos de destino desde el aeropuerto
         indicado

orígenes
public java.util.LinkedList<java.lang.String> orígenes()
     Retorna una lista con todos los aeropuertos desde los
     que vuela la compañía
     Returns:
         lista con los aeropuertos desde los que vuela la compañía

reservaBilletes
public void reservaBilletes
                    (java.lang.String origen,
                     java.lang.String destino,
                     java.util.LinkedList<Pasajero> pasajeros,
                     java.lang.String fecha)
                 throws NoHayBilletes
     Reserva un billete para el vuelo entre los aeropuertos de
     origen y destino en la fecha indicada para los pasajeros
     indicados
     Parameters:
         origen - aeropuerto de origen
         destino - aeropuerto de destino
         pasajeros - datos de los pasajeros
         fecha - fecha el vuelo
     Throws:
         NoHayBilletes - cuando no hay billetes disponibles para
         todos los pasajeros
```

```
Class Pasajero
  Datos de un pasajero

Pasajero
public Pasajero(java.lang.String dni,
                java.lang.String nombre)
    Crea un pasajero con su DNI y su nombre
    Parameters:
        dni - DNI del pasajero
        nombre - nombre del pasajero

dni
public java.lang.String dni()
    Retorna el DNI del pasajero
    Returns:
        DNI del pasajero

nombre
public java.lang.String nombre()
    Retorna el nombre del pasajero
    Returns:
        nombre del pasajero
```

Se pide escribir todo el código correspondiente a la aplicación (excepto la distribución de componentes de la clase `VentanaVuelos` y las clases ya realizadas `GestiónVuelos` y `Pasajero`).

Según la manera de resolver el problema podría ser necesario utilizar alguno de los métodos que se describen a continuación:

- `remove`: elimina un componente de un contenedor (`Panel`, `Frame` o `Dialog`).
- `removeAll`: elimina todos los componentes de un contenedor (`Panel`, `Frame` o `Dialog`).
- `dispose`: cierra y elimina un `Frame` o un `Dialog`.

Solución:

```
public class PrincipalVuelos {
  public static void main(String[] args) {
    // crea la ventana principal de la aplicación
    VentanaVuelos ventanaVuelos = new VentanaVuelos();
  }
}
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.LinkedList;
/**
 * Clase VentanaVuelos
 * Ventana principal de la aplicación
 */
public class VentanaVuelos extends Frame {

  // Gestión de vuelos
  private GestionVuelos gestiónVuelos = new GestionVuelos();

  // lista de aeropuertos de origen
  private List lOrigen = new List();

  // lista de aeropuertos de destino
  private List lDestino = new List();

  // botón para reservar los billetes
  private Button bReservar = new Button("Reservar Vuelos");

  // campo de texto fecha
  private TextField tfFecha = new TextField("dd/mm/aa");

  // campo texto número de pasajeros
  private TextField tfNumPasajeros = new TextField();

  // referencia la objeto actual para los manejadores
  private Frame ventanaActual = this;

  /**
   * Constructor
   */
  public VentanaVuelos() {
    super("Reserva de vuelos");

    // distribución de los componentes (no pedido en el enunciado)
    ...

    // rellena la lista de aeropuertos de origen
    for(String s:gestiónVuelos.orígenes())
      lOrigen.add(s);

    // asigna manejador al botón y los cuadros de texto
    ManejaEventoBoton manejaBotón = new ManejaEventoBoton();
    bReservar.addActionListener(manejaBotón);
    tfFecha.addActionListener(manejaBotón);
    tfNumPasajeros.addActionListener(manejaBotón);

    // asigna manejador de la lista de origen
    lOrigen.addItemListener(new ManejadorListaOrigen());

    // configuración final de la ventana
    addWindowListener(new ManejadorCierraVentana());
```

```
    pack();
    setResizable(false);
    setVisible(true);
  }

  // manejador de eventos de la ventana (sólo nos
  // interesa el evento de cierre)
  class ManejadorCierraVentana extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0); // acaba la aplicación
    }
  }


  // manejador de eventos del botón y de los cuadros de texto
  // si el número de pasajeros es correcto y están seleccionados
  // los aeropuertos de origen y destino, se crea el DiálogoReserva
  class ManejaEventoBoton implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
      try {
        // obtiene el número de pasajeros, si no es un número
        // se lanzará NumberFormatException
        int numPasajeros=
                   Integer.parseInt(tfNumPasajeros.getText());
        if (lOrigen.getSelectedIndex()!=-1 &&
            lDestino.getSelectedIndex()!=-1) {
          // Sólo se crea el diálogo si están seleccionados los
          // aeropuertos y el número de pasajeros es correcto
          new DialogoReserva(ventanaActual,numPasajeros);
        }
      } catch (NumberFormatException e) {
        // el enunciado no pide poner ningún mensaje, así
        // que valdría con no hacer nada en este manejador.
        tfNumPasajeros.setText("Número incorrecto");
      }
    }
  }

  // Manejador de eventos de la lista de orígenes
  // rellena la lista de aeropuertos de destino
  class ManejadorListaOrigen implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      lDestino.removeAll(); // borra todos los que había
      // añade los correspondientes al nuevo origen
      for(String s:gestiónVuelos.destinos(
          lOrigen.getSelectedItem())) {
        lDestino.add(s);
      }
    }
  }

  /**
   * invocado desde DiálogoReserva para intentar reservar
   * los billetes
```

```
     */
    public void reservaBilletes(LinkedList<Pasajero> pasajeros) {
      try {
        gestiónVuelos.reservaBilletes(lOrigen.getSelectedItem(),
            lDestino.getSelectedItem(), pasajeros,
            tfFecha.getText());
      } catch (NoHayBilletes e) {
        // si no se ha podido reservar los billetes se muestra
        // el mensaje de error
        tfNumPasajeros.setText("No hay billetes disponibles");
      }
    }


  } // fin clase VentanaVuelos


  import java.awt.*;
  import java.awt.event.*;
  import java.util.LinkedList;
  /**
   * Diálogo que permite introducir los datos de los
   * pasajeros para los que se pretende realizar la reserva
   */
  public class DiálogoReserva extends Dialog {
    // campos de texto para los nombres y DNIs de los pasajeros
    // Se crean en el constructor de forma que haya un campo
    // de texto para el nombre y uno para el DNI para cada
    // pasajero.
    private TextField[] tfNombres;
    private TextField[] tfDNIs;

    private Button bReservar = new Button("Reservar");
    private Button bCancelar = new Button("Cancelar");

    /**
     * constructor
     */
    public DiálogoReserva(Frame owner, int numPasajeros) {
      super(owner, "Datos pasajeros", true); // diálogo modal

      // crea los dos arrays de campos de texto de 'numPasajeros'
      // elementos
      tfNombres = new TextField[numPasajeros];
      tfDNIs = new TextField[numPasajeros];
      for(int i=0; i<tfNombres.length; i++) {
        // crea cada uno de los cuadros de texto
        tfNombres[i]=new TextField();
        tfDNIs[i]=new TextField();
      }

      // layout border para el diálogo
      setLayout(new BorderLayout());
```

```
    // Grid con los datos de los pasajeros (center)
    Panel pDatos = new Panel(new GridLayout(0,3));
    pDatos.add(new Label("Pasajero"));
    pDatos.add(new Label("Nombre"));
    pDatos.add(new Label("DNI"));
    for(int i=0; i<numPasajeros; i++) {
      pDatos.add(new Label("  "+(i+1)));
      pDatos.add(tfNombres[i]);
      pDatos.add(tfDNIs[i]);
    }
    add(pDatos, BorderLayout.CENTER);

    // flow con los botones (south)
    Panel pBotones = new Panel(new FlowLayout());
    pBotones.add(bReservar);
    pBotones.add(bCancelar);
    add(pBotones, BorderLayout.SOUTH);

    // pone manejadores a los botones
    bReservar.addActionListener(new ManejadorBotónReservar());
    bCancelar.addActionListener(new ManejadorBotónCancelar());

    // configuración final del diálogo
    pack();
    addWindowListener(new ManejadorCierra());
    setVisible(true);
  }

  // manejador de eventos de la ventana (sólo nos
  // interesa el evento de cierre)
  class ManejadorCierra extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      dispose(); // cierra el diálogo
    }
  }

  // manejador de eventos del botón reservar
  // lee los datos de los pasajeros e intenta reservar
  // los billetes
  class ManejadorBotónReservar implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      LinkedList<Pasajero> pasajeros =
        new LinkedList<Pasajero>();
      boolean introducidosTodosLosDatos = true;
      for(int i=0; i<tfNombres.length; i++) {
        // si no se han introducido los datos del pasajero
        // sale del lazo
        if (tfNombres[i].getText().equals("") ||
            tfDNIs[i].getText().equals("")) {
          // podría estar bien mostrar un mensaje de error, pero
          // el enunciado no lo pide
          introducidosTodosLosDatos=false;
          break;
        }
```

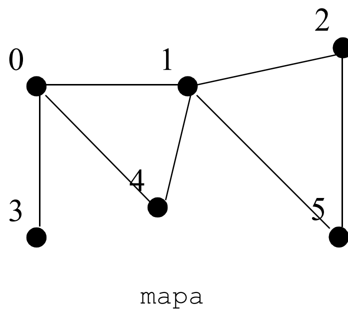
```
          // datos correctos: añade un nuevo pasajero
          pasajeros.add(new Pasajero(tfNombres[i].getText(),
              tfDNIs[i].getText()));
        }

        if (introducidosTodosLosDatos) {
          // intenta reservar los billetes
          ((VentanaVuelos)getOwner()).reservaBilletes(pasajeros);

          // elimina y cierra el diálogo
          dispose();
        }
      }
    }

    // manejador de eventos del botón cancelar
    class ManejadorBotónCancelar implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        // elimina y cierra el diálogo
        dispose();
      }
    }
  }
```

3) Esquemas algorítmicos

3.a) (2.5 puntos) Se dispone de un mapa de localidades representado mediante dos matrices:

- matriz de conexiones: un `1` en la casilla `(i,j)` indica que existe un camino directo entre las localidades `i` y `j` y un `0` que no existe tal camino. Se trata de una matriz simétrica.
- matriz de distancias en línea recta: el valor de la casilla `(i,j)` corresponde a la distancia en línea recta entre esas dos localidades. Se trata de una matriz simétrica.

Ejemplo sencillo:



mapa

$$\begin{bmatrix} 0 & 1.0 & 1.9 & 1.0 & 1.5 & 2.9 \\ 1.0 & 0 & 1.1 & 2.4 & 0.7 & 1.7 \\ 1.9 & 1.1 & 0 & 3.4 & 2.2 & 1.4 \\ 1.0 & 2.4 & 3.4 & 0 & 0.7 & 2.2 \\ 1.5 & 0.7 & 2.2 & 0.7 & 0 & 1.4 \\ 2.9 & 1.7 & 1.4 & 2.2 & 1.4 & 0 \end{bmatrix}$$

distancias

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

conexiones

Se desea implementar un algoritmo basado en Ramificación y Poda que permita encontrar el itinerario que une dos localidades cruzando el menor número de localidades posible (sin importar que la distancia recorrida pueda ser menor por otro itinerario que cruza más localidades).

La distancia puede utilizarse como un indicativo de la probabilidad de que haya un camino directo entre dos localidades: cuando más cerca estén las ciudades, más probable es que haya un camino que las una.

Para implementar el algoritmo se dispone de la clase `Itinerario` ya implementada, cuya interfaz es la mostrada a continuación:

```
Itinerario
public Itinerario(double[][] conexiones,
                  double[][] distancias,
                  int destino)
    Crea un itinerario vacío.
    Parameters:
        conexiones - matriz de conexiones entre localidades
        distancias - matriz de distancias entre localidades
        destino - localidad de destino

añadeLocalidad
public void añadeLocalidad(int localidad)
    Añade la localidad al final del itinerario.
    Parameters:
        localidad - localidad a añadir

localidadIncluida
public boolean localidadIncluida(int localidad)
    Permite conocer si una localidad ya ha sido incluida en el
    itinerario
    Parameters:
        localidad - localidad a ver si está incluida
    Returns:
       verdadero si la localidad ya está incluida en el itinerario
        y falso en caso contrario

últimaLocalidadVisitada
public int últimaLocalidadVisitada()
    Retorna la última localidad visitada en el itinerario
    Returns:
        última localidad visitada

copia
public Itinerario copia()
    Realiza una copia idéntica al objeto actual
    Returns:
        copia del objeto actual
```

Se pide escribir el algoritmo basado en Ramificación y Poda que resuelve el problema planteado y los métodos y/o atributos que pueda ser necesario añadir a la clase Itinerario.

Solución:

```
public static Itinerario calculaItinerario(double[][] conexiones,
    double[][] distancias, int origen, int destino) {
  Itinerario mejorSol = null;
  PriorityQueue<Itinerario> colaVivos =
                        new PriorityQueue<Itinerario>();

  // el primer e-nodo únicamente contiene la localidad de origen
  Itinerario eNodo =
            new Itinerario(conexiones, distancias, destino);
  eNodo.añadeLocalidad(origen);

  // mientras quedan nodos vivos y mejoran el coste de la mejor
  // solución encontrada hasta el momento se siguen procesando
  while (eNodo!=null
        && (mejorSol==null || eNodo.coste() < mejorSol.coste())) {

    // ramifica el eNodo: crea un hijo para cada una de
    // las localidades todavía no incluidas en el eNodo
    // y que estén conectadas con la última localidad visitada
    for(int loc=0; loc<conexiones.length; loc++) {
      // si la localidad no está en el eNodo y está conectada con
      // la última visitada crea el hijo
      if (conexiones[eNodo.últimaLocalidadVisitada()][loc]==1
            &&!eNodo.localidadIncluida(loc)) {
        // ramifica
        Itinerario nHijo = eNodo.copia();
        nHijo.añadeLocalidad(loc);

        // ¿se poda la rama que comienza en el hijo?
        if (mejorSol==null || nHijo.coste()<mejorSol.coste()) {
          if (loc==destino) {
            // si ha llegado al destino el hijo se convierte en
            // la mejor solución encontrada hasta el momento
            mejorSol=nHijo;
          } else {
            // si no ha llegado se encola
            colaVivos.add(nHijo);
          }
        }
      }
    } // for

    // Obtiene el nodo vivo de menor coste
    eNodo=colaVivos.poll();
  } // while

  // retorna la mejor solución encontrada
  return mejorSol;
}
```

A la clase `Itinerario` se le añade el atributo:

```
// Longitud del itinerario (número de localidades que
// le componen)
private int longitud=0;
```

Y al método `AñadeLocalidad` se le añade la línea:

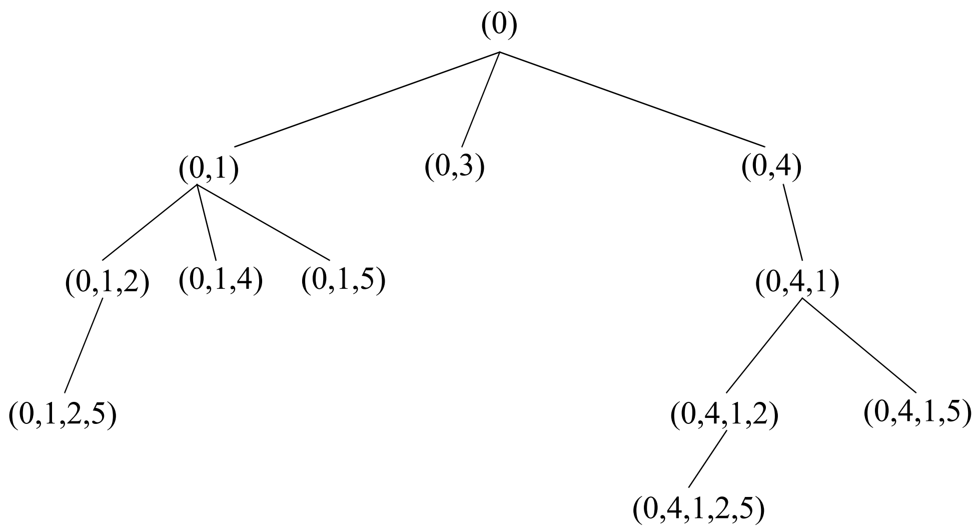
```
longitud++;
```

Además se añaden los métodos `coste` y `compareTo`:

```
/**
 * Un itinerario tiene mayor coste asociado
 * cuanto mayor sea el número de localidades que le componen
 * @return longitud del itinerario
 */
public int coste() {
  return longitud;
}

/**
 * Los itinerarios se comparan en base a su longitud (menor
 * el que menos localidades incluya)
 * A igualdad de coste, se considerará menor el itinerario cuya
 * última localidad se encuentre más cerca del destino (cuanto
 * más próximas estén dos localidades, más probable es que haya
 * un camino que las una, y por tanto, más probable será que el
 * itinerario se convierta en una solución)
 */
public int compareTo(Itinerario it) {
  // menor el que menos localidades incluya
  int ret = coste() - it.coste();
  // a igualdad de coste, es menor el más cercano al destino
  if (ret==0)
    return (int) (distancias[últimaLocalidadVisitada()][destino]
      - distancias[it.últimaLocalidadVisitada()][destino]);
  return ret;
}
```

3.b) (0.5 puntos) Dibujar el árbol de búsqueda correspondiente al mapa del ejemplo cuando la localidad de origen es la 0 y la de destino la 5.

Solución:



3.c) (0.5 puntos) ¿Sería posible resolver este mismo problema utilizando los esquemas "voraz" y "vuelta atrás?. Razona las respuestas.

Solución:

Sí se puede resolver utilizando un algoritmo voraz. Podría utilizarse el algoritmo de Dijkstra, sin más que suponer que cualquier par de localidades conectadas entre sí están separadas por la misma distancia. Con esta suposición, el camino mínimo encontrado por Dijkstra sería el que recorre el menor número de localidades posible.

También se puede resolver por VA, ya que todos los problemas resolubles por RyP lo son también por VA.

3.d) (0.5 puntos) Suponiendo que las soluciones parciales al problema se representan mediante objetos de la clase `ArrayList<Integer>` (p.e. el itinerario que pasa por las ciudades 0, 4 y 1 se representaría mediante el array [0,4,1]), describir las funciones de cruce y mutación para resolver el problema utilizando un algoritmo genético. Tener en cuenta que las funciones deben evitar crear individuos incorrectos para este problema. (para la descripción de las funciones utilizar pseudocódigo o lenguaje natural).

Solución:

Cruce: se obtiene una localidad de cruce aleatoria para la madre y otra para el padre. El primer hijo consiste en la primera parte del padre (hasta el punto de cruce) y la segunda de la madre y el segundo hijo estará formado por la primera de la madre y la segunda del padre. Habría que comprobar que los hijos obtenidos constituyen itinerarios válidos, es decir, que existe conexión entre cada localidad y la que va a continuación.

Para no tener que comprobar si los hijos son válidos, se podrían buscar las posibles parejas de puntos de cruce que generan hijos válidos y posteriormente elegir aleatoriamente una de esas parejas.

Mutación: una posible función de mutación consistiría en elegir aleatoriamente una localidad y sustituirla por otra elegida también de forma aleatoria. Posteriormente habría que comprobar si el itinerario "mutado" continúa siendo válido, es decir, habría que comprobar que la nueva localidad está conectada con las localidades anterior y posterior.

3.e) (0.5 puntos) Describir un heurístico sencillo para este problema (utilizar pseudocódigo o lenguaje natural). ¿Encontrará solución siempre?

Solución:

El algoritmo comienza con un itinerario formado únicamente por la localidad de origen y va añadiendo una nueva localidad a cada paso hasta llegar a la localidad de destino.

El criterio para añadir cada nueva localidad es el siguiente: de entre todas las localidades no visitadas aún que puedan ser alcanzadas desde la última localidad visitada se escoge la localidad más cercana a la de destino.

El algoritmo no encuentra solución siempre, ya que podría llegar a una localidad "terminal" desde la que no pudiera seguir avanzando (como por ejemplo la localidad 3 del ejemplo).

4) (1 punto) Escribir una función Haskell que recibe un carácter y un string y produce otro string igual al anterior pero con las ocurrencias del carácter repetidas. Ejemplo de invocación:

```
duplica_letra 'o' "hola adios" ------------------> "hoola adioos"
```

(Según la forma elegida para escribir la función podría ser necesario utilizar el operador "distinto", que en Haskell es "`/=`")

Solución:

```
duplica_letra :: Char -> String -> String
duplica_letra c [] = []
duplica_letra c (x:xs)
  | c == x     = c:c:(duplica_letra c xs)
  | c /= x     = x:(duplica_letra c xs)
```