Examen de Teoría de la Programación

E. U. ING. TEC. EN INFORMÁTICA DE OVIEDO

Final Junio – Curso 2004-2005 4 de junio de 2005

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

Soluciones

3.

a)

Precondición formal que verifique las condiciones de entrada incluyendo los límites de n.
La función tiene un argumento de entrada salida por tanto seguimos el convenio de dar otro nombre al parámetro que cambia en la precondición, para poder referirnos a él en la postcondición.

$\{Q \equiv (3 \leq n \leq 1000) \wedge (\forall i \in \{2..n-1\}.a[i-1]<a[i]) \wedge (a=copia)\}$

El límite inferior de n se establece en 3 para que se pueda comprobar la segunda parte de la condición, es decir, que haya elementos para comparar. Si la condición fuera otra los límites de n podrían ser: $(1 \leq n \leq 1000)$

b)

Postcondición formal.

$\{R \equiv (\forall i \in \{2..n\}.copia[i-1]<copia[i])\}$

Simplemente, comprobar que el vector resultado completo queda ordenado en orden creciente.

Sin embargo, con esta condición no verificamos que el resto de los elemento permanecen con el mismo valor. Es decir, el implementador podría devolver el resultado (0, 0, 0, 0, 0, 5) y verificaría la condición. Y esto no es lo que queremos.
Por tanto, habría que establecer una condición más restrictiva:

$\{R \equiv (\forall i \in \{2..n\}.copia[i-1]<copia[i]) \wedge$
$(\exists j \in \{1..n\}.copia[j]=a[n]) \wedge$
$(\forall i \in \{1..j-1\}.copia[i]= a[i]) \wedge (\forall i \in \{j+1..n\}.copia[i]= a[i-1])\}$

4.

a)

Aplicación del algoritmo divide y vencerás para multiplicar matrices optimizado con las relaciones de Strassen. Explica, simplemente como divides el problema en subproblemas, que llamadas recursivas tenemos y como realizarías la composición de los resultados de cada subproblema.

- Consideremos matrices de orden 2^x. Dividimos las matrices en cuatro cuadrantes iguales de orden 2^{x-1}. Por tanto, cada una de las letras en la expresión siguiente representará una submatriz dentro de la matriz original.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- Sabemos que se cumple lo siguiente:
  - C11= A11·B11 + A12·B21
  - C12= A11·B12 + A12·B22
  - C21= A21·B11 + A22·B21
  - C22= A21·B12 + A22·B22
- Con las relaciones de Strassen en vez de 8 multiplicaciones tendremos 7 multiplicaciones de matrices y otras operaciones añadidas. Cada multiplicación se corresponderá con una llamada recursiva, hasta que las matrices tengan orden 1, donde multiplicaremos directamente los elementos. Las relaciones de Strassen constituyen la formula de composición del resultado.

b)

Complejidad temporal del algoritmo planteado.

- Complejidad temporal:
  - Divide y vencerás con división
  - a=7, b=2, K=2 → $O(n^{\log 7}) = O(n^{2,81})$

c)

Algoritmo para multiplicar una matriz de 4x4
Es mejor, en tiempo de ejecución, emplear el algoritmo clásico, ya que pese a que la complejidad asintótica es menor en Divide y Vencerás con las relaciones de Strassen esto sólo da tiempos de ejecución menores para n grandes, debido a las constantes multiplicativas.

5.

a)

La técnica que se debe emplear es backtracking. El motivo de elegir esta técnica se basa en dos condiciones que se establecen para el problema: “número de monedas que manejan las cajeras sea siempre el mínimo posible” es decir nos piden el óptimo, teniendo en cuenta que “en un momento determinado puede quedarse sin algún tipo de monedas” esto descarta un algoritmo devorador ya que no siempre nos dará el óptimo, y por otra parte, “alto no nos importa demasiado el tiempo de cálculo”.

b)

Código en Java del algoritmo que a partir de las monedas disponibles en ese momento y la cantidad que hay que devolver, permita obtener siempre el número óptimo de monedas o un error en caso de que no sea posible devolver la cantidad.

```
public class Cambio {
	// constantes para el sistema monetario euro
	private static final double[] valor= {2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01
};

	private int[] cambio;
	private int numMonedas= 0;
	private int[] solucion;
	private int minMonedas= 10*8; // monedas iniciales

	public Cambio()
	{
		cambio= new int[valor.length];
		for (int i= 0; i<cambio.length; i++)
		{
			cambio[i]= 0;
		}
	}

	public void devolverCambio(double importeDevolver, int[] disponible)
	{
	  for (int i= 0; i<valor.length; i++)
	    // Comprobar si estado es posible
	    if ((disponible[i]>0) && ((importeDevolver - valor[i]) >= 0))
	    {
	        // Anotar el estado
	      importeDevolver= importeDevolver - valor[i];
			// decrementa lo que queda por devolver
	        disponible[i]--;  // una moneda menos de ese valor
	        cambio[i]++;            // mete moneda en la solución
	        numMonedas++;

	      if (importeDevolver!=0)
			devolverCambio(importeDevolver, disponible);
	      else
		 // quedarse con la mejor solución hasta el momento
			if (numMonedas<minMonedas)
			{
			solucion= cambio;
			minMonedas= numMonedas;
				imprimir(cambio);
			}

	      // Borrar anotación de estado
	        importeDevolver+= valor[i];
	        disponible[i]++;
	        cambio[i]--;
```

```
            numMonedas--;
        }

    }

    private void imprimir(int[] vector) {
        // TODO Auto-generated method stub
        for (int i= 0; i<vector.length; i++)
        {
            System.out.print(vector[i]+", ");
        }
        System.out.println();
    }
```

Código de llamada al método.

```
    public static void main(String[] args) {
        double cantidadDevolver= 0.07;
        int[] monedasDisponibles= {10, 10, 10, 10, 10, 10, 10, 10 };
        int[] cambio= {0, 0, 0, 0, 0, 0, 0, 0};

        int[] solucion= new int[8];

        Cambio c= new Cambio();
        c.devolverCambio(cantidadDevolver, monedasDisponibles);
        System.out.println("Fin calculo");

    }
```

6.

a)

Heurístico para la función de selección
Lo que hay que minimizar es:

$$T = \sum_{i=1}^{n} (tiempo\_en\_la\_tienda)$$

- Posibilidades:
  - Seleccionar el cliente que va a realizar la compra más rápido.
  - Seleccionar el cliente que más tarda en su compra.
  - Otras permutaciones de los clientes.

Tres clientes, con t_1= 8, t_2= 15 y t_3= 5.

Probamos con todas las permutaciones de los 3 clientes:

	1er cliente	2º cliente	3er cliente	Tiempo total que tardan todos los clientes en realizar su compra
Primero el más rápido	5	8	15	5 + (5 + 8) + (5 + 8 + 15) = 46
	8	5	15	8 + (5 + 8) + (5 + 8 + 15) = 49
	15	5	8	15 + (15 + 5) + (15 + 5 + 8) = 63
Primero el más lento	15	8	5	15 + (15 + 8) + (15 + 8 + 5) = 66
	5	15	8	5 + (5 + 15) + (5 + 15 + 8) = 53
	8	15	5	8 + (8 + 15) + (8 + 15 + 5) = 59

En ejercicio, no hacía falta hacerlo, con hacerlo con dos o tres, es suficiente para hacerse una idea.
Por tanto, el heurístico elegido será primero el más rápido.

b)

Algoritmo en Java para dar solución a este problema.

```
    public int calcularTiempoCompra(Cliente[] clientes)
    {
      int tiempoTotal= 0;
      int tiempoCliente;
      int i= 0;
```

```
  // Ordenar vector de clientes
  // para que queden ordenados de menor a mayor
  // por el tiempo de compra
  clientes= heuristicoOrdenarClientes(clientes);

  // Inicializamos vector X a 0
  while (i<clientes.length) // mientras haya clientes
  {
    // Siguiente cliente
       tiempoCliente= tiempoTotal + clientes[i].tiempo;
    tiempoTotal= tiempoTotal + tiempoCliente;
       i++;
  }

  return tiempoTotal;
}
```

7.

La expansión de cada nodo en anchura que utiliza el algoritmo de ramificación y poda necesita una estructura de datos para almacenar los nodos pendientes de explorar.

Se utiliza una cola de prioridad, debido a que esta estructura de datos ordena los elementos según una clave dada y no los deja colocados en el orden de inserción como otras estructuras secuenciales. En el algoritmo de ramificación y poda interesa **ordenar los nodos** por el valor del **heurístico de ramificación** y así nos permite **insertar y extraer** el que tiene mejor heurístico (el de menor valor) con una operación que tiene una **complejidad temporal baja O(log n)**.