Solución Examen de Programación 3 y III (10/02/2011)

Instituto de Computación, Facultad de Ingeniería, UdelaR

Parte Teórica Obligatoria

Ejercicio 1 (10 puntos)

El algoritmo divide los datos de entrada en tres vectores de igual tamaño, realiza una llamada recursiva sobre cada uno de estos vectores, y por último realiza un proceso sobre los datos de entrada con coste cuadrático.

Esto se traduce en:

$$T(N) = 3 \cdot T(\frac{N}{3}) + k \cdot N^2$$

Suponiendo N potencia de 3:

$$T(3^N) = 3.T(3^{N-1}) + k \cdot 3^{2N}$$

Se plantea la recurrencia, multiplicando cada fila por 3^i y realizando la suma:

$$\begin{array}{ll} & T(3^N) = 3.T(3^{N-1}) + k \cdot 3^{2N} \\ + & 3 * T(3^{N-1}) = 3^2.T(3^{N-2}) + 3 * k \cdot 3^{2(N-1)} \\ & \vdots \qquad\qquad \vdots \\ + & 3^i * T(3^{N-i}) = 3^{i+1}.T(3^{N-(i+1)}) + 3^i * k \cdot 3^{2(N-i)} \\ & \vdots \qquad\qquad \vdots \\ + & 3^N * T(3^{(N-N)}) = 3^N * k \cdot 3^{2(N-N)} \\ & \text{---} \\ = & T(3^N) = k * \sum_{i=0}^{i=N} 3^{2(N-i)+i} = 3^{2N} * k * \sum_{i=0}^{i=N} \frac{1}{3^i} \end{array}$$

.

Entonces:

$$T(3^N) = 3^{2N} * k * \sum_{i=0}^{i=N} \frac{1}{3^i} .$$

Cambio de variable $n = 3^N \Rightarrow N = \log_3(n)$.

$$T(n) = 3^{2.\log_3(n)} * k' * \sum_{i=0}^{i=\log_3(n)} \frac{1}{3^i} = 3^{\log_3(n^2)} * k * \sum_{i=0}^{i=\log_3(n)} \frac{1}{3^i} = k * n^2 * \sum_{i=0}^{i=\log_3(n)} \frac{1}{3^i}$$

Calculo del límite $\lim_{n\to\infty}(\frac{T(n)}{n^2}) = \lim_{n\to\infty}(\frac{k * n^2 * \sum_{i=0}^{i=\log_3(n)} \frac{1}{3^i}}{n^2}) = \lim_{n\to\infty}(k * \sum_{i=0}^{i=\log_3(n)} \frac{1}{3^i})$

Utilizando la sugerencia, $\sum_{i=0}^{\infty} \frac{1}{3^i}$ converge.

Entonces también converge el siguiente límite

$$\lim_{n\to\infty}(k * \sum_{i=0}^{i=\log_3(n)} \frac{1}{3^i})$$

Por lo tanto:

$$T(n) \in \Theta(n^2)$$

Ejercicio 2 (10 puntos)

a) Ver teórico.

b) El problema planteado es análogo al *Problema de la Mochila (Knapsack Problem)* visto en el curso, donde la capacidad de la mochila sería equivalente a los *M* pesos del Albañil. Los *n* objetos disponibles son en este caso, los *N* tipos de cuerdas y la ganancia sería equivalente al largo de la cuerda. Además, se tiene la misma función objetivo a maximizar.

Para resolver el problema, se utiliza la misma estrategia que en el KP, se busca incluir en cada etapa, el tipo de cuerda no incluido aún, que provea la mayor ganancia por metro (i.e. menor precio por metro).

```
void albanil(float M, int n, float* metros_disponibles, float* costo_por_metro, float *solucion) {
	int i;
	for (i =0;i<n;i++)
		solucion[i]=0;
	/*
	 * @Sort:
	 * Ordena los valores del arreglo en forma creciente.
	 * Retorna un vector con los índices que tiene en el vector
	 * original. Es decir, los más baratos primero.
	 */
	int * indice = sort(costo_por_metro);
	i = 0;
	while ((M > 0) && (i < N)) {
		int metros_posibles = M / costo_por_metro[indice[i]];
		if (metros_posibles <= metros_disponibles[indice[i]]) {
			solucion[indice[i]] = metros_posibles;
			M = 0;
		} else {
			solucion[indice[i]] = metros_disponibles[indice[i]];
			M = M - (costo_por_metro[indice[i]] * metros_disponibles[indice[i]]);
		}
		i++;
	}
}
```

Ejercicio 3 (10 puntos)

a) Sea $p_1,p_2,...,p_n$ una solución óptima al problema de construir una torre de altura N. Si encontramos una solución $r_2,r_3,...,r_k$ con menos piezas que $p_2,p_3,...,p_n$ (k<n) para una torre de altura N-h[p_1], entonces $p_1,r_2,r_3,...,r_k$ sería una solución mejor que la óptima original, lo cual es absurdo. Por lo tanto, el principio de optimalidad aplica a este problema.

b)

$$f[0][j] = \infty,\ \forall j\ :\ 0 \le j \le N$$
$$f[i][0] = 0,\ \forall i\ :\ 0 < i \le M$$

$$f[i][j] = \begin{cases} min(f[i-1][j], 1 + f[i][j-h[i]]), & \text{si } h[i] \le j \\ f[i-1][j], & \text{si } h[i] > j \end{cases}$$

La función debe invocarse de esta forma: f[M][N].

Ejercicio 4 (10 puntos)

a)

```
void Swap(int &a, int&b){
      int tmp = a;
      a = b;
      b = tmp;
}

void QuickSort(int *T, int ini, int fin) {
if (ini<fin){
      int posPiv = elegirPivote(T, ini, fin);
      Swap (T[ini],T[posPiv]);
      posPiv = ini;
      for (int i=ini+1; i<= fin; i++){
            if (T[i] < T[ini]){
                  posPiv = posPiv +1;
                  Swap(T[posPiv], T[i]);
            }
      }
      //mueve pivote a lugar final
      Swap(T[posPiv], T[ini]);
      QuickSort (T, ini, posPiv-1);
      QuickSort (T, posPiv+1, fin);
 }
}
```

b)

- ⇒ Divide & Conquer – Hard to split Easy to Join.
- ⇒ Peor caso: O(n^2) -- Caso Promedio: $\Theta(n * \log(n))$
- ⇒ Depende del reordenado de los elementos repetidos, usualmente ligado a la operación de comparación y/o selección del pivote debiendo asegurar que se mantenga el orden relativo de dichos elementos previo y luego de su ejecución para considerarse estable.

Problemas

Problema 1 (30 puntos)

a) Describa brevemente como se podría determinar el diámetro de G mediante BFS.

El diámetro del grafo es la mayor de las longitudes de los caminos más cortos entre todo par de vértices. Para cada vértice dado, se utiliza BFS para determinar los caminos más cortos y sus longitudes (niveles del árbol) entre éste vértice y los restantes. Dado que solo interesa saber la longitud máxima de los caminos más cortos, para cada recorrida BFS de un vértice dado se informa el largo máximo de los caminos más cortos (es decir el nivel máximo). Luego se determina el diámetro como la máxima de las longitudes de dichos caminos. Para el caso de grafo no-conexos el diámetro no está definido (o podría decirse que es infinito).

b) Implemente un algoritmo en C* que lo resuelva según lo descrito en a).

```
int BFS_MaxNivel(Grafo g, int s) {
    int max_nivel = 0;         // Máximo nivel del árbol del recorrido.
    int procesado[N] = {0};    // Marcas de procesado
    int nivel[N] = {MAX_INT};  // Niveles de los vértices según árbol,
           // inicializados en infinito para el caso de grafo no-conexo.

    Lista cola = CrearLista(); // Cola de vértices a procesar
    procesado[s] = 1;
    nivel[s] = 0;
    ListaInsertarUltimo(cola, s);
    while (!ListaEsVacia(cola)) {
        int v = listaPrimero(cola); listaSacar(cola);
        ady = adyacentesNodoGrafo(g, v);
        while (!ListaEsVacia(ady)) { // Control de adyacentes
            int w = listaPrimero(ady); listaSacar(ady);
            if (!procesado[w]) {
                ListaInsertarUltimo(cola, w);
                procesado[w] = 1;
                nivel[w] = nivel[v] + 1;
            }
        }
    }
    for (int v = 0; v < N; v++)
        if (nivel[v] > max_nivel)
            max_nivel = nivel[v];

    return max_nivel;
}

int Diametro(Grafo g) {
    int diametro = 0;

    for (int v = 0; v < N; v++) {
        int nivel = BFS_MaxNivel(g, v);
        if (nivel > diametro)
            diametro = nivel;
    }
    return diametro;
}
```

Problema 2 (30 puntos)

a) En el caso que no existan valores predefinidos siempre se puede completar la casilla utilizando los dígitos del 1 al n^2 para la fila inicial y luego desplazando (*shift*) n lugares para cada una de las siguientes n-1 filas. Cada n filas se repite el procedimiento incrementando en 1 el valor con el cual se comienza. (e.g. En el ejemplo n=3 se comienza con el valor 2 para la fila 3).

```
SudokuSinR(int** matriz,int n){
        int[][] matriz = new int[n*n][n*n];
        for (int i = 0; i < n*n; i++)
          for (int j = 0; j < n*n; j++)
                    matriz [i][j] = (i*n + i/n + j) % (n*n) + 1;
        }
```

b) **Formalización**

Forma de la solución
Tupla de largo fijo $N^2=n^2xn^2$ de la forma $S = <e_{00}, e_{01}, e_{02} \dots e_{0\ n^2-1}, e_{10}, e_{11}, e_{12} \dots e_{n^2-1\ n^2-1}>$ donde e_{ij} representa valor de la celda ubicada en la fila *i* y columna *j*, con $0 <= i, j < n^2$.

Sean:

- f_i el conjunto de elementos de S tal que $f_i = \{e_{i\ 0}, e_{i\ 1} \dots e_{i\ n^2-1}\}$ (Fila); con $0 <= i, j < n^2$.
- c_i el conjunto de elementos de S tal que $c_i = \{e_{0\ i}, e_{1\ i} \dots e_{n^2-1\ i}\}$ (Columna); con $0 <= i, j < n^2$.

Restricciones explícitas

- Todos los elementos de la tupla pertenecen al conjunto [1..n^2], $e_{i\ j} \in [1..n^2], \forall\, i, j / 0 \le i, j < n^2$
- Todos los valores predefinidos k_{ij} se encuentran asignados según corresponde $e_{ij} = k_{ij}$

Restricciones implícitas

- No hay elementos repetidos en f_i 0 <= i < n, $e_{ij} \neq e_{iz}, \forall e_{ij}, e_{iz} \in f_i \wedge j \neq z$
- No hay elementos repetidos en c_i, 0 <= i < n $e_{j_i} \neq e_{z\,i}\ , \forall e_{ji}, e_{zi} \in c_i \wedge j \neq z$
- No hay elementos repetidos en cada submatriz nxn. $e_a \neq e_b, \forall e_a, e_b \in toVector(i), 0 <= i < n^2 - 1 \wedge a \neq b$

c) **Implementación**

```
//funciones auxiliares que verifican las restricciones

bool RepEnFila(int**matriz, int fila, int columna, int size){
    bool encontreRepetido = false;
    int j=0;
    while (!encontreRepetido || j<size){
         if ((matriz[fila][columna]==matriz[fila][j]) && (columna!=j))
            encontreRepetido = true;
         j++;
         };
    return encontreRepetido;
    }

 bool RepEnColumna(int**matriz, int fila, int columna, int size){
     bool encontreRepetido = false;
     int i=0;
     while (!encontreRepetido || i<size){
          if ((matriz[fila][columna]==matriz[i][columna]) && (fila!=i))
             encontreRepetido = true;
          i++;
          };
    return encontreRepetido;
    }

 bool RepEnSubMatriz(int**matriz, int fila, int columna, int size){
     bool encontreRepetido = false;
     int iterVector=0;
     int iter;
     int[] vector = new int[size];
     int nroSubMatriz = (fila/size)*size + (columna/size); // este calculo tb es un poco complejo
     vector = ToVector(nroSubMatriz);
```

```
     while (!encontreRepetido || iterVector<size){
          iter =0;
          while (!encontreRepetido || iter<size){
               if ((vector[iterVector]==vector[iter]) && (iterVector != iter))
                  encontreRepetido = true;
               iter++;
               };
          iterVector++;
          };
     return encontreRepetido;
     }

//procedimiento de backtracking

void Backtracking(int**matriz,int fila, int columna, int size, bool&encontre){
    if (!RepEnFila(matriz, fila, columna, size) && !RepEnColumna(matriz, fila, columna, size)
        && !RepEnSubMatriz(matriz, fila, columna, size)){
            if ((fila==size*size -1) && (columna== size*size -1))
                encontre = true;
            else{
                int valores =1;
                while (valores<size && !encontre){
                    if (columna<size*size -1){
                       matriz[fila][columna+1]=valores;
                       Backtracking(matriz,fila,columna+1,size,encontre);
                       }
                    else{
                       matriz[fila+1][0]=valores;
                       Backtracking(matriz,fila+1,0,size,encontre);
                       }
                    valores++;
                    };
                }
            }
    }
//función pedida
bool sudoku(int**matriz, int n){
    bool encontre = false;
    Backtracking(matriz,0,0,n*n,encontre);
    return encontre;
    }
```