Algoritmos y Estructuras de Datos Avanzadas

Análisis Matemático de Algoritmos no Recursivos

Prof.: Dr. Pedro A. Rodríguez[1]

[1]Departamento de Sistemas de Información
Departamento de Ciencias de la Computación y TI
Universidad del Bio-Bio

Búsqueda del mayor elemento de una lista

El algoritmo

- Ejemplo: búsqueda del mayor elemento en una lista de tamaño n.

Algorithm 1 maxElement(int *A*[], int n)

```
1: Input: A, arreglo de tamaño n; n, tamaño del arreglo A.
2: Output: retorna el máximo valor en A.
3: int maxval = A[0];
4: for ( i = 1; i < n; i++ ) do
5:     if ( A[i] > maxval ) then
6:         maxval = A[i];
7:     end if
8: end for
9: return maxval;
```

Búsqueda del mayor elemento de una lista

Análisis previo

- Tamaño de la entrada: n.
- ¿Necesita el algoritmo de espacio de almacenamiento adicional? ¡NO!
- Operación básica: la comparación.
- Se debe notar que el número de comparaciones es igual al tamaño del arreglo menos 1.
- No existe necesidad de distinguir entre el peor, mejor y caso promedio.

Búsqueda del mayor elemento de una lista

Análisis matemático

- Sea T(n) el número de veces que se ejecuta la comparación.
- $T(n) = \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n)$.
- Demostración:

$$\lim_{n\to\infty} \frac{n-1}{n} = \lim_{n\to\infty} \frac{n(1-\frac{1}{n})}{n} = \lim_{n\to\infty} (1 - \underbrace{\frac{1}{n}}_{0}) = 1$$

Como $1 \in \mathbb{R} \Rightarrow$ f(n) = n-1 $\in \Theta(n)$

Problema de la unicidad de los elementos

El algoritmo

- Chequear si todos elementos de un arreglo dado de tamaño *n* son distintos.

Algorithm 2 UniqueElements(int *A*[], int n)

```
 1: Input: A: arreglo de tamaño n.
 2: Output: retorna true si los elementos son distintos, false en caso contrario.
 3: for ( i = 0; i < n-1; i++ ) do
 4:     for ( j = i+1; j < n; j++ ) do
 5:         if ( A[i] == A[j] ) then
 6:             return false;
 7:         end if
 8:     end for
 9: end for
10: return true;
```

Problema de la unicidad de los elementos

Análisis previo

- Tamaño de la entrada: n.
- ¿Necesita el algoritmo de espacio de almacenamiento adicional? ¡NO!
- Operación básica: la comparación.
- Se debe notar que el número de comparaciones no solo depende de *n* sino que también si existen elementos iguales en el arreglo, y si existen, en cual posición, ¿en la segunda posición encontramos el primer elemento repetido, en la i-ésima posición, en la posición *n*?
- Realizaremos sólo el análisis para el peor caso.

Problema de la unicidad de los elementos

Análisis previo

- Por definición, la entrada para el peor caso es un arreglo para el cual el número de comparaciones de elementos es $C_{worst}(n)$ es el más grande entre todos los arreglos de tamaño n.
- Si revisamos con atención el loop más interno, éste nos muestra que existen dos tipos de entrada para el peor caso: (a) cuando no existen elementos repetidos; (b) cuando sólo los dos últimos elementos son iguales.
- Se realiza una comparación por cada repetición del loop más interno, es decir, por cada valor de la variable j entre los límites $i + 1$ y $n - 1$; éste se repite por cada valor de i (entre 0 y $n - 2$) del loop más externo.

- Revisemos las siguientes reglas de sumatorias:

$$\sum_{i=k}^{n} 1 = n - k + 1, \text{ con } k \leq n.$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + ... + n = \frac{n(n+1)}{2} \approx \in \Theta(n^2).$$

- De acuerdo a este análisis previo, tenemos lo siguiente:

$$\mathrm{T(n)} = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2$$

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

Producto de matrices

El algoritmo

- Dadas dos matrices A y B de n$\times$n, calcular la complejidad temporal del producto C = AB.

Algorithm 3 MatrixMultiplication(int A[][], int B[][], int n)

```
1: Input: A, B: matrices de tamaño n×n.
2: Output: C: matriz resultante de tamaño n×n
3: for ( i = 0; i < n; i++ ) do
4:     for ( j = 0; j < n; j++ ) do
5:         C[i][j] = 0.0;
6:         for ( k = 0; k < n; k++ ) do
7:             C[i][j] = C[i][j] + A[i][k] * B[k][j];
8:         end for
9:     end for
10: end for
```

Producto de matrices

Análisis previo

- Tamaño de la entrada: $2n^2$.
- ¿Necesita el algoritmo de espacio de almacenamiento adicional? ¡NO!
- Operación básica: el producto (línea 7).
- El conteo de la operación básica depende sólo del tamaño de las matrices de entrada, entonces no es necesario analizar por separado los tres casos.
- Existe sólo una multiplicación que se ejecuta en cada iteración del loop más interno (de 0 hasta n-1).

- Para el loop más interno, el número de multiplicaciones realizadas por cada par de valores de las variables i y j:

$$\sum_{k=0}^{n-1} 1 = \text{n}.$$

- El número total de multiplicaciones $T(n)$ está dado por la siguiente expresión:

$$\text{T(n)} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

$$T(n) = n^3$$

$$\therefore T(n) \in \Theta(n^3)$$

Plan general para analizar algoritmos no recursivos

1. Identificar el parámetro que indica el tamaño de la entrada.
2. Identificar la operación básica del algoritmo.
3. Chequear si el número de veces que se ejecuta la operación básica depende sólo del tamaño de la entrada. Si también depende de alguna propiedad adicional, entonces analizar sólo el peor caso.
4. Definir la expresión que refleje la suma del número de veces que la operación básica se ejecuta, a través de alguna sumatoria o una ecuación de recurrencia.
5. Establecer la complejidad del algoritmo (orden de crecimiento).

Ordenamiento por conteo

El algoritmo

- Por cada elemento de una lista contar el número total de elementos menores y registrar el resultado en una tabla. Los números de esta tabla indicarán las posiciones de los elementos en la lista ordenada.

Table: Ejemplo de ordenamiento por conteo

A[7]		58	25	74	67	101	89	15
	count[7]	0	0	0	0	0	0	0
i=0	count[7]	**2**	0	1	1	1	1	0
i=1	count[7]	0	**1**	2	2	2	2	0
i=2	count[7]	0	0	**4**	2	3	3	0
i=3	count[7]	0	0	0	**3**	4	4	0
i=4	count[7]	0	0	0	0	**6**	4	0
i=5	count[7]	0	0	0	0	0	**5**	0
	count[7]	2	1	4	3	6	5	**0**
F[7]	count[7]	15	25	58	67	74	89	101

Ordenamiento por conteo

El algoritmo

Algorithm 4 CountingSort(int A[], int n)

```
 1: Input: A: arreglo de elementos entero; n: tamaño del arreglo
 2: Output: arreglo A ordenado ascendentemente.
 3: for (i=0; i<n; i++) do
 4:     count[i] = 0;
 5: end for
 6: for (i=0; i<n-1; i++) do
 7:     for (j=i+1; j<n; j++) do
 8:         if ( A[i] < A[j] ) then
 9:             count[j] = count[j] + 1;
10:         else
11:             count[i] = count[i] + 1;
12:         end if
13:     end for
14: end for
15: for (i=0; i<n; i++) do
16:     F[count[i]] = A[i];
17: end for
18: return F;
```

Ordenamiento por conteo

Análisis previo

- Tamaño de la entrada: n (tamaño del arreglo).
- ¿Necesita el algoritmo de espacio de almacenamiento adicional? Si, arreglo count[].
- Operación básica: la comparación (línea 8).
- El conteo de la operación básica depende sólo del tamaño del arreglo.

Ordenamiento por conteo

Análisis matemático

- ¿Cuál es la eficiencia del algoritmo?. Como en alguno de los ejemplos anteriores vistos, este algoritmo debería ser cuadrático.

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$T(n) = \frac{n(n-1)}{2}$$

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