2.1 Divide y Vencerás

Recordando un poco de historia, este término, oración o pensamiento viene desde los romanos cuando conquistaron Italia, Divide et impera, esta frase se dice de dudosa procedencia, sim embargo se atribuye al emperador romano Julio Cesar, el cual resume la estrategia de este con los gobernantes de ponerlos los unos contra los otros, como una estrategia política para dividir y vencer, consiguiendo objetivos como desviar la atención, evitar que vean nuestros errores o defectos, hacer que peleen entre ellos, etc. Pero ¿Por qué incluir el tema en la parte algorítmica?, pues conviene por las siguientes razones:,

1. Los problemas se pueden dividir en subproblemas de tamaño menor al original.
2. Los subproblemas no se solapan.
3. Los subproblemas se pueden resolver de forma independiente
4. La solución de problema inicial se puede obtener combinando las soluciones de cada subproblema.
5. Es importante determinar cuándo un problema se puede solucionar mediante este paradigma

Esquema del paradigma

dividirYVencer(p:Problema)

dividir($p_1, p_2, p_3, \ldots\ldots, p_k$)

para i = 1, 2, 3, …. , k

$s_i = resolver(p_i)$

$solucion = combinar(s_1, s_2, s_3, \ldots., s_k)$

Si el problema es pequeño, se puede resolver de forma directa, en caso contrario se recomienda emplear recursividad.

Requisitos para emplear el esquema

- El problema original deberá poderse dividir (fácilmente) en subproblemas del mismo tipo que el original, pero con una solución más simple o sencilla
- La solución de un subproblema deberá obtenerse independiente a la de los demás.
- Se requiere de un método directo para resolver el problema en problemas más pequeños.
- Se necesita un método para combinar las soluciones de cada subproblema.

Esquema recursivo del paradigma (2 subproblemas)

dividirYVencer(p,q:indice)

var m:índice

si esPequenio(p,q)

solución = resolverDirecto(p,q)

en otro caso

m = dividir(p, q)

$$solución = combinar(dividirYVencer(p, m), dividirYVencer(m + 1, q))$$

Donde las funciones (genéricas) son:

esPequenio() Determina cuando es problema es pequeño para aplicar la solución directa.

resolverDirecto() Método alternativo de solución para problemas pequeños.

combinar() Método que reúne las soluciones de los problemas pequeños.

Análisis de la complejidad del paradigma

- Como se pudo ver, este paradigma puede producir algoritmos recursivos cuyo tiempo de ejecución se puede expresar mediante una ecuación de recurrencia como la siguiente:

$$T(n) = \begin{cases} cn^k, & si\ 1 \le n < b \\ aT(n/b) + cn^k, & si\ n \ge b \end{cases}$$

- Donde
  - **a**, **c** y **k** son números reales
  - **n** y **b** son números naturales
  - **a**>0, **c**>0, **k**>=0 y **b**>1
  - **a** representa el número de subproblemas
  - **n** / **b** es el tamaño de cada subproblema
  - cn^k representa el costo de descomponer el problema en subproblemas sumado la combinación de estas para resolver el problema.
- La solución a un problema puede alcanzar distintas complejidades

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k \log(n)) & si\ a = b^k \\ \Theta\left(n^{log_b(c)}\right) & si\ a > b^k \end{cases}$$

- Las diferencias surgen de los distintos valores que pueden tomar $\boldsymbol{a}$ y $\boldsymbol{b}$, que en definitiva determinan el número de subproblemas y su tamaño. Lo importante es observar que en todos los casos la complejidad distinta que puede ser de orden polinómico o poli logarítmico, pero nunca exponencial.

Ejemplo

Encontrar los números máximo y mínimo de un conjunto de números enteros empleando el paradigma.

```
encontrarMaxMinDividirYVencer( i, j : integer; var max, min : tipo)
    si i<j-1
        // Dividir el problema en 2 subproblemas
        mitad = (i+j) div 2
        encontrarMaxMinDividirYVencer ( i, mitad, max1, min1 )
        encontrarMaxMinDividirYVencer( mitad+1, j, max2, min2 )
```

```
        // Combinar
        si max1>max2
                max = max1
        en otro caso
                max = max2
        si min1<min2
                min = min1
        en otro caso
                Min =Min2
        // Caso base
en otro caso
si i == j-1
        si A[i]>A[j]
                max = A[i];
                min = A[j]
        en otro caso
                max = A[j];
                min = A[i]
en otro caso
                max = A[i];
                min =Max
```

Donde

Operación básica: Asignaciones a max, min, max1, min1, max2 y min2.

Complejidad temporal Peor caso: Los números están hasta el final (el máximo y el mínimo).

$$T(n) = \begin{cases} 2n^k, & si\ 1 \leq n < 2 \\ 2T(n/2) + 2, & si\ n \geq 2 \end{cases}$$

Algunos ejemplos de éxito al aplicar el paradigma son:

- Búsqueda binaria.
- Ordenación mediante mezcla (mergesort).
- Ordenación rápida (quicksort).
- Transformada rápida de Fourier.

Programación Orientada a Objetos

La algoritmia y la programación orientada a objetos se llevan bien, en la siguiente clase programada en Java muestra el algoritmo de búsqueda binaria.

```
/**
 * Implementacion del algoritmo de busqueda binaria
 * @author Saul De La O Torres
 */
public class BusquedaBinaria {
    /**
     * Busca el x si esta presente y devuelve el indice del elemento si esta
     * en el arreglo, funcion recursiva.
     *
     * @param arr arreglo sobre el cual se busca
     * @param l
     * @param r
     * @param x el entero que se busca
     * @return devuelve el indice del elemento buscado
     */
    public int buscarBinariamente(int [] arr, int l, int r, int x) {
        // condicion de parada
        if( r>=l ) {
            int mitad = l + (r - l) / 2;

            // Si el elemento esta presente a la mitad del arreglo
            if (arr[mitad] == x) {
                return mitad;
                }
            // Si el elemento es mas pequenio que la mitad, entonces
            // estara presente en el subarreglo izquierdo
            if (arr[mitad] > x) {
                return buscarBinariamente(arr, l, mitad - 1, x);
                }
            // En caso contrario el elemento estara presente
            // en el subarreglo derecho
            return buscarBinariamente(arr, mitad + 1, r, x);
        }
        // El elemento no esta presente en el arreglo
        return -1;
    }
    /**
     * Busca iterativamente
     *
     * @param arr el arreglo
     * @param x el elemento a buscar
     * @return devuelve el indice del elemento buscado, -1 si no esta
     */
    int buscarBinariamenteIterativo(int arr[], int x) {
        int l = 0, r = arr.length - 1;
        while (l <= r) {
            // calcula la mitad
            int mitad = l + (r - l) / 2;
            // Check if x is present at mid
            if (arr[mitad] == x) {
```

```
            return mitad;
            }
         // si x es mas grande ignora la mitad derecha
         if( arr[mitad]<x ) {
            l = mitad + 1;
            }
         else { // si x mas paquenio, ignora la mitad derecha
            r = mitad - 1;
            }
         }
      // Si el elemento no esta en el arreglo
      return -1;
   }
}
```

Tarea.

Implementar una clase Java que:

1. Contenga una función que le pida al usuario el número de elementos del arreglo (entero).
2. Contenga un método que genere el doble de números enteros aleatorios entre el limite negativo y positivo que el usuario introdujo en el punto anterior, es decir, si introdujo 100 entonces el método generara números entre -100 y 100 para llenar el arreglo.
3. Un método que le pida al usuario el número a buscar.
4. Un método que calcule el tiempo de búsqueda binaria
5. Un método que mande llamar al método de búsqueda iterativa y otro al método de búsqueda recursiva.
6. Cree un pequeño reporte para incluir sus conclusiones, y una gráfica de tiempo contra el número de elementos en el arreglo 10, 100, 1000, 10000, 100000, 1000000 (una curva para el iterativo y otra para el recursivo).
7. Cree su clase aparte para el método main.

La tarea deberá ser enviada al correo sdelaot@gmail.com.