Marzo 15, 2021

Clase 04

Algoritmos Recursivos

.- Un algoritmo recursivo debe involucrar la evaluación de una condición que divida la ejecución en dos o más casos.

1.- Caso base: permite la terminación de las invocaciones recursivas y por tanto, del algoritmo.

2.- Caso recursivo: Maneja al menos una invocación a sí mismo. Las continuas invocaciones del algoritmo deben hacerse variando al menos un parámetro de modo que en algún momento se alcance el caso base

Relaciones de recorrencia.

.- Para analizar el tiempo de ejecución de un algoritmo recursivo y por tanto, su complejidad, necesitamos una técnica diferente a los que conocemos para trabajar con algoritmos que no lo son. Si queremos determinar la función $T(n)$ del tiempo de ejecución de un algoritmo, como éste se invoca a sí mismo con otros términos, esto debe reflejarse con ecuaciones en donde la función esté dada también en términos de sí misma, es decir, se establecen igualdades donde la función aparece de ambos lados de la igualdad con otros parámetros.

A este conjunto de ecuaciones que caracterizan el tiempo de ejecución de un algoritmo recursivo se le conoce como relación de recurrencia.

Ejemplo: Dado un real x y un entero n no negativo calcular x^n.

Solución:

- Tenemos que la potencia 0 (cero) de cualquier número real es 1. Para definir recursivamente la potencia x^n ("x elevado a la potencia n") necesitamos que la función reciba dos parámetros y al menos uno de ellos debe variar para poder definir la potencia en términos de sí misma.

Una forma de hacer esto en el caso general para potencias positivas, es manteniendo la base y reduciendo la potencia en uno, esto es, poniendo como segundo argumento $n-1$, es decir, estamos simplificando el cálculo de multiplicar x consigo mismo n veces a una multiplicación de x con el resultado de multiplicar x consigo mismo $n-1$ veces, ya que $x^n = x \cdot x^{n-1}$

De esta manera, empleamos la recursión para definir el cálculo de la potencia $f(x,n) = x^n$ de la siguiente manera:

$$f(x,n) = \begin{cases} 1 & \text{si } n=0 \\ x * f(x, n-1) & \text{si } n>0 \end{cases}$$

.- Tenemos la implementación en Java.

```
/**
 * Devuelve el valor de x elevado a la pot. n
 * @param  x la base
 * @param  n la potencia
 * @return  El valor de x elevado a la potencia n.
 * @throws  IllegalArgumentException si n es negativo.
*/
public static double pot (double x, int n) {
    if (n < 0) {
      throw new IllegalArgumentException ("La potencia
                                    no debe ser negativa");
    }
    return rPot (x, n);
}

/**
 * Devuelve el valor de x elevado a la potencia n en
 * tiempo lineal sobre n.
*/
private static double rPot (double x, int n) {
    if (n==0) {
      return 1;
    }
    return x * rPot (x, n-1);
}
```

.- Ahora, para el algoritmo pot (potencia), la relación de recurrencia es la siguiente:

* Relación de recurrencia:

$$T(0) = c$$

$$T(1) = c$$

$$T(n) = T(n-1) + c$$

Se toma un valor de c que sea el máximo de operaciones elementales que se llevan a cabo en los casos base y el recursivo. El peor caso tiene una impresión de un factor constante respecto al tiempo de ejecución real en una computadora

Si desglosamos más esta relación tenemos lo siguiente:

$$T(n) = T(n-1) + c$$
$$T(n-1) = T(n-2) + c$$
$$\cdots = \cdots$$
$$T(2) = T(1) + c$$
$$T(1) = T(0) + c$$
$$T(0) = c$$

Sumando ambos lados de las igualdades:

$$\sum_{i=0}^{n} T(i) = \sum_{i=0}^{n-1} T(i) + \sum_{i=0}^{n} c = \sum_{i=0}^{n-1} T(i) + c(n+1)$$

por otro lado:

$$\sum_{i=0}^{n} T(i) = \sum_{i=0}^{n-1} T(i) + T(n) \quad \text{de donde } T(n) = c(n+1)$$

$= cn + c$, lo cual es de complejidad $O(n)$ //

*** Ejemplo 2:** versión 2 del algoritmo potencia.

```
int pot2 (int b , int e){
    if (e==0){
        return 1;
    }else if ( e % 2 ==0){
        return pot2 (b*b, e/2);
    }else
        return b * pot2 (b*b ,e/2);
}
```

*** Solución:**

Supongamos que $n = 2^x$ es una potencia de dos. (esto es válido ya que por ahora estamos analizando la complejidad asintótica).

.. Ahora, de esta manera, empleamos la función de recursión para definir el cálculo de la potencia:

$$T(n) = \begin{cases} 3 & \text{si } n = 0 \\ 8 + T(n/2) & \text{si } n > 0 \text{ y } n \text{ es par} \\ 9 + T((n-1/2)) = 9 + T\left(\frac{n-1}{2}\right) & \text{si } n > 0 \text{ y } n \text{ es impar.} \end{cases}$$

Asumiendo que $n = 2^x$ es una potencia de dos tenemos:

$$\begin{aligned} T(n) &= 8 + T(n/2) \\ &= 8 + 8 + T(n/4) = 8*2 + T(n/2^2) \\ &= 8+8+8 + T(n/8) = 8*3 + T(n/2^3) \\ &\vdots \\ &= 8i + T(n/2^i) \end{aligned}$$

¿Cuándo se llega al caso base T(0)?

→ el parámetro de la función T es entero

→ i tiende a infinito pero esto no tendría sentido por lo que T(1) tendríamos:

$$\frac{n}{2^i} = 1 \quad \text{, es decir, cuando } i = \log_2 n$$

Sustituyendo

$$T(n) = 8\log_2 n + T(1) = 8\log_2 n + 9 + T(0) = 8\log_2 n + 9 + 3$$

$$= 8\log_2 n + 12 \in O(\log n)$$

** Pasos para el análisis de algoritmos recursivos **

1.- Decidir acerca del parámetro n que indica el tamaño de la entrada del algoritmo.

2.- Identificar la operación básica del algoritmo

3.- Determinar si el número de veces que la operación básica es ejecutada puede variar para diferentes entradas del mismo tamaño n (analizar los casos peor, promedio y mejor si existen).

4.- Expresar como una relación de recorrencia, con una condición inicial adecuada, el número total de ejecuciones de la operación básica.

5.- Resolver la relación de recurrencia (o al menos establecer el orden de crecimiento de su solución), es decir, encontrar una fórmula explícita para el término genérico que satisfaga la recurrencia y la condición inicial ó probar que no existe.

*Ejemplo: Algoritmo factorial recursivo.

```
factorialRec (int n) {
    if (n==0)
        return 1;  // caso base
    else
        return factorialRec(n-1) * n;
}
```

→ Siguiendo los pasos para el análisis tenemos:

1.- Decidir acerca del parámetro n que indica el tamaño de la entrada del algoritmo:

n será el número del cual se calculará el factorial

2.- Identificar la op. básica del algoritmo:

la multiplicación, denotamos como $f(n)$

3.- Determinamos si el número de veces que la operación básica es ejecutada puede variar para diferentes entradas del mismo tamaño n's

no existen variaciones.

4.- Expresar como una (o) relación de recurrencia;

Sea $f(n) = \underbrace{f(n-1)}_{\text{factorialRec}(n-1)} + \underbrace{1}_{\text{factorialRec}(n-1) * n}$ para $n > 0$

$f(0) = d = 1 \rightarrow$ condición inicial (caso i) una constante.

∴ Por lo tanto tenemos:

$$T(n) = \begin{cases} d = 1 & \text{si } n \leq 0 \\ T(n-1) + c, & c \text{ una constante si } n > 0 \end{cases}$$

$T(n) = f(n)$, $T(n-1) + c = f(n)$

∴- Para $n > 0$

$f(n) = f(n-1) + c = f(n-1) + 1$

∴- Para $n > 2$, sustituimos $f(n-1) = f(n-2) + 1$

$f(n) = [f(n-2) + 1] + 1 = f(n-2) + 2$

∴- Para $n > 3$, sustituimos $f(n-2) = f(n-3) + 1$

$f(n) = [f(n-3) + 1] + 2 = f(n-3) + 3$

Podemos determinar un patrón: M

$f(n) = f(n-i) + i$

Tomando en cuenta la condición inicial, tenemos que se sustituye $i=n$ en la fórmula anterior se obtiene:

$$f(n) = f(n-i) + i = f(n-n) + n$$
$$= n$$

$\therefore$ el algoritmo es lineal. //