TAREA #6 RECURSIVIDAD

CINTHIA GUADALUPE OLIVAS CALDERON NO.17212165

Es una técnica de programación en la cual un método puede llamarse a sí mismo, en la mayoría de casos un algoritmo iterativo es más eficiente que uno recursivo si de recursos de la computadora se trata, pero un algoritmo recursivo en muchos casos permite realizar problemas muy complejos de una manera más sencilla.

Un procedimiento o función se dice que es recursivo si durante su ejecución se invoca directa o indirectamente a sí mismo. Esta invocación depende al menos de una condición que actúa como condición de corte que provoca la finalización de la recursión.

Un algoritmo recursivo consta de:

1. Al menos un caso trivial o base, es decir, que no vuelva a invocarse y que se

activa cuando se cumple cierta condición.

2. El caso general que es el que vuelve a invocar al algoritmo con

un caso más pequeño del mismo.

Los lenguajes que soportan recursividad, dan al programador una herramienta poderosa para resolver ciertos tipos de problemas reduciendo la complejidad u ocultando los detalles del problema. La recursión es un medio particularmente poderoso en las definiciones matemáticas.

Ventajas de la Recursión

- Soluciones simples, claras.
- Soluciones elegantes.
- Soluciones a problemas complejos.

Desventajas de la Recursión: INEFICIENCIA

- Sobrecarga asociada con las llamadas a subalgoritmos.
- Una simple llamada puede generar un gran número de llamadas recursivas. (Fact(n) genera n llamadas recursivas).
- El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa.
- La ineficiencia inherente de algunos algoritmos recursivos.

Reglas de la recursividad:

Para que un problema pueda resolverse de forma recursiva debe cumplir las siguientes 3 reglas:

Regla 1: Por lo menos debe tener un caso base y una parte recursiva.

Regla 2: Toda parte recursiva debe tender a un caso base.

Regla 3: El trabajo nunca se debe duplicar resolviendo el mismo ejemplar de un problema en llamadas recursivas separadas.

Ejemplo: Calcular el factorial de un número.

FACTORIAL DE UN NÚMERO N

8! = 8 * 7!

7! = 7 * 6!

6! = 6 * 5!

En general,

n! = n * (n-1)!

Veamos un caso particular, calculemos el factorial de 5 (5!):

factorial de 5 = 5 * 4! ——–> “factorial de 5 es igual 5 multiplicado por factorial de 4”

factorial de 4 = 4 * 3! ——–> “factorial de 4 es igual 4 multiplicado por factorial de 3”

factorial de 3 = 3 * 2! ——–> “factorial de 3 es igual 3 multiplicado por factorial de 2”

factorial de 2 = 2 * 1! ——–> “factorial de 2 es igual 2 multiplicado por factorial de 1”

factorial de 1 = 1 ——–> “factorial de 1 es 1” ——> “caso base”

Una implementación en java seria:

```
public long factorial (int n) {
if (n == 0 || n==1) //Caso Base
return 1;
else
return n * factorial (n - 1); //Parte Recursiva
}
```

Complejidad de un Algoritmo Recursivo

Método del árbol de recursión

Existen varios métodos para calcular la complejidad computacional de algoritmos recursivos. Uno de me los métodos más simples es el árbol de recursión, el cual es adecuado para visualizar que pasa cuando una recurrencia es desarrollada. Un árbol de recursión se tiene en cuenta los siguientes elementos:

Nodo: Costo de un solo subproblema en alguna parte de la invocación recursiva.

Costo por Nivel: La suma de los costos de los nodos de cada nivel.

*Costo Total: Es la suma de todos los costos del árbol.

Ejemplo: Utilizando el método del árbol de recursión calcular la complejidad computacional del algoritmo recursivo del factorial. Lo primero es calcular las operaciones elementales de cada línea:

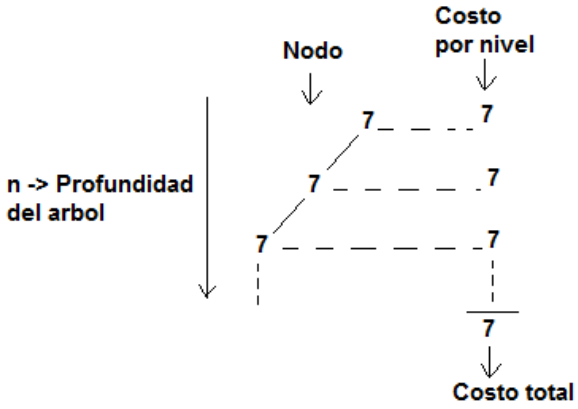
```
public long factorial (int n) { #operaciones elementales
if (n == 0 || n==1) //Caso Base 3
return 1; 1
else
return n * factorial (n - 1); //Parte Recursiva 4
}
```

$$T(n) = \begin{cases} 4 & \text{si } (n = 0) \text{ o } (n = 1) \\ 7 + T(n-1), & \text{Si } n>1 \end{cases}$$

Para hallar la complejidad se debe resolver esta recurrencia:

T(n) = 7 + T (n - 1)

El árbol de recursión es el siguiente:



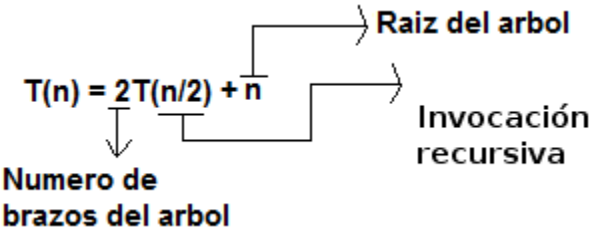
El costo total es la sumatoria de los costos de cada nivel:

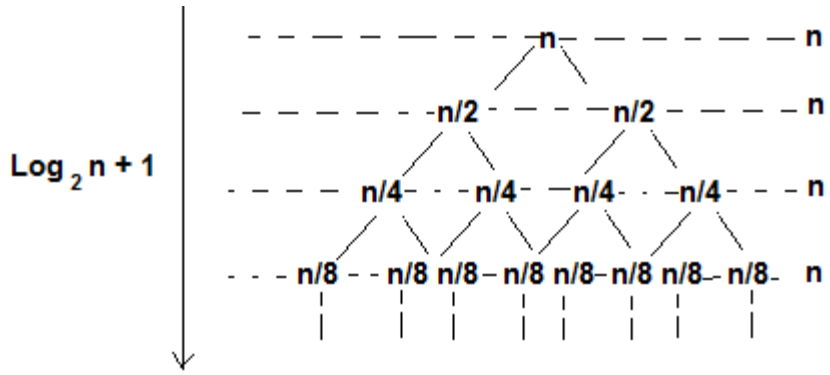
$\sum_{i=1}^{n} 7 = 7n\ O(n)$ Orden lineal

Ejemplo

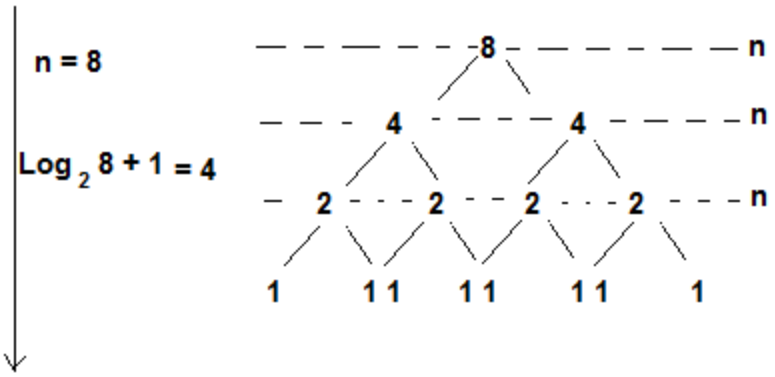
Utilizando el método del árbol de recursión, calcular la complejidad computacional de la siguiente recurrencia:

T(n) = 2 T(n/2) + n





Para entender mejor el árbol de recursión anterior, ilustramos cómo sería cuando n = 8:



Finalmente, la complejidad de la recurrencia está dada por la suma de los costos de cada nivel del árbol:

$$\sum_{i=1}^{log2\ n+1} n = n + n + n + n = nlog2(n+1) \quad O(nLog2(n))$$

REFERENCIAS

- Universidad Privada TELESUP. Recursividad . En Algorítmica y estructura de datos(130). Perú.
- José Fager W. Libardo Pantoja Yépez Marisol Villacrés Luz Andrea Páez Martínez Daniel Ochoa Ernesto Cuadros-Vargas. (2014). Introducción a las estructuras de datos. En Estructuras de datos(222). Latinoamérica: LATIn.
- Joyanes Aguilar, Luis (1996) Fundamentos de programación, Algoritmos y Estructura de datos. McGraw-Hill, México.