Recursividad y Eficiencia

Eficiencia

Definición

- La Eficiencia de un algoritmo o eficiencia algorítmica indica la cantidad de recursos que utiliza en un sistema de cómputo
- Métricas:
  - Complejidad Temporal
  - Complejidad Espacial

Métricas

- Complejidad Temporal. Se refiere al tiempo que le toma al algoritmo terminar su tarea.
- Complejidad Espacial. Relacionado con la cantidad de memoria necesaria.
  - Memoria del código fuente
  - Memoria de los datos utilizados

Notación O

- La Notación O (O Grande) representa la complejidad de un algoritmo en base a la cantidad de datos de entrada
- No mide la rapidez de un algoritmo, sino cuánto aumenta el tiempo de ejecución en razón del aumento de datos de entrada

Medidas de O

Notación	Nombre	Ejemplo
O(1)	Constante	Determinar si un número es par o impar
O(log n)	Logarítmica	Operaciones en árboles binarios balanceados
O(n)	Lineal / 1er orden	Buscar un elemento en un arreglo
O(n log n)	Lineal logarítmica	Ordenamiento *quick sort*
$O(n^2)$	Cuadrática / 2do orden	Ordenamiento por burbuja
O(n!)	Factorial	

Recursividad

Introducción

- La recursividad o recurrencia es uno de los conceptos fundamentales en matemáticas y computación
- La programación modular se basa en dividir un problema en bloques o módulos cada uno con una tarea en particular

Definiciones Formales

- Una definición recursiva dice como obtener conceptos nuevos utilizando el mismo concepto que intenta definir
- Un problema se divide en varias instancias del mismo problema, pero de tamaño menor hasta llegar a un punto en donde se conoce el resultado.

Definición

- Definición del Factorial
  - Factorial (n) = n * n-1 * n-2 * n-3 * …* 1 para n>0
  - Factorial (n) = 1 si n es igual a 0

- La definición Recursiva del Factorial es
  - Factorial (n) = 1 si n=0
  - Factorial (n) = n * Factorial (n-1) si n > 0

Ejemplo

- Así para calcular el factorial de 4 se tiene:
  - Factorial(4) = 4 * factorial(3) = 24
  - Factorial(3) = 3 * factorial(2) = 6
  - Factorial(2) = 2 * factorial(1) = 2
  - Factorial(1) = 1 * factorial(0) = 1
  - Factorial(0) = 1

Números de Fibonacci

- Los números de Fibonacci definen una secuencia infinita:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
- En dónde un término n es la suma de los términos anteriores n-1 y n-2
- Los términos F(0) y F(1) tienen los valores 0 y 1 respectivamente

Representación Recursiva

- La representación recursiva para obtener los números de Fibonacci es:
  - $F_N = F_{N-1} + F_{N-2}$ para $N > 2$
  - $F_0 = 0$
  - $F_1 = 1$

Ejemplo

- Fibonacci (5) es:
  - Fibonacci (5) = Fibonacci (4) + Fibonacci (3)
  - Fibonacci (4) = Fibonacci (3) + Fibonacci (2)
  - Fibonacci (3) = Fibonacci (2) + Fibonacci (1)
  - Fibonacci (2) = Fibonacci (1) + Fibonacci (0)
  - Fibonacci (1) = 1
  - Fibonacci (0) = 0

Programación Modular

- En un programa modular, sus módulos o funciones se invocan entre ellos dependiendo la tarea a realizar

Recursividad

- Considerando la existencia de módulos o funciones:
  - Una función es recursiva cuando se invoca a sí misma
- Se considera una alternativa al uso de estructuras cíclicas o de repetición

Razonamiento Recursivo

- Un razonamiento recursivo tiene dos partes:
  - Caso base (B)
  - Regla recursiva de construcción (R)
- Un conjunto de objetos esta definido recursivamente siempre que:
  - (B) Algunos elementos del conjunto se definan explícitamente
  - (R) El resto de los elementos se definan en términos de los ya definidos

Consideraciones

- La Regla recursiva de construcción (R) en algún momento debe permitir que se alcance el o los valores base (B)

Tipos de Recursividad

- Recursividad Directa. Cuando un procedimiento incluye una llamada a si mismo
- Recursividad Indirecta. Cuando en una secuencia de llamadas a métodos, se regresa al inicial u original
- Recursividad en Aumento. Cuando se crean operaciones diferidas que tienen que realizarse después de que se complete la ultima llamada recursiva
- Recursividad Simple. Solo se tiene una llamada recursiva en la función
- Recursividad Múltiple. Más de una llamada recursiva en el cuerpo de la función

Recursividad Directa

```
tipo nombre (int a, int b)
    inicio
        si (b = a)
            regresa a
        otro
            regresa nombre(a, b*5)
    fin
```

Recursividad Indirecta

```
tipo nombre (int a, int b)
    inicio
        si (b = a)
            regresa a
        otro
            regresa suma(a)
    fin
```

```
tipo suma (int a)
        inicio
            si (b = a)
                regresa a
            otro
                regresa nombre(a, a+1)
        fin
```

Recursividad en Aumento

```
tipo nombre (int a)
    inicio
        si (a = 3)
        regresa a
    otro
        regresa a + nombre(a-2)
    fin
```

Recursividad Simple

```
tipo nombre (int a)
    inicio
        si (a = 1)
            regresa a
        otro
            regresa nombre(a-1)
    fin
```

Recursividad Múltiple

```
tipo  nombre (int n, int m)
    inicio
        si m = 0
            regresa 1
    otro
        regresa nombre (n-1,m) + nombre (n-1,m-1)
    termina
```

Ventajas y Desventajas

- Ventajas:
  - Algunos problemas son más sencillos de modelar e implementar utilizando recursividad
- Desventajas:
  - Es necesario la creación de varias variables lo que puede ocasionar problemas en memoria
  - En general una función recursiva toma más tiempo en ejecutarse que una iterativa

Diseño Dividir para Vencer

- Un algoritmo recursivo es aquel que expresa la solución de un problema en términos de una llamada a si mismo.
- Después de cada llamada, el problema se reduce en tamaño hasta llegar a una solución trivial que es lo que se conoce como caso base.

Creación de Algoritmos Recursivos

- Las claves para crear un algoritmo recursivo para resolver un problema son:
  - Cada llamada recurrente se debe definir sobre un problema menor
  - Debe existir al menos un caso base para evitar recurrencia infinita
  - Se debe analizar en qué momento se realizará la llamada recursiva

Ejemplo

- El factorial de un numero n se define como n!
  - 1*2*3*4*5*6*…*n
- Considerar el caso cuando n=5
- En este caso:
  - 5! = 5 * 4!
  - 4! = 4 * 3!
  - 3! = 3 * 4!
- De esta forma se tiene:
  - n! = n * (n-1)!
- El caso base en el factorial por definición es:
  - 0! = 1

Algoritmo Recursivo

```
double factorial (entero n)
    inicio
    si(n=0)
        regresa 1
    otro
        regresa n * factorial (n-1)
    fin
```

Sumatoria Sencilla

```
int sumatoria (entero n)
    inicio
        si n<2
            regresa n
    otro
            regresa n + sumatoria (n-1)
    fin
```

Sumatoria Límite Inferior

```
int sumatoria (entero li, entero n)
    inicio
        si n=li
            regresa li
    otro
            regresa n + sumatoria (li,n-1)
    fin
```

Sumatoria Evaluada

```
int sumatoria (entero n)
    inicio
        si n=0
            suma ← 02+2(0)-3
        otro
        inicio
            suma ←n2+2n-3
            suma ← suma + sumatoria(n-1)
        fin
        regresa suma
    fin
```

Multiplicación

```
int multiplica (a,b)
    inicio
        si (b=0)
            regresa 0
        otro
            regresa a + multiplica(a,b-1)
    fin
```

Potencia

```
doble potencia (a,b)
    inicio
        si (b=0)
        regresa 1
    otro
        regresa a * potencia(a,b-1)
    fin
```

Combinaciones

- Cuantas combinaciones (n, m) de tamaño (m) pueden hacerse del tamaño total de un grupo (n)
- Definición:
  - Combinaciones ← g m=1
  - Combinaciones ← 1 m=g
  - Combinaciones ← combinaciones (n-1,m-1) + combinaciones(n-1,m)

Algoritmo

```
int combinaciones(int n, int m)
    inicio
        si m = 1
            comb ← n
        otro si m = n
            comb ← 1
        otro
            comb ← combinaciones(n-1,m-1)+ combinaciones(n-1,m)
    fin
```

Conversión a Binario

```
String binario (entero n)
    inicio
        si n ≥ 2
        inicio
            bin ←binario(n/2)
            bin ← bin + (n%2)
        fin
    otro
        bin ← bin + n
        regresa bin
    fin
```

Recorrer un Arreglo

```
void recorrer (arreglo, pos)
    inicio
        si pos = arreglo.longitud
        IMPRIMIR arreglo[pos]
    otro
        inicio
        IMPRIMIR arreglo[pos]
        recorrer (arreglo, pos -+1)
        fin
    fin

    recorrer(arreglo,0)
```

Recorrido Inverso

```
void recorrerInvertido (arreglo, pos)
    inicio
        si pos = 0
        IMPRIMIR arreglo[pos]
        otro
            inicio
                IMPRIMIR arreglo[pos]
                recorrerInvertido              (arreglo, pos - 1)
            fin
    fin
    recorrer(arreglo,arreglo.longitud-1)
```

Buscar un Elemento

```
void buscarElemento (arreglo, valor, pos)
    inicio
        si pos = arreglo.longitud
            IMPRIMIR "No encontrado"
        otro si val = arreglo[pos]
            IMPRIMIR "Elemento encontrado"
        otro
            buscarElemento (arreglo, valor,pos + 1)
    fin
```

buscarElemento(arreglo,valor, 0)

Menor de los Elementos

```
int menorElemento (arreglo, inicio, fin)
    inicio
        si inicio = fin
            regresa a[inicio]
        otro
            inicio
                menor ← menorElemento(a,inicio+1,fin)
                si (menor > a[inicio]
                    menor ← menor
                otro
                    menor ← a[inicio]
                regresa menor
            fin
    fin
    recorrer(arreglo,0, arreglo.longitud)
```

Suma de los Elementos

```
int sumatoria (arreglo, pos)
    inicio
        si pos = arreglo.longitud-1
            regresa arreglo-longitud-1
    otro
        regresa arreglo[pos]+sumatoria(arreglo,pos+1)
    fin
```