Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ingeniería en Inteligencia Artificial, Análisis y Diseño de Algoritmos
Sem: 2024-1, 3BV1, Práctica 3, 18 de octubre de 2023

Práctica 3: Funciones Recursivas vs Iterativas.

Catonga Tecla Daniel Isaí 1, Olguin Castillo Victor Manuel 2.

*daniel*9513*importantes*@*gmail.com*$_1$, *manuelevansipn*@*gmail.com*$_2$

Resumen:

Palabras Clave: Algoritmo, Big O, C++, Recursividad

1 Introducción

Texto ejemplo [1].

2 Conceptos Basicos

- **Algoritmo**. Un algoritmo es una secuencia de pasos lógicos que son precisos, ordenados y finitos que se ocupan para resolver un problema deseado[2].

- **Análisis de algoritmos** Es un proceso de evaluación donde conoceremos el rendimiento y la eficiencia de un algoritmo. Se evaluará el consumo de tiempo y de recursos computacionales que requiere el algoritmo para ser ejecutado con diversos datos de entrada, y esto determinará su complejidad[2].

- **Cota superior asintótica**. Es una función que delimita por la parte superior a otra función a medida que la entrada de la función delimitada crece.

- **Cota inferior asintótica**. Es una función que delimita por la parte inferior a otra función a medida que la entrada de la función delimitada crece.

- **Notacion O**. Esta notación se ocupa para describir la complejidad de un algoritmo definiendo una cota superior asintótica en el peor caso de ejecución de un algoritmo[2].

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\ c \text{ constante positiva y } n_0 \notin \mathbb{N} :$$
$$f(n) \leq cg(n),\ \forall\ n \geq n_0\}$$

- **Notacion Tetha "Θ"** Esta notación se ocupa para describir la complejidad de un algoritmo definiendo una cota superior y una cota inferior, esto nos da una idea más precisa del comportamiento y complejidad del mismo[2].

$$\Theta(g(n) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c_1, c_2 \text{ constantes positivas}, n_0\ :$$
$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall\, n \geq n_0\}$$

- **Notacion Omega "Ω"** Esta notación se ocupa para describir la complejidad de un algoritmo definiendo una cota inferior asintótica en el peor caso de ejecución de un algoritmo[2].

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, c \text{ constante positiva y } n_0\ :$$
$$0 < cg(n) \leq f(n), \forall\, n \geq n_0\}$$

- **Analisis a posteriori**. Es una evaluación que se realiza de forma empírica, donde los resultados se obtienen con la ejecución del algoritmo y la medición del tiempo y recursos computacionales que requiere[2].

- **Analisis a priori**. Es una evaluacion que se realiza de forma teorica, donde los resultados se pueden obtener con el conteo de operaciones y/o analisis matematico que en base a sus formulas se obtiene su complegidad algoritmica[2].

- **Sucesion de Fibonacci**. Es una secuencia matematica infinita que incia con los numeros 0 y 1, y el numero siguiente sera la suma de los dos numeros anteriores a este[1].

$$\text{El tercer número es } 0 + 1 = 1.$$
$$\text{El cuarto número es } 1 + 1 = 2.$$
$$\text{El quinto número es } 1 + 2 = 3.$$

- **Numero perfecto**.Se le considera numero perfecto a aquel que la suma de sus divisores propios positivos da por por resultado el mismo numero[3].

$$\text{Los divisores positivos de 28 son 1, 2, 4, 7 y 14.}$$
$$\text{Si sumamos estos divisores: } 1 + 2 + 4 + 7 + 14 = 28.$$

- **Funciones recursivas**. En la programacion las funciones recursivas son aquellas que durante su proceso se invocan asi mismas.

```
Suma_Recursiva(n):
If n == 1
    return 1
Else
    return n + sumaRecursiva(n - 1)
```

- **Funciones iterativas**. En la programacion las funciones iterativas son aquellas se ejecuta en un ciclo n numero de veces y existe una condicion de validacion en cada iteracion que controla si se itera una ves mas o finaliza en ciclo

```
Suma_iterativa(n):
resultado = 0
for i = 0 to n do:
    resultado += i
return resultado
```

- **Pseudocodigo Sucesion de Fibonacci version iterativa**

```
fibonacci_iterativa(num_anterior, num_actual, n)
for i = 0 to n do
    print num_actual
    aux = num_actual;
    num_actual += num_anterior;
    num_anterior = aux;
```

- **Pseudocodigo Sucesion de Fibonacci version recursiva**

```
fibonacci_recursiva(num_anterior, num_actual, n)
    if n == 0
        return
    else
        print num_actual
        fibonacci_recursiva(num_actual, num_actual + num_anterior, n-1);
```

- **Pseudocódigo Perfecto**. El algoritmo de Perfecto dice si el número n es perfecto o no, haciendo la suma de sus divisores con un for, retorna un 0 si el número no es perfecto y retorna un 1 si es perfecto.

```
Perfecto(n):
contador = 0
for i = 1 to i = n-1 do
  if n%i == 0
    contador = contador + i
if contador == n
  retorna 1
retorna 0
```

- **Pseudocódigo MostrarPerfectos**. El algoritmo de MostrarPerfectos, muestra n números perfectos, para cuando $n = 2$ muestra dos números perfectos tal que el output es: $[6, 28]$. Utiliza la función Perfecto para calcular los números perfectos.

```
MostrarPerfectos(n):
contador = 0
for i = 1 to contador != n do
  if Perfecto(i) do
    imprime i
    contador = contador + i
```

3 Experimentación y Resultados

4 Conclusiones

Texto ejemplo, conclusión general.

Conclusiones Catonga Tecla Daniel Isaí 1
Texto ejemplo.

Conclusiones Olguin Castillo Victor Manuel 2
Texto ejemplo.

5 Bibliografía

Ejemplo de referencias.

References

[1] Viggiani Rocha María Isabel. “La sucesión de Fibonacci”. In: *Famaf* 3.1 (2022).

[2] R. C. t. Lee. *Introducción al diseño y análisis de algoritmos: un enfoque estratégico*. Ed. by Mc Graw Hill. 2014.

[3] José L. Bueso Montero. “Números Perfectos y Matemáticos Imperfectos”. In: (2019).