Escuela Superior de Cómputo.
Instituto Politécnico Nacional, México.

Práctica 2: Funciones recursivas vs iterativas.

Blancas Pérez Bryan Israel
orionmunecaycanica@gmail.com

Resumen: Demostrar de manera experimental y formal, el orden de complejidad de algoritmos (cuyo propósito es el mismo) implementados de dos formas distintas (algoritmo recursivo e iterativo), y comparar los resultados con el fin de identificar cuál de los algoritmos es más eficiente.

Palabras Clave: Algoritmo recursivo, algoritmo iterativo, complejidad computacional, cálculo formal del tiempo de ejecución.

1. Introducción

Un algoritmo iterativo es aquel que se ejecuta mediante ciclos iterativos, definidos o indefinidos, para llegar a la resolución del problema. Como opción a éstos, existen los algoritmos recursivos, los cuales usan la recursividad como herramienta para resolver el problema.
La recursividad utiliza más recursos del sistema para su ejecución, es precisamente por ello que en ocasiones se tiene la creencia de que un algoritmo recursivo es siempre más lento que un algoritmo iterativo.
Bajo la premisa de que todo algoritmo recursivo se puede expresar como un algoritmo iterativo y viceversa, en esta práctica mediante el análisis experimental y formal, se compararán ciertos algoritmos en sus dos implementaciones, con el fin de saber cuál es más eficiente dependiendo del problema.

2. Conceptos Básicos

Recursividad.
La recursividad es la forma en la cual se especifica un proceso basado en su propia definición, es decir, construcción a partir del mismo tipo [1]. La técnica de solución de problemas "Divide y Vencerás", comúnmente nos lleva a la recursividad, debido a que se divide el problema en sub-problemas de la misma naturaleza, y por ende el mismo algoritmo para solucionar.
Un ejemplo clásico del uso de la recursividad es el calculo del factorial de un número n. Ya que $n!$ se define como el producto de todos los números anteriores a n, se puede resolver el problema como $n(n-1)! = n(n-1)(n-2)!$ y así sucesivamente hasta llegar $0! = 1$.

Iteración.
Iteración, según el DLE, significa repetir. En programación se interpreta como el acto de repetir un conjunto de instrucciones de manera definida o indefinida, con el fin de alcanzar el resultado deseado [2].

3. Experimentación y Resultados

Ejercicio 1.

Implementar la sucesión de Fibonacci mediante un algoritmo recursivo y mediante un algoritmo iterativo.

Pseudocódigo del algoritmo recursivo:

```
1 F(int n)
2   if n < 2
3     return n
4   else
5     return F(n-1)+F(n-2)
```

La función F, recibe como parámetro un entero n, que representa el n-ésimo termino de la sucesión. La condición de paro en este algoritmo, es cuando n ¡2, ya que se llego al final de todos los términos. En caso contrario, se retorna a $F(n-1) + F(n-2)$, que se interpreta como la suma de los dos números anteriores al número actual.

Pseudocódigo del algoritmo iterativo:

```
1 FI(int n)
2   if n < 2
3     aux=1
4   for i = 2 to i<=n do
5     aux=a+b
6     a=b
7     b=aux
8   return aux;
```

La función FI, recibe como parámetro un entero n, que representa el n-ésimo termino de la sucesión. El for inicia en 2, ya que previamente existe una condición para los primeros dos término de la sucesión, los cuales son iguales a 1. El ciclo calcula la suma de los dos números anteriores al número actual, y después actualiza los valores de las variables para el la siguiente iteración.

Programa en ejecución

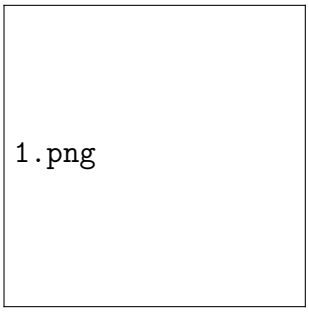

Figura 1: Ejecución del programa 1.

En la figura 1, se aprecia que el programa recibe como entrada el número el cual representa la pregunta ¿Cuál es el n-ésimo término de la sucesión de Fibonacci?. Como salida en la terminal, se muestra el número, primeramente el calculado con el algoritmo iterativo y después el calculado con el algoritmo recursivo.

Figura 2: Ejemplo de output en archivo del programa 1.

En la figura 2, se muestra la salida del programa 1 en el archivo. La primer columna representa el n-ésimo término de la secesión de Fibonacci y la segunda columna representa el tiempo de ejecución (cont) que se requirió para terminar exitosamente el programa.

Gráfica de resultado algoritmo recursivo.

Figura 3: Gráfica del ejecicio 1 recursivo.

En la figura 3, se muestra la gráfica $T(n)$ vs n, en la que los puntos marcados corresponden a los datos obtenidos en el archivo de salida del ejercicio 1. Como es apreciable, el orden de complejidad del algoritmo recursivo, para calcular el n-ésimo término de la sucesión de Fibonacci tiene orden exponencial.

Gráfica de resultado algoritmo recursivo.

Figura 4: Gráfica del ejecicio 1 recursivo.

En la figura 4, se muestra la gráfica $T(n)$ vs n, en la que los puntos marcados corresponden a los datos obtenidos en el archivo de salida del ejercicio 1. Como es apreciable, el orden de complejidad del algoritmo iterativo, para calcular el n-ésimo término de la sucesión de Fibonacci tiene orden lineal.

Comprobación del orden de complejidad del algoritmo iterativo.

```
1 FI(int n)
2   if n < 2
3     aux=1
4   for i = 2 to i<=n do
5     aux=a+b
6     a=b
7     b=aux
8   return aux;
```

Análisis del Algoritmo		
Línea de Código	**Tiempo Ejecución**	**Número de Ejecuciones**
2	C1	1
3	C2	1
4	C3	n
5	C4	$(n-1)$
6	C5	$(n-1)$
7	C6	$(n-1)$
8	C7	1

Cuadro 1: Análisis del algoritmo.

Encontrando el orden de complejidad.
$T(n) = C1 + C2 + C3n + C4(n-1) + C5(n-1) + C6(n-1) + C7.$
$T(n) = (C3 + C4 + C5 + C6)n + C1 + C2 - C4 - C5 - C6 + C7.$
$T(n) = An + B.$

Por lo tanto $T(n)\ \epsilon\ \theta(n).$

Ejercicio 2.

Implementar el algoritmo dado para calcular la suma de los primeros n cubos , e implementar otro algoritmo pero de manera iterativa.

Pseudocódigo del algoritmo recursivo del ejercicio 2:

```
1 S (int n)
2   if n == 1
3     return 1
4   else
5     return S(n-1) + (n * n * n)
```

La función S, recibe como parámetro un número entero positivo. La condición de paro es cuando el número es igual a uno, en caso contrario, se retornará la función $S(n-1)$, es decir el número que continúa en orden descendente, más el número actual elevado al cubo.

Pseudocódigo del algoritmo iterativo del ejercicio 2:

```
1 SI(int n)
2   for i = 1 to i <= n do
3         aux += i * i * i
4 return aux;
```

La función SI, recibe como parámetro un número entero positivo. El algoritmo inicia un for, hasta que i sea igual a el número dado. En cada iteración, se guarda en una variable auxiliar la suma de el número elevado al cubo más lo que había anteriormente.

Programa en ejecución

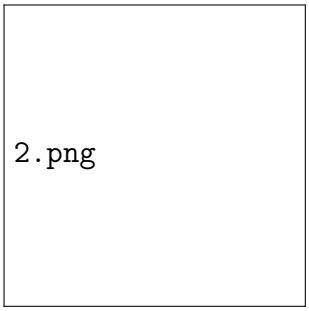

Figura 5: Ejecución del programa 2.

En la figura 5, se muestra que el programa recibe como entrada el número del cual se quiere saber cuanto es la suba de los n primeros números al cubo. Como salida en la terminal, se muestra al cálculo con ambos algoritmos.

Figura 6: Ejemplo de output en archivo del programa 2.

En la figura 4, se muestra la salida del programa 2 en el archivo. La primera columnas es el número n y la segunda columna representa el tiempo de ejecución (cont) que se requirió para calcular la suma.

Gráfica de resultado.

Figura 7: Gráfica 1 del ejercicio 2.

En la figura 7, se muestra la gráfica $T(n)$ vs n, en la que los puntos marcados corresponden a los datos obtenidos en el archivo de la ejecución del programa 2. La práctica pide proponer una función $g(n)$ tal que $T(n)$ ϵ $O(g(n))$, en este caso $g(n) = 3x$

Figura 8: Gráfica 2 del ejercicio 2.

En la figura 8, se muestra la gráfica $T(n)$ vs n, en la que los puntos marcados corresponden a los datos obtenidos en el archivo de salida del

ejercicio 2 iterativo. La práctica pide proponer una función $g(n)$ tal que $T(n)\ \epsilon\ O(g(n))$, en este caso $g(n) = 3x$.

Comprobación del orden de complejidad del algoritmo iterativo.

```
1 SI(int n)
2   for i = 1 to i <= n do
3     aux += i * i * i
4   return aux;
```

Análisis del Algoritmo		
Línea de Código	**Tiempo Ejecución**	**Número de Ejecuciones**
2	C1	$n+1$
3	C2	n
4	C3	1

Cuadro 2: Análisis del algoritmo.

Encontrando el orden de complejidad.
$T(n) = C1(n-1) + C2n + C3$.
$T(n) = (C1 + C2)n + C3 - C1$.
$T(n) = An + B$.
Por lo tanto $T(n)\ \epsilon\ \theta(n)$.

Comprobación de complejidad del algoritmo recursivo.

```
1 S (int n)
2   if n == 1
3     return 1
4   else
5     return S(n-1) + (n * n * n)
```

Encontrando el orden de complejidad.
Se tiene que

$$f(n) = \begin{cases} 1 \ si \ n = 1 \\ s(n-1) + n^3 \ si \ n \ > \ 1 \end{cases}$$

Sea M(n) el número de sumas para f(n).

$$M(n) = \begin{cases} 0 \ si \ n = 1 \\ M(n-1) + 1 \ si \ n \ > \ 1 \end{cases}$$

Entonces:
$M(n) = M(n-1) + 1$
$M(n) = M(n-2) + 2$
i-ésimo término.
$M(n) = M(n-i) + i$
cuando i=n.
$M(n) = M(0) + n$
$M(n) = 1 + n$

Por lo tanto $M(n) \ \epsilon \ \theta(n)$.

4. Conclusiones

Esta práctica para mi es muy útil para darme cuenta de que no hay un algoritmo que sea mejor que otro simplemente por ser de un tipo. En el desarrollo de la práctica se puede apreciar que hay algoritmos los cuales tiene el mismo orden de complejidad computacional, tanto en su implementación recursiva como en la iterativa.
Uno de los problemas que tuve al desarrollar la práctica, es que al graficar el output del algoritmo recursivo de Fibonacci, me graficaba una función lineal, lo cual era incorrecto, ya que como nos había dicho el profesor previamente, el algoritmo recursivo de fibonacci tiene orden de complejidad exponencial. Este problema lo solucioné cambiando el algoritmo, debido a que la primera versión de éste que había hecho, estaba mal implementada.

5. Anexo

Resolver el siguiente problema.

Calcular el orden de complejidad de algoritmo de la burbuja (BubbleSort).

```
1 BubbleSort(A)
2   for i = 1 to i <= A.length-1 do
3     for j = A.length downto j >= i+1
4       if A[j] < A[j-1]
5         exchange A[j] with A[j-1]
```

Encontrando el orden de complejidad.

Análisis del Algoritmo		
Línea de Código	**Tiempo Ejecución**	**Número de Ejecuciones**
2	C1	n
3	C2	$\sum_1^n(ti)$
4	C3	$\sum_1^n(ti-1)$
5	C4	$\sum_1^n(ti-1)$

Cuadro 3: Análisis del algoritmo.

Sea t_i el número de veces que se ejecuta esa línea de código. Entonces. $T(n) = C1(n) + C2\sum_1^n(ti) + (C3 + C4)\sum_1^n(ti - 1)$.

Pero.

Encontrando ti		
i	**i+1 $\leq$ j $\leq$ n**	**ti**
1	$2 \leq j \leq n$	$n-1$
2	$3 \leq j \leq n$	$n-2$
3	$4 \leq j \leq n$	$n-3$
...	...	...
i	$i \leq j \leq n$	$n-i$

Cuadro 4: Encontrando ti

Entonces.
$T(n) = C1(n) + C2\sum_1^n(n - i) + (C3 + C4)\sum_1^n(n - i - 1)$.
$T(n) = C1(n) + C2(n^2 - \frac{n(n+1)}{2}) + (C3 + C4)(n^2 - \frac{n(n+1)}{2} - n)$.
$T(n) = (C2 + C3 + C4)\frac{n^2}{2} + (C1 - \frac{C2}{2} - \frac{3(C3+C4)}{2})n$.
$T(n) = An^2 + Bn$.

por lo tanto $T(n)\ \epsilon\ \theta(n^2)$

6. Bibliografía

[1] es.wikipedia,org/wiki/Recursión.

[2] es.wikipedio.org/wiki/Iteración.