Big O en funciones iterativas, recursivas y directas

César Silva- A01252916

Repositorio de actividad en Github

1. *Función Iterativa* ---> $O(n)$

```
int sumaIterativa (int n){
int sum = 0;
for (int i = 1; i <= n ; i++){
sum += i;
}
return sum;
}
```

Complejidad

Este algoritmo tiene una complejidad $O(n)$ya que se esta iterando de una manera lineal en la cual el desempeño de esta dependerá de la entrada n , se utiliza sum para mantener el control de la acumulación de cada dato i en cada ciclo for, de tal manera que se puede decir que el número de iteraciones que realiza el ciclo for es directamente proporcional a n.

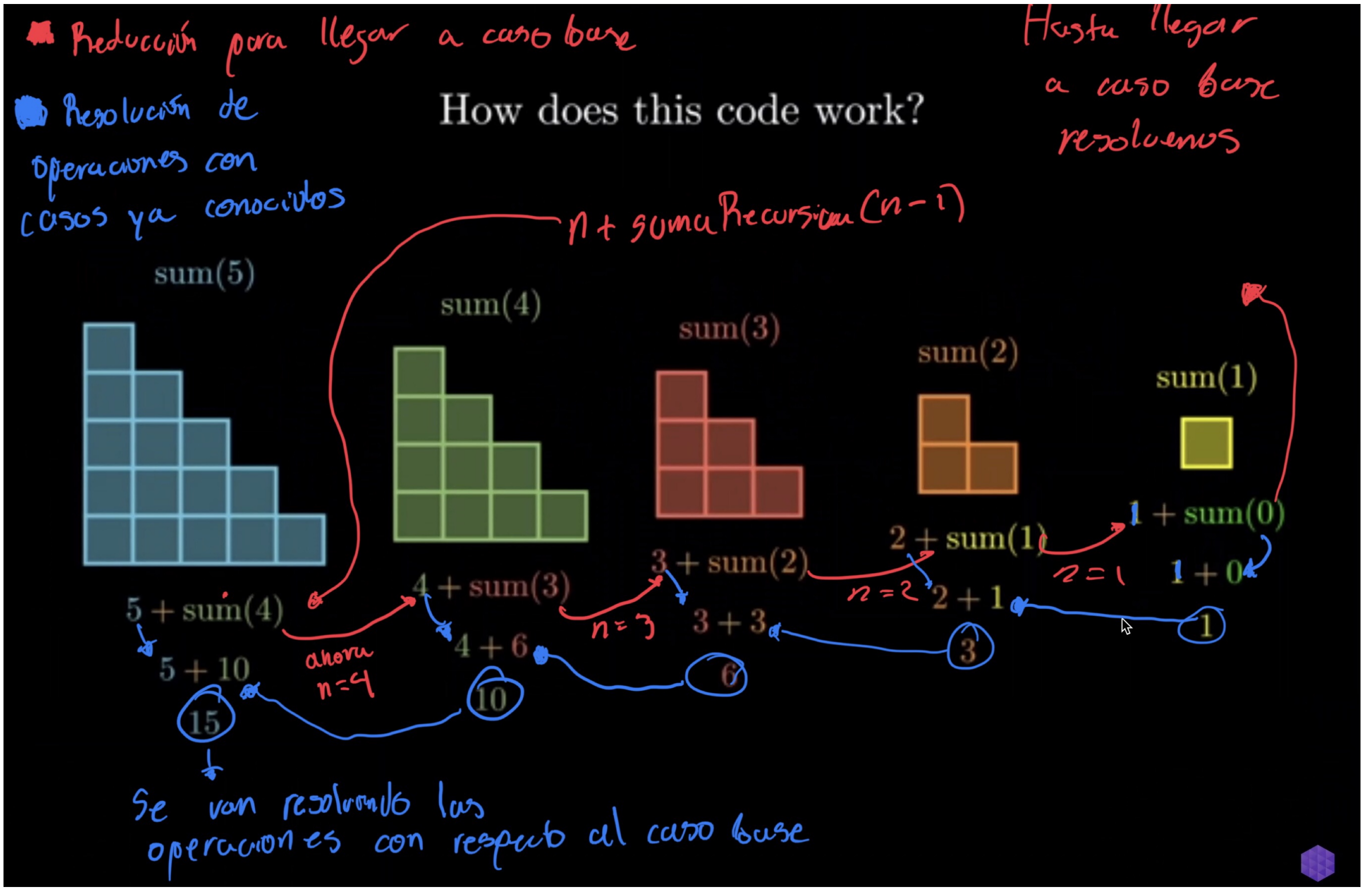
Otra forma de entender que la complejidad temporal es de $O(n)$ es la siguiente:
adentro de nuestro for loop tenemos `sum += i` lo que tiene una complejidad de temporal por si sola de $O(1)$, ya que una suma siempre tardara lo mismo independientemente de del tamaño de n, por otra parte tenemos que el for loop depende directamente de n, es decir entre más grande sea n más tiempo se tardara en resolver es lo que este se tarde depende de N veces. Para conocer la complejidad final del algoritmo multiplicamos $N * O(1)$ y obtenemos que la complejidad final es de $O(n)$.

2. *Función Recursiva* ---> $O(n)$

```
int sumaRecursiva(int n){
if (n <= 0){
return 0;
}
else return n + sumaRecursiva(n-1);
}
```

Complejidad

Este algoritmo también cuenta con una complejidad de $O(n)$ debido a que en cada llamada recursiva la función hace una suma constante y se llama a sí misma con una entrada de uno menor que el valor actual, este numero de llamadas a si misma depende de n. Esto produce que el número total de "iteraciones" sea igual al valor de entrada original, creando así un crecimiento lineal. El tiempo aumenta a medida que aumenta el valor de entrada de una manera proporcional.



También podemos utilizar las formulas de recurrencia para poder entender el patrón:

Partimos de la fórmula base:

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

Para calcular el termino anterior y así poder acercarnos cada vez mas al caso base sustituimos con $T(n-1)$

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

$$T(n-1) = T(n-2) + 1$$

Si sustituimos nuestro nuevo valor de n en la formula base obtenemos:

$$T(n) = (T(n-2) + 1) + 1$$

$$T(n) = T(n-2) + 2$$

Hasta este punto ya podemos comenzar a observar el patrón que cae en lo siguiente: cada vez que se reste un termino interno de T se le suma de manera proporcional a $T(n)$ y hasta aquí podemos decir que el algoritmo esta sucediendo de forma lineal por lo tanto tiene una complejidad de $O(n)$.

3. *Función Directa* ---> $O(1)$

```
int sumaDirecta(int n){
if ( n > 0){
return n * (n + 1)/2;
}else{
return 0;
}
}
```

Complejidad

En este caso tenemos una complejidad constante ya que independientemente del valor que tenga n las operaciones realizadas (suma y multiplicación) no se ven afectadas por n y por tanto siempre se ejecutara en un tiempo constante $O(1)$ al ser un algoritmo independiente de n y tener solamente una misma cantidad de operaciones siempre.

Casos Prueba

En esta sección se evaluarán las entradas y salidas con 4 entradas distintas leídas desde un archivo txt En cada sección de entrada se evalúan las 3 funciones con su respectivo caso.

Archivo de entrada ns.txt:

```
1
0
420
-2
```

Salidas:

```
--------------------------------------------------
La suma de 1 hasta 1 de manera iterativa es: 1
La suma de 1 hasta 1 de manera recursiva es: 1
La suma de 1 hasta 1 de manera directa es: 1
--------------------------------------------------
La suma de 1 hasta 0 de manera iterativa es: 0
La suma de 1 hasta 0 de manera recursiva es: 0
La suma de 1 hasta 0 de manera directa es: 0
--------------------------------------------------
La suma de 1 hasta 420 de manera iterativa es: 88410
La suma de 1 hasta 420 de manera recursiva es: 88410
La suma de 1 hasta 420 de manera directa es: 88410
--------------------------------------------------
Se ingreso el número negativo -2 .Por favor ingrese un entero positivo en el archivo.
```