TP 2 – Análisis de Complejidad

Alumno: Nuñez Souto, Aaron Agustín

1. El concepto de complejidad en el tiempo hace referencia a la cantidad de tiempo que un algoritmo requiere cuando los datos de entrada se incrementan.
2. Con relación a Ave, Internet y Cohete, al hablar de Ave nos referimos a un algoritmo en el cual si el tamaño de los archivos, el ‘ave’ siempre tardará lo mismo en llegar. Por otra parte, al aplicar el algoritmo de ‘Internet’ encontramos que el tiempo de llegada depende de la velocidad, por ende el tiempo sí varia, a mayor tamaño de archivos más tiempo se demora. Por último, el algoritmo del ‘cohete’ es súper rápido, por lo que el tiempo también varía. Estos 3 conceptos hacen alusión a algoritmos con diferentes grados de complejidad, lo cual desencadena en que cada uno presente una eficiencia diferente en comparación a los otros.
3. Las notaciones asintóticas son lenguajes que nos permitan analizar el tiempo de ejecución de un algoritmo identificando su comportamiento si el tamaño de entrada para el algoritmo aumenta. Esto también se conoce como la tasa de crecimiento de un algoritmo. En el análisis asintótico, evaluamos el rendimiento de un algoritmo en términos de tamaño de entrada (no medimos el tiempo de ejecución real). Calculamos cómo aumenta el tiempo (o espacio) que tarda un algoritmo con el tamaño de entrada.
4. ***Big Theta***:  La notación theta limita una función desde arriba y desde abajo, por lo que define el comportamiento asintótico exacto.  
   Una forma sencilla de obtener la notación Theta de una expresión es eliminar los términos de orden bajo e ignorar las constantes principales.

Eliminar términos de orden inferior siempre está bien porque siempre habrá un número (n) después del cual Θ (n3) tiene valores más altos que Θ(n2) independientemente de las constantes involucradas.

1. ***Big O***: La notación Big O define un límite superior de un algoritmo, limita una función solo desde arriba. Por ejemplo, considere el caso de Insertion Sort. Toma tiempo lineal en el mejor de los casos y tiempo cuadrático en el peor de los casos. Podemos decir con seguridad que la complejidad temporal de la ordenación de inserción es O(n^2). Tenga en cuenta que O(n^2) también cubre el tiempo lineal. Si usamos la notación Θ para representar la complejidad temporal del orden de inserción, tenemos que usar dos instrucciones para los mejores y peores casos:
2. La complejidad de tiempo en el peor de los casos de ordenación de inserción es Θ(n^2).
3. La complejidad de tiempo en el mejor de los casos de Insertion Sort es Θ(n).

La notación Big O es útil cuando solo tenemos un límite superior en la complejidad del tiempo de un algoritmo. Muchas veces encontramos fácilmente un límite superior simplemente mirando el algoritmo.

1. ***Big Omega***: Así como la notación Big O proporciona un límite superior asintótico en una función, Ω notación proporciona un límite inferior asintótico.  
   Ω notación puede ser útil cuando tenemos un límite inferior en la complejidad del tiempo de un algoritmo. Como se discutió en la publicación anterior, el mejor rendimiento de un algoritmo generalmente no es útil,la notación Omega es la notación menos utilizada entre las tres.
2. La O viene de clasificar el ORDEN de magnitud de la función a analizar, resumiéndose en que nos sirve para determinar cómo un algoritmo escala en complejidad tanto en tiempo como en espacio necesario, en función de su tamaño de entrada.

Podemos resumir la nota dada a unos pocos valores, ya que descartamos las constantes y sólo atendemos a unos pocos casos según el orden de magnitud de crecimiento, y sin importar por ejemplo el detalle de la inclinación de cada curva y sólo la forma de comportamiento de crecimiento. En resumen, nos podemos quedar con que tenemos:

• O(1) – Tiempo constante: es el mejor resultado, y quiere decir que el tiempo de ejecución no varía conforme aumenta el tamaño de los datos de entrada, y la respuesta siempre tarda lo mismo sin importar la magnitud de entrada.

• O(n) – Tiempo lineal: el crecimiento es lineal en tanto el tiempo de ejecución es cada vez mayor de modo proporcional a cómo se incrementa el tamaño de la entrada. Por lo que si tenemos el doble de elementos de entrada, tardará el doble, aunque despreciamos realmente la pendiente de la misma y sólo nos quedamos con que aumenta de forma lineal.

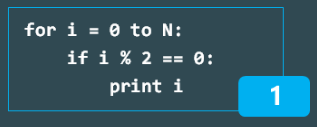
• O(log n) – Tiempo logarítmico: una forma de crecimiento que crece al inicio pero tiende a estabilizarse conforme aumentan el tamaño de entrada, por lo que es una buena nota para un algoritmo ya que no tiende a resentirse.

• O(n2) – Tiempo cuadrático: el crecimiento es de forma exponencial por lo que será un algoritmo a evitar ya que para valores pequeños de entrada el tiempo será asumible, pero conforme aumente el tamaño de los datos de entrada el tiempo tenderá a ser muy elevado y es probable que el procesador se quede inoperativo.

• O(n!) – Tiempo factorial: el crecimiento es factorial, por lo que rápidamente tiende a valores imposibles de tratar, en lo que sería una recta vertical.

1. La **complejidad del tiempo** es una función que describe cualitativamente el tiempo de ejecución de un algoritmo. La complejidad del tiempo generalmente se expresa mediante el gran símbolo O, que puede entenderse simplemente como el número de operaciones básicas en este algoritmo. En el cálculo, se usa un límite matemático. Veamos el cálculo de la complejidad del espacio. Por otra parte, la **complejidad espacial** de un programa se refiere a la cantidad de memoria requerida para ejecutar un programa. Usando la complejidad espacial del programa, es posible estimar de antemano cuánta memoria requiere el programa. Además del espacio de almacenamiento y las instrucciones, constantes, variables y datos de entrada utilizados por un programa, un programa necesita algunas unidades de trabajo para operar en los datos y el espacio auxiliar para almacenar cierta información necesaria para el cálculo real. El espacio de almacenamiento requerido durante la ejecución del programa incluye las siguientes dos partes: *parte fija* y *espacio variable*.

**Marco Práctico**

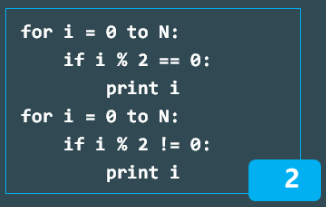
1. 

*for i = 0 to N* => O(n)

*if i % 2 == 0*:

*print I*  => O(1)

n \* 1 = n = **O(n)**

1. 

*for i = 0 to N:* => O(n)

if i % 2 == 0:

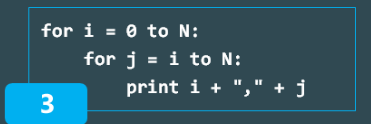
*print i* => O(1)

*for I = 0 to N:* => O(n)

if i % 1 != 0:

*print i* => O(1)

n \* 1 + n \* 1 = n + n = 2n => n = **O(n)**

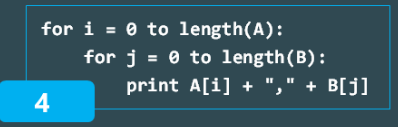
1. 

for i = 0 to N: => O(n)

for j = i to N: => O(n)

print i + “,” + j => O(1)

n \* n \* 1 = n2 = **O(n2)**

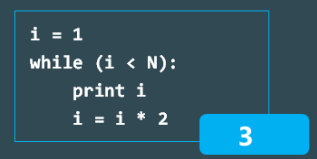
1. 

for i = 0 to length(A): => O(n)

for j = 0 to length(B): => O(n)

print A[i] + “”, + B[j] => O(1)

n \* n \* 1 = n2 = **O(n2)**

1. 

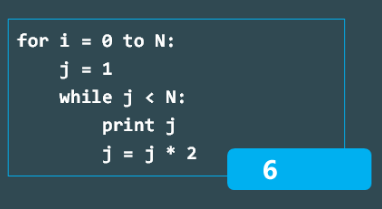
i = 1 => O(1)

while (i < N): => O(n)

print i => O(1)

i = i \* 2 => O(1)

1 + n\*1\*1 = 1 + n = n = **O(n)**

1. 

for i = 0 to N: => O(n)

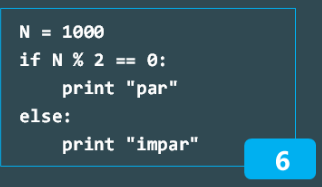
j = 1 => O(1)

while j < N: => O(n)

print j => O(1)

j = j \* 2 => O(1)

n \* [1 + n \* (1 + 1)] = n \* [1 + 2n] = n + 2n2 => n2 = **O(n2)**

1. 

N = 1000 => O(1)

if N % 2 == 0:

print “par” => O(1)

else:

print “impar” => O(1)

1 + 1 + 1 = 3 => 1 = **O(1)**