

COMPLEJIDAD DE ALGORITMOS

Agenda

- ❑ Introducción
- ❑ Análisis de tiempo de ejecución.
- ❑ Tasas de crecimiento comúnmente utilizadas
- ❑ Notación Asintótica (Big-O, Big- Ω , θ Theta).
- ❑ Complejidad en Algoritmos con Bucles.
- ❑ Ejercicios

Introducción

Para ir de la ciudad “A” a la ciudad “B”, puede haber muchas formas de lograrlo: en avión, en autobús, en tren y también en bicicleta.

Dependiendo de la disponibilidad y conveniencia, elegimos la que más nos convenga. De manera similar, en computación, existen múltiples algoritmos disponibles para resolver el mismo problema. Por ejemplo, para el problema de ordenamiento podemos elegir entre el algoritmo de burbuja o el algoritmo de ordenamiento rápido (Quicksort).

El análisis de algoritmos nos ayuda a determinar qué algoritmo es más eficiente en términos de tiempo y espacio consumido.

Introducción

El objetivo del análisis de algoritmos es comparar algoritmos (o soluciones) principalmente en términos de tiempo de ejecución, pero también en términos de otros factores (por ejemplo, memoria, esfuerzo del desarrollador, etc.).

¿Qué es el análisis de tiempo de ejecución?

Es el proceso de determinar cómo aumenta el tiempo de procesamiento a medida que aumenta el tamaño del problema (tamaño de entrada).

El tamaño de entrada es el número de elementos en la entrada y, según el tipo de problema, la entrada puede ser de diferentes tipos.

Los siguientes son los tipos comunes de entradas.

- Tamaño de una matriz
- Grado polinómico
- Número de elementos en una matriz.
- Número de bits en la representación binaria de la entrada.
- Vértices y aristas en un grafo.

¿Cómo calcular el tiempo de ejecución?

El tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

A continuación se listan algunas operaciones elementales:

- operaciones aritméticas,
- operaciones lógicas,
- comparaciones entre escalares,
- accesos a variables escalares,
- accesos a elementos de vectores o matrices,
- asignaciones de valores a variables,
- asignaciones de valores a elementos de vectores o matrices,

Ejemplo:

Código 1

```
Inicio  
  Leer n  
   $m \leftarrow n * n$   
  Escribir m  
Fin
```

Código 2

```
Inicio  
  Leer n  
   $m \leftarrow 0$   
  para( $i \leftarrow 0$ ;  $i < n$ ;  $i++$ )  
     $m \leftarrow m + n$   
  Escribir m  
Fin
```

Código 3

```
Inicio  
  Leer n  
   $m \leftarrow 0$   
  para( $i \leftarrow 0$ ;  $i < n$ ;  $i++$ )  
    para( $j \leftarrow 0$ ;  $j < n$ ;  $j++$ )  
       $m++$ ;  
  Escribir m  
Fin
```

Complejidad de Algoritmos con Bucles

- Para contar las instrucciones debemos saber cómo contar las operaciones en un bucle de tipo **for**.
- Asumiendo que el número de iteraciones en el bucle es **n** entonces:
 - Condición del bucle es ejecutado **n+1** veces
 - Cada una de las sentencias que están dentro del cuerpo del bucle es ejecutado **n** veces
 - El contador del bucle se actualiza **n** veces

Complejidad de Algoritmos con Bucles

```
double x, y;  
x = 2.5 ; y = 3.0;  
for(int i = 0; i < n; i++){  
    a[i] = x * y;  
    x = 2.5 * x;  
    y = y + a[i];  
}
```

Bucle *for* con (<)

- Considerando el bucle *for*:

```
for (int i = k; i < n; i = i + m) {  
    statement1;  
    statement2;  
}
```

```
for (int i = n; i > k; i = i - m) {  
    statement1;  
    statement2;  
}
```

- El número de iteraciones es: $\lceil (n - k) / m \rceil$
- La instrucción de inicialización $i=k$, es ejecutada una sola vez
- La condición $i < n$, es ejecutada $\lceil (n - k) / m \rceil + 1$ veces.
- La instrucción de actualización $i=i+m$, es ejecutada $\lceil (n - k) / m \rceil$ veces.
- Cada una de las instrucciones **statement1** y **statement2** es ejecutado $\lceil (n - k) / m \rceil$ veces.

Bucle *for* con (\leq)

- Considerando el bucle *for*:

```
for (int i = k; i <= n; i = i + m) {  
    statement1;  
    statement2;  
}
```

```
for (int i = n; i >= k; i = i - m) {  
    statement1;  
    statement2;  
}
```

- El número de iteraciones es: $\left\lfloor \left(\frac{n-k}{m} \right) + 1 \right\rfloor$
- La instrucción de inicialización $\mathbf{i=k}$, es ejecutada una sola vez.
- La condición $\mathbf{i \leq n}$, es ejecutada $\left\lfloor \left(\frac{n-k}{m} \right) + 1 \right\rfloor + 1$ veces.
- La instrucción de actualización $\mathbf{i=i+m}$, es ejecutada $\left\lfloor \left(\frac{n-k}{m} \right) + 1 \right\rfloor$ veces.
- Cada una de las instrucciones **statement1** y **statement2** es ejecutado $\left\lfloor \left(\frac{n-k}{m} \right) + 1 \right\rfloor$ veces.

Bucles *for* simples logarítmicas

Considerando el bucle *for* con (<)

```
for (int i = k; i < n; i = i * m) {  
    statement1;  
    statement2;  
}
```

- El número de iteraciones es: $\lceil (\text{Log}_m (n / k)) \rceil$

Considerando el bucle *for* con (<=)

```
for (int i = k; i <= n; i = i * m) {  
    statement1;  
    statement2;  
}
```

- El número de iteraciones es: $\lfloor (\text{Log}_m (n / k) + 1) \rfloor$

Ejemplo de bucles *for* simples logarítmicas

Considerando el bucle *for* con (<)

```
for(int i = 1; i < n; i = i * 2){  
    statement1;  
    statement2;  
}
```

```
for(int i = n; i > 1; i = i / 2){  
    statement1;  
    statement2;  
}
```

- El número de iteraciones es: $\log_2 n$

Considerando el bucle *for* y *while* con (<=)

```
for (int i = 1; i <= n; i = i * 2) {  
    statement1;  
    statement2;  
}
```

```
for (int i = n; i >= 1; i = i / 2) {  
    statement1;  
    statement2;  
}
```

```
int i = n;  
while(i >= 1){  
    statement1;  
    i = i / 2;  
}
```

- El número de iteraciones es: $\log_2 n + 1$

Bucles *for* o *While* anidados

Considerando el bucles *for* con (<)

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        // do something  
    }  
}
```

- El número de iteraciones es: n^2

Considerando el bucles *for* y un *for* logarítmico

```
for (int i = 0; i < n; i++){  
    for (int j = n; j >= 1; j = j / 2){  
        // do something  
    }  
}
```

- El número de iteraciones es: $n \log_2 n$

Bucles *for* o *While* anidados

Considerando el bucles *for* con (<)

```
for (int i = 0; i < n; i++){  
    for (int j = 1; j < i; j++){  
        // do something  
    }  
}
```

- El número de iteraciones es: $n(n+1)/2$

Considerando el bucles *for* y un *for* logarítmico

```
for (int i = 0; i < n; i++){  
    for (int j = n; j >= 1; j = j / 2){  
        // do something  
    }  
}
```

- El número de iteraciones es: $n \log_2 n$

¿Cómo comparar algoritmos?

Solución ideal

Supongamos que expresamos el tiempo de ejecución de un algoritmo dado como una función del tamaño de entrada n (es decir, $T(n)$) y comparemos estas diferentes funciones correspondientes a los tiempos de ejecución. Este tipo de comparación es independiente del tiempo de la máquina, el estilo de programación, etc.

¿Cómo comparar algoritmos?

¿Qué es la tasa de crecimiento?

La tasa a la que aumenta el tiempo de ejecución en función de la entrada se denomina tasa de crecimiento.

En muchos casos, se toman ciertas aproximaciones para realizar mediciones. Por ejemplo, si medimos el peso de una persona, sabemos que su ropa agrega un peso adicional. Sin embargo, en muchos casos no se toma en cuenta considerando que el aporte al peso total es mínimo.

¿Cómo comparar algoritmos?

¿Qué es la tasa de crecimiento?

$$\text{Peso_Total} = \text{Peso_Persona} + \text{Peso_Ropa}$$

$$\text{Peso_Total} \approx \text{Peso_Persona} \text{ (aproximación)}$$

En general, podemos representar magnitudes utilizando funciones y, para dichas funciones ignorar los términos de orden inferior que son relativamente insignificantes (para un gran valor del tamaño de entrada, n).

¿Cómo comparar algoritmos?

Por ejemplo:

En el siguiente caso, n^4 , $2n^2$, $100n$ y 500 son los costos individuales de alguna función y se aproximan a n^4 ya que n^4 es la tasa de crecimiento más alta.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

Tasas de Crecimiento comúnmente utilizadas

La siguiente es la relación entre las diferentes tasas de crecimiento.

$$1 < \log(n) < \sqrt{n} < n < n \log(n) < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$$

Tasas de Crecimiento comúnmente utilizadas

TIPOS DE ANÁLISIS

Para analizar el algoritmo dado, necesitamos saber con qué entradas tarda menos tiempo el algoritmo y con qué entradas el algoritmo tarda una mayor cantidad de tiempo.

Ya hemos visto que un algoritmo se puede representar en forma de una expresión. Eso significa que representamos el algoritmo con múltiples expresiones: una para el caso en que lleva menos tiempo y otra para el caso en que lleva más tiempo.

Tasas de Crecimiento comúnmente utilizadas

Hay tres tipos de análisis:

Peor de los casos

- Define la entrada para la cual el algoritmo tarda mucho tiempo (la más lenta hora de completar).
- La entrada es aquella para la cual el algoritmo se ejecuta más lentamente.

Tasas de Crecimiento comúnmente utilizadas

Hay tres tipos de análisis:

Mejor caso

- Define la entrada para la que el algoritmo tarda menos tiempo (más rápido hora de completar).
- La entrada es aquella para la cual el algoritmo se ejecuta más rápido.

Tasas de Crecimiento comúnmente utilizadas

Hay tres tipos de análisis:

Caso promedio

- Proporciona una predicción sobre el tiempo de ejecución del algoritmo.
- Ejecuta el algoritmo muchas veces, utilizando muchas entradas diferentes que vienen a partir de alguna distribución que genera estas entradas, calcula el total tiempo de ejecución (agregando los tiempos individuales) y dividir por el número de intentos.
- Supone que la entrada es aleatoria.

Límite inferior \leq Tiempo promedio \leq Límite superior

Notación Asintótica

Teniendo las expresiones para el mejor, el promedio y el peor de los casos, para los tres casos necesitamos identificar los límites superior e inferior.

Big-O límite superior

Big Ω límite inferior

θ Theta límite medio

Para representar estos límites superior e inferior, necesitamos algún tipo de sintaxis. Supongamos que el algoritmo dado se representa en la forma de la función $f(n)$.

Notación Asintótica

Notación Big-O [O Grande - Función de límite superior]

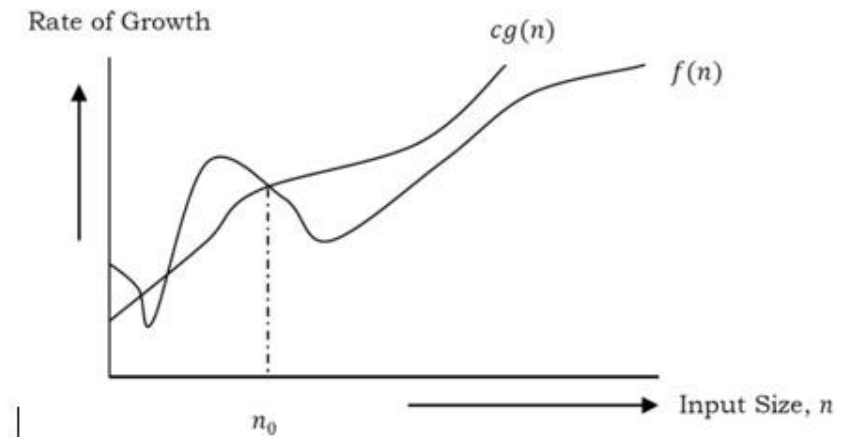
Esta notación da el límite superior ajustado de la función dada.

Se representa como $f(n) = O(g(n))$.

$f(n) = O(g(n))$ si y solo si

\exists las constantes c y n_0 tal que

$$f(n) \leq c * g(n) \quad \forall n \geq n_0$$




Eso significa que, a valores mayores de n , el límite superior de $f(n)$ es $g(n)$.

Notación Asintótica

Ejemplo

Si $f(n) = 2n + 3$, entonces podemos escribir:

$$2n + 3 \leq \text{_____} n$$


$$\underbrace{2n + 3}_{f(n)} \leq \underbrace{10}_C \underbrace{n}_{g(n)}$$

$$n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = O(n)$$

$$\underbrace{2n + 3}_{f(n)} \leq \underbrace{7}_C \underbrace{n}_{g(n)}$$

$$n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = O(n)$$

Notación Asintótica

Ejemplo

Si $f(n) = 2n + 3$, entonces podemos escribir:

$$\underbrace{2n + 3}_{f(n)} \leq \underbrace{5}_C \underbrace{n}_{g(n)} \quad n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = O(n)$$

2n+3n
↙

Pero también puede ser

$$\underbrace{2n + 3}_{f(n)} \leq \underbrace{5}_C \underbrace{n^2}_{g(n)} \quad n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = O(n^2)$$

2n²+3n²
↙

Notación Asintótica

$$\underbrace{2n + 3}_{f(n)} \leq \underbrace{5}_C \underbrace{n}_{g(n)} \quad n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = O(n)$$

$$1 < \log(n) < \sqrt{n} < \boxed{n} < n \log(n) < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$$

Límite inferior

Límite medio

Límite superior

Por lo tanto, para la función $f(n) = 2n + 3$, el O grande es la función más cercana; es decir $f(n) = O(n)$.

Notación Asintótica

Notación Big-Ω [Función de límite inferior]

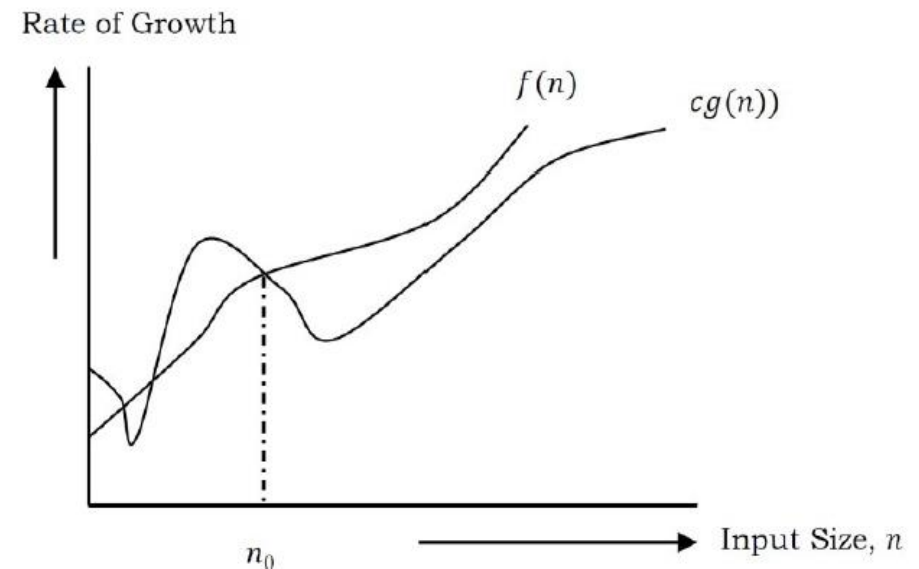
Esta notación da el límite inferior ajustado de la función dada.

Se representa como $f(n) = \Omega(g(n))$.

$f(n) = \Omega(g(n))$ si y solo si

\exists las constantes c y n_0 tal que

$$f(n) \geq c * g(n) \quad \forall n \geq n_0$$




Eso significa que, a valores mayores de n , el límite inferior de $f(n)$ es $g(n)$.

Notación Asintótica

Ejemplo

Si $f(n) = 2n + 3$, entonces podemos escribir:

$$2n + 3 \geq \underline{\quad} n$$


$$\underbrace{2n + 3}_{f(n)} \geq \underbrace{1}_c \underbrace{n}_{g(n)}$$

$$n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = \Omega(n)$$

$$\underbrace{2n + 3}_{f(n)} \geq \underbrace{1}_c \underbrace{\text{Log}(n)}_{g(n)}$$

$$n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = \Omega(\text{Log}(n))$$

Notación Asintótica

Ejemplo

Pero,

$$\underbrace{2n + 3}_{f(n)} \geq \underbrace{1}_{\underbrace{C}} \underbrace{n^2}_{g(n)} \quad n \geq 1 \quad \text{No cumple}$$

Así: $\underbrace{2n + 3}_{f(n)} \geq \underbrace{1}_{\underbrace{C}} \underbrace{n}_{g(n)} \quad n \geq \underbrace{1}_{n_0} \text{ luego } f(n) = \Omega(n)$

$1 < \log(n) < \sqrt{n} < \textcolor{red}{n} < n \log(n) < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$

Notación Asintótica

Notación Theta- θ

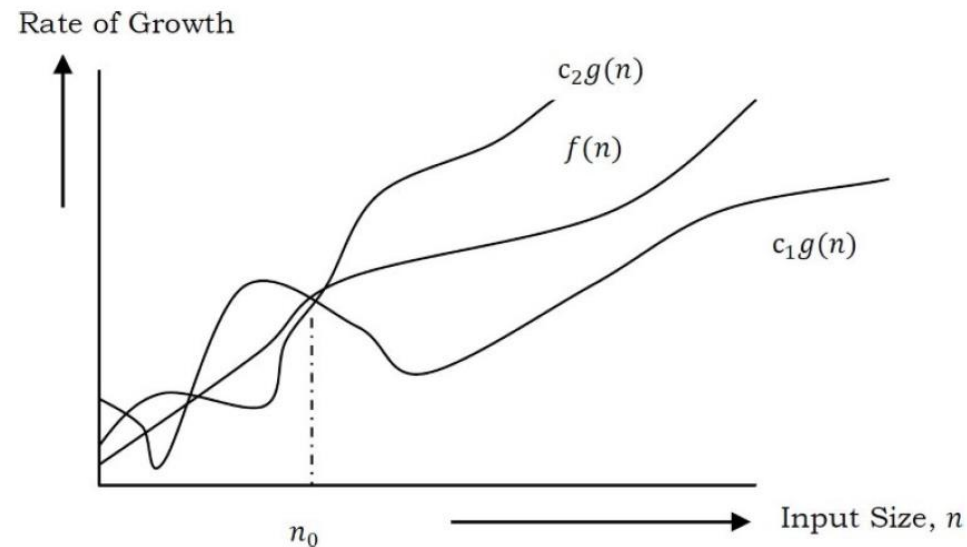
Esta notación da el límite inferior ajustado de la función dada.

Se representa como $f(n) = \theta(g(n))$.

$f(n) = \theta(g(n))$ si y solo si

\exists las constantes c_1, c_2 y n_0 tal que

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall n \geq n_0$$



Notación Asintótica

Nota:

- ☐ Esta notación decide si los **límites superior e inferior** de una función dada son los mismos.
- ☐ El tiempo de ejecución promedio de un algoritmo siempre está entre el límite inferior y el límite superior.
- ☐ Si el límite superior (O) y el límite inferior (Ω) dan el mismo resultado, entonces la notación Θ también tendrá la misma tasa de crecimiento.
- ☐ Para una función dada, si las tasas de crecimiento (límites) para O y Ω no son las mismas, entonces la tasa de crecimiento para el caso Θ puede no ser la misma.

Notación Asintótica

Ejemplo

Si $f(n) = 2n + 3$, entonces podemos escribir:

$$\underbrace{1}_{c_1} * \underbrace{n}_{g(n)} \leq \underbrace{2n + 3}_{f(n)} \leq \underbrace{5}_{c_2} * \underbrace{n}_{g(n)} \quad n \geq 1, \therefore f(n) = \theta(n)$$

