Ejercicios resueltos de análisis de complejidad algorítmica

Lógica y Algorítmica

Curso 2025-2026

Índice

1. Bucles sencillos

Ejercicio 1

```
function ejercicio1(int n)
    for (int i = 0; i < n; i += 3)          // Línea 1
        print(i)                            // Línea 2
    end_for                                 // Línea 3
    return n                                // Línea 4
end_function
```

1.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Donde:

- $T_{bucle}(n)$ es el tiempo de ejecución del bucle (líneas 1-3).
- $T_{return}(n)$ es el tiempo de ejecución de la línea 4.

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$ vamos a reescribir el bucle de la siguiente manera:

```
    for (int i = 0; i <= n - 1; i += 3)     // Línea 1
        print(i)                            // Línea 2
    end_for                                 // Línea 3
```

Hemos reescrito la condición original $i<n$ como $i \leq n-1$ para identificar el límite superior del intervalo, lo que simplifica la construcción del sumatorio.

Para determinar el número de iteraciones del bucle (líneas 1-3), analizamos la progresión de los valores que toma la variable i:

- **Límite inferior:** $i = 0$
- **Límite superior:** $i \leq n-1$
- **Incremento:** 3 en cada paso ($i = i+3$)

Dado que el bucle no se incrementa de 1 en 1, calculamos el número total de iteraciones aplicando la fórmula de los términos de una progresión aritmética:

$$\text{Número de iteraciones} = \frac{\text{Límite superior} - \text{Límite inferior}}{\text{Incremento}} + 1$$

Sustituyendo los valores de nuestro bucle:

$$\text{Número de iteraciones} = \frac{(n-1) - 0}{3} + 1$$

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice recorre desde la primera hasta la última iteración:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor \frac{n-1}{3} + 1 \rfloor} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle. Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) \leq c_2 \cdot \left(\frac{n-1}{3} + 1 \right)$$

Simplificando los términos de la expresión:

$$T_{bucle}(n) \leq c_2 \cdot \left(\frac{n}{3} - \frac{1}{3} + 1 \right) = \frac{c_2}{3} n + \frac{2c_2}{3}$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = \frac{c_2}{3} n + \frac{2c_2}{3} + c_1$$

1.2 Cálculo del orden de complejidad $O(n)$

En el análisis asintótico, despreciamos las constantes multiplicativas y los términos de menor orden:

1. El término de mayor orden es n.
2. Las constantes $\frac{c_2}{3}$, $\frac{2c_2}{3}$ y c_1 no afectan al crecimiento asintótico.

El algoritmo tiene una complejidad **lineal**, ya que el número de operaciones crece de manera directamente proporcional a n.

$$T(n) \in \mathcal{O}(n)$$

Ejercicio 2

```
function ejercicio2(int n)
    for (int i = 0; i < n/2; i += 5)        // Línea 1
        print(i)                            // Línea 2
    end_for                                 // Línea 3
    return n                                // Línea 4
end_function
```

2.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Donde:

- $T_{bucle}(n)$ es el tiempo de ejecución del bucle (líneas 1-3).
- $T_{return}(n)$ es el tiempo de ejecución de la línea 4.

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$ vamos a reescribir el bucle de la siguiente manera:

```
    for (int i = 0; i <= (n/2) - 1; i += 5)  // Línea 1
        print(i)                             // Línea 2
    end_for                                  // Línea 3
```

Hemos reescrito la condición original $i<n/2$ como $i \leq (n/2) - 1$ para identificar el límite superior del intervalo, lo que simplifica la construcción del sumatorio.

Para determinar el número de iteraciones del bucle (líneas 1-3), analizamos la progresión de los valores que toma la variable i:

- **Límite inferior:** $i = 0$
- **Límite superior:** $i \leq (n/2) - 1$
- **Incremento:** 5 en cada paso ($i = i+5$)

Dado que el bucle no se incrementa de 1 en 1, calculamos el número total de iteraciones aplicando la fórmula de los términos de una progresión aritmética:

$$\text{Número de iteraciones} = \frac{\text{Límite superior} - \text{Límite inferior}}{\text{Incremento}} + 1$$

Sustituyendo los valores de nuestro bucle:

$$\text{Número de iteraciones} = \frac{(n/2-1)-0}{5} + 1$$

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice recorre desde la primera hasta la última iteración:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor \frac{n/2-1}{5} + 1 \rfloor} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle. Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) \leq c_2 \cdot \left(\frac{n/2-1}{5} + 1 \right)$$

Simplificando los términos de la expresión:

$$T_{bucle}(n) \leq c_2 \cdot \left(\frac{n}{10} - \frac{1}{5} + 1 \right) = \frac{c_2}{10} n + \frac{4c_2}{5}$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = \frac{c_2}{10} n + \frac{4c_2}{5} + c_1$$

2.2 Cálculo del orden de complejidad $O(n)$

En el análisis asintótico, despreciamos las constantes multiplicativas y los términos de menor orden:

1. El término de mayor orden es n.
2. Las constantes $\frac{c_2}{10}$, $\frac{4c_2}{5}$ y c_1 no afectan al crecimiento asintótico.

El algoritmo tiene una complejidad **lineal**, ya que el número de operaciones crece de manera directamente proporcional a n.

$$T(n) \in \mathcal{O}(n)$$

Ejercicio 3

```
function ejercicio3(int n)
    for (int i = n; i > 0; i--)            // Línea 1
        print(i)                           // Línea 2
    end_for                                // Línea 3
    return n                               // Línea 4
end_function
```

3.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Donde:

- $T_{bucle}(n)$ es el tiempo de ejecución del bucle (líneas 1-3).
- $T_{return}(n)$ es el tiempo de ejecución de la línea 4.

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para determinar el número de iteraciones del bucle (líneas 1-3), analizamos la progresión de los valores que toma la variable i:

- **Valor inicial:** $i = n$
- **Condición:** $i > = 1$
- **Decremento:** 1 en cada paso ($i = i - 1$)

La lógica en bucles con decremento es idéntica a la de los bucles con incremento. La siguiente fórmula nos permite calcular el número de iteraciones basándonos en los límites del intervalo:

$$\text{Número de iteraciones} = \frac{\text{Límite superior (inicio)} - \text{Límite inferior (fin)}}{\text{Decremento}} + 1$$

Sustituyendo los valores de nuestro bucle:

$$\text{Iteraciones} = \frac{n - 1}{1} + 1 = n$$

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice recorre desde la primera hasta la última iteración:

$$T_{bucle}(n) = \sum_{i = 1}^{n} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle. Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) = n \cdot c_2$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = c_2 \cdot n + c_1$$

3.2 Cálculo del orden de complejidad $O(n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico de la expresión $T(n) = c_2 \cdot n + c_1$:

1. **Descartamos los términos de menor orden.** La constante c_1 es un término de orden inferior comparado con $c_2 \cdot n$.
2. **Descartamos las constantes multiplicativas.** La constante c_2 no afecta al crecimiento asintótico de la función.

El término dominante es n, por lo que el algoritmo tiene una complejidad **lineal**.

$$T(n) \in \mathcal{O}(n)$$

Ejercicio 4

```
function ejercicio4(int n)
    for (int i = 1; i <= n; i = i * 2)      // Línea 1
        print(i)                            // Línea 2
    end_for                                 // Línea 3
    return n                                // Línea 4
end_function
```

4.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Donde:

- $T_{bucle}(n)$ es el tiempo de ejecución del bucle (líneas 1-3).
- $T_{return}(n)$ es el tiempo de ejecución de la línea 4.

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para determinar el número de iteraciones del bucle (líneas 1-3), analizamos la progresión geométrica de los valores que toma la variable i:

- **Iteración 0:** $i = 1 = 2^0$
- **Iteración 1:** $i = 2 = 2^1$
- **Iteración 2:** $i = 4 = 2^2$
- **...**
- **Iteración k:** $i = 2^k$

El bucle continúa mientras se cumpla la condición $i \leq n$. Para despejar el número de iteraciones en la última vuelta (k), resolvemos la siguiente inecuación:

$$\begin{aligned} 2^k &\leq n \\ \log_2(2^k) &\leq \log_2(n) \\ k &\leq \log_2 n \end{aligned}$$

Por las propiedades de los logaritmos $\log_a(a^x) = x$.

El número total de iteraciones es $\lfloor \log_2 n \rfloor + 1$, contando desde la primera iteración $k = 0$ hasta la última $\lfloor \log_2 n \rfloor$.

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice k representa

cada iteración:

$$T_{bucle}(n) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle. Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) = c_2 \cdot (\lfloor \log_2 n \rfloor + 1)$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = c_2 \cdot (\lfloor \log_2 n \rfloor + 1) + c_1$$

4.2 Cálculo del orden de complejidad $O(\log n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Las constantes c_1 y c_2 son términos de orden inferior comparados con $\log_2 n$.
2. **Descartamos las constantes multiplicativas.** Las constantes no afectan al crecimiento asintótico.
3. **Cambio de base del logaritmo.** Todas las funciones logarítmicas pertenecen al mismo orden de complejidad, independientemente de la base.

El término dominante es $\log n$, por lo que el algoritmo tiene una complejidad **logarítmica**.

$$T(n) \in \mathcal{O}(\log n)$$

Ejercicio 5

```
function ejercicio5(int n)
    for (int i = n; i >= 1; i = i / 2)      // Línea 1
        print(i)                            // Línea 2
    end_for                                 // Línea 3
    return n                                // Línea 4
end_function
```

5.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Donde:

- $T_{bucle}(n)$ es el tiempo de ejecución del bucle (líneas 1-3).
- $T_{return}(n)$ es el tiempo de ejecución de la línea 4.

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para determinar el número de iteraciones del bucle (líneas 1-3), analizamos la progresión geométrica decreciente de los valores que toma la variable i:

- **Iteración 0:** $i = n = n / 2^0$
- **Iteración 1:** $i = n / 2 = n / 2^1$
- **Iteración 2:** $i = n / 4 = n / 2^2$
- **...**
- **Iteración k:** $i = n / 2^k$

El bucle continúa mientras se cumpla la condición $i \geq 1$. Para despejar el número de iteraciones en la última vuelta (k), resolvemos la siguiente inecuación:

$$\frac{n}{2^k} \geq 1$$
$$n \geq 2^k$$
$$\log_2(n) \geq \log_2(2^k)$$
$$\log_2 n \geq k$$

Por las propiedades de los logaritmos $\log_a(a^x) = x$.

El número total de iteraciones es $\lfloor \log_2 n \rfloor + 1$, contando desde la primera iteración $k = 0$ hasta la última $\lfloor \log_2 n \rfloor$.

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice k representa cada iteración:

$$T_{bucle}(n) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle. Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) = c_2 \cdot (\lfloor \log_2 n \rfloor + 1)$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = c_2 \cdot (\lfloor \log_2 n \rfloor + 1) + c_1$$

5.2 Cálculo del orden de complejidad $O(\log n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Las constantes c_1 y c_2 son términos de orden inferior comparados con $\log_2 n$.
2. **Descartamos las constantes multiplicativas.** Las constantes no afectan al crecimiento asintótico.
3. **Cambio de base del logaritmo.** Todas las funciones logarítmicas pertenecen al mismo orden de complejidad, independientemente de la base.

El término dominante es $\log n$, por lo que el algoritmo tiene una complejidad **logarítmica**.

$$T(n) \in \mathcal{O}(\log n)$$

Ejercicio 6

```
function ejercicio6(int n)
    for (int i = 1; i <= n; i = i * 3)      // Línea 1
        print(i)                            // Línea 2
    end_for                                 // Línea 3
    return n                                // Línea 4
end_function
```

6.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Donde:

- $T_{bucle}(n)$ es el tiempo de ejecución del bucle (líneas 1-3).
- $T_{return}(n)$ es el tiempo de ejecución de la línea 4.

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para determinar el número de iteraciones del bucle (líneas 1-3), analizamos la progresión geométrica de los valores que toma la variable i:

- **Iteración 0:** $i = 1 = 3^0$
- **Iteración 1:** $i = 3 = 3^1$
- **Iteración 2:** $i = 9 = 3^2$
- **...**
- **Iteración k:** $i = 3^k$

El bucle continúa mientras se cumpla la condición $i \leq n$. Para despejar el número de iteraciones en la última vuelta (k), resolvemos la siguiente inecuación:

$$\begin{aligned} 3^k &\leq n \\ \log_3(3^k) &\leq \log_3(n) \\ k &\leq \log_3 n \end{aligned}$$

Por las propiedades de los logaritmos $\log_a(a^x) = x$.

El número total de iteraciones es $\lfloor \log_3 n \rfloor + 1$, contando desde la primera iteración $k = 0$ hasta la última $\lfloor \log_3 n \rfloor$.

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice k representa

cada iteración:

$$T_{bucle}(n) = \sum_{k=0}^{\lfloor \log_3 n \rfloor} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle. Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) = c_2 \cdot (\lfloor \log_3 n \rfloor + 1)$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = c_2 \cdot (\lfloor \log_3 n \rfloor + 1) + c_1$$

6.2 Cálculo del orden de complejidad $O(\log n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Las constantes c_1 y c_2 son términos de orden inferior comparados con $\log_3 n$.
2. **Descartamos las constantes multiplicativas.** Las constantes no afectan al crecimiento asintótico.
3. **Cambio de base del logaritmo.** Todas las funciones logarítmicas pertenecen al mismo orden de complejidad, independientemente de la base (recordemos que $\log_3 n = \frac{\log_2 n}{\log_2 3}$, siendo $\log_2 3$ una constante).

El término dominante es $\log n$, por lo que el algoritmo tiene una complejidad **logarítmica**.

$$T(n) \in \mathcal{O}(\log n)$$

2. Bucles anidados independientes

Ejercicio 7

```
function ejercicio7(int n)
    for (int i = 0; i < n/2; i++)          // Línea 1
        for(int j = 0; j < n/2; j++)       // Línea 2
            print(i + j)                   // Línea 3
        end_for                            // Línea 4
    end_for                                // Línea 5
    return n                               // Línea 6
end_function
```

7.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle anidado y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Como $T_{return}(n) = c_1$, la expresión es:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos la estructura anidada. El tiempo total es la suma de las ejecuciones del bucle interno multiplicada por las iteraciones del bucle externo.

Para determinar el número de iteraciones de ambos bucles:

- **Bucle externo (i):** Va de 0 a $n/2-1$. El número de iteraciones es $\lfloor n/2 \rfloor$.
- **Bucle interno (j):** Va de 0 a $n/2-1$. El número de iteraciones es $\lfloor n/2 \rfloor$.

Podemos expresar el tiempo de ejecución total como un sumatorio anidado:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=1}^{\lfloor n/2 \rfloor} c_2$$

Donde c_2 es el tiempo constante de la operación dentro del bucle interno. Resolviendo el sumatorio interno:

$$\sum_{j=1}^{\lfloor n/2 \rfloor} c_2 = \lfloor n/2 \rfloor \cdot c_2$$

Sustituyendo en el sumatorio externo:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} (\lfloor n/2 \rfloor \cdot c_2) = \lfloor n/2 \rfloor \cdot \lfloor n/2 \rfloor \cdot c_2 \approx \frac{n^2}{4} c_2$$

Sustituyendo en la expresión original:

$$T(n) = \frac{c_2}{4}n^2 + c_1$$

7.2 Cálculo del orden de complejidad $O(n^2)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** La constante c_1 es de orden inferior frente a n^2.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{4}$ no afecta al crecimiento asintótico.

El término dominante es n^2, por lo que el algoritmo tiene una complejidad **cuadrática**.

$$T(n) \in \mathcal{O}(n^2)$$

Ejercicio 8

```
function ejercicio8(int n)
    int m = 3000                              // Línea 1
    for (int i = 0; i < n/2; i++)             // Línea 2
        for(int j = 0; j < m * 1000; j++)     // Línea 3
            print(i + j + m)                  // Línea 4
        end_for                               // Línea 5
    end_for                                   // Línea 6
    return n                                  // Línea 7
end_function
```

8.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución de la inicialización, el bucle anidado y el retorno:

$$T(n) = T_{init}(n) + T_{bucle}(n) + T_{return}(n)$$

Donde $T_{init}(n) = c_0$ y $T_{return}(n) = c_1$. La expresión se simplifica a:

$$T(n) = c_0 + T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle:

- **Bucle externo (i):** Va de 0 a $n/2 - 1$. El número de iteraciones es $\lfloor n/2 \rfloor$.
- **Bucle interno (j):** Va de 0 a $m \cdot 1000 - 1$. Como $m = 3000$, el número de iteraciones es $3{,}000 \cdot 1{,}000 = 3{,}000{,}000$.

Es muy importante observar que el número de iteraciones del bucle interno **no depende de** n. Por tanto, el tiempo de ejecución del bucle interno es una constante $C = 3{,}000{,}000 \cdot c_2$.

Podemos expresar el tiempo de ejecución del bucle total como:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} C = \lfloor n/2 \rfloor \cdot C$$

Sustituyendo los valores constantes:

$$T_{bucle}(n) = \lfloor n/2 \rfloor \cdot (3{,}000{,}000 \cdot c_2)$$

La expresión final de $T(n)$ es:

$$T(n) = \frac{3{,}000{,}000 \cdot c_2}{2} n + (c_0 + c_1)$$

8.2 Cálculo del orden de complejidad $O(n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** La constante $(c_0 + c_1)$ es despreciable frente al término que contiene n.
2. **Descartamos las constantes multiplicativas.** Aunque el factor multiplicativo $\frac{3{,}000{,}000 \cdot c_2}{2}$ sea muy grande, sigue siendo una constante que no escala con el tamaño de la entrada n.

El término dominante es n, por lo que el algoritmo tiene una complejidad **lineal**.

$$T(n) \in \mathcal{O}(n)$$

Ejercicio 9

```
function ejercicio9(int n)
    int m = n / 10000                       // Línea 1
    for (int i = 0; i < n/2; i++)           // Línea 2
        for(int j = 0; j < m; j += 20)      // Línea 3
            print(i + j)                    // Línea 4
        end_for                             // Línea 5
    end_for                                 // Línea 6
    return n                                // Línea 7
end_function
```

9.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución de la inicialización, el bucle anidado y el retorno:

$$T(n) = T_{init}(n) + T_{bucle}(n) + T_{return}(n)$$

Donde $T_{init}(n) = c_0$ y $T_{return}(n) = c_1$.

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle:

- **Bucle externo (i):** Va de 0 a $n/2 - 1$. El número de iteraciones es $\lfloor n/2 \rfloor$.
- **Bucle interno (j):** Va de 0 a $m - 1$ con incrementos de 20. Dado que $m = n/10000$, el número de iteraciones es $\lfloor \frac{(n/10000 - 1) - 0}{20} + 1 \rfloor$.

Podemos expresar el tiempo de ejecución del bucle total como un sumatorio anidado:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=1}^{\lfloor \frac{n/10000 - 1}{20} + 1 \rfloor} c_2$$

Resolviendo el sumatorio interno:

$$\sum_{j=1}^{\lfloor \frac{n/10000 - 1}{20} + 1 \rfloor} c_2 = \left(\frac{n/10000 - 1}{20} + 1 \right) \cdot c_2$$

Sustituyendo en el sumatorio externo:

$$T_{bucle}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} \left(\frac{n}{200000} + 1 \right) \cdot c_2 = \lfloor n/2 \rfloor \cdot \left(\frac{n}{200000} + 1 \right) \cdot c_2 \approx \frac{n^2}{400000} c_2 + \frac{n}{2} c_2$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) \approx \frac{c_2}{400000} n^2 + \frac{c_2}{2} n + (c_0 + c_1)$$

9.2 Cálculo del orden de complejidad $O(n^2)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Los términos n y las constantes son despreciables frente al crecimiento de n^2.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{400000}$ no afecta al crecimiento asintótico.

El término dominante es n^2, por lo que el algoritmo tiene una complejidad **cuadrática**.

$$T(n) \in \mathcal{O}(n^2)$$

Ejercicio 10

```
function ejercicio10(int n)
    int m = n * n                          // Línea 1
    for (int i = n/2; i > 1; i = i / 3)    // Línea 2
        for(int j = 0; j < m; j++)         // Línea 3
            print(i + j)                   // Línea 4
        end_for                            // Línea 5
    end_for                                // Línea 6
    return n                               // Línea 7
end_function
```

10.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución de la inicialización, el bucle anidado y el retorno:

$$T(n) = T_{init}(n) + T_{bucle}(n) + T_{return}(n)$$

Donde $T_{init}(n) = c_0$ y $T_{return}(n) = c_1$. La expresión se simplifica a:

$$T(n) = c_0 + T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle:

- **Bucle externo (i):** Analizamos la progresión geométrica decreciente de los valores que toma la variable i, con valor inicial $n/2$ y dividiéndose por 3 en cada paso ($n/2$, $\frac{n/2}{3}$, ..., $\frac{n/2}{3^k}$). El bucle continúa mientras $i>1$. Despejando el número de iteraciones en la última vuelta (k):

  $$\frac{n/2}{3^k} > 1$$
  $$n/2 > 3^k$$
  $$\log_3(n/2) > k$$

  El número de iteraciones es $\lfloor \log_3(n/2) \rfloor + 1$.

  > Por las propiedades de los logaritmos $\log_a(a^x) = x$.

- **Bucle interno (j):** Va de 0 a $m-1$ con incrementos de 1. Como $m = n^2$, el número de iteraciones es n^2.

Podemos expresar el tiempo de ejecución del bucle total como un sumatorio anidado:

$$T_{bucle}(n) = \sum_{k=0}^{\lfloor \log_3(n/2) \rfloor} \sum_{j=0}^{n^2-1} c_2$$

Resolviendo el sumatorio interno:

$$\sum_{j=0}^{n^2-1} c_2 = n^2 \cdot c_2$$

Sustituyendo en el sumatorio externo:

$$T_{bucle}(n) = \sum_{k=0}^{\lfloor \log_3(n/2) \rfloor} (n^2 \cdot c_2) = (\lfloor \log_3(n/2) \rfloor + 1) \cdot n^2 \cdot c_2$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) = c_2 \cdot n^2 \cdot (\lfloor \log_3(n/2) \rfloor + 1) + c_0 + c_1$$

10.2 Cálculo del orden de complejidad $O(n^2 \log n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Las constantes c_0 y c_1 son despreciables frente al término dominante que incluye n^2 y el logaritmo. Además, la constante aditiva $+1$ del logaritmo es de menor orden.
2. **Descartamos las constantes multiplicativas.** El factor c_2 no afecta al crecimiento asintótico.
3. **Cambio de base y constantes en el logaritmo.** Recordamos que $\log_3(n/2) = \log_3(n) - \log_3(2)$. La resta de una constante no altera el orden asintótico, y la base del logaritmo se omite en la notación de orden de complejidad.

El término dominante es $n^2 \log n$, por lo que el algoritmo tiene una complejidad **cuadrática logarítmica**.

$$T(n) \in \mathcal{O}(n^2 \log n)$$

Ejercicio 11

```
function ejercicio11(int n)
    int m = 2147483647; //MAX_INTEGER       // Línea 1
    for (int i = 0; i < m; i++)             // Línea 2
        for(int j = m; j < m; j = j / 2)    // Línea 3
            print(i + j)                    // Línea 4
        end_for                             // Línea 5
    end_for                                 // Línea 6
    return n                                // Línea 7
end_function
```

11.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución de la inicialización, el bucle anidado y el retorno:

$$T(n) = T_{init}(n) + T_{bucle}(n) + T_{return}(n)$$

Donde $T_{init}(n) = c_0$ y $T_{return}(n) = c_1$. La expresión se simplifica a:

$$T(n) = c_0 + T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle:

- **Bucle interno (j):** Su inicialización es `j = m` y su condición de continuación es `j < m`. Como la variable `j` comienza con un valor igual a `m`, la condición `j < m` es **falsa** desde la primera evaluación. Por lo tanto, el cuerpo del bucle interno (línea 4) **no se ejecuta nunca** (0 iteraciones). Sin embargo, evaluar la condición del bucle interno en cada iteración del bucle externo tiene un coste constante c_2.
- **Bucle externo (i):** Va de `0` a $m-1$ con incrementos de 1. El número de iteraciones es exactamente m. Como $m = 2147483647$, este es un valor enorme, pero **totalmente constante** (no depende de la variable de entrada n).

El bucle externo se ejecuta m veces, y en cada iteración solo se realiza la evaluación de la condición del bucle interno (c_2).

Podemos expresar el tiempo de ejecución del bucle total como un sumatorio:

$$T_{bucle}(n) = \sum_{i=0}^{m-1} c_2 = m \cdot c_2$$

Sustituyendo el valor constante de m:

$$T_{bucle}(n) = 2147483647 \cdot c_2$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) = 2147483647 \cdot c_2 + c_0 + c_1$$

11.2 Cálculo del orden de complejidad $O(1)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. El número de operaciones que realiza la función no depende del valor de n.
2. Todos los términos de la expresión $T(n) = 2147483647 \cdot c_2 + c_0 + c_1$ son valores constantes. En el análisis asintótico, toda expresión constante se agrupa y asimila a $\mathcal{O}(1)$.

Dado que el algoritmo requiere el mismo tiempo de ejecución independientemente del tamaño de la entrada, tiene una complejidad **constante**.

$$T(n) \in \mathcal{O}(1)$$

3. Bucles anidados dependientes

Ejercicio 12

```
function ejercicio12(int n)
    for (int i = 1; i <= n; i++)            // Línea 1
        for(int j = 1; j <= i; j++)         // Línea 2
            print(i + j)                    // Línea 3
        end_for                             // Línea 4
    end_for                                 // Línea 5
    return n                                // Línea 6
end_function
```

12.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle anidado y el retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Como $T_{return}(n) = c_1$, la expresión se simplifica a:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle. A diferencia de los bucles anidados independientes, en este caso el límite superior del bucle interno **depende de la variable del bucle externo**.

- **Bucle externo (i):** Va de 1 a n con incrementos de 1.
- **Bucle interno (j):** Va de 1 a i con incrementos de 1. El número de iteraciones para un valor dado de i es exactamente i.

Podemos expresar el tiempo de ejecución del bucle total mediante un doble sumatorio:

$$T_{bucle}(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} c_2$$

Donde c_2 es el coste constante de las operaciones ejecutadas en el bucle interno (línea 3).

Resolvemos primero el sumatorio interno:

$$\sum_{j=1}^{i} c_2 = c_2 \cdot i$$

Ahora sustituimos este resultado en el sumatorio externo:

$$T_{bucle}(n) = \sum_{i=1}^{n} (c_2 \cdot i) = c_2 \sum_{i=1}^{n} i$$

Recordamos que la suma de los n primeros números naturales (progresión aritmética) es $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Sustituyendo la fórmula anterior en la expresión del sumatorio:

$$T_{bucle}(n) = c_2 \cdot \frac{n(n+1)}{2} = \frac{c_2}{2} n^2 + \frac{c_2}{2} n$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) = \frac{c_2}{2} n^2 + \frac{c_2}{2} n + c_1$$

12.2 Cálculo del orden de complejidad $O(n^2)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Los términos proporcionales a n y las constantes (c_1) son despreciables frente al término cuadrático n^2 cuando n es grande.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{2}$ no afecta al crecimiento asintótico de la función.

El término dominante es n^2, por lo que el algoritmo tiene una complejidad **cuadrática**.

$$T(n) \in \mathcal{O}(n^2)$$

Ejercicio 13

```
function ejercicio13(int n)
    for (int i = 1; i <= n * n - 10; i++)          // Línea 1
        for(int j = 1; j <= i; j++)                // Línea 2
            print(i + j)                           // Línea 3
        end_for                                    // Línea 4
    end_for                                        // Línea 5
    return n                                       // Línea 6
end_function
```

13.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle anidado y el retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Como $T_{return}(n) = c_1$, la expresión se simplifica a:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle. A diferencia de los bucles anidados independientes, en este caso el límite superior del bucle interno **depende de la variable del bucle externo**.

- **Bucle externo (i):** Va de 1 a $n^2 - 10$ con incrementos de 1.
- **Bucle interno (j):** Va de 1 a i con incrementos de 1. El número de iteraciones para un valor dado de i es exactamente i.

Podemos expresar el tiempo de ejecución del bucle total mediante un doble sumatorio:

$$T_{bucle}(n) = \sum_{i=1}^{n^2-10} \sum_{j=1}^{i} c_2$$

Donde c_2 es el coste constante de las operaciones ejecutadas en el bucle interno (línea 3).

Resolvemos primero el sumatorio interno:

$$\sum_{j=1}^{i} c_2 = c_2 \cdot i$$

Ahora sustituimos este resultado en el sumatorio externo:

$$T_{bucle}(n) = \sum_{i=1}^{n^2-10} (c_2 \cdot i) = c_2 \sum_{i=1}^{n^2-10} i$$

Recordamos que la suma de los m primeros números naturales (progresión

aritmética) es $\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$.

Aplicando la fórmula para el límite $m = n^2 - 10$:

$$T_{bucle}(n) = c_2 \cdot \frac{(n^2-10)((n^2-10)+1)}{2} = c_2 \cdot \frac{(n^2-10)(n^2-9)}{2}$$

Desarrollando el producto polinómico:

$$T_{bucle}(n) = \frac{c_2}{2}(n^4 - 9n^2 - 10n^2 + 90) = \frac{c_2}{2}n^4 - \frac{19c_2}{2}n^2 + 45c_2$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) = \frac{c_2}{2}n^4 - \frac{19c_2}{2}n^2 + 45c_2 + c_1$$

13.2 Cálculo del orden de complejidad $O(n^4)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Los términos proporcionales a n^2 y las constantes ($45c_2 + c_1$) son despreciables frente al término polinómico de mayor grado n^4 cuando n es grande.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{2}$ no afecta al crecimiento asintótico de la función.

El término dominante es n^4, por lo que el algoritmo tiene una complejidad **polinómica de grado 4**.

$$T(n) \in \mathcal{O}(n^4)$$

Ejercicio 14

```
function ejercicio14(int n)
    for (int i = 1; i <= n; i++)                    // Línea 1
        for(int j = 1; j <= i; j++)                 // Línea 2
            for(int k = 1; k <= i; k++)             // Línea 3
                print(i + j + k)                    // Línea 4
            end_for                                 // Línea 5
        end_for                                     // Línea 6
    end_for                                         // Línea 7
    return n                                        // Línea 8
end_function
```

14.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle anidado y el retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Como $T_{return}(n) = c_1$, la expresión se simplifica a:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle. A diferencia de los bucles anidados independientes, en este caso los límites superiores de los bucles interno e intermedio **dependen de la variable del bucle externo**.

- **Bucle externo (i):** Va de 1 a n con incrementos de 1.
- **Bucle intermedio (j):** Va de 1 a i con incrementos de 1. Su número de iteraciones es i.
- **Bucle interno (k):** Va de 1 a i con incrementos de 1. Su número de iteraciones es i.

Dado que los bucles `j` y `k` son independientes entre sí pero ambos dependen de `i`, el número total de iteraciones de los dos bucles más internos para un valor dado de i es $i \cdot i = i^2$.

Podemos expresar el tiempo de ejecución del bucle total mediante un triple sumatorio:

$$T_{bucle}(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{i} c_2$$

Donde c_2 es el coste constante de las operaciones ejecutadas en el bucle interno (línea 4).

Resolvemos los sumatorios de adentro hacia afuera. Primero el sumatorio interno en k:

$$\sum_{k=1}^{i} c_2 = c_2 \cdot i$$

Luego el sumatorio intermedio en j:

$$\sum_{j=1}^{i}(c_2 \cdot i) = c_2 \cdot i \sum_{j=1}^{i} 1 = c_2 \cdot i \cdot i = c_2 \cdot i^2$$

Ahora sustituimos este resultado en el sumatorio externo en i:

$$T_{bucle}(n) = \sum_{i=1}^{n}(c_2 \cdot i^2) = c_2 \sum_{i=1}^{n} i^2$$

Recordamos que la suma de los cuadrados de los n primeros números naturales es $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

Aplicando esta fórmula:

$$T_{bucle}(n) = c_2 \cdot \frac{n(n+1)(2n+1)}{6} = c_2 \cdot \frac{2n^3 + 3n^2 + n}{6}$$

Desarrollando la fracción:

$$T_{bucle}(n) = \frac{c_2}{3}n^3 + \frac{c_2}{2}n^2 + \frac{c_2}{6}n$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) = \frac{c_2}{3}n^3 + \frac{c_2}{2}n^2 + \frac{c_2}{6}n + c_1$$

14.2 Cálculo del orden de complejidad $O(n^3)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Los términos proporcionales a n^2, n y las constantes (c_1) son despreciables frente al término polinómico de mayor grado n^3 cuando n es grande.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{3}$ no afecta al crecimiento asintótico de la función.

El término dominante es n^3, por lo que el algoritmo tiene una complejidad **cúbica** (polinómica de grado 3).

$$T(n) \in \mathcal{O}(n^3)$$

Ejercicio 15

```
function ejercicio15(int n)
    for (int i = 1; i <= n; i++)                    // Línea 1
        for(int j = 1; j <= i; j++)                 // Línea 2
            for(int k = 1; k <= j; k++)             // Línea 3
                print(i + j + k)                    // Línea 4
            end_for                                 // Línea 5
        end_for                                     // Línea 6
    end_for                                         // Línea 7
    return n                                        // Línea 8
end_function
```

15.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle anidado y el retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Como $T_{return}(n) = c_1$, la expresión se simplifica a:

$$T(n) = T_{bucle}(n) + c_1$$

Para calcular $T_{bucle}(n)$, analizamos las iteraciones de cada bucle. A diferencia de los bucles anidados independientes, en este caso los límites superiores de los bucles interno e intermedio **dependen de la variable del bucle externo**.

- **Bucle externo (i):** Va de 1 a n con incrementos de 1.
- **Bucle intermedio (j):** Va de 1 a i con incrementos de 1.
- **Bucle interno (k):** Va de 1 a j con incrementos de 1. Su número de iteraciones depende del valor de j.

Dado que los bucles están anidados y cada uno depende del anterior, el número total de iteraciones se calcula mediante un triple sumatorio:

$$T_{bucle}(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} c_2$$

Donde c_2 es el coste constante de las operaciones ejecutadas en el bucle interno (línea 4).

Resolvemos los sumatorios de adentro hacia afuera. Primero el sumatorio interno en k:

$$\sum_{k=1}^{j} c_2 = c_2 \cdot j$$

Luego el sumatorio intermedio en j:

$$\sum_{j=1}^{i}(c_2 \cdot j) = c_2 \sum_{j=1}^{i} j = c_2 \cdot \frac{i(i+1)}{2} = \frac{c_2}{2}(i^2+i)$$

Ahora sustituimos este resultado en el sumatorio externo en i:

$$T_{bucle}(n) = \sum_{i=1}^{n} \frac{c_2}{2}(i^2+i) = \frac{c_2}{2}\left(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i\right)$$

Recordamos las fórmulas de las sumas:

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

Aplicando las fórmulas:

$$T_{bucle}(n) = \frac{c_2}{2}\left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right)$$

Para simplificar esta expresión, realizamos los siguientes pasos algebraicos.

Factorizamos el término común $n(n+1)$:

$$T_{bucle}(n) = \frac{c_2 \cdot n(n+1)}{2}\left(\frac{2n+1}{6} + \frac{1}{2}\right)$$

Sumamos las fracciones dentro del paréntesis (denominador común 6):

$$\frac{2n+1}{6} + \frac{3}{6} = \frac{2n+4}{6} = \frac{2(n+2)}{6} = \frac{n+2}{3}$$

Multiplicamos los resultados para obtener la fórmula compacta:

$$T_{bucle}(n) = \frac{c_2 \cdot n(n+1)}{2} \cdot \frac{n+2}{3} = c_2 \cdot \frac{n(n+1)(n+2)}{6}$$

Desarrollamos el polinomio y distribuimos el denominador:

$$T_{bucle}(n) = c_2 \cdot \frac{n^3+3n^2+2n}{6} = \frac{c_2}{6}n^3 + \frac{c_2}{2}n^2 + \frac{c_2}{3}n$$

Sustituyendo en la expresión original de $T(n)$:

$$T(n) = \frac{c_2}{6}n^3 + \frac{c_2}{2}n^2 + \frac{c_2}{3}n + c_1$$

15.2 Cálculo del orden de complejidad $O(n^3)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** Los términos proporcionales a n^2, n y las constantes (c_1) son despreciables frente al término polinómico de mayor grado n^3 cuando n es

grande.

2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{6}$ no afecta al crecimiento asintótico de la función.

El término dominante es n^3, por lo que el algoritmo tiene una complejidad **cúbica** (polinómica de grado 3).

$$T(n) \in \mathcal{O}(n^3)$$

4. Bucles con while

Ejercicio 16

```
function ejercicio16(int n)
    while (n > 1)                    // Línea 1
        print(n)                     // Línea 2
        n = n / 2                    // Línea 3
    end_while                        // Línea 4
    return 0                         // Línea 5
end_function
```

16.1 Cálculo del $T(n)$

El tiempo de ejecución de la función viene dado por la suma del tiempo de ejecución del bucle y el tiempo de ejecución de la operación de retorno:

$$T(n) = T_{bucle}(n) + T_{return}(n)$$

Como $T_{return}(n) = c_1$, donde c_1 es el tiempo constante de la operación de retorno:

$$T(n) = T_{bucle}(n) + c_1$$

Para determinar el número de iteraciones del bucle `while` (líneas 1-4), analizamos la progresión geométrica decreciente de los valores que toma la variable n:

- **Iteración 0:** n
- **Iteración 1:** $n / 2$
- **Iteración 2:** $n / 4$
- **...**
- **Iteración k:** $n / 2^k$

El bucle continúa iterando mientras se cumpla la condición $n>1$. Para despejar el número de iteraciones en la última vuelta (k), establecemos la inecuación para el límite de iteraciones:

$$\frac{n}{2^k}>1$$
$$n>2^k$$
$$\log_2(n)>\log_2(2^k)$$
$$\log_2 n>k$$

Por las propiedades de los logaritmos $\log_a(a^x) = x$.

El número total de iteraciones está acotado por $\lfloor \log_2 n \rfloor$.

Podemos expresar el tiempo de ejecución del bucle como un sumatorio donde el índice k representa

cada iteración:

$$T_{bucle}(n) = \sum_{k=1}^{\lfloor \log_2 n \rfloor} c_2$$

Donde c_2 es el tiempo constante de las operaciones dentro del bucle (líneas 2 y 3). Resolviendo el sumatorio obtenemos:

$$T_{bucle}(n) = c_2 \cdot \lfloor \log_2 n \rfloor$$

Sustituyendo en la expresión original:

$$T(n) = T_{bucle}(n) + c_1 = c_2 \cdot \lfloor \log_2 n \rfloor + c_1$$

16.2 Cálculo del orden de complejidad $O(\log n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico:

1. **Descartamos los términos de menor orden.** La constante c_1 es un término de orden inferior comparado con $\log_2 n$.
2. **Descartamos las constantes multiplicativas.** La constante c_2 no afecta al crecimiento asintótico.
3. **Cambio de base del logaritmo.** Todas las funciones logarítmicas pertenecen al mismo orden de complejidad, independientemente de la base.

El término dominante es $\log n$, por lo que el algoritmo tiene una complejidad **logarítmica**.

$$T(n) \in \mathcal{O}(\log n)$$

5. Algoritmos recursivos

Ejercicio 17

```
long int factorial(long int n) {
    if (n <= 1)                         // Línea 1
        return 1;                       // Línea 2
    else                                // Línea 3
        return n * factorial(n - 1);    // Línea 4
}
```

17.1 Cálculo de la ecuación de recurrencia

El tiempo de ejecución de un algoritmo recursivo se expresa mediante una **ecuación de recurrencia**, donde se separa el tiempo de ejecución del caso base del tiempo del caso recursivo.

Para la función `factorial(n)`, consideramos:

- c_1: Tiempo de ejecución del caso base ($n \leq 1$).
- c_2: Tiempo constante de las operaciones realizadas en cada llamada recursiva (comparaciones, resta, multiplicación y retorno).

La ecuación de recurrencia queda expresada como:

$$T(n) = \begin{cases} c_1 & \text{si} n \leq 1 \\ T(n-1) + c_2 & \text{si} n > 1 \end{cases}$$

17.2 Resolución de la ecuación $T(n)$

Para encontrar una expresión no recursiva de $T(n)$, utilizamos el **método de sustitución**:

1. Partimos de la expresión original para el caso recursivo:

$$T(n) = T(n-1) + c_2 \quad (1)$$

2. Obtenemos la expresión de $T(n-1)$ sustituyendo n por $n-1$ en la ecuación original:

$$T(n-1) = T(n-1-1) + c_2 = T(n-2) + c_2 \quad (2)$$

3. Sustituimos la ecuación (2) en la (1):

$$T(n) = (T(n-2) + c_2) + c_2 = T(n-2) + 2c_2 \quad (3)$$

4. Repetimos el proceso para $T(n-2)$:

$$T(n-2) = T(n-3) + c_2 \quad (4)$$

$$T(n) = (T(n-3) + c_2) + 2c_2 = T(n-3) + 3c_2 \quad (5)$$

5. Podemos inferir que tras i sustituciones, el tiempo de ejecución está dado por:

$$T(n) = T(n-i) + i \cdot c_2 \quad (6)$$

6. El proceso de sustitución termina cuando alcanzamos el **caso base** ($n - i = 1$). Despejando i:

$$i = n-1$$

7. Sustituimos el valor de i en la ecuación (6):

$$T(n) = T(1) + (n-1)c_2 \quad (7)$$

8. Como sabemos que $T(1) = c_1$ (caso base), la expresión final es:

$$T(n) = c_1 + (n-1)c_2 = c_2 n + (c_1 - c_2) \quad (8)$$

17.3 Cálculo del orden de complejidad $O(n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico sobre la expresión obtenida:

1. **Descartamos los términos de menor orden.** En la expresión $c_2 n + (c_1 - c_2)$, el término dominante es el que depende de n. La constante $(c_1 - c_2)$ es despreciable cuando n es grande.
2. **Descartamos las constantes multiplicativas.** El factor c_2 no afecta a la tasa de crecimiento de la función.

El término dominante es n, por lo que el algoritmo tiene una complejidad **lineal**.

$$T(n) \in \mathcal{O}(n)$$

Ejercicio 18

```
int recursiva1(int n) {
    if (n <= 1)                         // Línea 1
        return 1;                       // Línea 2
    else                                // Línea 3
        return 2 * recursiva1(n / 2);   // Línea 4
}
```

18.1 Cálculo de la ecuación de recurrencia

Para la función `recursiva1(n)`, definimos:

- c_1: Tiempo de ejecución del caso base ($n \leq 1$).
- c_2: Tiempo constante de las operaciones realizadas en cada llamada recursiva (comparación, división por 2, multiplicación por 2 y retorno).

La ecuación de recurrencia queda expresada como:

$$T(n) = \begin{cases} c_1 & \text{si} n \leq 1 \\ T(n/2) + c_2 & \text{si} n > 1 \end{cases}$$

18.2 Resolución de la ecuación $T(n)$

Para encontrar una expresión no recursiva de $T(n)$, utilizamos el **método de sustitución**:

1. Partimos de la expresión original para el caso recursivo:

$$T(n) = T(n/2) + c_2 \quad (1)$$

2. Obtenemos la expresión de $T(n/2)$ sustituyendo n por $n/2$ en la ecuación original:

$$T(n/2) = T((n/2)/2) + c_2 = T(n/2^2) + c_2 \quad (2)$$

3. Sustituimos la ecuación (2) en la (1):

$$T(n) = (T(n/2^2) + c_2) + c_2 = T(n/2^2) + 2c_2 \quad (3)$$

4. Repetimos el proceso para el siguiente paso:

$$T(n/2^2) = T(n/2^3) + c_2 \quad (4)$$

$$T(n) = (T(n/2^3) + c_2) + 2c_2 = T(n/2^3) + 3c_2 \quad (5)$$

5. Podemos inferir que tras i sustituciones, el tiempo de ejecución está dado por:

$$T(n) = T(n/2^i) + i \cdot c_2 \quad (6)$$

6. El proceso de sustitución termina cuando alcanzamos el **caso base** ($n/2^i = 1$). Despejamos i:

$$2^i = n \Rightarrow i = \log_2 n$$

7. Sustituimos el valor de i en la ecuación (6):

$$T(n) = T(1) + (\log_2 n) \cdot c_2 \quad (7)$$

8. Como sabemos que $T(1) = c_1$ (caso base), la expresión final es:

$$T(n) = c_1 + c_2 \log_2 n \quad (8)$$

18.3 Cálculo del orden de complejidad $O(\log n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico sobre la expresión obtenida:

1. **Descartamos los términos de menor orden.** La constante c_1 es un término de orden inferior comparado con $\log_2 n$.
2. **Descartamos las constantes multiplicativas.** El factor c_2 no afecta a la tasa de crecimiento asintótico.
3. **Cambio de base del logaritmo.** Todas las funciones logarítmicas pertenecen al mismo orden de complejidad, independientemente de la base del logaritmo.

El término dominante es $\log n$, por lo que el algoritmo tiene una complejidad **logarítmica**.

$$T(n) \in \mathcal{O}(\log n)$$

Ejercicio 19

```
int recursiva2(int n, int x) {
    int i;
    if (n <= 1)                          // Línea 1
        return 1;                        // Línea 2
    else {                               // Línea 3
        for (i = 1; i <= n; i++)         // Línea 4
            x = x + 1;                   // Línea 5
        return recursiva2(n - 1, x);     // Línea 6
    }
}
```

19.1 Cálculo de la ecuación de recurrencia

Para la función `recursiva2(n, x)`, definimos:

- c_1: Tiempo de ejecución del caso base ($n \leq 1$).
- c_2: Tiempo constante asociado a cada iteración del bucle `for` (líneas 4 y 5).
- Consideramos que el resto de operaciones constantes se asimilan en el coste del bucle o en las constantes finales.

La ecuación de recurrencia es:

$$T(n) = \begin{cases} c_1 & \text{si} n \leq 1 \\ T(n-1) + c_2 n & \text{si} n > 1 \end{cases}$$

19.2 Resolución de la ecuación $T(n)$

Para encontrar una expresión no recursiva de $T(n)$, utilizamos el **método de sustitución**:

1. Partimos de la expresión original para el caso recursivo:

$$T(n) = T(n-1) + c_2 n \quad (1)$$

2. Obtenemos la expresión de $T(n-1)$ sustituyendo en la ecuación original:

$$T(n-1) = T(n-2) + c_2(n-1) \quad (2)$$

3. Sustituimos la ecuación (2) en la (1):

$$T(n) = (T(n-2) + c_2(n-1)) + c_2 n = T(n-2) + c_2(n + n - 1) \quad (3)$$

4. Repetimos el proceso para $T(n-2)$:

$$T(n-2) = T(n-3) + c_2(n-2) \quad (4)$$

$$T(n) = (T(n-3) + c_2(n-2)) + c_2(n + n - 1) = T(n-3) + c_2(n + n - 1 + n - 2) \quad (5)$$

5. Podemos inferir que tras i sustituciones, el tiempo de ejecución está dado por:

$$T(n) = T(n-i) + c_2 \sum_{k=0}^{i-1} (n-k) \quad (6)$$

6. El proceso termina cuando alcanzamos el **caso base** ($n - i = 1$), lo que implica $i = n - 1$. Sustituimos en la ecuación (6):

$$T(n) = T(1) + c_2 \sum_{k=0}^{n-2} (n-k) \quad (7)$$

7. Evaluamos el sumatorio. Representa la suma de los enteros desde n hasta 2:

$$\sum_{k=0}^{n-2} (n-k) = n + (n-1) + (n-2) + \ldots + 2 = \left(\sum_{j=1}^{n} j \right) - 1$$

8. Aplicando la fórmula de la suma de los n primeros enteros ($\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$):

$$T(n) = c_1 + c_2 \left(\frac{n(n+1)}{2} - 1 \right) = \frac{c_2}{2} n^2 + \frac{c_2}{2} n + (c_1 - c_2) \quad (8)$$

19.3 Cálculo del orden de complejidad $O(n^2)$

Para determinar el orden de complejidad, realizamos el análisis asintótico sobre la expresión obtenida:

1. **Descartamos los términos de menor orden.** Los términos proporcionales a n y las constantes son despreciables frente a n^2 cuando n es grande.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_2}{2}$ no afecta al crecimiento asintótico.

El término dominante es n^2, por lo que el algoritmo tiene una complejidad **cuadrática**.

$$T(n) \in \mathcal{O}(n^2)$$

Ejercicio 20

```
int recursiva3(int n) {
    if (n <= 1)                                      // Línea 1
        return 1;                                    // Línea 2
    else                                             // Línea 3
        return recursiva3(n - 1) + recursiva3(n - 1); // Línea 4
}
```

20.1 Cálculo de la ecuación de recurrencia

Para la función `recursiva3(n)`, definimos:

- c_1: Tiempo de ejecución del caso base ($n \leq 1$).
- c_2: Tiempo constante asociado a las operaciones en el caso recursivo (comparación, resta, suma de los resultados y retorno).

La ecuación de recurrencia es:

$$T(n) = \begin{cases} c_1 & \text{si} n \leq 1 \\ 2T(n-1) + c_2 & \text{si} n > 1 \end{cases}$$

20.2 Resolución de la ecuación $T(n)$

Para encontrar una expresión no recursiva de $T(n)$, utilizamos el **método de sustitución**:

1. Partimos de la expresión original para el caso recursivo:

$$T(n) = 2T(n-1) + c_2 \quad (1)$$

2. Obtenemos la expresión de $T(n-1)$ sustituyendo en la ecuación original:

$$T(n-1) = 2T(n-2) + c_2 \quad (2)$$

3. Sustituimos la ecuación (2) en la (1):

$$T(n) = 2(2T(n-2) + c_2) + c_2 = 2^2T(n-2) + 2c_2 + c_2 = 4T(n-2) + 3c_2 \quad (3)$$

4. Repetimos el proceso para $T(n-2)$:

$$T(n-2) = 2T(n-3) + c_2 \quad (4)$$

$$T(n) = 4(2T(n-3) + c_2) + 3c_2 = 2^3T(n-3) + 4c_2 + 3c_2 = 8T(n-3) + 7c_2 \quad (5)$$

5. Podemos inferir que tras i sustituciones, el tiempo de ejecución está dado por:

$$T(n) = 2^iT(n-i) + (2^i - 1)c_2 \quad (6)$$

6. El proceso termina cuando alcanzamos el **caso base** ($n - i = 1$), lo que implica $i = n - 1$. Sustituimos en la ecuación (6):

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1)c_2 \quad (7)$$

7. Como sabemos que $T(1) = c_1$, simplificamos la expresión:

$$T(n) = 2^{n-1}c_1 + 2^{n-1}c_2 - c_2 = 2^{n-1}(c_1 + c_2) - c_2$$

$$T(n) = \frac{c_1 + c_2}{2} \cdot 2^n - c_2 \quad (8)$$

20.3 Cálculo del orden de complejidad $O(2^n)$

Para determinar el orden de complejidad, realizamos el análisis asintótico sobre la expresión obtenida:

1. **Descartamos los términos de menor orden.** La constante c_2 es despreciable frente al crecimiento exponencial de 2^n cuando n es grande.
2. **Descartamos las constantes multiplicativas.** El factor $\frac{c_1 + c_2}{2}$ no afecta a la tasa de crecimiento asintótico.

El término dominante es 2^n, por lo que el algoritmo tiene una complejidad **exponencial**.

$$T(n) \in \mathcal{O}(2^n)$$

6. Referencias

- Algorithm analysis: Exercises with solutions for AAHPS tutorials. Matej Pičulin.
- Analysis of Algorithms: Practice Examples. Princeton University. Ibrahim Albluw.
- Introducción al Análisis y al Diseño de Algoritmos. María del Carmen Gómez Fuentes, Jorge Cervantes Ojeda.

7. Licencia

Licencia **CC BY-NC-ND 4.0**