Divide & Conquer

DC - FCEN, UBA

20 de Agosto, 2025

UNIVERSIDAD DE BUENOS AIRES

Ejercicio 1 *(MergeSort)*

Dado el algoritmo de *mergesort*, implementado en el siguiente código Python:

1. Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.
2. ¿En cuántos subproblemas se divide?
3. ¿De qué tamaño son estos subproblemas?
4. ¿Cuál es el costo de combinar los resultados de los subproblemas?
5. Escribir la función $T(n)$ de manera recursiva.
6. Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Ejercicio 1 - Función merge_sort

```
def merge_sort(arr):
 if len(arr) <= 1:
  return arr

 medio = len(arr) // 2
 mitad_izq = merge_sort(arr[:medio])
 mitad_der = merge_sort(arr[medio:])

 return merge(mitad_izq, mitad_der)
```

Ejercicio 1 - Función merge

```
def merge(izq, der):
  mergeados = []
  i = j = 0

  while i < len(izq) and j < len(der):
    if izq[i] < der[j]:
      mergeados.append(izq[i])
      i += 1
    else:
      mergeados.append(der[j])
      j += 1

  mergeados.extend(izq[i:])
  mergeados.extend(der[j:])
  return mergeados
```

Ejercicio 1 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Ejercicio 1 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Respuesta:

- **Divide:**
  - `medio = len(arr) // 2` (calcular el punto medio)

Ejercicio 1 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Respuesta:

- **Divide:**
  - `medio = len(arr) // 2` (calcular el punto medio)
- **Conquer:**
  - `mitad_izq = merge_sort(arr[:medio])`
  - `mitad_der = merge_sort(arr[medio:])`

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Respuesta:

- **Divide:**
  - `medio = len(arr) // 2` (calcular el punto medio)
- **Conquer:**
  - `mitad_izq = merge_sort(arr[:medio])`
  - `mitad_der = merge_sort(arr[medio:])`
- **Combine:**
  - `return merge(mitad_izq, mitad_der)`
  - Toda la función `merge`

Pregunta 2: ¿En cuántos subproblemas se divide?

Pregunta 2: ¿En cuántos subproblemas se divide?

Respuesta: Se divide en **2 subproblemas**

- La mitad izquierda: `arr[:medio]`
- La mitad derecha: `arr[medio:]`

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Ejercicio 1 - Resolución

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Respuesta: Cada subproblema tiene tamaño n/2

- Si el arreglo original tiene n elementos

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Respuesta: Cada subproblema tiene tamaño n/2

- Si el arreglo original tiene n elementos
- La mitad izquierda tiene $\lfloor n/2 \rfloor$ elementos
- La mitad derecha tiene $\lceil n/2 \rceil$ elementos

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Respuesta: Cada subproblema tiene tamaño n/2

- Si el arreglo original tiene n elementos
- La mitad izquierda tiene $\lfloor n/2 \rfloor$ elementos
- La mitad derecha tiene $\lceil n/2 \rceil$ elementos
- Para el análisis asintótico: ambos son $\Theta(n/2)$

Ejercicio 1 - Resolución

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Ejercicio 1 - Resolución

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Respuesta: El costo de combinar es $O(n)$

- La función `merge` recorre cada elemento una sola vez

Ejercicio 1 - Resolución

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Respuesta: El costo de combinar es $O(n)$

- La función `merge` recorre cada elemento una sola vez
- En el peor caso, compara todos los elementos de ambas mitades

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Respuesta: El costo de combinar es O(n)

- La función `merge` recorre cada elemento una sola vez
- En el peor caso, compara todos los elementos de ambas mitades
- Total de operaciones: n comparaciones + n inserciones = $O(n)$

Ejercicio 1 - Resolución

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo de 1 elemento o vacío)

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo de 1 elemento o vacío)
- $2T(n/2)$: Dos llamadas recursivas de tamaño $n/2$

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo de 1 elemento o vacío)
- $2T(n/2)$: Dos llamadas recursivas de tamaño $n/2$
- $\Theta(n)$: Costo de la función `merge`

Ejercicio 1 - Resolución

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Ejercicio 1 - Resolución

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(n)$ (costo de combinar)

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Como $f(n) = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_c a})$

Ejercicio 1 - Resolución

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Como $f(n) = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Como $f(n) = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Por lo tanto: $\boxed{T(n) = \Theta(n \log n)}$

Ejercicio 2 *(BúsquedaBinaria)*

Dado el algoritmo de *búsqueda binaria*, implementado en el siguiente código Python:

1. Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.
2. ¿En cuántos subproblemas se divide?
3. ¿De qué tamaño son estos subproblemas?
4. ¿Cuál es el costo de combinar los resultados de los subproblemas?
5. Escribir la función $T(n)$ de manera recursiva.
6. Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Ejercicio 2 - Función busqueda_binaria

```
def busqueda_binaria(arr, objetivo, izq=0,
  der=len(arr)-1):
    if izq > der:
      return False  # Elemento no encontrado

    medio = (izq + der) // 2
    if arr[medio] == objetivo:
      return medio
    elif arr[medio] > objetivo:
      return busqueda_binaria(arr, objetivo,
      izq, medio - 1)
    else:
      return busqueda_binaria(arr, objetivo,
      medio + 1, der)
```

Ejercicio 2 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Ejercicio 2 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Respuesta:

- **Divide:**
  - `medio = (izq + der) // 2` (calcular el punto medio)
  - Comparación `arr[medio] > objetivo` (decidir qué mitad explorar)

Ejercicio 2 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Respuesta:

- **Divide:**
  - `medio = (izq + der) // 2` (calcular el punto medio)
  - Comparación `arr[medio] > objetivo` (decidir qué mitad explorar)
- **Conquer:**
  - `busqueda_binaria(arr, objetivo, izq, medio - 1)`
  - `busqueda_binaria(arr, objetivo, medio + 1, der)`

Ejercicio 2 - Resolución

Pregunta 1: Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles el *combine*.

Respuesta:

- **Divide:**
  - `medio = (izq + der) // 2` (calcular el punto medio)
  - Comparación `arr[medio] > objetivo` (decidir qué mitad explorar)
- **Conquer:**
  - `busqueda_binaria(arr, objetivo, izq, medio - 1)`
  - `busqueda_binaria(arr, objetivo, medio + 1, der)`
- **Combine:**
  - **No hay fase de combinación** - simplemente se retorna el resultado
  - La búsqueda binaria no necesita combinar resultados

Pregunta 2: ¿En cuántos subproblemas se divide?

Pregunta 2: ¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- A diferencia de MergeSort que explora ambas mitades

Pregunta 2: ¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- A diferencia de MergeSort que explora ambas mitades
- La búsqueda binaria solo explora **una** mitad:
  - La mitad izquierda si `arr[medio] > objetivo`
  - La mitad derecha si `arr[medio] < objetivo`

Ejercicio 2 - Resolución

Pregunta 2: ¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- A diferencia de MergeSort que explora ambas mitades
- La búsqueda binaria solo explora **una** mitad:
  - La mitad izquierda si `arr[medio] > objetivo`
  - La mitad derecha si `arr[medio] < objetivo`
- Esto es clave para su eficiencia: $O(\log n)$ vs $O(n \log n)$

Ejercicio 2 - Resolución

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Ejercicio 2 - Resolución

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Respuesta: El subproblema tiene tamaño n/2

- En cada llamada recursiva, el espacio de búsqueda se reduce a la mitad

Ejercicio 2 - Resolución

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Respuesta: El subproblema tiene tamaño n/2

- En cada llamada recursiva, el espacio de búsqueda se reduce a la mitad
- Si el arreglo original tiene n elementos:
  - Primera llamada: n elementos
  - Segunda llamada: $n/2$ elementos
  - Tercera llamada: $n/4$ elementos
  - ...hasta llegar a 1 elemento

Ejercicio 2 - Resolución

Pregunta 3: ¿De qué tamaño son estos subproblemas?

Respuesta: El subproblema tiene tamaño n/2

- En cada llamada recursiva, el espacio de búsqueda se reduce a la mitad
- Si el arreglo original tiene n elementos:
  - Primera llamada: n elementos
  - Segunda llamada: $n/2$ elementos
  - Tercera llamada: $n/4$ elementos
  - ...hasta llegar a 1 elemento
- Máximo número de llamadas: $\log_2 n$

Ejercicio 2 - Resolución

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Ejercicio 2 - Resolución

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Respuesta: El costo de combinar es $O(1)$

- No hay fase de combinación real

Ejercicio 2 - Resolución

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Respuesta: El costo de combinar es $O(1)$

- No hay fase de combinación real
- Solo se retorna directamente el resultado:
  - El índice si se encuentra el elemento
  - `False` si no se encuentra

Pregunta 4: ¿Cuál es el costo de combinar los resultados de los subproblemas?

Respuesta: El costo de combinar es O(1)

- No hay fase de combinación real
- Solo se retorna directamente el resultado:
  - El índice si se encuentra el elemento
  - `False` si no se encuentra
- Las operaciones adicionales son:
  - Calcular el punto medio: $O(1)$
  - Comparar con el objetivo: $O(1)$

Ejercicio 2 - Resolución

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo vacío o de 1 elemento).

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo vacío o de 1 elemento).
- $T(n/2)$: **Una** llamada recursiva de tamaño $n/2$.

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo vacío o de 1 elemento).
- $T(n/2)$: **Una** llamada recursiva de tamaño $n/2$.
- $\Theta(1)$: Costo de las comparaciones y cálculo del medio.

Pregunta 5: Escribir la función $T(n)$ de manera recursiva.

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (arreglo vacío o de 1 elemento).
- $T(n/2)$: **Una** llamada recursiva de tamaño $n/2$.
- $\Theta(1)$: Costo de las comparaciones y cálculo del medio.

Nota: Solo hay **una** llamada recursiva, no dos como en MergeSort

Ejercicio 2 - Resolución

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 1$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(1)$ (costo de dividir y decidir)

Ejercicio 2 - Resolución

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 1$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(1)$ (costo de dividir y decidir)

Calculamos: $\log_c a = \log_2 1 = 0$

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 1$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(1)$ (costo de dividir y decidir)

Calculamos: $\log_c a = \log_2 1 = 0$

Como $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_c a})$

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 1$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(1)$ (costo de dividir y decidir)

Calculamos: $\log_c a = \log_2 1 = 0$

Como $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Pregunta 6: Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Respuesta: Aplicamos el Teorema Maestro con:

- $a = 1$ (número de subproblemas)
- $c = 2$ (factor de división)
- $f(n) = \Theta(1)$ (costo de dividir y decidir)

Calculamos: $\log_c a = \log_2 1 = 0$

Como $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Por lo tanto: $\boxed{T(n) = \Theta(n^0 \log n) = \Theta(\log n)}$

Problema: Búsqueda Lineal Modificada

Definición

Dado un arreglo A y un elemento *elem*, determinar si *elem* está presente en A.

Pregunta clave

¿Cualquier algoritmo recursivo que divide el problema es *divide and conquer*?

Respuesta: NO

Primera versión: BúsquedaLínealModificada

Algoritmo

```
BúsquedaLínealModificada(A, elem)
  If |A| == 0 return false
  If |A| == 1 and elem == A[0] return true
  If |A| == 1 and elem != A[0] return false
  return BúsquedaLínealModificada(subarray(A, 0, 1), elem)
         OR
         BúsquedaLínealModificada(subarray(A, 1, |A|), elem)
```

Primera versión: BúsquedaLínealModificada

Algoritmo

```
BúsquedaLínealModificada(A, elem)
  If |A| == 0 return false
  If |A| == 1 and elem == A[0] return true
  If |A| == 1 and elem != A[0] return false
  return BúsquedaLínealModificada(subarray(A, 0, 1), elem)
         OR
         BúsquedaLínealModificada(subarray(A, 1, |A|), elem)
```

División del Problema

- Subproblema 1: Tamaño $\mathbf{1}$ (un elemento).
- Subproblema 2: Tamaño $n-1$ (resto del arreglo).

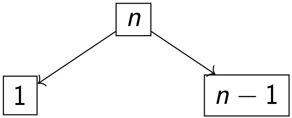
¿Por qué NO es Divide and Conquer?

Características de D&C

1. **Dividir** en subproblemas más pequeños.
2. Subproblemas de tamaño considerablemente menor.
3. División balanceada (idealmente).
4. Reducción significativa del tamaño.

BúsquedaLínealModificada

- División: $n \to 1 + (n-1)$
- Subproblema principal: $n-1$
- Reducción: solo 1 elemento
- División desbalanceada



Análisis de Complejidad - Primera Versión

Recurrencia

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

Análisis de Complejidad - Primera Versión

Recurrencia

$$
\begin{aligned}
T(n) &= T(1) + T(n-1) + O(1) \\
T(n) &= T(n-1) + O(1)
\end{aligned}
$$

Resolución

$$
\begin{aligned}
T(n) &= T(n-1) + O(1) \\
&= T(n-2) + O(1) + O(1) \\
&= T(n-3) + 3 \cdot O(1) \\
&\vdots \\
&= T(1) + (n-1) \cdot O(1) \\
&= O(n)
\end{aligned}
$$

¡Es simplemente una recursión lineal disfrazada!

Algoritmo

```
BúsquedaLínealModV2(A, elem)
  If |A| == 0 return false
  If |A| == 1 and elem == A[0] return true
  If |A| == 1 and elem != A[0] return false
  return BúsquedaLínealModV2(subarray(A, 0, |A|/2), elem)
        OR
        BúsquedaLínealModV2(subarray(A, |A|/2, |A|), elem)
```

Segunda Versión: BúsquedaLínealModV2

Algoritmo

```
BúsquedaLínealModV2(A, elem)
  If |A| == 0 return false
  If |A| == 1 and elem == A[0] return true
  If |A| == 1 and elem != A[0] return false
  return BúsquedaLínealModV2(subarray(A, 0, |A|/2), elem)
        OR
        BúsquedaLínealModV2(subarray(A, |A|/2, |A|), elem)
```

División del Problema

- Subproblema 1: Tamaño $n/2$
- Subproblema 2: Tamaño $n/2$

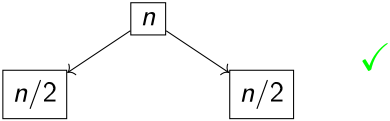
¿Por qué SÍ es Divide and Conquer?

Verificación D&C

1. ✓División en subproblemas
2. ✓Tamaño $n/2$ (fracción del original)
3. ✓División balanceada
4. ✓Reducción significativa

BúsquedaLínealModificadaV2

- División: $n \to n/2 + n/2$
- Reducción: 50 % en cada nivel
- División perfectamente balanceada
- Cumple paradigma D&C



Análisis de Complejidad - Segunda Versión

Recurrencia

$$T(n) = 2 \cdot T(n/2) + O(1)$$

Recurrencia

$$T(n) = 2 \cdot T(n/2) + O(1)$$

Aplicando el Teorema Maestro

- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de reducción)
- $f(n) = O(1)$
- $\log_c a = \log_2 2 = 1$

Análisis de Complejidad - Segunda Versión

Recurrencia

$$T(n) = 2 \cdot T(n/2) + O(1)$$

Aplicando el Teorema Maestro

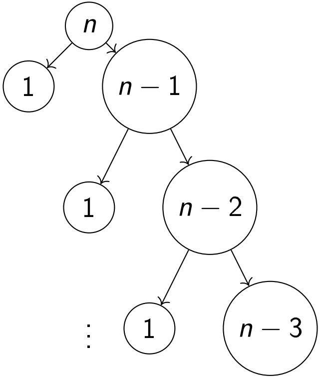
- $a = 2$ (número de subproblemas)
- $c = 2$ (factor de reducción)
- $f(n) = O(1)$
- $\log_c a = \log_2 2 = 1$

Resultado Como $f(n) = O(1) = O(n^0)$ y $\log_c a = 1 > 0$:

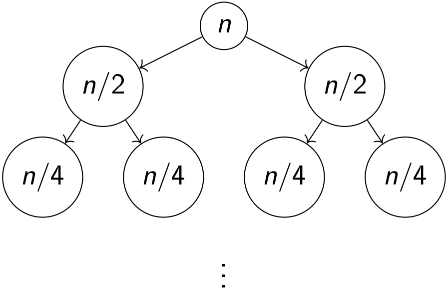
$$T(n) = O(n^{\log_2 2}) = O(n)$$

Aunque es D&C verdadero, ¡no mejora la complejidad! Sigue siendo $O(n)$.

V1: Árbol Degenerado



V2: Árbol Balanceado



Profundidad: $O(n)$

Profundidad: $O(\log n)$

Conclusiones

No todo algoritmo recursivo que divide el problema es *divide and conquer*.

NO es D&C si:

- División desbalanceada extrema
- Subproblema de tamaño $n - k$
- Reducción no significativa
- Recursión lineal disfrazada

SÍ es D&C si:

- División balanceada
- Subproblemas de tamaño n/c
- Reducción por factor constante
- Verdadera descomposición

Divide and Conquer requiere una división **significativa y balanceada** del problema original.

Ejercicio 6 *(MaximoMontaña)*

Un arreglo de enteros se denomina *montaña* si está compuesto por una secuencia estrictamente creciente seguida de una estrictamente decreciente.

Dado un arreglo *montaña* de longitud n, dar un algoritmo que encuentre el máximo del arreglo en complejidad $O(\log n)$.

Ejemplo: Para el arreglo $[-1, 3, 8, 22, 30, 22, 8, 4, 2, 1]$, el máximo está en la posición 4 y vale 30.

Objetivo:

- Diseñar un algoritmo divide and conquer
- Demostrar que tiene complejidad $O(\log n)$
- Aplicar el Teorema Maestro

Ejercicio 6 - Idea del Algoritmo

Observación clave: En un arreglo montaña, el máximo es el punto donde cambia de creciente a decreciente.

Estrategia Divide and Conquer:

1. Examinar el elemento del medio
2. Compararlo con sus vecinos
3. Decidir en qué mitad buscar

Casos posibles para `arr[medio]`:

- Es mayor que ambos vecinos → ¡Es el máximo!
- Es menor que el vecino izquierdo → Máximo está a la izquierda
- Es menor que el vecino derecho → Máximo está a la derecha

Ejercicio 6 - Función maximo_montana

```
def maximo_montana(arr, izq=0, der=None):
  if der is None:
    der = len(arr) - 1
  if izq == der:
    return izq  # Un solo elemento
  medio = (izq + der) // 2
  # Comparar con vecinos
  if medio > 0 and arr[medio] < arr[medio - 1]:
    # El máximo está en la mitad izquierda
    return maximo_montana(arr, izq, medio - 1)
  elif medio < len(arr)-1 and arr[medio] < arr[medio + 1]:
    # El máximo está en la mitad derecha
    return maximo_montana(arr, medio + 1, der)
  else:
    # arr[medio] es el máximo
    return medio
```

Ejercicio 6 - Análisis Divide and Conquer

Pregunta: Identificar las partes del algoritmo divide and conquer.

Ejercicio 6 - Análisis Divide and Conquer

Pregunta: Identificar las partes del algoritmo divide and conquer.

Respuesta:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Comparaciones con vecinos para decidir qué mitad explorar

Pregunta: Identificar las partes del algoritmo divide and conquer.

Respuesta:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Comparaciones con vecinos para decidir qué mitad explorar
- **Conquer:**
  - `maximo_montana(arr, izq, medio - 1)`
  - `maximo_montana(arr, medio + 1, der)`
  - Solo se ejecuta **una** de las dos llamadas

Ejercicio 6 - Análisis Divide and Conquer

Pregunta: Identificar las partes del algoritmo divide and conquer.

Respuesta:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Comparaciones con vecinos para decidir qué mitad explorar
- **Conquer:**
  - `maximo_montana(arr, izq, medio - 1)`
  - `maximo_montana(arr, medio + 1, der)`
  - Solo se ejecuta **una** de las dos llamadas
- **Combine:**
  - No hay combinación - se retorna directamente el resultado

Ejercicio 6 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Ejercicio 6 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- Similar a la búsqueda binaria

Ejercicio 6 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- Similar a la búsqueda binaria
- En cada paso, descartamos una mitad del arreglo

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- Similar a la búsqueda binaria
- En cada paso, descartamos una mitad del arreglo
- La decisión depende de las comparaciones con vecinos:
  - Si `arr[medio] < arr[medio-1]`: explorar izquierda
  - Si `arr[medio] < arr[medio+1]`: explorar derecha
  - Si no: encontramos el máximo

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **1 subproblema**

- Similar a la búsqueda binaria
- En cada paso, descartamos una mitad del arreglo
- La decisión depende de las comparaciones con vecinos:
  - Si `arr[medio] < arr[medio-1]`: explorar izquierda
  - Si `arr[medio] < arr[medio+1]`: explorar derecha
  - Si no: encontramos el máximo

¿De qué tamaño es el subproblema? $n/2$

Ejercicio 6 - Función de Recurrencia

Escribir la función $T(n)$ de manera recursiva:

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)
- $T(n/2)$: Una llamada recursiva sobre la mitad del arreglo

Ejercicio 6 - Función de Recurrencia

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)
- $T(n/2)$: Una llamada recursiva sobre la mitad del arreglo
- $\Theta(1)$: Costo de las comparaciones con vecinos

Ejercicio 6 - Función de Recurrencia

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)
- $T(n/2)$: Una llamada recursiva sobre la mitad del arreglo
- $\Theta(1)$: Costo de las comparaciones con vecinos

Observación: Esta recurrencia es idéntica a la búsqueda binaria

Ejercicio 6 - Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Ejercicio 6 - Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 1$ (un subproblema)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(1)$ (costo constante de comparaciones)

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 1$ (un subproblema)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(1)$ (costo constante de comparaciones)

Calculamos: $\log_c a = \log_2 1 = 0$

Ejercicio 6 - Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 1$ (un subproblema)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(1)$ (costo constante de comparaciones)

Calculamos: $\log_c a = \log_2 1 = 0$

Tenemos $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_c a})$

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 1$ (un subproblema)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(1)$ (costo constante de comparaciones)

Calculamos: $\log_c a = \log_2 1 = 0$

Tenemos $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 1$ (un subproblema)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(1)$ (costo constante de comparaciones)

Calculamos: $\log_c a = \log_2 1 = 0$

Tenemos $f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Por lo tanto: $\boxed{T(n) = \Theta(n^0 \log n) = \Theta(\log n)}$

¿Por qué funciona este algoritmo?

Ejercicio 6 - Correctitud del Algoritmo

¿Por qué funciona este algoritmo?

Propiedades del arreglo montaña:

- Existe un único máximo global

Ejercicio 6 - Correctitud del Algoritmo

¿Por qué funciona este algoritmo?

Propiedades del arreglo montaña:

- Existe un único máximo global
- A la izquierda del máximo: secuencia creciente
- A la derecha del máximo: secuencia decreciente

¿Por qué funciona este algoritmo?

Propiedades del arreglo montaña:

- Existe un único máximo global
- A la izquierda del máximo: secuencia creciente
- A la derecha del máximo: secuencia decreciente

Invariante: El máximo siempre está en el intervalo $[izq, der]$ actual

¿Por qué funciona este algoritmo?

Propiedades del arreglo montaña:

- Existe un único máximo global
- A la izquierda del máximo: secuencia creciente
- A la derecha del máximo: secuencia decreciente

Invariante: El máximo siempre está en el intervalo $[izq, der]$ actual

Demostración por casos:

- Si `arr[medio] < arr[medio-1]`: estamos en la parte decreciente o antes del pico $\rightarrow$ máximo a la izquierda

Ejercicio 6 - Correctitud del Algoritmo

¿Por qué funciona este algoritmo?

Propiedades del arreglo montaña:

- Existe un único máximo global
- A la izquierda del máximo: secuencia creciente
- A la derecha del máximo: secuencia decreciente

Invariante: El máximo siempre está en el intervalo $[izq, der]$ actual

Demostración por casos:

- Si `arr[medio] < arr[medio-1]`: estamos en la parte decreciente o antes del pico $\rightarrow$ máximo a la izquierda
- Si `arr[medio] < arr[medio+1]`: estamos en la parte creciente $\rightarrow$ máximo a la derecha

¿Por qué funciona este algoritmo?

Propiedades del arreglo montaña:

- Existe un único máximo global
- A la izquierda del máximo: secuencia creciente
- A la derecha del máximo: secuencia decreciente

Invariante: El máximo siempre está en el intervalo $[izq, der]$ actual

Demostración por casos:

- Si `arr[medio] < arr[medio-1]`: estamos en la parte decreciente o antes del pico $\rightarrow$ máximo a la izquierda
- Si `arr[medio] < arr[medio+1]`: estamos en la parte creciente $\rightarrow$ máximo a la derecha
- Si ninguna: `arr[medio]` es el máximo

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 9$, $medio = 4$
   - $arr[4] = 30$, $arr[3] = 22$, $arr[5] = 22$
   - $30 > 22$ y $30 > 22 \rightarrow$ ¡Encontrado!

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 9$, $medio = 4$
   - $arr[4] = 30$, $arr[3] = 22$, $arr[5] = 22$
   - $30 > 22$ y $30 > 22 \rightarrow$ ¡Encontrado!

Otro ejemplo: $[1, 5, \boxed{9}, 7, 3, 2, 1]$

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 9$, $medio = 4$
   - $arr[4] = 30$, $arr[3] = 22$, $arr[5] = 22$
   - $30 > 22$ y $30 > 22 \rightarrow$ ¡Encontrado!

Otro ejemplo: $[1, 5, \boxed{9}, 7, 3, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 6$, $medio = 3$
   - $arr[3] = 7$, $arr[2] = 9$, $arr[4] = 3$
   - $7 < 9$ y $7 > 3 \rightarrow$ Máximo a la izquierda
   - Buscar en $[0, 2]$

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 9$, $medio = 4$
   - $arr[4] = 30$, $arr[3] = 22$, $arr[5] = 22$
   - $30 > 22$ y $30 > 22 \rightarrow$ ¡Encontrado!

Otro ejemplo: $[1, 5, \boxed{9}, 7, 3, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 6$, $medio = 3$
   - $arr[3] = 7$, $arr[2] = 9$, $arr[4] = 3$
   - $7 < 9$ y $7 > 3 \rightarrow$ Máximo a la izquierda
   - Buscar en $[0, 2]$
2. **Iteración 2:** $izq = 0$, $der = 2$, $medio = 1$
   - $arr[1] = 5$, $arr[0] = 1$, $arr[2] = 9$
   - $5 > 1$ y $5 < 9 \rightarrow$ Máximo a la derecha
   - Buscar en $[2, 2]$

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 9$, $medio = 4$
   - $arr[4] = 30$, $arr[3] = 22$, $arr[5] = 22$
   - $30 > 22$ y $30 > 22 \rightarrow$ ¡Encontrado!

Otro ejemplo: $[1, 5, \boxed{9}, 7, 3, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 6$, $medio = 3$
   - $arr[3] = 7$, $arr[2] = 9$, $arr[4] = 3$
   - $7 < 9$ y $7 > 3 \rightarrow$ Máximo a la izquierda
   - Buscar en $[0, 2]$
2. **Iteración 2:** $izq = 0$, $der = 2$, $medio = 1$
   - $arr[1] = 5$, $arr[0] = 1$, $arr[2] = 9$
   - $5 > 1$ y $5 < 9 \rightarrow$ Máximo a la derecha
   - Buscar en $[2, 2]$
3. **Iteración 3:** $izq = 2$, $der = 2$
   - ¡Máximo encontrado en posición 2!

Ejercicio 6 - Ejemplo de Ejecución

Arreglo: $[-1, 3, 8, 22, \boxed{30}, 22, 8, 4, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 9$, $medio = 4$
    - $arr[4] = 30$, $arr[3] = 22$, $arr[5] = 22$
    - $30 > 22$ y $30 > 22 \rightarrow$ ¡Encontrado!

Otro ejemplo: $[1, 5, \boxed{9}, 7, 3, 2, 1]$

1. **Iteración 1:** $izq = 0$, $der = 6$, $medio = 3$
    - $arr[3] = 7$, $arr[2] = 9$, $arr[4] = 3$
    - $7 < 9$ y $7 > 3 \rightarrow$ Máximo a la izquierda
    - Buscar en $[0, 2]$
2. **Iteración 2:** $izq = 0$, $der = 2$, $medio = 1$
    - $arr[1] = 5$, $arr[0] = 1$, $arr[2] = 9$
    - $5 > 1$ y $5 < 9 \rightarrow$ Máximo a la derecha
    - Buscar en $[2, 2]$
3. **Iteración 3:** $izq = 2$, $der = 2$
    - ¡Máximo encontrado en posición 2!

Complejidad: En el peor caso, $\log_2 n$ comparaciones

Diferentes estrategias para encontrar el máximo

Enfoque	**Complejidad**	**Descripción**
Fuerza bruta	$O(n)$	Recorrer todo el arreglo
Divide and Conquer	$O(\log n)$	Descartar mitades

Ejercicio 6 - Comparación de Enfoques

Diferentes estrategias para encontrar el máximo

Enfoque	**Complejidad**	**Descripción**
Fuerza bruta	$O(n)$	Recorrer todo el arreglo
Divide and Conquer	$O(\log n)$	Descartar mitades

Ventajas del enfoque Divide and Conquer:

- Aprovecha la estructura especial del arreglo montaña

Ejercicio 6 - Comparación de Enfoques

Diferentes estrategias para encontrar el máximo

Enfoque	**Complejidad**	**Descripción**
Fuerza bruta	$O(n)$	Recorrer todo el arreglo
Divide and Conquer	$O(\log n)$	Descartar mitades

Ventajas del enfoque Divide and Conquer:

- Aprovecha la estructura especial del arreglo montaña
- Complejidad logarítmica vs lineal

Ejercicio 6 - Comparación de Enfoques

Diferentes estrategias para encontrar el máximo

Enfoque	**Complejidad**	**Descripción**
Fuerza bruta	$O(n)$	Recorrer todo el arreglo
Divide and Conquer	$O(\log n)$	Descartar mitades

Ventajas del enfoque Divide and Conquer:

- Aprovecha la estructura especial del arreglo montaña
- Complejidad logarítmica vs lineal
- Eficiente para arreglos grandes

Ejercicio 6 - Comparación de Enfoques

Diferentes estrategias para encontrar el máximo

Enfoque	**Complejidad**	**Descripción**
Fuerza bruta	$O(n)$	Recorrer todo el arreglo
Divide and Conquer	$O(\log n)$	Descartar mitades

Ventajas del enfoque Divide and Conquer:

- Aprovecha la estructura especial del arreglo montaña
- Complejidad logarítmica vs lineal
- Eficiente para arreglos grandes

Aplicación práctica:

- Búsqueda de picos en señales
- Optimización unimodal
- Análisis de series temporales

Ejercicio 8 *(MaximaSubsecuencia)*

Dada una secuencia de n enteros, se desea encontrar el máximo valor que se puede obtener sumando elementos contiguos.

Diseñar un algoritmo basado en la técnica de dividir y conquistar que resuelva el problema en $O(n \log n)$.

Ejemplo: Para la secuencia $[3, -1, 4, 8, -2, 2, -7, 5]$, este valor es 14, que se obtiene de la subsecuencia $[3, -1, 4, 8]$.

Nota: Este problema también se conoce como el problema del subarreglo de suma máxima (Maximum Subarray Problem).

Ejercicio 8 - Estrategia Divide and Conquer

Idea principal: Dividir el arreglo por la mitad y considerar tres casos:

1. **Caso Izquierda:** La subsecuencia máxima está completamente en la mitad izquierda

Ejercicio 8 - Estrategia Divide and Conquer

Idea principal: Dividir el arreglo por la mitad y considerar tres casos:

1. **Caso Izquierda:** La subsecuencia máxima está completamente en la mitad izquierda
2. **Caso Derecha:** La subsecuencia máxima está completamente en la mitad derecha

Ejercicio 8 - Estrategia Divide and Conquer

Idea principal: Dividir el arreglo por la mitad y considerar tres casos:

1. **Caso Izquierda:** La subsecuencia máxima está completamente en la mitad izquierda
2. **Caso Derecha:** La subsecuencia máxima está completamente en la mitad derecha
3. **Caso Cruzado:** La subsecuencia máxima cruza el punto medio

Ejercicio 8 - Estrategia Divide and Conquer

Idea principal: Dividir el arreglo por la mitad y considerar tres casos:

1. **Caso Izquierda:** La subsecuencia máxima está completamente en la mitad izquierda
2. **Caso Derecha:** La subsecuencia máxima está completamente en la mitad derecha
3. **Caso Cruzado:** La subsecuencia máxima cruza el punto medio

Solución: El máximo de los tres casos

$$\text{max\_suma} = \max(\text{caso\_izq}, \text{caso\_der}, \text{caso\_cruzado})$$

Ejercicio 8 - Estrategia Divide and Conquer

Idea principal: Dividir el arreglo por la mitad y considerar tres casos:

1. **Caso Izquierda:** La subsecuencia máxima está completamente en la mitad izquierda
2. **Caso Derecha:** La subsecuencia máxima está completamente en la mitad derecha
3. **Caso Cruzado:** La subsecuencia máxima cruza el punto medio

Solución: El máximo de los tres casos

$$\text{max\_suma} = \max(\text{caso\_izq}, \text{caso\_der}, \text{caso\_cruzado})$$

Observación clave: El caso cruzado requiere un tratamiento especial

Ejercicio 8 - Caso Cruzado

¿Cómo calcular la suma máxima que cruza el medio?

Ejercicio 8 - Caso Cruzado

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Ejercicio 8 - Caso Cruzado

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Algoritmo:

1. Desde el medio hacia la izquierda: encontrar la suma máxima

Ejercicio 8 - Caso Cruzado

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Algoritmo:

1. Desde el medio hacia la izquierda: encontrar la suma máxima
2. Desde el medio+1 hacia la derecha: encontrar la suma máxima

Ejercicio 8 - Caso Cruzado

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Algoritmo:

1. Desde el medio hacia la izquierda: encontrar la suma máxima
2. Desde el medio+1 hacia la derecha: encontrar la suma máxima
3. Sumar ambas partes

Ejercicio 8 - Caso Cruzado

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Algoritmo:

1. Desde el medio hacia la izquierda: encontrar la suma máxima
2. Desde el medio+1 hacia la derecha: encontrar la suma máxima
3. Sumar ambas partes

Ejemplo: $[3, -1, 4, 8| - 2, 2, -7, 5]$

- Máximo izquierda (desde medio): $8 + 4 + (-1) + 3 = 14$

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Algoritmo:

1. Desde el medio hacia la izquierda: encontrar la suma máxima
2. Desde el medio+1 hacia la derecha: encontrar la suma máxima
3. Sumar ambas partes

Ejemplo: $[3, -1, 4, 8| - 2, 2, -7, 5]$

- Máximo izquierda (desde medio): $8 + 4 + (-1) + 3 = 14$
- Máximo derecha (desde medio+1): $(-2) + 2 = 0$

¿Cómo calcular la suma máxima que cruza el medio?

La subsecuencia debe incluir:

- Al menos el último elemento de la mitad izquierda
- Al menos el primer elemento de la mitad derecha

Algoritmo:

1. Desde el medio hacia la izquierda: encontrar la suma máxima
2. Desde el medio+1 hacia la derecha: encontrar la suma máxima
3. Sumar ambas partes

Ejemplo: $[3, -1, 4, 8| - 2, 2, -7, 5]$

- Máximo izquierda (desde medio): $8 + 4 + (-1) + 3 = 14$
- Máximo derecha (desde medio+1): $(-2) + 2 = 0$
- Suma cruzada: $14 + 0 = 14$

Ejercicio 8 - Función Principal

```python
def max_subsecuencia(arr, izq=0, der=None):
  if der is None:
    der = len(arr) - 1

  if izq == der:
    return arr[izq]

  medio = (izq + der) // 2

  max_izq = max_subsecuencia(arr, izq, medio)
  max_der = max_subsecuencia(arr, medio + 1, der)

  max_cruzado = suma_maxima_cruzada(arr, izq, medio, der)

  return max(max_izq, max_der, max_cruzado)
```

Ejercicio 8 - Función Suma Cruzada

```
def suma_maxima_cruzada(arr, izq, medio, der):
  suma_izq = float('-inf')
  suma = 0
  for i in range(medio, izq - 1, -1):
    suma += arr[i]
    suma_izq = max(suma_izq, suma)

  suma_der = float('-inf')
  suma = 0
  for i in range(medio + 1, der + 1):
   suma += arr[i]
   suma_der = max(suma_der, suma)

  return suma_izq + suma_der
```

Complejidad de esta función: $O(n)$ donde $n = der - izq + 1$

Ejercicio 8 - Análisis Divide and Conquer

Identificar las partes del algoritmo:

Ejercicio 8 - Análisis Divide and Conquer

Identificar las partes del algoritmo:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Separar el arreglo en dos mitades

Ejercicio 8 - Análisis Divide and Conquer

Identificar las partes del algoritmo:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Separar el arreglo en dos mitades
- **Conquer:**
  - `max_izq = max_subsecuencia(arr, izq, medio)`
  - `max_der = max_subsecuencia(arr, medio + 1, der)`

Identificar las partes del algoritmo:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Separar el arreglo en dos mitades
- **Conquer:**
  - `max_izq = max_subsecuencia(arr, izq, medio)`
  - `max_der = max_subsecuencia(arr, medio + 1, der)`
- **Combine:**
  - Calcular la suma máxima cruzada
  - `return max(max_izq, max_der, max_cruzado)`

Identificar las partes del algoritmo:

- **Divide:**
  - `medio = (izq + der) // 2`
  - Separar el arreglo en dos mitades
- **Conquer:**
  - `max_izq = max_subsecuencia(arr, izq, medio)`
  - `max_der = max_subsecuencia(arr, medio + 1, der)`
- **Combine:**
  - Calcular la suma máxima cruzada
  - `return max(max_izq, max_der, max_cruzado)`

Observación: A diferencia de búsqueda binaria, aquí sí hay una fase de combinación significativa con costo $O(n)$

Ejercicio 8 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Ejercicio 8 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **2 subproblemas**

- Mitad izquierda: `arr[izq...medio]`
- Mitad derecha: `arr[medio+1...der]`

Ejercicio 8 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **2 subproblemas**

- Mitad izquierda: `arr[izq...medio]`
- Mitad derecha: `arr[medio+1...der]`

¿De qué tamaño son estos subproblemas?

Ejercicio 8 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **2 subproblemas**

- Mitad izquierda: `arr[izq...medio]`
- Mitad derecha: `arr[medio+1...der]`

¿De qué tamaño son estos subproblemas?

Respuesta: Cada subproblema tiene tamaño n/2

Ejercicio 8 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **2 subproblemas**

- Mitad izquierda: `arr[izq...medio]`
- Mitad derecha: `arr[medio+1...der]`

¿De qué tamaño son estos subproblemas?

Respuesta: Cada subproblema tiene tamaño n/2

¿Cuál es el costo de combinar?

Ejercicio 8 - Análisis de Subproblemas

¿En cuántos subproblemas se divide?

Respuesta: Se divide en **2 subproblemas**

- Mitad izquierda: `arr[izq...medio]`
- Mitad derecha: `arr[medio+1...der]`

¿De qué tamaño son estos subproblemas?

Respuesta: Cada subproblema tiene tamaño n/2

¿Cuál es el costo de combinar?

Respuesta: O(n) debido al cálculo de la suma cruzada

- Recorrer desde el medio hacia la izquierda: $O(n/2)$
- Recorrer desde medio+1 hacia la derecha: $O(n/2)$
- Total: $O(n)$

Ejercicio 8 - Función de Recurrencia

Escribir la función $T(n)$ de manera recursiva:

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)

Ejercicio 8 - Función de Recurrencia

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)
- $2T(n/2)$: Dos llamadas recursivas sobre mitades

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)
- $2T(n/2)$: Dos llamadas recursivas sobre mitades
- $\Theta(n)$: Costo de calcular la suma cruzada y comparar

Escribir la función $T(n)$ de manera recursiva:

Respuesta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $\Theta(1)$: Caso base (un elemento)
- $2T(n/2)$: Dos llamadas recursivas sobre mitades
- $\Theta(n)$: Costo de calcular la suma cruzada y comparar

Observación: Esta recurrencia es idéntica a MergeSort

Ejercicio 8 - Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Ejercicio 8 - Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 2$ (dos subproblemas)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(n)$ (costo de combinar)

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 2$ (dos subproblemas)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Ejercicio 8 - Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 2$ (dos subproblemas)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Tenemos $f(n) = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_c a})$

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 2$ (dos subproblemas)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Tenemos $f(n) = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Determinar la complejidad usando el Teorema Maestro:

Parámetros:

- $a = 2$ (dos subproblemas)
- $c = 2$ (división por la mitad)
- $f(n) = \Theta(n)$ (costo de combinar)

Calculamos: $\log_c a = \log_2 2 = 1$

Tenemos $f(n) = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_c a})$
$\Rightarrow$ Estamos en el **Caso 2** del Teorema Maestro

Por lo tanto: $\boxed{T(n) = \Theta(n \log n)}$

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12
1	$[3, -1, 4, 8]$	3	12	14

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12
1	$[3, -1, 4, 8]$	3	12	14
3	$[-2]$	-	-	-
3	$[2]$	-	-	-
3	$[-7]$	-	-	-
3	$[5]$	-	-	-

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12
1	$[3, -1, 4, 8]$	3	12	14
3	$[-2]$	-	-	-
3	$[2]$	-	-	-
3	$[-7]$	-	-	-
3	$[5]$	-	-	-
2	$[-2, 2]$	-2	2	0
2	$[-7, 5]$	-7	5	-2

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12
1	$[3, -1, 4, 8]$	3	12	14
3	$[-2]$	-	-	-
3	$[2]$	-	-	-
3	$[-7]$	-	-	-
3	$[5]$	-	-	-
2	$[-2, 2]$	-2	2	0
2	$[-7, 5]$	-7	5	-2
1	$[-2, 2, -7, 5]$	2	5	0

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12
1	$[3, -1, 4, 8]$	3	12	14
3	$[-2]$	-	-	-
3	$[2]$	-	-	-
3	$[-7]$	-	-	-
3	$[5]$	-	-	-
2	$[-2, 2]$	-2	2	0
2	$[-7, 5]$	-7	5	-2
1	$[-2, 2, -7, 5]$	2	5	0
0	Completo	14	5	14

Ejercicio 8 - Ejemplo de Ejecución

Arreglo: $[3, -1, 4, 8, -2, 2, -7, 5]$

Nivel	Subarreglo	Max Izq	Max Der	Max Cruz
3	$[3]$	-	-	-
3	$[-1]$	-	-	-
3	$[4]$	-	-	-
3	$[8]$	-	-	-
2	$[3, -1]$	3	-1	2
2	$[4, 8]$	4	8	12
1	$[3, -1, 4, 8]$	3	12	14
3	$[-2]$	-	-	-
3	$[2]$	-	-	-
3	$[-7]$	-	-	-
3	$[5]$	-	-	-
2	$[-2, 2]$	-2	2	0
2	$[-7, 5]$	-7	5	-2
1	$[-2, 2, -7, 5]$	2	5	0
0	Completo	14	5	14

Resultado final: $\text{máx}(14, 5, 14) = 14$

Diferentes enfoques para el problema

Algoritmo	**Complejidad**	**Descripción**
Fuerza bruta	$O(n^3)$	Probar todos los pares (i,j)
Fuerza bruta mejorada	$O(n^2)$	Sumas acumuladas
Divide and Conquer	$O(n \log n)$	Este ejercicio
Algoritmo de Kadane	$O(n)$	Programación dinámica

Diferentes enfoques para el problema

Algoritmo	**Complejidad**	**Descripción**
Fuerza bruta	$O(n^3)$	Probar todos los pares (i, j)
Fuerza bruta mejorada	$O(n^2)$	Sumas acumuladas
Divide and Conquer	$O(n \log n)$	Este ejercicio
Algoritmo de Kadane	$O(n)$	Programación dinámica

Ventajas del enfoque D&C:

- Más eficiente que fuerza bruta

Diferentes enfoques para el problema

Algoritmo	Complejidad	Descripción
Fuerza bruta	$O(n^3)$	Probar todos los pares (i, j)
Fuerza bruta mejorada	$O(n^2)$	Sumas acumuladas
Divide and Conquer	$O(n \log n)$	Este ejercicio
Algoritmo de Kadane	$O(n)$	Programación dinámica

Ventajas del enfoque D&C:

- Más eficiente que fuerza bruta
- Ilustra bien la técnica divide and conquer

Diferentes enfoques para el problema

Algoritmo	**Complejidad**	**Descripción**
Fuerza bruta	$O(n^3)$	Probar todos los pares (i, j)
Fuerza bruta mejorada	$O(n^2)$	Sumas acumuladas
Divide and Conquer	$O(n \log n)$	Este ejercicio
Algoritmo de Kadane	$O(n)$	Programación dinámica

Ventajas del enfoque D&C:

- Más eficiente que fuerza bruta
- Ilustra bien la técnica divide and conquer
- Paralelizable (las llamadas recursivas son independientes)

Diferentes enfoques para el problema

Algoritmo	**Complejidad**	**Descripción**
Fuerza bruta	$O(n^3)$	Probar todos los pares (i, j)
Fuerza bruta mejorada	$O(n^2)$	Sumas acumuladas
Divide and Conquer	$O(n \log n)$	Este ejercicio
Algoritmo de Kadane	$O(n)$	Programación dinámica

Ventajas del enfoque D&C:

- Más eficiente que fuerza bruta
- Ilustra bien la técnica divide and conquer
- Paralelizable (las llamadas recursivas son independientes)

Desventaja: Existe una solución $O(n)$ usando programación dinámica

Ejercicio 14 *(Diferencia Mínima)*

Se tienen dos arreglos de n naturales A y B:

- A está ordenado de manera **creciente**
- B está ordenado de manera **decreciente**
- Ningún valor aparece más de una vez en el mismo arreglo

Para cada posición i consideramos la diferencia absoluta entre los valores de ambos arreglos $|A[i] - B[i]|$. Se desea buscar el mínimo valor posible de dicha cuenta.

Ejemplo:

- $A = [1, 2, 3, 4]$ y $B = [6, 4, 2, 1]$
- Diferencias: $|1 - 6| = 5$, $|2 - 4| = 2$, $|3 - 2| = 1$, $|4 - 1| = 3$
- Resultado: 1

Objetivo: Implementar `minDif` con complejidad $O(\log n)$

Ejercicio 14 - Análisis del Problema

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande

Ejercicio 14 - Análisis del Problema

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande
- Al avanzar: $A[i]$ crece, $B[i]$ decrece

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande
- Al avanzar: $A[i]$ crece, $B[i]$ decrece
- En algún punto: las diferencias se minimizan

Ejercicio 14 - Análisis del Problema

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande
- Al avanzar: $A[i]$ crece, $B[i]$ decrece
- En algún punto: las diferencias se minimizan
- Al final: $A[n-1]$ es máximo, $B[n-1]$ es mínimo

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande
- Al avanzar: $A[i]$ crece, $B[i]$ decrece
- En algún punto: las diferencias se minimizan
- Al final: $A[n-1]$ es máximo, $B[n-1]$ es mínimo

Propiedad importante:

La función $f(i) = |A[i] - B[i]|$ tiene comportamiento **unimodal**:

- Puede decrecer y luego crecer

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande
- Al avanzar: $A[i]$ crece, $B[i]$ decrece
- En algún punto: las diferencias se minimizan
- Al final: $A[n-1]$ es máximo, $B[n-1]$ es mínimo

Propiedad importante:
La función $f(i) = |A[i] - B[i]|$ tiene comportamiento **unimodal**:

- Puede decrecer y luego crecer
- O solo decrecer, o solo crecer

Ejercicio 14 - Análisis del Problema

Observación clave sobre el comportamiento de las diferencias:

Como A es creciente y B es decreciente:

- Al inicio: $A[0]$ es mínimo, $B[0]$ es máximo $\rightarrow$ diferencia grande
- Al avanzar: $A[i]$ crece, $B[i]$ decrece
- En algún punto: las diferencias se minimizan
- Al final: $A[n-1]$ es máximo, $B[n-1]$ es mínimo

Propiedad importante:

La función $f(i) = |A[i] - B[i]|$ tiene comportamiento **unimodal**:

- Puede decrecer y luego crecer
- O solo decrecer, o solo crecer
- Tiene un mínimo global

Ejercicio 14 - Estrategia Divide and Conquer - Búsqueda ternaria (Parte 1)

1. **Inicialización**

$$\text{left} = 0 \tag{1}$$

$$\text{right} = n - 1 \tag{2}$$

Ejercicio 14 - Estrategia Divide and Conquer - Búsqueda ternaria (Parte 1)

1. **Inicialización**

$$\text{left} = 0 \tag{1}$$
$$\text{right} = n - 1 \tag{2}$$

2. **División en tercios**
   Mientras right − left > 2:

$$\text{mid}_1 = \text{left} + \left\lfloor \frac{\text{right} - \text{left}}{3} \right\rfloor \tag{3}$$
$$\text{mid}_2 = \text{right} - \left\lfloor \frac{\text{right} - \text{left}}{3} \right\rfloor \tag{4}$$

Ejercicio 14 - Estrategia Divide and Conquer - Búsqueda ternaria (Parte 1)

1. **Inicialización**

$$\text{left} = 0 \tag{1}$$
$$\text{right} = n - 1 \tag{2}$$

2. **División en tercios**
   Mientras right − left > 2:

$$\text{mid}_1 = \text{left} + \left\lfloor \frac{\text{right} - \text{left}}{3} \right\rfloor \tag{3}$$
$$\text{mid}_2 = \text{right} - \left\lfloor \frac{\text{right} - \text{left}}{3} \right\rfloor \tag{4}$$

3. **Evaluación**

$$\text{dif}_1 = |A[\text{mid}_1] - B[\text{mid}_1]| \tag{5}$$
$$\text{dif}_2 = |A[\text{mid}_2] - B[\text{mid}_2]| \tag{6}$$

Ejercicio 14 - Estrategia Divide and Conquer - Búsqueda ternaria (Parte 2)

4. **Decisión**
   - **Si** $\text{dif}_1 > \text{dif}_2$:
     - El mínimo está más cerca de mid_2
     - Actualizamos: $\text{left} = \text{mid}_1$
     - Descartamos el primer tercio $[\text{left}, \text{mid}_1)$
   - **Si** $\text{dif}_1 \leq \text{dif}_2$:
     - El mínimo está más cerca de mid_1
     - Actualizamos: $\text{right} = \text{mid}_2$
     - Descartamos el último tercio $(\text{mid}_2, \text{right}]$

Ejercicio 14 - Estrategia Divide and Conquer - Búsqueda ternaria (Parte 2)

4. **Decisión**
   - **Si** $\text{dif}_1 > \text{dif}_2$:
     - El mínimo está más cerca de mid_2
     - Actualizamos: $\text{left} = \text{mid}_1$
     - Descartamos el primer tercio $[\text{left}, \text{mid}_1)$
   - **Si** $\text{dif}_1 \leq \text{dif}_2$:
     - El mínimo está más cerca de mid_1
     - Actualizamos: $\text{right} = \text{mid}_2$
     - Descartamos el último tercio $(\text{mid}_2, \text{right}]$
5. **Búsqueda final**
   Cuando quedan 3 o menos elementos:

$$\text{resultado} = \min_{i \in [\text{left}, \text{right}]} |A[i] - B[i]| \tag{7}$$

Ejercicio 14 - Función `minDif`

```
def minDif(A, B):
  n = len(A)
  left, right = 0, n - 1
  while right - left > 2:
    mid1 = left + (right - left) // 3
    mid2 = right - (right - left) // 3
    dif1 = abs(A[mid1] - B[mid1])
    dif2 = abs(A[mid2] - B[mid2])
    if dif1 > dif2:
      left = mid1
    else:
      right = mid2
  # Verificar los últimos elementos
  min_dif = float('inf')
  for i in range(left, right + 1):
    min_dif = min(min_dif, abs(A[i] - B[i]))
  return min_dif
```

Ejercicio 14 - Análisis de Complejidad

Pregunta: ¿Cuál es la complejidad del algoritmo?

Ejercicio 14 - Análisis de Complejidad

Pregunta: ¿Cuál es la complejidad del algoritmo?

Análisis para búsqueda ternaria:

- En cada iteración, reducimos el espacio de búsqueda a $\frac{2n}{3}$

Ejercicio 14 - Análisis de Complejidad

Pregunta: ¿Cuál es la complejidad del algoritmo?

Análisis para búsqueda ternaria:

- En cada iteración, reducimos el espacio de búsqueda a $\frac{2n}{3}$
- Número de iteraciones: $\log_{3/2} n = O(\log n)$

Ejercicio 14 - Análisis de Complejidad

Pregunta: ¿Cuál es la complejidad del algoritmo?

Análisis para búsqueda ternaria:

- En cada iteración, reducimos el espacio de búsqueda a $\frac{2n}{3}$
- Número de iteraciones: $\log_{3/2} n = O(\log n)$

Función de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ T\left(\frac{2n}{3}\right) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Ejercicio 14 - Análisis de Complejidad

Pregunta: ¿Cuál es la complejidad del algoritmo?

Análisis para búsqueda ternaria:

- En cada iteración, reducimos el espacio de búsqueda a $\frac{2n}{3}$
- Número de iteraciones: $\log_{3/2} n = O(\log n)$

Función de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ T\left(\frac{2n}{3}\right) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Aplicando el Teorema Maestro:

- $a = 1,\ c = 3/2,\ f(n) = \Theta(1)$

Ejercicio 14 - Análisis de Complejidad

Pregunta: ¿Cuál es la complejidad del algoritmo?

Análisis para búsqueda ternaria:

- En cada iteración, reducimos el espacio de búsqueda a $\frac{2n}{3}$
- Número de iteraciones: $\log_{3/2} n = O(\log n)$

Función de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ T\left(\frac{2n}{3}\right) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Aplicando el Teorema Maestro:

- $a = 1,\ c = 3/2,\ f(n) = \Theta(1)$
- $\log_{3/2} 1 = 0$

Pregunta: ¿Cuál es la complejidad del algoritmo?

Análisis para búsqueda ternaria:

- En cada iteración, reducimos el espacio de búsqueda a $\frac{2n}{3}$
- Número de iteraciones: $\log_{3/2} n = O(\log n)$

Función de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ T\left(\frac{2n}{3}\right) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Aplicando el Teorema Maestro:

- $a = 1,\ c = 3/2,\ f(n) = \Theta(1)$
- $\log_{3/2} 1 = 0$
- Caso 2: $T(n) = \Theta(\log n)$

Ejercicio 14 - Ejemplo de Ejecución

Arreglos: $A = [1, 2, 3, 4, 5]$, $B = [9, 7, 5, 3, 1]$

Diferencias: $[8, 5, 2, 1, 4]$

Ejercicio 14 - Ejemplo de Ejecución

Arreglos: $A = [1, 2, 3, 4, 5]$, $B = [9, 7, 5, 3, 1]$

Diferencias: $[8, 5, 2, 1, 4]$

Traza del algoritmo (búsqueda ternaria):

1. **Iteración 1:** $izq = 0$, $der = 4$
   - $tercio1 = 1$: $|A[1] - B[1]| = |2 - 7| = 5$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $5 > 1$, buscar en $[1, 4]$

Ejercicio 14 - Ejemplo de Ejecución

Arreglos: $A = [1, 2, 3, 4, 5]$, $B = [9, 7, 5, 3, 1]$

Diferencias: $[8, 5, 2, 1, 4]$

Traza del algoritmo (búsqueda ternaria):

1. **Iteración 1:** $izq = 0$, $der = 4$
   - $tercio1 = 1$: $|A[1] - B[1]| = |2 - 7| = 5$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $5 > 1$, buscar en $[1, 4]$
2. **Iteración 2:** $izq = 1$, $der = 4$
   - $tercio1 = 2$: $|A[2] - B[2]| = |3 - 5| = 2$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $2 > 1$, buscar en $[2, 4]$

Ejercicio 14 - Ejemplo de Ejecución

Arreglos: $A = [1, 2, 3, 4, 5]$, $B = [9, 7, 5, 3, 1]$

Diferencias: $[8, 5, 2, 1, 4]$

Traza del algoritmo (búsqueda ternaria):

1. **Iteración 1:** $izq = 0$, $der = 4$
   - $tercio1 = 1$: $|A[1] - B[1]| = |2 - 7| = 5$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $5 > 1$, buscar en $[1, 4]$
2. **Iteración 2:** $izq = 1$, $der = 4$
   - $tercio1 = 2$: $|A[2] - B[2]| = |3 - 5| = 2$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $2 > 1$, buscar en $[2, 4]$
3. **Iteración 3:** $izq = 2$, $der = 4$
   - Caso base: verificar posiciones 2, 3, 4
   - Mínimo $= 1$ (en posición 3)

Arreglos: $A = [1, 2, 3, 4, 5]$, $B = [9, 7, 5, 3, 1]$

Diferencias: $[8, 5, 2, 1, 4]$

Traza del algoritmo (búsqueda ternaria):

1. **Iteración 1:** $izq = 0$, $der = 4$
   - $tercio1 = 1$: $|A[1] - B[1]| = |2 - 7| = 5$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $5 > 1$, buscar en $[1, 4]$
2. **Iteración 2:** $izq = 1$, $der = 4$
   - $tercio1 = 2$: $|A[2] - B[2]| = |3 - 5| = 2$
   - $tercio2 = 3$: $|A[3] - B[3]| = |4 - 3| = 1$
   - Como $2 > 1$, buscar en $[2, 4]$
3. **Iteración 3:** $izq = 2$, $der = 4$
   - Caso base: verificar posiciones 2, 3, 4
   - Mínimo $= 1$ (en posición 3)

Resultado: Diferencia mínima $= 1$

Ejercicio 14 - Justificación de $O(\log n)$

¿Por qué funciona en $O(\log n)$?

Ejercicio 14 - Justificación de $O(\log n)$

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$
2. **Comportamiento de la diferencia:**
   - La función diferencia tiene a lo más un valle

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$
2. **Comportamiento de la diferencia:**
   - La función diferencia tiene a lo más un valle
   - Podemos descartar porciones del arreglo sin examinarlas

Ejercicio 14 - Justificación de $O(\log n)$

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$
2. **Comportamiento de la diferencia:**
   - La función diferencia tiene a lo más un valle
   - Podemos descartar porciones del arreglo sin examinarlas
3. **Reducción del espacio:**
   - Cada comparación descarta al menos $1/3$ del espacio

Ejercicio 14 - Justificación de $O(\log n)$

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$
2. **Comportamiento de la diferencia:**
   - La función diferencia tiene a lo más un valle
   - Podemos descartar porciones del arreglo sin examinarlas
3. **Reducción del espacio:**
   - Cada comparación descarta al menos $1/3$ del espacio
   - Después de k iteraciones: espacio $\leq n \cdot (2/3)^k$

Ejercicio 14 - Justificación de $O(\log n)$

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$
2. **Comportamiento de la diferencia:**
   - La función diferencia tiene a lo más un valle
   - Podemos descartar porciones del arreglo sin examinarlas
3. **Reducción del espacio:**
   - Cada comparación descarta al menos 1/3 del espacio
   - Después de k iteraciones: espacio $\leq n \cdot (2/3)^k$
   - Cuando $(2/3)^k \cdot n = 1$: $k = \log_{3/2} n = O(\log n)$

Ejercicio 14 - Justificación de $O(\log n)$

¿Por qué funciona en $O(\log n)$?

1. **Propiedad de los arreglos:**
   - A creciente: $A[i] < A[i+1]$
   - B decreciente: $B[i] > B[i+1]$
2. **Comportamiento de la diferencia:**
   - La función diferencia tiene a lo más un valle
   - Podemos descartar porciones del arreglo sin examinarlas
3. **Reducción del espacio:**
   - Cada comparación descarta al menos 1/3 del espacio
   - Después de k iteraciones: espacio $\leq n \cdot (2/3)^k$
   - Cuando $(2/3)^k \cdot n = 1$: $k = \log_{3/2} n = O(\log n)$

Conclusión: La estructura especial del problema permite búsqueda logarítmica

Diferentes enfoques para el problema

Algoritmo	**Complejidad**	**Descripción**
Fuerza bruta	$O(n)$	Calcular todas las diferencias
Búsqueda binaria	$O(\log n)$	Aprovechar la unimodalidad
Búsqueda ternaria	$O(\log n)$	Dividir en tercios

Ejercicio 14 - Comparación de Soluciones

Diferentes enfoques para el problema

Algoritmo	Complejidad	Descripción
Fuerza bruta	$O(n)$	Calcular todas las diferencias
Búsqueda binaria	$O(\log n)$	Aprovechar la unimodalidad
Búsqueda ternaria	$O(\log n)$	Dividir en tercios

Ventajas de la solución D&C:

- Complejidad logarítmica óptima

Diferentes enfoques para el problema

Algoritmo	Complejidad	Descripción
Fuerza bruta	$O(n)$	Calcular todas las diferencias
Búsqueda binaria	$O(\log n)$	Aprovechar la unimodalidad
Búsqueda ternaria	$O(\log n)$	Dividir en tercios

Ventajas de la solución D&C:

- Complejidad logarítmica óptima
- Aprovecha la estructura especial del problema

Diferentes enfoques para el problema

Algoritmo	Complejidad	Descripción
Fuerza bruta	$O(n)$	Calcular todas las diferencias
Búsqueda binaria	$O(\log n)$	Aprovechar la unimodalidad
Búsqueda ternaria	$O(\log n)$	Dividir en tercios

Ventajas de la solución D&C:

- Complejidad logarítmica óptima
- Aprovecha la estructura especial del problema
- No necesita examinar todos los elementos

Diferentes enfoques para el problema

Algoritmo	Complejidad	Descripción
Fuerza bruta	$O(n)$	Calcular todas las diferencias
Búsqueda binaria	$O(\log n)$	Aprovechar la unimodalidad
Búsqueda ternaria	$O(\log n)$	Dividir en tercios

Ventajas de la solución D&C:

- Complejidad logarítmica óptima
- Aprovecha la estructura especial del problema
- No necesita examinar todos los elementos

Aplicaciones similares:

- Búsqueda en funciones unimodales
- Optimización convexa
- Búsqueda de puntos de equilibrio