

# ALGORITMOS Y ESTRUCTURAS DE DATOS II

## Divide y Vencerás



Alumnos: Rafael Guillén García

Grupo: 2.4

## ÍNDICE

### **1. Diseño de una Solución Divide y Vencerás**

- 1.1 Planteamiento del Problema
- 1.2 Estrategia Divide y Vencerás
- 1.3 Pseudocódigo del Algoritmo
- 1.4 Estructuras de Datos

### **2. Análisis Teórico**

- 2.1 Análisis Teórico del Tiempo de Ejecución +
- 2.2 Análisis del Mejor Caso
- 2.3 Análisis del Peor Caso
- 2.4 Orden de Ejecución

### **3. Implementación**

- 3.1 Función esValida
- 3.2 Función solucionDirecta
- 3.3 Función combinar
- 3.4 Función divideYVenceras

### **4. Validación de algoritmo**

- 4.1 Caso Normal
- 4.2 Caso de Cadena Corta
- 4.3 Caso de Caracteres Repetidos
- 4.4 Caso de Cadena Vacío
- 4.5 Caso de Cadena con Caracteres No Validos

### **5. Estudio Experimental**

### **6. Contracte entre estudio teórico y estudio práctico**

### **7. Conclusión**

### **Enunciado del Problema a Realizar**

Dada una cadena A con n caracteres y un conjunto S de 5 caracteres distintos, hay que encontrar todas las subcadenas de A formadas por 3 elementos de S sin repetir. Habrá que devolver como solución el número de subcadenas y su posición en la cadena C.

Por ejemplo, si A = abbfabcddfcbbade, n=16 si consideramos un conjunto de cinco caracteres S = {a, b, c, d, e} la solución es 4, en las posiciones 5, 6, 13 y 14.

## 1. Diseño de una Solución Divide y Vencerás

### 1.1) Planteamiento del Problema

El problema consiste en encontrar todas las subcadenas de longitud 3 en una cadena A teniendo en cuenta que:

- Están formadas por caracteres pertenecientes a un conjunto S de 5 caracteres.
- Los caracteres en cada subcadena no se pueden repetir.
- Se debe devolver el número total de subcadenas y sus posiciones.

### 1.2) Estrategia Divide y Vencerás

La solución sigue el método Divide y Vencerás:

#### a) División del Problema.

- Se divide la cadena original en dos mitades de tamaño aproximadamente igual.
- El punto de división es:  $\text{mitad} = (\text{inicio} + \text{fin}) / 2$ ;

```
68 | | int mitad = (inicio + fin) / 2;
```

- Se generan dos subproblemas:

- Subcadena izquierda: [inicio, mitad]

```
69 | | vector<int> izq = divideYVenceras(inicio, mitad);
```

- Subcadena derecha: [mitad + 1, fin]

```
70 | | vector<int> der = divideYVenceras(mitad + 1, fin);
```

#### b) Caso Base

```
1 | Si (fin - inicio + 1 <= 4) entonces
2 |   Resolver directamente buscando todas las subcadenas válidas
3 |   en el segmento [inicio, fin]
```

**Se elige un tamaño de 4**, ya que si el segmento tiene 4 caracteres o menos:

- Es posible verificar todas las cadenas de longitud 3 en un número constante de iteraciones (como máximo 2 subcadenas).
- No hace falta dividir más el problema, ya que el costo de resolverlo directamente es bajo.

#### Resolución directa:

En un principio, resolvimos el problema de otra manera, pero a la hora de implementar el método Divide y Vencerás tuvimos muchos problemas para que funcionara bien por ello, decidimos buscar otra forma para resolver el problema directamente.

Cuando se cumple la condición del caso base, se llama a la función `solucionDirecta` para procesar el segmento. Esta función:

```

24 vector<int> solucionDirecta(int inicio, int fin)
25 {
26     vector<int> posiciones;
27     for (int i = inicio; i <= fin - 2; i++)
28     {
29         if (esValida(i)) posiciones.push_back(i + 1);
30     }
31     return posiciones;
32 }

```

#### Itera sobre el segmento:

- Recorre todas las posiciones posibles en las que puede comenzar una subcadena de longitud 3.
- Se hace mediante un bucle que va desde inicio hasta fin - 2 (una subcadena de longitud 3 necesita al menos 3 caracteres).

#### Verifica si cada subcadena es válida:

- Llama a la función esValida para comprobar si la subcadena cumple las condiciones: los tres caracteres son distintos y los tres caracteres pertenecen al conjunto S.

```

10 bool esValida(int inicio)
11 {
12     if (inicio + 2 >= A.size())
13     {
14         return false;
15     }
16
17     char c1 = A[inicio];
18     char c2 = A[inicio + 1];
19     char c3 = A[inicio + 2];
20
21     return c1 != c2 && c1 != c3 && c2 != c3 && S.count(c1) && S.count(c2) && S.count(c3);
22 }

```

#### Guarda las posiciones válidas:

- Si una subcadena es válida, se guarda su posición inicial.

Ejemplo: supongamos que tenemos la cadena A = “abbeabc” y el conjunto S = {a, b, c, d, e}. Si el segmento actual es A[0...3] = “abbe”, se resolvería:

#### Iteración sobre las posiciones:

- i = 0: subcadena = “abb”, no es válida porque b se repite.
- i = 1: subcadena = “bbe”, no es válida porque b se repite.
- i = 2: subcadena = “bea”, válida porque: los tres caracteres son distintos y todos pertenecen al conjunto S.

La posición válida es 3, por ello, el resultado de solucionDirecta(0, 3) sería: 3.

c) Combinación de soluciones.

La función de combinación debe considerar tres casos:

- Subcadenas encontradas en la mitad izquierda.
- Subcadenas encontradas en la mitad derecha.
- Subcadenas que cruzan la separación entre ambas mitades.

La combinación se realiza en la función combinar:

```
vector<int> combinar(vector<int> izq, vector<int> der, int mitad)
{
    // Usamos un set temporal para eliminar duplicados
    set<int> posicionesUnicas;

    // Agregamos las posiciones de la izquierda
    for (int pos : izq)
    {
        posicionesUnicas.insert(pos);
    }

    // Agregamos las posiciones de la derecha
    for (int pos : der)
    {
        posicionesUnicas.insert(pos);
    }

    // Verificamos las subcadenas que cruzan la mitad
    for (int i = max(0, mitad - 2); i <= mitad; i++)
    {
        if (esValida(i))
        {
            posicionesUnicas.insert(i + 1);
        }
    }

    // Convertimos el set de vuelta a vector
    return vector<int>(posicionesUnicas.begin(), posicionesUnicas.end());
}
```

### 1.3) Pseudocódigo del Algoritmo

Función divideYVenceras(inicio, fin):

```
// Caso Base

Si (fin - inicio + 1 <= 4) entonces
    Devolver solucionDirecta(inicio, fin)

// Dividir

mitad = (inicio + fin) / 2
izquierda = divideYVenceras(inicio, mitad)
derecha = divideYVenceras(mitad + 1, fin)

// Combinar

Devolver combinar(izquierda, derecha, mitad)
```

Función esValida(posición):

```
Si posición + 2 >= longitud(A) entonces
    Devolver falso

c1 = A[posición]
c2 = A[posición + 1]
c3 = A[posición + 2]

Devolver (c1 != c2 Y c1 != c3 Y c2 != c3) Y (c1 PERTENECE_A S Y c2 PERTENECE_A
S Y c3 PERTENECE_A S)
```

Función combinar(izquierda, derecha, mitad):

```
resultado = conjunto_vacio()

// Añadir resultados de ambas mitades

Añadir todos los elementos de izquierda a resultado
Añadir todos los elementos de derecha a resultado

// Verificar subcadenas que cruzan la separación

Para i desde max(0, mitad - 2) hasta mitad:
    Si esValida(i) entonces
        Añadir (i + 1) a resultado

Devolver resultado
```

#### 1.4) Estructuras de Datos

En cuanto a las estructuras de datos utilizadas, encontramos:

##### 1. **set<char> S**

El conjunto S almacena 5 caracteres, que son aquellos permitidos para formar parte de las subcadenas válidas.

Un set, es una estructura de datos basada en árboles balanceados, y hemos decidido usarla ya que permite verificar si un carácter pertenece al conjunto S de una forma muy rápida. (Mediante la `S.count()`, que tiene un costo de  $O(\log 5)$ , que es  $O(1)$  en la práctica).

##### 2. **vector<int> posiciones**

El vector posiciones almacena las posiciones iniciales de las subcadenas válidas que se hayan encontrado en la cadena A.

Un vector es una estructura de datos dinámica que permite almacenar elementos en una lista contigua en memoria. Hemos decidido usarlo ya que nos permite almacenar las posiciones sin necesidad de saber cuántas subcadenas válidas habrá y por su rápido acceso por índice y capacidad de iteración eficiente.

##### 3. **set<int> posicionesUnicas**

El conjunto posicionesUnicas se utiliza en la función combinar de forma temporal para eliminar duplicados al unir las posiciones de las subcadenas válidas de las mitades izquierda y derecha.



## 2. Análisis Teórico

### 2.1) Análisis Teórico del Tiempo de Ejecución

#### Recurrencia del Algoritmo:

El tiempo de ejecución del algoritmo se puede expresar como:

$$T(n) = 2T(n/2) + C(n)$$

Donde:

- $T(n)$ : tiempo de ejecución para resolver el problema de tamaño  $n$ .
- $2T(n/2)$ : tiempo necesario para resolver los dos subproblemas de tamaño  $n/2$ .
- $C(n)$ : tiempo necesario para combinar las soluciones de los subproblemas.

#### Análisis de las Partes:

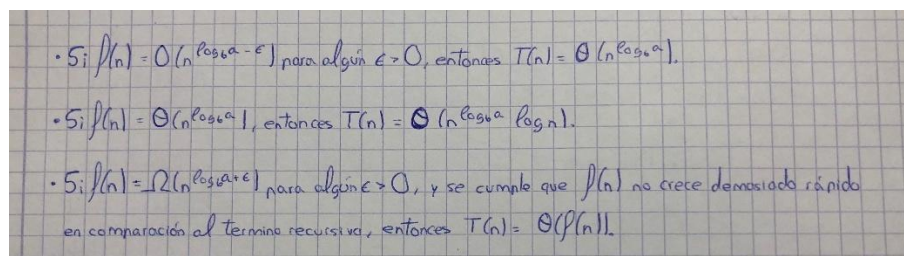
- **División:** dividir el problema en dos mitades tiene un costo constante  $O(1)$ , ya que solo se calcula el punto medio de la cadena.
- **Resolución:** se resuelven dos subproblemas de tamaño  $n/2$ , lo que contribuye con  $2T(n/2)$  al tiempo total.
- **Combinación:** se combinan las posiciones de las mitades izquierda y derecha (costo  $O(n)$ ), y verifica las subcadenas que cruzan la frontera (costo constante  $O(1)$ ). Por tanto, el costo de la combinación (función combinar) es  $C(n) = O(n)$

#### Resolución de la Recurrencia

La recurrencia es:  $T(n) = 2T(n/2) + O(n)$ .

- $a = 2$ : número de subproblemas
- $b = 2$ : factor de reducción del tamaño del problema ( $n \rightarrow n/2$ )
- $f(n) = O(n)$ : coste de la combinación

Sabemos que:



• Si  $f(n) = O(n^{\log_b a - \epsilon})$  para algún  $\epsilon > 0$ , entonces  $T(n) = O(n^{\log_b a})$ .

• Si  $f(n) = \Theta(n^{\log_b a})$ , entonces  $T(n) = \Theta(n^{\log_b a} \log n)$ .

• Si  $f(n) = \Omega(n^{\log_b a + \epsilon})$  para algún  $\epsilon > 0$ , y se cumple que  $f(n)$  no crece demasiado rápido en comparación al término recursivo, entonces  $T(n) = O(f(n))$ .

En este caso:

- $\log_b a = \log_2 2 = 1$
- $f(n) = O(n)$ , que es lo mismo que  $n^{\log_b a}$

Por tanto, el tiempo de ejecución es:  $T(n) = O(n \log n)$

## 2.2) Análisis del Mejor Caso

El Mejor Caso se da cuando no hay subcadenas válidas en la cadena:

- La función esValida siempre devolvería false, y no se almacenarían posiciones válidas.
- El tiempo de ejecución seguiría siendo  $T(n) = \Theta(n \log n)$ , ya que el algoritmo aún debe dividir la cadena, resolver los problemas y combinar los resultados.

## 2.3) Análisis del Peor Caso

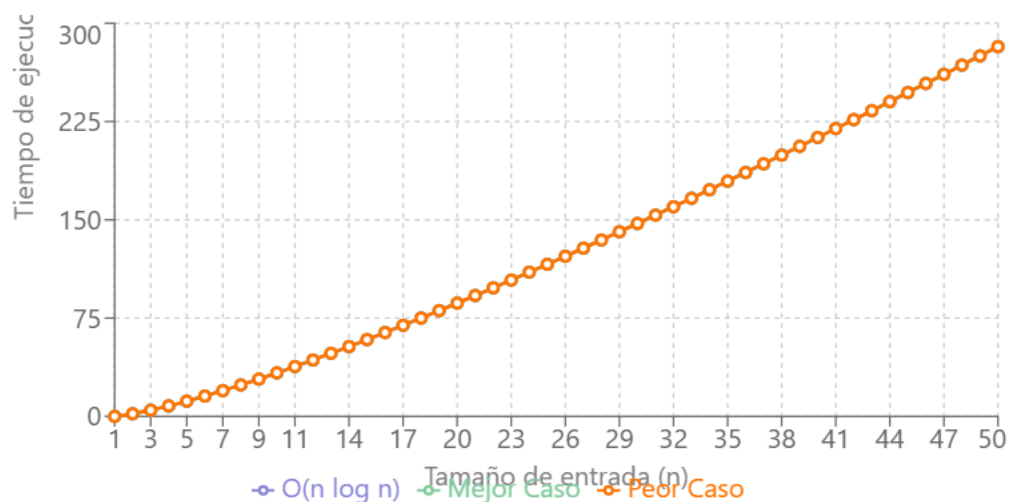
El Peor Caso ocurre cuando todas las subcadenas de longitud 3 son válidas:

- La función esValida siempre devolvería true, y se almacenarían todas las cadenas posibles.
- El tiempo de ejecución sigue siendo  $T(n) = \Theta(n \log n)$ , ya que el costo de verificar las subcadenas y combinar los resultados no cambia el orden de complejidad.

## 2.4) Orden de Ejecución

El Orden de Ejecución del algoritmo es:  $T(n) = \Theta(n \log n)$ .

Esto significa que el tiempo de ejecución crece de manera proporcional a  $n \log n$ , siendo  $n$  el tamaño de la cadena de entrada.



### 3. Implementación

#### 3.1) Función esValida

```
// Verifica si una subcadena de longitud 3 que comienza en 'inicio' es válida
bool esValida(const string& A, const set<char>& S, int inicio)
{
    // Verificar que hay espacio para una subcadena de longitud 3
    if (inicio + 2 >= A.size())
    {
        return false;
    }

    // Obtener los tres caracteres consecutivos
    char c1 = A[inicio];
    char c2 = A[inicio + 1];
    char c3 = A[inicio + 2];

    // Verificar que son diferentes entre sí y pertenecen al conjunto S
    return c1 != c2 && c1 != c3 && c2 != c3 && S.count(c1) && S.count(c2) && S.count(c3);
}
```

Esta función verifica si una subcadena de longitud 3 es válida:

- Recibe la cadena A, el conjunto S y la posición inicial.
- Primero, comprueba si hay espacio suficiente para una subcadena de 3 caracteres.
- Después, obtiene los tres caracteres consecutivos desde la posición de inicio.
- Verifica dos condiciones: que los tres caracteres sean distintos entre sí y que los tres caracteres pertenecen al conjunto S.

#### 3.2) Función solucionDirecta

```
// Resuelve directamente el problema para segmentos pequeños (<= 4 caracteres)
vector<int> solucionDirecta(const string& A, const set<char>& S, int inicio, int fin)
{
    vector<int> posiciones;
    // Revisar cada posición posible para una subcadena de longitud 3
    for (int i = inicio; i <= fin - 2; i++)
    {
        if (esValida(A, S, i)) posiciones.push_back(i + 1); // Se suma 1 para ajustar
    }
    return posiciones;
}
```

Esta función resuelve el problema para segmentos pequeños:

- Recibe la cadena A, el conjunto S, y los límites del segmento.
- Crea un vector para almacenar las posiciones válidas.
- Revisa cada posición posible dentro del segmento.
- Si se encuentra una subcadena válida, guarda su posición.
- Devuelve todas las posiciones encontradas

### 3.3) Función combinar

```
// Combina las soluciones de dos subsegmentos y verifica las subcadenas que cruzan la frontera
vector<int> combinar(const string& A, const set<char>& S, vector<int> izq, vector<int> der, int mitad)
{
    // Set temporal para eliminar duplicados
    set<int> posicionesUnicas;

    // Agregar todas las posiciones del segmento izquierdo
    for (int pos : izq)
    {
        posicionesUnicas.insert(pos);
    }

    // Agregar todas las posiciones del segmento derecho
    for (int pos : der)
    {
        posicionesUnicas.insert(pos);
    }

    // Verificar las subcadenas que podrían cruzar la frontera entre segmentos
    for (int i = max(0, mitad - 2); i <= mitad; i++)
    {
        if (esValida(A, S, i))
        {
            posicionesUnicas.insert(i + 1); // Ajuste de posiciones
        }
    }

    // Convertir el set a vector para el return
    return vector<int>(posicionesUnicas.begin(), posicionesUnicas.end());
}
```

Esta función combina las soluciones de los segmentos izquierdo y derecho, y verifica si hay subcadenas válidas que cruzan la frontera entre ambos segmentos.

- Se utiliza un set para almacenar las posiciones de las subcadenas válidas, sin duplicados.
- Se revisan las posiciones cercanas a la frontera (desde mitad – 2 hasta mitad) para verificar si hay subcadenas válidas que abarcan elementos de ambos subsegmentos.
- Si se encuentra una válida, se agrega su posición
- Se convierte el set a vector.

### 3.4) Función divideYVenceras

```
// Implementación principal del algoritmo
vector<int> divideYVenceras(const string& A, const set<char>& S, int inicio, int fin)
{
    // Caso base: segmento pequeño (<= 4 caracteres)
    if (fin - inicio + 1 <= 4) return solucionDirecta(A, S, inicio, fin);

    // Dividir el problema en dos partes
    int mitad = (inicio + fin) / 2;
    // Resolver recursivamente cada mitad
    vector<int> izq = divideYVenceras(A, S, inicio, mitad);
    vector<int> der = divideYVenceras(A, S, mitad + 1, fin);

    // Combinar las soluciones y manejar el caso de la frontera
    return combinar(A, S, izq, der, mitad);
}
```

Esta función busca todas las posiciones de subcadenas válidas en la cadena A:

- Si el segmento actual tiene 4 o menos caracteres (Caso Base), se resuelve directamente (solucionDirecta).
- El segmento actual se divide en dos partes: izquierda (inicio hasta mitad) y derecha (mitad + 1 hasta fin).
- Se llama recursivamente a divideYVenceras para resolver cada subsegmento (el izquierdo y el derecho).
- Las soluciones de ambos subsegmentos se combinan mediante la función combinar.
- Se devuelve un vector con todas las posiciones de las subcadenas válidas.

## 4. Validación del Algoritmo

### 4.1) Caso Normal

#### Prueba 1:

Usaremos:

- Cadena A = "abcde", n = 5.
- Conjunto S = {'a', 'b', 'c', 'd', 'e'}

El resultado esperado es:

- Tres Subcadenas válidas.
- Posiciones: 1, 2 y 3.

```
Ingrese la cadena A:  
abcde  
Ingrese los 5 caracteres distintos del conjunto S: (Ejemplo: abcde)  
abcde  
Número de subcadenas encontradas: 3  
Posiciones: 1 2 3
```

#### Prueba 2:

Usaremos:

- Cadena A = "abbbfabcdcfcbade", n = 16
- Conjunto S = {'a', 'b', 'c', 'd', 'e'}

El resultado esperado es:

- Cuatro subcadenas válidas.
- Posiciones: 5, 6, 13 y 14.

### 4.2) Caso de Cadena Corta

#### Prueba 1:

Usaremos:

- Cadena A = "ab"
- Conjunto S = {'a', 'b', 'c', 'd', 'e'}

El resultado esperado es:

- Cero subcadenas válidas.

```
Ingrese la cadena A:  
ab  
Ingrese los 5 caracteres distintos del conjunto S: (Ejemplo: abcde)  
abcde  
Número de subcadenas encontradas: 0  
Posiciones:
```

### Prueba 2:

Usaremos:

- Cadena A = "a"
- Conjunto S = {'a', 'b', 'c', 'd', 'e'}

El resultado esperado es:

- Cero subcadenas válidas.

```
Ingrese la cadena A:  
a  
Ingrese los 5 caracteres distintos del conjunto S: (Ejemplo: abcde)  
abcde  
Número de subcadenas encontradas: 0  
Posiciones:
```

#### 4.3) Caso de Caracteres Repetidos

##### Prueba:

Usaremos:

- Cadena A = "aabcd"
- Conjunto S = {'a', 'b', 'c', 'd', 'e'}

El resultado esperado es:

- Dos subcadenas válidas.
- Posiciones: 2 y 3.

```
Ingrese la cadena A:  
aabcd  
Ingrese los 5 caracteres distintos del conjunto S: (Ejemplo: abcde)  
abcde  
Número de subcadenas encontradas: 2  
Posiciones: 2 3
```

#### 4.4) Caso de Cadena Vacía

En el caso de una cadena vacía, el programa quedará esperando a que el usuario introduzca una cadena.

#### 4.5) Caso de Cadena con Caracteres No Válidos

##### Prueba 1

Usaremos:

- Cadena A = “xyzabc”
- Conjunto S = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’}

El resultado esperado es:

- Una subcadena válida.
- Posición: 4.

```
Ingrese la cadena A:
xyzabc
Ingrese los 5 caracteres distintos del conjunto S: (Ejemplo: abcde)
abcde
Número de subcadenas encontradas: 1
Posiciones: 4
```

##### Prueba 2

Usaremos:

- Cadena A = “vwzyx”
- Conjunto S = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’}

El resultado esperado es:

- Cero subcadenas encontradas.
- Posición: Nula.

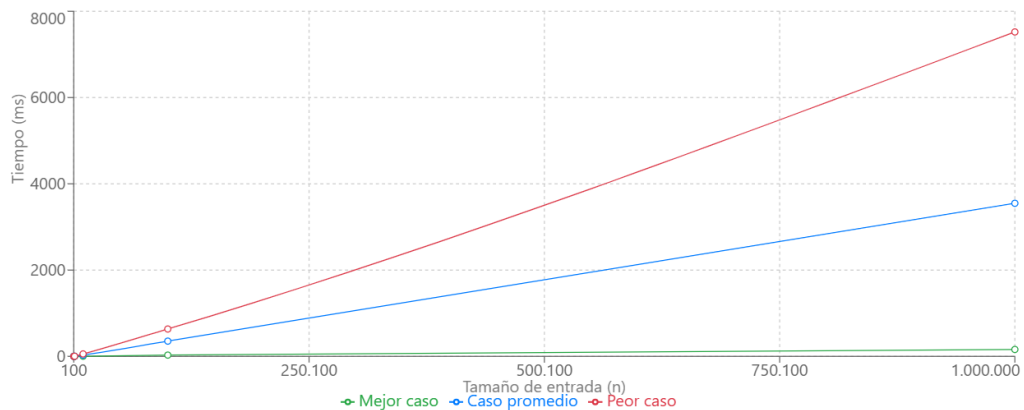
```
Ingrese la cadena A:
vwxyz
Ingrese los 5 caracteres distintos del conjunto S: (Ejemplo: abcde)
abcde
Número de subcadenas encontradas: 0
Posiciones:
```



## 5. Estudio Experimental.

Para este estudio, preguntamos a una Inteligencia Artificial para que nos ayudará a crear el generador de los casos, ya que tuvimos problemas para poder generar las cadenas A de distintas longitudes (las potencias de 10) y para poder generar los ficheros.

A la hora de medir el tiempo de ejecución utilizamos el comando time de Linux, ya que lo conocíamos de haberlo usado en AED I, y lo probamos con los casos que habíamos generado. El resultado es:



Para los datos:

|           | n favorable | promedio | desfavorable |
|-----------|-------------|----------|--------------|
| 100       | 0           | 0.999    | 0            |
| 1.000     | 0           | 2.555    | 3.041        |
| 10.000    | 3.999       | 29.734   | 56.696       |
| 100.000   | 29.341      | 353.433  | 634.23       |
| 1.000.000 | 160.021     | 3548.9   | 7520.99      |

## 6. Contraste entre Estudio Teórico y Estudio Práctico

El estudio experimental confirma el comportamiento teórico del algoritmo, mostrando un crecimiento compatible con  $\Theta(n \log n)$ .

En el mejor caso, el tiempo de ejecución es bastante bajo ya que el algoritmo apenas realiza trabajo en estos casos, ya que hay pocas cadenas válidas que procesar.

En el peor caso, el tiempo de ejecución es mayor que el promedio, algo que se esperaba, pero no se aleja tanto de lo esperado teóricamente.

## **7. Conclusión**

Nos ha sorprendido la dificultad de la práctica, ya que no estábamos familiarizados con el algoritmo de Divide y Vencerás, además de lo extensa que es, por los distintos apartados como el generador de casos, las mediciones del tiempo y sobre todo el diseño del algoritmo, ya que tuvimos bastantes errores a la hora de gestionar subcadenas, tanto como para contar las posiciones como para que funcionara correctamente el código.

Le hemos dedicado más o menos entre 20-25 horas, siempre trabajando los dos juntos, contando las horas de las clases de prácticas dedicadas a realizar dicho ejercicio.