Análisis y Diseño de Algoritmos

Divide y Vencerás

Francisco Javier Mercader Martínez
Pedro Alarcón Fuentes

En esta memoria se explicará el código utilizado para resolver el problema de encontrar, en una cadena dada, la subcadena de longitud `m` que contiene la mayor cantidad de apariciones consecutivas de un carácter específico `C`. Las dos funciones son:

1. **`resolver_directo(A, m, C)`**
2. **`divide_y_venceras(A, m, C, l, r)`**

1. Función `resolver_directo(A, m, C)`

```python
def resolver_directo(A, m, C):
    """
    Encuentra la subcadena óptima utilizando un método directo

    :param A: Cadena original (str)
    :param m: Longitud de la cadena (int)
    :param C: Carácter a buscar (str)
    :return: tuple: Índice de inicio de la subcadena óptima y el número máximo de
    ↪ apariciones consecutivas.
    """
    n = len(A)
    max_consecutivos = 0
    inicio_optimo = -1

    # Recorrer todas las subcadenas de longitud m
    for i in range(n - m + 1):
        subcadena = A[i:i + m]

        # Contar el número máximo de apariciones consecutivas de C en la
        ↪ subcadena
        contador_actual = 0
        max_actual = 0

        for char in subcadena:
            if char == C:
                contador_actual += 1
                if contador_actual > max_actual:
                    max_actual = contador_actual
```

```
            else:
                contador_actual = 0

        # Actualizar la mejor solución encontrada
        if max_actual > max_consecutivos:
            max_consecutivos = max_actual
            inicio_optimo = i

    return inicio_optimo, max_consecutivos
```

Descripción general

Esta función implementa un algoritmo directo que examina todas las posibles de longitud **m** en la cadena **A** y determina cuál de ellas contiene el mayor número de apariciones consecutivas del carácter **C**. Es un enfoque de fuerza bruta que garantiza encontrar la solución óptima al evaluar exhaustivamente todas las opciones posibles.

Parámetros de entrada

- **A**: Cadena original donde se buscarán las subcadenas.
- **m**: Longitud de las subcadenas a considerar.
- **C**: Carácter cuyo número de apariciones consecutivas se desea maximizar.

Proceso del algoritmo

1. **Preparación:**
   - Calculamos **n**, que es la longitud de la cadena **A**.
   - Inicializamos **max_consecutivos** en 0 para almacenar el máximo de **C** consecutivos encontrados hasta ahora.
   - Inicializamos **inicio_optimo** en -1 para guardar el índice de inicio de la mejor subcadena encontrada.
2. **Recorrido de subcadenas:**
   - Se utiliza un bucle **for** que va desde **i = 0** hasta **i = n - m**, de modo que se puedan extraer todas las subcadenas de longitud **m** sin exceder los límites de la cadena.
   - En cada iteración, se extrae la subcadena **subcadena = A[i:i + m]**.
3. **Cálculo de apariciones consecutivas:**
   - Para cada subcadena, se inicializan **contador_actual** y **max_actual** a 0.
   - Recorremos cada carácter de la subcadena:
     - Si el carácter es igual a **C**, se incrementa **contador_actual** y se actualiza **max_actual** si **contador_actual** es mayor.
     - Si el carácter no es **C**, se reinicia **contador_actual** a 0.

- Este proceso permite determinar el número máximo de apariciones consecutivas de **C** en la subcadena actual.

4. **Actualización de la mejor solución:**

   - Si **max_actual** es mayor que **max_cosecutivos**, se actualizan **max_consecutivos** con **max_actual** y **inicio_optimo** con el índice actual **i**.

5. **Resultado:**

   - Al finalizar el bucle, se retorna una tupla **(inicio_optimo, max_consecutivos)**, que indica el índice de inicio de la subcadena óptima y el número máximo de apariciones consecutivas de **C** en dicha subcadena.

Ejemplo de uso

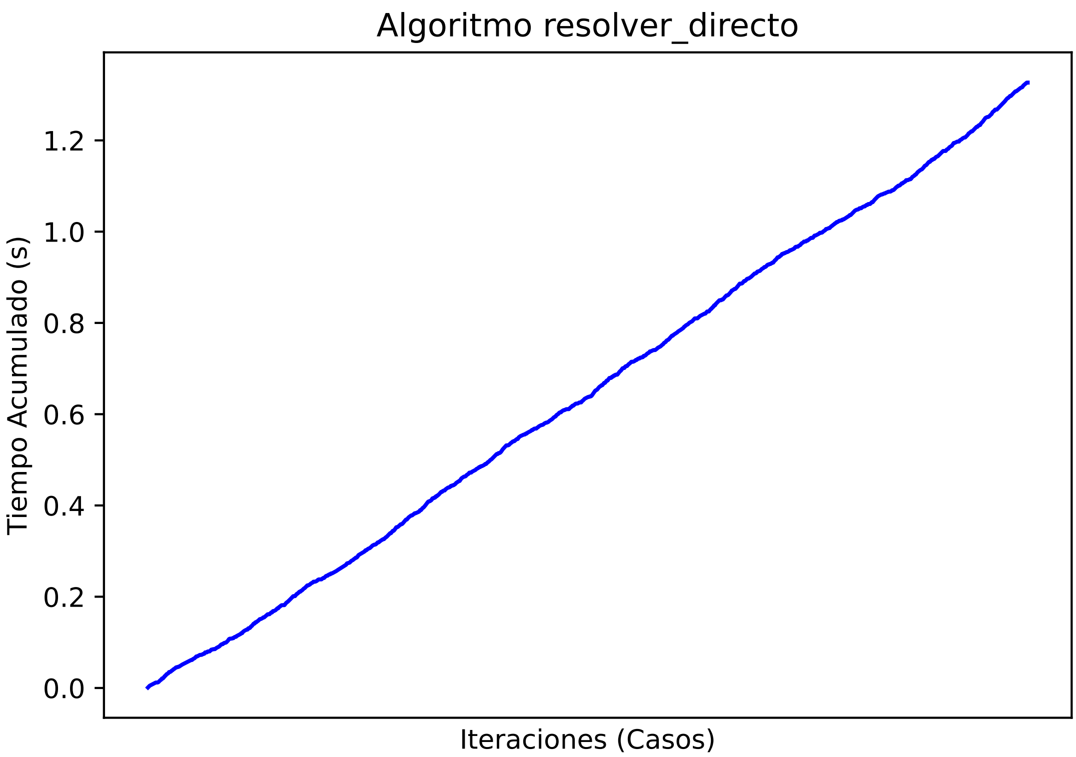
```
if __name__ == '__main__':
    # Generar una cadena de ejemplo con el alfabeto
    alfabeto = "abcdefghijklmnopqrstuvwxyz"
    C = 'c'  # Carácter a buscar
    m = 100  # Tamaño de la subcadena fijo
    num_pruebas = 10 # Número de pruebas a realizar para comprobar que el código
↪   funciona

    for i in range(num_pruebas):
        print(f"\n -- Prueba {i + 1} --")
        A = ''.join(random.choices(alfabeto, k=10000))  # Cadena aleatoria de
↪   longitud 10000
        resultado = resolver_directo(A, m, C)
        print(f"Índice de inicio: {resultado[0]} \nMáximo de apariciones
         ↪  consecutivas: {resultado[1]}")
```

```
##
##  -- Prueba 1 --
## Índice de inicio: 84
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 2 --
## Índice de inicio: 7476
## Máximo de apariciones consecutivas: 3
##
##  -- Prueba 3 --
## Índice de inicio: 692
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 4 --
## Índice de inicio: 619
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 5 --
```

```
## Índice de inicio: 258
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 6 --
## Índice de inicio: 1931
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 7 --
## Índice de inicio: 131
## Máximo de apariciones consecutivas: 3
##
##  -- Prueba 8 --
## Índice de inicio: 60
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 9 --
## Índice de inicio: 365
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 10 --
## Índice de inicio: 274
## Máximo de apariciones consecutivas: 2
```

Análisis de la complejidad



El algoritmo tiene una complejidad temporal lineal respecto al tamaño de la cadena `A` y la longitud de las subcadenas `m`, ya que recorren `n - m + 1` subcadenas y, para cada una, se realiza un recorrido de longitud `m`. Por lo tanto tiene una complejidad teórica de $O(n \cdot m)$.

2. Función divide_y_venceras(A, m, C, l, r)

```python
def divide_y_venceras(A, m, C, l, r):
    """
    Esquema recursivo del algoritmo divide y vencerás.

    :param A: Cadena original (str)
    :param m: Longitud de la subcadena (int)
    :param C: Carácter a buscar (str)
    :param l: Índice izquierdo del rango actual (int)
    :param r: Índice derecho del rango actual (int)
    :return: tuple: Índice de inicio de la subcadena óptima y el número máximo de
    ↪ apariciones consecutivas.
    """
    if r - l + 1 <= m:
        return resolver_directo(A[l:r + 1], m, C)

    mid = (l + r) // 2

    # Soluciones para las dos mitades
    sol_izq = divide_y_venceras(A, m, C, l, mid)
    sol_der = divide_y_venceras(A, m, C, mid + 1, r)

    # Solución que cruza el centro
    max_central_consecutivos = 0
    inicio_central = -1

    # Optimización de la solución central
    max_izq = 0
    contador = 0
    for i in range(mid, l - 1, -1):
        if A[i] == C:
            contador += 1
            max_izq = max(max_izq, contador)
        else:
            break

    max_der = 0
    contador = 0
    for i in range(mid + 1, r + 1):
        if A[i] == C:
            contador += 1
            max_der = max(max_der, contador)
        else:
            break

    sol_central = (mid - max_izq + 1, max_izq + max_der)

    return max(sol_izq, sol_der, sol_central, key=lambda x: x[1])
```

Descripción general

Esta función aplica el método "divide y vencerás" para encontrar la mejor subcadena de manera más eficiente, es decir, divide recursivamente la cadena en mitades y resuelve el problema de cada mitad, combinando las soluciones para encontrar la óptima.

Parámetros de entrada

- `A`: Cadena original.
- `m`: Longitud de las subcadenas a considerar.
- `C`: Carácter cuyo número de apariciones consecutivas se desea maximizar.
- `l`: Índice izquierdo del rango actual de la cadena `A`.
- `r`: Índice derecho del rango actual de la cadena `A`.

Proceso del algoritmo

1. **Casos base:**
   - Si el tamaño del segmento actual (`r - l + 1`) es menor o igual a `m`, se resuelve el problema llamando a `resolver_directo` con la subcadena `A[l:r + 1]`.
2. **División de la cadena:**
   - Se calcula el punto medio `mid = (l + r) // 2`.
   - Se realizan dos llamadas recursivas:
     - `sol_izq`: Resultado de aplicar el algoritmo a la mitad izquierda (`l` a `mid`).
     - `sol_der`: Resultado de aplicar el algoritmo a la mitad derecha (`mid + 1` a `r`).
3. **Solución general:**
   - Se busca una solución que cruce el punto medio, ya que la subcadena óptima podría abarcar ambas mitades.
   - Se inicializan `max_central_consecutivos` a 0 y `inicio_central` a -1.
   - Se recorre desde `i = mid - m + 1` hasta `i = mid` para considerar todas subcadenas de longitud `m` que cruzan el punto medio.
     - Se verifica que `i` esté dentro de los límites (`l` y `r`).
     - Para cada subcadena, se calcula el número máximo de apariciones consecutivas de `C` de manera similar al método directo.
     - Si se encuentra un `max_actual` mayor que `max_central_consecutivos`, se actualizan `max_central_consecutivos` e `inicio_central`.
4. **Combinación de soluciones:**
   - Se comparan las soluciones `sol_izq`, `sol_der` y `sol_central` utilizando una función clave que evalúa el segundo elemento de las tuplas (el número máximo de apariciones consecutivas).
   - Devuelve la solución que tenga el mayor número de apariciones consecutivas de `C`.

Ejemplo de uso

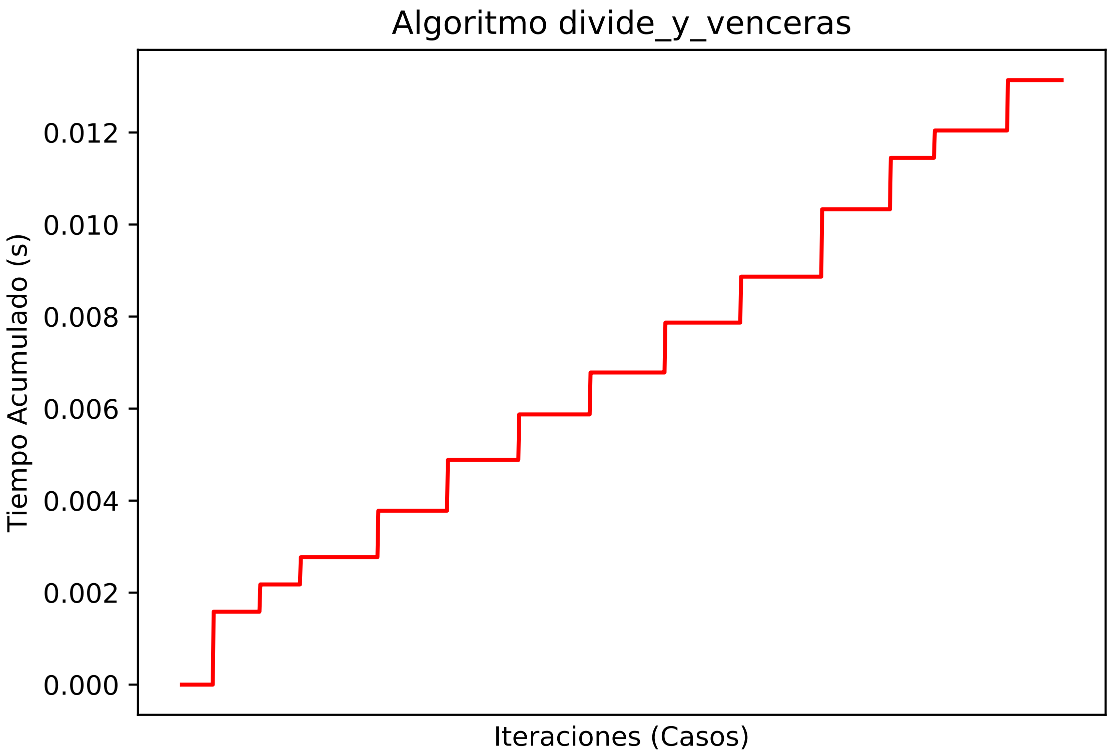
```python
if __name__ == '__main__':
    # Generar una cadena de ejemplo con el alfabeto
    alfabeto = "abcdefghijklmnopqrstuvwxyz"
    C = 'c'  # Carácter a buscar
    m = 100  # Tamaño de la subcadena fijo
    num_pruebas = 10 # Número de pruebas a realizar para comprobar que el código
↪   funciona

    for i in range(num_pruebas):
        print(f"\n -- Prueba {i + 1} --")
        A = ''.join(random.choices(alfabeto, k=10000))  # Cadena aleatoria de
↪   longitud 10000
        resultado = divide_y_venceras(A, m, C, 0, len(A) - 1)
        print(f"Índice de inicio: {resultado[0]} \nMáximo de apariciones
         ↪  consecutivas: {resultado[1]}")
```

```
##
##  -- Prueba 1 --
## Índice de inicio: 313
## Máximo de apariciones consecutivas: 1
##
##  -- Prueba 2 --
## Índice de inicio: 1718
## Máximo de apariciones consecutivas: 1
##
##  -- Prueba 3 --
## Índice de inicio: 1328
## Máximo de apariciones consecutivas: 1
##
##  -- Prueba 4 --
## Índice de inicio: 3593
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 5 --
## Índice de inicio: 2187
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 6 --
## Índice de inicio: 157
## Máximo de apariciones consecutivas: 1
##
##  -- Prueba 7 --
## Índice de inicio: 1719
## Máximo de apariciones consecutivas: 1
##
##  -- Prueba 8 --
## Índice de inicio: 3436
## Máximo de apariciones consecutivas: 2
##
```

```
##  -- Prueba 9 --
## Índice de inicio: 5546
## Máximo de apariciones consecutivas: 2
##
##  -- Prueba 10 --
## Índice de inicio: 2813
## Máximo de apariciones consecutivas: 2
```

Análisis de la complejidad



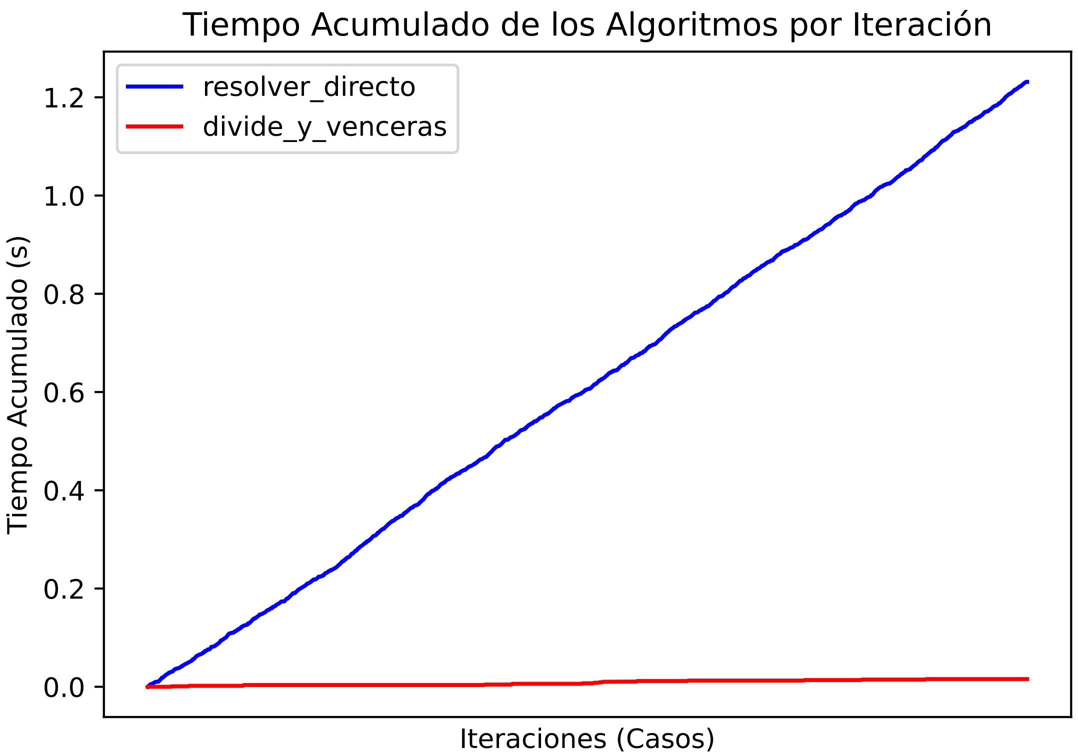
El algoritmo divide la cadena en mitades logarítmicamente y en cada nivel realiza un trabajo lineal, lo que resultado en una complejidad teórica de $O(n \log n)$.

Conclusión

El código proporciona dos enfoques para resolver el problema de encontrar la subcadena de longitud `m` con el mayor número de apariciones consecutivas de un carácter `C` en una cadena `A`:

- **Método Directo**: Es sencillo de entender e implementar, pero puede ser ineficiente para cadenas muy largas debido a su complejidad $O(n \cdot m)$.
- **Método Divide y Vencerás**: Es más eficiente con una complejidad $O(n \log n)$, pero es más complejo y requiere un manejo cuidadoso de los casos base y la combinación de soluciones.

En la siguiente gráfica se mostrará la diferencia de tiempo que necesitan ambos algoritmos para resolver el mismo número de casos:


Tiempo Acumulado de los Algoritmos por Iteración
resolver_directo
divide_y_venceras
Tiempo Acumulado (s)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Iteraciones (Casos)