Análisis de Algoritmos
Taller 3
Carlos Barón
Andrés Cocunubo

1 Descripción problema

Encontrar la subsecuencia palíndroma consecutiva más larga a partir de una secuencia de entrada.

2 Formalización

Dada una secuencia I de n elementos, donde algunos poseen una relación de equivalencia, encontrar la máxima subsecuencia I' que cumpla la condición de ser $= I'$ invertido.

2.1 Entradas

Una secuencia I de n elementos:

$$x_1, x_2, ..., x_n$$

, donde cada

$$x_n$$

está definido en la relación de equivalencia.

2.2 Salidas

Una subsecuencia de elementos

$$x_0, ..., x_i, ..., x_{n-1}$$

donde,

$$x_0 = x_{n-1}$$

$$x_i = x_{n-1-i}$$

3 Algoritmos

3.1 Fuerza Bruta

```
procedimiento EsPalindroma(palabra)
    cad <- []
    for k <- |palabra| downto 1 do
        cad <- palabra[k]
    if cad == palabra return True

procedimiento palindroma(palabra)
    indices <- [ ]
    for i <- 1 to |palabra| do
        for j <- |palabra| downto 0 do
            if EsPalindroma(palabra[i : j]) then
                if |palabra[indices[1] : indices[2]]| < |palabra[i : j]| then
                    indices[1] <- i
                    indices[2] <- j
    return indices
```

3.2 Dividir y vencer

```
procedimiento IsPalindrome(A, l, h)
    x <- True
    aux <- 0
    for i <- l to h do
        if A[i] != A[h - aux] then
            x <- False
        aux <- aux + 1
    return x

procedimiento compSubPalindrome(A, B, C)
    if A[2]-A[1] >= B[2]-B[1] and A[2]-A[1] >= C[2]-C[1] then return A
    else if B[2]-B[1] >= A[2]-A[1] and B[2]-B[1] >= C[2]-C[1] then return B
    else return C

procedimiento FindMaxCrossPalindromeAux(i, j, r, A)
    retorno <- r
    cond <- True
    while i >= 0 and j <= |A| - 1 and cond do
        if A[i] == A[j] do
            retorno <- i
            retorno <- j
        else
            cond <- False
        i <- i - 1
```

```
        j < -j + 1
    return retorno

procedimiento FindMaxCrossPalindrome(A, l, m, h)
    B < -[l, l]
    C < -[l, l]
    D < -[l, l]
    if m - 1 >= 0 and A[m - 1] == A[m + 1] do
        B < -m - 1
        B < -m + 1
        B < -FindMaxCrossPalindromeAux(m - 2, m + 2, B, A)
    if A[m] == A[m + 1] do
        C < -m
        C < -m + 1
        C < -FindMaxCrossPalindromeAux(m - 1, m + 2, C, A)
    if m - 1 >= 0 and A[m - 1] == A[m] do
        D < -m - 1
        D < -m
        D < -FindMaxCrossPalindromeAux(m - 2, m + 1, D, A)
    return compSubPalindrome(B, C, D)

procedimiento FindMaxPalindrome(A, l, h)
    if h <= l do
        X < -[]
        X < -l
        X < -h
        returnX
    else
        L < -[]
        R < -[]
        C < -[]
        m < -(l + h)/2
        L < -FindMaxPalindrome(A, l, m)
        R < -FindMaxPalindrome(A, m + 1, h)
        C < -FindMaxCrossPalindrome(A, l, m, h)
        LP < -IsPalindrome(A, l, m)
        RP < -IsPalindrome(A, m + 1, h)
        FINAL < -[]
        if RP do
            R < -m + 1
            R < -h
        if LP do
            L < -l
            L < -m
        FINAL < -compSubPalindrome(L, R, C)
        if FINAL[1] - FINAL[0] > maxp[1] maxp[0] do
```

$maxp < -FINAL[0]$
$maxp < -FINAL[1]$
$return\ FINAL$

4 Comparación

La complejidad teórica por inspección del algoritmo de fuerza bruta es

$$O(n^3)$$

y para el algoritmo dividir y vencer usando el teorema maestro es

$$\Theta(n * log_2 n)$$

Teniendo en cuenta esto y los algoritmos se puede notar que en el primero se deben recorrer los 3 ciclos en el peor de los casos, mientras que para el segundo se realizan dos llamados recurrentes y en cada uno se hace un ciclo de complejidad n.
Para la realización del algoritmo de fuerza bruta no se requirió el mayor esfuerzo en el planteamiento del diseño, sin embargo, para dividir y vencer fué necesario determinar un diseño mucho más elaborado, puesto que al momento de dividir el problema se debe conocer que operaciones aplicar y poder determinar la mejor solución.
Al momento de realizar la pruebas se logró evidenciar que el algoritmo de fuerza bruta tardó un tiempo en mostrar el resultado correspondiente, mientras que el algoritmo dividir y vencer entregaba la solución casi de inmediato, demostrando así que este último es mucho más eficiente.
Además, dividir y vencer necesita más variables para ir almacenando sus datos y fuerza bruta menor cantidad por lo que en cierto punto puede, dividir y vencer consumir más memoria.

5 Manual de uso

Para ejecutar el programa se debe ejecutar Palindroma.py con python3 para la solucion divide y venceras ó palindromaBruta.py para la solucion de fuerza bruta Ejemplo ejecución en ubuntu:

1. Abrir una terminal en linux en el directorio del programa.
2. Ejecutar en la terminal "python3 Palindroma.py" ó "python3 palindromaBruta.py".

6 Descripción de pruebas

- ListT: Secuencia en la que toda la secuencia es una palindroma.

- ListI: Secuencia en la que la maxima subsecuencia palindroma se encuentra en el extremo izquierdo.
- ListD: Secuencia en la que la maxima subsecuencia palindroma se encuentra en el extremo derecho.
- ListM: Secuencia en la que la maxima subsecuencia palindroma se encuentra entre otras subsecuencias.
- List100: Secuencia de 100 elementos con una palindroma visible entre otras subsecuencias.
- List500: Secuencia de 500 elementos con una palindroma visible entre otras subsecuencias.
- List1000: Secuencia de 1000 elementos con una palindroma visible entre otras subsecuencias.