



Instituto Politécnico Nacional Escuela Superior de Cómputo



Análisis de Algoritmos, Sem: 2022-2, 3CV11, Práctica 4, 14 de mayo de 2022

PRÁCTICA 4: DIVIDE Y VENCERÁS

Luis Francisco Renteria Cedillo, Denzel Omar Vazquez Perez.

lrenteriac1400@alumno.ipn.mx, dvazquezp1600@alumno.ipn.mx

Resumen: En el presente documento se muestra la implementación y el análisis tanto *a Priori* como *a Posteriori* de dos algoritmos, cuyo objetivo de ambos es girar una imagen textual en arte ASCII 90° utilizando estrategias de diseño de algoritmos como "*Divide y Vencerás*" y "*Fuerza Bruta*".

Palabras Clave: Imagen, Rotar, Análisis, Recursividad, Fuerza bruta.

1 Introducción

Con el tiempo el uso de algoritmos ha aumentado teniendo un crecimiento exponencial puesto a la mejora de estrategias para dar solución a cualquier problema de programación, ya que estos tienen una suma importancia en el desarrollo de cualquier aplicación.

Ante esto, el llevar a cabo un análisis de complejidad del algoritmo permite conocer si este es adecuado para el caso presentado, dada su entrada de datos de tamaño n, pero ¿qué es n?, la definición de n depende de la naturaleza del problema, ya que puede representar el tamaño de una arreglo, los nodos de un grafo o el número de elementos de una matriz.

Como ejemplo de lo dicho anteriormente son las imágenes digitales, que al tener características discretas, se puede asociar a una matriz de tamaño $M \times N$ donde cada elemento de esta guarda su definición de color al tomar valores numéricos de 0 a 255.

Así al conocer lo antes mencionado es posible las operaciones como de ajuste de brillo, invertir colores, el ajuste de canales y la rotación de la imagen misma, pero ¿cómo se logran a cada una de estas?.

De este modo en el presente documento se implementan 2 algoritmos para rotación de matrices que permitirán girar 90° una imagen textual en arte ASCII, haciendo uso de la estrategia "*Divide y vencerás*" comparando esta misma ante la implementación iterativa del problema a resolver.

2 Conceptos teóricos

2.1 Imagen

Una imagen es una función bidimensional $f(x, y)$, cuyo valor representa la intensidad de la imagen en las coordenadas (x, y) .

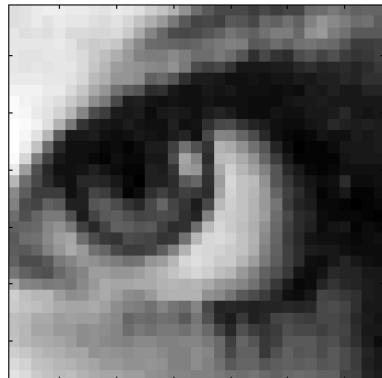


Figura 1: Imagen digital

2.2 Imagen textual en arte ASCII

Las imágenes textuales en arte ASCII son la formación de figuras o "arte" a partir de los caracteres disponibles en el gráfico conocido como ASCII . Estas imágenes eran populares antes de Internet y la banda ancha cuando los usuarios se conectaban a través de un BBS (Bulletin Board System) el cual era una pequeña red donde los usuarios se comunicaban y compartían cosas.

2.3 Recursividad

Es una opción distinta para llevar a cabo construcciones de repetición (ciclos). Se puede utilizar en toda situación en la cual la solución logre ser expresada como una serie de movimientos, pasos o transformaciones gobernadas por un grupo de normas no ambiguas.

2.4 Fuerza bruta

Es un enfoque de solución, usualmente basado en el enunciado del problema y las definiciones de conceptos involucrados.

2.5 Divide y vencerás

Algunos algoritmos se orientan a la hora de resolver un problema utilizando el enfoque conocido como ”divide y vencerás” donde la cuestión a tratar se divide en distintos subproblemas parecidos al problema raíz, pero más pequeños, estos se resuelven recursivamente, es decir, una y otra vez, y después las soluciones a estos se juntan para dar una solución final al problema principal. El modelo se lleva a cabo en tres pasos por cada grado de recursividad que pasa:

- ”Dividir” la cuestión a tratar en diversos subproblemas siendo estos más pequeños que el problema principal.
- ”Vencer” los subproblemas dándoles solución de manera recursiva. Hay que tener en cuenta que, si el subproblema es muy pequeño, simplemente se resuelve directamente sin hacer ninguna recursividad.
- ”Combina” las soluciones obtenidas de los subproblemas anteriores en la solución del problema raíz.

2.6 Algoritmos

El siguiente algoritmo tiene la funcionalidad de rotar una imagen 90° utilizando la estrategia de divide y vencerás. Como puede observarse, siempre se trabaja con cuadrantes, y cada uno de ellos se divide a su vez a nuevas matrices hasta llegar a las submatrices 2×2 . En ese instante se rotan los elementos y regresa dicha submatriz al proceso recursivo que a su vez rota los cuadrantes relacionados hasta llegar a la función original.

Algorithm 1 Rotar Divide y Vencerás

Input: Matriz **i**, tamaño de matriz **n**

Result: Imagen rotada 90°

Rotar(*i*, *n*)

```

if n==2 then
|   return [i[0][1] i[1][1] , i[0][0] i[1][0]]
else
|   t = Matriz(n,n)
|   t.segundoCuadrante = rotar(primerCuadrante(i),n/2)
|   t.tercerCuadrante = rotar(segundoCuadrante(i),n/2)
|   t.primerCuadrante = rotar(cuartoCuadrante(i),n/2)
|   t.cuartoCuadrante = rotar(tercerCuadrante(i),n/2)
return t
end

```

El algoritmo que se presenta a continuación, convierte cada una de las columnas originales de la imagen en columnas para crear la nueva imagen rotada. Dicho proceso se le conoce como fuerza bruta, ya que cada uno de los elementos de la fila le corresponde un elemento de la columna de la imagen original. Esto se logra con la función Transpuesta que de antemano se sabe que tiene complejidad lineal.

Algorithm 2 Rotar por fuerza bruta

Input: Matriz **m**, tamaño de matriz **n****Result:** Imagen rotada 90°**Rotar**(*i, n*) *t* = Matriz(*n, n*) **for** *i=0 to n do* *t.fila(n-i)* = Transpuesta(*m.columna(i)*)

3 Experimentación y resultados

El problema a resolver es el implementar una función que rote 90° una imagen de entrada con formato bmp, dicha imagen debe tener dimensiones de potencia de 2 tanto en alto como en ancho.

3.1 Función implementada por medio de la estrategia Divide y Vencerás

3.1.1 Análisis a Priori

Se determina que el tamaño del problema esta definido por el tamaño de los lados de la matriz a dividir, dado que ambos lados son iguales, entonces se tiene que $n = 2^k$ por tanto al dividir a la mitad a n se tiene que $\frac{n}{2} = 2^{k-1}$, véase Figura 2.

```

1 def rotar(i,n):
2     %% si la matriz es 2x2
3     if n==2:
4         t=np.array([[i[0][1],i[1][1]],[i[0][0],i[1][0]]])
5         return t
6     else:
7         t=np.zeros((n,n),dtype="uint8")
8         %% la imagen se divide en 4 cuadrantes y se trasladan
9         t[0:n//2,0:n//2]=rotar(i[0:n//2, n//2:],n//2)
10        t[n//2:,0:n//2]=rotar(i[0:n//2,0:n//2],n//2)
11        t[0:n//2, n//2:]=rotar(i[n//2:, n//2:],n//2)
12        t[n//2:, n//2:]=rotar(i[n//2:,0:n//2],n//2)
13        return t

```

Figura 2: Análisis por bloques de código de la función rotar().

Así en la Figura 2 se muestra la implementación de la estrategia Divide y vencerás para la rotación de la matriz, donde para cada bloque de código se muestra la obtención del orden de complejidad para el peor caso de este por medio del análisis a bloques del código, por lo que se determina que la función de recurrencia de este algoritmo es

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 2 \\ 4T\left(\frac{n}{2}\right) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Sea $a = 4$, $b = 2$ y $f(n) = \Theta(1) = C$ donde $a \geq 1$ y $b > 1$ se tiene que

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

por otro lado se tiene que

$$f(n) = C \in \mathcal{O}(n^{\log_2 4 - \epsilon}) = \mathcal{O}(n^{2-2})$$

Entonces por el Teorema Maestro, caso I)

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

3.1.2 Análisis a Posteriori

Para el análisis a posteriori se realizó una gráfica de ejecuciones vs n, dado que la imagen tiene dimensiones n^*n , véase Figura 3.

Así siguiendo la definición formal de $\Theta(n)$, y proponiendo los valores de: $C_1 = 20$, $C_2 = 40$ y $n_0 = 64$ puesto que se tiene una gráfica de una ecuación cuadrática. Dichos valores obedecen a la desigualdad de la definición formal de $\Theta(n)$ ya que se acotan los puntos tanto en la forma superior como inferior, además se cumplen a partir del punto de cruce propuesto. Por lo tanto la complejidad del algoritmo según el análisis a posteriori es de $\Theta(n^2)$.

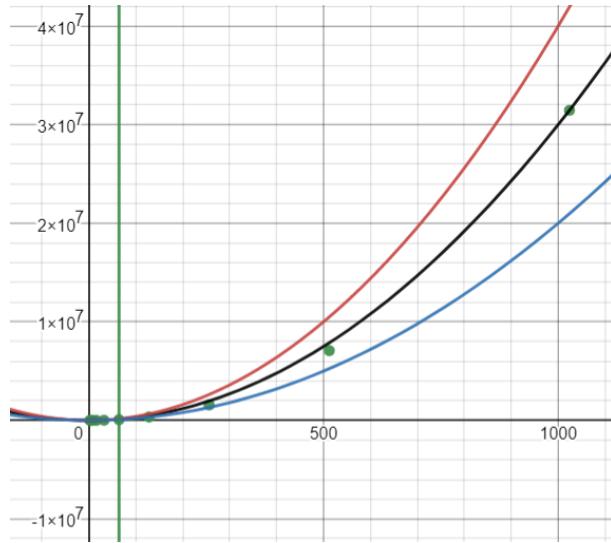


Figura 3: Gráfico de ejecuciones vs n para el algoritmo 1

3.2 Función implementada por medio de fuerza bruta

3.2.1 Análisis a Priori

En la siguiente figura observamos que el análisis por bloques del código nos indica que la función transpose convierte una fila en columna, dicha función

tiene una complejidad de $O(n)$, y dado que está en un ciclo for que se repite n veces, entonces esta tiene una complejidad de $O(n^2)$.

Podemos observar que python puede manejar matrices con índices negativos, esto sucede porque se le aplica la función modulo a los índices negativos regresando valores validos, facilitando nuestra función de rotación.

```

1 def rotar2(ii,i,n):
2     for y in range(n):————— Θ(n)
3         ii[-y,:]=np.transpose(i[:,y]) —— Θ(n) } Θ(n2)
4     return ii —————— Θ(1)

```

Figura 4: Análisis por bloques de código de la función rotar().

3.2.2 Análisis a Posteriori

Para el desarrollo del análisis a posteriori realizamos una gráfica de ejecuciones vs n , dado que nuestra imagen tiene dimensiones $n \times n$. A continuación se presenta la gráfica de dichos puntos.

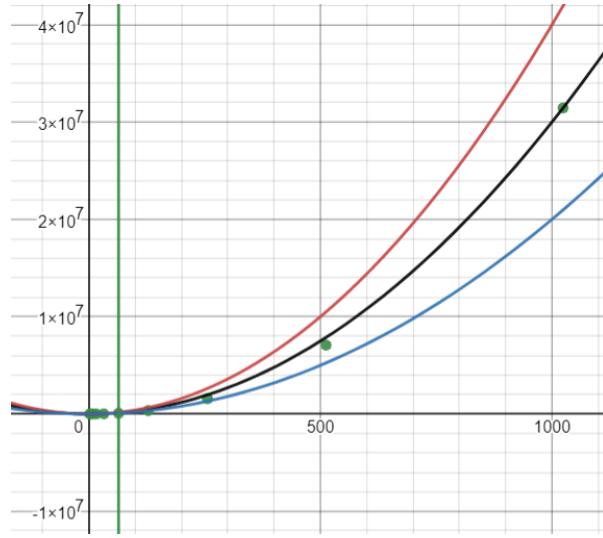


Figura 5: Gráfico de ejecuciones vs n para el algoritmo 2

Siguiendo la definición formal de $\Theta(n)$ proponemos los valores de: $C_1 = 2$, $C_2 = 4$ y $n_0 = 64$ puesto que se tiene una gráfica de una ecuación cuadrática. Dichos valores obedecen a la desigualdad de la definición formal de $\Theta(n)$ ya que se acotan los puntos tanto en la forma superior como inferior, además se

cumplen a partir del punto de cruce propuesto. Por lo tanto la complejidad del algoritmo según el análisis a posteriori es de $\Theta(n^2)$. En la siguiente figura se muestran los resultados al rotar la imagen 90° tanto para el algoritmo 1 como el 2.

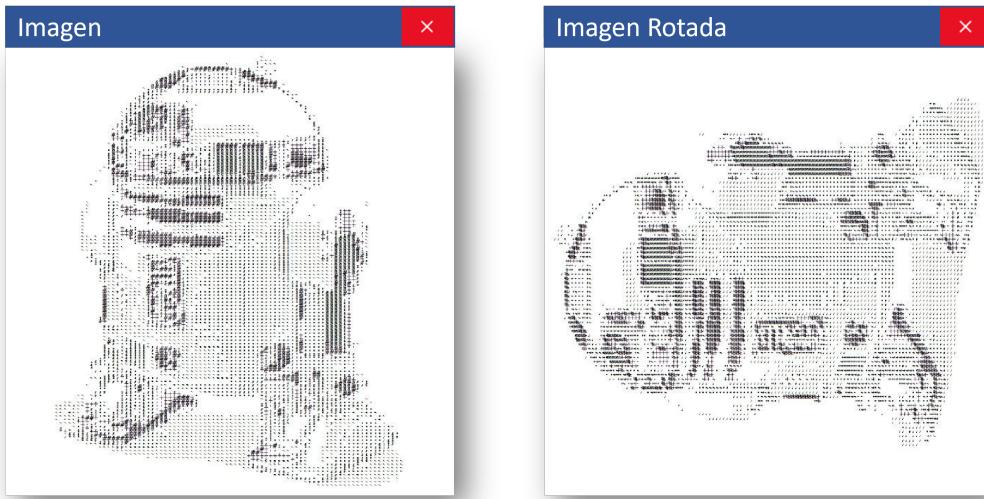


Figura 6: Resultado al rotar la imagen

4 Conclusiones

Luis Francisco Renteria Cedillo



Siguiendo el lema de "Dividir, vencer y combinar" logramos realizar esta práctica y comprender mejor el funcionamiento dicha estrategia, ya que es fácil reconocer en todo momento como la imagen es dividida recursivamente en cuatro cuadrantes hasta que la matriz llega a tamaño 2×2 . una vez llegado a este punto se rotan los valores y regresa al bloque anterior, a partir de aquí los valores son ahora submatrices de tamaño $n \times n$, y se realiza la misma operación hasta combinar tanto los cuadrantes como los canales RGB que componen la imagen.

A pesar de que en la experimentación nos arrojaron resultados que indican que se tiene la misma complejidad tanto algoritmo 1 como el 2, se logra un mayor comprendimiento de como funcionan ambas estrategias.

Denzel Omar Vazquez Perez

Es importante percibir que la utilización de la estrategia "*Divide y vencerás*" permite tener ventajas sobre cualquier otro tipo enfoque de solución, la separación en subproblemas ayuda a que el manejo de datos o del tamaño del problema sea digerible y se pueda lograr obtener buenos resultados a partir de la división de tareas y recursividad de estas.

Esto se logra observar en los resultados y experimentación de la práctica para rotar cada una de las matrices por las que se conforma la imagen, algo interesante es que pude ver que el algoritmo implementado con "*Divide y vencerás*" puede que no tenga ninguna ventaja ante el de fuerza bruta.

Sin embargo su complejidad espacial disminuye al no hacer uso de una matriz auxiliar dando solución al problema de forma paralela, teniendo mayor eficiencia frente a cualquier algoritmo clásico de rotación de matrices.

5 Anexo

5.1 Probar por sustitución hacia atrás que $T(n) \in \theta(n \log_2(n))$ dado que $T(n)$ esta definido por la ecuación de recursividad:

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

Sol.

Sea $n = 2^k \Rightarrow k = \log_2(n)$ se tiene que

$$T(2^k) = \begin{cases} \Theta(1) & \text{si } k = 0 \\ 2T(2^{k-1}) + C2^k & \text{si } k > 0 \end{cases}$$

Resolviendo por sustitución hacia atrás se tiene

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + C2^k \\ &= 2[2T(2^{k-2}) + C2^{k-1}] + C2^k \\ &= 4T(2^{k-2}) + 2C2^{k-1} + C2^k \\ &= 4[2T(2^{k-3}) + C2^{k-2}] + 2C2^{k-1} + C2^k \\ &= 8T(2^{k-3}) + 4C2^{k-2} + 2C2^{k-1} + C2^k \\ &= 8[2T(2^{k-4}) + C2^{k-3}] + 4C2^{k-2} + 2C2^{k-1} + C2^k \\ &= 16T(2^{k-4}) + 8C2^{k-3} + 4C2^{k-2} + 2C2^{k-1} + C2^k \\ &\vdots \\ (i) &= 2^i T(2^{k-i}) + 2^{i-1} C2^{k-i+1} + \cdots + C2^k \end{aligned}$$

para $K = i \Rightarrow K - i = 0$

$$\begin{aligned} &= 2^k T(2^0) + 2^{k-1} C2 + \cdots + C2^k \\ &= 2^k T(1) + 2^{k-1} C2 + \cdots + C2^k \\ &= 2^k [C + 2^{-1} C2 + \cdots + C] \end{aligned}$$

$$\text{donde } 2^{-1} C2 + \cdots + C = C \sum_{j=1}^k \frac{2^j}{2^j} = C \sum_{j=1}^k 1 = Ck \therefore$$

(1)

$$\begin{aligned}
&= 2^k[C + Ck] \\
&= C2^k + Ck2^k \\
&\text{Sustituyendo } n = 2^k \Rightarrow k = \log_2(n) \\
&= Cn + Cn \log_2(n) = C[n + n \log_2(n)] \\
&\therefore T(n) \in \Theta(n \log_2(n))
\end{aligned}$$

5.2 Utilizando decremento por uno pruebe que $T(n) \in \mathcal{O}(n^2)$ donde $T(n)$ es la ecuación de recursividad:

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

Sol.

Resolviendo por decremento por uno se tiene que:

$$f(n) = C(n + 1)$$

luego sabiendo que el orden de complejidad esta determinado por la siguiente expresión se tiene que

$$\begin{aligned}
T(n) &= \sum_{j=1}^n f(j) \\
&= \sum_{j=1}^n f(j + 1) \\
&= \sum_{j=1}^n C(j + 1) = \sum_{j=1}^n Cj + C1 \\
&\text{después} \\
&= C \sum_{j=1}^n j + C \sum_{j=1}^n 1 \\
&= C \frac{n(n + 1)}{2} + Cn = C \frac{n^2 + n}{2} + Cn \\
&\therefore T(n) \in \mathcal{O}(n^2)
\end{aligned}$$

5.3 Utilizando Teorema Maestro pruebe que $T(n) \in \Omega(n \log_2(n))$ donde $T(n)$ es la ecuación de recursividad:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Sol.

Sea $a = 2$, $b = 2$ y $f(n) = \Theta(n) = Cn$ donde $a \geq 1$ y $b > 1$ se tiene que

$$n^{\log_b a} = n^{\log_2 2} = n$$

por otro lado se tiene que

$$f(n) = Cn \in \Theta(n^{\log_2 2}) = \Theta(n)$$

Entonces por el Teorema Maestro, caso II)

$$T(n) = \Theta(n^{\log_b a} \log_2 n) = \Theta(n \log_2 n)$$

Finalmente, $T(n) \in \Theta(n \log_2 n)$ ssi $T(n) \in \mathcal{O}(n \log_2 n)$ y $T(n) \in \Omega(n \log_2 n)$.

5.4 Comprobar que la multiplicación usual tiene complejidad $\Theta(n^2)$

Sol.

Se propone la función multiplication la cual permite manejar la longitud o cifra de números como un arreglo, a continuación se muestra el análisis por bloques de código que determina que la complejidad del algoritmo $T(n) \in \Theta(n^2)$ ante este problema, véase la Figura 7.

```

1 multiplication(X[n], Y[n])
2     product = [0, ..., 2n]
3     for i=0 to n-1 ━━━━━━━━━━ Θ(n)
4         carry = 0
5         for j=0 to n-1 ━━━━━━ Θ(n)
6             product[i+j] += carry + (X[j]*Y[i]) } Θ(1)
7             carry = product[i+j]/10 } Θ(1)
8             product[i+j] = product[i+j] mod 10 } Θ(1)
9
10    return product

```

Figura 7: Función iterativa de la multiplicación usual

5.5 Comprobar que $T(n) \in \Theta(n^2)$ donde $T(n)$ es la ecuación de recursividad:

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$

Sol.

Sea $a = 4$, $b = 2$ y $f(n) = \Theta(n) = Cn$ donde $a \geq 1$ y $b > 1$ se tiene que

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

por otro lado se tiene que

$$f(n) = Cn \in \mathcal{O}(n^{\log_2 4 - \epsilon}) = \mathcal{O}(n^{2-1})$$

Entonces por el Teorema Maestro, caso I)

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

5.6 Comprobar que $T(n) \in \Theta(n^{\log_2 3})$ donde $T(n)$ es la ecuación de recursividad:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

Sol.

Sea $a = 3$, $b = 2$ y $f(n) = \Theta(n) = Cn$ donde $a \geq 1$ y $b > 1$ se tiene que

$$n^{\log_b a} = n^{\log_2 3}$$

por otro lado se tiene que

$$f(n) = Cn \in \mathcal{O}(n^{\log_2 3 - \epsilon})$$

Entonces por el Teorema Maestro, caso I)

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

5.7 Análisis a Priori de la implementación del algoritmo Kadane para encontrar el máximo subarreglo

Sol.

Para la implementación del algoritmo Kadane tiene como datos de entrada un arreglo A[] de tamaño n esto para almacenar los elementos con los que interactuara el algoritmo, posterior a eso se declaran 4 variables MaxSum que es la

valor máximo global de A[], AcomSum que es un valor máximo local de A[], inicio que guarda el valor del índice de inicio del subarreglo y final que guarda el valor del índice del fin del subarreglo.

Así para cada bloque de sentencia del algoritmo se muestra el orden de complejidad para el peor y mejor caso por medio del análisis de segmentos de código, véase Figura 8, donde se muestra que $T(n) \in \Theta(n)$.

```

1 kadane(A[0,...,n-1])
2     MaxSum = 0
3     AcomSum = 0
4     inicio = 0
5     final = 0
6     for(i=0; i<n; i++) —————— Θ(n)
7         AcomSum = AcomSum + A[i]
8         if(AcomSum > MaxSum)      Θ(1)
9             MaxSum = AcomSum
10            final = i
11            if(AcomSum < 0)          Θ(1)
12                AcomSum = 0
13                inicio = i+1
14    return (MaxSum, inicio, final)   Θ(n)

```

Figura 8: Algoritmo Kadane

6 Bibliografía

Brassard, G. (1997). *Fundamentos de Algoritmia*. España: Ed. Prentice Hall.

Cormen, E. A. (2022). *Introduction To Algorithms*, 3Rd Ed. Phi.

Sánchez, P. J. I. (2006). *Análisis y diseño de algoritmos: un enfoque teórico y práctico*. Servicio de Publicaciones y Divulgación Científica de la UMA.