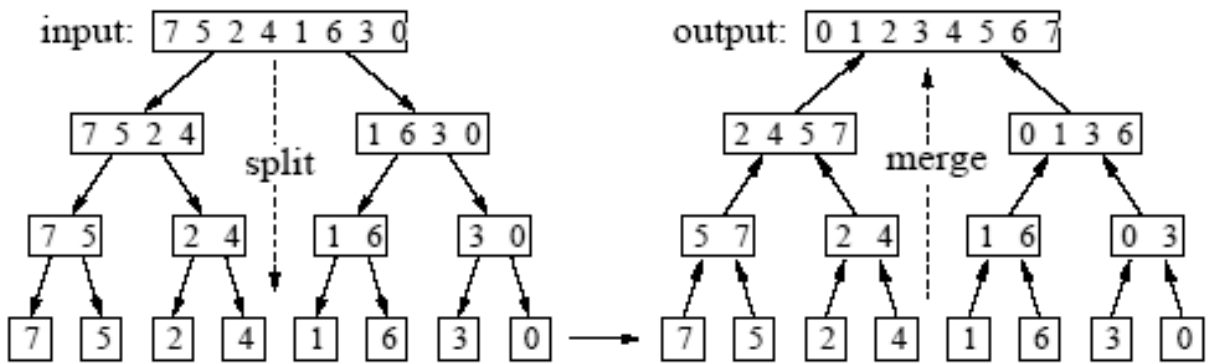


Divide y Vencerás

Estructuras de Datos II
Erwin Meza Vega

MergeSort



```

MergeSort(array A, int p, int r) {
    if (p < r) {
        q = (p + r)/2
        MergeSort(A, p, q)
        MergeSort(A, q+1, r)
        Merge(A, p, q, r)
    }
}

```

// we have at least 2 items

// sort A[p..q]

// sort A[q+1..r]

// merge everything together

```

Merge(array A, int p, int q, int r) {
    array B[p..r]
    i = k = p
    j = q+1
    while (i <= q and j <= r) {
        if (A[i] <= A[j]) B[k++] = A[i++]
        else B[k++] = A[j++]
    }
    while (i <= q) B[k++] = A[i++]
    while (j <= r) B[k++] = A[j++]
    for i = p to r do A[i] = B[i]
}

```

// merges A[p..q] with A[q+1..r]

// initialize pointers

// while both subarrays are nonempty

// copy from left subarray

// copy from right subarray

// copy any leftover to B

// copy B back to A

MergeSort es un algoritmo de ordenamiento estable

De los dos ciclos en el procedimiento Merge, sólo uno se ejecuta

MergeSort

- Mergesort(A, p, r): Ordena el arreglo $A[p \dots r]$
 - Si $r=p$, sólo hay un elemento para ordenar: retornar
 - Si $p < r$, hay al menos dos elementos
 - $q = (p+r)/2$
 - Se divide el arreglo en $A[p \dots q]$ y $A[q+1, r]$ y se invoca MergeSort en cada arreglo
 - Se invoca un procedimiento para unir los dos arreglos

MergeSort

- Merge(A, p, q, r)
 - Asume que $A[p \dots q]$ y $A[q + 1 \dots r]$ están ordenados
 - Se copian los elementos a un arreglo $B[p \dots r]$
 - Se mueve el elemento más pequeño a $B[k]$, y se incrementa el índice respectivo (i o j)
 - Cuando se terminan los elementos en un arreglo, se copian los elementos faltantes del otro arreglo
 - Finalmente se copia $B[p \dots r]$ en $A[p \dots r]$

Análisis de MergeSort

- Tiempo total de ejecución

$$T(n) = \begin{cases} 1, & n=1 \\ T(n/2)+T(n/2)+T_{\text{merge}}, & n>1 \end{cases}$$

- Recurrencia: Función que se encuentra definida en términos de ella misma.

$$T(n) = \begin{cases} 1, & n=1 \\ 2 T(n/2)+ n, & n>1 \end{cases}$$

Números de Fibonacci

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$

- Se puede utilizar una solución recursiva
 - Existe solapamiento: repetición en los cálculos
- Solución: programación dinámica

Programación Dinámica

- Se debe tener en cuenta que:
 - La solución del problema ha de ser alcanzada a través de una secuencia de decisiones
 - Toda secuencia de decisiones debe cumplir el principio de óptimo (Bellman): “En una secuencia de decisiones óptima toda subsecuencia debe ser también óptima”

Fibonacci

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

```
PROCEDURE FibRec(n: CARDINAL): CARDINAL;  
BEGIN  
    IF n <= 1 THEN RETURN 1  
    ELSE  
        RETURN FibRec(n-1) + FibRec(n-2)  
    END  
END FibRec;
```

Se puede solucionar la recursividad construyendo una tabla:

$Fib(0)$	$Fib(1)$	$Fib(2)$...	$Fib(n)$
----------	----------	----------	-----	----------

Fibonacci



```
TYPE TABLA = ARRAY [0..n] OF CARDINAL
PROCEDURE FibIter(VAR T:TABLA;n:CARDINAL):CARDINAL;
    VAR i:CARDINAL;
BEGIN
    IF n<=1 THEN RETURN 1
    ELSE
        T[0]:=1;
        T[1]:=1;
        FOR i:=2 TO n DO
            T[i]:=T[i-1]+T[i-2]
        END;
        RETURN T[n]
    END
END FibIter;
```

```
PROCEDURE FibIter2(n: CARDINAL):CARDINAL;
    VAR i,suma,x,y:CARDINAL; (* x e y son los 2 ultimos terminos *)
BEGIN
    IF n<=1 THEN RETURN 1
    ELSE
        x:=1; y:=1;
        FOR i:=2 TO n DO
            suma:=x+y; y:=x; x:=suma;
        END;
        RETURN suma
    END
END FibIter2;
```

Coeficientes Binomiales

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n, \quad \binom{n}{0} = \binom{n}{n} = 1.$$

El algoritmo recursivo que los calcula resulta ser de complejidad exponencial por la repetición de los cálculos que realiza. No obstante, es posible diseñar un algoritmo con un tiempo de ejecución de orden $O(nk)$ basado en la idea del Triángulo de Pascal. Para ello es necesario la creación de una tabla bidimensional en la que ir almacenando los valores intermedios que se utilizan posteriormente:

	0	1	2	3	...	$k-1$	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...		
...	
$n-1$						$C(n-1,k-1) +$	$C(n-1,k)$
n							 $C(n,k)$

Coeficientes Binomiales

```
PROCEDURE CoefIter(n,k: CARDINAL):CARDINAL;  
    VAR i,j: CARDINAL;  
        C: TABLA;  
BEGIN  
  
    FOR i:=0 TO n DO C[i,0]:=1 END;  
    FOR i:=1 TO n DO C[i,1]:=i END;  
    FOR i:=2 TO k DO C[i,i]:=1 END;  
    FOR i:=3 TO n DO  
        FOR j:=2 TO i-1 DO  
            IF j<=k THEN  
                C[i,j]:=C[i-1,j-1]+C[i-1,j]  
            END  
        END  
    END;  
    RETURN C[n,k]  
END CoefIter.
```

Transformación de Cadenas

- Sean u y v dos cadenas de caracteres. Se desea transformar u en v con el mínimo número de operaciones básicas del tipo siguiente: eliminar un carácter, añadir un carácter, y cambiar un carácter. Por ejemplo, podemos pasar de *abbac* a *abcbc* en tres pasos:
- *abbac* ? *abac* (eliminamos b en la posición 3)
- ? *ababc* (añadimos b en la posición 4)
- ? *abcbc* (cambiamos a en la posición 3 por c)
- Sin embargo, esta transformación no es óptima. Lo que queremos en este caso es diseñar un algoritmo que calcule el número mínimo de operaciones, de esos tres tipos, necesarias para transformar u en v y cuáles son esas operaciones, estudiando su complejidad en función de las longitudes de u y v .