



Algoritmos y Estructuras de Datos II

Clase 7

Carreras:

Licenciatura en Informática

Ingeniería en Informática

2024

Unidad III

Técnicas de diseño de algoritmos

Técnicas de diseño de algoritmos

- Las técnicas de diseño de algoritmos, también llamadas estrategias o paradigmas son prácticas que enfocan la resolución de un problema de manera algorítmica, de manera que puede aplicarse a una gran variedad de problemas en el área de ciencias de la computación.

**Técnicas de
diseño de
algoritmos**

```
graph LR; A[Técnicas de diseño de algoritmos] --- B[Dividir para Conquistar (Divide & Conquer).]; A --- C[Técnica voraz (Algoritmos greedy).]; A --- D[Programación dinámica.]; A --- E[Vuelta atrás (Backtracking).]; A --- F[Ramificación y poda (Branch and Bound).]; A --- G[Algoritmos Probabilistas.]; A --- H[Algoritmos Heurísticos];
```

Dividir para Conquistar
(Divide & Conquer).

Técnica voraz
(Algoritmos greedy).

Programación dinámica.

Vuelta atrás
(Backtracking).

Ramificación y poda
(Branch and Bound).

Algoritmos Probabilistas.

Algoritmos Heurísticos

Técnicas de diseño de algoritmos

Dividir para Conquistar (Divide & Conquer)

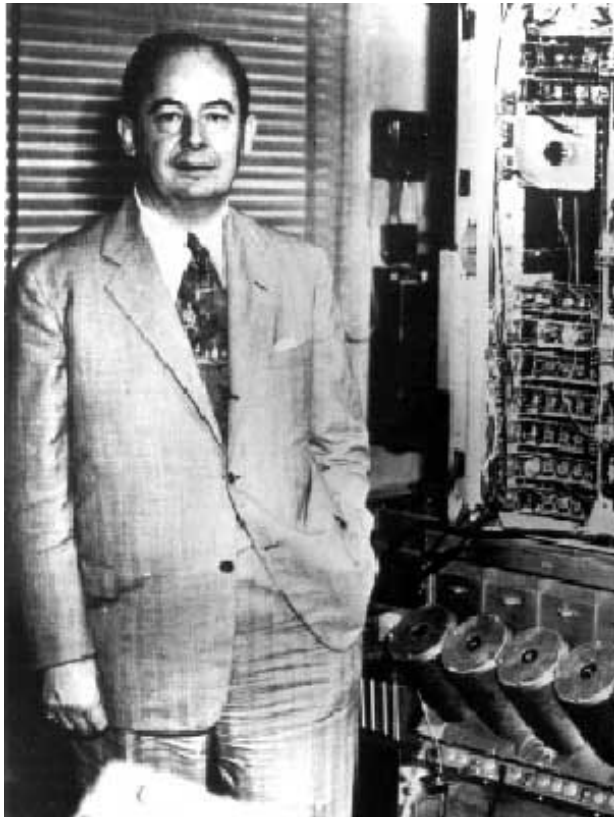


Proverbio Latín: ***Divide et vinctes*** o ***Divide et impera***

Julio César: "**Divide y vencerás**"

Divide & Conquer

John von Neumann (1903 - 1957)



*John von Neumann y la
ENIAC (1945)*

Fue uno de los más grandes matemáticos del siglo XX.

Nacido en Budapest y radicado en Estados Unidos, realizó contribuciones importantes en física cuántica, análisis funcional, teoría de conjuntos, ciencias de la computación, economía, análisis numérico, cibernética, hidrodinámica (de explosiones), estadística y muchos otros campos.

En 1945 inventó el conocido algoritmo MergeSort aplicando por primera vez la metodología Divide & Conquer

Decrease & Conquer

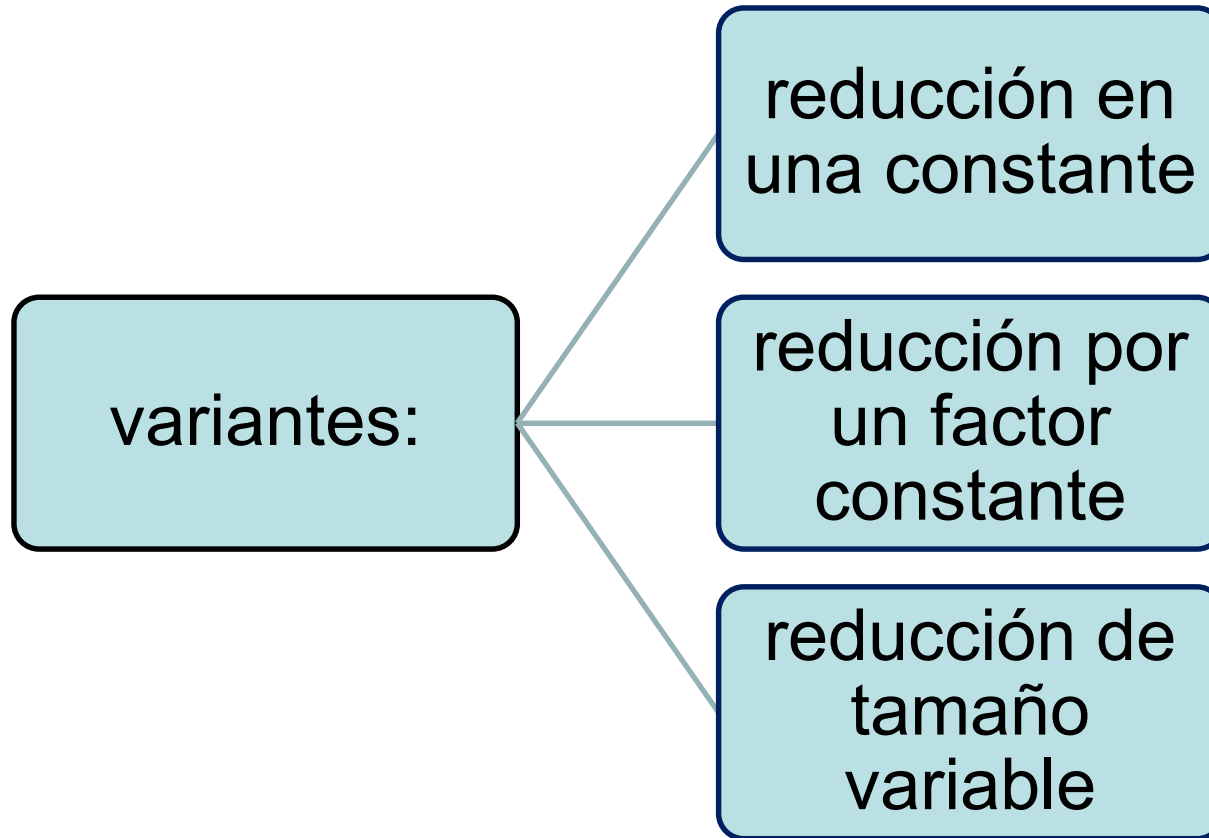
La técnica decrease & conquer (reducción y conquista)

- Se basa en explotar la relación entre una solución entre una dada instancia de un problema y la solución de una instancia más pequeña.

Implementación

- Es generalmente recursiva y top down.
- También puede resolverse bottom up mediante una iteración.

Decrease & Conquer



Decrease & Conquer

Reducción en una constante:

El tamaño se reduce en la misma constante en cada iteración o llamada recursiva. Generalmente la constante es 1.

Ejemplo:

Calcular a^n donde $a \neq 0$ y n es un entero positivo.

$$a^n = a^{n-1}$$

$$a^0 = 1 \quad f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

Decrease & Conquer

Reducción en una constante:

Ejemplo: **Torres de Hanoi**

```
Algoritmo Hanoi(n,i,j)
    SI  $n > 0$  ENTONCES
        Hanoi (n-1 , i , 6-i-j)
        Mover un disco de i  $\rightarrow$  j
        Hanoi (n-1 , 6-i-j , j)
    Fin.
```

Ejemplo: **Ordenación por inserción directa**

En el paso $i = 2, \dots, n$, se inserta el próximo $= A(i)$ donde corresponda, controlando entre los anteriores desde $A(i-1)$ hasta $A(1)$ que ya están ordenados

Decrease & Conquer

Reducción en una constante:

Ejemplo: **Generar permutaciones de números**

Idea del algoritmo: Dados n números $\{1, 2, \dots, n\}$, reducir el problema de tamaño n en 1 y generar las $(n-1)!$ permutaciones de $n-1$ números y luego insertar el número n .

Si $n=3$

Iniciar en: 1

Agregar 2: 12 21

Agregar 3: 123 132 312

Nota: Hay algoritmos mas eficientes como el de JohnsonTrotter, pero su crecimiento es también proporcional al numero de permutaciones, $\in \Theta(n!)$

Decrease & Conquer

Reducción en una constante:

Ejemplo: **Generar subconjuntos de un conjunto dado**

Dado un conjunto de n elementos $A_n = \{a_1, a_2, \dots, a_n\}$ se pueden generar 2^n subconjuntos posibles.

Idea: Los subconjuntos de A se pueden generar a partir de los subconjuntos de $A_{n-1} = \{a_1, a_2, \dots, a_{n-1}\}$ y además todos los conjuntos que se forman agregándoles a cada uno a_n .

Sea:

$n=0$ \emptyset

$n=1$ $\emptyset \{a_1\}$

$n=2$ $\emptyset \{a_1\} \{a_2\} \{a_1, a_2\}$

$n=3$ $\emptyset \{a_1\} \{a_2\} \{a_1, a_2\} \{a_3\} \{a_1, a_3\} \{a_2, a_3\} \{a_1, a_2, a_3\}$

Decrease & Conquer

Reducción en una constante:

Ejemplo: **Generar códigos binarios**

Dado un valor n generar una lista con los 2^n códigos binarios posibles de n bits.

Ej. Si $n=3$, $L = 000\ 001\ 010\ 011\ 100\ 101\ 110\ 111$.

Si se genera la lista de los llamados *binary reflected Gray code* cada código difiere del anterior en solo 1 bit.

Ej. Si $n=3$, $L = 000\ 001\ 011\ 010\ 110\ 111\ 101\ 100$.

Frank Gray, investigador de AT&T Bell Laboratories, los usó en 1940 para minimizar los errores en la transmisión de señales digitales. 13

Decrease & Conquer

Reducción en una constante:

Ejemplo: **Generar Binary Reflected Gray Code**

ALGORITMO BRGC(n)

Entrada: n, entero positivo

Salida: L, lista de los códigos generados

SI $n = 1$ ENTONCES $L \leftarrow "0, 1"$

SINO $L1 \leftarrow \text{BRGC}(n - 1)$

copiar la lista L1 a L2 en orden inverso

Agregar 0 en el frente de cada tira de bit en L1

Agregar 1 en el frente de cada tira de bit en L2

$L \leftarrow \text{concatenar } L1 \text{ y } L2$

Retorna L

Decrease & Conquer

Reducción en una constante:

Ejemplo: **Generar binary reflected Gray code**

Aplicar el ALGORITMO BRGC(n)

$n = 1$ $L = "0, 1"$

$n=2$ $L1 = "0, 1"$ $L2 = "1, 0"$
 $L1 = "00, 01"$ $L2 = "11, 10"$
 $L = "00, 01, 11, 10"$

$n=3$ $L1 = "00, 01, 11, 10"$ $L2 = "10, 11, 01, 00"$
 $L1 = "000, 001, 011, 010"$ $L2 = "110, 111, 101, 100"$
 $L1 = "000, 001, 011, 010, 110, 111, 101, 100"$

Decrease & Conquer

Reducción por un factor *constante*:

Se hace la reducción de la instancia del problema por el ***mismo factor constante*** en cada iteración o llamada recursiva del algoritmo, en la muchas de las aplicaciones el factor es dos.

Ejemplo: Calcular a^n donde $a \neq 0$ y n es un entero positivo.

$$a^n = \left\{ \begin{array}{ll} 1 & \text{si } n = 0 \\ (a^{n/2})^2 & \text{si } n \text{ es par} \\ (a^{(n-1)/2})^2 \cdot a & \text{si } n \text{ es impar} \end{array} \right\}$$

Decrease & Conquer

Reducción por un factor constante:

Otros ejemplos:

- Búsqueda Binaria en un arreglo ordenado de n datos
- Determinación de la Moneda Falsa en un conjunto de n monedas
- Multiplicación a la rusa de dos números enteros
- Problema de Josefo

Decrease & Conquer

Reducción de tamaño variable:

En este caso el tamaño de la reducción del problema tiene **valores variables** entre una iteración y otra o entre las llamadas recursivas del mismo algoritmo.

Ejemplo: Algoritmo de Euclides para el máximo común divisor de 2 números:

$$\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$$

$$\text{gcd}(m, 0) = m$$

Decrease & Conquer

Reducción de tamaño variable:

Otros ejemplos:

- Partición en un vector
- Problema de selección, QuickSelect
- Obtener la Mediana
- Búsqueda por interpolación en un arreglo ordenado
- Búsqueda e inserción en un árbol binario

Divide & Conquer

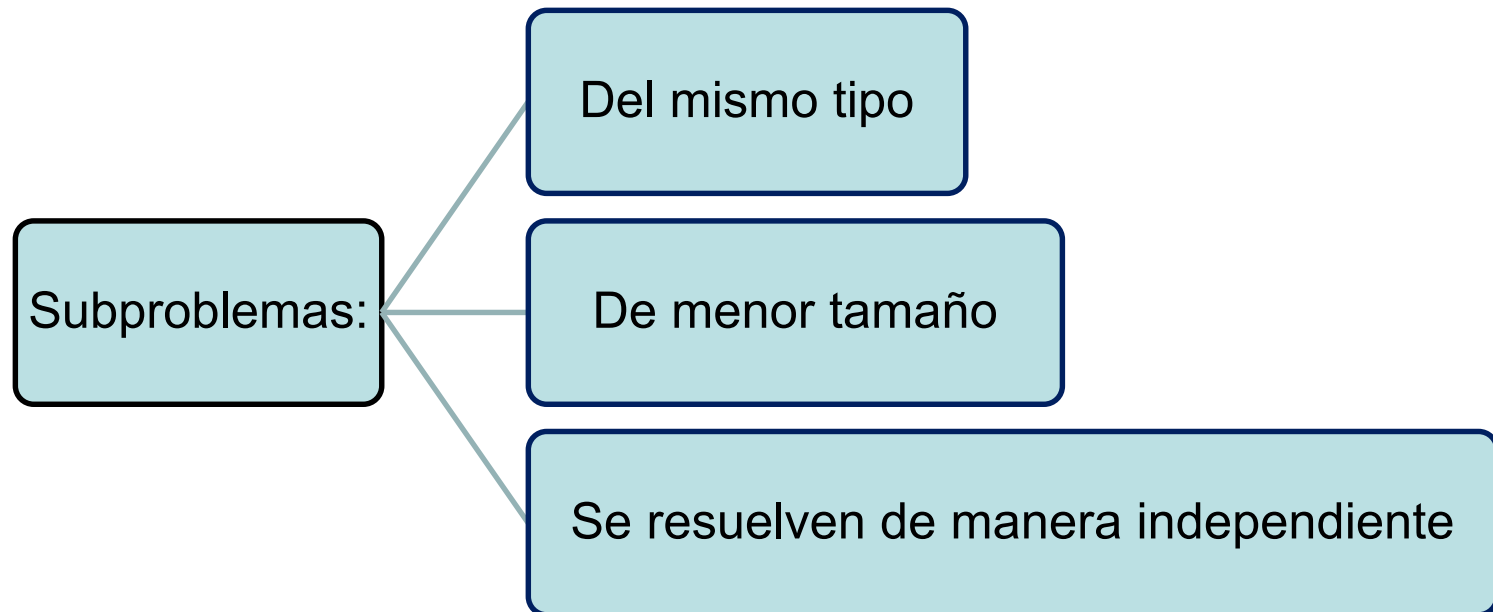
- El término **Divide & Conquer** en su acepción más amplia es algo más que una técnica de diseño de algoritmos.
- Suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.

Divide & Conquer

- Esta técnica no se puede aplicar a todo tipo de problemas. En algunos casos se puede aplicar pero no resulta en soluciones óptimas, en otros casos sí.
- La estrategia **Divide & Conquer** es una metodología *top-down* ya que resuelve los problemas de una manera *descendente*.

Divide & Conquer

- Divide & Conquer es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la descomposición y solución de **subproblemas**:



Divide & Conquer

- Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica de descomposición hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente.

Subproblemas
triviales:

Se resuelven de manera directa

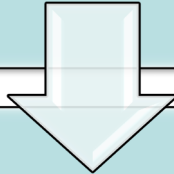
Divide & Conquer

- Divide & Conquer naturalmente sugiere el uso de la *recursión* en las implementaciones de estos algoritmos, aunque puede aplicarse también de manera *iterativa*.
- Las soluciones obtenidas en los pasos anteriores se *combinan* para obtener la solución del problema original.

Divide & Conquer

La **resolución de un problema** mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. Descomponer en subproblemas



2. Resolver los subproblemas.



3. Combinar las soluciones.

Divide & Conquer

1. En primer lugar debe plantearse el problema de forma que pueda ser *descompuesto* en k subproblemas del *mismo tipo*, pero de *menor tamaño*.

Si el tamaño de la entrada es n , hay que conseguir dividir el problema en k subproblemas ($1 \leq k \leq n$), cada uno con una entrada de tamaño n_k y donde $0 \leq n_k < n$.

A esta tarea se le conoce como *división*.

Divide & Conquer

2. En segundo lugar deben *resolve* independientemente todos los subproblemas, directamente si son elementales o bien de forma recursiva.

El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema garantiza la convergencia hacia los casos elementales, también denominados casos *base*.

Divide & Conquer

3. Por último, se debe encontrar la forma de *combinar* las soluciones obtenidas en el paso anterior 2 para construir la solución del problema original.

Divide & Conquer

El funcionamiento de los algoritmos que siguen la técnica de Divide & Conquer detallada anteriormente se refleja en el esquema general:

ALGORITMO : D&C

ENTRADA: x (problema)

SALIDA: y (solución)

P1. **SI** x es “suficientemente chico” **ENTONCES**
 aplicar un algoritmo básico para x

SINO

Descomponer x en instancias mas chicas (x_1, x_2, \dots, x_k)

PARA i=1,k **HACER**

$y_i \leftarrow \text{D\&C}(x_i)$

Combinar los (y_1, y_2, \dots, y_k) para obtener la solucion y.

P2. **FIN**

Divide & Conquer

La **eficiencia** del método Divide & Conquer descansa en como se hace la *descomposición*, en como se resuelven los subproblemas y en como se *combinan* sus soluciones.

Como a menudo los subproblemas generados son instancias disjuntas mas chicas del problema original, pueden ser resueltas usando Divide & Conquer recursivamente.

1. **Divide** : en sub problemas mas chicos
2. **Resolver** : los sub problemas de forma independiente
3. **Conquer** : la solución del problema original se forma de la solución de los sub problemas.

Divide & Conquer

Para que la estrategia sea *eficiente* se requiere que:

1. Debe ser posible descomponer una instancia en subinstancias disjuntas y recombinar la solución en forma eficiente.
2. Se debe evitar seguir avanzando recursivamente cuando el tamaño de los casos no lo justifique. Si el tamaño del problema es pequeño, este debe resolverse aplicando un método directo. La decisión de cuando usar un subalgoritmo básico en lugar de hacer llamadas recursivas se debe analizar con cuidado.
3. Las subinstancias deben ser del mismo tamaño, tan balanceadas como se pueda.

Divide & Conquer

El número de subinstancias k es generalmente chico e independiente de la instancia x . En especial cuando se tiene $k=1$, se llama ***Simplificación***.

La ventaja de los algoritmos de Simplificación es que:

- Consiguen reducir el tamaño del problema en cada paso:
 - ***tiempos de ejecución suelen ser muy buenos*** (normalmente de orden logarítmico o lineal).
- Se puede eliminar fácilmente la recursión mediante el uso de iteración, aunque en algunos casos esto vaya en detrimento de la legibilidad del código resultante:
 - ***menor complejidad espacial*** al no utilizar la pila de recursión. Se consigue un algoritmo más rápido y es posible ahorrar una cantidad considerable de memoria.

Divide & Conquer

Algunas consideraciones:

- No siempre los algoritmos D&C son los mas eficientes, incluso pueden ser tan costosos como los algoritmos por la fuerza bruta.
- Divide & Conquer es la técnica ideal para adaptar a cálculos en paralelo , en los cuales cada subproblema se puede resolver simultáneamente en su propio procesador.

Divide & Conquer

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de *Divide & Conquer* van a heredar las **ventajas** e **inconvenientes** que la recursión plantea:

- Por un lado el diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
- Sin embargo, los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Divide & Conquer

Tiempo de Ejecución

El análisis del tiempo de ejecución de los algoritmos divide & conquer se puede hacer en forma sencilla:

- Un problema de tamaño n se divide en a subproblemas de tamaño n/b cada uno.
- El costo de descomponer combinar las soluciones se puede considerar una función de n .
- El tiempo total requerido por un algoritmo D&C puede ser expresado con la siguiente ecuación de recurrencia:

$$\begin{aligned} T(n) &= a T(n/b) + c n^k & \text{si } b \leq n \\ T(n) &= c n^k & \text{si } 1 \leq n < b \end{aligned}$$

Divide & Conquer

Tiempo de Ejecución

Teorema (*Master Theorem*)

La solución de la ecuación:

$$T(n) = a T(n/b) + c n^k$$

donde $c \geq 0$ real, $a \geq 1$ real, $b \geq 2$ entero y $k \geq 0$ entero, está dada por :

- $T(n) \in \Theta(n^{\log_b a})$ si $a > b^k$
- $T(n) \in \Theta(n^k \log_b n)$ si $a = b^k$
- $T(n) \in \Theta(n^k)$ si $a < b^k$

Estos resultados se pueden aplicar a las otras notaciones O y Ω .

Divide & Conquer

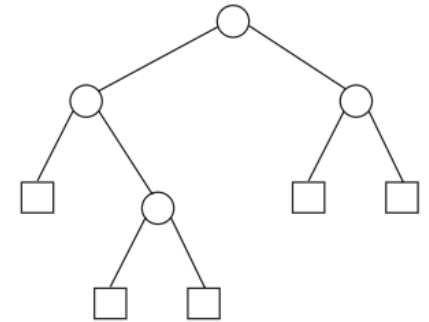
Ejemplos de aplicación

- Ordenación por método de Mezcla o Fusión
- Ordenación por método QuickSort
- Multiplicación de Matrices
- Multiplicación de Enteros Largos
- Búsqueda del máximo y del mínimo
- Encontrar puntos máximos en un espacio bidimensional
- El problema del par más cercano
- Transformada rápida de Fourier
- Exponenciación
- Criptografía

Divide & Conquer

Ejemplos de aplicación

- Algoritmos en Arboles binarios, AB(item):



Ejemplo:

Funcion **Altura** (T) : AB \rightarrow entero ≥ 0

Si ESABVACIO (T) OR eshoja (T) entonces

Retorna 0

sino

Retorna 1 + MAXIMO (**Altura**(IZQUIERDO(T)),
Altura(DERECHO(T)))

Fin

Divide & Conquer

Algoritmo de Mezcla por Fusión

Merge-sort

Propuesto en 1945 por John Von Neumann

Idea:

- Si el arreglo tiene 0 o 1 elemento está ordenado.
- Sino Dividir el arreglo en 2 partes.
- Ordenar cada parte recursivamente
- Mezclar las 2 partes ya ordenadas para conseguir el arreglo ordenado.

Divide & Conquer

Algoritmo de Mezcla por Fusión

Algoritmo **Merge**(a, izq, der)

Entrada: a: arreglo de n items

izq, der: números enteros positivos correspondientes a los índices entre los cuales se ordenan las componentes del arreglo

Salida: a: arreglo de n items ORDENADO entre izq y der

Si der > izq entonces

$m \leftarrow (der + izq) / 2$

Merge(a, izq, m)

Merge(a, m+1, der)

 Fusionar(a, izq, m, der)

Fin

Se invoca con: **Merge** (a, 1, n)

Divide & Conquer

Algoritmo de Mezcla por Fusión

Algoritmo Fusionar (a, izq, m, der)

// Fusiona la parte: a(izq, m) con: a(m+1, der)

Auxiliar b(max)

Para i=izq, m hacer

$b(i) \leftarrow a(i)$

Para j=m, der-1 hacer

$b(\text{der}+m-j) \leftarrow a(j+1)$

i ← izq

j ← der

Para k=izq, der

 Si $b(i) < b(j)$ entonces

$a(k) \leftarrow b(i)$

$i \leftarrow i+1$

 sino

$a(k) \leftarrow b(j)$

$j \leftarrow j-1$

Fin

Ejemplo Merge

Merge(a,1,8)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,1,4)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,1,2)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,1,1)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,2,2)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Fusionar(a,1,1,2)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,3,4)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,3,3)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,4,4)

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Fusionar(a,3,3,4)

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Fusionar(a,1,2,4)

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Merge(a,5,8)

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

3	5	2	6	4	1	3	7
---	---	---	---	---	---	---	---

2	3	5	6	4	1	3	7
---	---	---	---	---	---	---	---

2	3	5	6	4	1	3	7
---	---	---	---	---	---	---	---

⋮

Fusionar(a,5,6,8)

2	3	5	6	1	3	4	7
---	---	---	---	---	---	---	---

Fusionar(a,1,4,8)

1	2	3	3	4	5	6	7
---	---	---	---	---	---	---	---

Divide & Conquer

Algoritmo de Mezcla por Fusión

Merge Sort



Problem Size: 20 · 30 · 40 · 50 Magnification: 1x · 2x · 3x

Algorithm: Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3



<https://www.toptal.com/developers/sorting-algorithms/merge-sort>

Divide & Conquer

Algoritmo de Mezcla por Fusión

Costos:

- **Fusionar** $\in \Theta(n)$
- **Merge** $\in ?$

$$\begin{aligned} T(n) &= a T(n/b) + c n^k && \text{(teorema)} \\ T(n) &= 2T(n/2) + c n^l && \text{si } n > 0 \end{aligned}$$

Según el teorema:

$$\text{si: } a = b^k \quad T(n) \in \Theta(n^k \log_b n)$$

$$\text{como: } 2 = 2^1 \quad T(n) \in \Theta(n^1 \log_2 n)$$

$$\text{Merge} \in \Theta(n \log_2 n)$$

Divide & Conquer

Algoritmo de Mezcla por Fusión

Costo:

