




Ordenamiento por mezcla


Merge Sort

El enfoque de divide y vencerás

- Muchos algoritmos útiles tienen una estructura recursiva: para resolver un problema dado, se llaman a sí mismos de forma recursiva una o más veces para tratar con subproblemas estrechamente relacionados.
- Estos algoritmos suelen seguir un enfoque de divide y vencerás: dividir el problema en varios subproblemas que son similares al problema original pero de menor tamaño, resuelve los subproblemas de forma recursiva y luego combina estas soluciones para crear una solución al problema original.

- 
- El paradigma divide y vencerás implica tres pasos en cada nivel de la recursividad:
 - **Divide** el problema en varios subproblemas que sean instancias más pequeñas del el mismo problema.
 - **Conquista** los subproblemas resolviéndolos de forma recursiva. Si los tamaños de los subproblemas son lo suficientemente pequeños, sin embargo, simplemente resuelve los subproblemas de una manera sencilla.
 - **Combina** las soluciones de los subproblemas en la solución del problema original.

- El algoritmo de ordenación por mezcla sigue de cerca el paradigma de divide y vencerás.
Intuitivamente, funciona de la siguiente manera:
- **Dividir:** Divide la secuencia de n elementos para clasificar en dos subsecuencias de $n/2$ elementos cada una.
- **Conquistar:** Ordena las dos subsecuencias de forma recursiva utilizando merge sort.
- **Combinar:** Combina las dos subsecuencias ordenadas para producir la respuesta ordenada.

- 
- La recursividad "toca fondo" cuando la secuencia que se va a ordenar tiene una longitud de 1, en tal caso que no hay trabajo por hacer, ya que cada secuencia de longitud 1 ya está en orden ordenado.

5 2 4 7 1 3 2 6

initial sequence



len = 8
mitad = $8/2=4$

5 2 4 7

initial s

1 3 2 6

equene



len = 4
mitad = 2

5 2

4 7

1 3

2 6

len=2
mitad=1



5 2

4 7

1 3

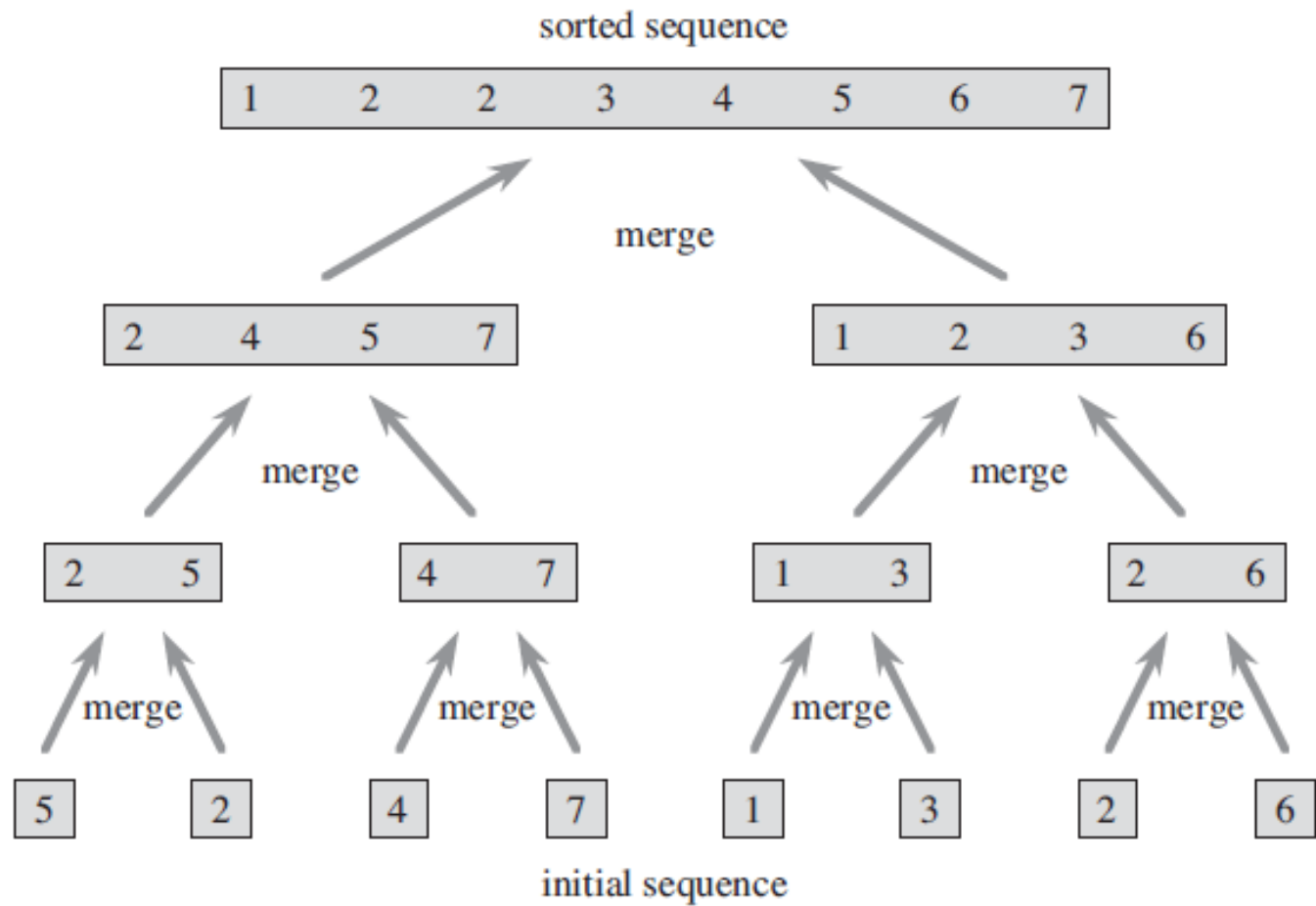
2

6

Si $len \leq 1$ entonces se termina la recursividad

- La operación clave del algoritmo de ordenamiento por mezcla es la combinación de dos secuencias en el paso "combinar".
- Las combinamos llamando a un procedimiento auxiliar MERGE(A , p , q , r), donde:
 - A es un arreglo y p , q y r son índices en el arreglo tal que $p \leq q < r$.
 - El procedimiento asume que los subarreglos $A[p \dots q]$ y $A[q+1 \dots r]$ están ordenados.
 - Los combina para formar un único subarreglo ordenado que reemplaza al subarreglo actual $A[p \dots r]$.

Ejemplo



Pseudocódigo – Descripción general

- Volviendo a nuestro juego de cartas, supongamos que tenemos dos pilas de cartas boca arriba sobre una mesa.
- Cada pila esta ordenada, con las tarjetas más pequeñas en la parte superior.
- Deseamos combinar las dos pilas en una sola pila de salida ordenada, que debe estar boca abajo sobre la mesa.
- Nuestro paso básico consiste de elegir la más pequeña de las dos cartas en la parte superior de las pilas boca arriba, quitarla de su pila (que expone una nueva carta superior), y colocando esta carta boca abajo en la pila de salida.
- Repetimos este paso hasta que una pila de entrada esté vacía, momento en el que simplemente tomamos la pila de entrada restante y la colocamos boca abajo en la pila de salida.

- Implementando la idea anterior, pero con un giro adicional que evita tener que comprobar si alguna pila está vacía en cada paso básico.
- Colocamos en la parte inferior de cada pila una carta centinela, que contiene un valor especial que usamos para simplificar nuestro código.
- Aquí, usamos ∞ como valor centinela, de modo que siempre que una carta con ∞ esté expuesta, no puede ser la carta más pequeña a menos que ambas pilas expongan sus cartas de centinela.
- Pero una vez que eso sucede, todas las cartas no centinelas ya se han colocado en la pila de salida. Ya que sabemos de antemano que exactamente $r - p + 1$ cartas se colocarán en la pila de salida, podemos detenernos una vez que han realizado tantos pasos básicos.

Pseudocódigo Función Merge

MERGE(A, p, q, r)

1 2 3 4

5 6 7 8

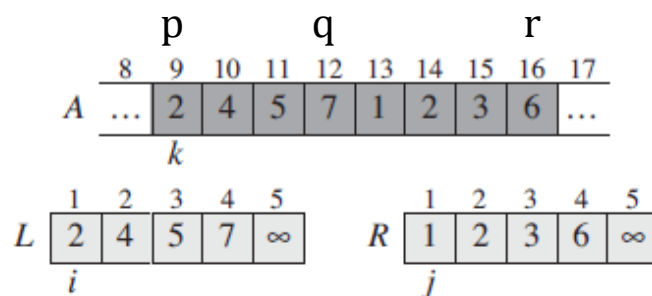
2 4 5 7

1 2 3 6

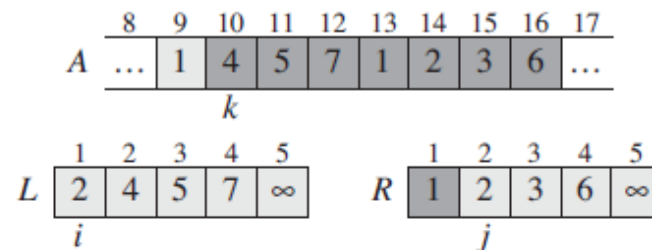
```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

$p=1$
 $q=4$
 $r=8$

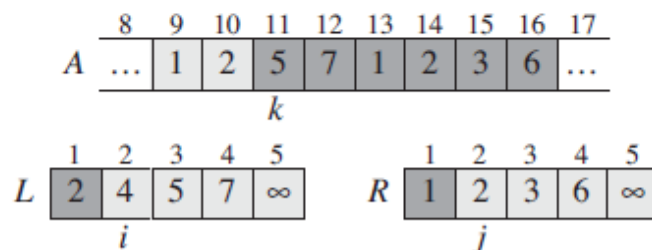
- El procedimiento MERGE trabaja de la siguiente manera:
- La línea 1 calcula la longitud n_1 del subarreglo $A[p..q]$, y la línea 2 calcula la longitud n_2 del subarreglo $A[q+1..r]$.
- Creamos matrices L y R ("izquierda" y "derecha"), de longitudes n_1+1 y n_2+1 , respectivamente, en la línea 3; la posición adicional en cada matriz tendrá el valor centinela.
- El bucle for de las líneas 4-5 copia el subarreglo $A[p..q]$ en $L[1..n_1]$, y el bucle for de las líneas 6-7 copia el subarreglo $A[q+1..r]$ en $R[1..n_2]$.
- Las líneas 8 a 9 colocan a los centinelas en los extremos de las matrices L y R .



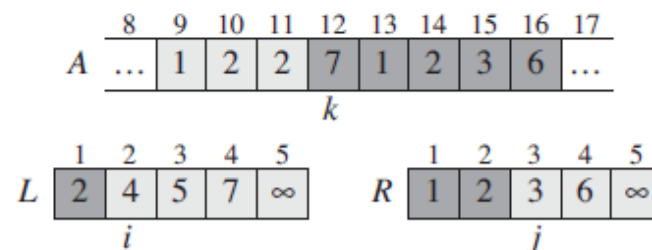
(a)



(b)

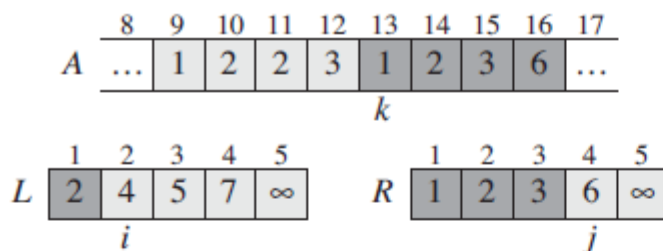


(c)

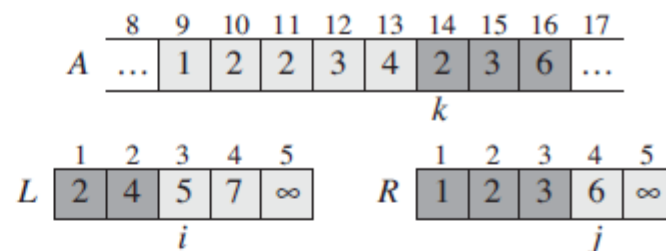


(d)

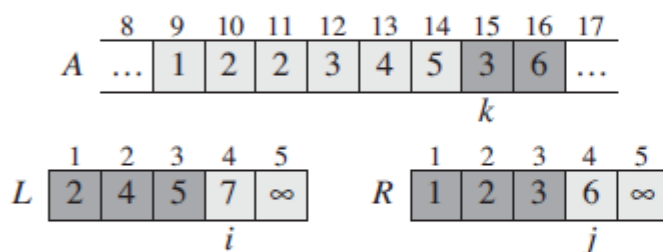
The operation of lines 10–17 in the call $\text{MERGE}(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A . Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A . (a)–(h) The arrays A , L , and R , and their respective indices k , i , and j prior to each iteration of the loop of lines 12–17.



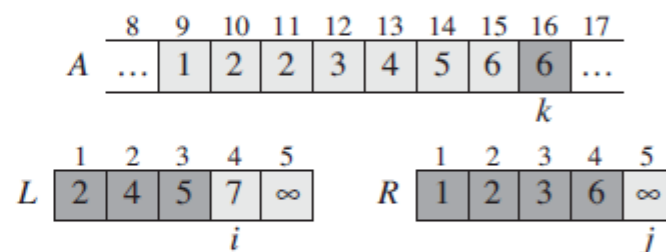
(e)



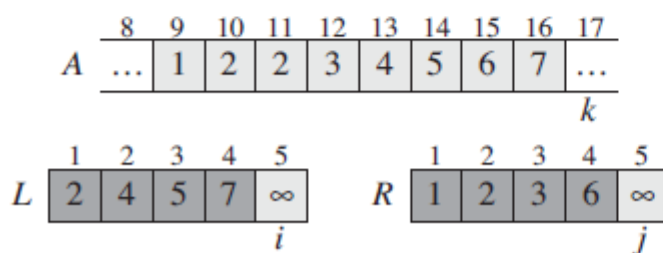
(f)



(g)



(h)



(i)

Figure 2.3, continued (i) The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A .

$$p=1$$

$$q=4$$

$$r=8$$

$$r - p + 1$$

$$8 - 1 + 1 = 8$$

- La figura anterior ilustra como se realiza los **$r - p + 1$** pasos básicos manteniendo el siguiente ciclo invariante:
- Al comienzo de cada iteración del bucle for de las líneas 12-17:

```
for k=p to r  
for k=1 to 8
```

 - El subarreglo $A[p \dots k - 1]$ contiene los $k - p$ elementos más pequeños de $L[1..n_1 + 1]$ y $R[1..n_2 + 1]$ de forma ordenada.
 - Además, $L[i]$ y $R[j]$ son los elementos más pequeños de sus arreglos que no se han copiado en A .

- **Inicialización:**

- Antes de la primera iteración del ciclo, tenemos $k=p$, de modo que el subarreglo $A[p..k-1]$ está vacío.
- Este subarreglo vacío contiene los $k - p = 0$ elementos más pequeños de L y R , y dado que $i = j = 1$, tanto $L[i]$ como $R[i]$ son los elementos más pequeños de sus arreglos que no han sido copiados en A .

- **Mantenimiento:**

- Para mostrar que cada iteración mantiene el ciclo invariante, primero supongamos que $L[i] \leq R[j]$. Entonces $L[i]$ es el elemento más pequeño aún no copiado en A .
- Porque $A[p..k - 1]$ contiene los $k - p$ elementos más pequeños, después de que la línea 14 copie $L[i]$ en $A[k]$, el subarreglo $A[p..k]$ contendrá los $k - p + 1$ elementos más pequeños.
- Incrementando k (en la actualización del bucle for) y también i (en la línea 15) se restablece el ciclo invariante para la siguiente iteración.
- Si en cambio $L[i] > R[j]$, entonces las líneas 16-17 realizan la acción apropiada para mantener el ciclo invariante.

- **Terminación:**
- Al término, cuando $k = r + 1$. Por el ciclo invariante, el subarreglo $A[p..k - 1]$ que es $A[p..r]$, contiene los $k - p = r - p + 1$ elementos más pequeños de $L[1..n_1 + 1]$ y $R[1..n_2 + 1]$ de forma ordenada.
- Los arreglos L y R juntos contienen elementos $n_1 + n_2 + 2 = r - p + 3$ elementos.
- Pero, todos los elementos, menos los dos más grandes se han copiado en A , y estos dos elementos más grandes son los centinelas.

Pseudocódigo Función Merge

MERGE(A, p, q, r)

1 2 3 4

5 6 7 8

2 4 5 7

1 2 3 6

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

$p=1$
 $q=4$
 $r=8$

- Veamos que el procedimiento MERGE se ejecuta en tiempo $\Theta(n)$, donde $n = r - p + 1$
- Observe que cada una de las líneas 1-3 y 8-11 toman un tiempo constante.
- Los bucles for de las líneas 4-7 toman un tiempo $\Theta(n_1 + n_2)$.
- Y hay n iteraciones para el bucle for de las líneas 12-17, cada una de las cuales toma un tiempo constante.

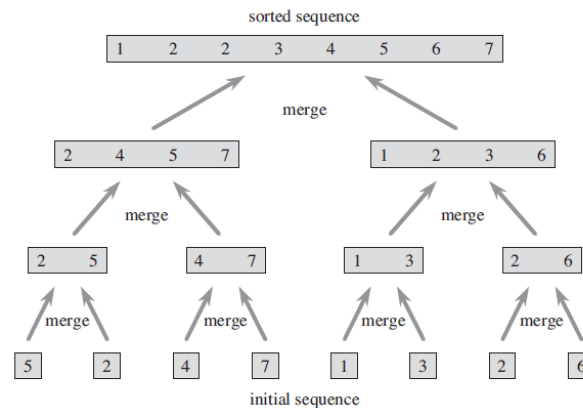
- Ahora podemos usar el procedimiento MERGE como una subrutina en el algoritmo de ordenamiento por mezcla.
- El procedimiento MERGE-SORT(A, p, r) ordena los elementos en el subarreglo $A[p..r]$.
- Si $p \geq r$, el subarreglo tiene como máximo un elemento y, por lo tanto, esta ya ordenado.
- De lo contrario, el paso de división simplemente calcula un índice q que divide $A[p..r]$ en dos subarreglos:
 - $A[p..q]$, que contiene $\lceil n/2 \rceil$ elementos, y $A[q+1..r]$ que contiene elementos $\lceil n/2 \rceil$

MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r)/2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

- Para ordenar la secuencia completa $A = \langle A[1], A[2], \dots, A[n] \rangle$ hacemos la llamada inicial MERGE-SORT($A, 1, A.length$) donde una vez más $A.length = n$.

La siguiente figura ilustra la operación del procedimiento de abajo hacia arriba cuando n es una potencia de 2.



El algoritmo consiste en fusionar pares de secuencias de 1 elemento para formar secuencias ordenadas de longitud 2.

Fusionando pares de secuencias de longitud 2 para formar secuencias ordenadas de longitud 4.

Y así sucesivamente, hasta que dos secuencias de longitud $n/2$ sean mezcladas para formar la secuencia final ordenada de longitud n .

Analizando los algoritmos de divide y vencerás

- Cuando un algoritmo contiene una llamada recursiva a sí mismo, a menudo podemos describir su tiempo de ejecución mediante una ecuación recurrente o recurrencia, la cual describe el tiempo de ejecución de un problema de tamaño n en términos del tiempo de ejecución para entradas más pequeñas.
- Luego podemos usar herramientas matemáticas para resolver la recurrencia y proporcionar límites en el rendimiento del algoritmo.

- Una recurrencia del tiempo de ejecución de un algoritmo de divide y vencerás parte de los tres pasos del paradigma básico.
- Decimos que $T(n)$ es el tiempo de corrida de un problema de tamaño n . Si el tamaño del problema es lo suficientemente pequeño, $n \leq c$ para alguna constante c , la solución sencilla toma un tiempo constante, el cual podemos escribir como $\Theta(1)$.
- Supongamos que nuestra división del problema produce a subproblemas, cada uno de los cuales es $1/b$ del tamaño del original. (Para la ordenación por mezcla, tanto a como b son 2)
- Se necesita tiempo $T(n/b)$ para resolver un subproblema de tamaño n/b , por lo que se necesita un tiempo de $aT(n/b)$ para resolver uno de ellos.
- Si tomamos el tiempo $D(n)$ para dividir el problema en subproblemas y el tiempo $C(n)$ para combinar las soluciones de los subproblemas en la solución al problema original, obtenemos la recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

Análisis de Merge Sort

- Aunque el pseudocódigo para MERGE-SORT funciona correctamente cuando el número de elementos no es par, nuestro análisis basado en la recurrencia se simplifica si asumimos que el tamaño del problema original es una potencia de 2.
- Cada paso de división produce dos subsecuencias de tamaño exactamente igual $n/2$.
- Merge Sort de un solo elemento toma un tiempo constante. Cuando tenemos $n > 1$ elementos, desglosamos el tiempo de ejecución de la siguiente manera:

- **Dividir:** El paso de división solo calcula la mitad del subarreglo, que toma un tiempo constante. Así, $D(n) = \Theta(1)$.
- **Conquista:** Resolvemos de forma recursiva dos subproblemas, cada uno de tamaño $n/2$, lo que contribuye $2T(n/2)$ al tiempo de ejecución.
- **Combina:** Ya hemos notado que el procedimiento MERGE de n elementos en un subarreglo toma un tiempo $\Theta(n)$, y entonces, $C(n) = \Theta(n)$.

- Cuando agregamos las funciones $D(n)$ y $C(n)$ en el análisis de ordenación por mezcla, estamos agregando una función que es $\Theta(n)$ y una función que es $\Theta(1)$.
- Esta suma es una función lineal de n , es decir, $\Theta(n)$.
- Agregándolo al término $2T(n/2)$ del paso "conquistar" da la recurrencia para el peor tiempo de ejecución $T(n)$ de merge sort:

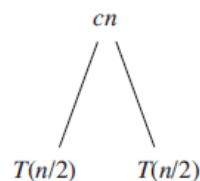
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

- Hay un "teorema maestro" que se puede usar para demostrar que **$T(n)$ es $\Theta(n \lg n)$** , donde $\lg n$ representa $\log_2 n$.
- Porque la función logaritmo crece más lentamente que cualquier función lineal, para entradas lo suficientemente grandes, merge sort, con su tiempo de ejecución **$\Theta(n \lg n)$** supera a la ordenación por inserción, cuyo tiempo de ejecución es $\Theta(n^2)$ en el peor de los casos.
- Se puede comprender intuitivamente por qué la solución a la recurrencia anterior es **$T(n) = \Theta(n \lg n)$** . Reescribamos la recurrencia como:

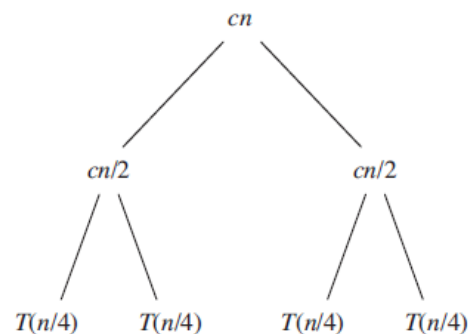
$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases}$$

Donde la constante c representa el tiempo requerido para resolver problemas de tamaño 1 como así como el tiempo por elemento del arreglo para dividir y combinar los pasos.

$T(n)$



(a)



(b)

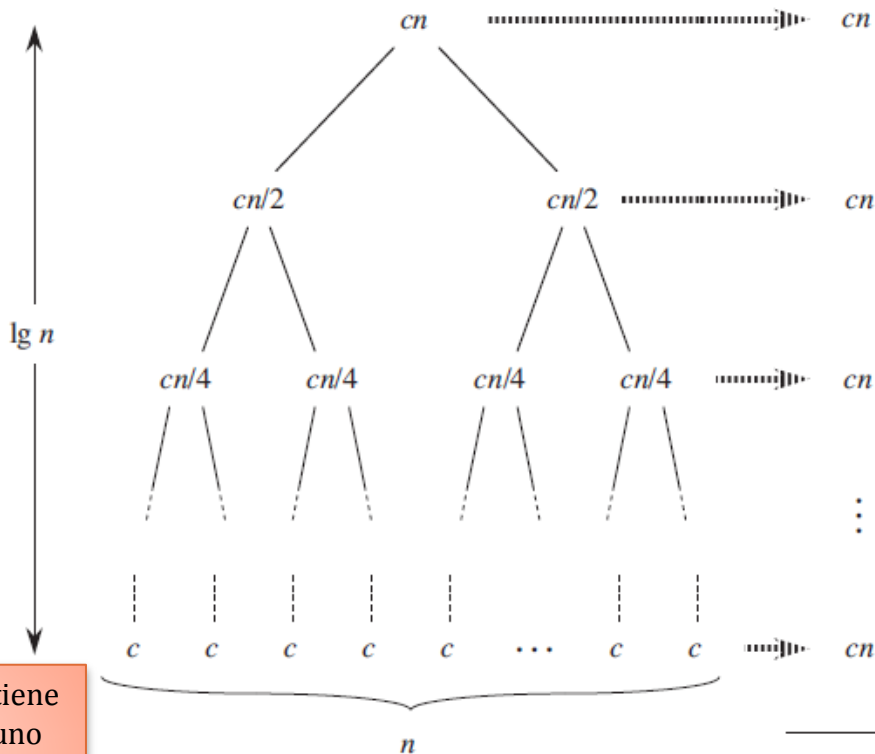
(c)

Cómo construir un árbol de recursividad para la recurrencia $T(n) = 2T(n/2)$.

La parte (a) muestra $T(n)$ que se expande progresivamente en (b), (c) y (d) para formar el árbol de recursividad.

El árbol completamente expandido en la parte (d) tiene $\lg n + 1$ niveles (es decir, tiene una altura $\lg n$, como se indica), y cada nivel aporta un costo total de cn .

El costo total, por lo tanto, es $cn \lg n + cn$, que es $\Theta(n \lg n)$



(d)

Total: $cn \lg n + cn$

El último nivel tiene n nodos cada uno contribuye con un costo c , teniendo un costo total de cn .

