

selectionSort(A, n):

for $i = 1 \dots n - 1$:

$smallest = i$

for $j = i + 1 \dots n$:

if $A[j] < A[smallest]$:

$smallest = j$

Intercambiar $A[i]$ con $A[smallest]$

Algoritmos de ordenación

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?

Vimos dos algoritmos de ordenación, ambos *in place* y $O(n^2)$ en el peor caso

Selection sort:

- no tiene un mejor caso claramente mejor que el peor caso

Insertion sort:

- es $O(n)$ en el mejor caso —cuando los datos ya vienen ordenados
- sigue siendo $O(n^2)$ en el caso promedio —promedio sobre todas las permutaciones posibles igualmente probables

Vimos que el número promedio de inversiones en un arreglo de n elementos es $n(n-1)/4$

... por lo que un algoritmo que ordena intercambiando elementos (después de compararlos)

... y sólo corrige una inversión por intercambio (es decir, compara e intercambia sólo elementos adyacentes),

... no puede ordenar más rápidamente que $O(n^2)$ en promedio (y en el peor caso)

Pero, ¿qué pasa si un intercambio corrige más de una inversión?

P.ej., $A = [34 \ 8 \ 64 \ 51 \ 32 \ 21]$ tiene 9 inversiones:

- $(34, 8)$, $(34, 32)$, $(34, 21)$, $(64, 51)$, $(64, 32)$, $(64, 21)$, $(51, 32)$, $(51, 21)$, $(32, 21)$

Si intercambiamos 34 y 8, corregimos sólo una inversión:

- $(34, 8)$

... pero si intercambiamos 34 y 21, corregimos seis:

- $(34, 8)$, $(34, 32)$, $(34, 21)$, $(64, 21)$, $(51, 21)$, $(32, 21)$
- (...aunque introducimos algunas inversiones nuevas)

Otra instancia del problema de ordenación



Si sabemos que los datos están separados en dos secuencias ordenadas,

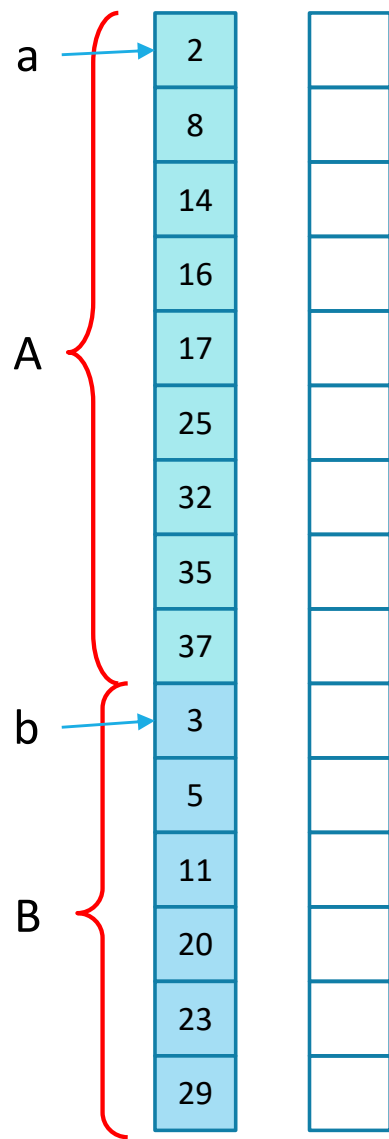
... ¿cómo podemos aprovechar este hecho para ordenar todos los datos?



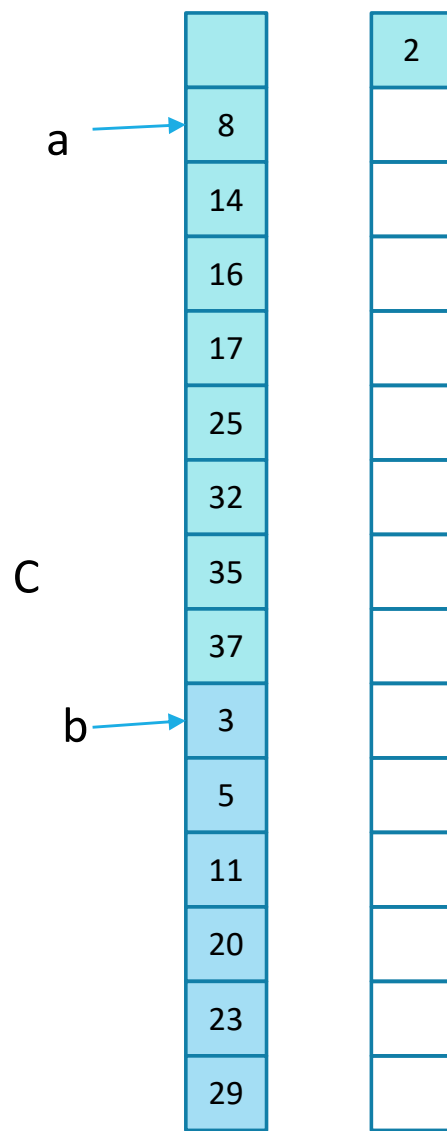
Mezcla (o *merge*) de dos secuencias ordenadas, en una tercera secuencia ordenada

Para dos secuencias ordenadas, A y B :

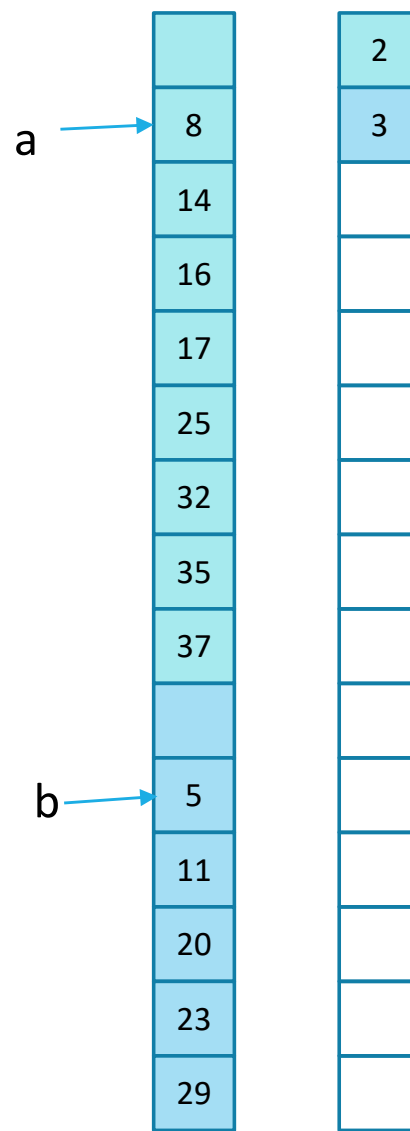
1. Sea C una secuencia ordenada, inicialmente vacía
2. Sean a y b el primer elemento de A y B , respectivamente
3. Extraer de su respectiva secuencia el menor entre a y b
4. Insertar ese elemento extraído al final de C
5. Si quedan elementos en ambas A y B , volver a 2.
6. Concatenar C con la secuencia que aún tenga elementos



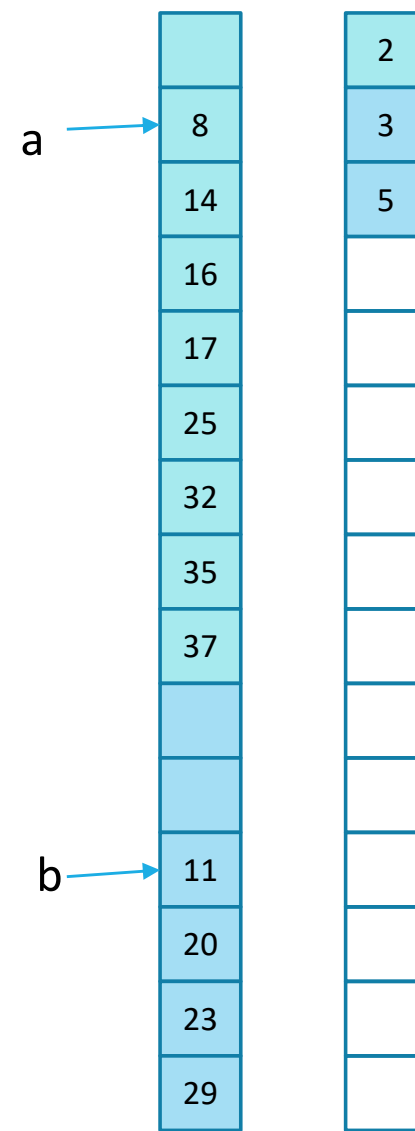
estado
inicial



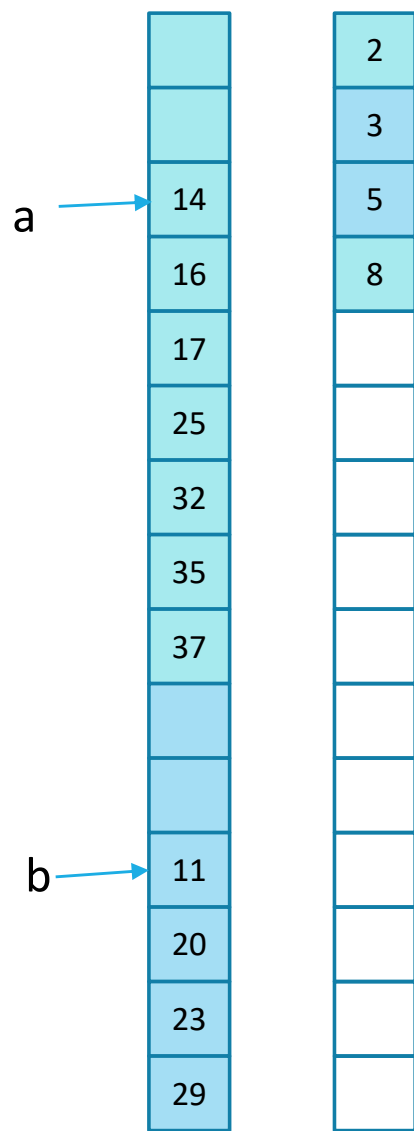
primera
ejecución de
pasos 3 y 4



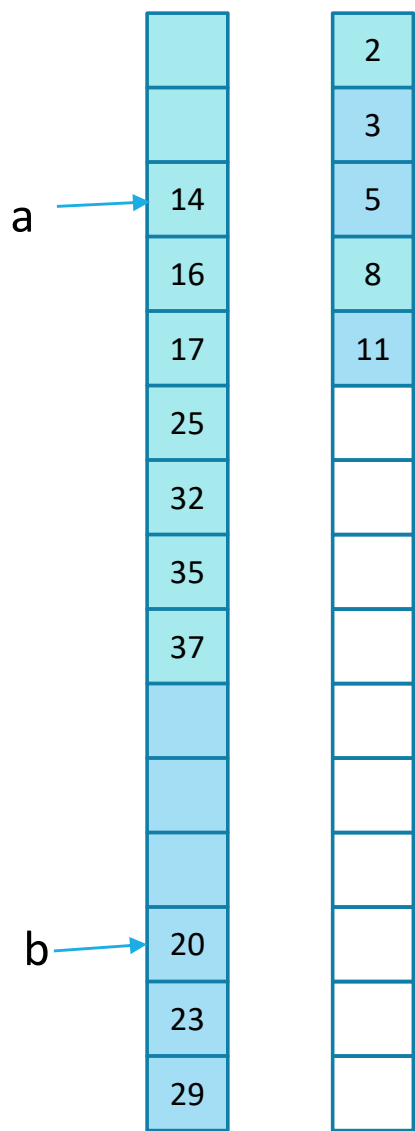
segunda
ejecución de
pasos 3 y 4



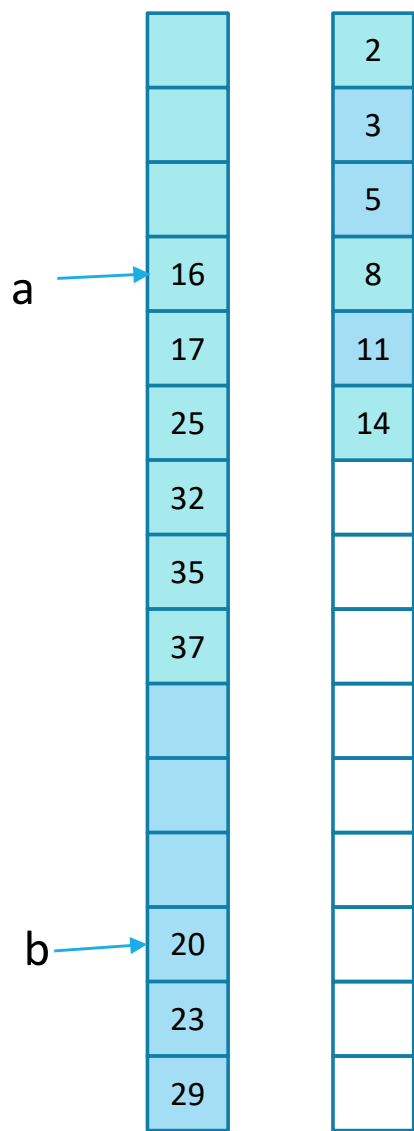
tercera
ejecución de
pasos 3 y 4



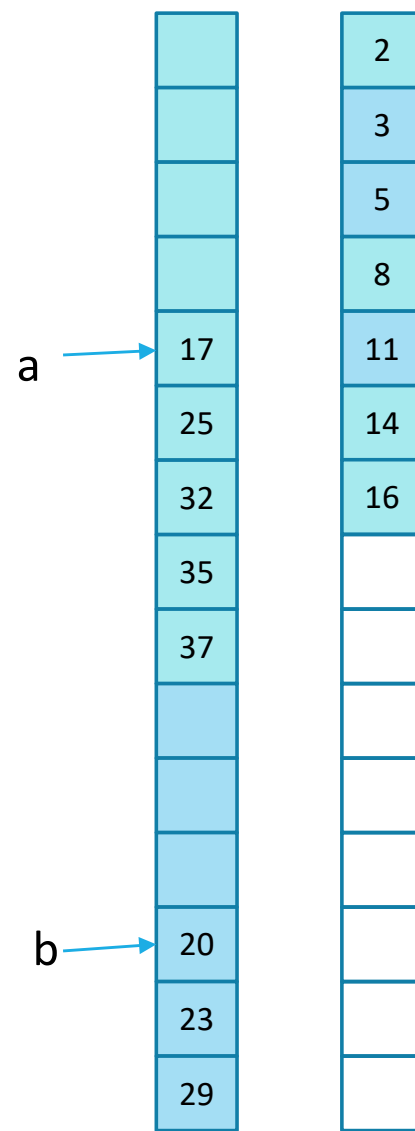
cuarta
ejecución de
pasos 3 y 4



quinta
ejecución de
pasos 3 y 4



sexta
ejecución de
pasos 3 y 4



séptima
ejecución de
pasos 3 y 4

etc.

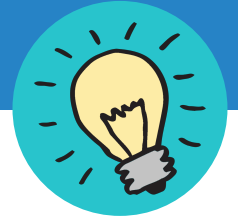
Propiedades de *merge*



¿Cuál es su complejidad de memoria?

¿Se puede ejecutar *in place*?

Propiedades de *merge*



¿Cuál es su complejidad de memoria?

¿Se puede ejecutar *in place*?

(en principio, *merge* se podría ejecutar *in place*, pero, como se puede deducir de la figura anterior, esto requeriría desplazar la secuencia *A* un lugar hacia abajo cada vez que se saca un elemento de la secuencia *B*)

¿Cómo demostramos que *merge* es correcto?

- es decir, termina y ordena

Finitud

En cada paso *merge* extrae un elemento de A o B y lo pone en C

Cuando una de las secuencias queda vacía, se toma todo lo de la otra secuencia y se pone en C

En total se hacen $|A| + |B|$ pasos, y como tanto A como B son finitos, el algoritmo es finito

Corrección

(por inducción sobre las inserciones en C)

PD: Luego de insertar el último elemento en C , ésta está ordenada

Caso Base: Luego de la primera inserción, C tiene un solo elemento x_1 , por lo que está ordenada.

Hipótesis Inductiva: Luego de la i -ésima inserción (la del elemento x_i), C está ordenada. Ahora toca la siguiente inserción.

- Si quedan elementos en A y en B , sea x_{i+1} el menor entre las cabezas de A y de B .
- Si sólo quedan elementos en una de las dos secuencias, sea x_{i+1} la cabeza de ésta.

Se elimina x_{i+1} de su respectiva secuencia y se inserta al final de C .

$x_i \leq x_{i+1}$. De no ser así x_{i+1} habría salido antes, ó A y B no habrían estado ordenadas.

Como C estaba ordenada, $x_1 \leq x_2 \leq \dots \leq x_i$, y como $x_i \leq x_{i+1}$, entonces C está ordenada.

Por lo tanto, luego de insertar el último elemento x_n , C está ordenada.

Propiedades de *merge*



¿Cuál es su complejidad de memoria? $\rightarrow O(n)$

¿Se puede ejecutar *in place*? \rightarrow Sí pero demora más

¿Es correcto *merge*? \rightarrow Sí (demostrado)

¿Cuál es su complejidad de tiempo?

Complejidad de tiempo de *merge*

Para dos secuencias ordenadas, A y B :

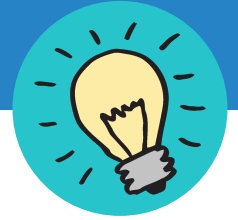
1. Sea C una secuencia ordenada, inicialmente vacía
2. Sean a y b el primer elemento de A y B , respectivamente
3. Extraer de su respectiva secuencia el menor entre a y b
4. Insertar ese elemento extraído al final de C
5. Si quedan elementos en ambos A y B , volver a 2.
6. Concatenar C con la secuencia que aún tenga elementos

Definiciones \rightarrow no influyen

$O(1)$ c/u $\rightarrow O(1)$ en total

Sumar el $O(1)$ anterior tantas veces como la cantidad de elementos que hay en A y B combinados $\rightarrow O(n)$

Ordenación basada en *merge*



¿Podemos usar **merge** para ordenar una secuencia arbitraria?

(recordemos: *merge* necesita partir de dos (sub)secuencias ya ordenadas)

Si de algún modo primero podemos crear dos secuencias ordenadas

(p.ej., la mitad izquierda y la mitad derecha de la secuencia arbitraria inicial)

... entonces luego podemos combinarlas usando *merge*, ordenando así la secuencia completa

La estrategia algorítmica dividir para reinar

1. Dividir el problema original en dos (o más) subproblemas del mismo tipo
2. Resolver (recursivamente) cada subproblema
3. Encontrar la solución al problema original a partir de las soluciones a cada subproblema

El algoritmo *mergeSort*

Para una secuencia A :

1. Si A tiene un solo elemento, entonces A está ordenada; terminar en este paso
2. Dividir la secuencia en mitades
3. Ordenar cada mitad **recursivamente** usando *mergeSort*
4. Combinar las mitades (ya ordenadas) usando *merge*

29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
----	---	---	----	----	----	----	----	----	----	----	---	----	----	----	---

29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
----	---	---	----	----	----	----	----	----	----	----	---	----	----	----	---

dividir
(simplemente, partir en la mitad)

29	5	3	59	19	43	17	13
----	---	---	----	----	----	----	----

47	53	31	2	11	37	23	7
----	----	----	---	----	----	----	---

ordenar recursivamente

ordenar recursivamente

3	5	13	17	19	29	43	59
---	---	----	----	----	----	----	----

2	7	11	23	31	37	47	53
---	---	----	----	----	----	----	----

mezclar
(como ya sabemos)

2	3	5	7	11	13	17	19	23	29	31	37	43	47	53	59
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

dividir

29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7

mezclar

5	29	3	59	19	43	13	17	47	53	2	31	11	37	7	23
3	5	29	59	13	17	19	43	2	31	47	53	7	11	23	37
3	5	13	17	19	29	43	59	2	7	11	23	31	37	47	53
2	3	5	7	11	13	17	19	23	29	31	37	43	47	53	59

... y sus propiedades

Demuestra que *mergeSort* es correcto

Calcula y justifica su complejidad

mergeSort es un algoritmo recursivo

Todo algoritmo recursivo debe chequear en primer lugar el *caso base*:

- el caso cuya solución se calcula **sin hacer recursión**

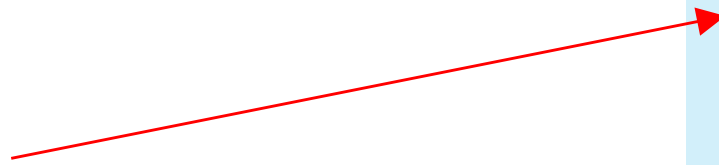
Las llamadas recursivas deben ser para **casos** distintos al caso original y **que se acerquen un poco más al caso base**

mergeSort(secuencia *A*):

1. Si *A* tiene un solo elemento, entonces *A* está ordenada; terminar en este paso
2. Dividir la secuencia en mitades
3. Ordenar cada mitad recursivamente usando *mergeSort*
4. Combinar las mitades (ya ordenadas) usando *merge*

Sea $T(n)$ el número de pasos que *mergeSort* toma para ordenar n elementos

El caso base se calcula aparte:

$$T(1) = 1$$


El caso general se calcula a partir de una *ecuación de recurrencia*:

$$T(n) = 2T(n/2) + n$$


mergeSort(secuencia A):

1. Si A tiene un solo elemento, entonces A está ordenada; terminar en este paso
2. Dividir la secuencia en mitades
3. Ordenar cada mitad recursivamente usando *mergeSort*
4. Combinar las mitades (ya ordenadas) usando *merge*

Resolvamos la recurrencia $T(n) = 2T(n/2) + n$ sabiendo que $T(1) = 1$

$$\begin{array}{rcl} T(n)/n & = & T(n/2)/(n/2) + 1 \\ T(n/2)/(n/2) & = & T(n/4)/(n/4) + 1 \\ T(n/4)/(n/4) & = & T(n/8)/(n/8) + 1 \\ & \vdots & \\ T(2)/2 & = & T(1)/1 + 1 \end{array} \quad \left. \vphantom{\begin{array}{rcl} T(n)/n & = & T(n/2)/(n/2) + 1 \\ T(n/2)/(n/2) & = & T(n/4)/(n/4) + 1 \\ T(n/4)/(n/4) & = & T(n/8)/(n/8) + 1 \\ & \vdots & \\ T(2)/2 & = & T(1)/1 + 1 \end{array}} \right\} \log_2 n \text{ niveles}$$

Sumando ambas columnas (las sumas son iguales) y cancelando los términos que aparecen a ambos lados del signo "=", obtenemos

$$T(n)/n = T(1)/1 + \log n$$

$$\Rightarrow T(n) = n \cdot \log n + n = O(n \cdot \log n)$$

En resumen ...

mergeSort es un algoritmo de ordenación

... de complejidad de tiempo $O(n \log n)$

- una mejora importante con respecto a los algoritmos $O(n^2)$

... y de complejidad de espacio $O(n)$

- debido a que *merge* requiere un arreglo adicional para mezclar eficientemente dos secuencias ordenadas

mergesort es un algoritmo basado en la estrategia dividir para reinar

A. Dividir el problema original en dos subproblemas del mismo tipo

B. Resolver (recursivamente) cada subproblema

C. Encontrar la solución al problema original a partir de las soluciones a cada subproblema

mergeSort(secuencia *A*):

1. Si *A* tiene un solo elemento, terminar en este paso
2. Dividir la secuencia en mitades
3. Ordenar cada mitad recursivamente usando *mergeSort*
4. Combinar las mitades (ya ordenadas) usando *merge*

Algoritmos de Ordenación

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
?	?	?	?	?
?	?	?	?	?