Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant

Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a+b)/2 \rfloor$: $A[a, q]$ and $A[q+1, b]$; or $A[a, q-1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant

Stable sorting algorithm

- Stable sorting algorithm has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input.

Running Time for Merge-Sort Using Recurrence

- $T(n) =$ running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

Running Time for Merge-Sort Using Recurrence

- $T(n) =$ running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Even simpler: $T(n) = 2T(n/2) + O(n)$. (Implicit assumption: $T(n) = O(1)$ if n is at most some constant.)

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Even simpler: $T(n) = 2T(n/2) + O(n)$. (Implicit assumption: $T(n) = O(1)$ if n is at most some constant.)
- Solving this recurrence, we have $T(n) = O(n \lg n)$ (we shall show how later)

Outline

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:

$$10 \qquad 8 \qquad 15 \qquad 9 \qquad 12$$

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:

10	8	15	9	12
8	9	10	12	15

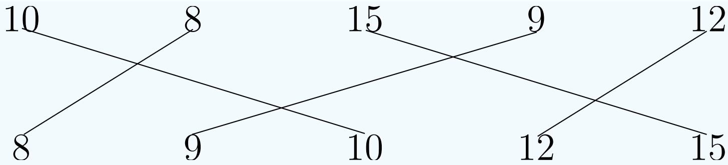
Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:



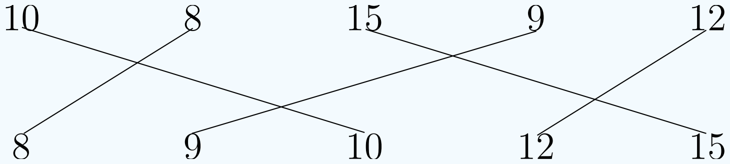
Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:



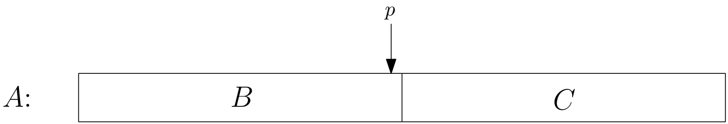
- 4 inversions (for convenience, using numbers, not indices): $(10, 8), (10, 9), (15, 9), (15, 12)$

Naive Algorithm for Counting Inversions

count-inversions(A, n)

```
c ← 0
for every i ← 1 to n − 1 do
    for every j ← i + 1 to n do
        if A[i] > A[j] then c ← c + 1
return c
```

Divide-and-Conquer



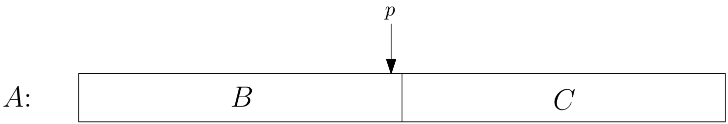
- $p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p+1..n]$
- $$\#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m$$ $$m = \Big|\big\{(i,j) : B[i] > C[j]\big\}\Big|$$

Q: How fast can we compute m, via trivial algorithm?

A: $O(n^2)$

- Can not improve the $O(n^2)$ time for counting inversions.

Divide-and-Conquer



- $p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p+1..n]$
- $$\#\mathsf{invs}(A) = \#\mathsf{invs}(B) + \#\mathsf{invs}(C) + m$$
$$m = \Big|\big\{(i,j) : B[i] > C[j]\big\}\Big|$$

Lemma If both B and C are sorted, then we can compute m in $O(n)$ time!

Counting Inversions between B and C

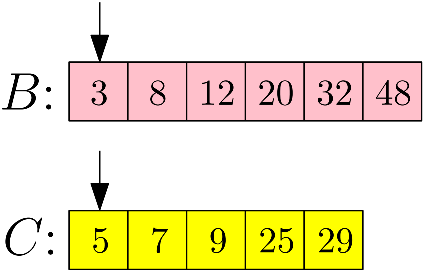
Count pairs i, j such that $B[i] > C[j]$:

B: | 3 | 8 | 12 | 20 | 32 | 48 | total $= 0$

C: | 5 | 7 | 9 | 25 | 29 |

Counting Inversions between B and C

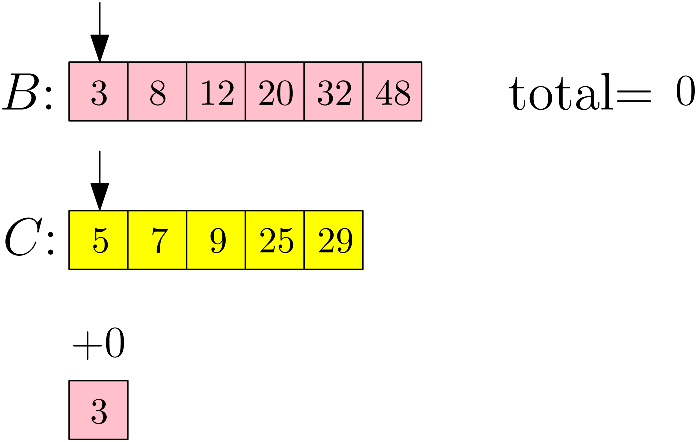
Count pairs i, j such that $B[i] > C[j]$:



$\text{total} = 0$

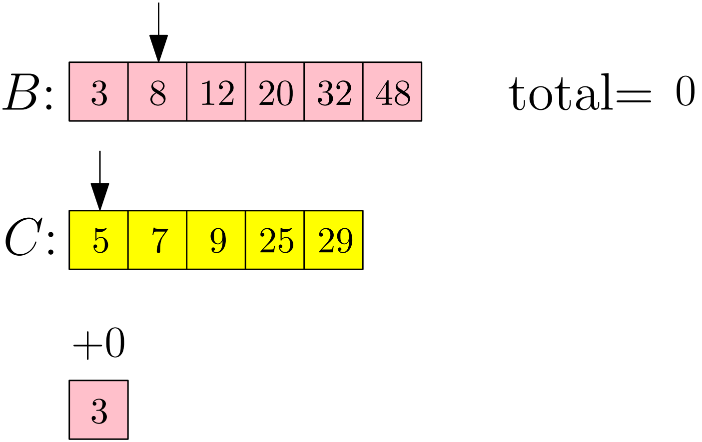
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B:

3	8	12	20	32	48

total $=0$

C:

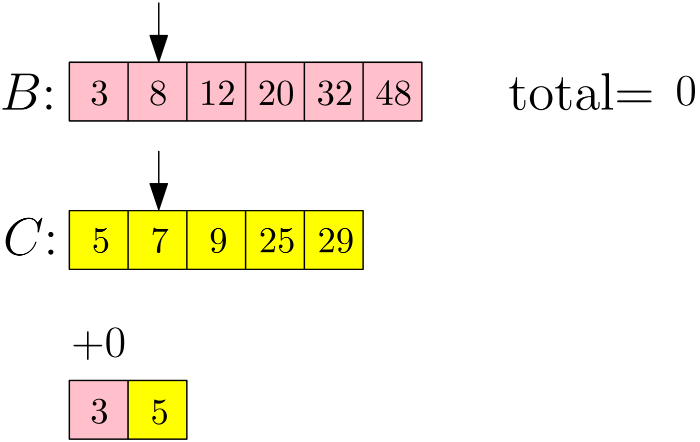
5	7	9	25	29

$+0$

3	5

Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B:

3	8	12	20	32	48

total$=$ 0

C:

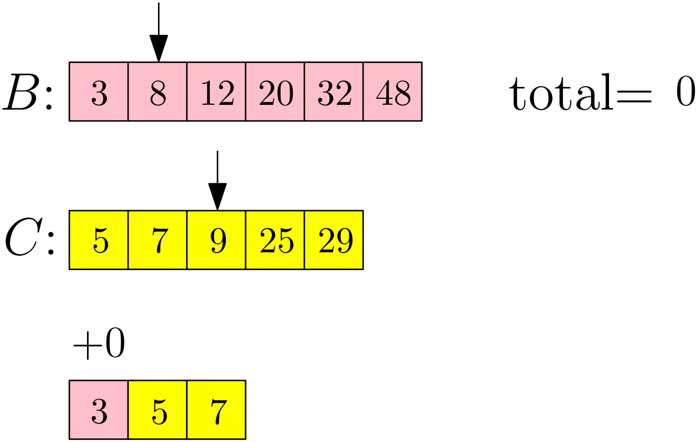
5	7	9	25	29

$+0$

3	5	7

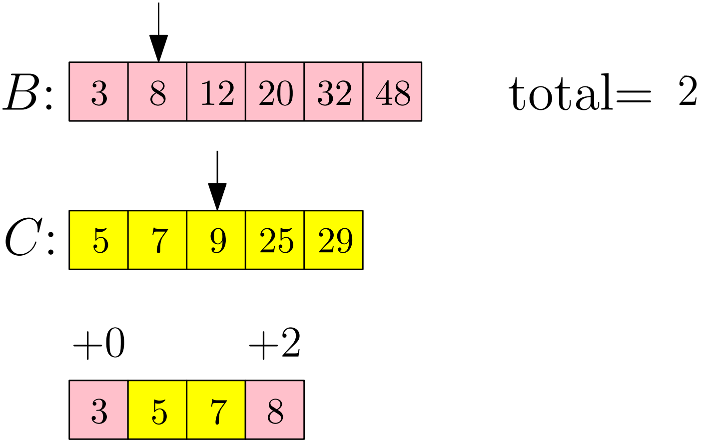
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



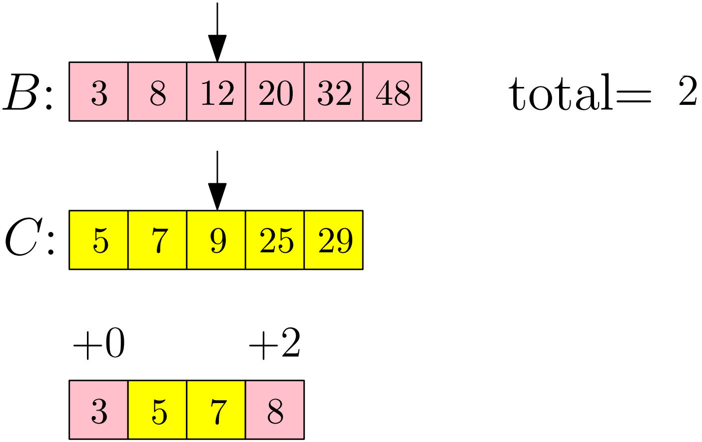
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B: | 3 | 8 | 12 | 20 | 32 | 48 |

total $= 2$

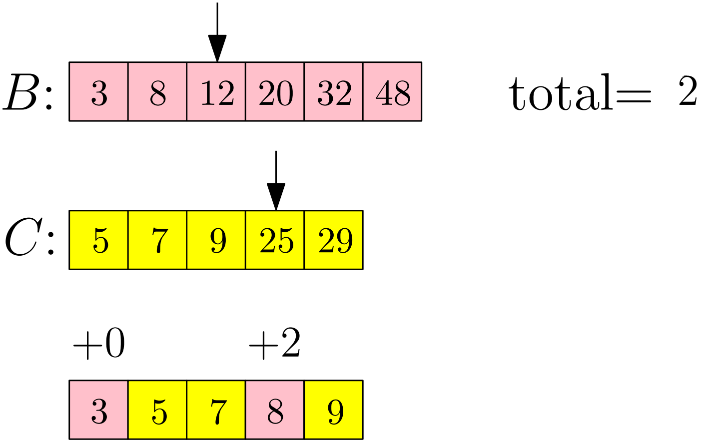
C: | 5 | 7 | 9 | 25 | 29 |

$+0$ $+2$

| 3 | 5 | 7 | 8 | 9 |

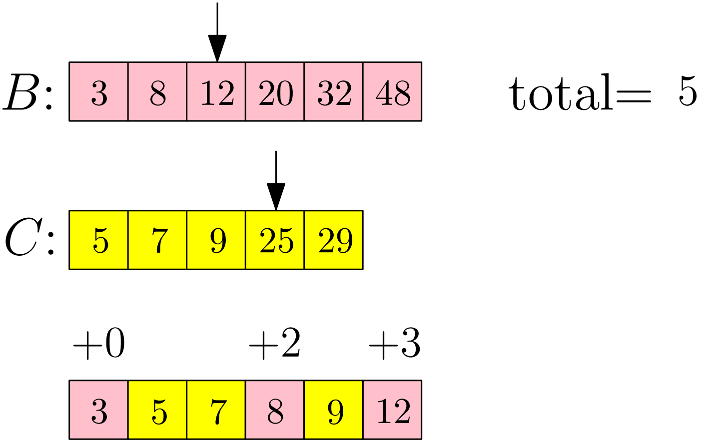
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



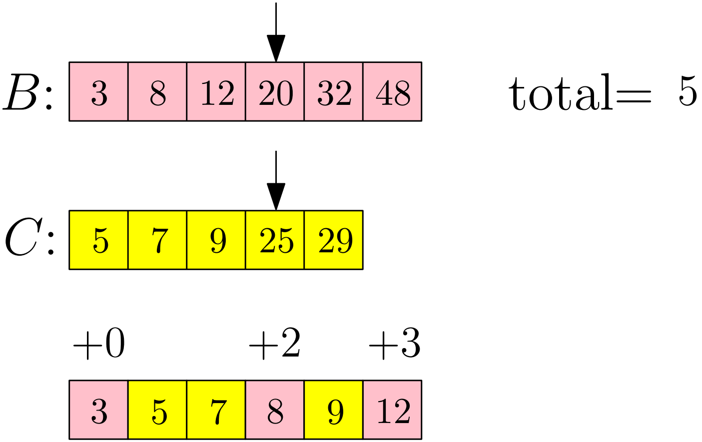
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



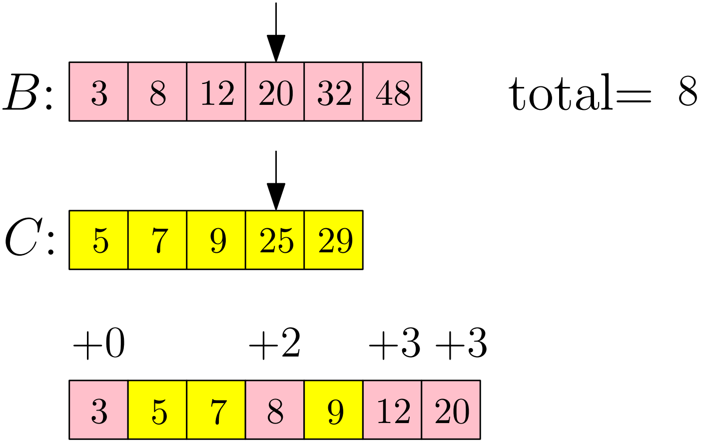
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



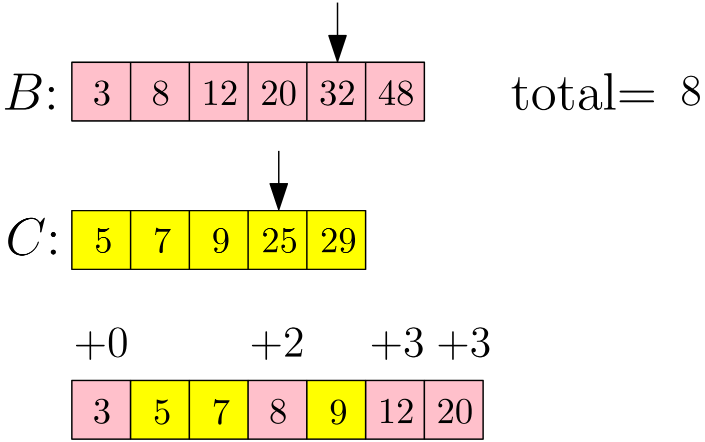
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



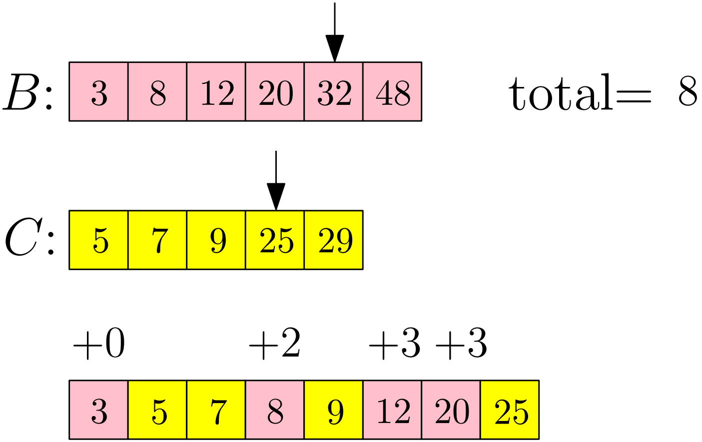
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B:

3	8	12	20	32	48

total $= 8$

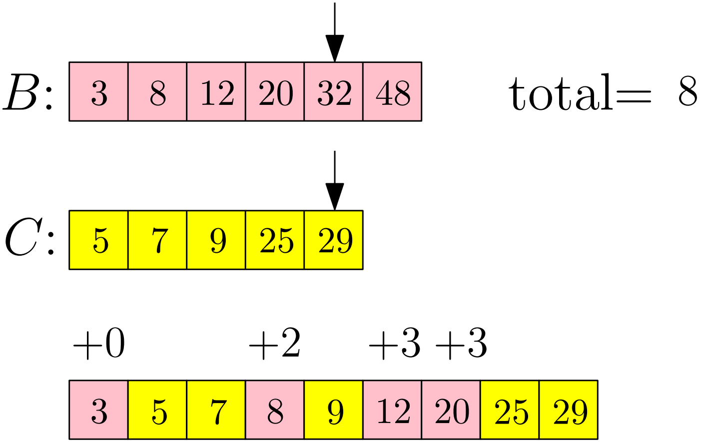
C:

5	7	9	25	29

$+0$			$+2$		$+3$	$+3$	
3	5	7	8	9	12	20	25

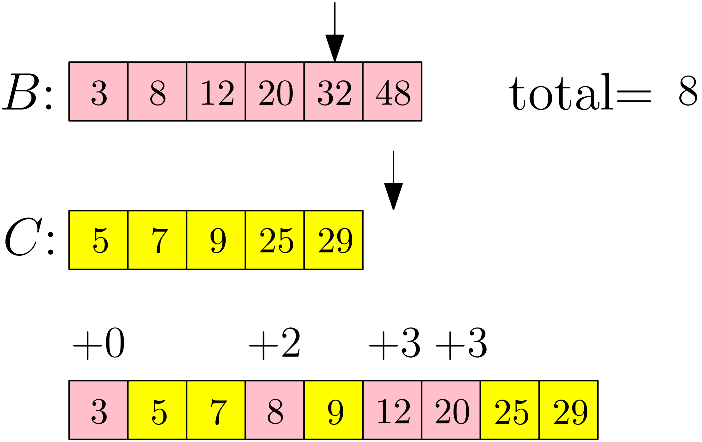
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



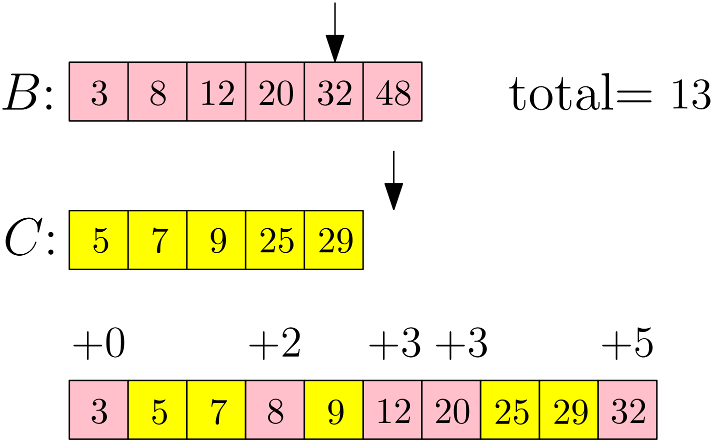
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



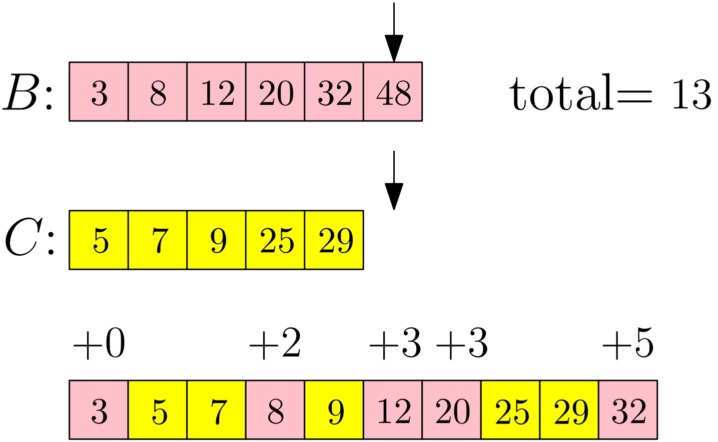
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



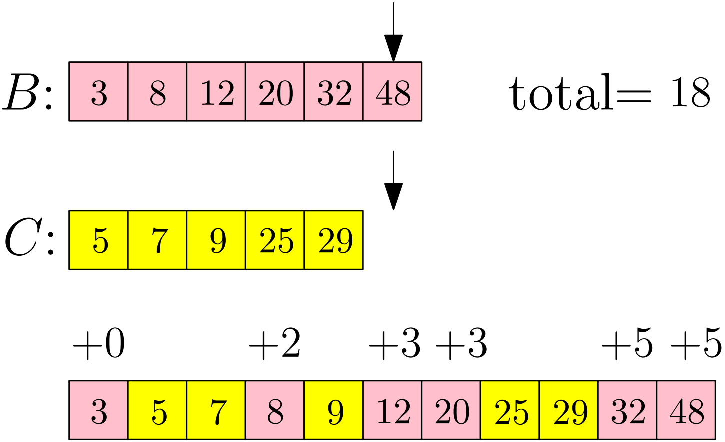
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



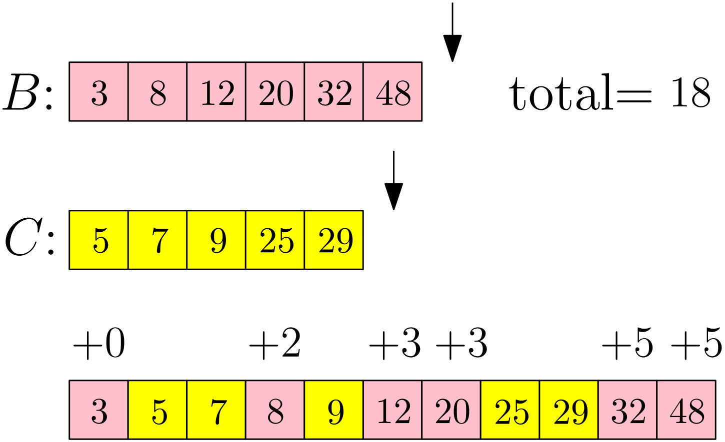
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Count Inversions between B and C

- Procedure that merges B and C and counts inversions between B and C at the same time

merge-and-count(B, C, n_1, n_2)

$count \leftarrow 0$;
$A \leftarrow$ array of size $n_1 + n_2$; $i \leftarrow 1$; $j \leftarrow 1$
while $i \leq n_1$ or $j \leq n_2$ **do**
    if $j > n_2$ or $(i \leq n_1$ and $B[i] \leq C[j])$ **then**
        $A[i+j-1] \leftarrow B[i]$; $i \leftarrow i+1$
        $count \leftarrow count + (j-1)$
    else
        $A[i+j-1] \leftarrow C[j]$; $j \leftarrow j+1$
return $(A, count)$

Sort and Count Inversions in A

- A procedure that returns the sorted array of A and counts the number of inversions in A:

sort-and-count(A, n)

if $n = 1$ **then**
  return $(A, 0)$
else
  $(B, m_1) \leftarrow$ sort-and-count$\Big(A\big[1..\lfloor n/2 \rfloor\big], \lfloor n/2 \rfloor\Big)$
  $(C, m_2) \leftarrow$ sort-and-count$\Big(A\big[\lfloor n/2 \rfloor + 1..n\big], \lceil n/2 \rceil\Big)$
  $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
  return $(A, m_1 + m_2 + m_3)$

Sort and Count Inversions in A

- A procedure that returns the sorted array of A and counts the number of inversions in A:

sort-and-count(A, n)

```
1: if n = 1 then
2:     return (A, 0)
3: else
4:     (B, m_1) ← sort-and-count(A[1..⌊n/2⌋], ⌊n/2⌋)
5:     (C, m_2) ← sort-and-count(A[⌊n/2⌋ + 1..n], ⌈n/2⌉)
6:     (A, m_3) ← merge-and-count(B, C, ⌊n/2⌋, ⌈n/2⌉)
7:     return (A, m_1 + m_2 + m_3)
```

- Divide: trivial
- Conquer: 4, 5
- Combine: 6, 7

sort-and-count(A, n)

if $n = 1$ **then**
    return $(A, 0)$
else
    $(B, m_1) \leftarrow$ sort-and-count$\Big(A\big[1..\lfloor n/2 \rfloor\big], \lfloor n/2 \rfloor\Big)$
    $(C, m_2) \leftarrow$ sort-and-count$\Big(A\big[\lfloor n/2 \rfloor + 1..n\big], \lceil n/2 \rceil\Big)$
    $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
    return $(A, m_1 + m_2 + m_3)$

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$

sort-and-count(A, n)

```
if n = 1 then
    return (A, 0)
else
    (B, m_1) ← sort-and-count(A[1..⌊n/2⌋], ⌊n/2⌋)
    (C, m_2) ← sort-and-count(A[⌊n/2⌋ + 1..n], ⌈n/2⌉)
    (A, m_3) ← merge-and-count(B, C, ⌊n/2⌋, ⌈n/2⌉)
    return (A, m_1 + m_2 + m_3)
```

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$
- Running time $= O(n \lg n)$