CSE 431/531: Algorithm Analysis and Design (Fall 2024)
Divide-and-Conquer

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Outline

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: a 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: a 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: a sequence A of n numbers

Output: number of inversions in A

Example:

10	8	15	9	12
8	9	10	12	15

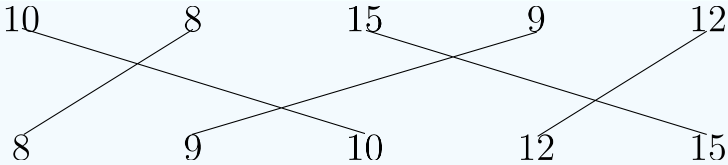
Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: a sequence A of n numbers

Output: number of inversions in A

Example:



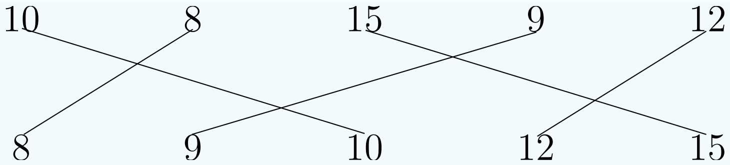
Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: a sequence A of n numbers

Output: number of inversions in A

Example:



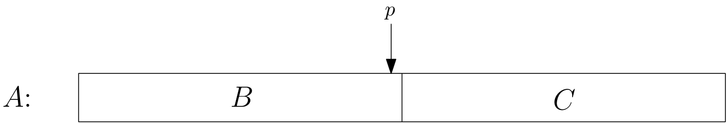
- 4 inversions (for convenience, using numbers, not indices): $(10, 8), (10, 9), (15, 9), (15, 12)$

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



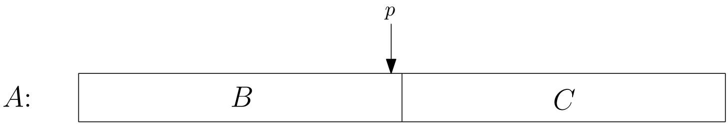
- $p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p+1..n]$
- $$\#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m$$
  $$m = \left|\left\{(i,j) : B[i] > C[j]\right\}\right|$$

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



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

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 $B[i] > C[j]$:

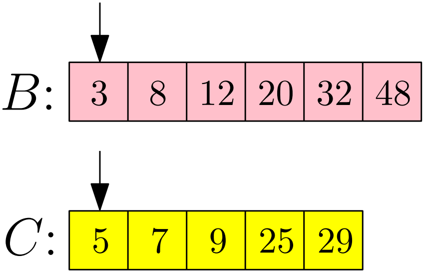
B:	3	8	12	20	32	48

total$=$ 0

C:	5	7	9	25	29

Counting Inversions between B and C

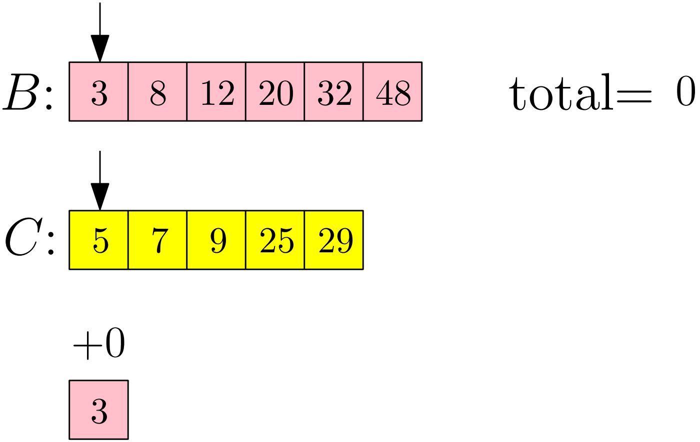
Count pairs i, j such that $B[i] > C[j]$:



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 $B[i] > C[j]$:

B:

3	8	12	20	32	48

total$=$ 0

C:

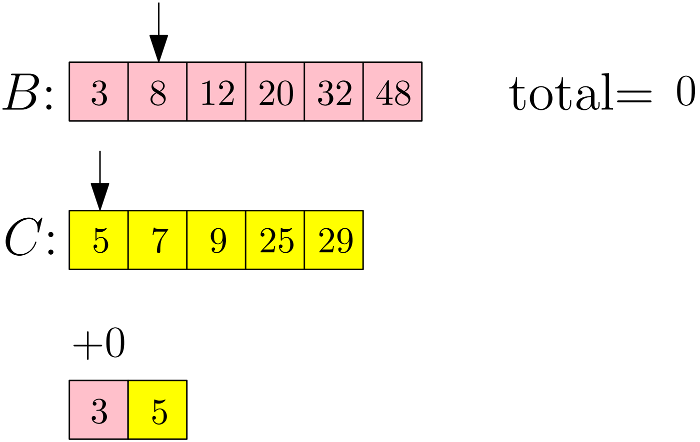
5	7	9	25	29

$+0$

3

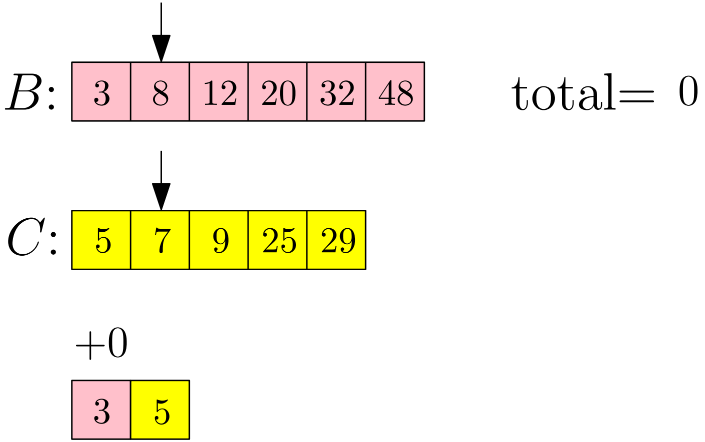
Counting Inversions between B and C

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:

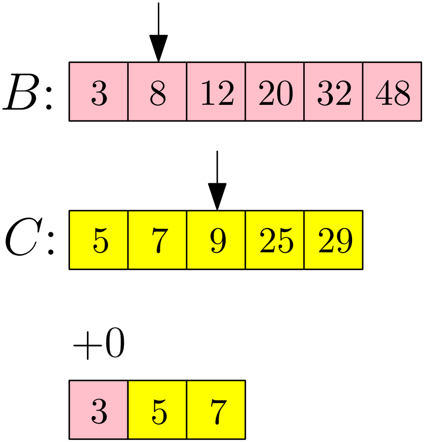
5	7	9	25	29

$+0$

3	5	7

Counting Inversions between B and C

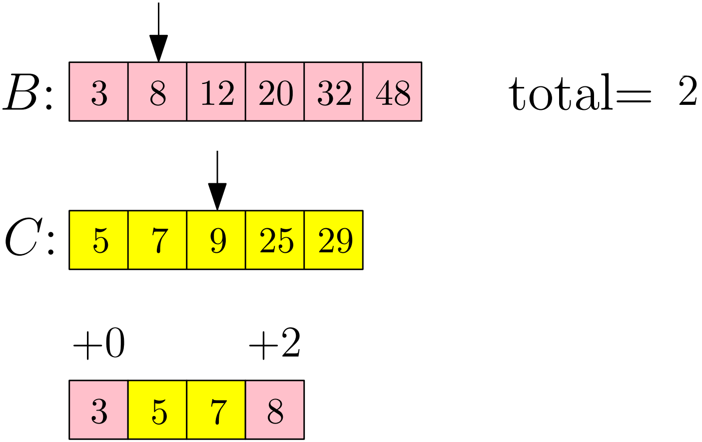
Count pairs i, j such that $B[i] > C[j]$:



total$=$ 0

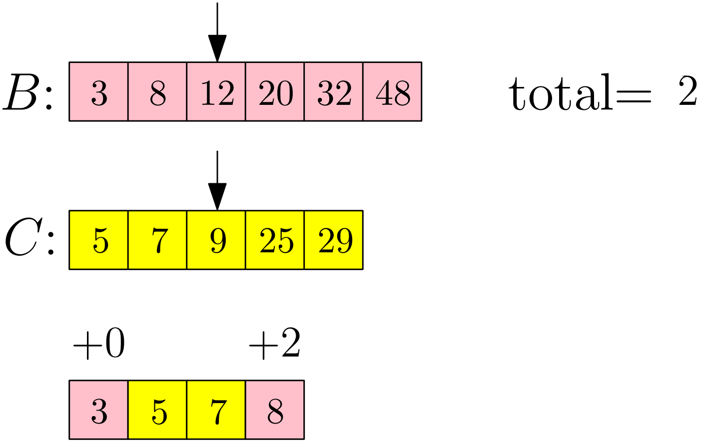
Counting Inversions between B and C

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$=$ 2

C:

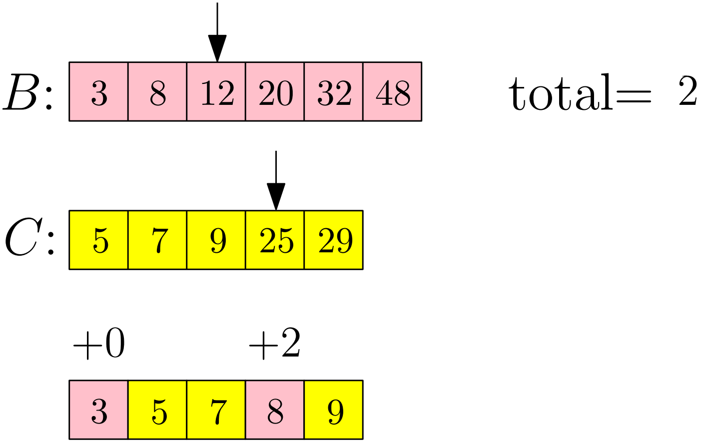
5	7	9	25	29

$+0$ $\quad$ $+2$

3	5	7	8	9

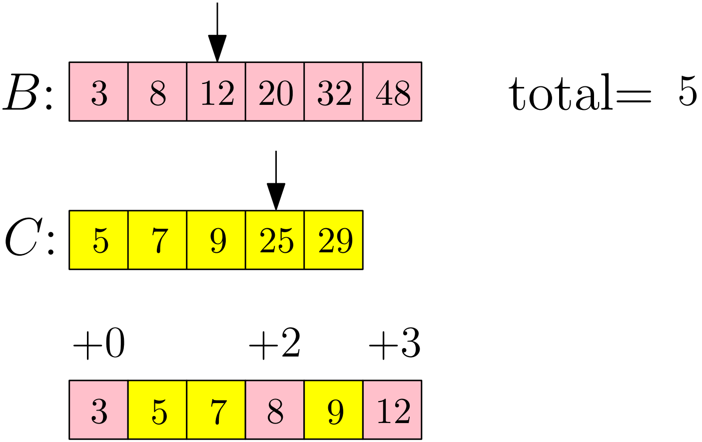
Counting Inversions between B and C

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



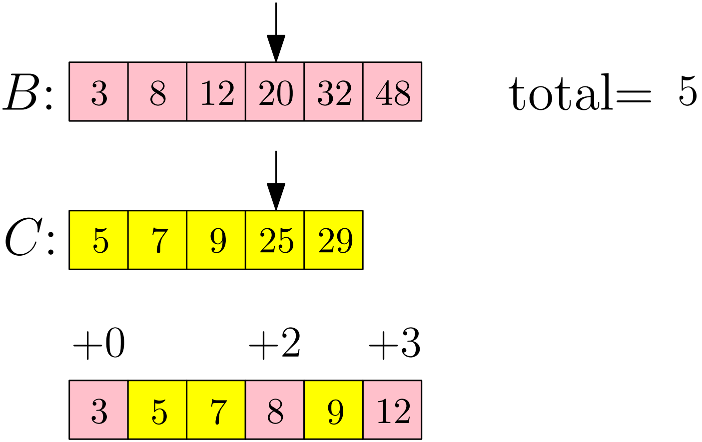
Counting Inversions between B and C

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



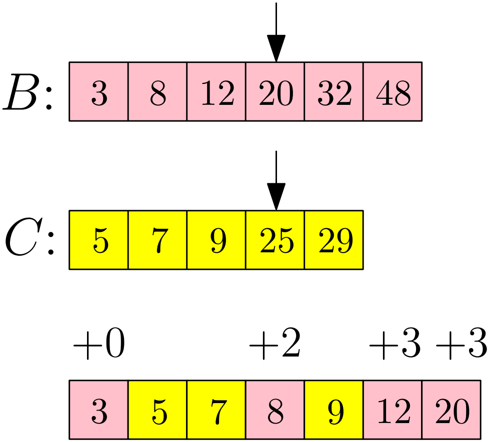
Counting Inversions between B and C

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



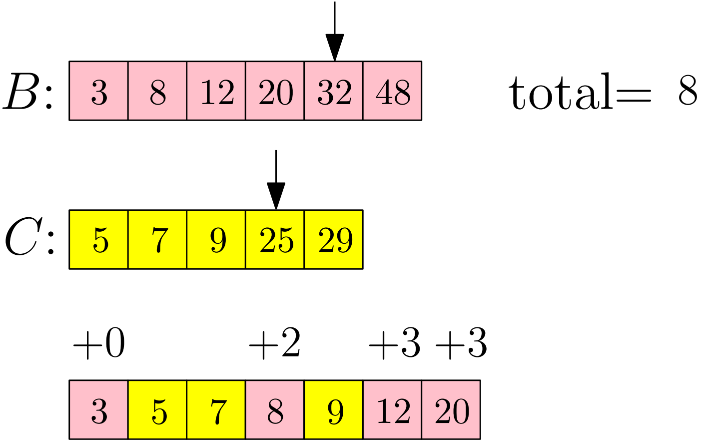
Counting Inversions between B and C

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



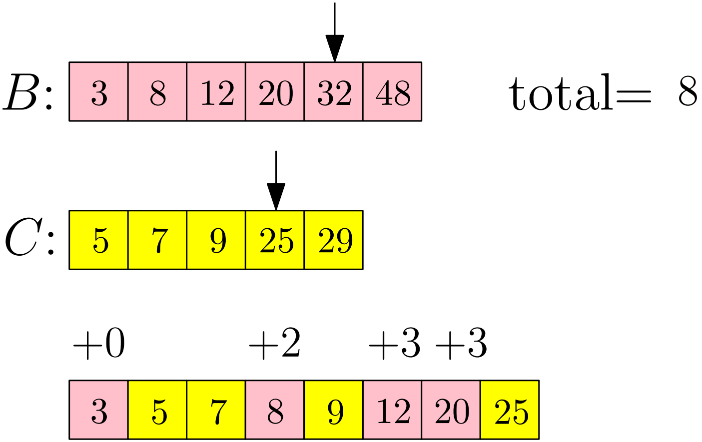
total= 8

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



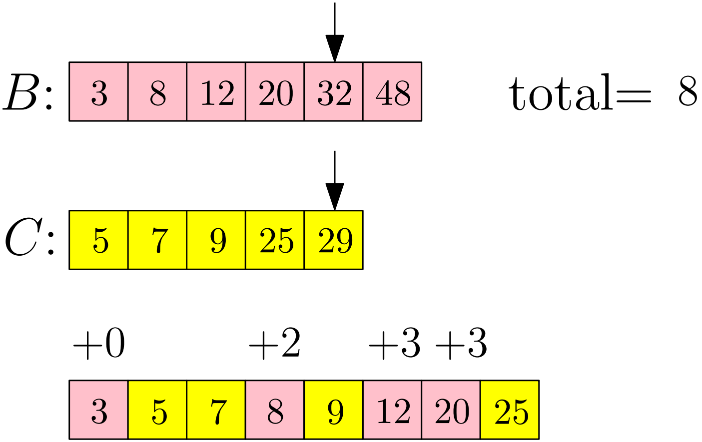
Counting Inversions between B and C

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	29

Counting Inversions between B and C

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

B:

3	8	12	20	32	48

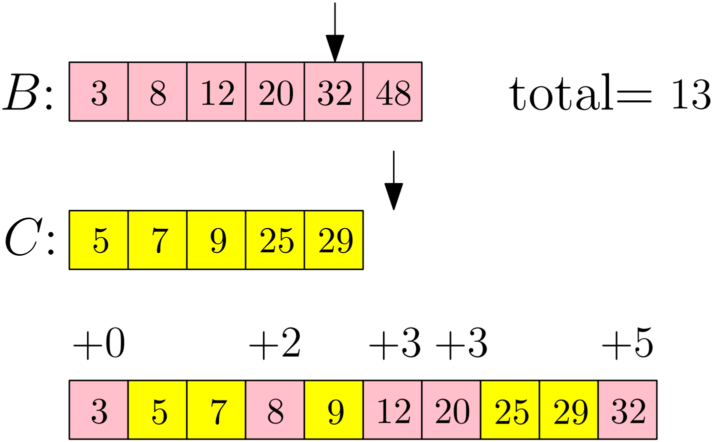
total= 8

C:

5	7	9	25	29

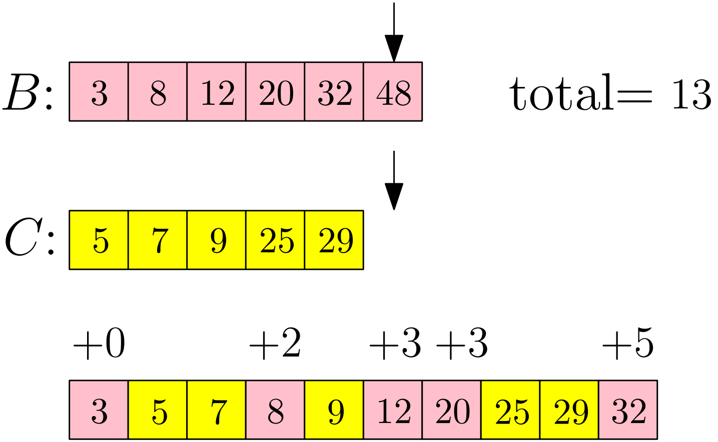
$+0$			$+2$		$+3$	$+3$		
3	5	7	8	9	12	20	25	29

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



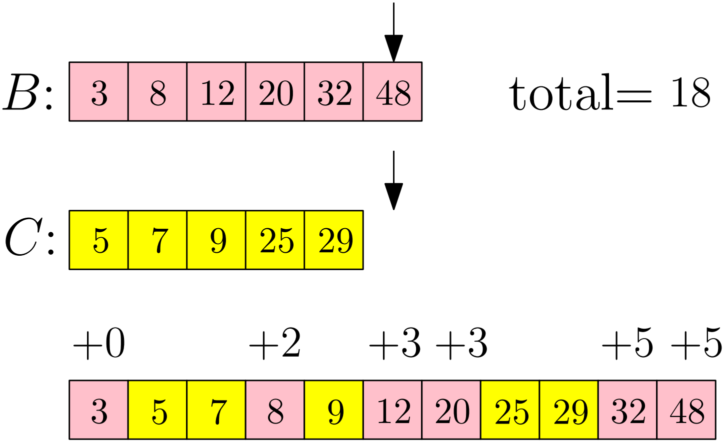
Counting Inversions between B and C

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



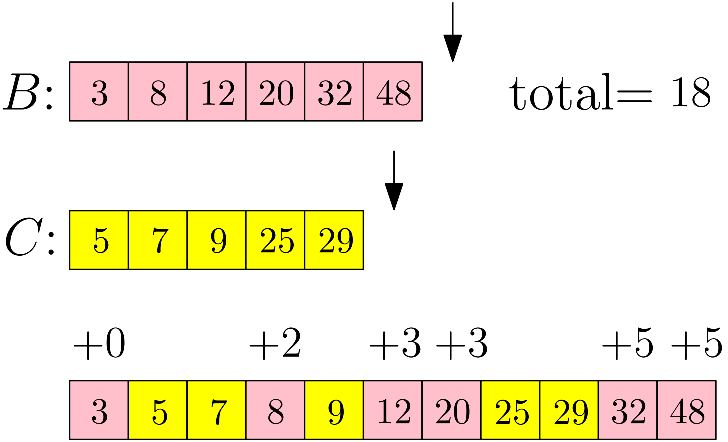
Counting Inversions between B and C

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) ← 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)
```

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) ← 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: $T(n) = 2T(n/2) + O(n)$

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

Outline

Outline

Quicksort vs Merge-Sort

	Merge Sort	**Quicksort**
Divide	Trivial	Separate small and big numbers
Conquer	Recurse	Recurse
Combine	Merge 2 sorted arrays	Trivial

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

25	15	17	29	38	45	37	64	82	75	94	92	69	76	85

Quicksort

quicksort(A, n)

if $n \leq 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ array of elements in A that are less than x \\ Divide
$A_R \leftarrow$ array of elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return concatenation of B_L, t copies of x, and B_R

Quicksort

quicksort(A, n)

if $n \leq 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ array of elements in A that are less than x \\ Divide
$A_R \leftarrow$ array of elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return concatenation of B_L, t copies of x, and B_R

- Recurrence $T(n) \leq 2T(n/2) + O(n)$

Quicksort

quicksort(A, n)

if $n \leq 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ array of elements in A that are less than x \\ Divide
$A_R \leftarrow$ array of elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return concatenation of B_L, t copies of x, and B_R

- Recurrence $T(n) \leq 2T(n/2) + O(n)$
- Running time $= O(n \lg n)$

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

A:

1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

A:

1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
2. Choose a pivot randomly and pretend it is the median (it is practical)

Quicksort Using A Random Pivot

quicksort(A, n)

if $n \le 1$ **then return** A
$x \leftarrow$ a random element of A (x is called a pivot)
$A_L \leftarrow$ array of elements in A that are less than x \\ Divide
$A_R \leftarrow$ array of elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return concatenation of B_L, t copies of x, and B_R

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

- In practice: use pseudo-random-generator, a deterministic algorithm returning numbers that "look like" random

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

- In practice: use pseudo-random-generator, a deterministic algorithm returning numbers that "look like" random
- In theory: assume they can.

Quicksort Using A Random Pivot

quicksort(A, n)

if $n \le 1$ **then return** A
$x \leftarrow$ a random element of A (x is called a pivot)
$A_L \leftarrow$ array of elements in A that are less than x \\ Divide
$A_R \leftarrow$ array of elements in A that are greater than x \\ Divide
$B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$ \\ Conquer
$B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$ \\ Conquer
$t \leftarrow$ number of times x appear A
return concatenation of B_L, t copies of x, and B_R

Lemma The expected running time of the algorithm is $O(n \lg n)$.

Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.