# Divide and conquer
## Lecture 06.02.

*by Marina Barsky*

# Main algorithm design strategies

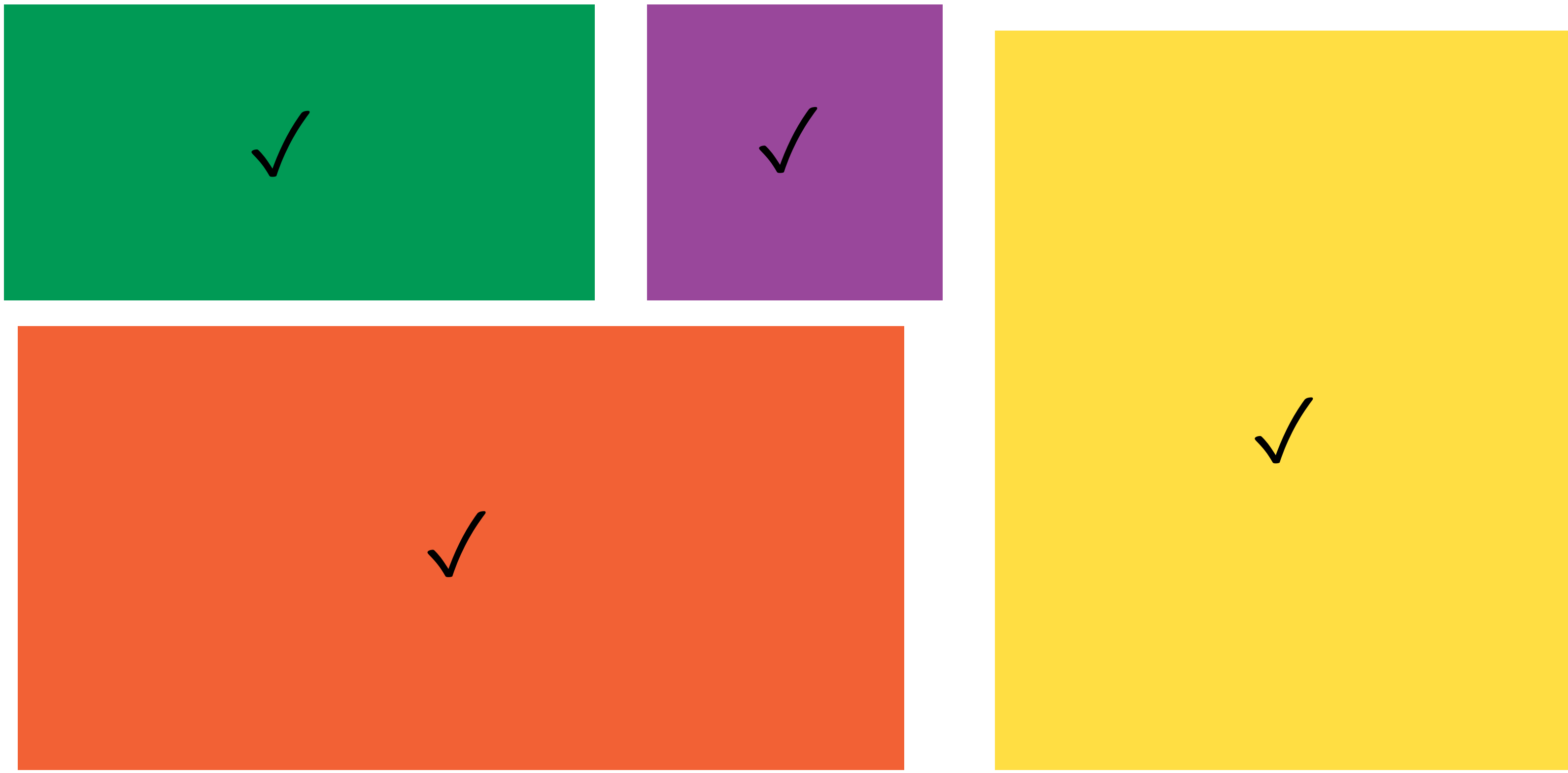
- ✓ ***Exhaustive Computation.*** Generate every possible candidate solution and select an optimal solution.
- ✓ ***Greedy. Create next candidate solution one step at a time by using some greedy choice.***
- ***Divide and Conquer.*** Divide the problem into non-overlapping subproblems of the same type, solve each subproblem with the same algorithm, and combine sub-solutions into a solution to the entire problem.
- ***Dynamic Programming.*** Start with the smallest subproblem and combine optimal solutions to smaller subproblems into optimal solution for larger subproblems, until the optimal solution for the entire problem is constructed.

# Divide-and-conquer technique

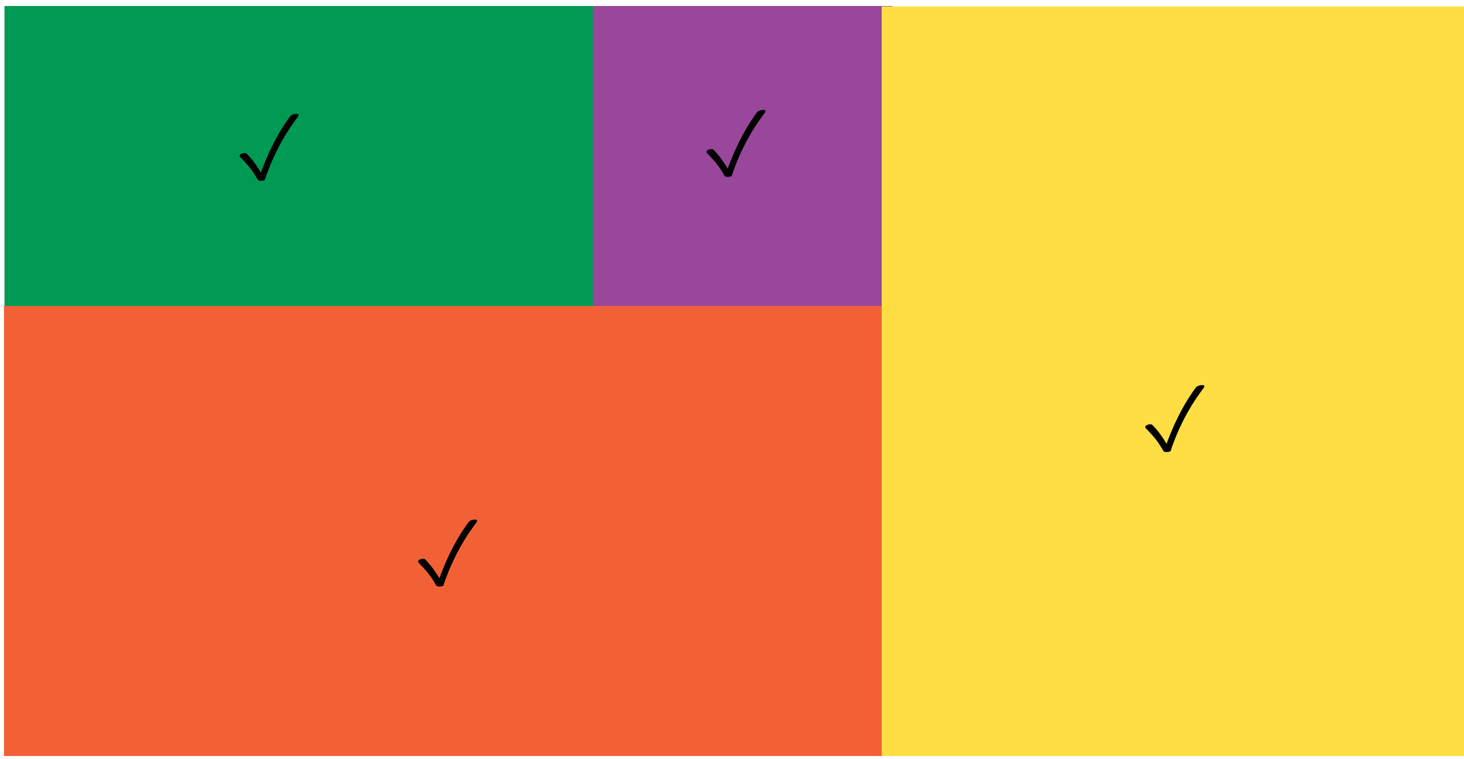
1. Break into *non-overlapping* subproblems *of the same type*
2. Solve subproblems
3. Combine results

**Divide**: break apart

**Conquer:** solve



# Combine



✓

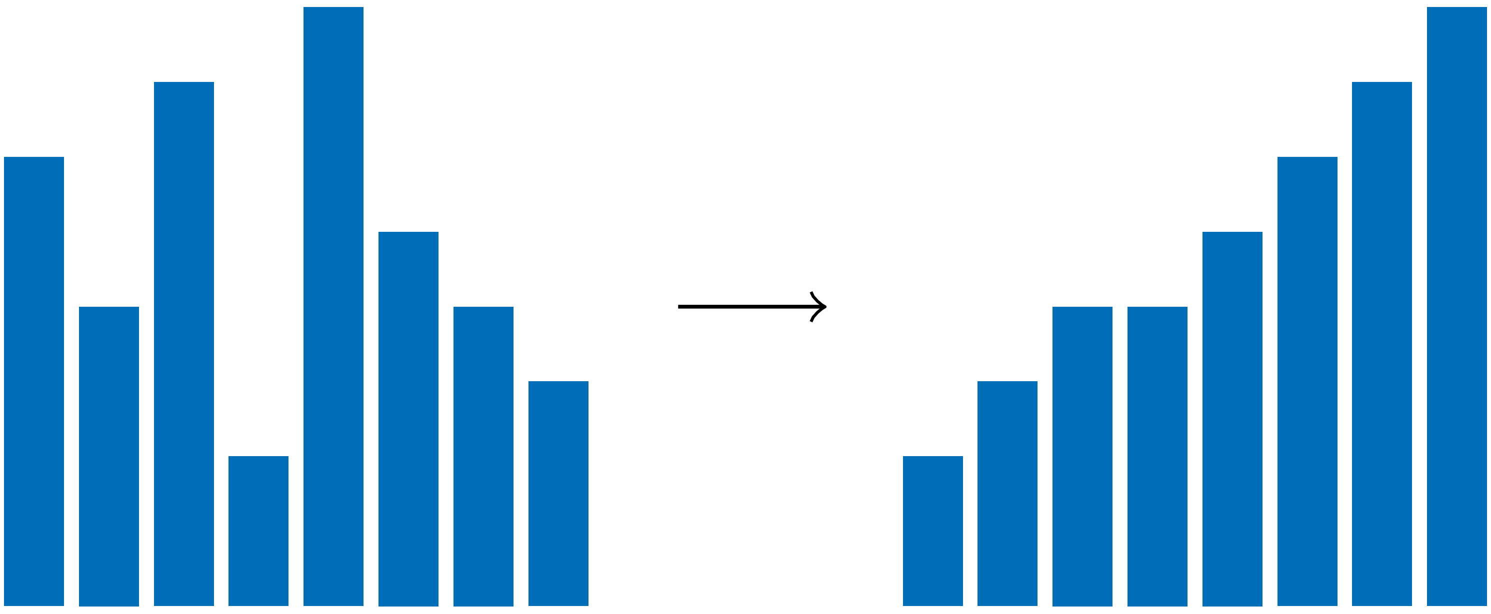
https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/sorting

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.toptal.com/developers/sorting-algorithms

# Sorting things

# Sorting Problem



## Sorting Problem

Input: Sequence *A* of n elements

Output: Permutation *A′* of elements in *A* such that all elements of *A′* are in non-decreasing order.

# Why Sorting?

- Sorting data is an important step of many efficient algorithms
- Sorted data allows for more efficient queries (binary search)

# Recap: merge sort

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 2 | 5 | 3 | 7 | 13 | 1 | 6 |

split the array into two halves

| | | | |
|---|---|---|---|
| 7 | 2 | 5 | 3 |

| | | | |
|---|---|---|---|
| 7 | 13 | 1 | 6 |

# merge sort

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 2 | 5 | 3 | 7 | 13 | 1 | 6 |

split the array into two halves

| | | | |
|---|---|---|---|
| 7 | 2 | 5 | 3 |

| | | | |
|---|---|---|---|
| 7 | 13 | 1 | 6 |

sort the halves recursively

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 7 |

| | | | |
|---|---|---|---|
| 1 | 6 | 7 | 13 |

# merge sort

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 2 | 5 | 3 | 7 | 13 | 1 | 6 |

split the array into two halves

| | | | |
|---|---|---|---|
| 7 | 2 | 5 | 3 |

| | | | |
|---|---|---|---|
| 7 | 13 | 1 | 6 |

sort the halves recursively

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 7 |

| | | | |
|---|---|---|---|
| 1 | 6 | 7 | 13 |

merge the sorted halves into one array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 7 | 7 | 13 |

## **Algorithm merge_sort (array $A[1...n]$)**

if $n = 1$: return $A$

$m \leftarrow \lfloor n/2 \rfloor$

$B \leftarrow$ merge_sort($A$[1 ... $m$])

$C \leftarrow$ merge_sort($A$[$m + 1$ ... $n$])

$A' \leftarrow$ merge($B$, $C$ )

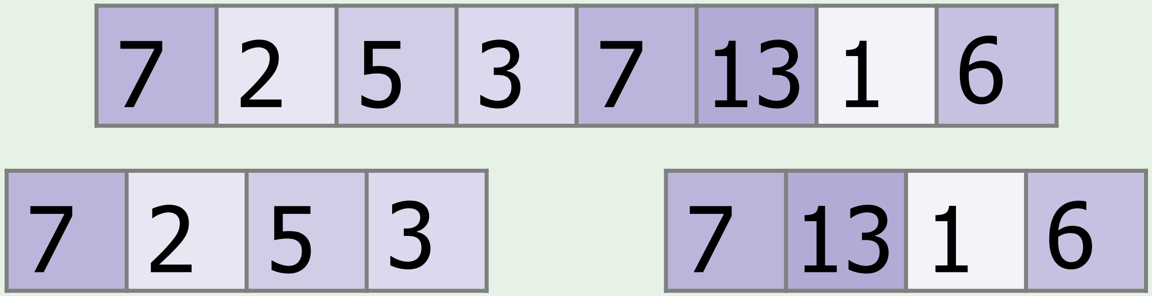
return $A'$

# Merging Two Sorted Arrays

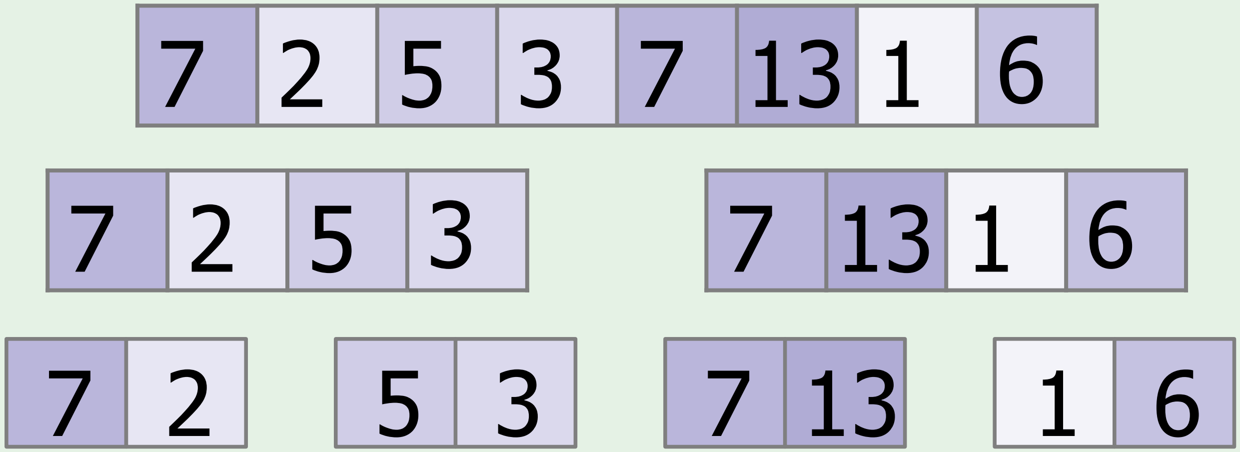
## Algorithm merge($B[1... p]$, $C[1... q]$)

```
# B and C are sorted
D ← empty array of size p + q
while B and C are both non-empty:
    b ← the first element of B
    c ← the first element of C
    if b ≤ c:
        move b from B to the end of D
    else:
        move c from C to the end of D
move what remains of B or C to the end of D
return D
```

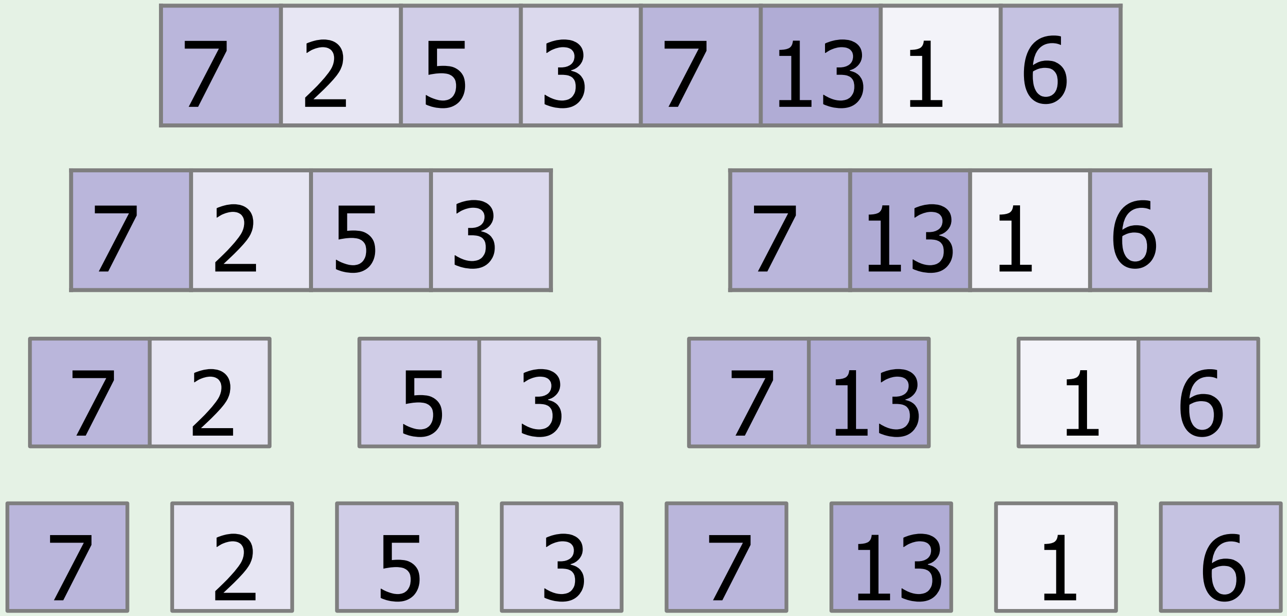
# Merge sort: example



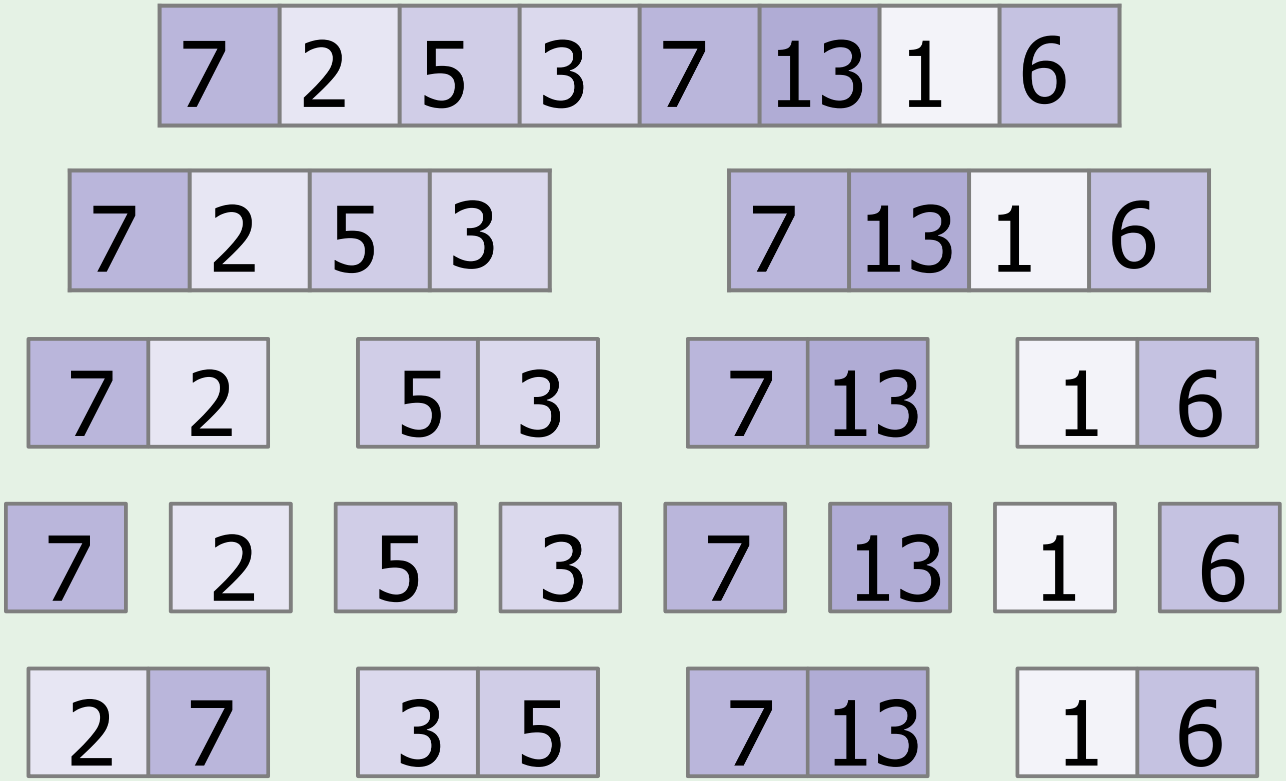
# Merge sort: example



# Merge sort: example

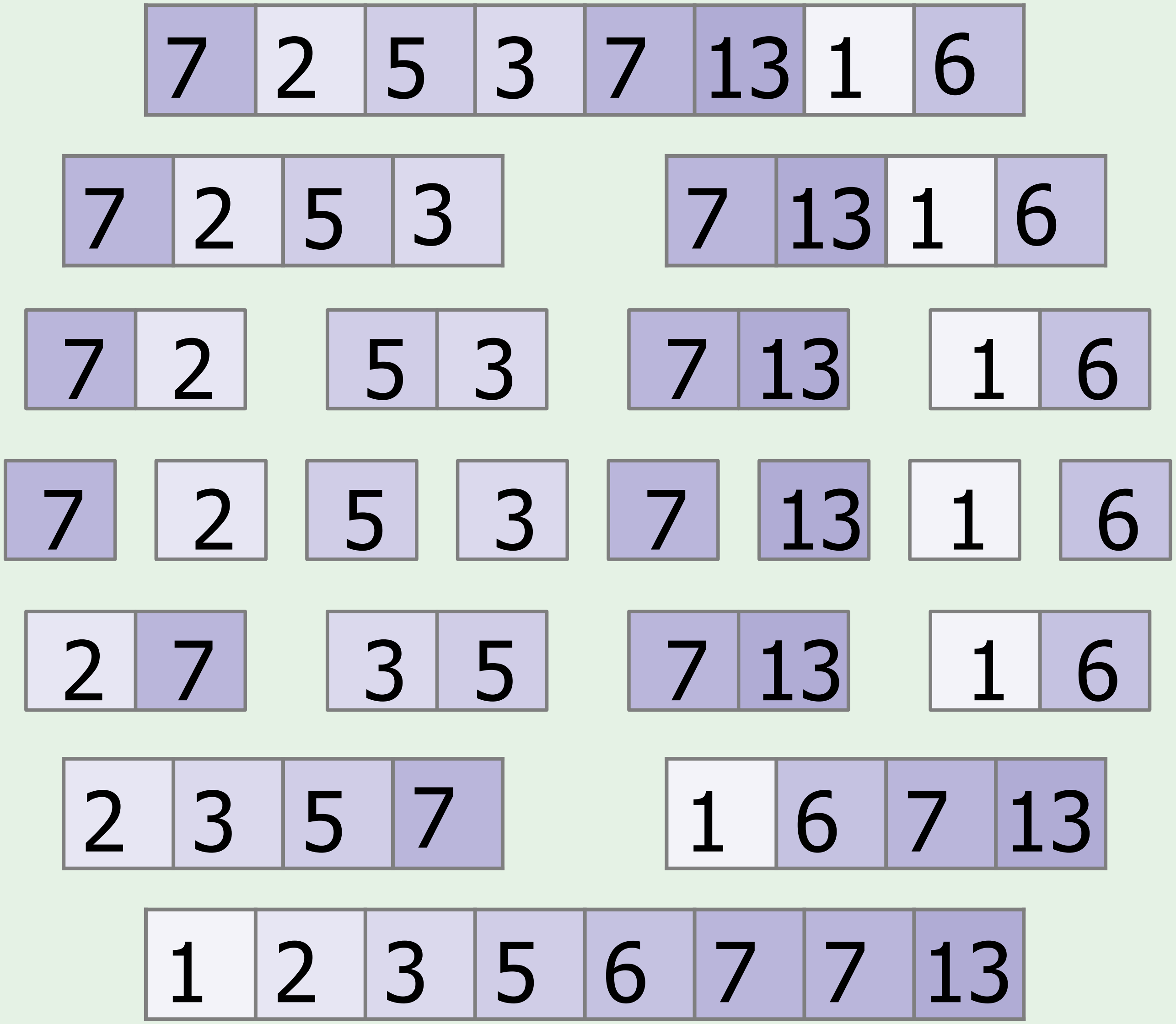


# Merge sort: example

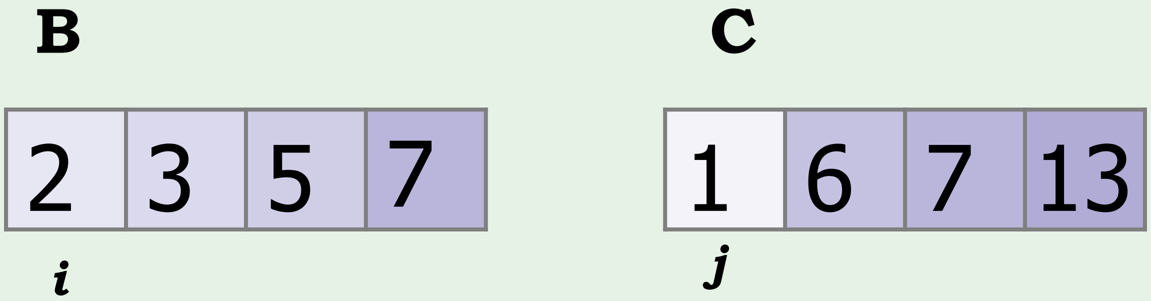


# Merge sort: example

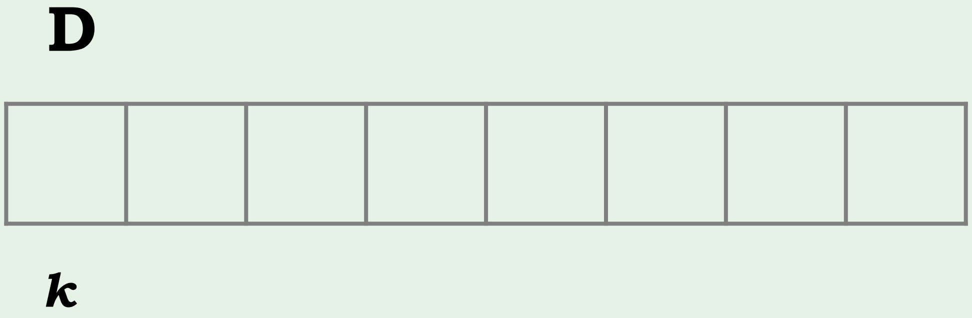
# Merge sort: example



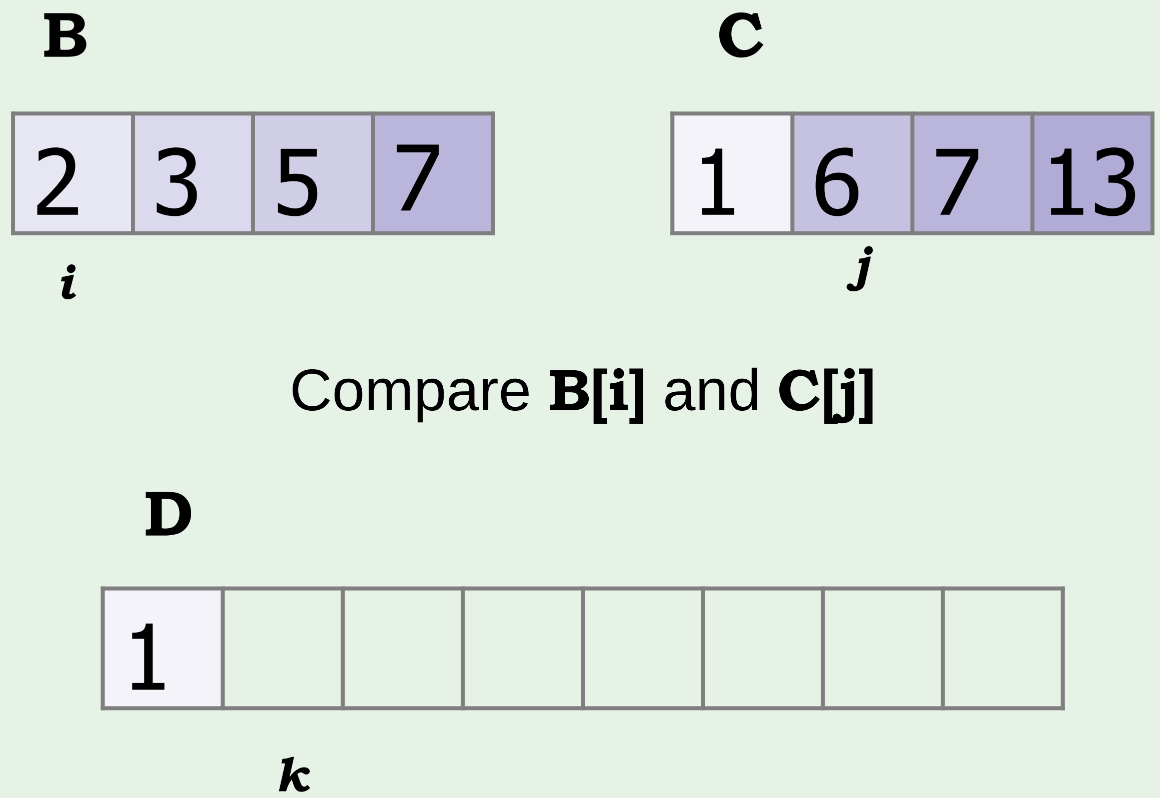
# Merge: example



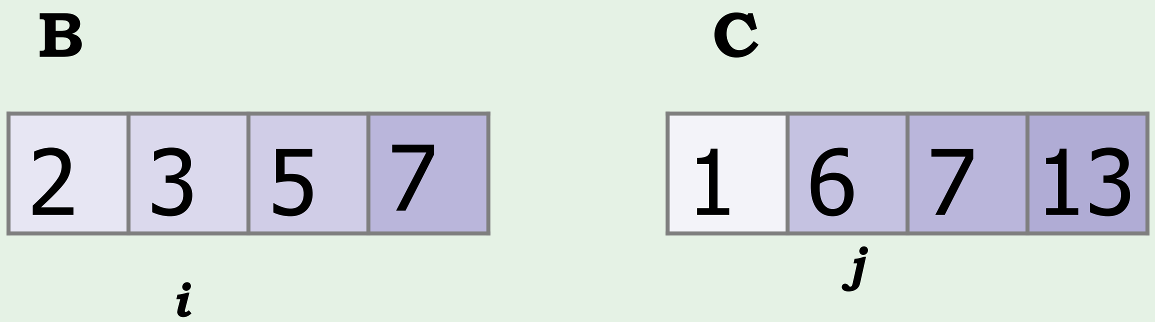

Compare **B[i]** and **C[j]**



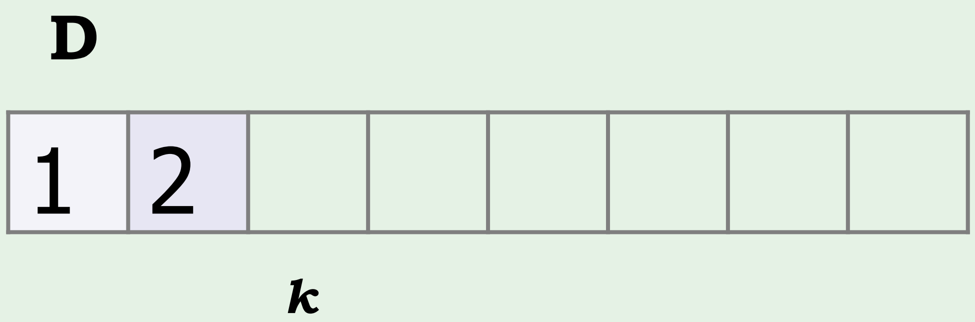
# Merge: example



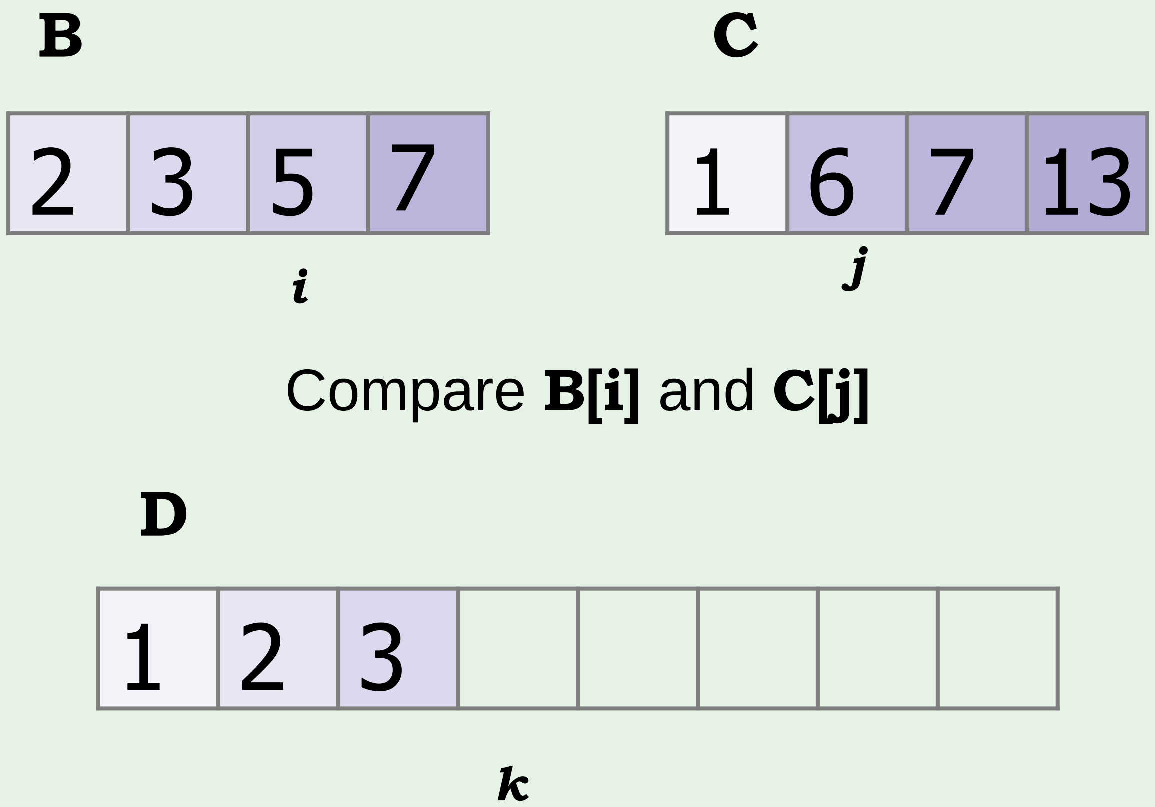
# Merge: example



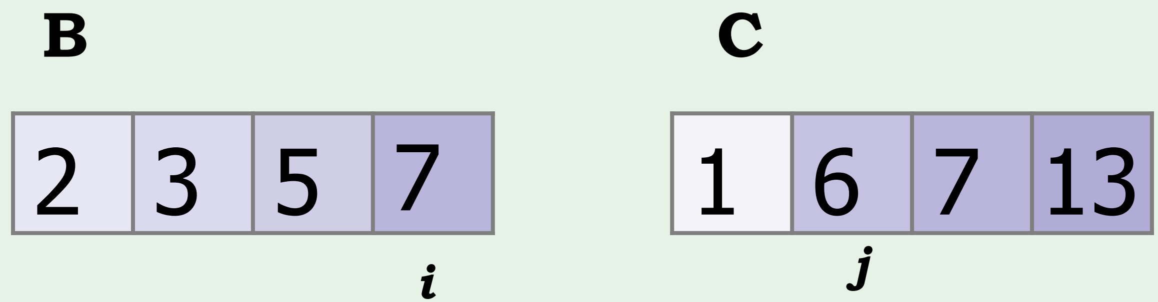

Compare **B[i]** and **C[j]**



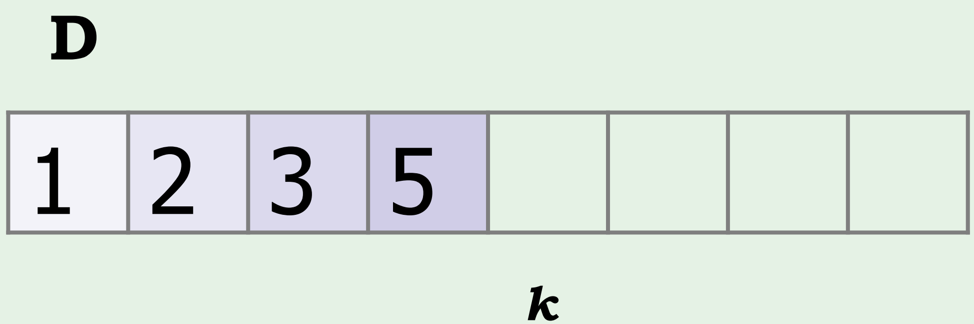
# Merge: example



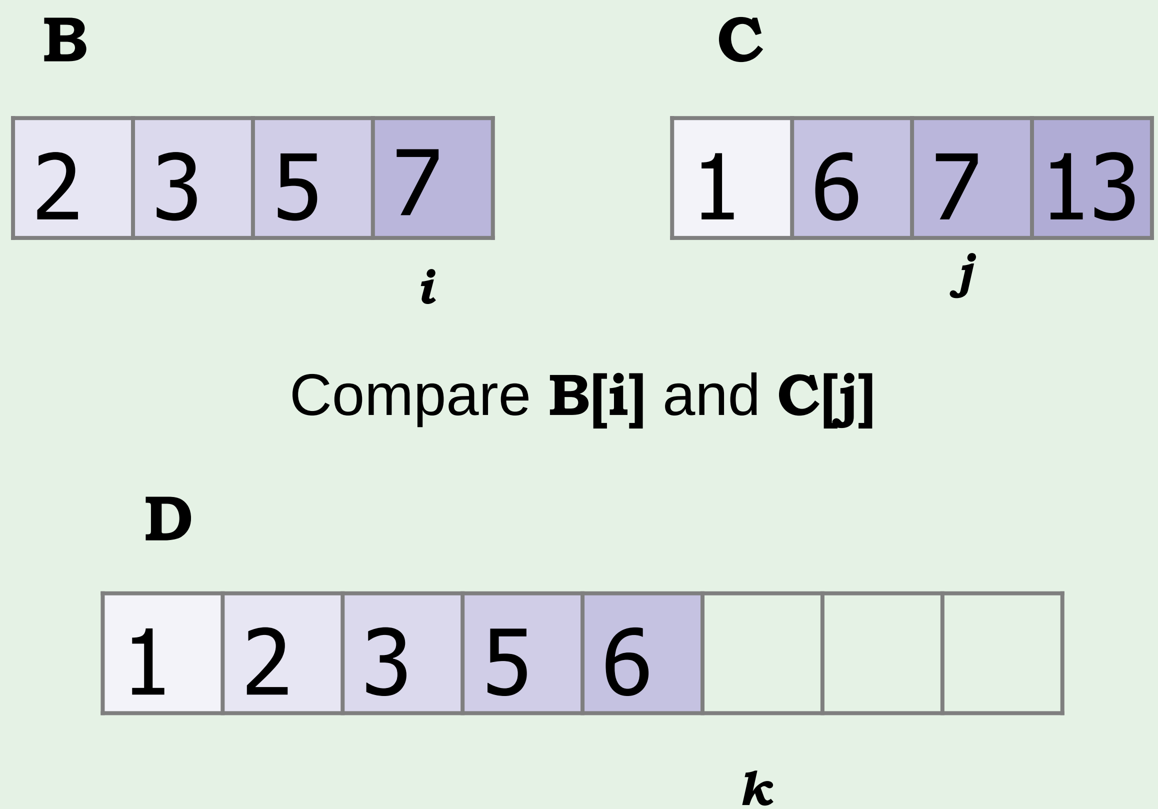
# Merge: example



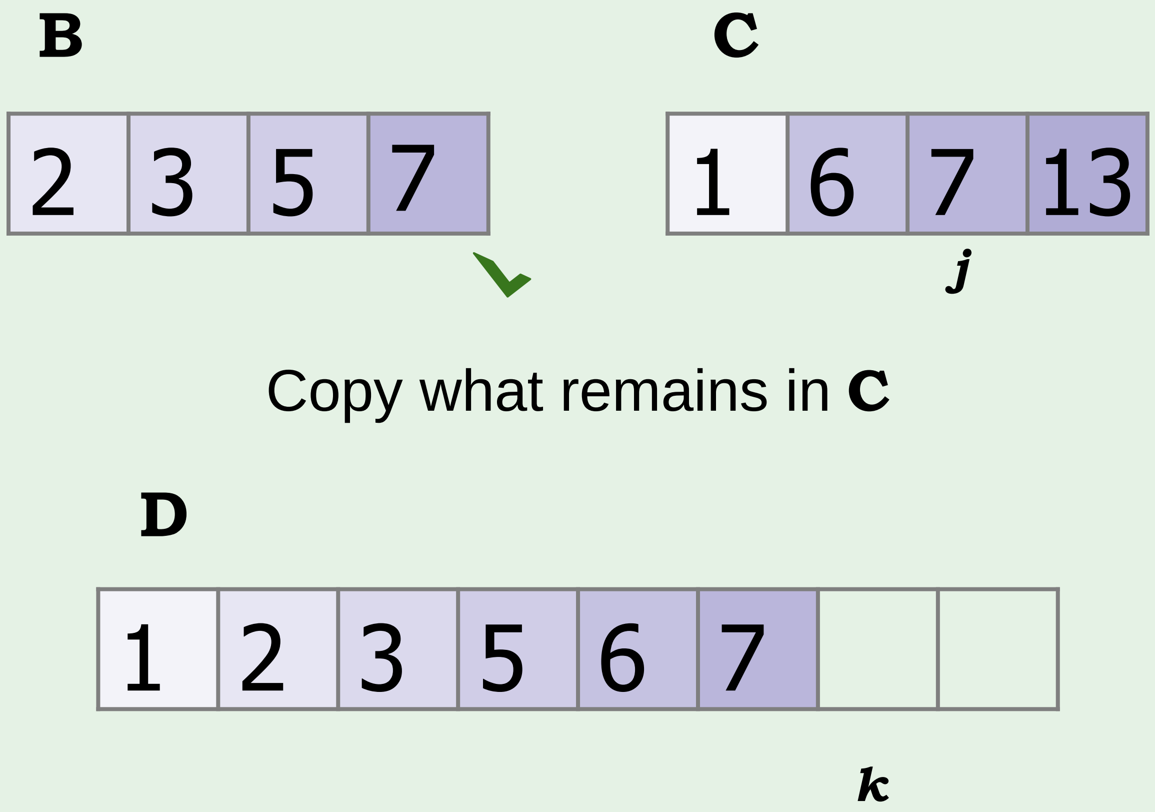

Compare **B[i]** and **C[j]**



# Merge: example



# Merge: example



# Merge: example

**B**

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 7 |

**C**

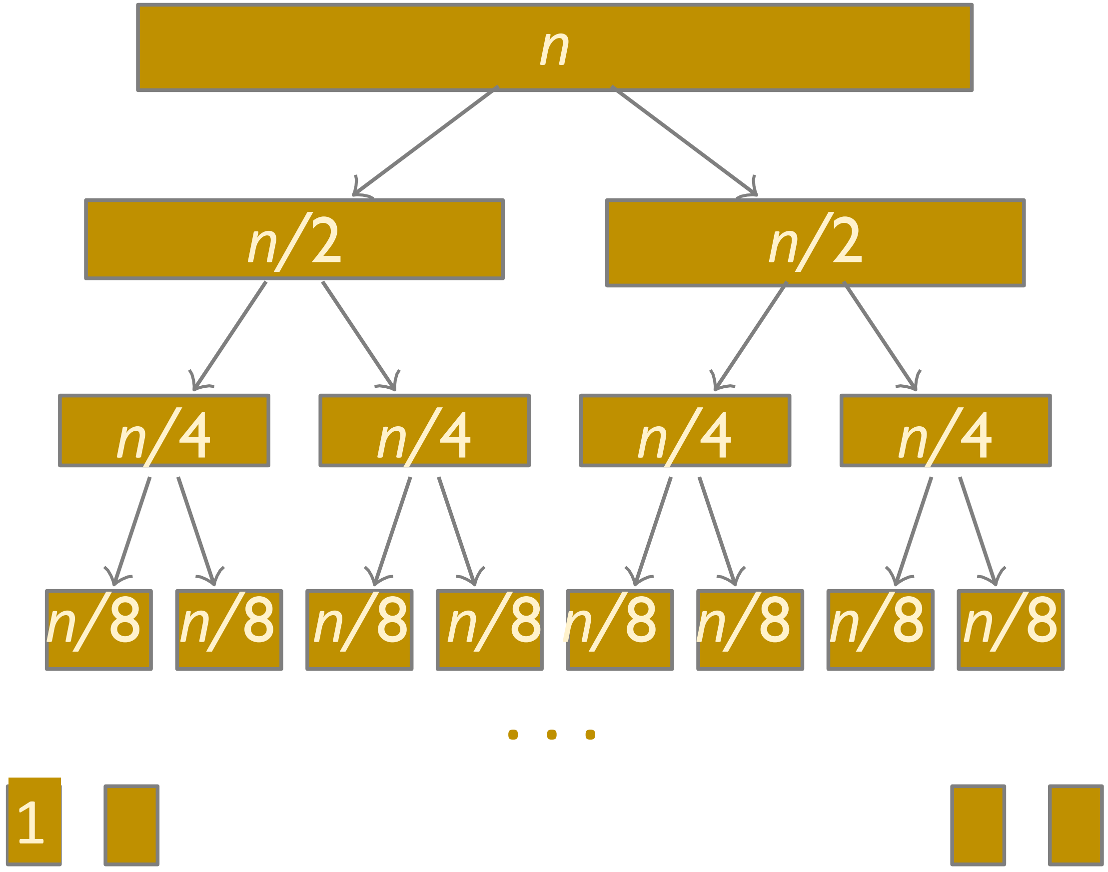
| | | | |
|---|---|---|---|
| 1 | 6 | 7 | 13 |

**D**

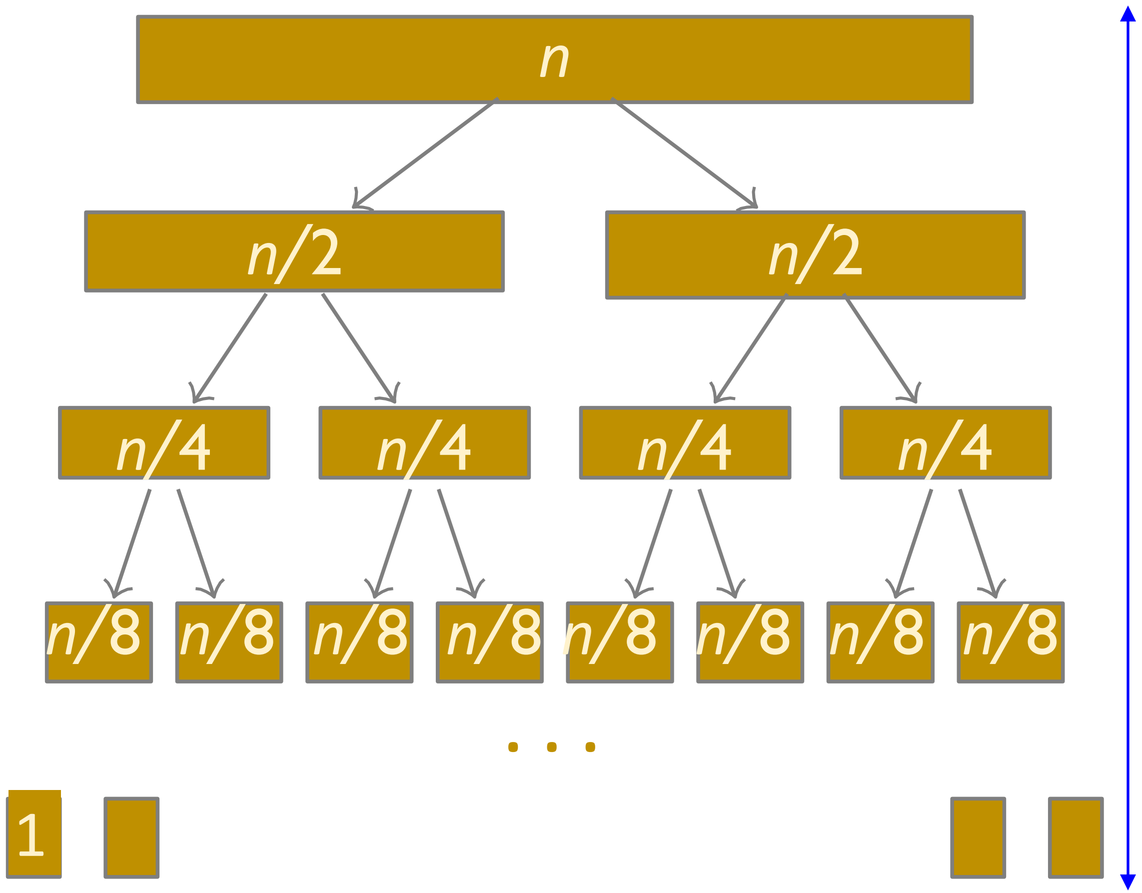
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 7 | 7 | 13 |

# Merge sort: running time

Subproblem size at each level

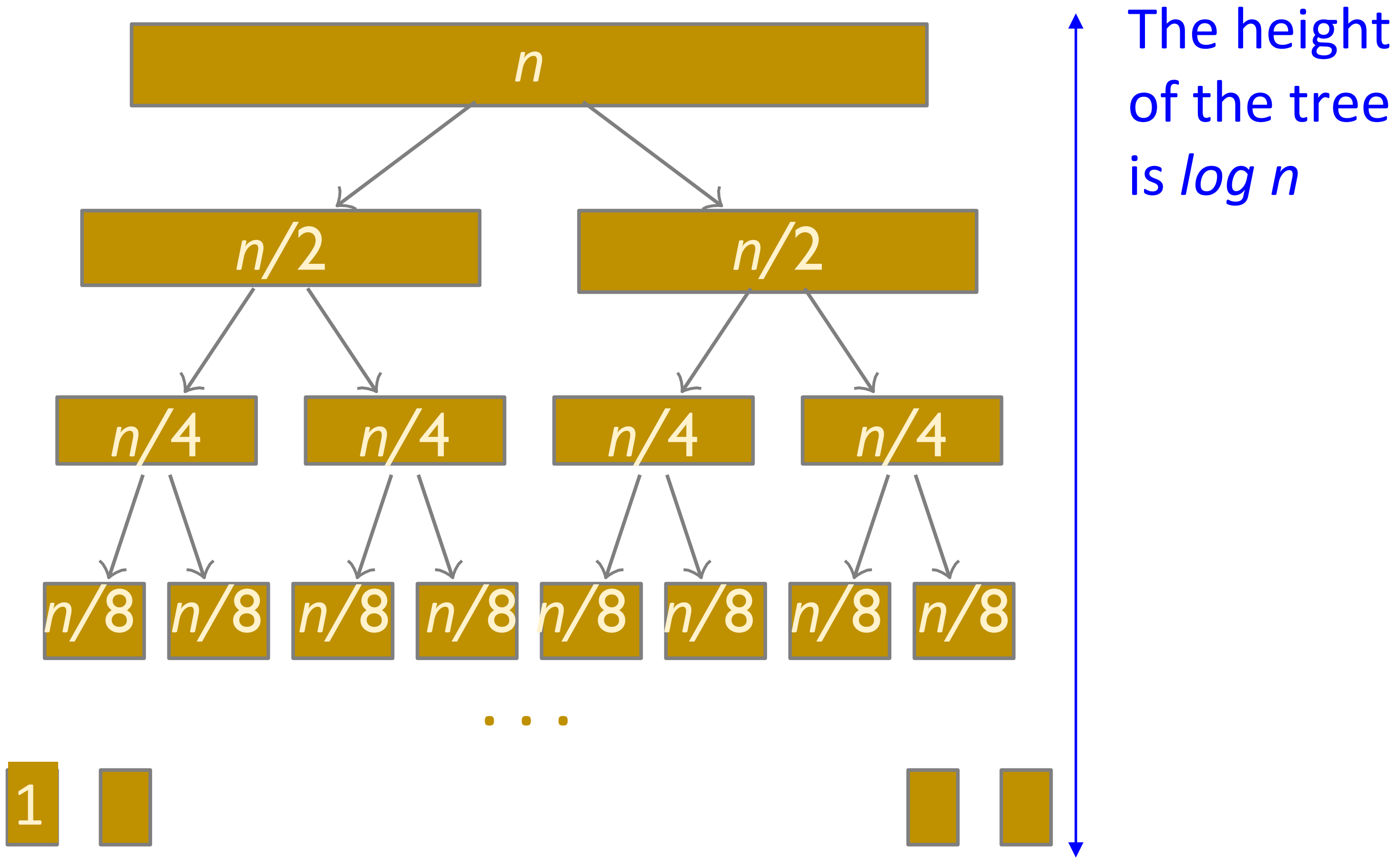


# Merge sort: recursion tree



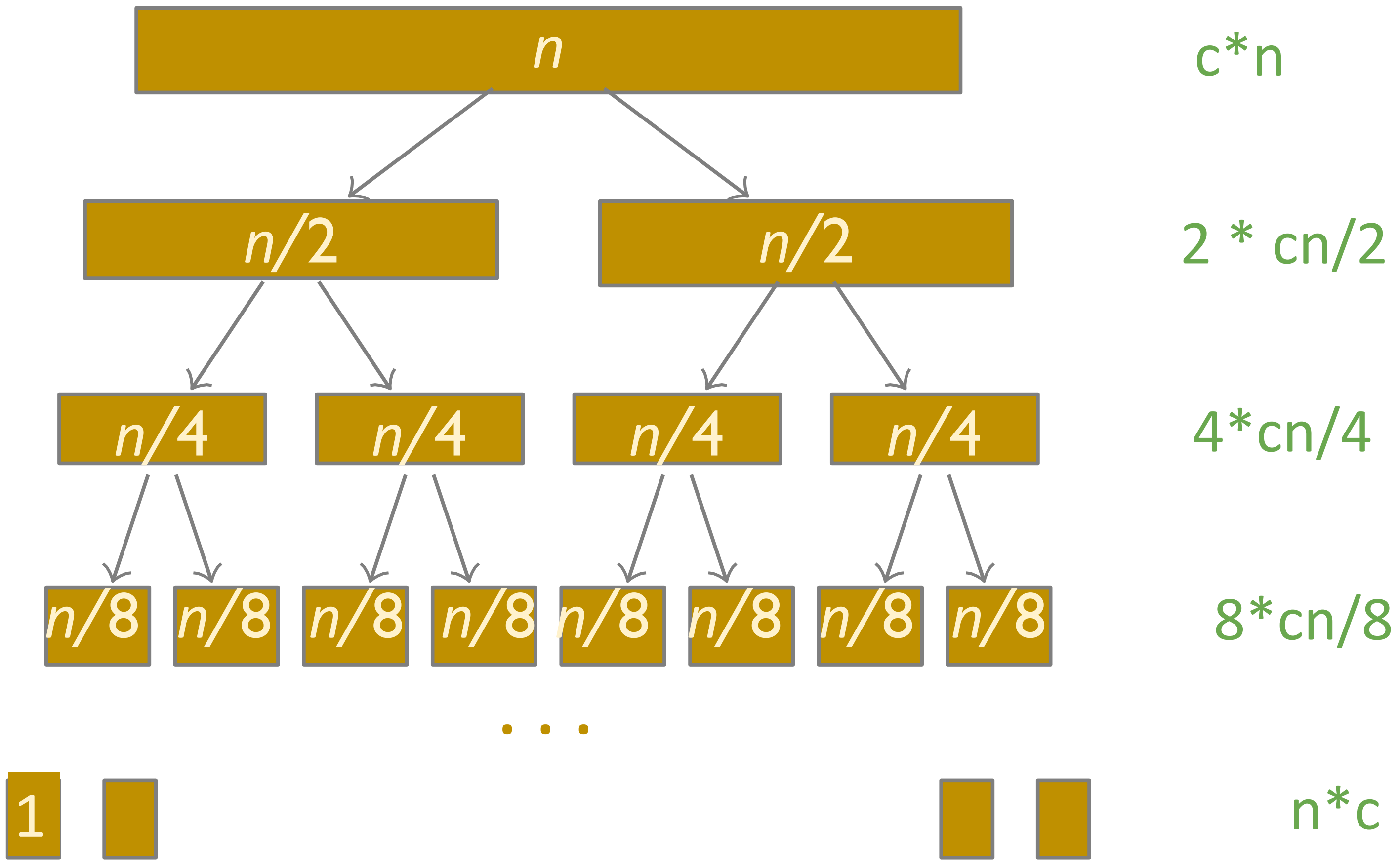

The height of the tree is...

# Merge sort: recursion tree



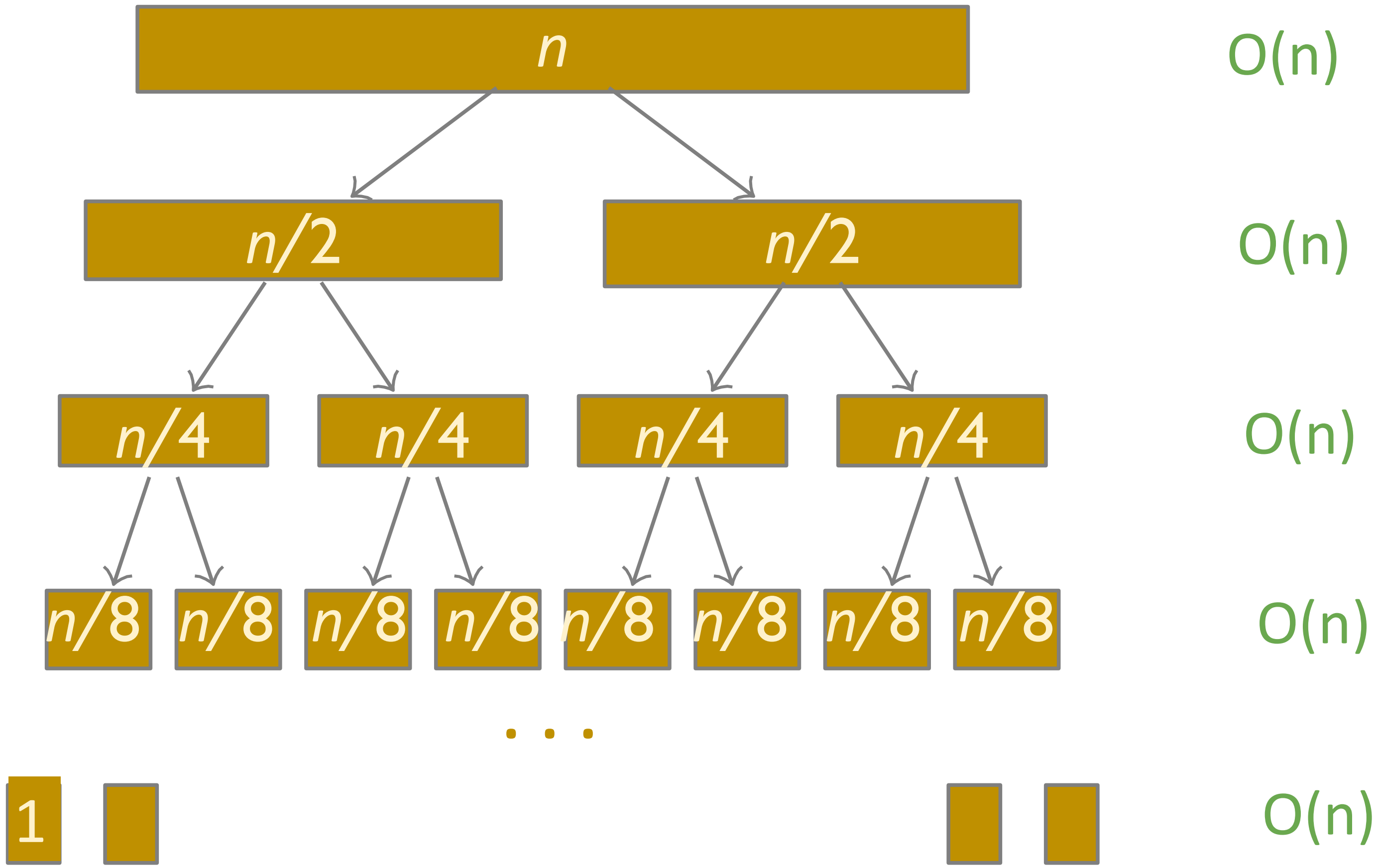
# Merge sort: recursion tree

Work at each level: **all the work is done during *merge***



# Merge sort: recursion tree

Work at each level: **O(n)**




Total: $O(n) * \log n = O(n \log n)$

## Algorithm merge_sort ($A[1...n]$)

if $n = 1$: return $A$

$m \leftarrow \lfloor n/2 \rfloor$

$B \leftarrow$ merge_sort($A[1 \ldots m]$)

$C \leftarrow$ merge_sort($A[m + 1 \ldots n]$)

$A' \leftarrow$ merge($B, C$)

return $A'$

The running time of merge_sort($A[1 \ldots n]$) is $O(n \log n)$.

## Merge Sort

The running time of **`MergeSort`**($A[1 . . . n]$) is $O(n \log n)$.

**Can we do better?**

# Lower bound for Comparison-Based Sorting

## Definition

A *comparison-based sorting* algorithm sorts objects by comparing pairs of them.

## Example

Selection sort and merge sort are comparison based.

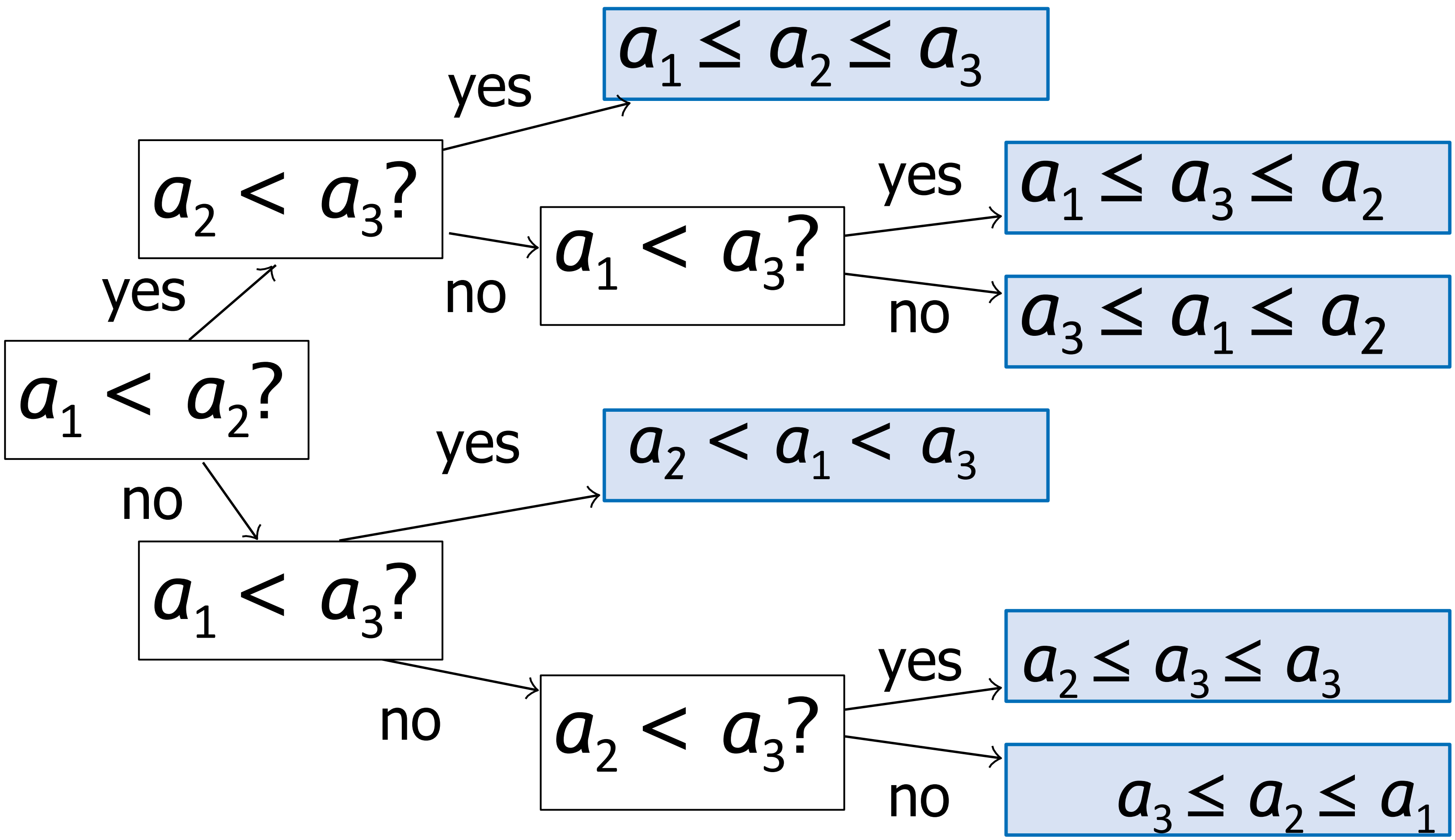
## Lemma

Any comparison-based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort $n$ objects.

## In other words

For any comparison-based sorting algorithm, there exists an input array $A[1 \ldots n]$ such that the algorithm performs **at least** $\Omega(n \log n)$ comparisons to sort $A$.

# Decision Tree
# for deciding the order of 3 objects



# Estimating max leaf depth

- The number of leaves ℓ in the tree must be $n!$ (the total number of permutations of $n$ array elements)

- For the worst-case input the number of comparisons made is equal to the maximum depth $d$ of this tree

- The max depth of any node in a binary tree with ℓ leaves is at least O(log ℓ): the minimum happens when the binary tree is complete. In all other incomplete binary trees the max depth will be > log ℓ.

$$d \geq \log_2 \ell \text{ (or, equivalently, } 2^d \geq \ell)$$

- The number of leaves ℓ in our decision tree is n!
- Let's show that:

$$\log_2(n!) = \Omega(n \log n)$$

## Lemma

$$\log_2(n!) = \Omega(n \log n)$$

## Proof

$$\log_2(n!) = \log_2(1 \cdot 2 \cdot \cdots \cdot n)$$

$$= \log_2 1 + \log_2 2 + \cdots + \log_2 n$$

$$\geq \log_2 (n/2) + \cdots + \log_2 n$$

$$\geq (n/2) \log_2(n/2) = \Omega(n \log n)$$

Consider only the second half of the sum

Consider only the smallest element of the sum

■

## Corollary

Any **comparison-based sorting** algorithm performs (at least) $\boldsymbol{\Omega(n \log n)}$ comparisons on the worst case input of size $n$.

## Merge Sort

The running time of **`MergeSort`**$(A[1\ .\ .\ .\ n])$ is $O(n \log n)$.

This running time is ***optimal*** if we consider **sorting based on comparing pairs of numbers**

# Sorting not based on comparison: can be faster

## Example: sorting small integers

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 2 | 3 | 2 | 1 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 1 |

# Non-comparison based sorting

## Sorting small integers

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 2 | 3 | 2 | 1 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 1 |

| | 1 | 2 | 3 |
|---|---|---|---|
| Count | 2 | 7 | 3 |

# Non-comparison based sorting

## Sorting small integers

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 2 | 3 | 2 | 1 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 1 |

| | 1 | 2 | 3 |
|---|---|---|---|
| Count | 2 | 7 | 3 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |

# Non-comparison based sorting

## Sorting small integers

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 2 | 3 | 2 | 1 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 1 |

we have sorted these numbers without actually comparing them!

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |

# Count Sort

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to $M$.
- By a single scan of the array $A$, count the number of occurrences of each $1 \leq k \leq M$ in the array $A$ and store it in $Count[k]$.
- Using this information, fill in the sorted array $A'$.

# Algorithm count_sort($A[1 \ldots n]$)

$A'[1 \ldots n] \leftarrow [0, \ldots, 0]$ # to store sorted values of $A$
$Count[1 \ldots M] \leftarrow [0, \ldots, 0]$
for $i$ from 1 to $n$:
    $Count[A[i]] \leftarrow Count[A[i]] + 1$

# number $k$ appears $Count[k]$ times in $A$
$Pos[1 \ldots M] \leftarrow [0, \ldots, 0]$
$Pos[1] \leftarrow 1$
for $j$ from 2 to $M$ :
    $Pos[j] \leftarrow Pos[j - 1] + Count[j - 1]$

# number $k$ will occupy range $[Pos[k] \ldots Pos[k + 1] - 1]$
for $i$ from 1 to $n$:
    $A'[Pos[A[i]]] \leftarrow A[i]$
    $Pos[A[i]] \leftarrow Pos[A[i]] + 1$

return $A'$

## Lemma

Provided that all elements of $A[1 \mathinner{.\,.\,.} n]$ are integers from 1 to $M$, **`count_sort`**($A$) sorts $A$ in time $O(n + M)$.

## Note

If $M = O(n)$, then the running time is $O(n)$.

# Summary on sorting

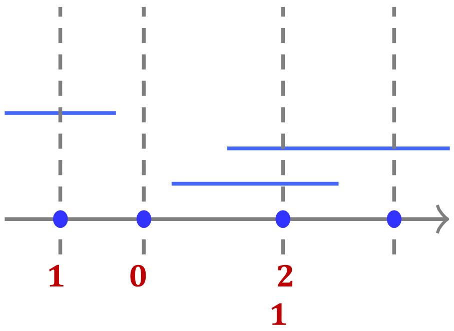
- Merge sort uses the divide-and-conquer strategy to sort an $n$-element array in time $O(n \log n)$
- No comparison-based algorithm can do this (asymptotically) faster
- One **can** do faster if something special is known about the input in advance (e.g., it contains small integers)

# Application of sorting: points and segments

**Activity**

# Points and Segments Problem

Given a set of points and a set of segments on a line, compute, for each point, the number of segments it is contained in.




**Points and segments in one dimension**

Input: A set of $S$ segments and a set of $P$ points.

Output: For each point - the number of segments it is contained in.

# Sample input and output

**Input:**

(0, 5)

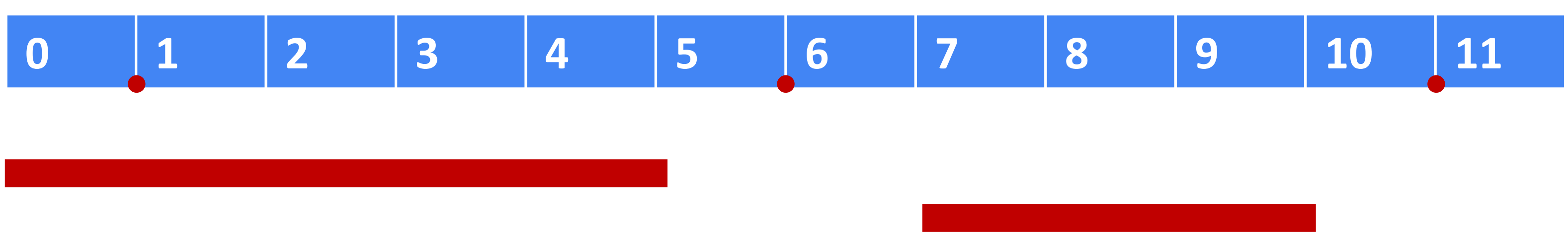
(7, 10)

2 segments

1, 6, 11 — 3 points

**Output:**

1 0 0 — Number of covering segments for each point



## Points and Segments Problem

Input: A set of *S* segments and a set of *P* points.

Output: For each point - the number of segments it is contained in.

What a naïve algorithm would do?

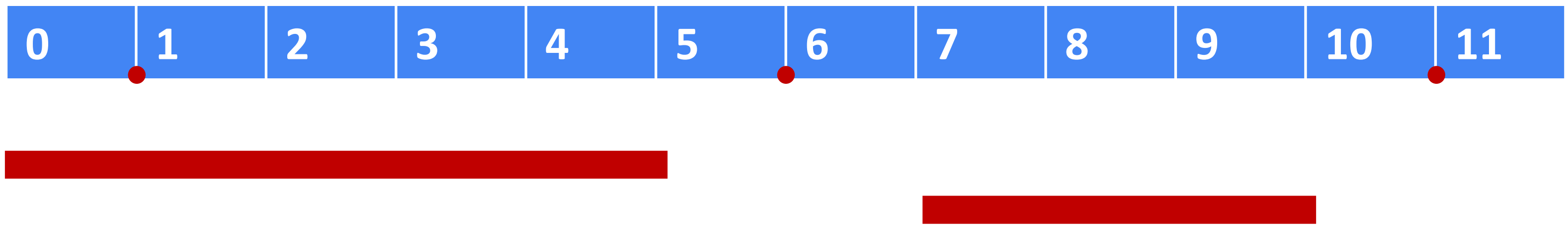
What is the complexity of the naïve algorithm?

Let N = S + P

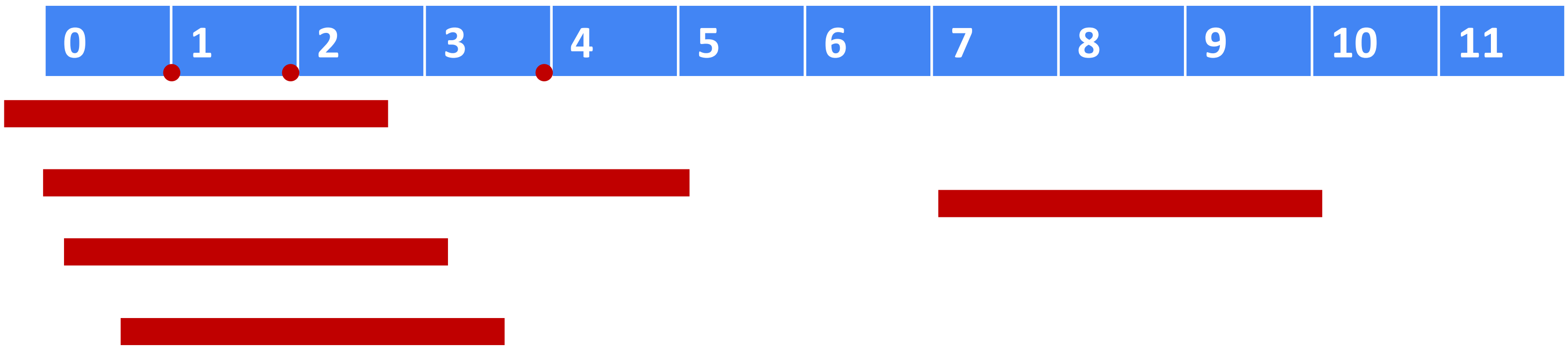
**The goal**: use O(N log N) sorting algorithm

After that, solve the problem in O(N) steps

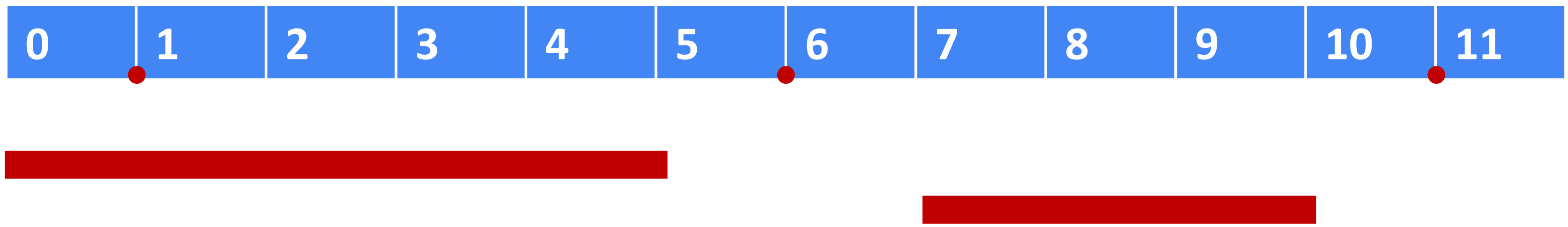
Ideas? Can sorting help?



Ideas? Can sorting help? What should we sort?



Ideas? Can sorting help? What should we sort?




What if we could sort everything together?

(0, start), (1, point), (5, end), (6, point), (7, start), (10, end), (11, point)

Do you see a linear-time solution now?