COM1009
Introduction to Algorithms and Data Structures

Topic 04: Divide-and-Conquer

Reading: Section 2.3
(optional: lots more details and advanced material in Chapter 4)

Aims of this lecture

- To introduce the divide-and-conquer design paradigm.
- To introduce the MergeSort algorithm – a recursive algorithm using divide-and-conquer.
- To show how to prove correctness of a recursive algorithm
- To show how to analyse the runtime of a recursive algorithm using recurrence equations.

Design Paradigms

- InsertionSort used an incremental approach:
  - Having sorted the subarray A[1..j-1], we inserted A[j] into its proper place, yielding the sorted subarray A[1..j].
  - **Idea:** **incrementally build up** a solution to the problem.
- Alternative design approach: **divide-and-conquer**
  1. **Divide:** Break the problem into smaller subproblems, smaller instances of the original problem.
  2. **Conquer:** Solve these problems recursively.
  3. **Combine** the solutions to subproblems into the solution for the original problem.
- Coming up: correctness and runtimes for recursive algorithms.

MergeSort

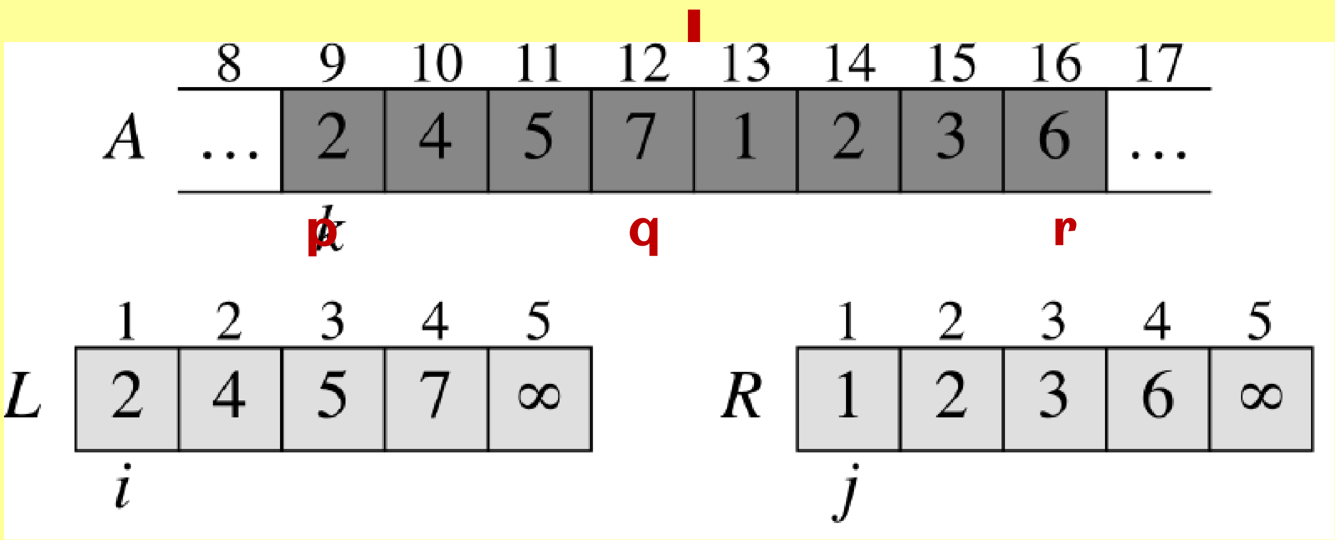
- MergeSort - sorting using **divide-and-conquer:**

  1. **Divide** the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
  2. **Conquer:** Sort the two subsequences recursively using MergeSort.
  3. **Combine:** merge the two subsequences to produce the sorted answer.

- The recursion stops when the sequence is just 1 element.
- The key here is the procedure Merge
- Tedious bit: copying elements between arrays.

Merge(A, p, q, r)

- Assume subarrays $A[p \dots q]$and $A[q+1 \dots r]$ are sorted.
- Copy these subarrays to new arrays L and R.
- Both L and R contain an additional element ∞ at the end (“sentinel”), so we don’t have to check for end of array.
- Merge L and R back into A



MERGE(A, p, q, r)

Runtime = $\Theta(n) + \Theta(n)$. $\Theta(1) = \Theta(n)$

```
n₁ = q − p + 1
n₂ = r − q
let L[1 … n₁ + 1] and R[1 … n₂ + 1] be new arrays
for i = 1 to n₁ do
        L[i] = A[p + i − 1]
for j = 1 to n₂ do
        R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
i = 1
j = 1
for k = p to r do
        if L[i] ≤ R[j] then
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
```

$\Theta(n)$

Set up arrays L and R (boring)

Iterate $\Theta(n)$ times

$\Theta(1)$ each time

Actual merge

What does this look like in practice? (see Mike's Videos)

Correctness of Merge: The Invariant

- **Loop invariant:** At the start of the k'th iteration of the merge
  - the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order and
  - $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A.

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2						…
		p			k				r	

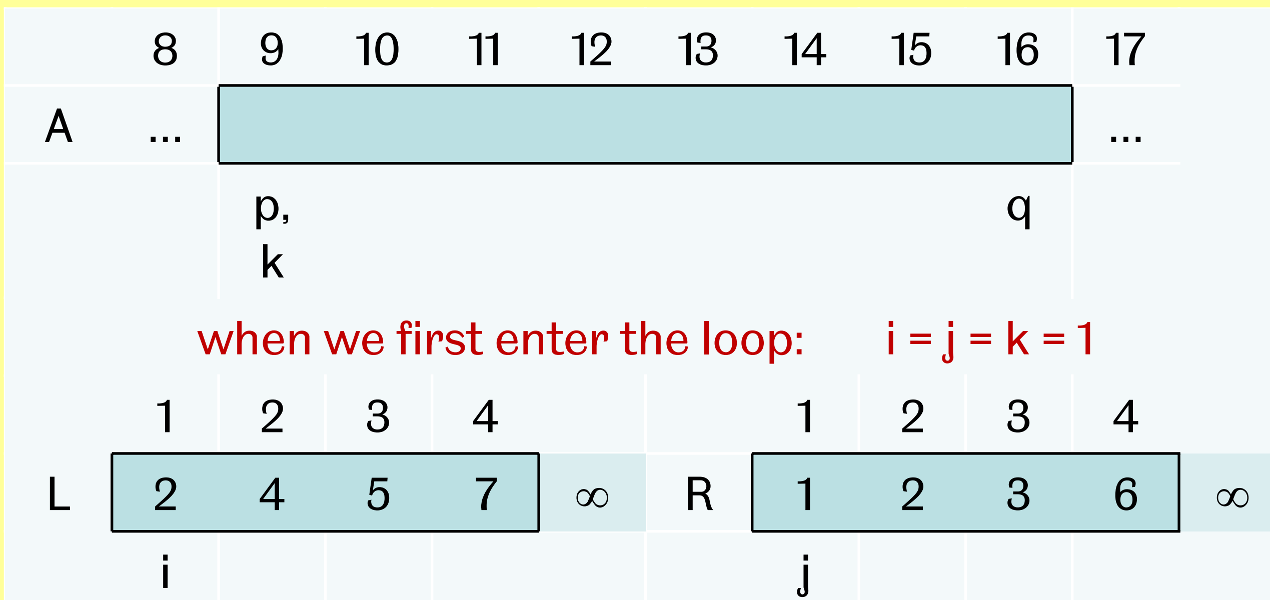
Merging L and R back into A

	1	2	3	4	
L	2	4	5	7	∞
		i			

	1	2	3	4	
R	1	2	3	6	∞
			j		

▶Correctness of Merge: Initialisation

- **Loop invariant:** the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order; and $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A. ✓



Correctness of Merge: Maintenance

	8	9	10	11	12	13	14	15	16	17
A	...	1	2	2						...
		p			k				r	

We start the k'th iteration ...

	1	2	3	4			1	2	3	4	
L	2	4	5	7	∞	R	1	2	3	6	∞
		i							j		

end of
one iteration
=
start of
next iteration

	8	9	10	11	12	13	14	15	16	17
A	...	1	2	2	3					...
		p				k			r	

... which inserts the k'th value

	1	2	3	4			1	2	3	4	
L	2	4	5	7	∞	R	1	2	3	6	∞
		i								j	

▶Correctness of Merge: Termination

- **Loop invariant:** the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order; and $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A.

	8	9	10	11	12	13	14	15	16	17
A	...	1	2	2	3	4	5	6	7	...
		p							r	k

The loop exits when k = r+1

	1	2	3	4	
L	2	4	5	7	∞
					i

	1	2	3	4	
R	1	2	3	6	∞
					j

And now everything has been sorted correctly ✓

MergeSort: The Complete Algorithm

Notation: $\lfloor x \rfloor$ means "floor of *x*" (rounding down).

$\textsc{MergeSort}(A, p, r)$

1: **if** $p < r$ **then**
2: $\quad q = \lfloor (p + r)/2 \rfloor$
3: $\quad \textsc{MergeSort}(A, p, q)$
4: $\quad \textsc{MergeSort}(A, q + 1, r)$
5: $\quad \textsc{Merge}(A, p, q, r)$

Initial call: $\textsc{MergeSort}(A, 1, \mathrm{A.length})$

MergeSort: Runtime Analysis

- Looking for time *T(n):* time for MergeSort to sort *n* elements.
- Assume for simplicity that n is an exact power of 2.

$\textsc{MergeSort}(A, p, r)$	Time
if $p < r$ **then**	$\Theta(1)$
$q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
$\textsc{MergeSort}(A, p, q)$	*T(n/2)*
$\textsc{MergeSort}(A, q+1, r)$	*T(n/2)*
$\textsc{Merge}(A, p, q, r)$	$\Theta(n)$

Time for MergeSort to sort *n/2* elements.

- Yields a **recurrence equation** where *T(n)* depends on *T(n/2)*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 2^0 = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

- "The time for MergeSort to sort *n* elements is twice the time for MergeSort to sort *n/2* elements plus $\Theta(n)$ time (for Merge)."

How to Solve a Recurrence Equation

$$T(n) = \begin{cases} d & \text{if } n = 2^0 \\ 2T(n/2) + cn & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

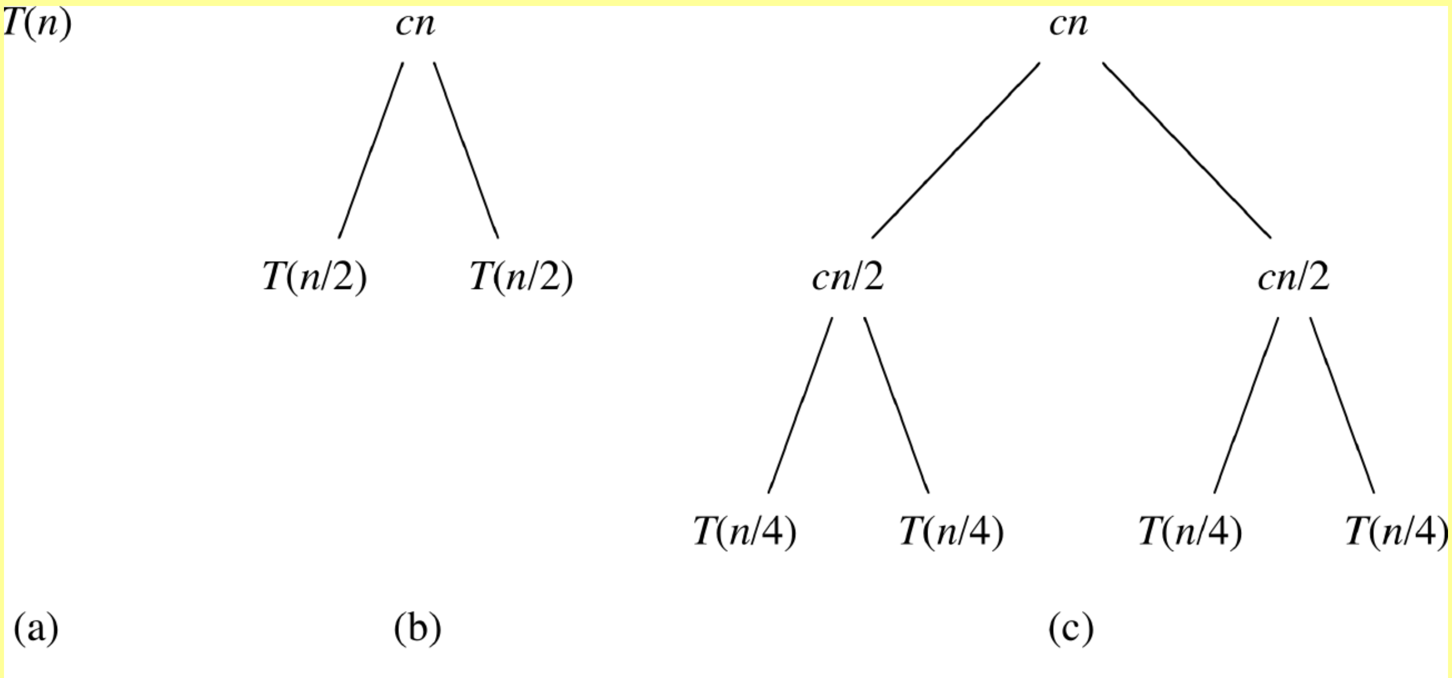
1. **Substitution method** (Sec 4.3): guess a solution and verify using **induction** (over *k*).
   - Tutorial exercise.
2. Draw a **recursion tree** (Sec 4.4), add times across the tree.
3. Use the **Master Theorem** (Sec 4.5) to solve a general recurrence equation in the shape of:

$$T(n) = aT(n/b) + f(n).$$

(this is out of the scope of this module and not examinable)

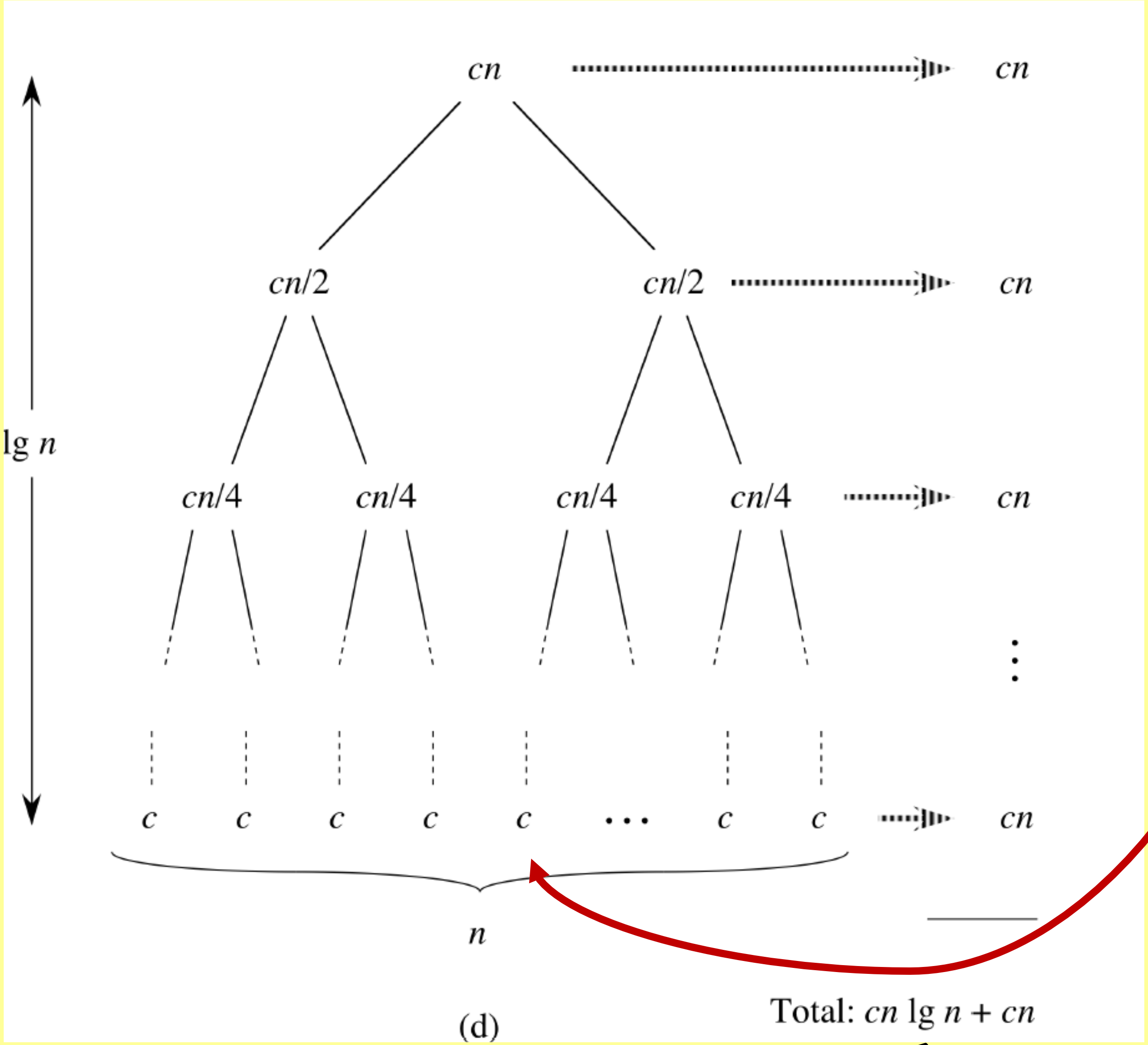
▶Runtime Visualised as Recursion Tree (*d=c*)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 2^0 = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n = 2^k \text{, for } k \geq 1 \end{cases}$$



$$T(n) = \begin{cases} d & \text{if } n = 2^0 \\ 2T(n/2) + cn & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

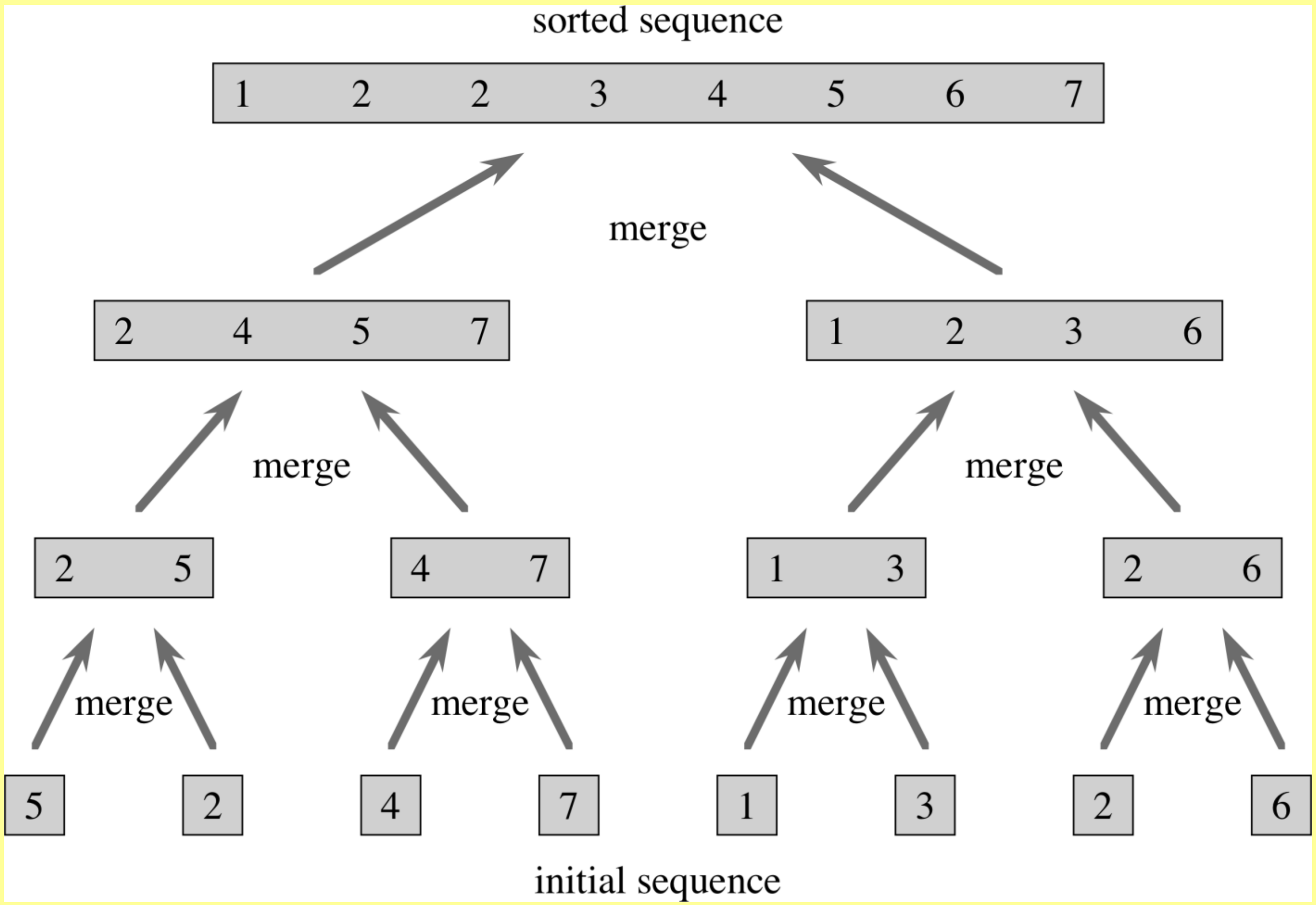
(in our case d ≈ c)



“lg” here means $\log_2$

▶Another way of seeing it:

$\theta(\log n)$ times



$\theta(n)$

$\theta(n)$

$\theta(n)$

Comparison with InsertionSort

- MergeSort always runs in time $\Theta(n \log n)$.
- Way better than worst case and average case of $\Theta(n^2)$ for InsertionSort.
- Worse than the best-case time $\Theta(n)$ of InsertionSort.
  - InsertionSort might be faster if your array is almost sorted.
- MergeSort needs **more space** than InsertionSort:
  - MergeSort always stores $\Omega(n)$ elements outside the input.
  - InsertionSort only needs $O(1)$ additional space.
  - We say that InsertionSort sorts **in place**:

A sorting algorithm sorts **in place**
if it only uses $O(1)$ additional space.

Summary

- The divide-and-conquer design paradigm
  - **Divides** a problem into smaller sub-problems of the same kind
  - **Solves** these sub-problems recursively, and then
  - **Combines** these solutions to an overall solution.
- MergeSort uses divide-and-conquer to sort in time $\Theta(n \log n)$ (best case = worst case).
- It's possible to sort n elements in worst-case time $\Theta(n \log n)$.
- Drawback: MergeSort does not sort in place.
  - "In place": sorting using only $O(1)$ additional space.
- The runtime of recursive algorithms can be analysed by solving a **recurrence equation**.