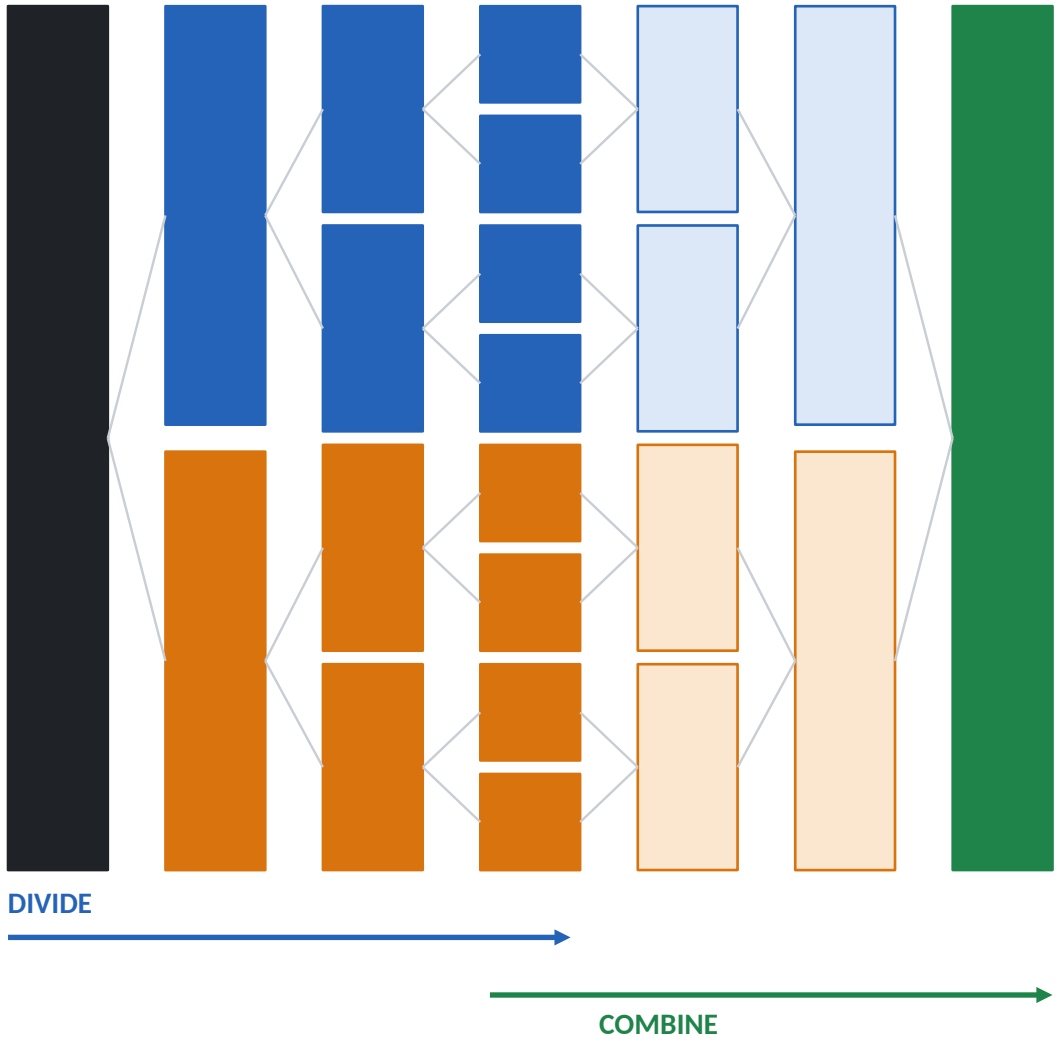
DATA STRUCTURES AND ALGORITHMS II

Divide and Conquer Algorithms

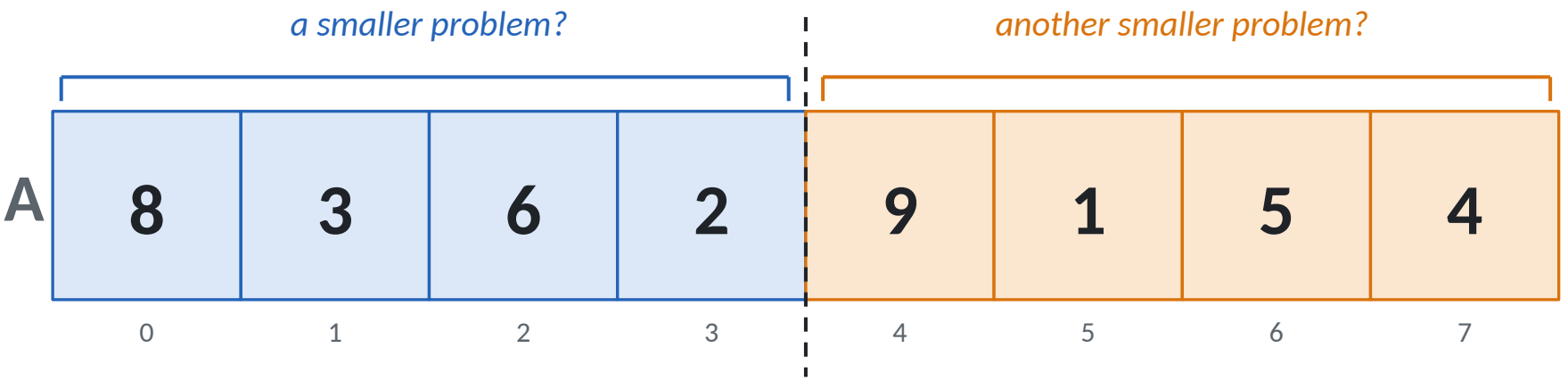
Breaking Complex Problems into Smaller, Solvable Problems

Sheikh Tanvir Ahmed
Lecturer
Department of Computer Science and Engineering (CSE)
United International University (UIU)



Start with a problem

Suppose we need to find the **minimum** and the **maximum** element of this array.



> **Can we make the problem easier by breaking it into smaller parts?**

What is Divide and Conquer?

Divide and Conquer is an algorithmic paradigm in which a problem is **divided** into smaller subproblems, those subproblems are **solved recursively**, and their solutions are **combined** to obtain the solution to the original problem.

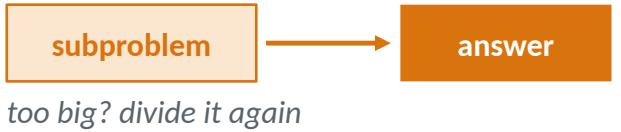
1 DIVIDE

Break the original problem into smaller subproblems.



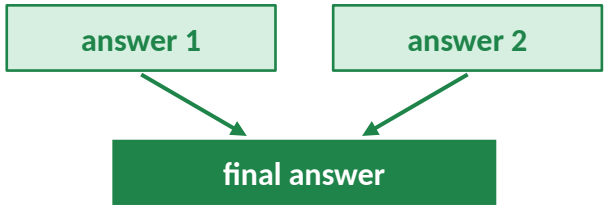
2 CONQUER

Solve the smaller subproblems recursively. If a subproblem is small enough, solve it directly.



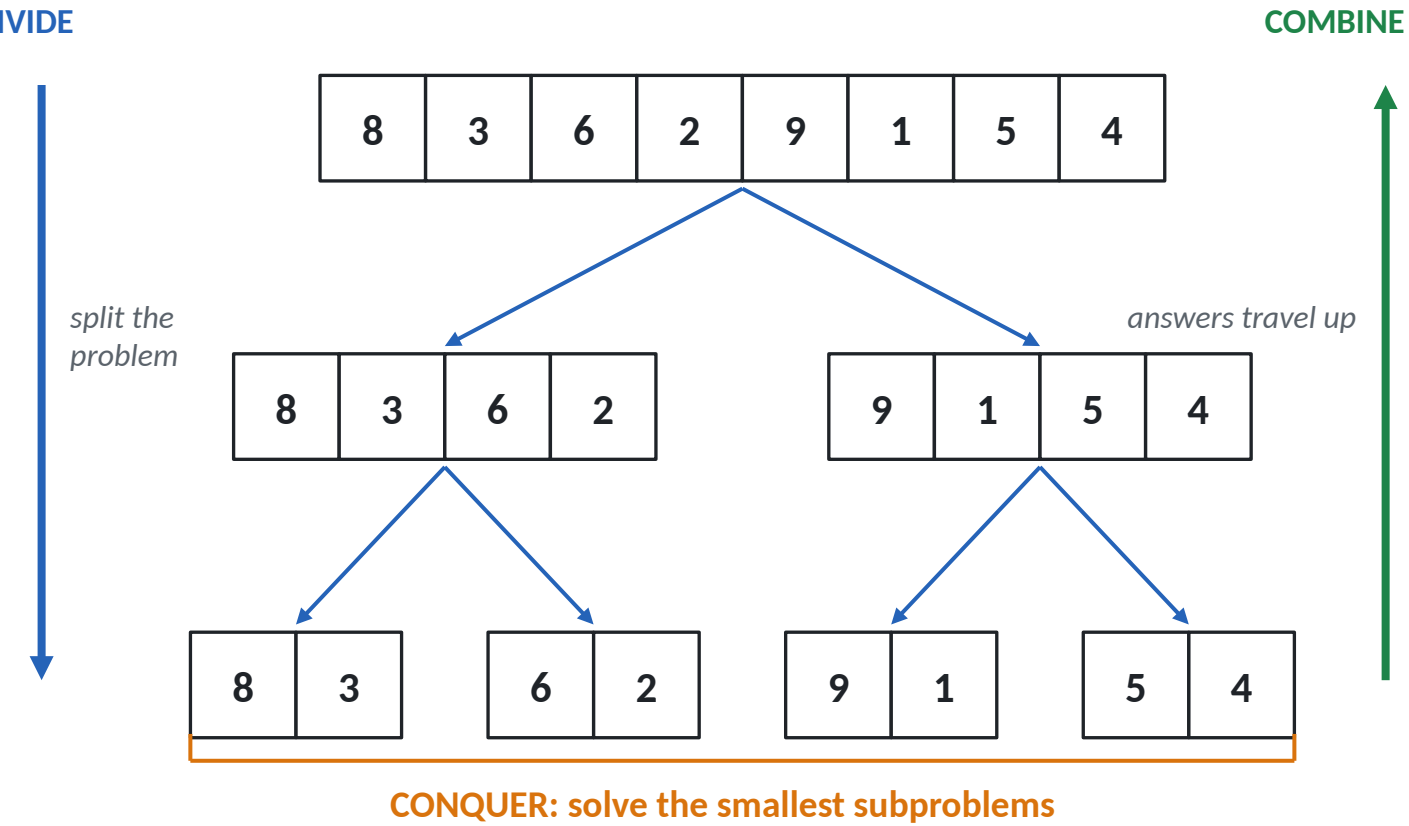
3 COMBINE

Use the solutions of the smaller subproblems to construct the solution to the original problem.



Note: merging is one possible way of combining solutions (as in Merge Sort). It is not a fourth step.

Understanding the three steps



DIVIDE

Keep splitting the problem.

CONQUER

Solve the smallest subproblems.

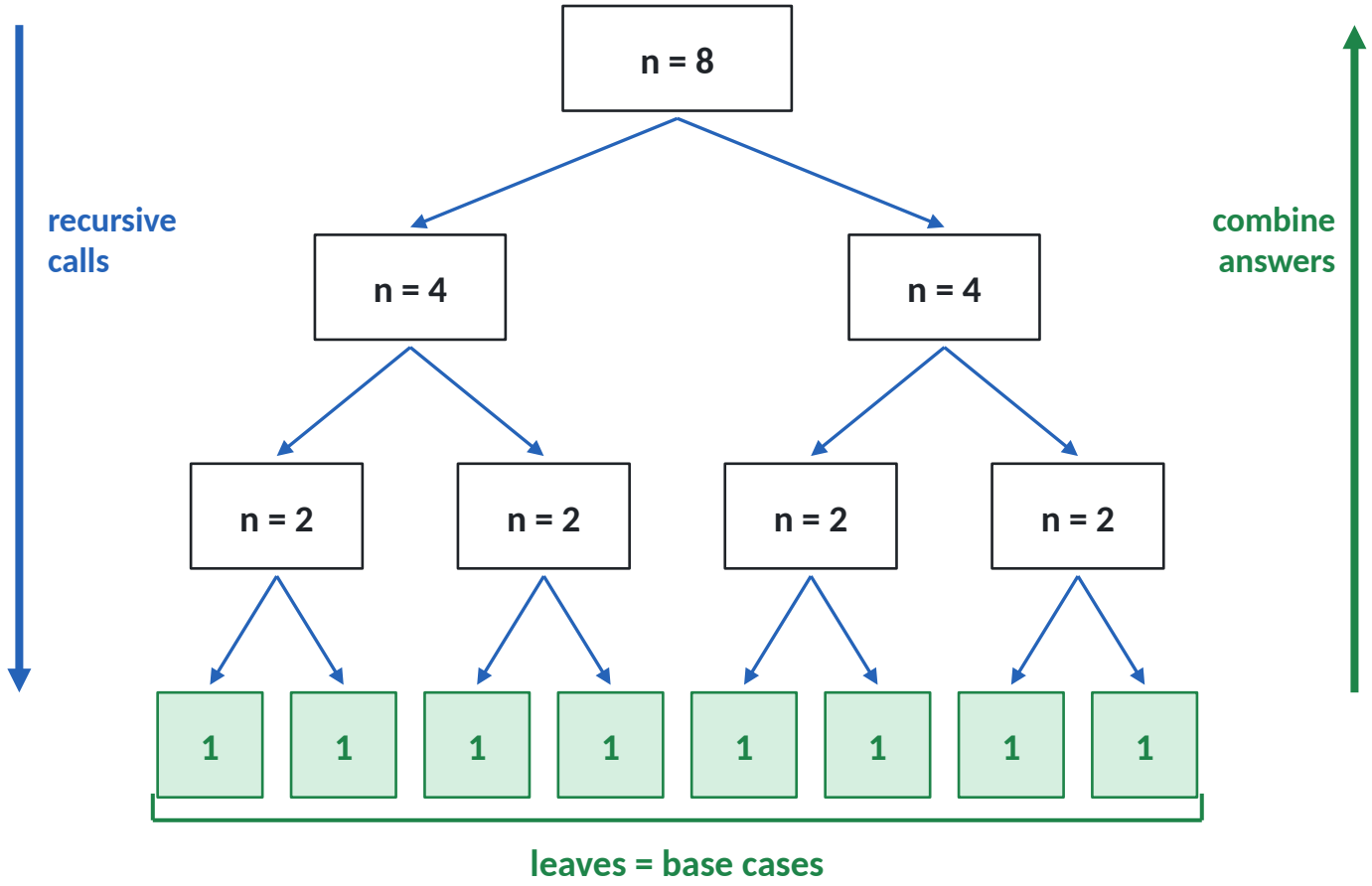
COMBINE

Bring the answers together.

Recursion goes down while dividing; results travel up while combining.

Base case and recursion

A recursion tree: one node for each recursive call



Recursive case

Divide the current problem into smaller subproblems.

Base case

The problem is small enough: solve it directly, with no further recursion.

The base case **determines when we stop dividing.**

The combine operation **determines how we construct the answer.**

Finding the minimum and the maximum

Given an array of numbers, find both its minimum and maximum elements.

7	2	9	4	1	6	3	8

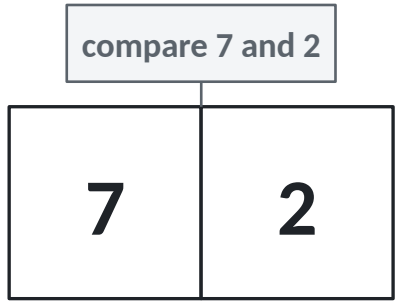
Minimum
?

Maximum
?

We need two results, not one.

A tiny case

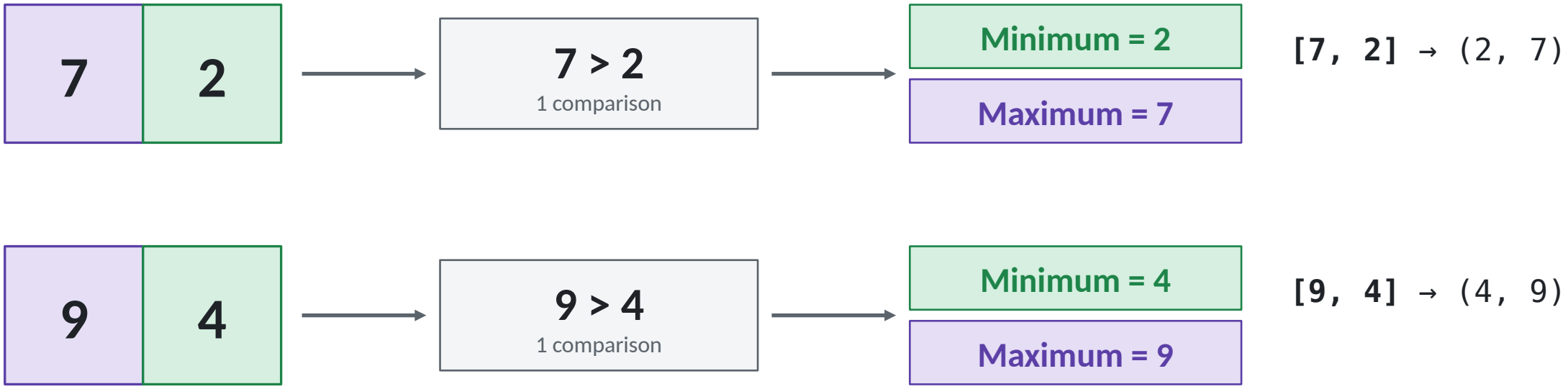
If an array contains only two elements, how many comparisons are needed to find both the minimum and the maximum?



One comparison is enough.

That observation gives us our base case.

Why stop at two elements?



Base case

A subproblem with two elements is solved directly: one comparison gives both answers.

Odd sizes

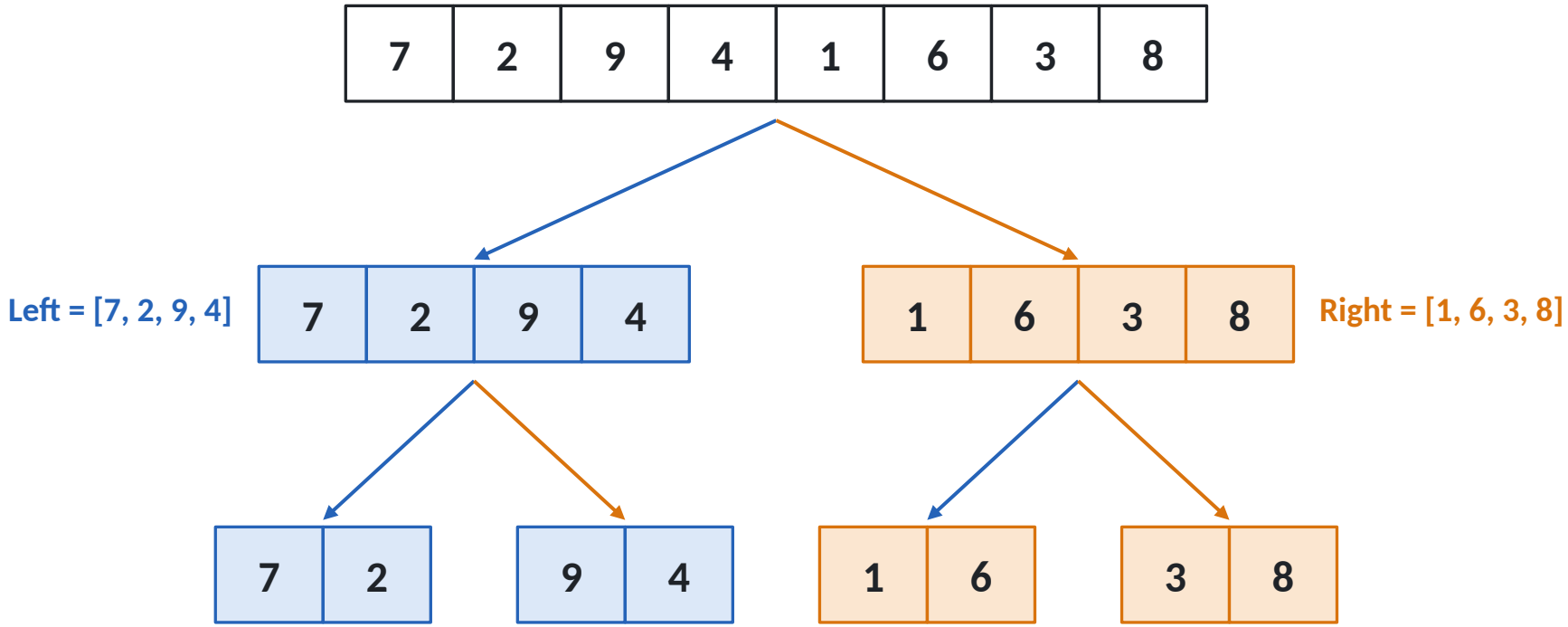
Our example has an even size, so every base case has exactly two elements. An implementation for any n must also handle a leftover single element.

Divide the array

Level 0
the whole array

Level 1
two halves

Level 2
four pairs

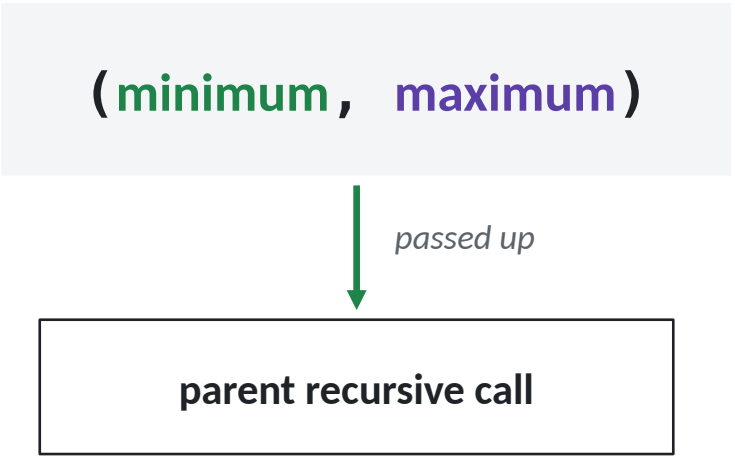


> Only dividing so far. Nothing has been compared yet, and the problem is not solved.

Conquer: solve the smallest subproblems

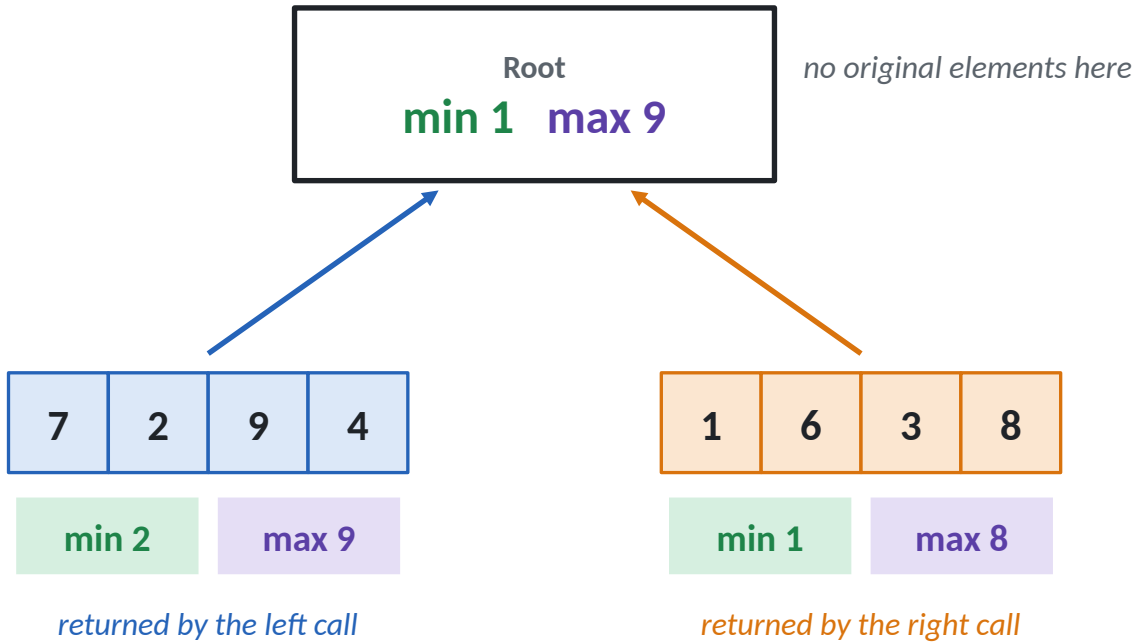
Pair	Minimum	Maximum
7 2	2	7
9 4	4	9
1 6	1	6
3 8	3	8

Every smallest subproblem returns exactly two pieces of information:



The parent receives only these two values, not the pair itself. One comparison per pair, four pairs, so four comparisons so far.

Combine: construct the answer



The parent never looks at the original elements again. It only needs the minimum and maximum returned by its two children.

Combine rule

Min = min(LeftMin, RightMin)

Max = max(LeftMax, RightMax)

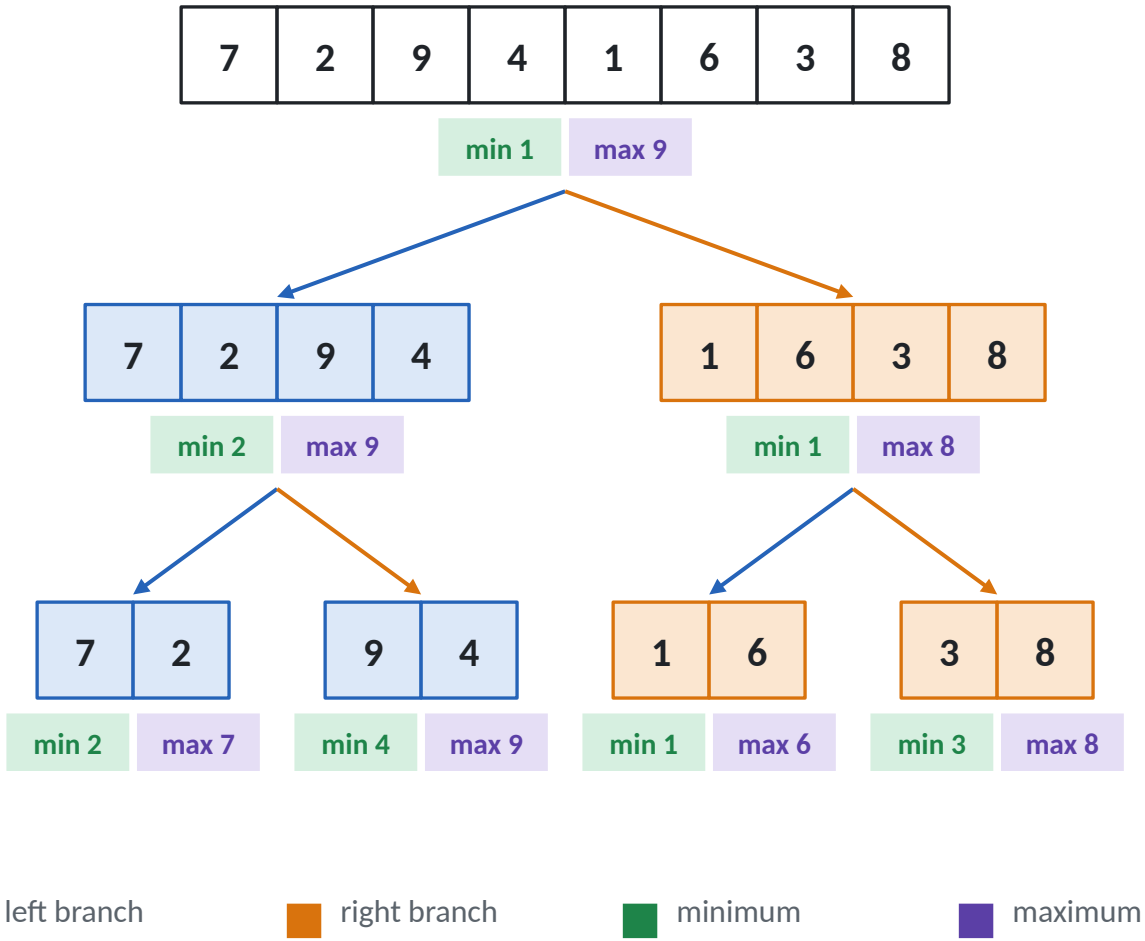
At the root

Min = min(2, 1) = **1**

Max = max(9, 8) = **9**

Answer: min 1, max 9

The complete Min-Max recursion tree



The algorithm

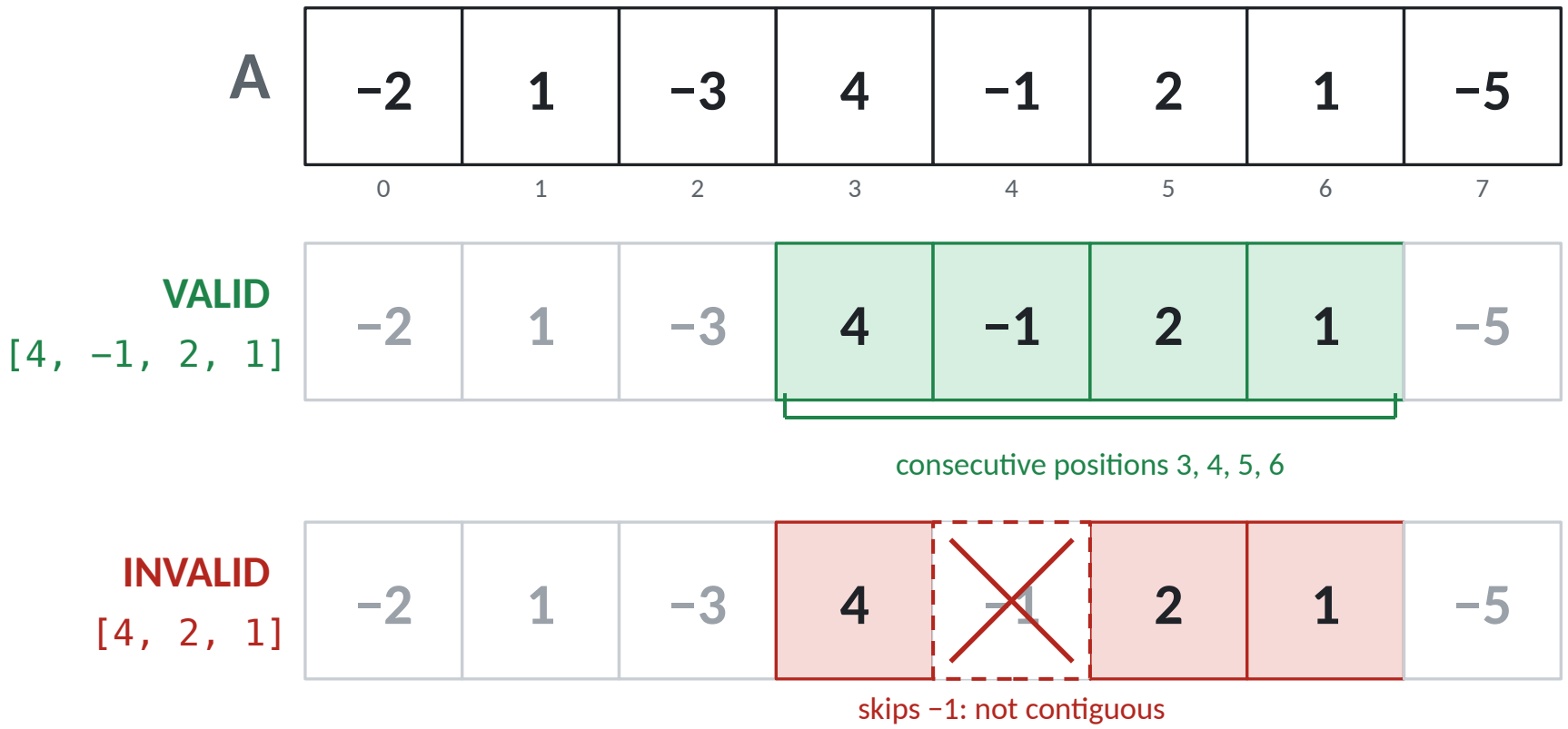
1. Divide the array into two halves.
2. Recursively find the minimum and maximum of each half.
3. Compare the two minimums.
4. Compare the two maximums.
5. Return the combined result.

For n = 8: 10 comparisons instead of 14.

3n/2 − 2 versus 2n − 2 (n a power of two).

What is a subarray?

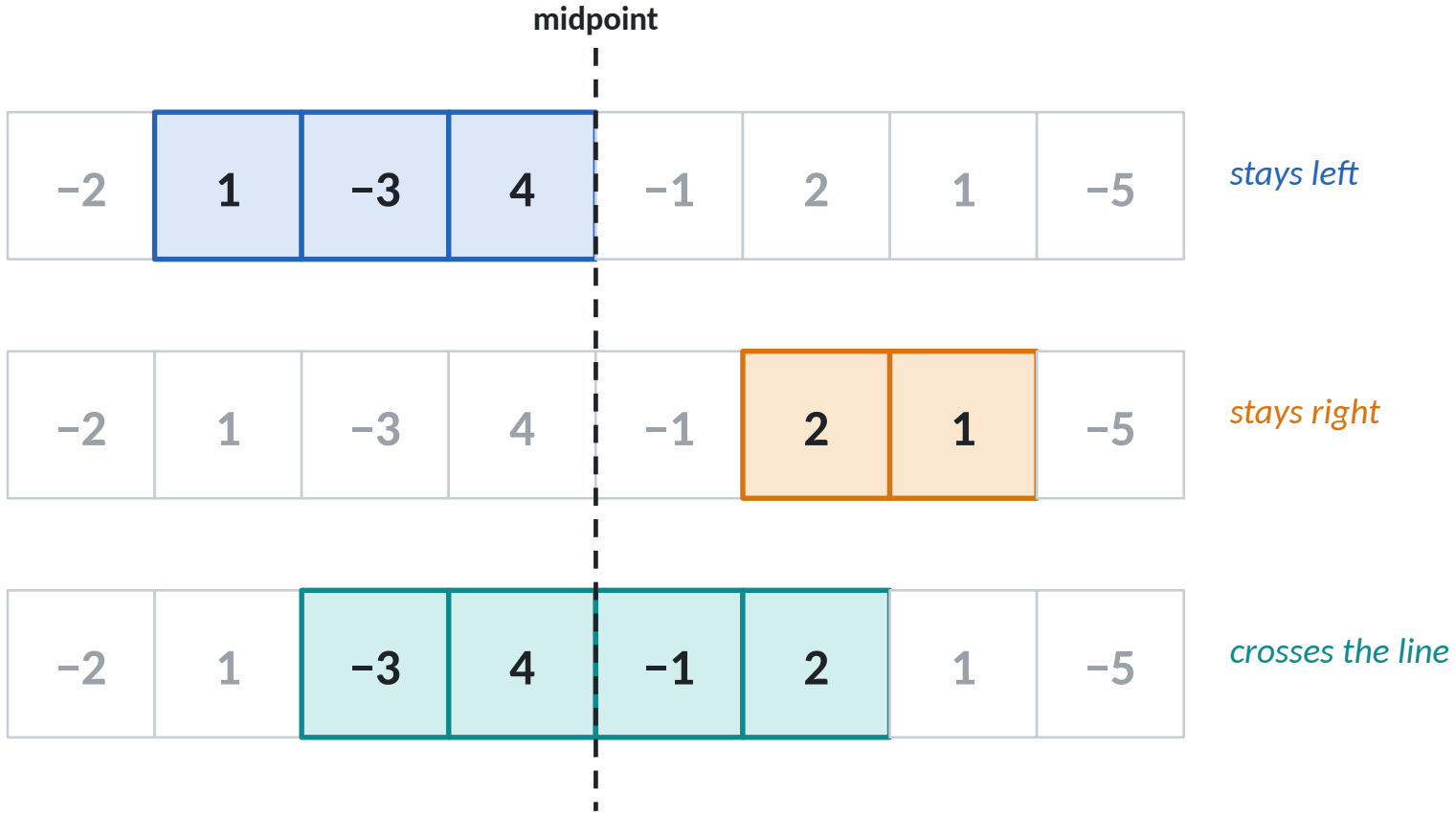
A subarray is a contiguous sequence of elements selected from an array.



Goal: find the contiguous subarray whose sum is maximum.

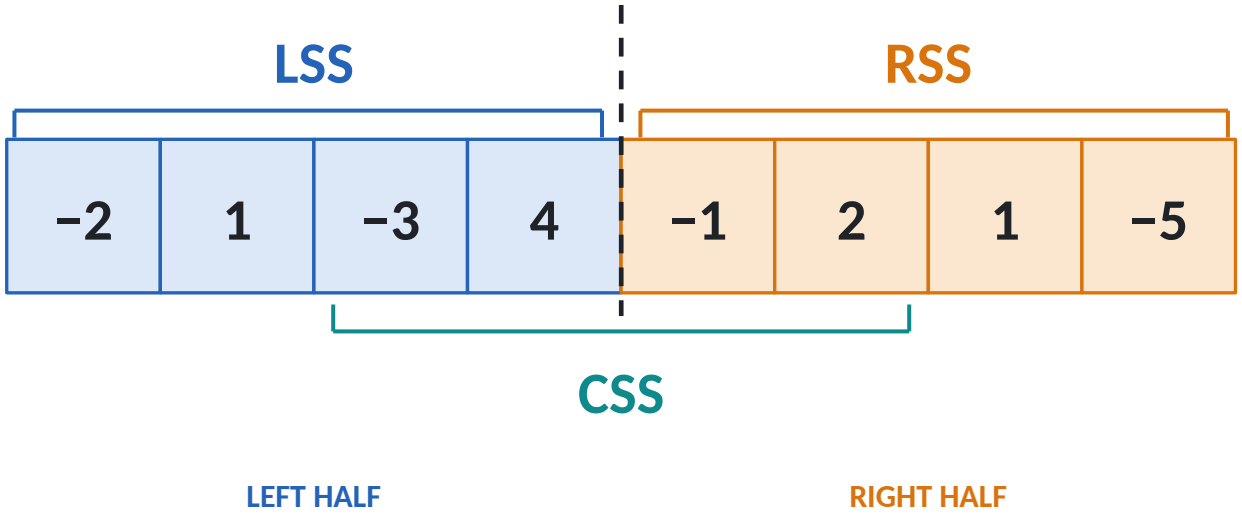
Where can the maximum subarray be?

After dividing at the midpoint, there are three possibilities.



> We must evaluate all three candidates and keep the best one.

Three candidates: LSS, RSS and CSS



Best = max(LSS, RSS, CSS)

LSS Left Subarray Sum

The maximum subarray sum that lies entirely within the left half.

RSS Right Subarray Sum

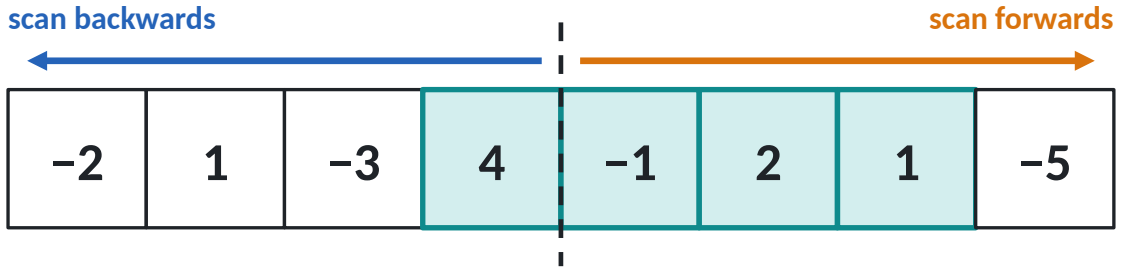
The maximum subarray sum that lies entirely within the right half.

CSS Cross Subarray Sum

The maximum subarray sum that crosses the midpoint and includes elements from both halves.

How do we calculate CSS?

Scan outward from the midpoint and keep the best running sum on each side.



Suffixes of the left half	Sum
[4]	4
[−3, 4]	1
[1, −3, 4]	2
[−2, 1, −3, 4]	0

Prefixes of the right half	Sum
[−1]	−1
[−1, 2]	1
[−1, 2, 1]	2
[−1, 2, 1, −5]	−3

Maximum left suffix sum = 4

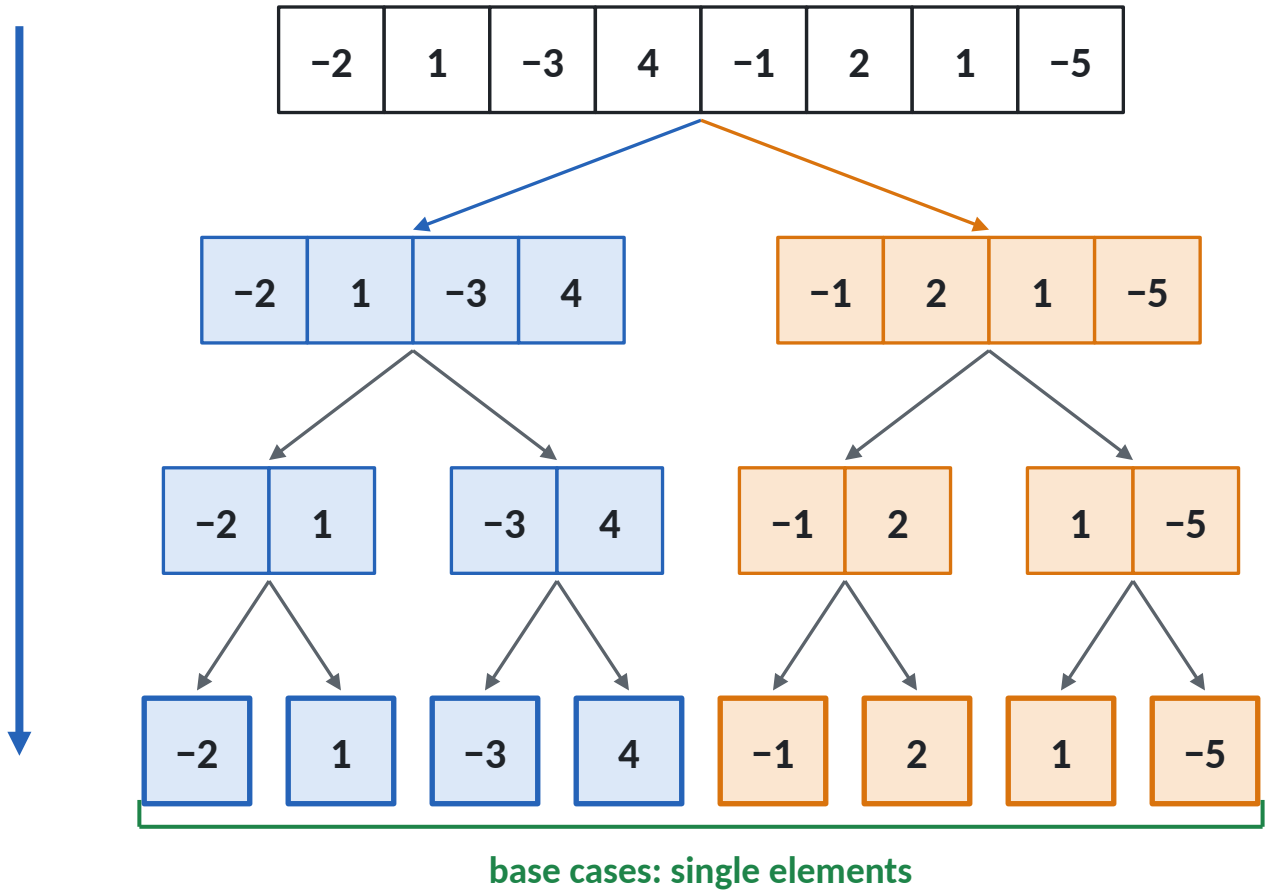
Maximum right prefix sum = 2

CSS = Maximum Left Suffix Sum + Maximum Right Prefix Sum = 4 + 2 = 6

[4, −1, 2, 1]

Divide down to single elements

DIVIDE



Why stop at one element?

A single element is already a valid, non-empty subarray. Its maximum subarray sum is the element itself.



Do not return 0 for a negative element.

The subarray must be non-empty, so [−3] gives −3.

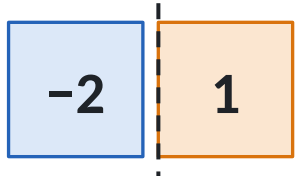
After the leaves, the results travel upward and are combined.

First combination level: the four pairs

Subarray	LSS	RSS	CSS	Best
−2 1	−2	1	−1	1
−3 4	−3	4	1	4
−1 2	−1	2	1	2
1 −5	1	−5	−4	1

Each pair splits into two single elements, so LSS and RSS are just those elements. The Best value of every pair is passed upward.

Worked example



LSS = −2 the left element

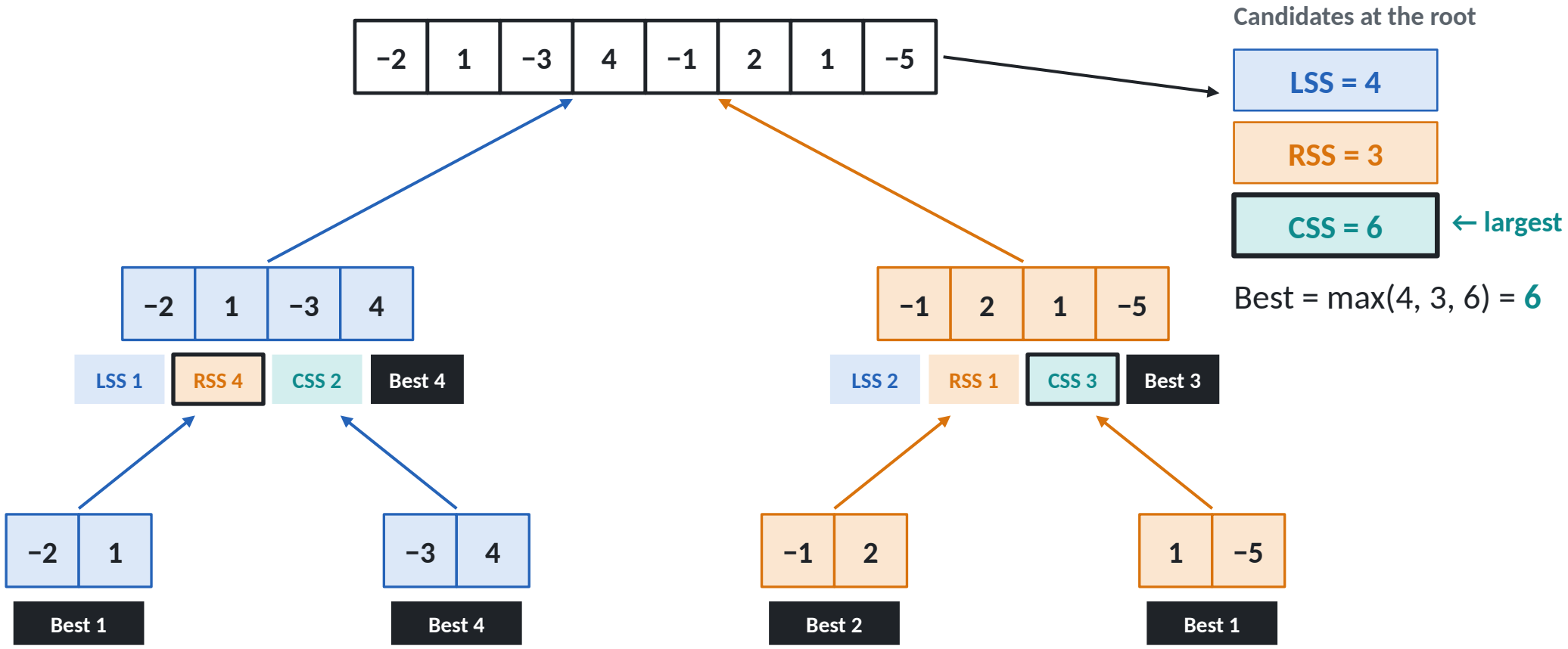
RSS = 1 the right element

CSS = −2 + 1 = −1 crossing sum

Best = max(−2, 1, −1) = 1

passed up to the parent

Final combination at the root



Maximum subarray: [4, −1, 2, 1] Maximum sum = 6

RECAP

One strategy, different problems

	Min-Max	Maximum Subarray
Base case	Two elements (for our even-sized example)	One element
Returned information	Minimum and maximum	Maximum subarray sum
Combine operation	Compare two minimums and two maximums	Compare LSS, RSS and CSS
Main idea	Combine extrema	Combine candidate subarray sums

> *Divide and Conquer is not about dividing an array for the sake of dividing it. The real challenge is understanding what information each smaller problem should return and how those results can be combined to solve the original problem.*

Next: Merge Sort and Quick Sort, two more important algorithms that follow this paradigm.