



分治策略 Divide and Conquer

钮鑫涛 王智彬
Nanjing University
2025 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports! We also use some materials from stanford-cs161.



The Divide-and-Conquer Approach

- **Divide** the given problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them *recursively*.
 - Or, use brute-force if a subproblem is small enough.
- **Combine** the solutions for the subproblems to obtain the solution for the original problem.



Described in pseudocode

Solve (I):

if I is small enough:

$solution := \text{DirectSolve}(I)$

Direct solve the basic case, or use brute-force if (sub)problem is simple

else

$\langle I_1, I_2, \dots, I_k \rangle := \text{DivideProblem}(I)$

Divide the problem into **smaller** subproblems.

for $j := 1$ to k

$solution_j = \text{Solve}(I_j)$

Recursively solve subproblems.

$solution = \text{Combine}(solution_1, \dots, solution_k)$

Combine solutions of subproblems to get solution for original problem.

return $solution$



Correctness of Divide-and-Conquer

- How to prove the correctness of a divide-and-conquer algorithm?
 - Use (strong) mathematical induction, proceeding by induction on the “size” of the inputs.
- **Induction basis**: prove the algorithm can correctly solve small problem instances.
 - Prove `DirectSolve` is correct if $|I| \leq c$.
- **Induction hypothesis**: the algorithm can correctly solve **any** problem instance of size at most, say, n .
 - `Solve` is correct if $|I| \leq n$.
- **Inductive step**: assuming induction hypothesis, prove the algorithm can correctly solve problem instance of size $n + 1$.
 - Assume `Solve` is correct if $|I| \leq n$, Prove `Solve` is correct if $|I| = n + 1$

Solve (I):

if I is small enough:

solution := DirectSolve(I)

else

$\langle I_1, I_2, \dots, I_k \rangle := \text{DivideProblem}(I)$

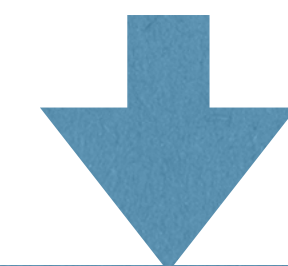
for $j := 1$ to k

solution _{j} = Solve(I_j)

solution = Combine(solution₁, ..., solution _{k})

return solution

Partial or Total Correctness?



Termination and partial correctness can be encapsulated !

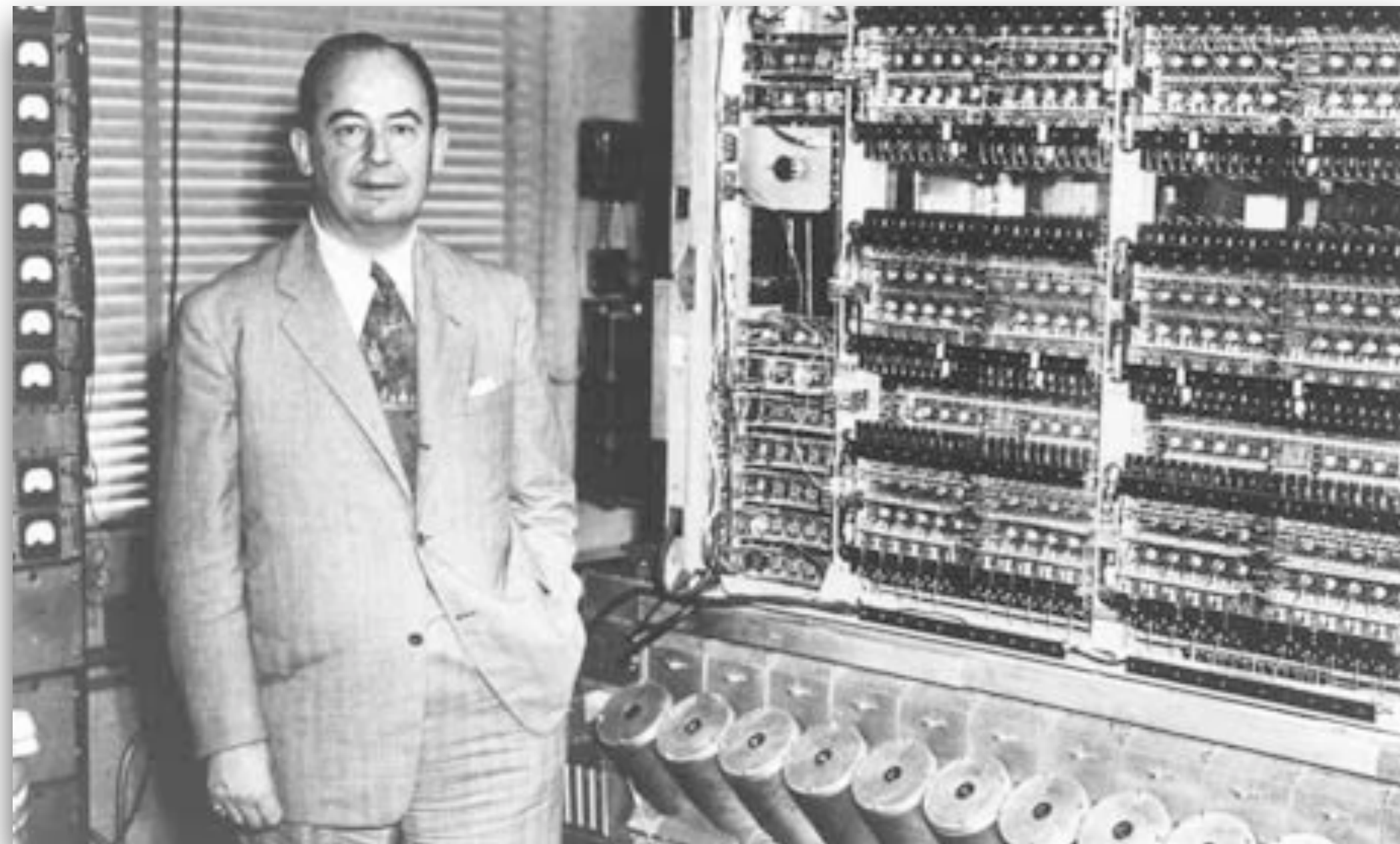


Merge Sort



MergeSort

- An efficient divide-and-conquer algorithm for sorting.
- Invented by John von Neumann in the 1940s.





MergeSort

Divide-and-Conquer Template

Solve (I):

if I is small enough:

$solution := DirectSolve(I)$

else

$\langle I_1, I_2, \dots, I_k \rangle := DivideProblem(I)$

for $j := 1$ to k

$solution_j = Solve(I_j)$

$solution = Combine(solution_1, \dots, solution_k)$

return $solution$

MergeSort ($A[1..n]$):

if $n = 1$:

$sol[1..n] := [1..n]$

else

$solLeft[1..(n/2)] := MergeSort(A[1..(n/2)])$

$solRight[1..(n/2)] := MergeSort(A[(n/2+1)..n])$

$sol[1..n] := Merge(solLeft[1..(n/2)], solRight[1..(n/2)])$

return $sol[1..n]$

Merge ($A[1..n], B[1..m]$):

$Aindex := 1, Bindex := 1, Result := []$

// Scan A and B from left to right,

// Append the currently smallest to the result array

while $Aindex \leq A.length$ **and** $Bindex \leq B.length$

if $A[Aindex] \leq B[Bindex]$

$Result.AddLast(A[Aindex])$

$Aindex := Aindex + 1$

else

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

// Copy the remaining elements of A and B

while $Aindex \leq A.length$

$Result.AddLast(A[Aindex])$

$Aindex := Aindex + 1$

while $Bindex \leq B.length$

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

return $Result$



The Merge Subroutine

Merge ($A[1..n], B[1..m]$):

$Aindex := 1, Bindex := 1, Result := []$

// Scan A and B from left to right,

// Append the currently smallest to the result array

while $Aindex \leq A.length$ **and** $Bindex \leq B.length$

if $A[Aindex] \leq B[Bindex]$

$Result.AddLast(A[Aindex])$

$Aindex := Aindex + 1$

else

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

// Copy the remaining elements of A and B

while $Aindex \leq A.length$

$Result.AddLast(A[Aindex])$

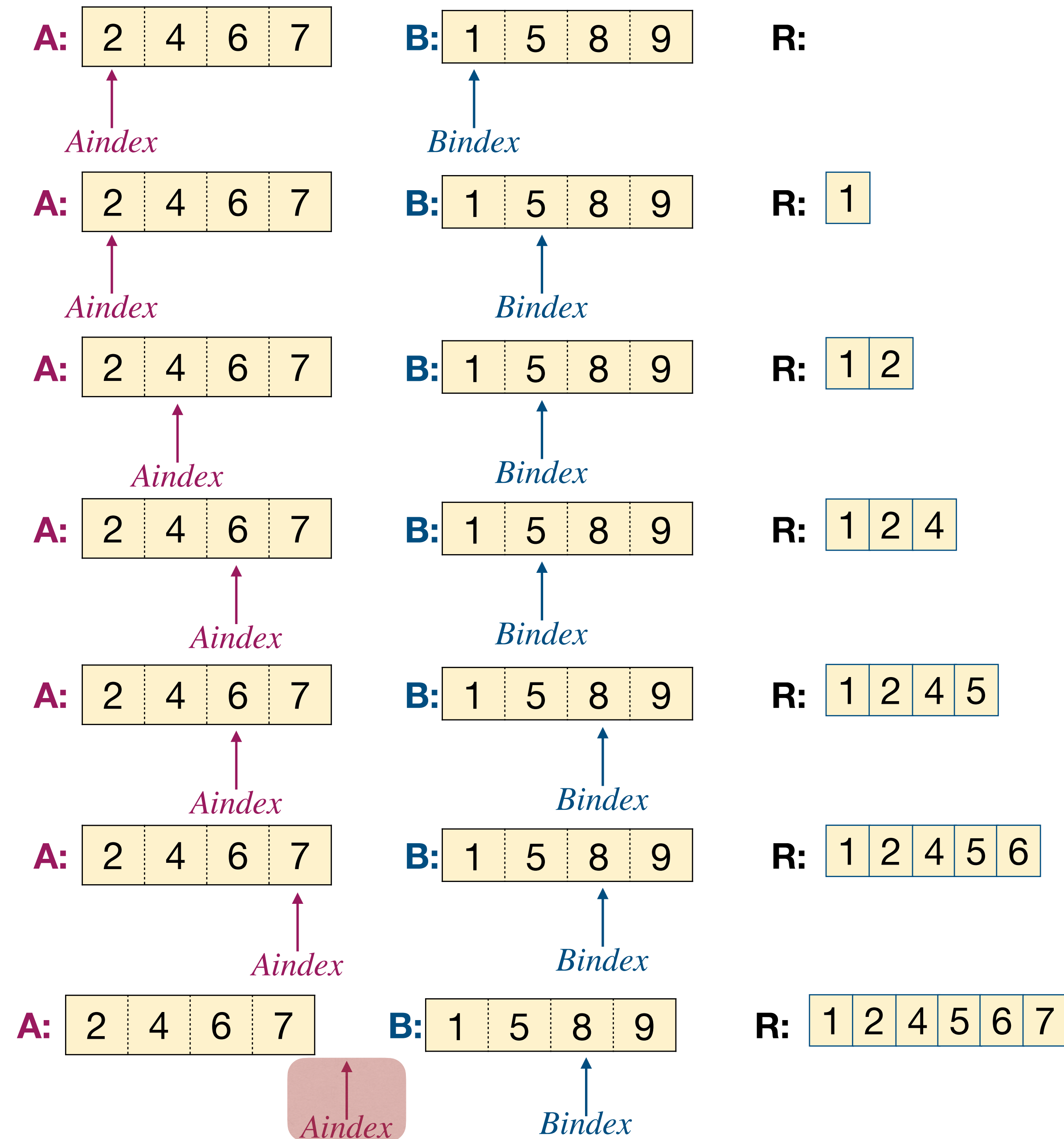
$Aindex := Aindex + 1$

while $Bindex \leq B.length$

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

return $Result$





The Merge Subroutine

Merge ($A[1..n], B[1..m]$):

$Aindex := 1, Bindex := 1, Result := []$

// Scan A and B from left to right,

// Append the currently smallest to the result array

while $Aindex \leq A.length$ **and** $Bindex \leq B.length$

if $A[Aindex] \leq B[Bindex]$

$Result.AddLast(A[Aindex])$

$Aindex := Aindex + 1$

else

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

// Copy the remaining elements of A and B

while $Aindex \leq A.length$

$Result.AddLast(A[Aindex])$

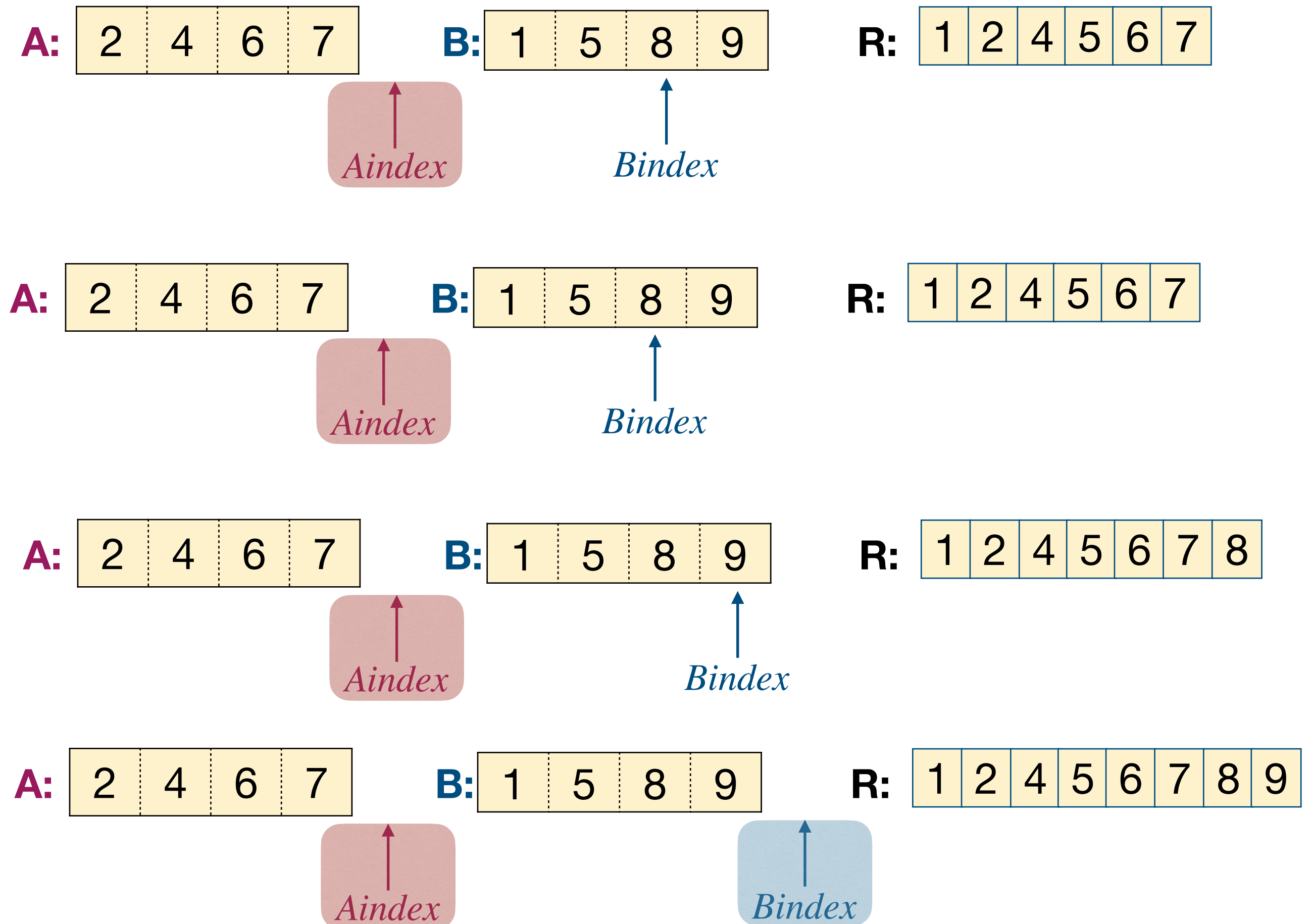
$Aindex := Aindex + 1$

while $Bindex \leq B.length$

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

return $Result$

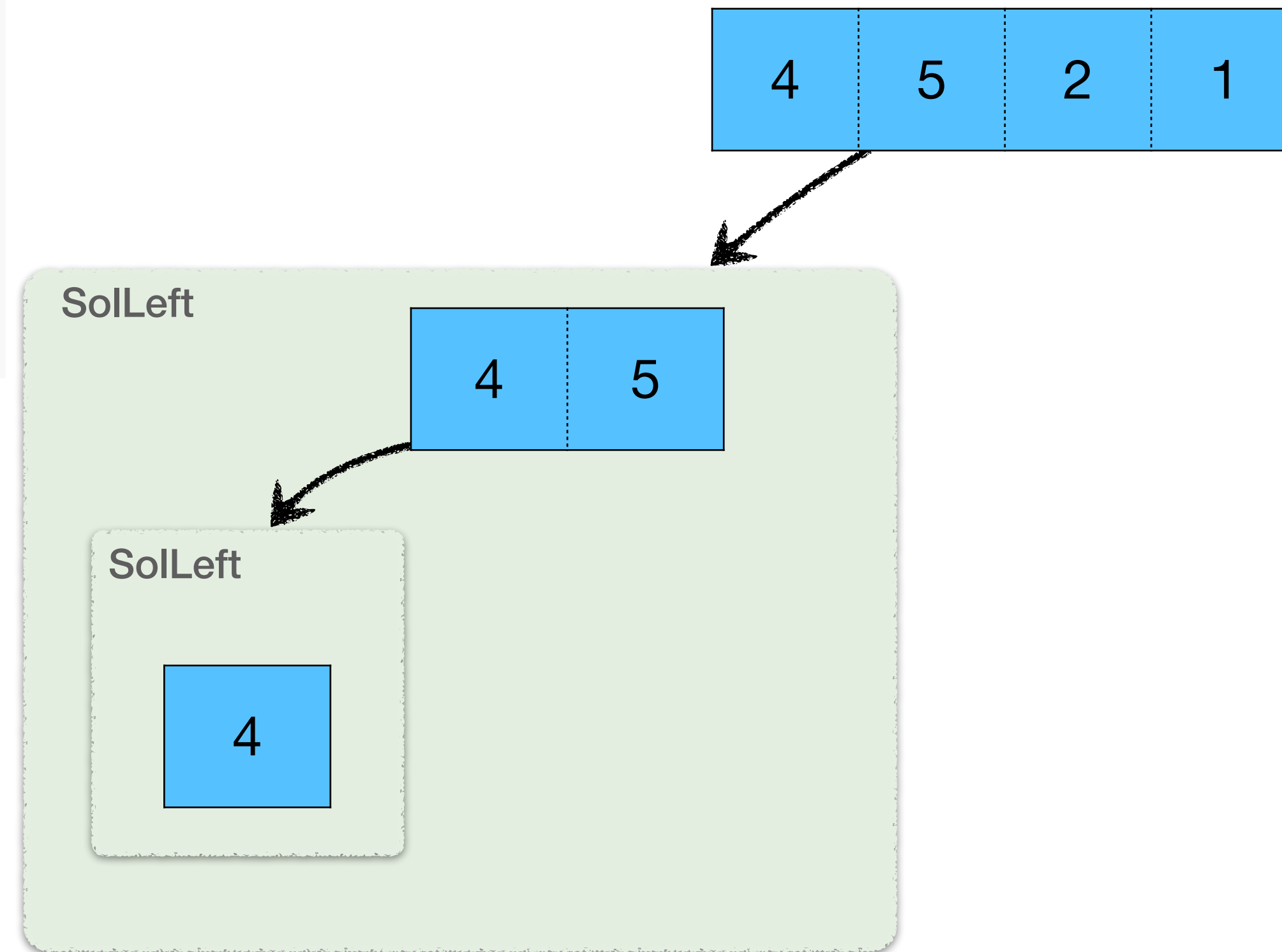




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

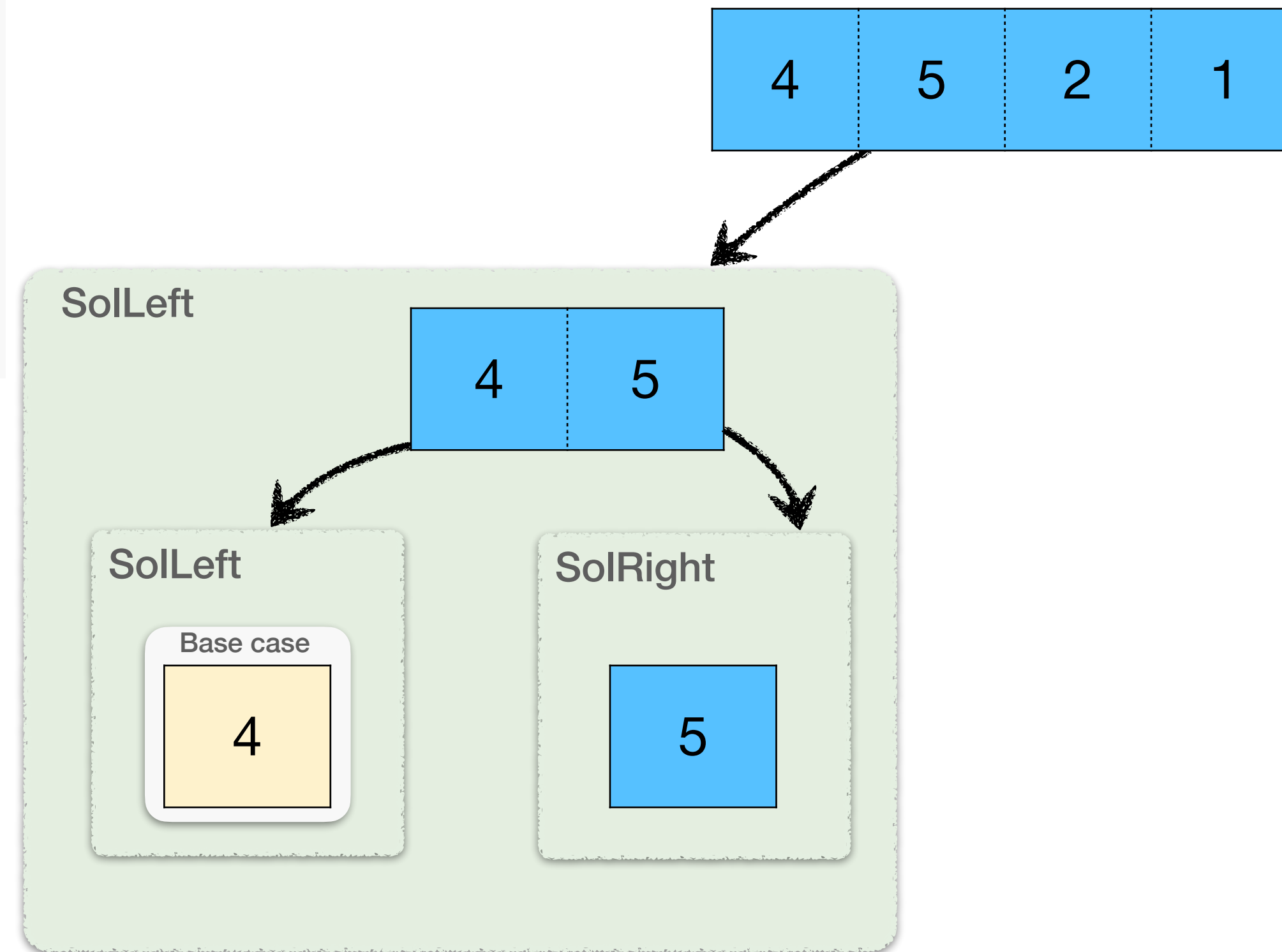




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1..(n/2)] := MergeSort(A[1..(n/2)])$   
     $solRight[1..(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1..(n/2)], solRight[1..(n/2)])$   
return  $sol[1..n]$ 
```

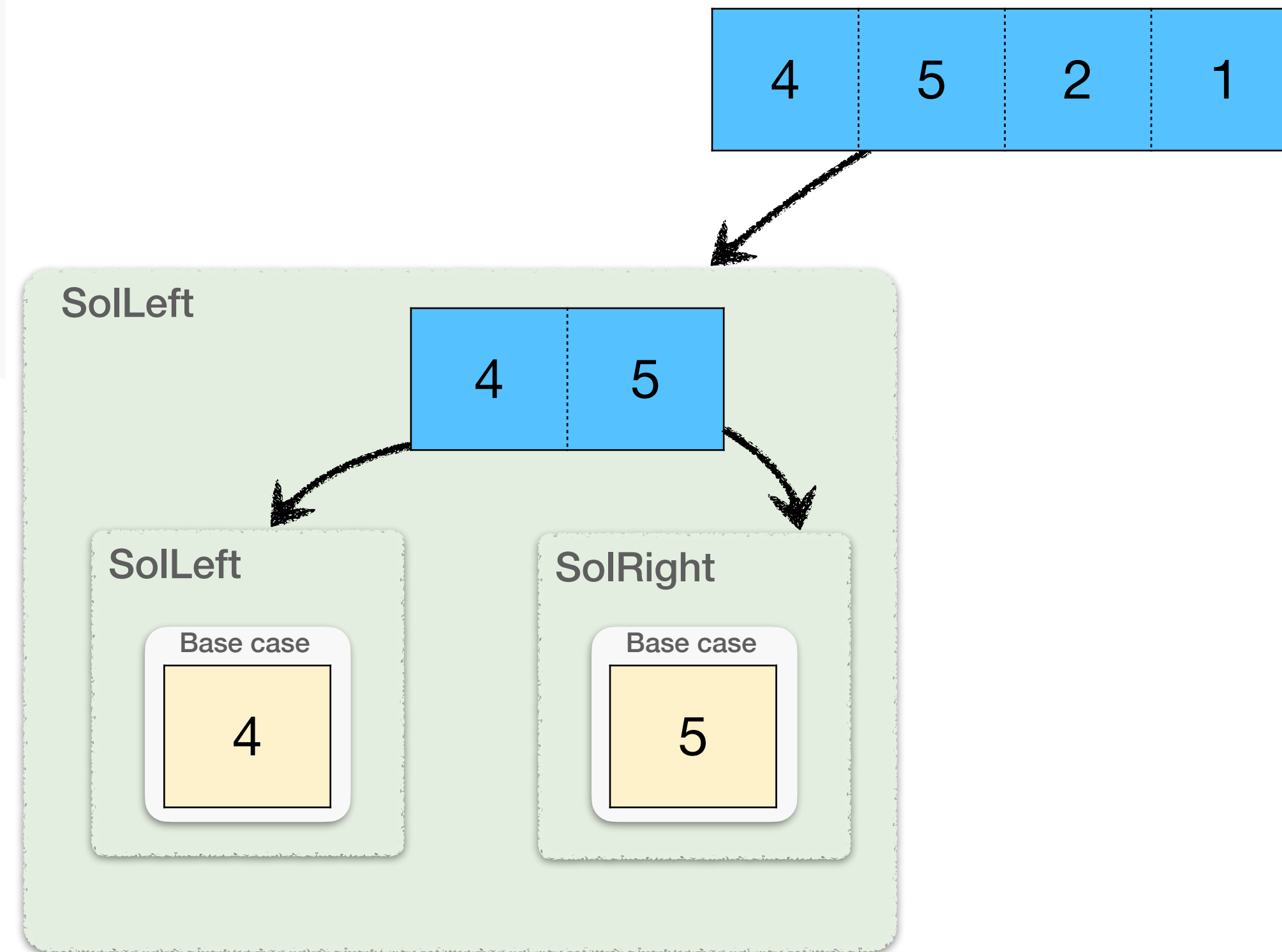




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

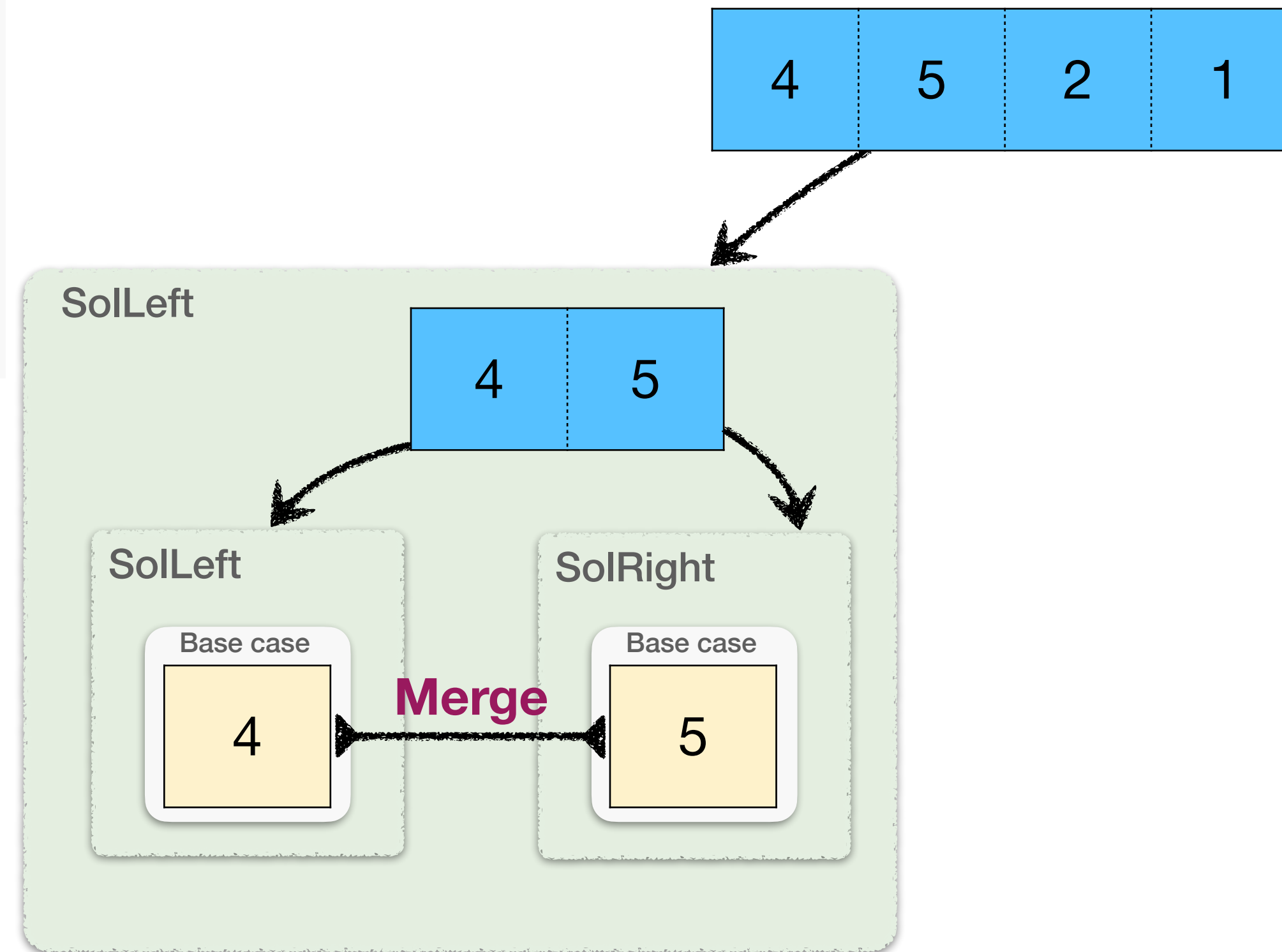




Sample execution of MergeSort

MergeSort ($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

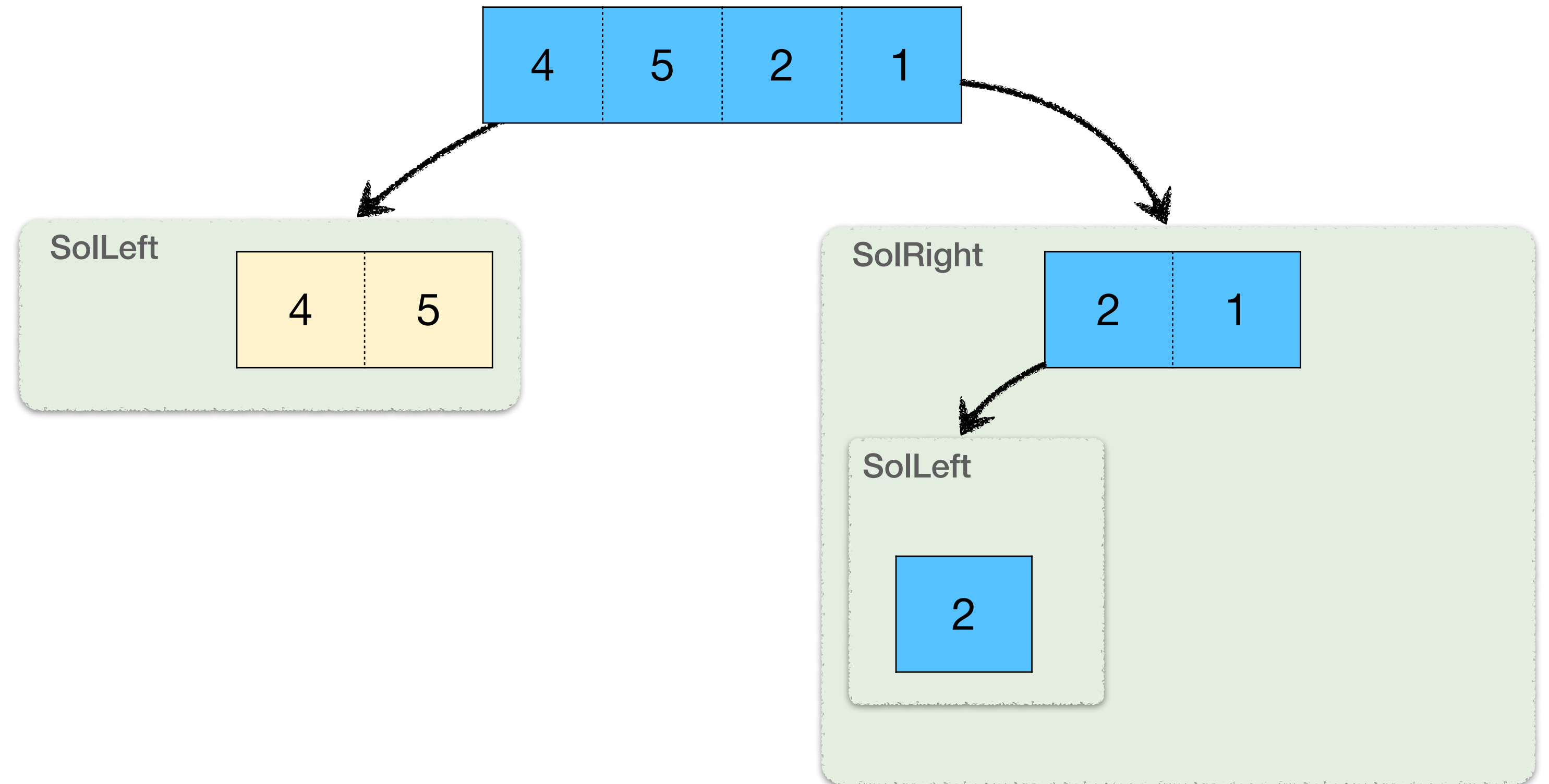




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

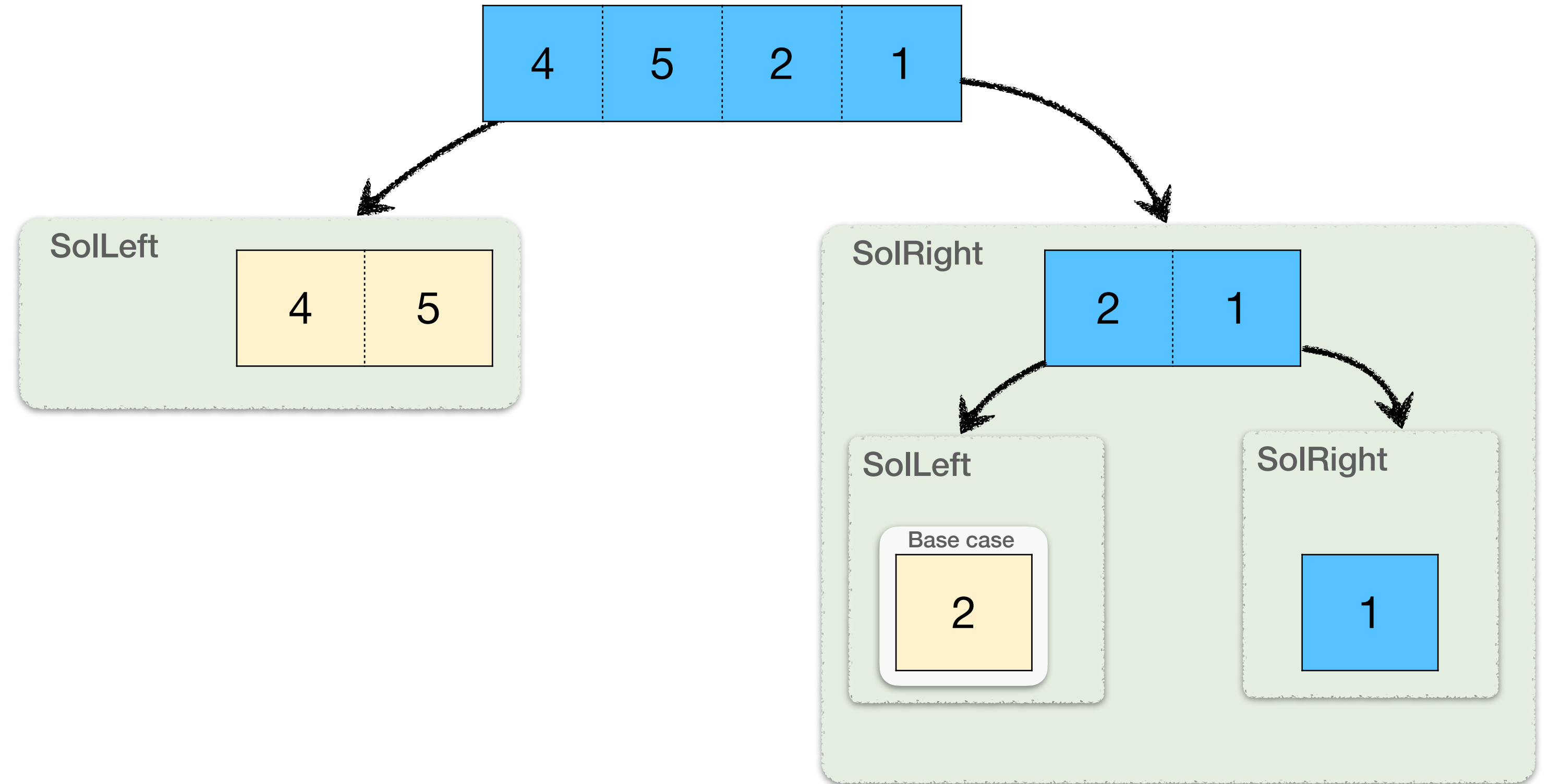




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1..(n/2)] := MergeSort(A[1..(n/2)])$   
     $solRight[1..(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1..(n/2)], solRight[1..(n/2)])$   
return  $sol[1..n]$ 
```

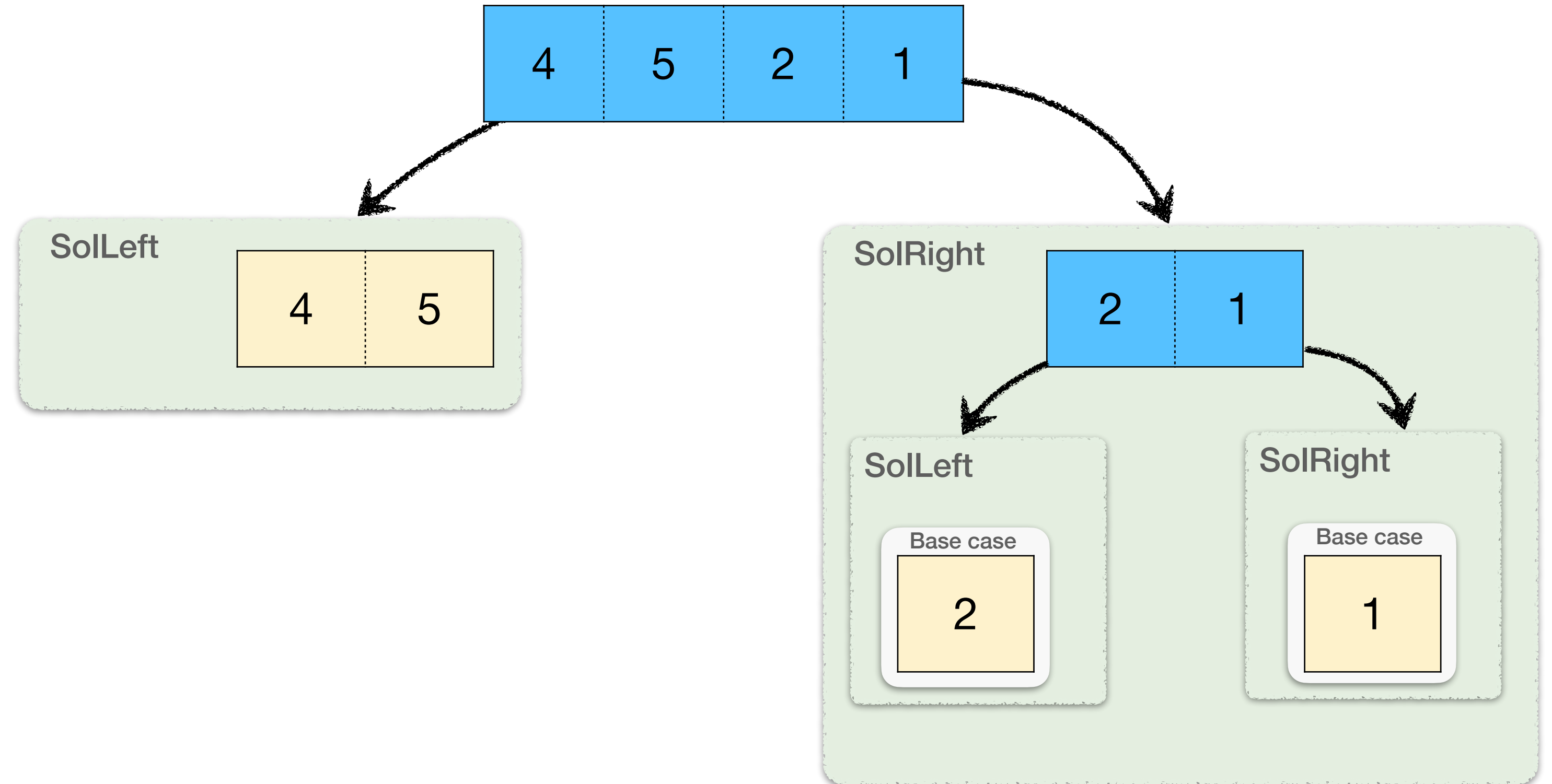




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

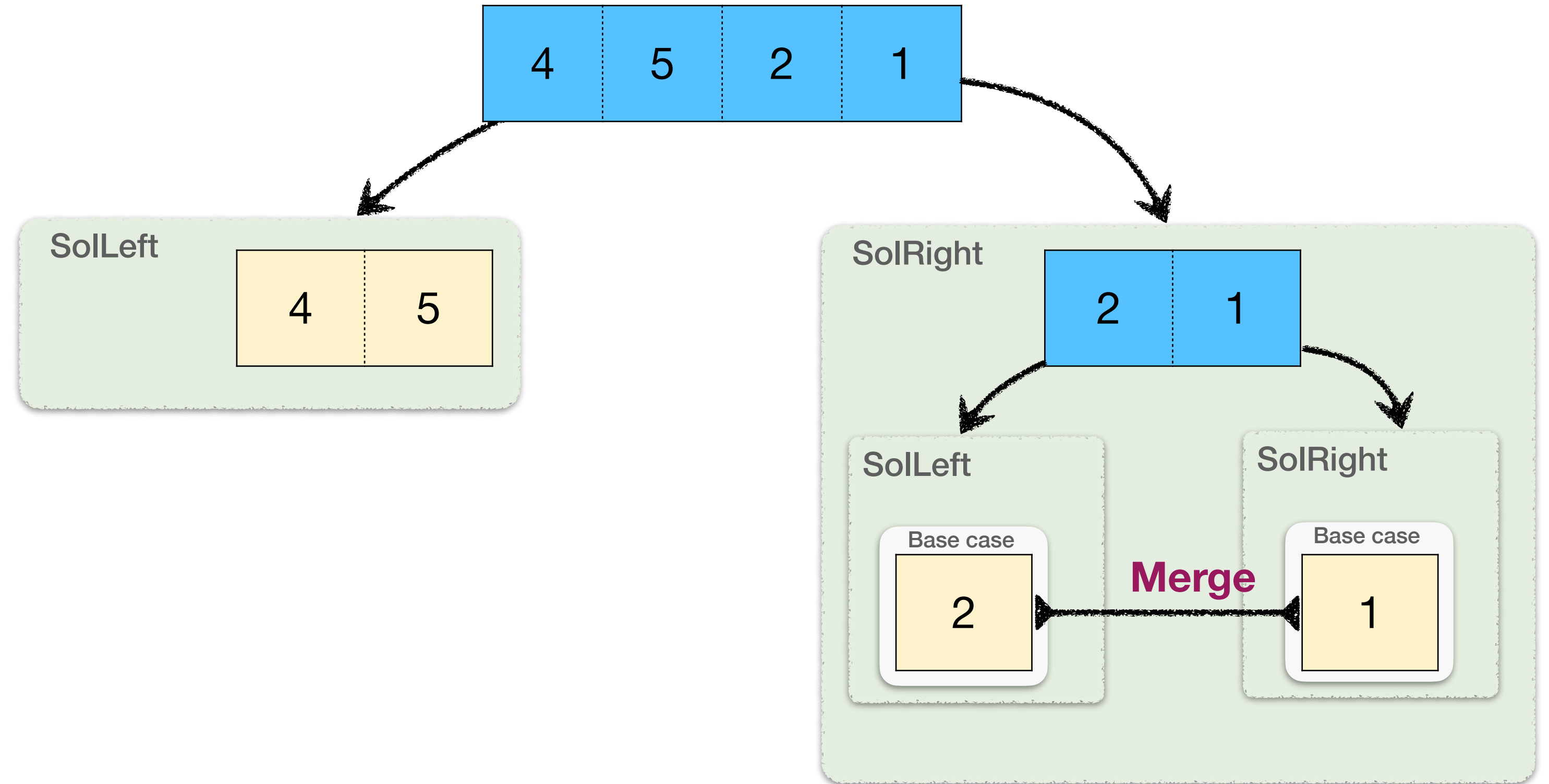




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

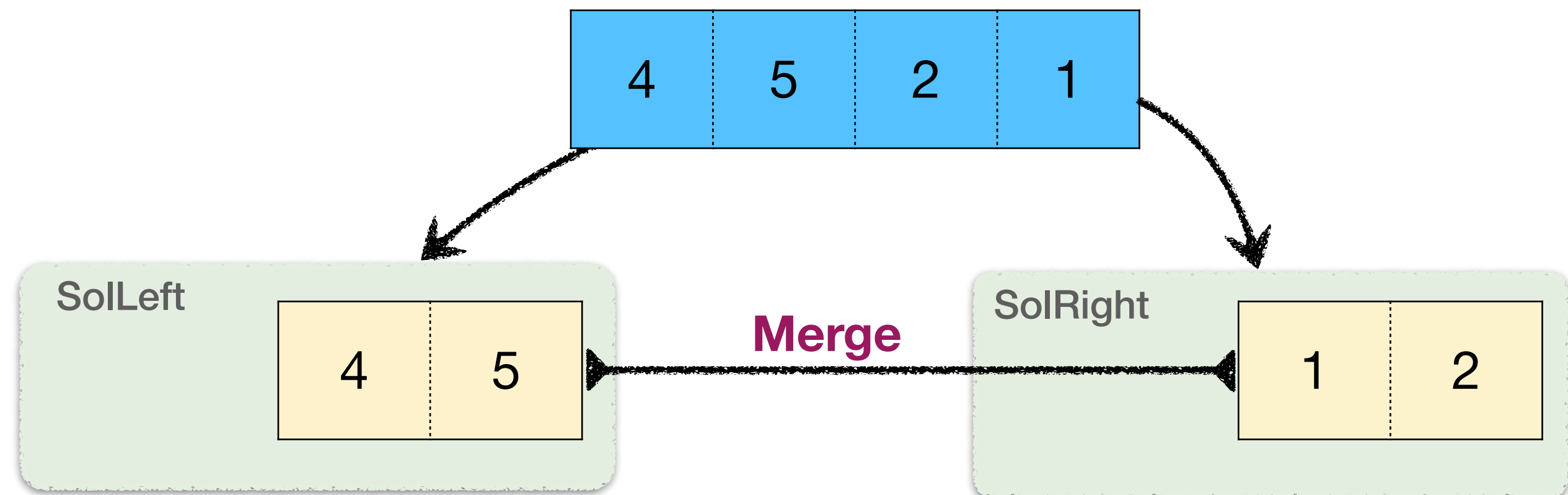




Sample execution of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```





Sample execution of MergeSort

```
MergeSort ( $A[1 \dots n]$ ):  
  if  $n = 1$ :  
     $sol[1 \dots n] := [1 \dots n]$   
  else  
     $solLeft[1 \dots (n/2)] := MergeSort(A[1 \dots (n/2)])$   
     $solRight[1 \dots (n/2)] := MergeSort(A[(n/2+1) \dots n])$   
     $sol[1 \dots n] := Merge(solLeft[1 \dots (n/2)], solRight[1 \dots (n/2)])$   
  return  $sol[1 \dots n]$ 
```

1	2	4	5
---	---	---	---



Correctness of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

- **Induction basis:** MergeSort is correct when $n = 1$.
- **Induction hypothesis:** Assume MergeSort is correct if $n \leq n'$
- **Inductive step:** MergeSort is correct when $n = n' + 1$



How to prove the correctness of the subroutine?

- Correctness of this routine?
 - ▶ Find proper loop invariant!
 - ▶ What is it?

Merge ($A[1..n]$, $B[1..m]$):

$Aindex := 1, Bindex := 1, Result := []$

// Scan A and B from left to right,

// Append the currently smallest to the result array

while $Aindex \leq A.length$ **and** $Bindex \leq B.length$

if $A[Aindex] \leq B[Bindex]$

$Result.AddLast(A[Aindex])$

$Aindex := Aindex + 1$

else

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

// Copy the remaining elements of A and B

while $Aindex \leq A.length$

$Result.AddLast(A[Aindex])$

$Aindex := Aindex + 1$

while $Bindex \leq B.length$

$Result.AddLast(B[Bindex])$

$Bindex := Bindex + 1$

return $Result$



Time complexity of MergeSort

MergeSort ($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

Merge ($A[1..n], B[1..m]$):

```
 $Aindex := 1, Bindex := 1, Result := []$   
  
// Scan A and B from left to right,  
// Append the currently smallest to the result array  
while  $Aindex \leq A.length$  and  $Bindex \leq B.length$   
    if  $A[Aindex] \leq B[Bindex]$   
         $Result.AddLast(A[Aindex])$   
         $Aindex := Aindex + 1$   
    else  
         $Result.AddLast(B[Bindex])$   
         $Bindex := Bindex + 1$   
  
// Copy the remaining elements of A and B  
while  $Aindex \leq A.length$   
     $Result.AddLast(A[Aindex])$   
     $Aindex := Aindex + 1$   
while  $Bindex \leq B.length$   
     $Result.AddLast(B[Bindex])$   
     $Bindex := Bindex + 1$   
return  $Result$ 
```

- For Subroutine *Merge*, the three “*while*” processes involves scanning all the elements in *A* and *B*.
- The “*if*” processes has fewer comparisons than “*while*” processes
- Therefore, the time complexity of Subroutine *Merge* is $\Theta(n)$, where n is the sum of the elements of *A* and *B*.



Time complexity of MergeSort

MergeSort($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1...(n/2)] := MergeSort(A[1...(n/2)])$   
     $solRight[1...(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1...(n/2)], solRight[1...(n/2)])$   
return  $sol[1..n]$ 
```

- For the main procedure MergeSort :
 - Let $T(n)$ be the runtime of MergeSort on instance of size n .
 - Clearly, $T(1) = c_1 = \Theta(1)$ for some constant c_1 .
 - For larger n , $T(n) = 2 \cdot T(n/2) + c_2 \cdot n = 2T(n/2) + \Theta(n)$.



Time complexity of MergeSort

- A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

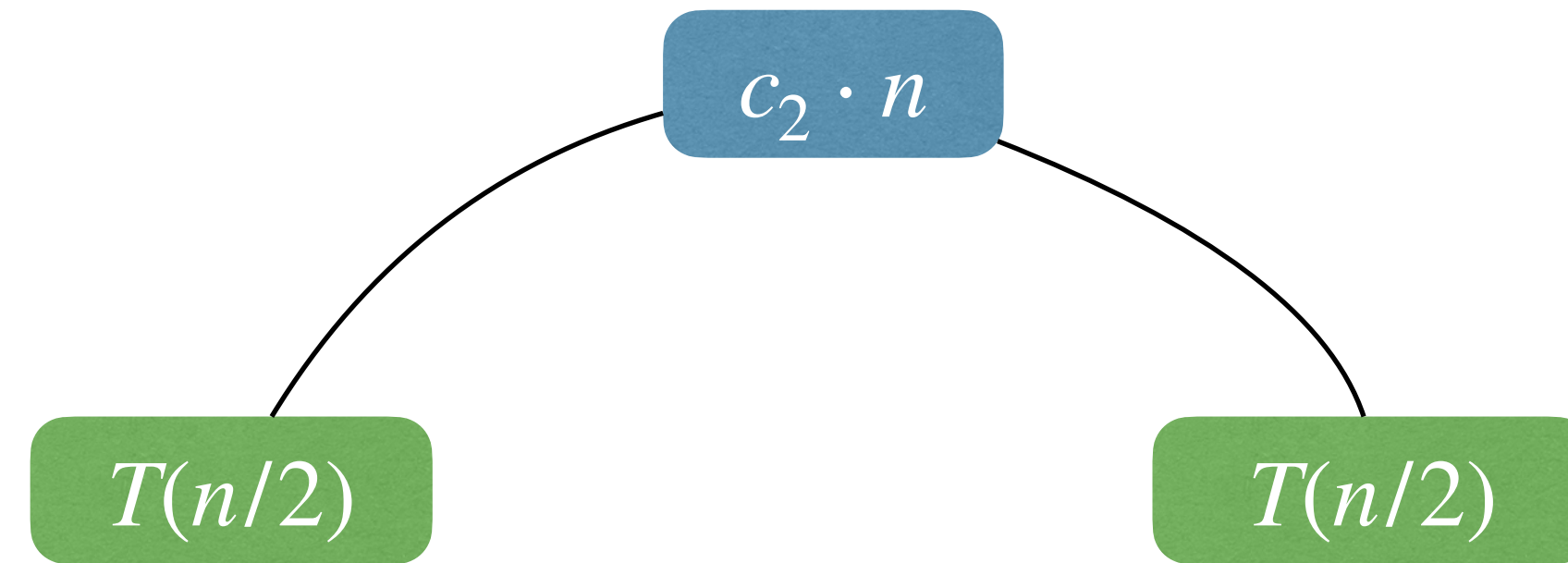
$T(n)$



Time complexity of MergeSort

- A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

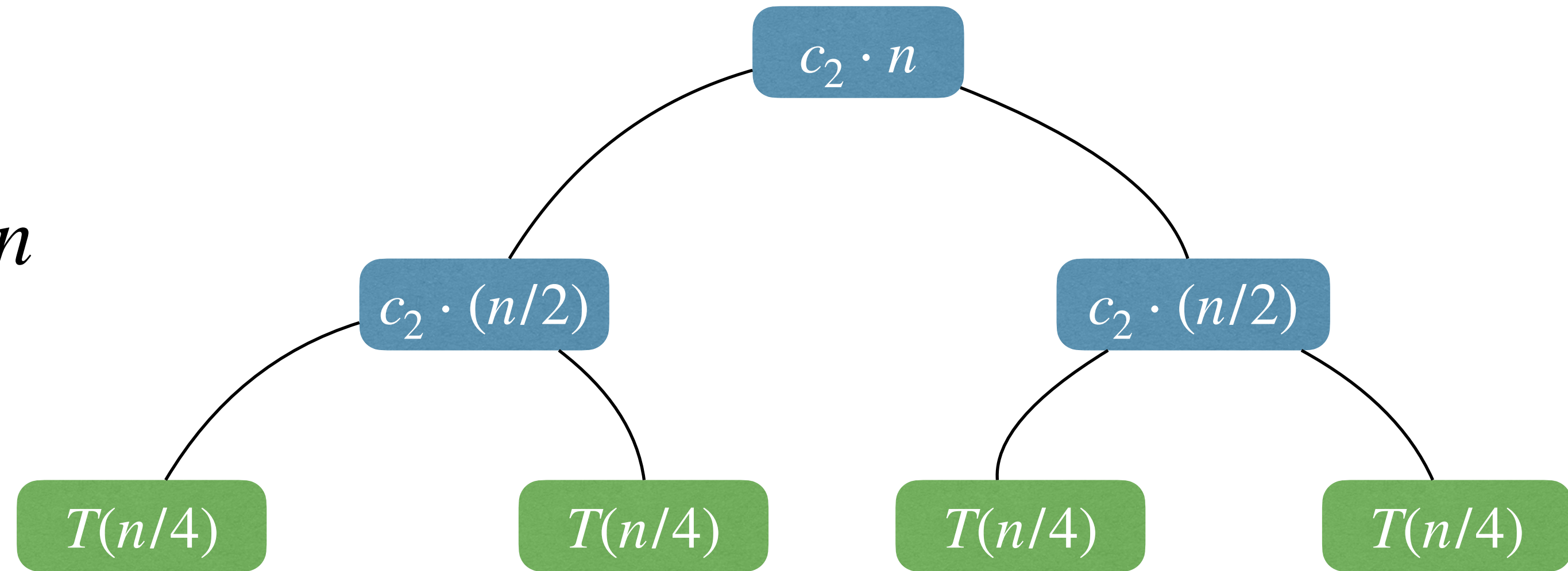




Time complexity of MergeSort

- A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

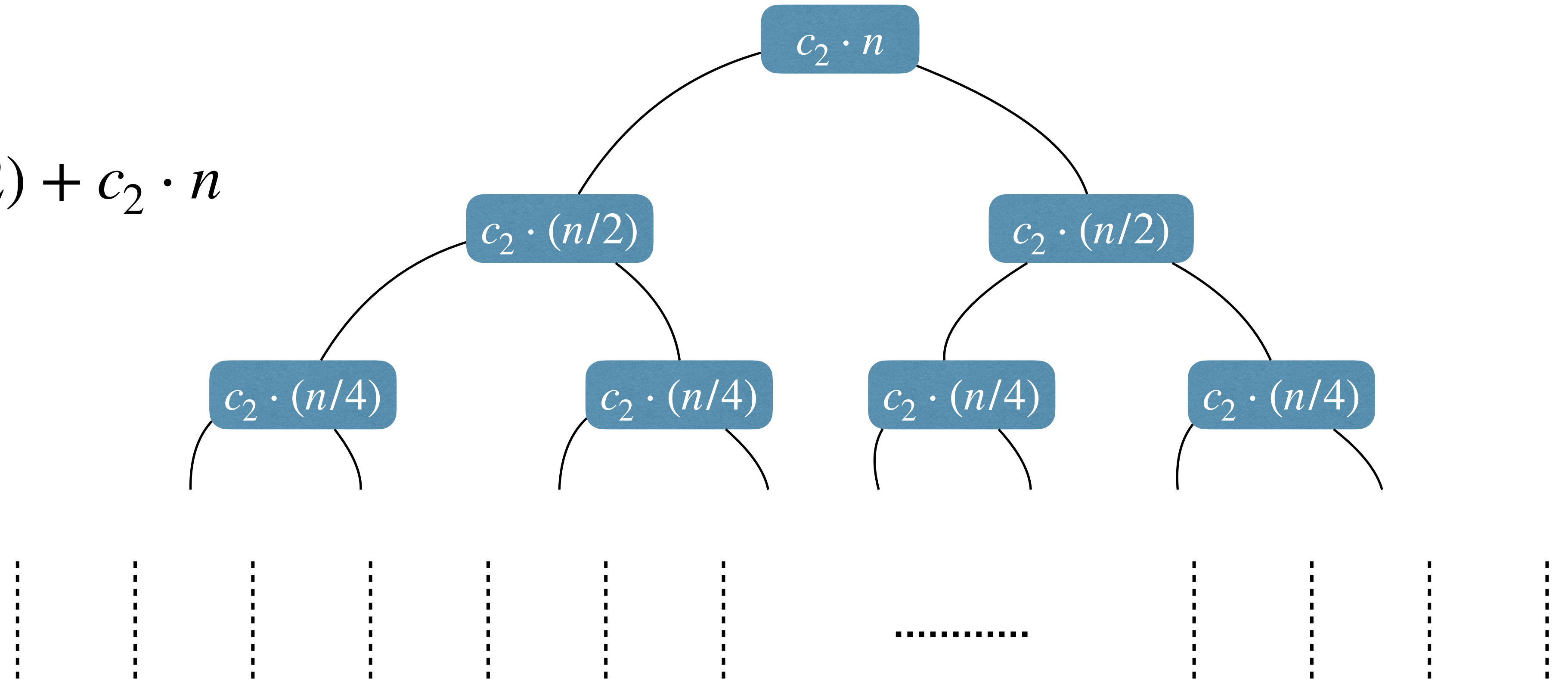




Time complexity of MergeSort

- A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$



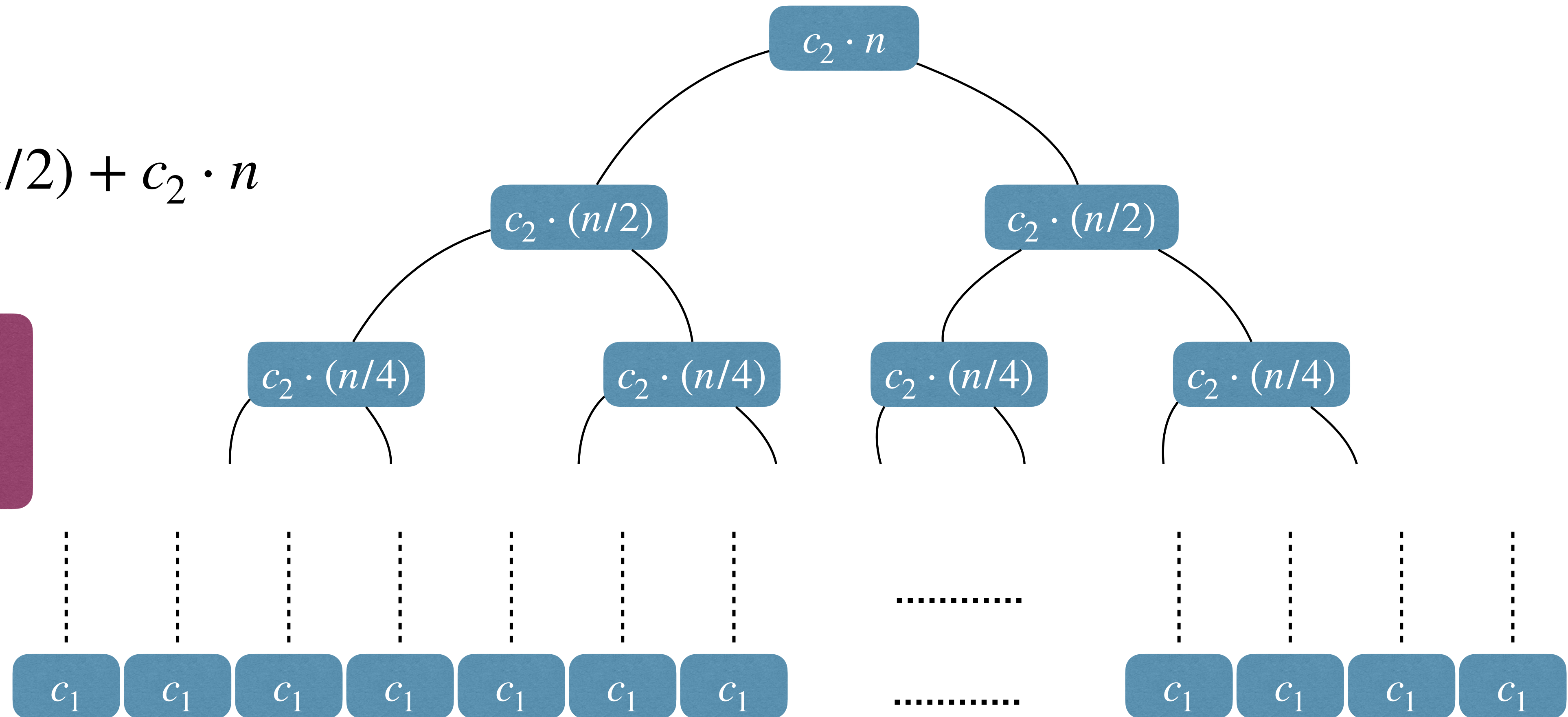


Time complexity of MergeSort

- A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

There are $\log_2 n + 1$ levels
Each level incur $\Theta(n)$
Total cost is $\Theta(n \cdot \log_2 n)$



Recursion tree



Iterative MergeSort

MergeSort ($A[1..n]$):

```
if  $n = 1$ :  
     $sol[1..n] := [1..n]$   
else  
     $solLeft[1..(n/2)] := MergeSort(A[1..(n/2)])$   
     $solRight[1..(n/2)] := MergeSort(A[(n/2+1)..n])$   
     $sol[1..n] := Merge(solLeft[1..(n/2)], solRight[1..(n/2)])$   
return  $sol[1..n]$ 
```

- Any recursive algorithm can be converted into an iterative one, we just simulate the call stack!



Iterative MergeSort

IterMergeSort ($A[1..n]$):

Deque Q

for $i := 1$ **to** n

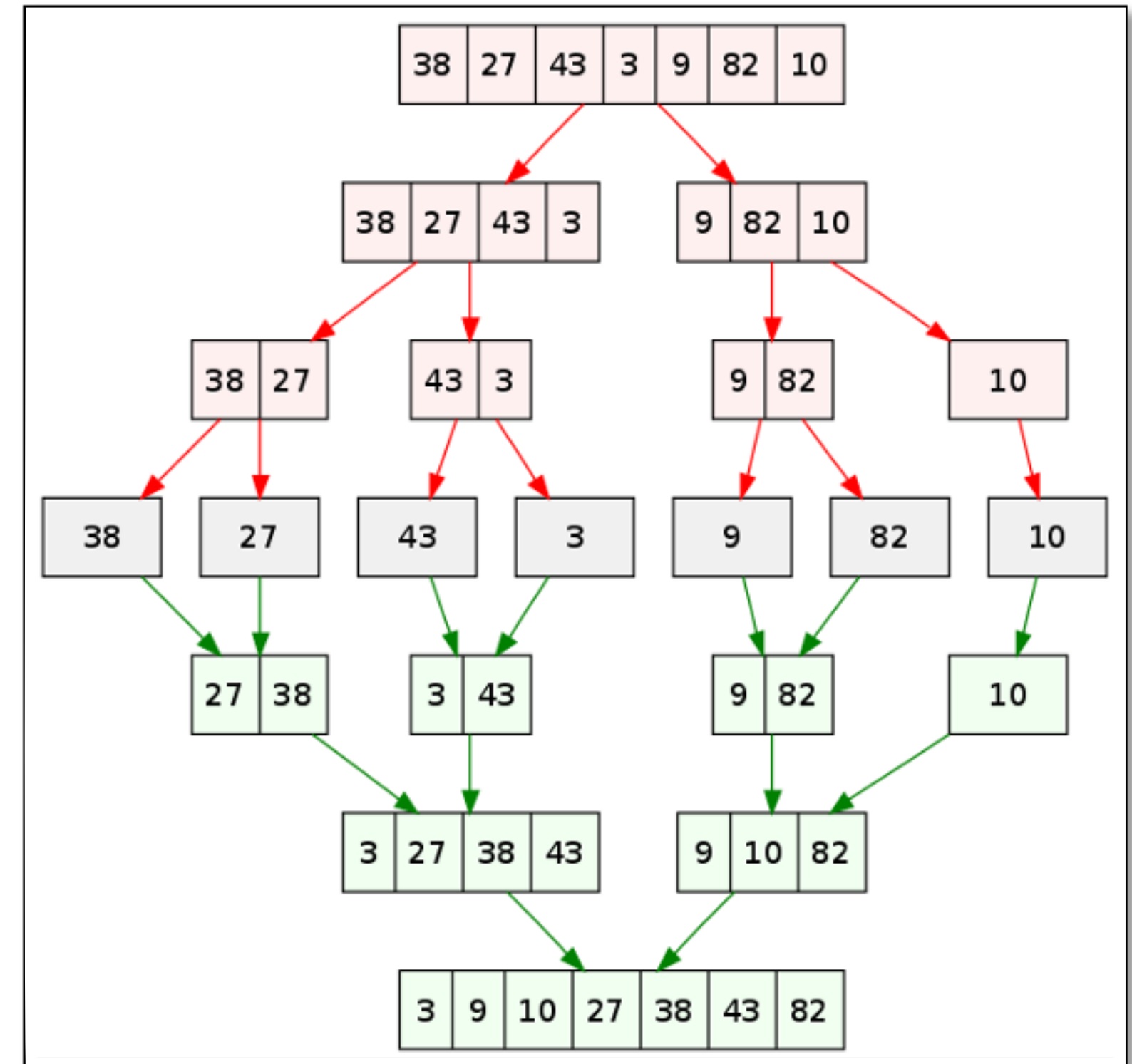
$Q.addLast(A[i])$

while $Q.size() > 1$

$L := Q.removeFirst(), R := Q.removeFirst()$

$Q.AddLast(Merge(L, R))$

return $Q.removeFirst()$



Do “Merge” operation layer by layer!

The time complexity is $\Theta(n \cdot \log n)$



Matrix Multiplication





Matrix Multiplication

- Suppose we want to multiply two $n \times n$ matrices X and Y .
- The most straightforward method needs $\Theta(n^3)$ time.
- Matrix multiplication can be performed block-wise!

$$\triangleright X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$\triangleright XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$



Matrix Multiplication

- $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$
- $XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$
- The recurrence equation is $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$
- Thus, $T(n) = \Theta(n^3)$, which has no improvement...



Strassen's algorithm for Matrix Multiplication

- $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$

- $XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$

▸ where:

$$P_1 = A(F - H), P_2 = (A + B)H, P_3 = (C + D)E, P_4 = D(G - E)$$

$$P_5 = (A + D)(E + H), P_6 = (B - D)(G + H), P_7 = (A - C)(E + F)$$

- Recurrence: $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$



Invented by Volker Strassen at 1969



Time complexity of Strassen's algorithm

- The **substitution** method (or, **guess and verify**)
 - Guess the form of the solution;
 - Use induction to find proper constants and prove the solution works



Time complexity of Strassen's algorithm

- Recurrence: $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$
- $T(n) = 7 \cdot T(n/2) + cn^2, T(1) = c$
- Let's guess $T(n) \leq d \cdot n^{\log_2 7} = O(n^{\log_2 7})$
- **Induction basis:**
 - $T(1) = c \leq d \cdot 1^{\log_2 7}$, as long as $d \geq c$
- **Inductive step:**
 - $T(n) = 7 \cdot T(n/2) + cn^2 \leq 7d(n/2)^{\log_2 7} + cn^2 = dn^{\log_2 7} + cn^2$

Inconsistent!



Time complexity of Strassen's algorithm

- $T(n) = 7 \cdot T(n/2) + cn^2, T(1) = c$
- The guess $T(n) \leq d \cdot n^{\log_2 7} = O(n^{\log_2 7})$ does not work out...
- However, in fact, $O(n^{\log_2 7})$ is the right answer...
 - So we add some lower order term (such as n^2) to our guess?
 - No, we should **subtract** some lower order term from our guess!
 - **Subtraction** gives us stronger induction hypothesis to work with!



Time complexity of Strassen's algorithm

- $T(n) = 7 \cdot T(n/2) + cn^2, T(1) = c$
- Guess $T(n) \leq dn^{\log_2 7} - d'n^2 = O(n^{\log_2 7})$
- **Induction basis:**
 - $T(1) = c \leq d \cdot 1^{\log_2 7} - d' \cdot 1^2$, as long as $d - d' \geq c$
- **Inductive step:**
 - $$\begin{aligned} T(n) &= 7 \cdot T(n/2) + cn^2 \leq 7d(n/2)^{\log_2 7} - 7d'(n/2)^2 + cn^2 \\ &= dn^{\log_2 7} - (7d'/4 - c)n^2 \leq dn^{\log_2 7} - d'n^2, \text{ as long as } 3d'/4 \geq c \end{aligned}$$



Making a good guess

- There is no general way to correctly guess the tightest asymptotic solution to an arbitrary recurrence.
- Making a good guess takes experience and, occasionally, creativity.
- Sometimes need to repeat the guessing process (first determine loose upper and lower bounds on the recurrence and then reduce your range of uncertainty)



Further reading

- [CLRS] Ch.2 (2.3), Ch.4
- [Erickson] Ch.1 (excluding 1.5 and 1.8)

