DIVIDE AND CONQUER

INTRODUCTION

- Divide-and-conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these sub-problems into an overall solution.

MERGE SORT

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

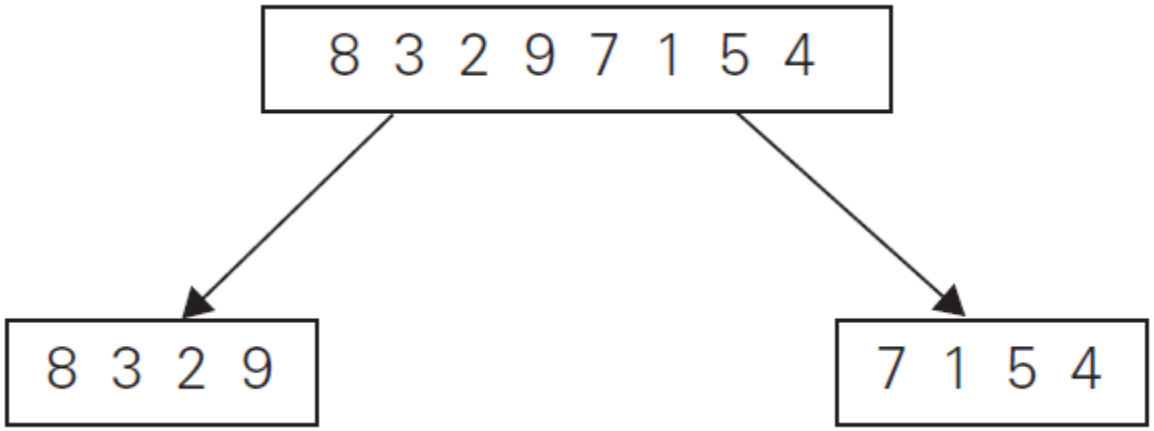
MERGE SORT

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

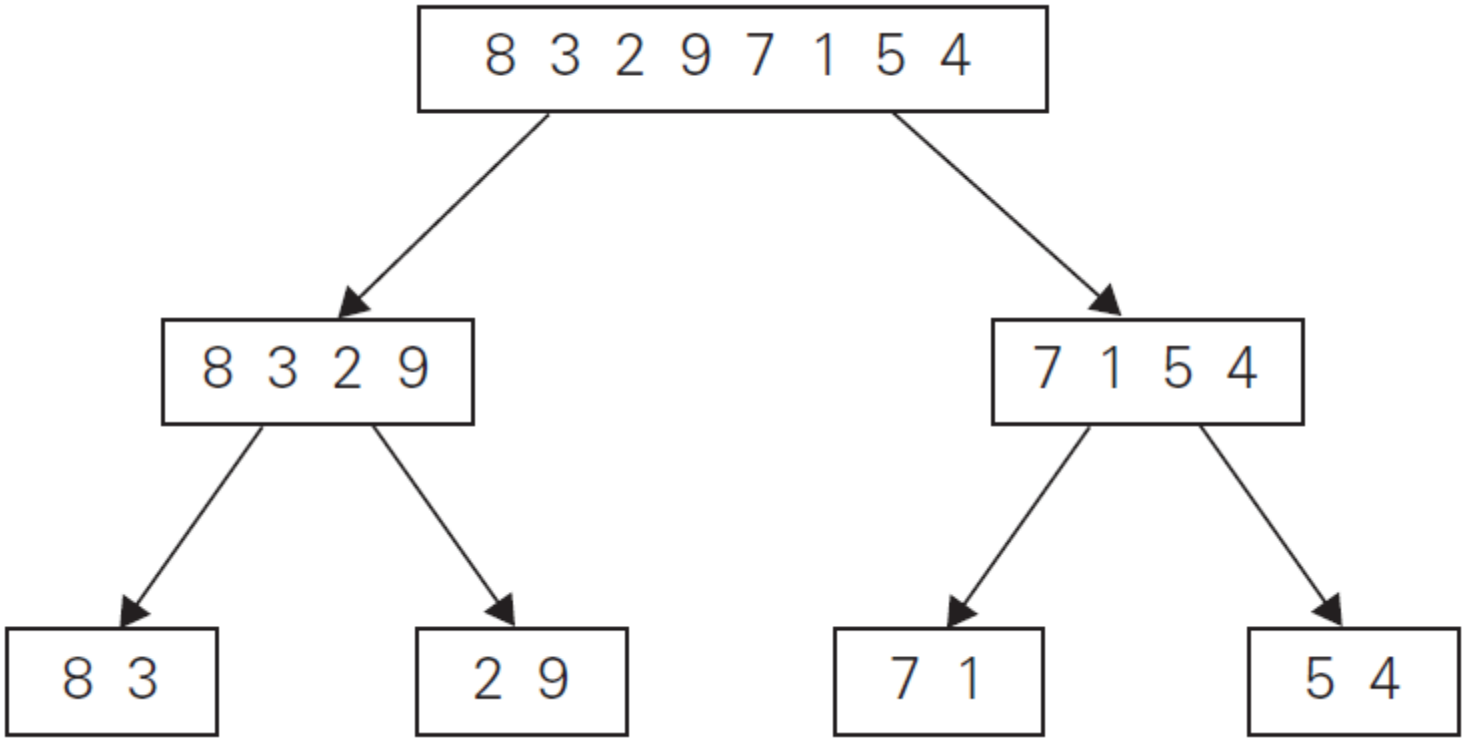
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ \ j \leftarrow 0;\ \ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ \ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ \ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

MERGE SORT

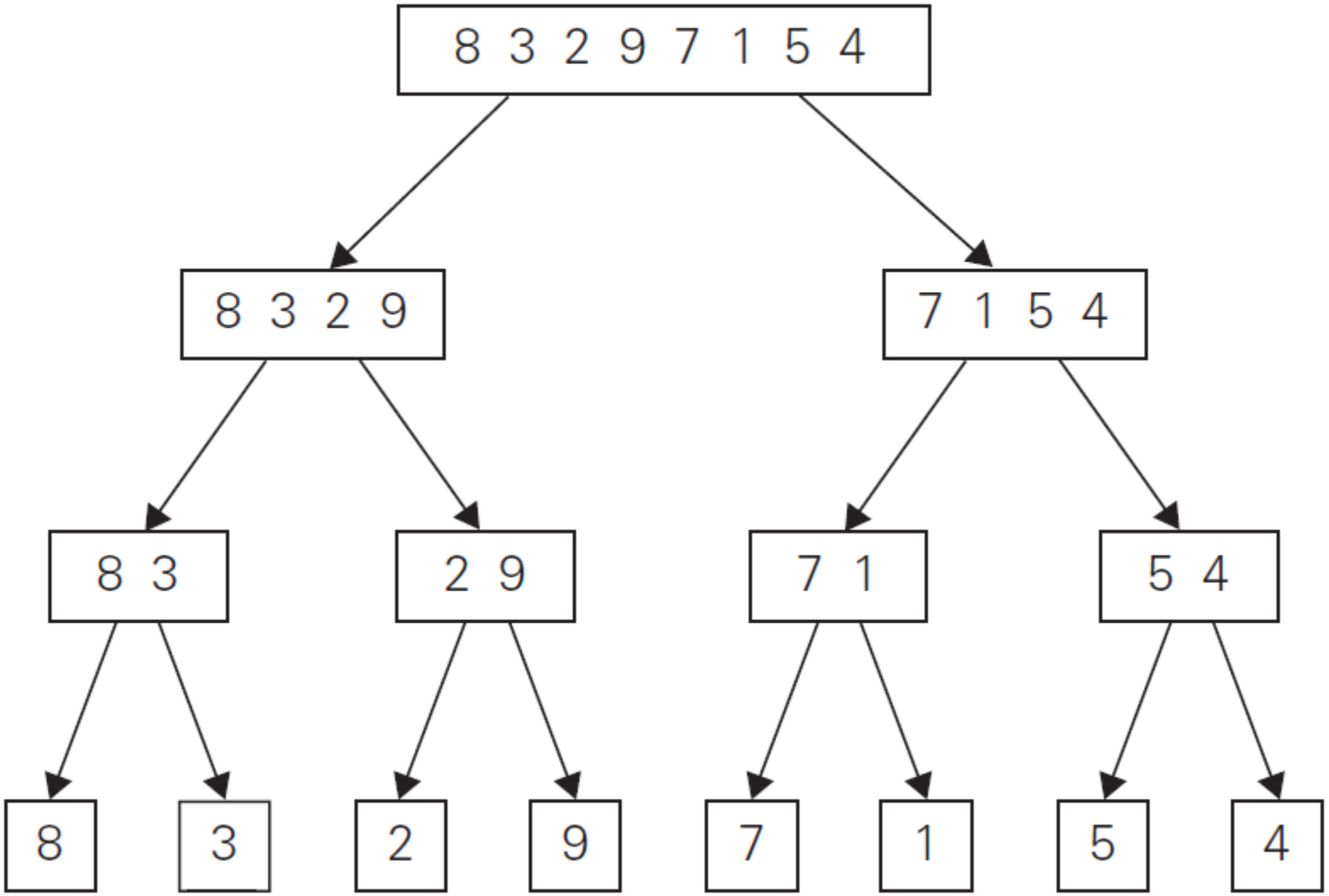
MERGE SORT



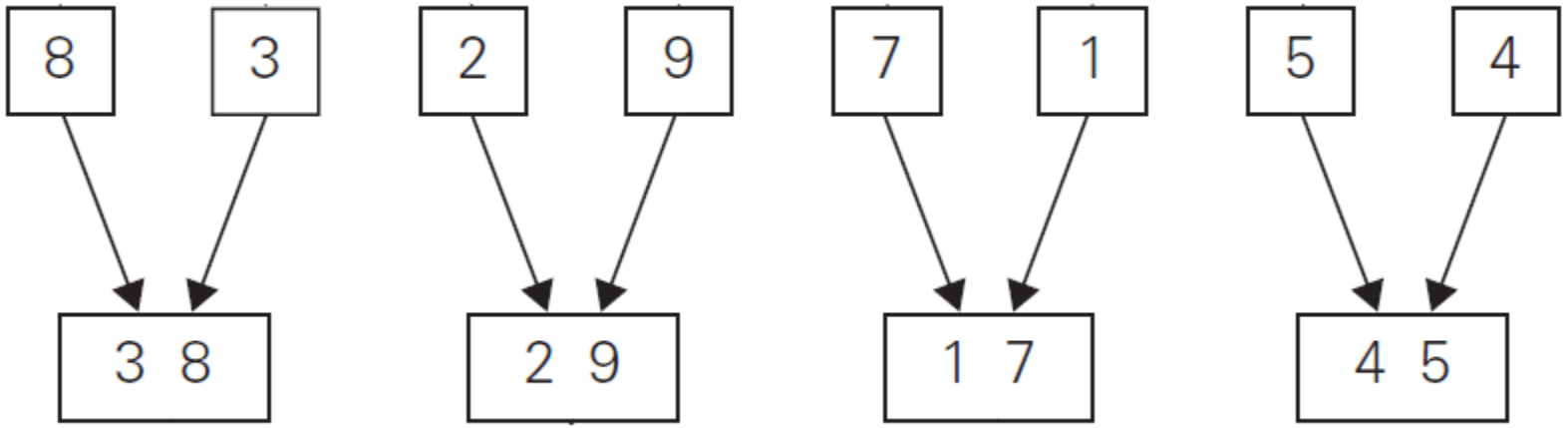
MERGE SORT



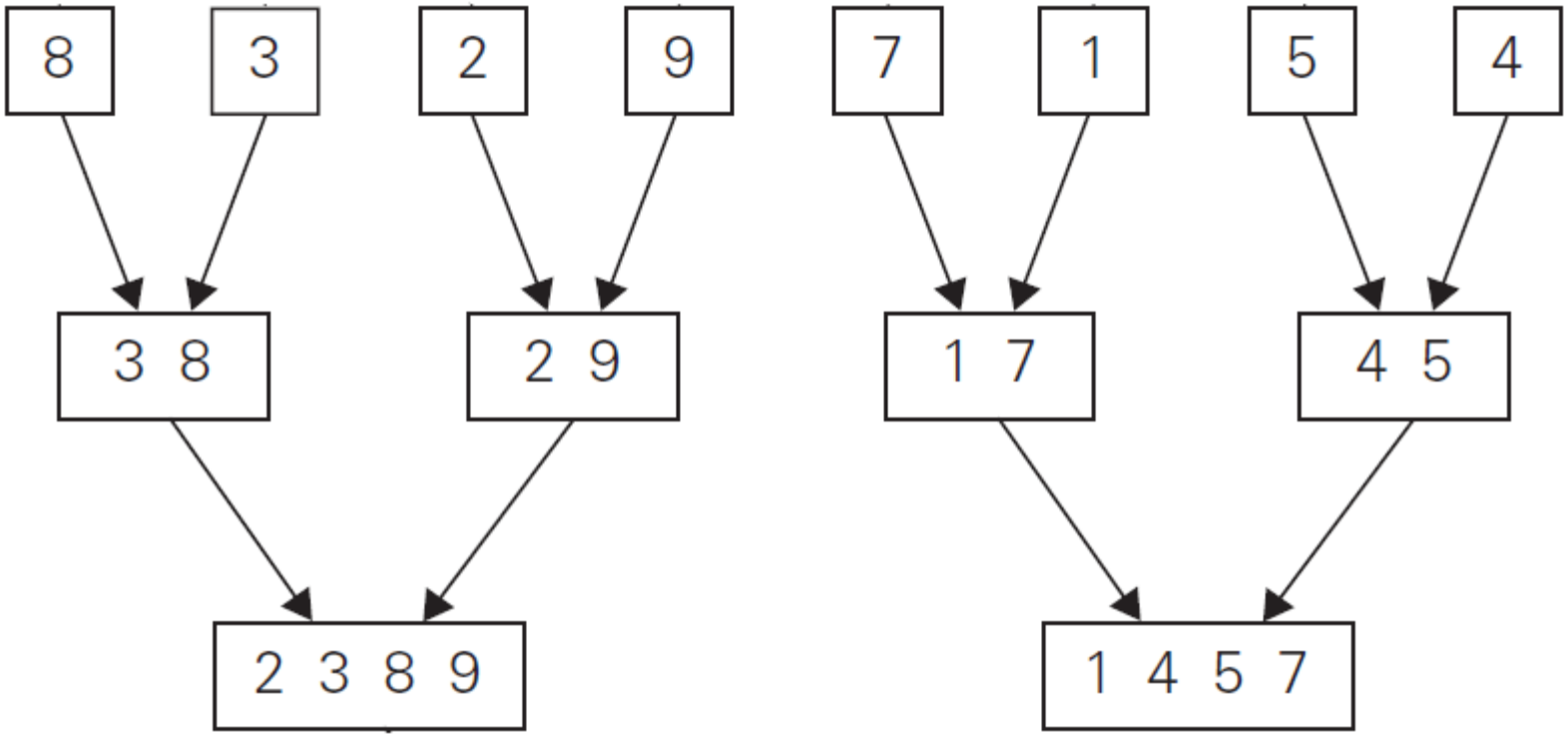
MERGE SORT



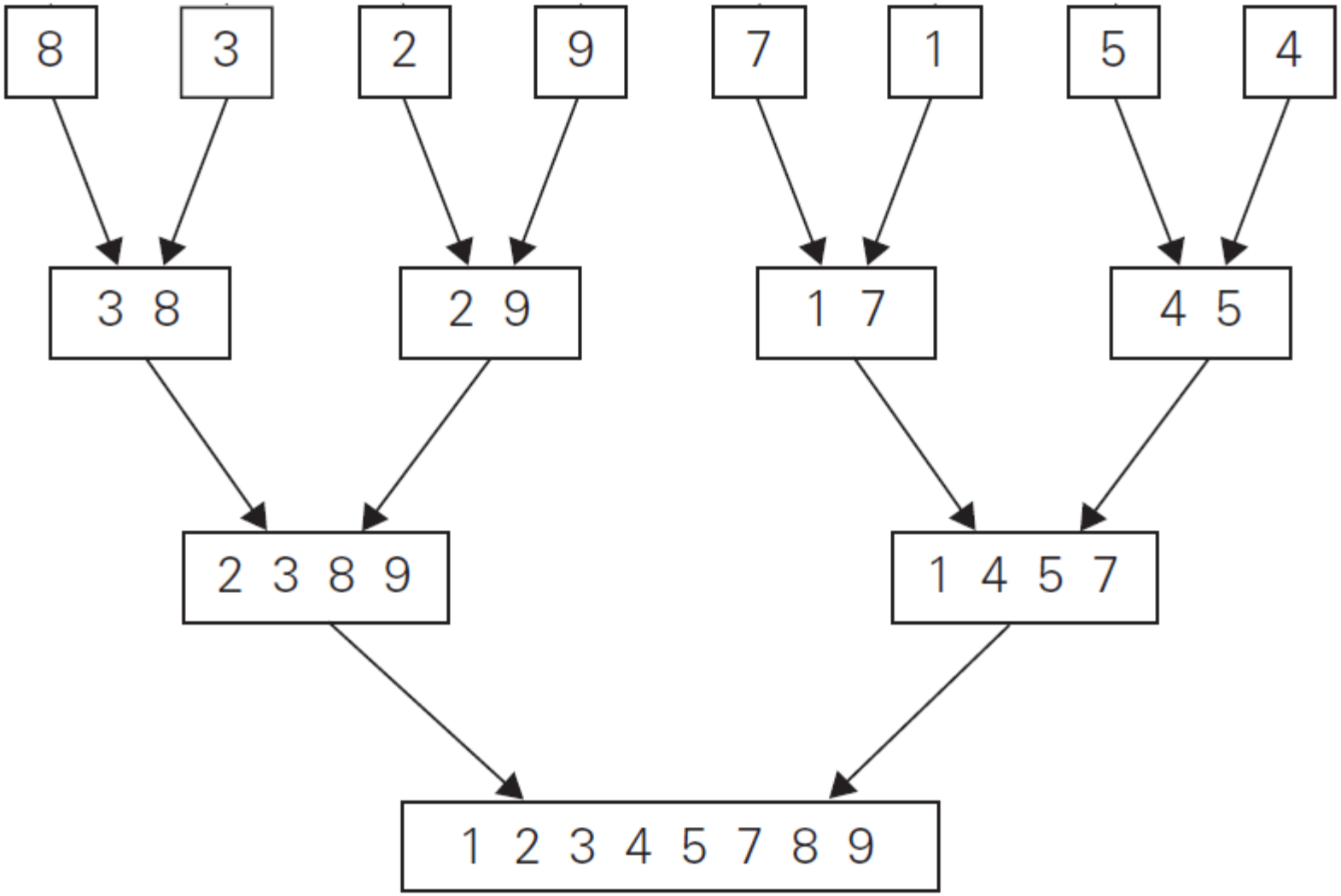
MERGE SORT



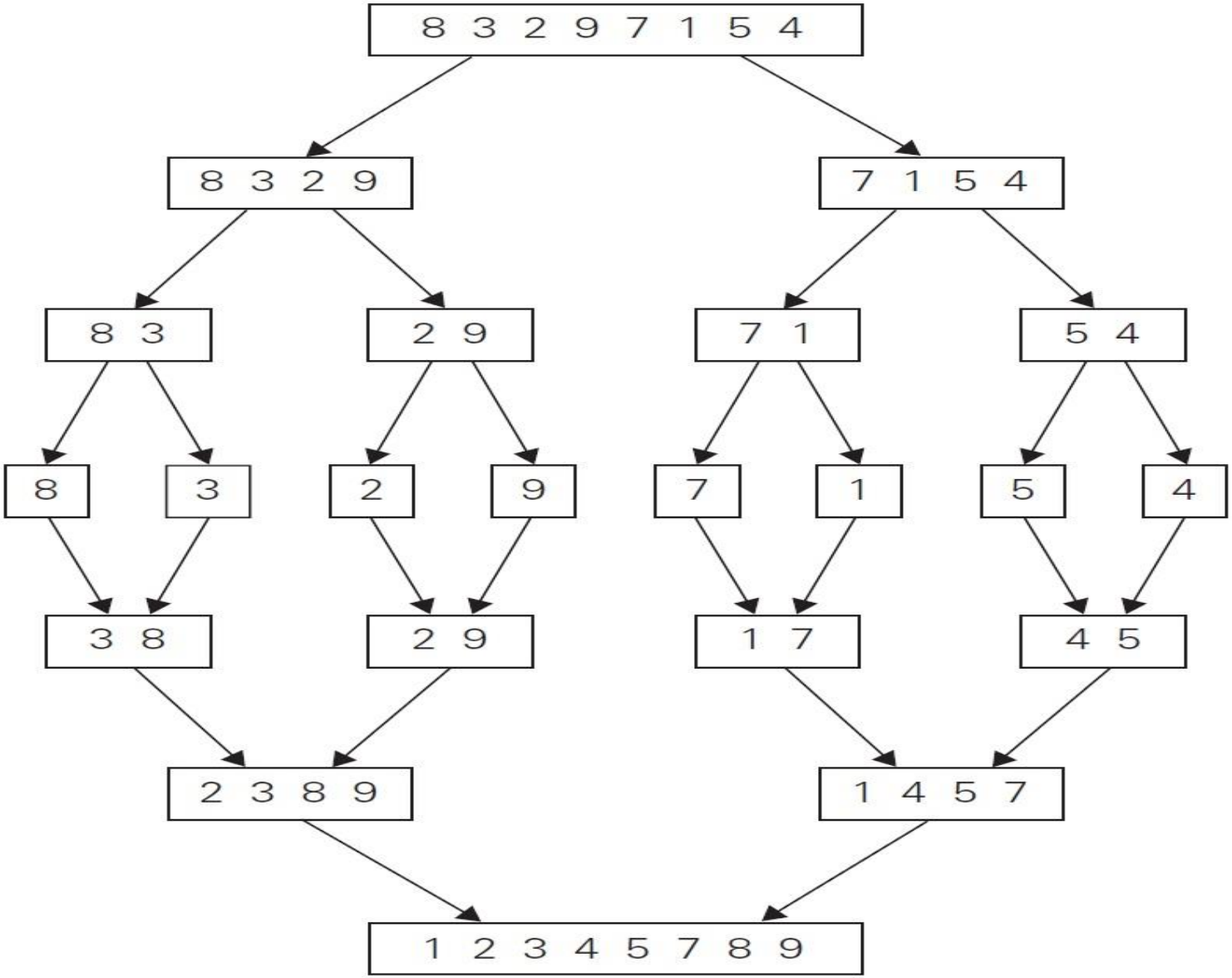
MERGE SORT



MERGE SORT



MERGE SORT

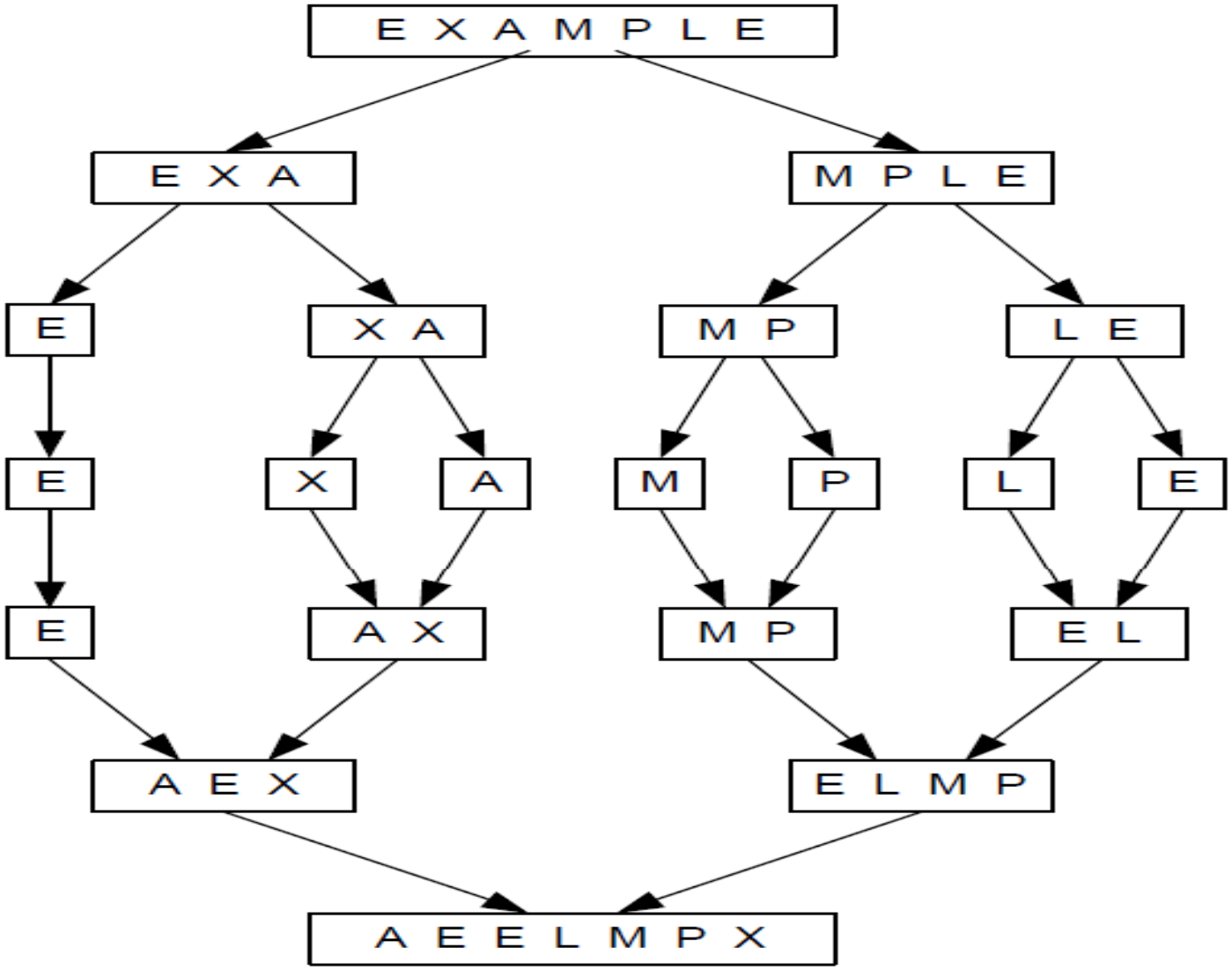

8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

MERGE SORT

Sort the following list using Merge sort

E	X	A	M	P	L	E

MERGE SORT



MERGESORT ANALYSIS

APPROACHES TO SOLVE RECURSION

Approach 1:

1. Intuitive solution to recurrence is to “unroll” the recursion, accounting for the running time of first few levels.

2. Identify a pattern that can be continued as the recursion expands.

3. Sum the running times over all levels of the recursion and thereby arrives at a total running time.

Step1: Analyze the first few levels.

- 1st level of recursion → Single problem of size n → O(n)
- 2nd level of recursion → 2 problems each of size n/2 → O(n/2)
- 3rd level of recursion → 4 problems each of size n/4 → O(n/4)

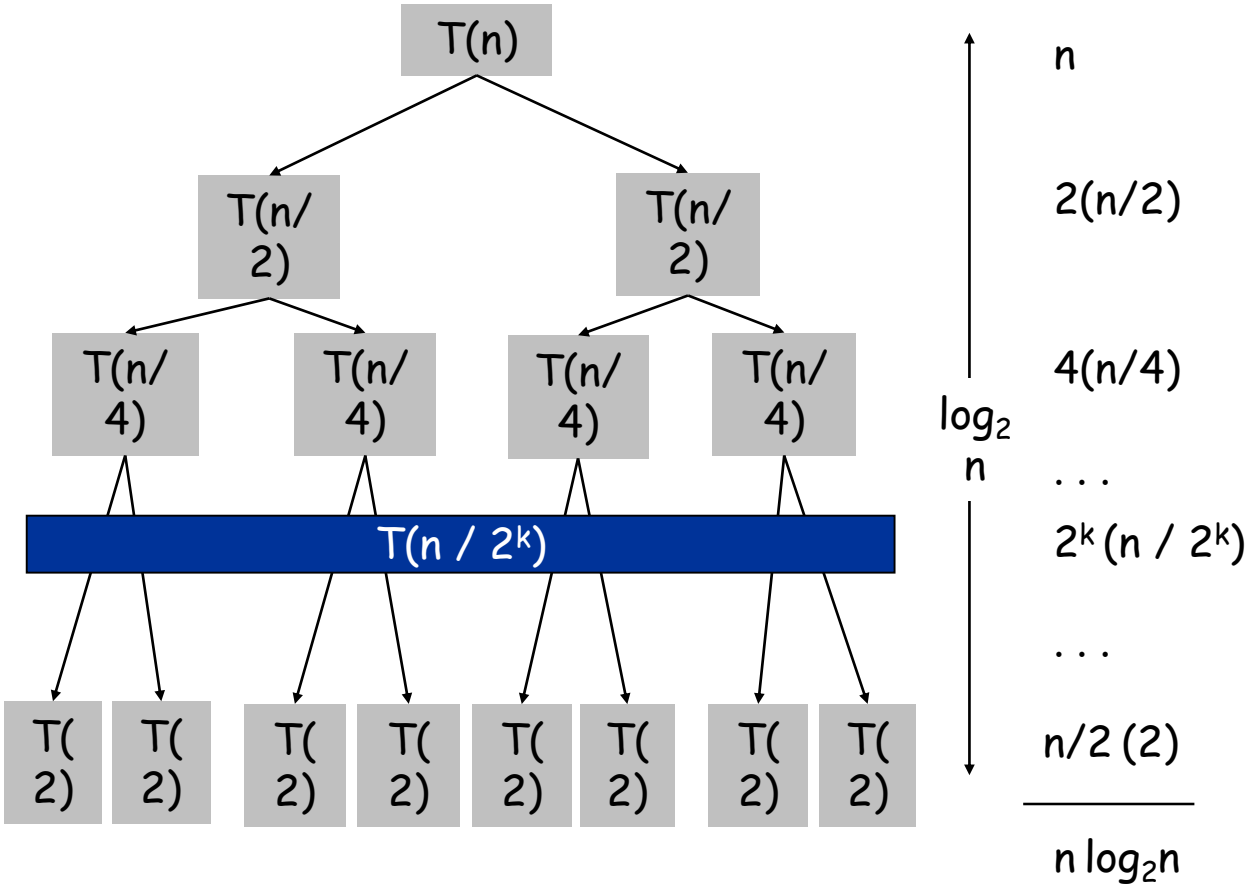
Step 2: Identifying the pattern.

- At level *j* of the recursion, the number of subproblems are now a total of $\mathbf{2^j}$
- Each problem has shrunk in size by factor of 2 "j" time → n/ $\mathbf{2^j}$

Step 3: Summing overall levels of recursion.

- The number of times the input must be halved to reduce the size of n to 2 is **log n**
- There are totally "n" levels of recursion → **O(n logn)**

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



SUBSTITUTING A SOLUTION INTO THE MERGESORT RECURSION

T(n) = 2T(n/2) + n .....(1)

T(n/2) = 2T(n/4) + n/2 .....(2)

Put (2) in (1)

$T(n) = 2\{ 2T(n/4) + n/2 \} + n$

$= 2^2T(n/4) + n + n$

$= 2^3T(n/8) + n + n + n$

$= 2^4T(n/16) + n + n + n + n$

$= 2^kT(n/2^k) + n + n + n + n + n$.....(*k times n will be present*)

$= 2^kT(n/2^k) + nk$

Now put $2^k = n$, so k will be $\log_2 n$

So, $T(n) = nT(2^k/2^k) + n\log_2 n$

$= nT(1) + n\log_2 n$

$= n + n\log_2 n$

Hence, ***T(n) = O(nlog$_2$n)***

COUNTING INVERSION

COUNTING INVERSION

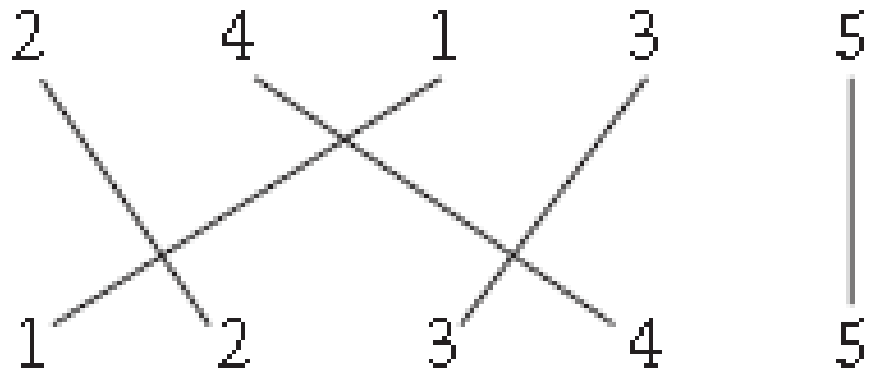
- **Introduction**
- We are given a sequence of n numbers a1, . . . , an.
- We will assume that all the numbers are distinct.
- We want to define a measure that tells us how far this list is from being in ascending order.
- The value of the measure should be 0 if a1< a2 < . . .< an, and should increase as the numbers become more scrambled.

COUNTING INVERSION

- **Counting the Number of Inversions**
- The two indices i < j form an inversion if ai > aj, that is, if the two elements ai and aj are “out of order.”
- Counting the number of inversions is to determine the number of inversions in the sequence a1, . . . , an.

COUNTING INVERSION

- **Example**
- Sequence is 2, 4, 1, 3, 5.
- There are three inversions in this sequence: (2, 1), (4, 1), and (4, 3).



COUNTING INVERSION

- **Example**
- If the sequence is in descending order, then every pair forms an inversion, and so there are **n(n-1)/2** pairs.

5	4	3	2	1

{5,4}, {5,3},{5,2},{5,1}
{4,3}, {4,2}, {4,1}
{3,2}, {3,1}
{2,1}

COUNTING INVERSION

- **Algorithm**
- Look at every pair of numbers (ai , aj) and determine whether they constitute an inversion.
- This would take O(n^**2**) time.
- The basic idea is to follow the strategy of divide and conquer.
- We set m = _n/2_ and divide the list into the two pieces a1, . . . , am and am+1, . . . , an.
- We first count the number of inversions in each of these two halves separately.
- Then we count the number of inversions (ai , aj), where the two numbers belong to different halves.

COUNTING INVERSION

- **Algorithm**
- Note that these first-half/second-half inversions have a particularly nice form: they are precisely the pairs (ai , aj), where ai is in the first half, aj is in the second half.
- Suppose we have recursively sorted the first and second halves of the list and counted the inversions in each.
- We now have two sorted lists A and B, containing the first and second halves, respectively.
- We want to produce a single sorted list C from their union, while also counting the number of pairs (a, b) with $a \in A$, $b \in B$, and $a > b$.

COUNTING INVERSION

- **Algorithm: Counting Inversions**

//Purpose: To count the inversions for a given list L(a1,a2,…….an)

//Input: An unsorted list of distinct numbers L(a1,a2,…….an)

//Output: The number of inversions r for the list L(a1,a2,…….an)

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements.
         B contains the remaining ⌊n/2⌋ elements.
      (r_A, A)=Sort-and-Count(A)
      (r_B, B)=Sort-and-Count(B)
      (r, L)=Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

COUNTING INVERSION

```
Merge-and-Count(A,B)
 Maintain a Current pointer into each list, initialized to point to the front
 elements
 Maintain a variable Count for the number of inversions, initialized to 0

 While both lists are nonempty:
        Let ai and bj be the elements pointed to by the Current pointer
        Append the smaller of these two to the output list
        If bj is the smaller element then
                Increment Count by the number of elements remaining in A
        Endif
        Advance the Current pointer in the list from which the smaller element
        was selected.
 EndWhile

 Once one list is empty, append the remainder of the other list to the output
 Return Count and the merged list
```

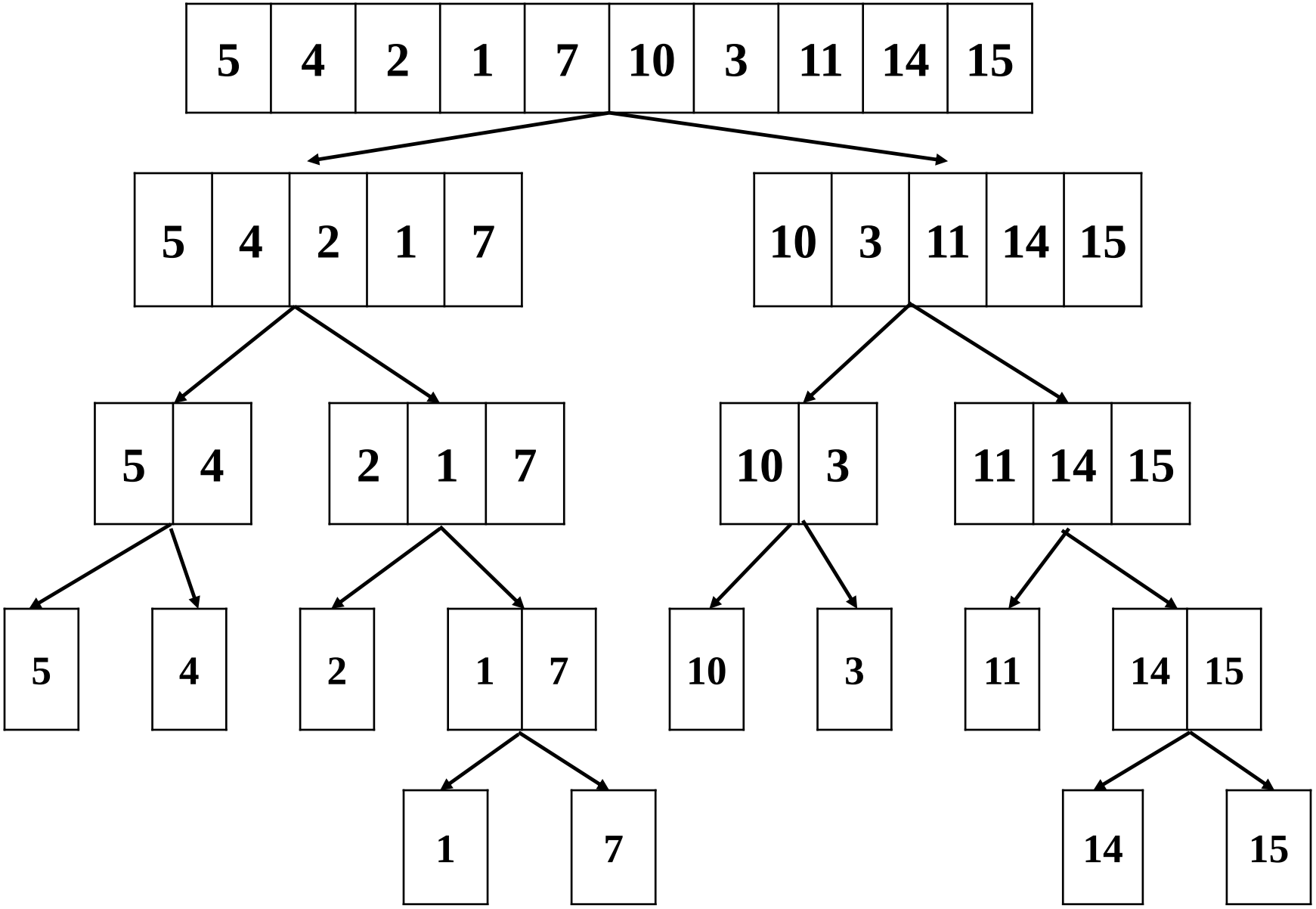
COUNTING INVERSION

- **EXAMPLE**

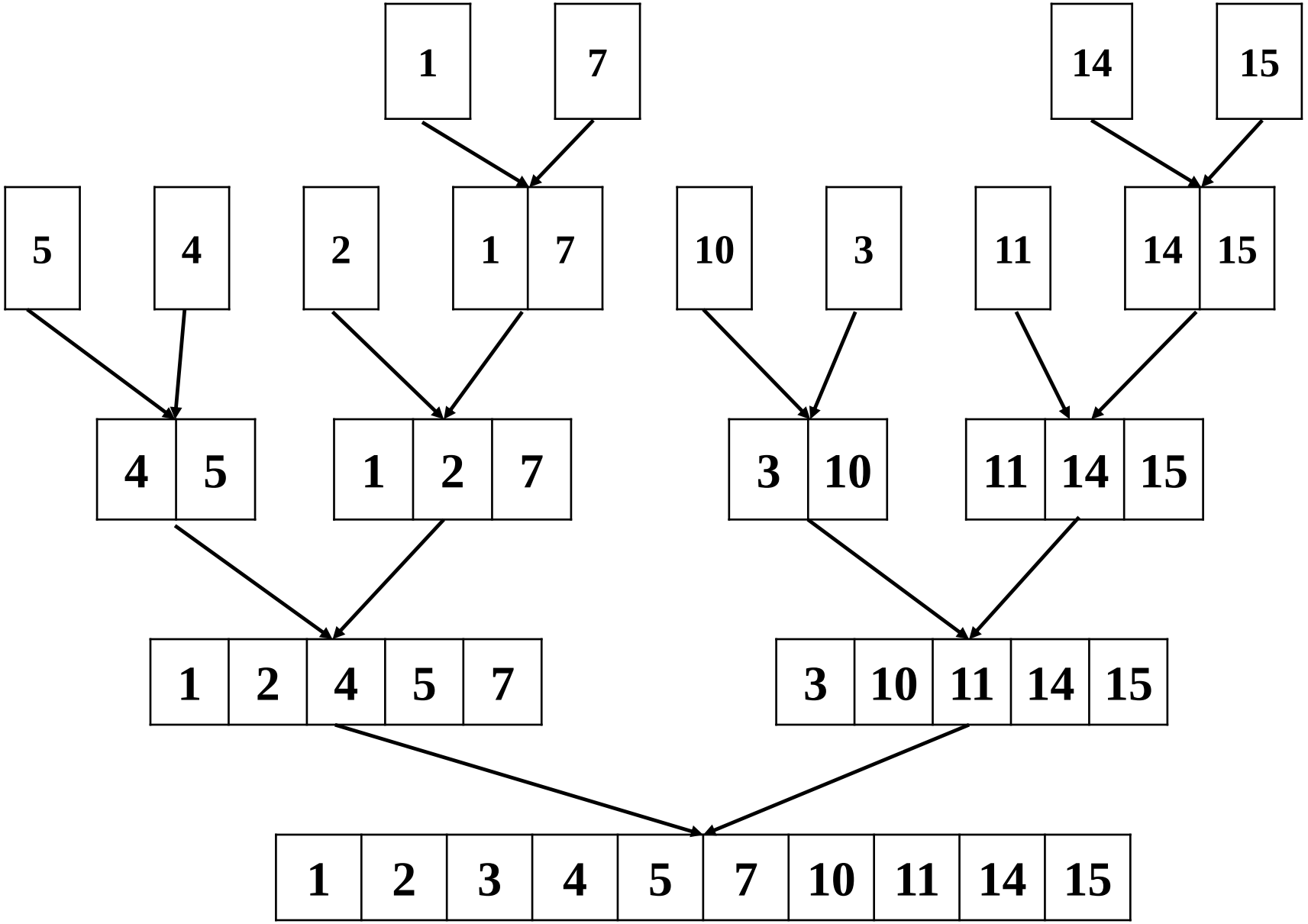
5	4	2	1	7	10	3	11	14	15

- **Total Inversions: 10**

COUNTING INVERSION



COUNTING INVERSION



THANK YOU