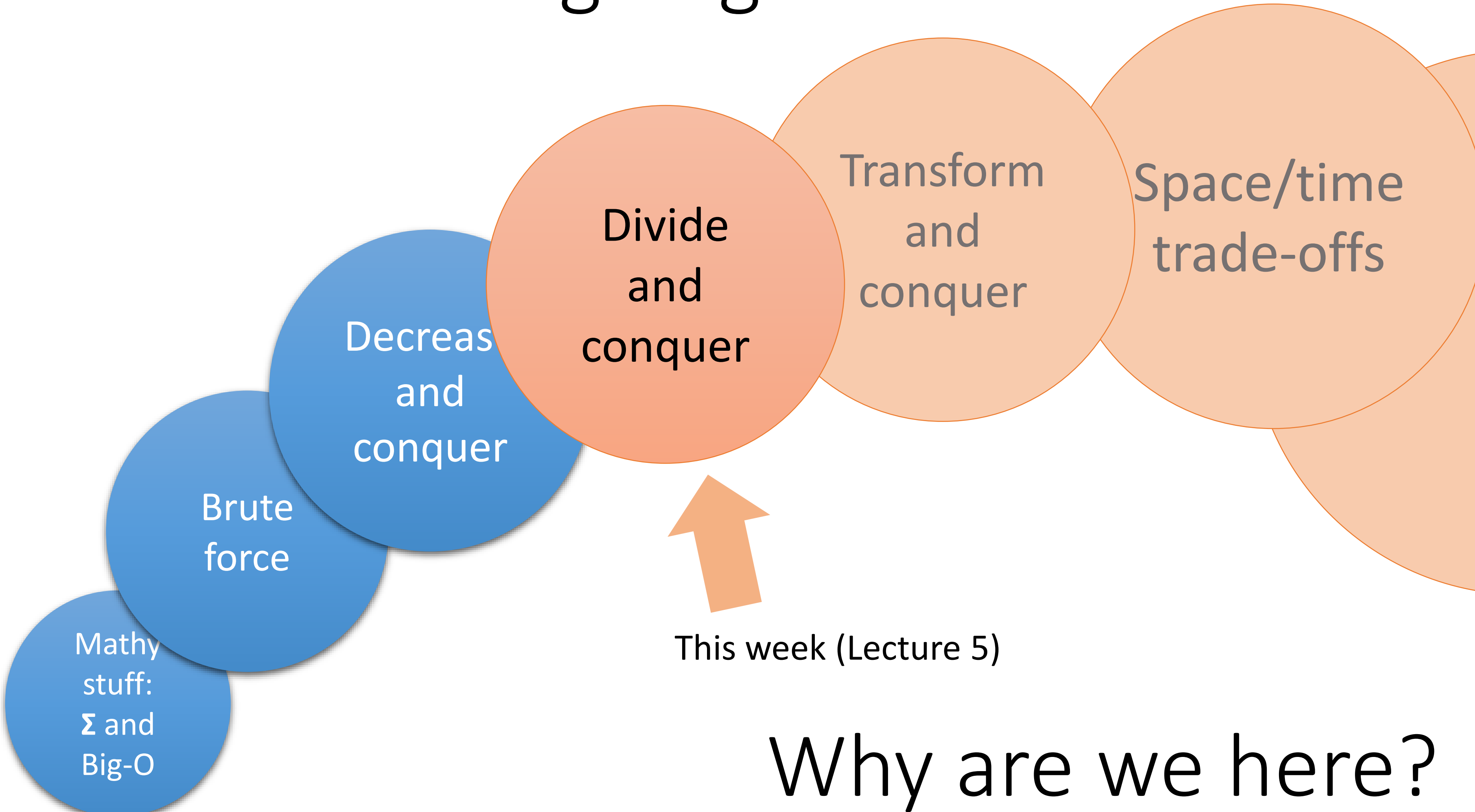
Where have we been?
Where are we going?



Why are we here?
No. Too deep.

This week:

- *Divide and conquer* algorithms
- Example: Count a key in an array
- How to analyze Divide and Conquer (the “Master Theorem”)
- Example: Mergesort
- Binary tree examples
    - Compute the height
    - Compute the number of leaves

But first …

Divide and conquer algorithms

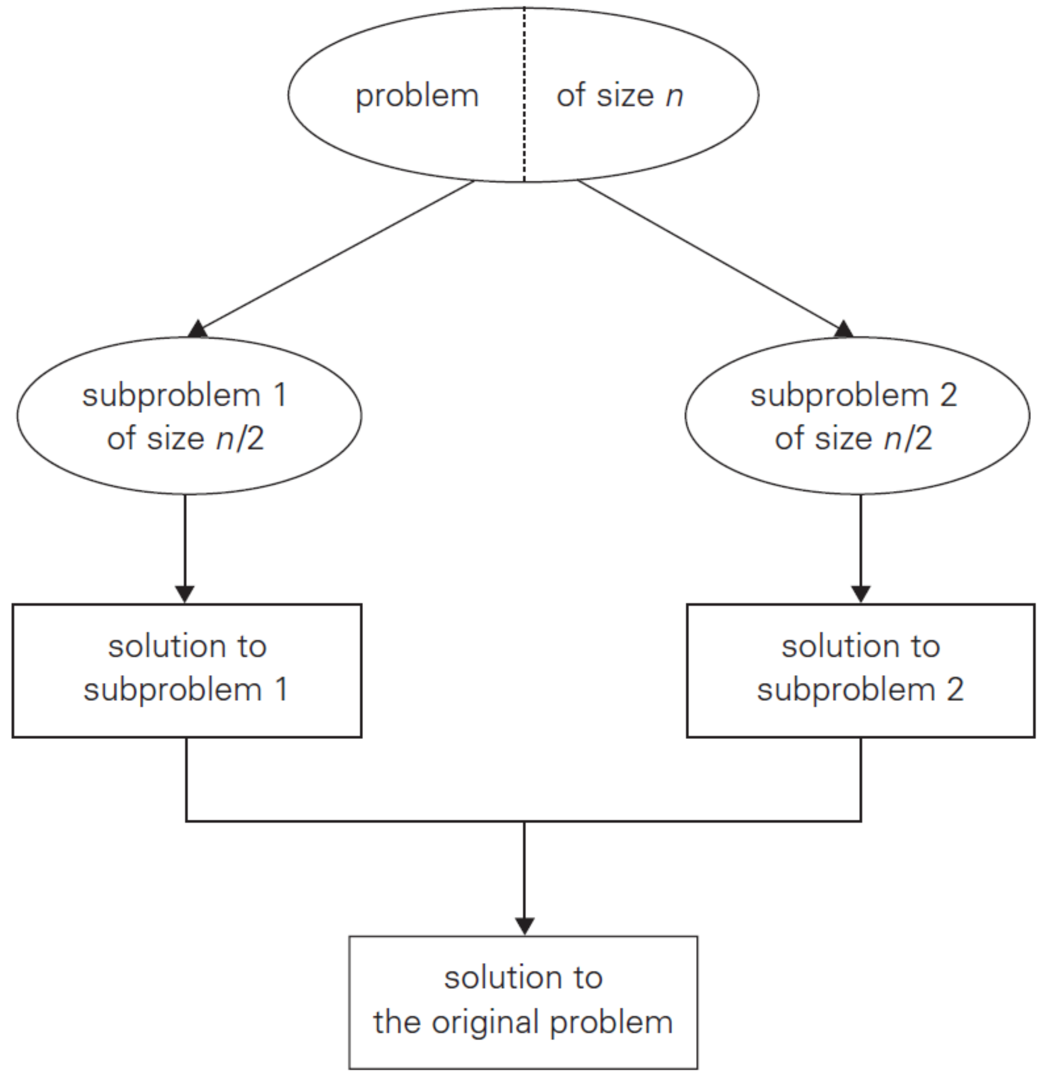
Text sections:

5.1, 5.3

Divide and conquer algorithms

- Divide a problem into two or more smaller instances
- Solve smaller instances (often recursively)
- Obtain solution to original (larger) instance by combining these solutions

Divide and conquer technique



Divide-and-conquer vs. decrease-and-conquer

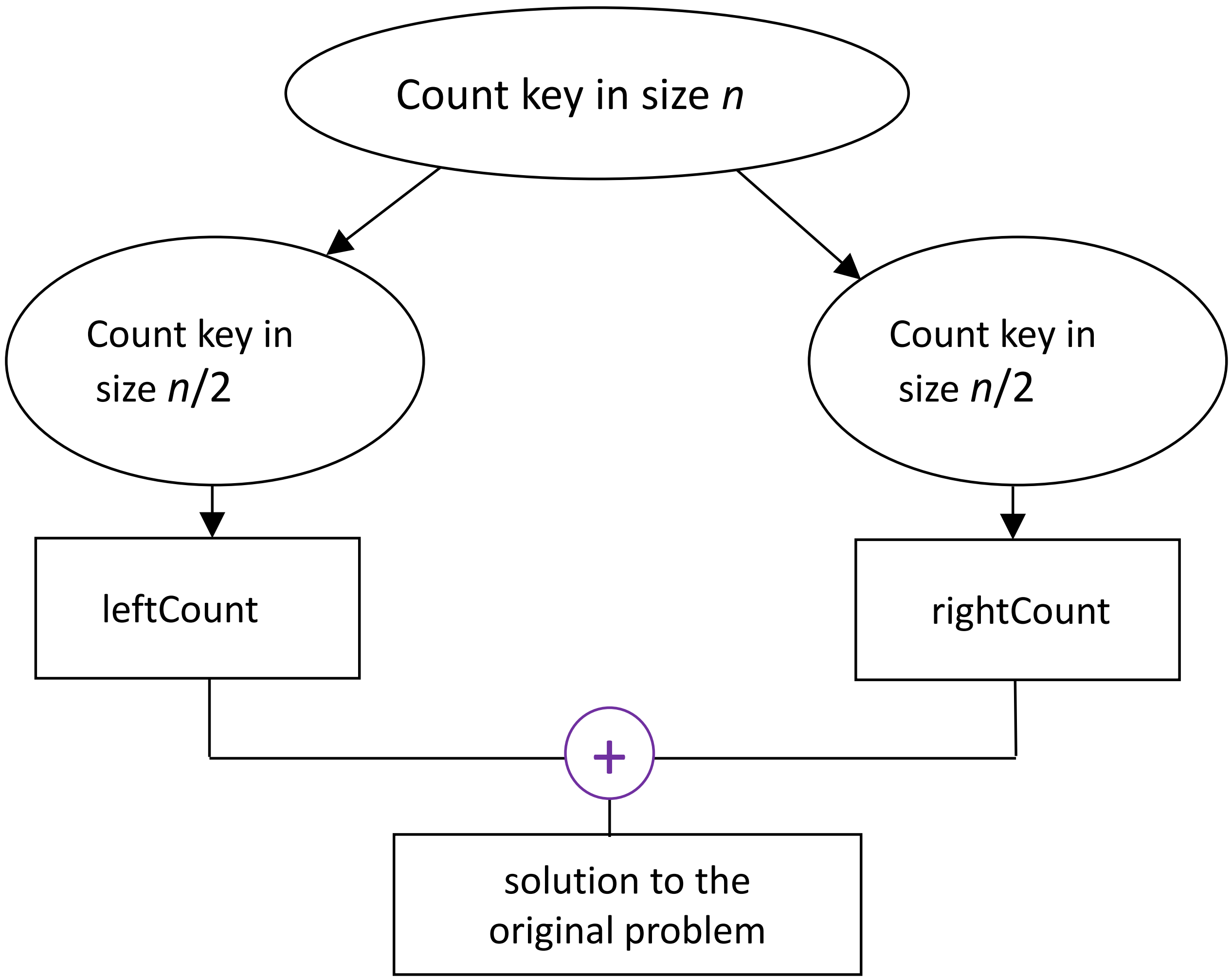
- Think of the fake coin problem (decrease-and-conquer):
  - We discarded half the coins at each step
  - So we didn’t do any work on those “subproblems”
- For divide and conquer…
  - You need to solve all of the subproblems

Example:
Count a key in an array

Count a key in an array

- Problem:
  - Count the number of times a specific key occurs in an array.
- For example:
  - If input array is A=[2,7,6,6,2,4,6,9,2] and key=6…
  - ... should return the value 3.
- Design an algorithm using divide and conquer technique

Count a key in an array



Count a key in an array

```
Algorithm CountKey(A[], Key, L, R)

//Input: A[] is an array A[0..n−1]
//      L & R (L ≤ R) are boundaries of the current search
//Output: The number of times Key exists in A[L..R]

if L = R
   if (A[L] = Key) return 1
   else return 0
else
   leftCount = CountKey(A[], Key,  L,     ⌊(L+R)/2⌋)
   rightCount = CountKey(A[], Key,  ⌊(L+R)/2⌋+1,  R)
   return leftCount + rightCount
```

Count a key in an array

- CountKey looks familiar…
  - What's the difference between Binary Search and CountKey?
- We have to search both sides
  - In the counter, both sides must be searched
  - In Binary Search, one half gets ignored

Analysis of divide and conquer

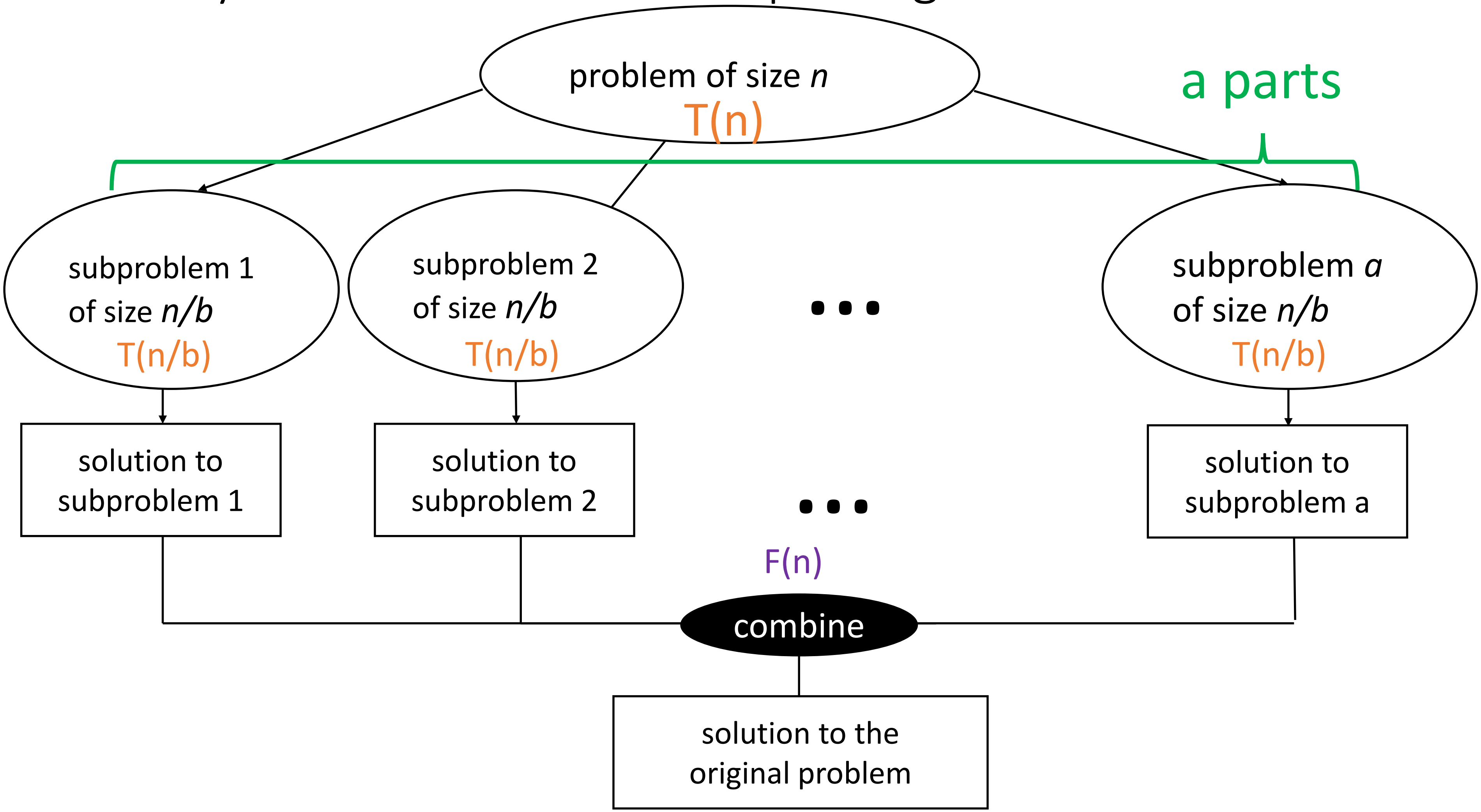
Analyzing a divide-and-conquer algorithm

- What matters:
  1. Number of parts — a
  2. Size of each part — n/b
  3. Cost of combining subproblems — $F(n)$

This expression is your new friend:

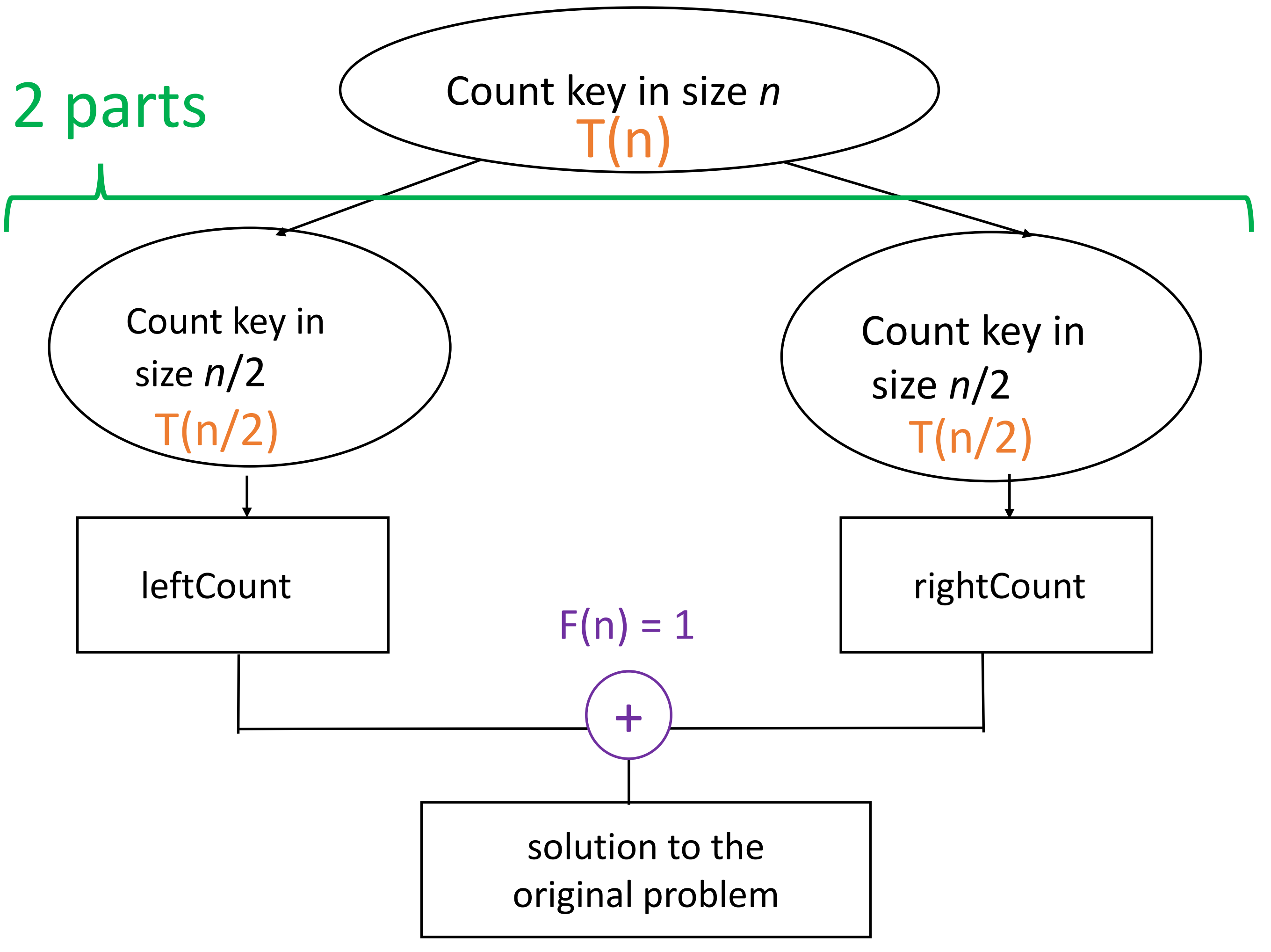
$$n^{\log_b a}$$

Analysis of a divide and conquer algorithm



$$T(n) = a\,T(n/b) + F(n)$$

Example: analysis of Count a key in an array



$$T(n) = 2\,T(n/2) + 1$$

O(*???*)

What is the big-O efficiency class of T(n)?

$T(n) = a\, T(n/b) + F(n)$

1 Compare $n^{\log_b a}$ and F(n)

2a The bigger one wins

2b If they're equal: $O(n^{\log_b a} \log n)$

The Master Theorem

If $T(n) = a\,T(n/b) + F(n)$

1) If $n^{\log_b a} < F(n)$, $\quad T(n) \in O(F(n))$
2) If $n^{\log_b a} > F(n)$, $\quad T(n) \in O(n^{\log_b a})$
3) If $n^{\log_b a} = F(n)$, $\quad T(n) \in O(n^{\log_b a} \log n)$

Master Theorem examples

Example 1: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

a = 4
b = 2
F(n) = n

$\rightarrow n^{\log_b a} \rightarrow n^{\log_2 4} \rightarrow n^2$

F(n) = n

$\rightarrow T(n) \in O(n^2)$

Example 2: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

a = 4
b = 2
F(n) = n^2

$\rightarrow n^{\log_b a} \rightarrow n^{\log_2 4} \rightarrow n^2$

F(n) = n^2

$\rightarrow T(n) \in O(n^2 \log n)$

Example 3: $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

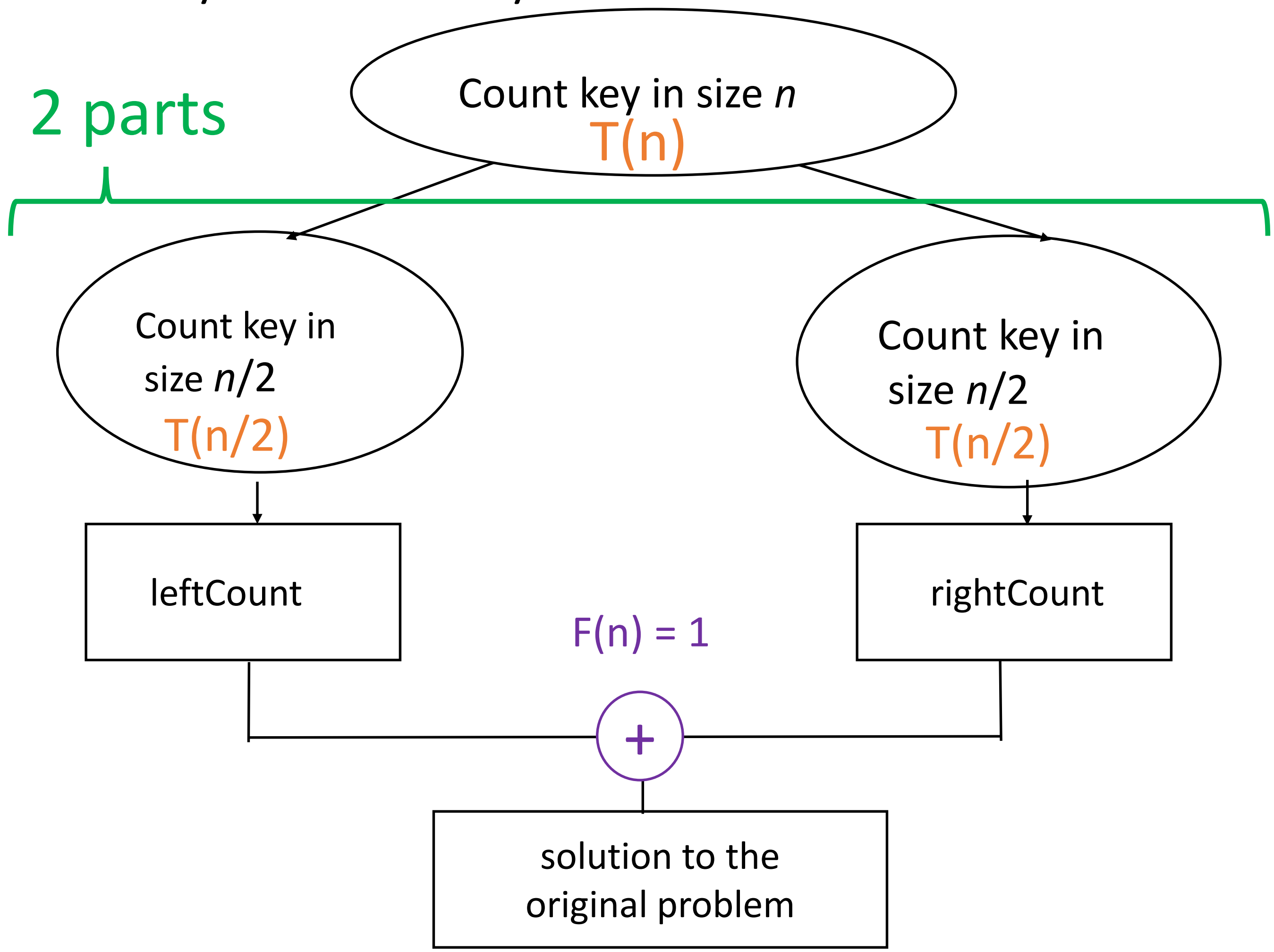
a = 4
b = 2
F(n) = n^3

$\rightarrow n^{\log_b a} \rightarrow n^{\log_2 4} \rightarrow n^2$

F(n) = n^3

$\rightarrow T(n) \in O(n^3)$

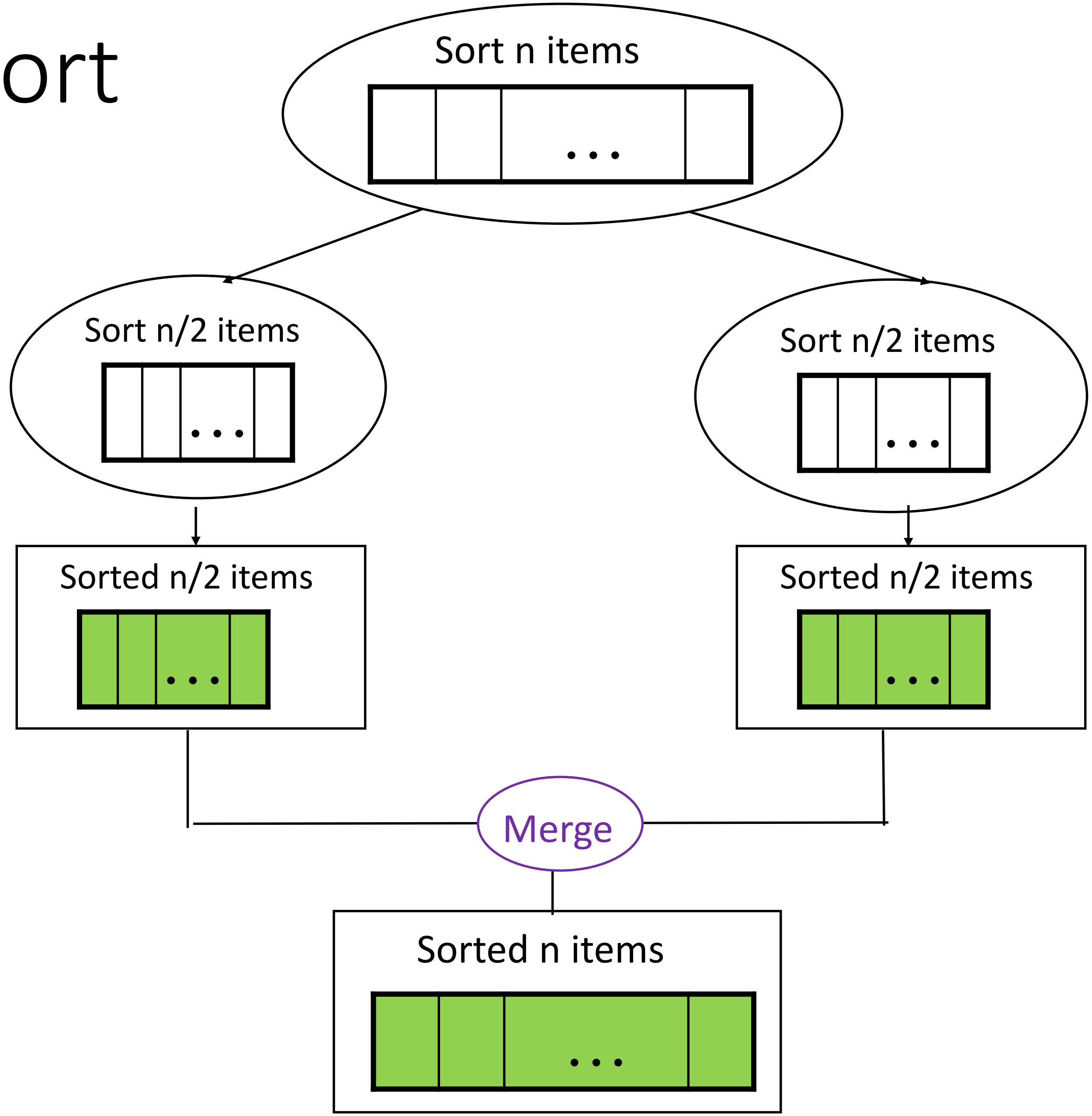
Analysis of
Count a key in an array



$$T(n) = 2\ T(n/2) + 1 \quad \Rightarrow \quad T(n) \in O(n)$$

Mergesort

Mergesort



Sort n items
. . .
Sort n/2 items
. . .
Sort n/2 items
. . .
Sorted n/2 items
. . .
Sorted n/2 items
. . .
Merge
Sorted n items
. . .

Pseudocode of Mergesort

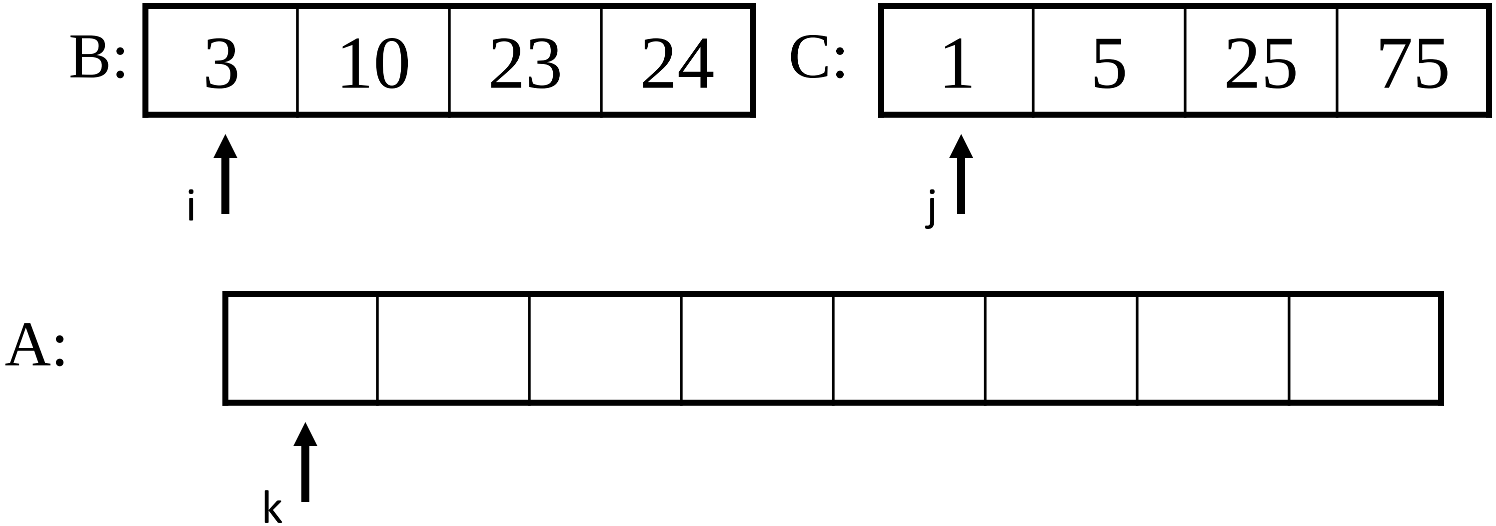
ALGORITHM $Mergesort(A[0..n-1])$

//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 $C[0..\lceil n/2 \rceil - 1]$
    $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
    $Mergesort(C[0..\lceil n/2 \rceil - 1])$
    $Merge(B, C, A)$

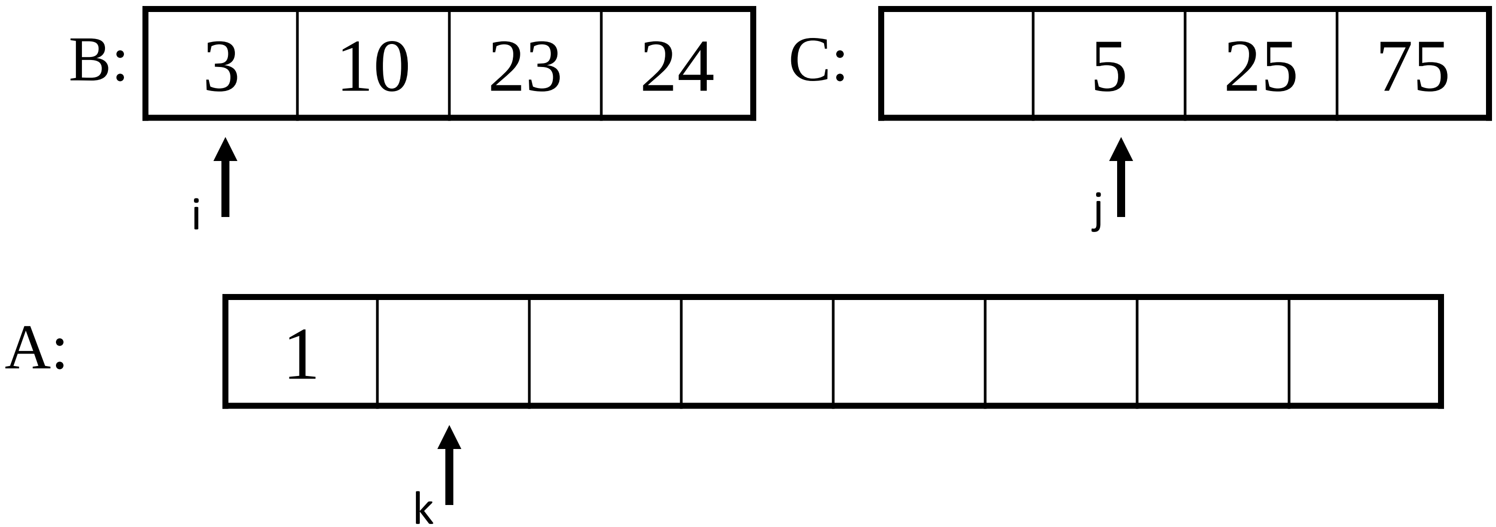
Mergesort

- The “combine partial solutions” part of mergesort is to merge two sorted arrays into one
- Example:
  - B = { 3 8 9 }   C = { 1 5 7 }
  - merge(B, C) = { 1 3 5 7 8 9 }

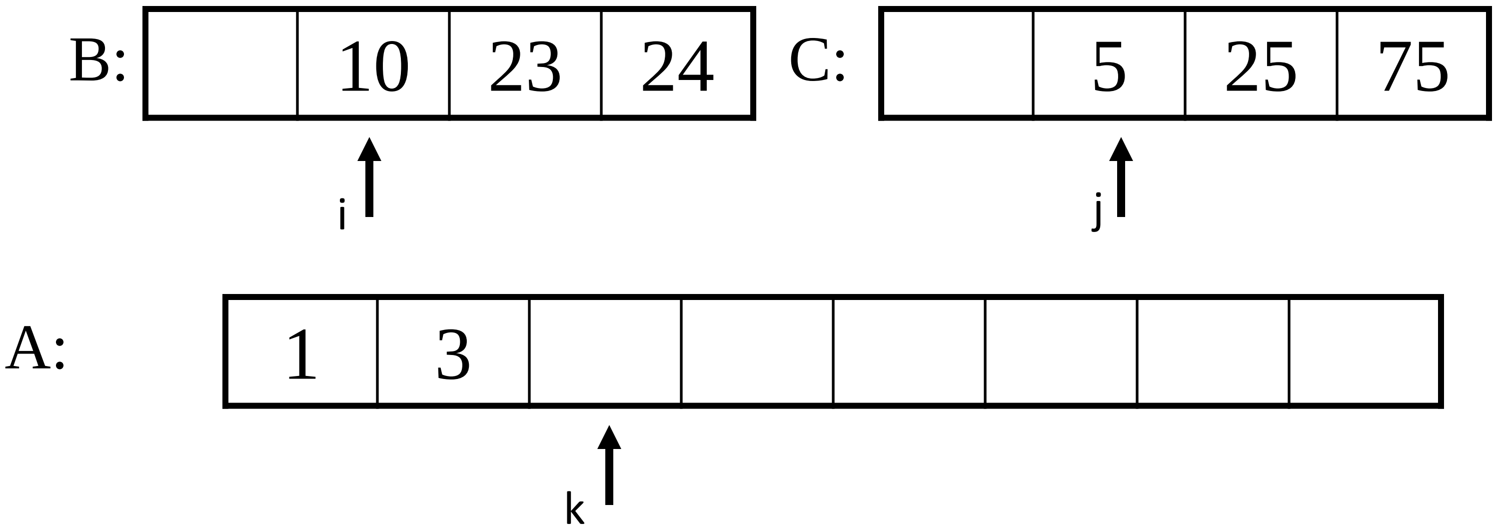
Merging



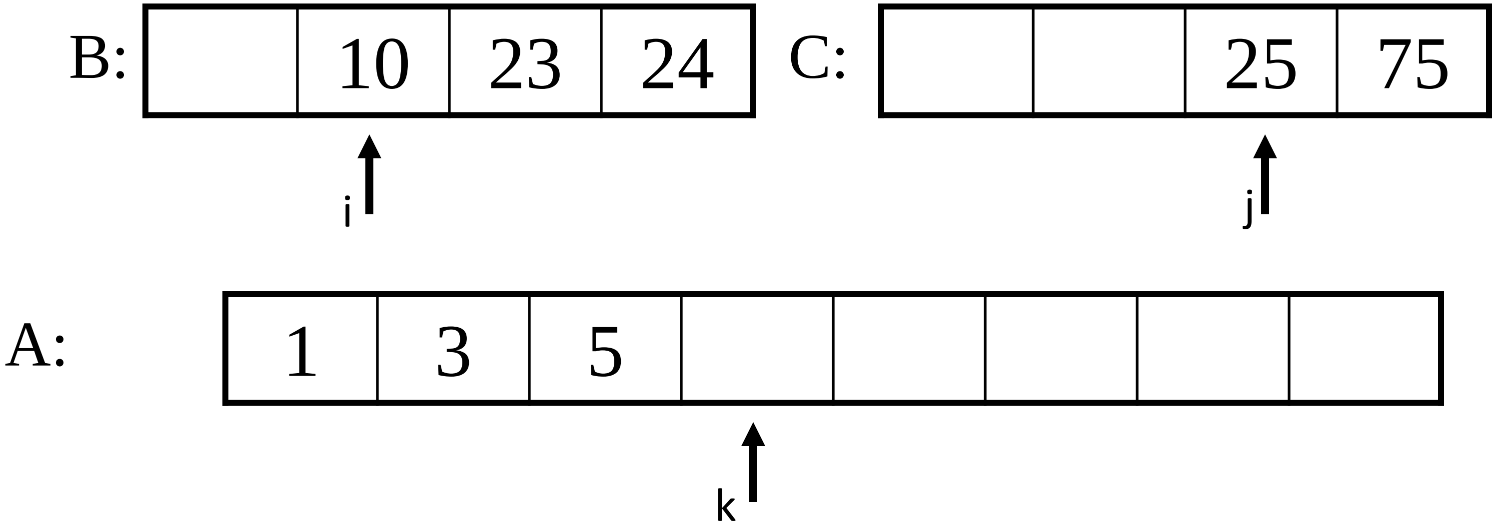
Merging (cont.)



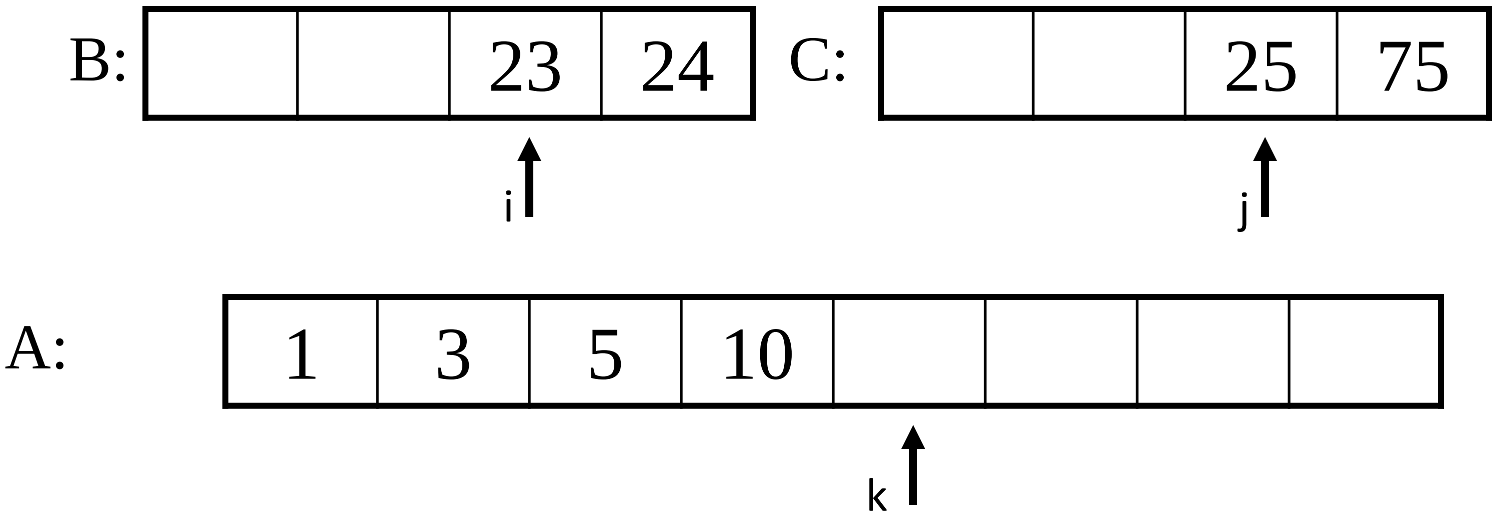
Merging (cont.)



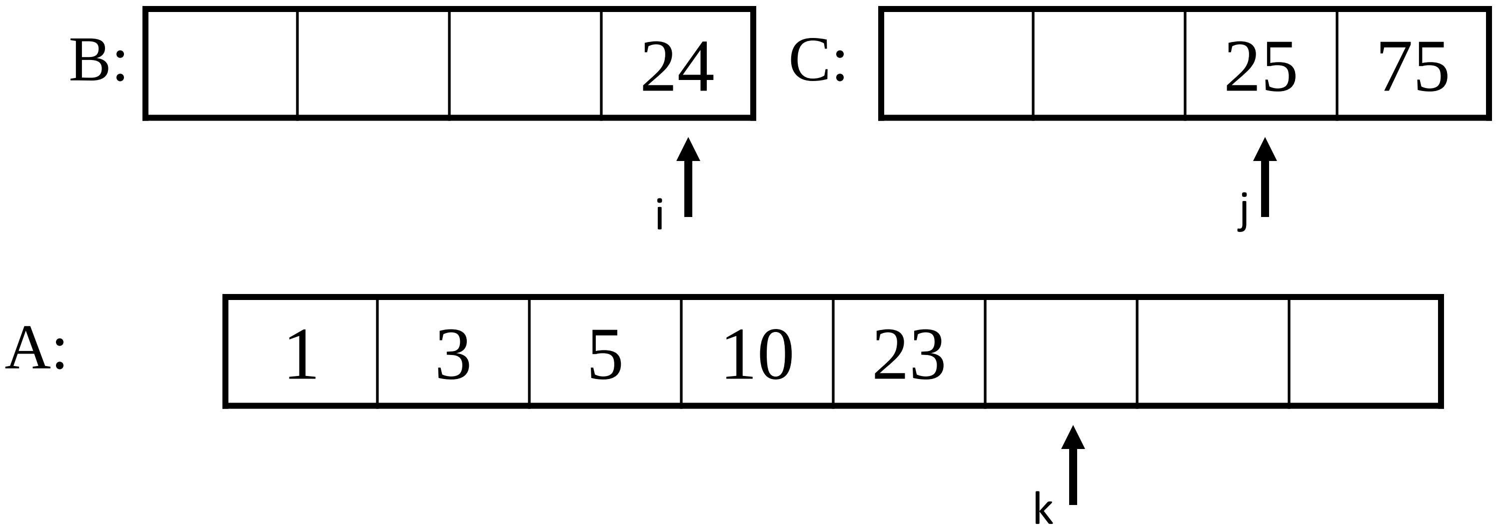
Merging (cont.)



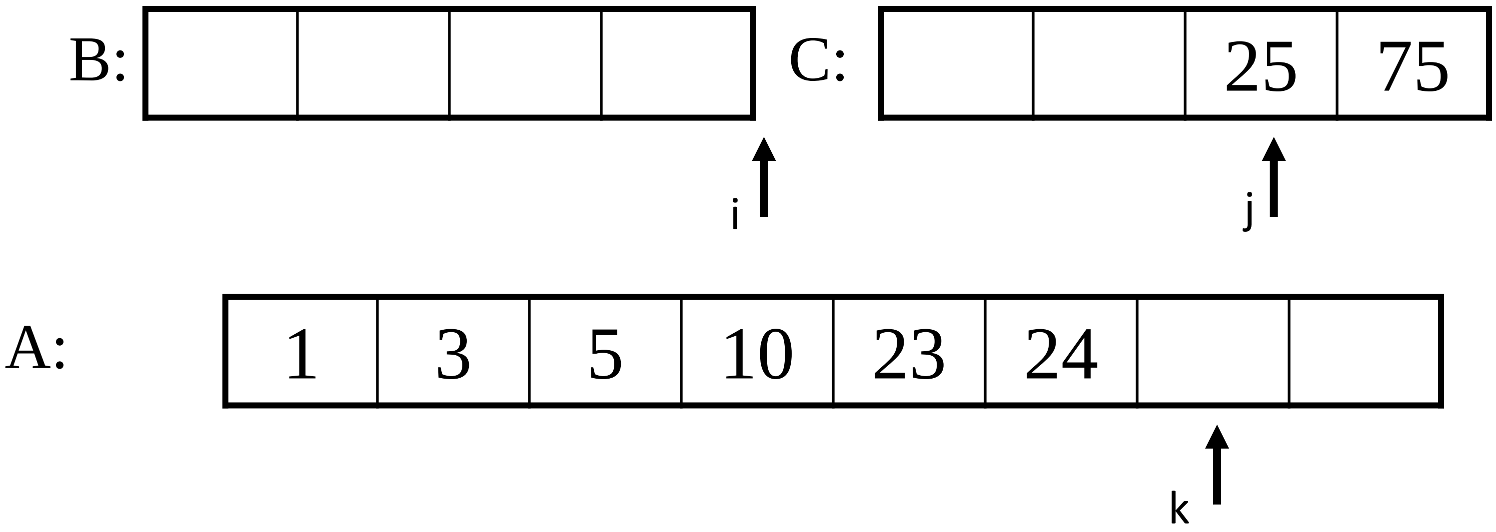
Merging (cont.)



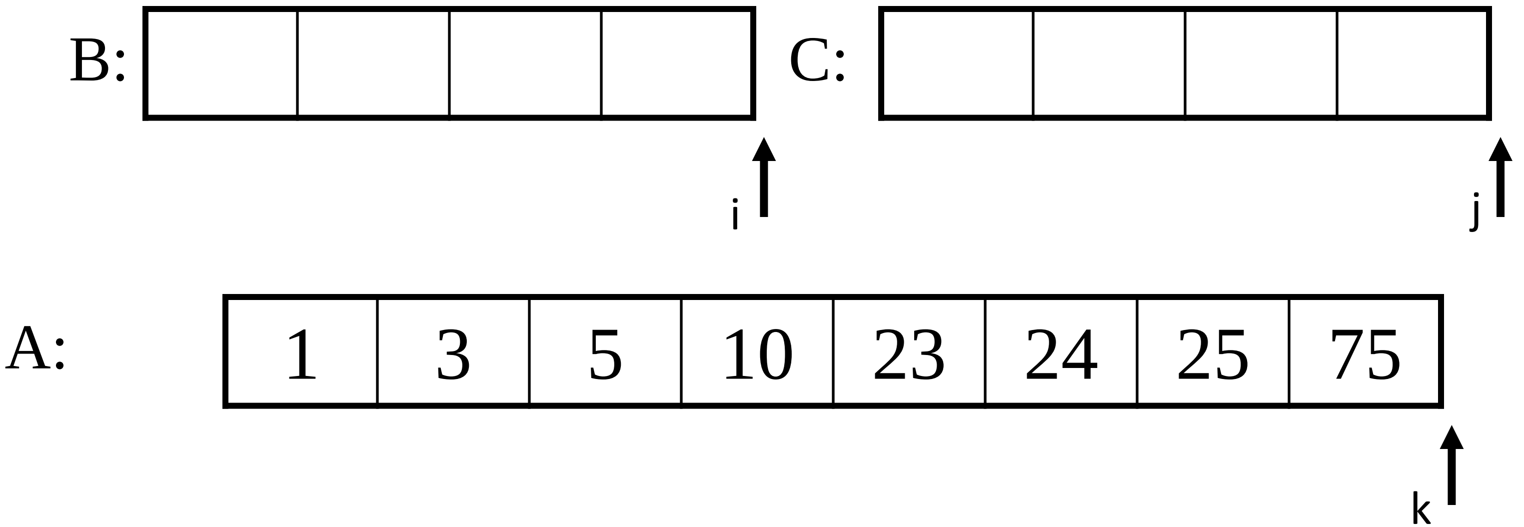
Merging (cont.)



Merging (cont.)



Merging (cont.)



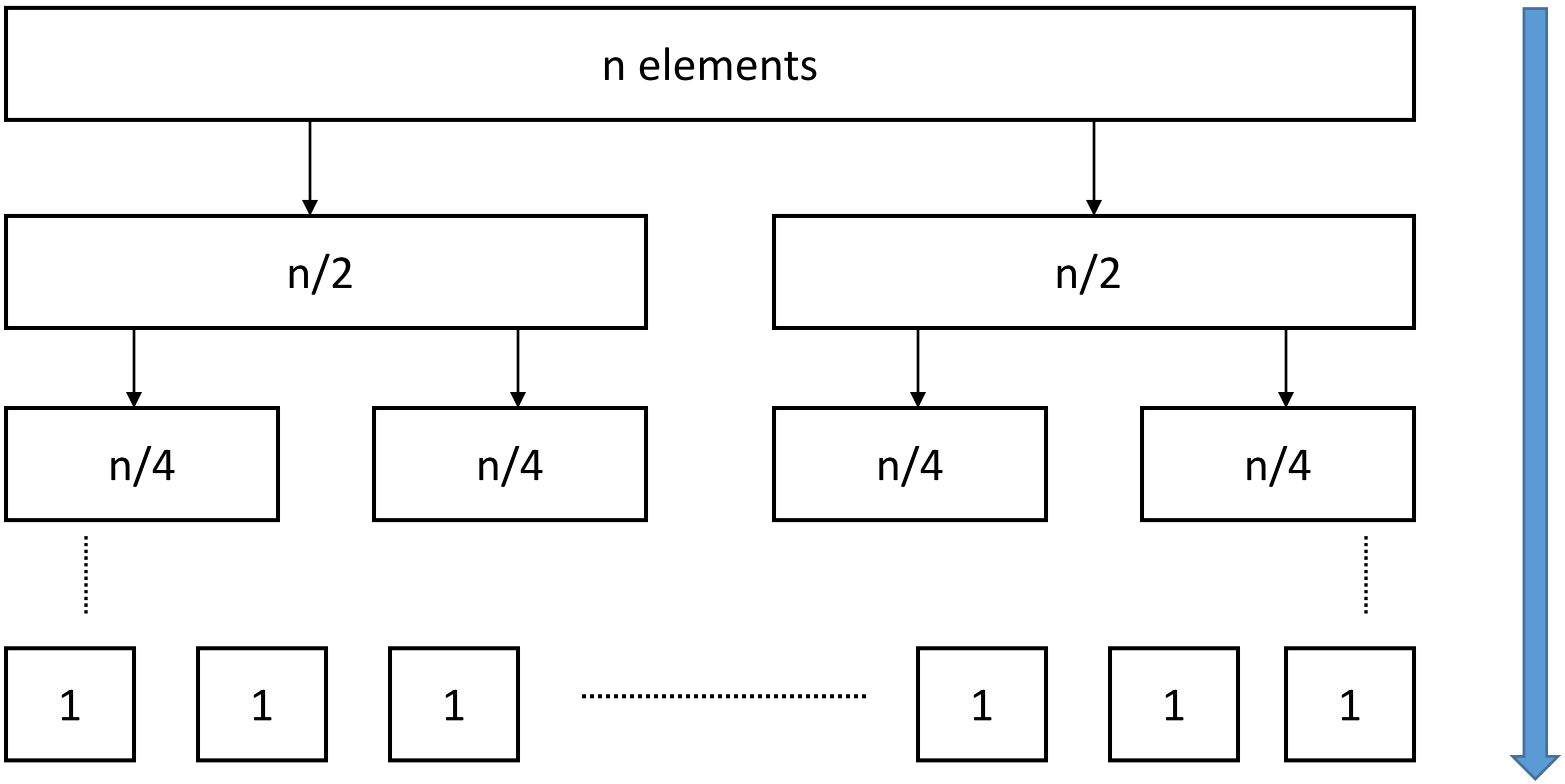
Pseudocode of Merge

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 $B[i] \le C[j]$
        $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
    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 $A[k..p+q-1]$

Mergesort



Mergesort Example

99	6	86	15	58	35	86	4	0

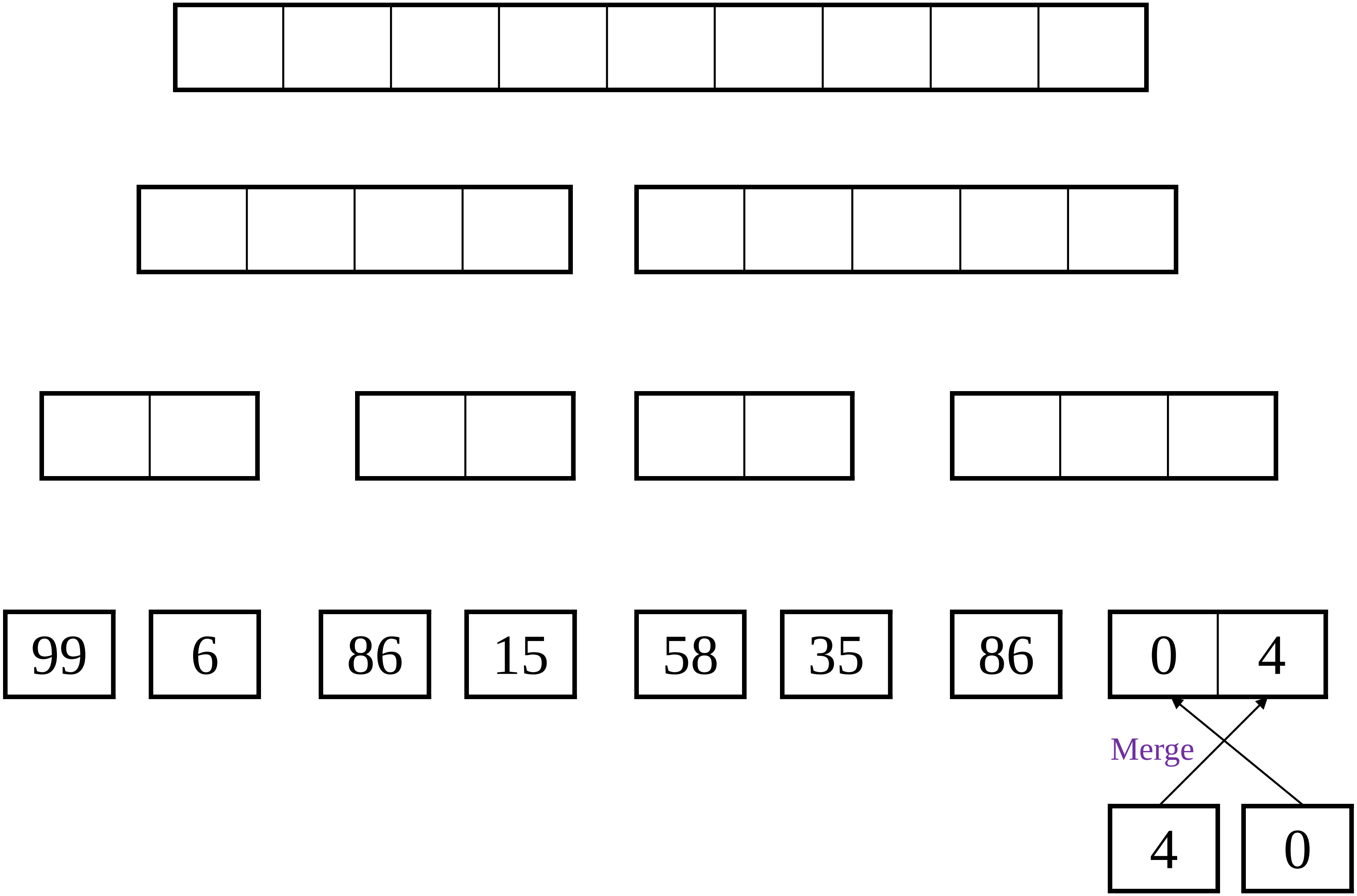
[99 | 6 | 86 | 15]   [58 | 35 | 86 | 4 | 0]

[99 | 6]   [86 | 15]   [58 | 35]   [86 | 4 | 0]

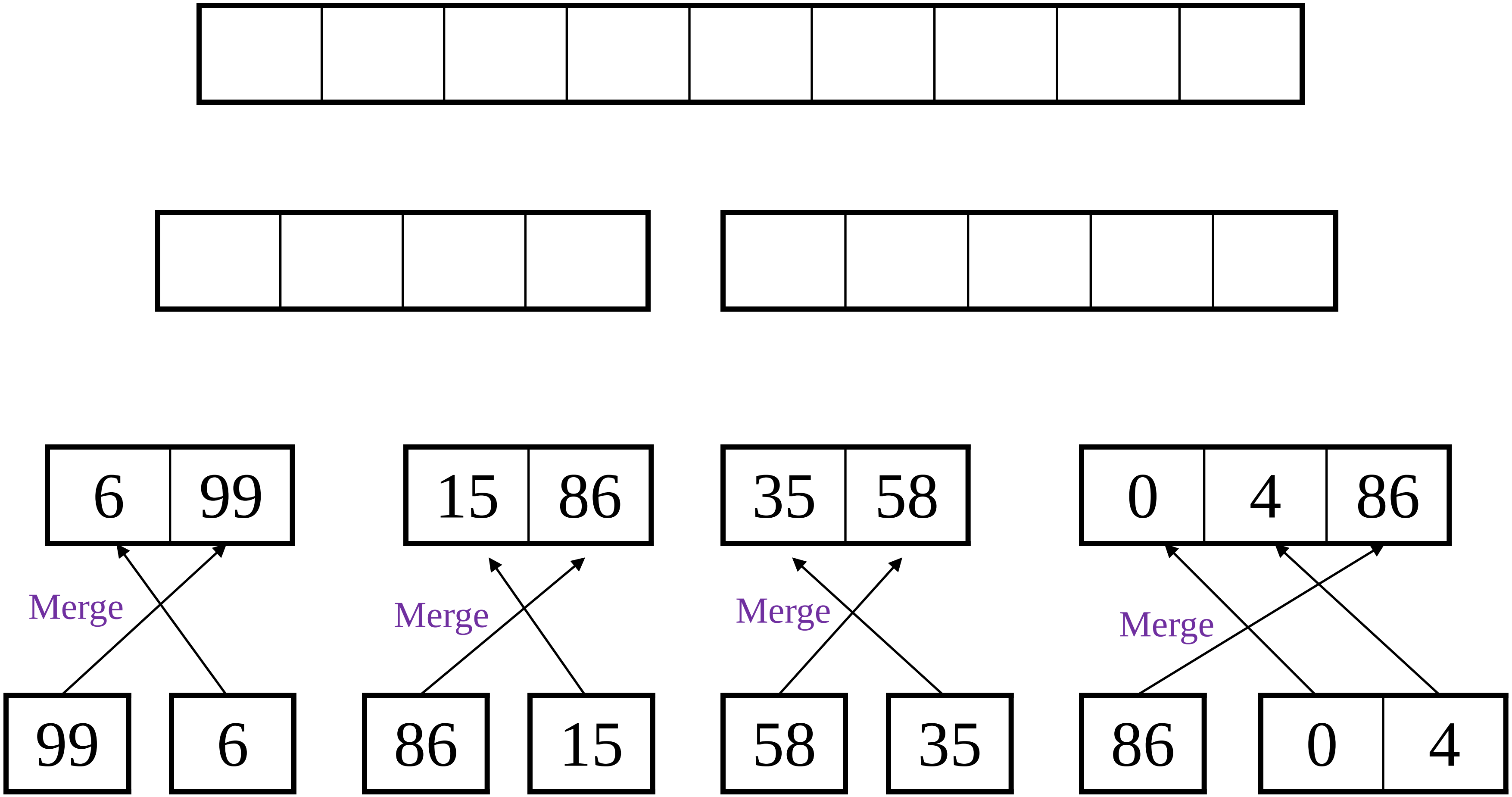
[99]   [6]   [86]   [15]   [58]   [35]   [86]   [4 | 0]

[4]   [0]

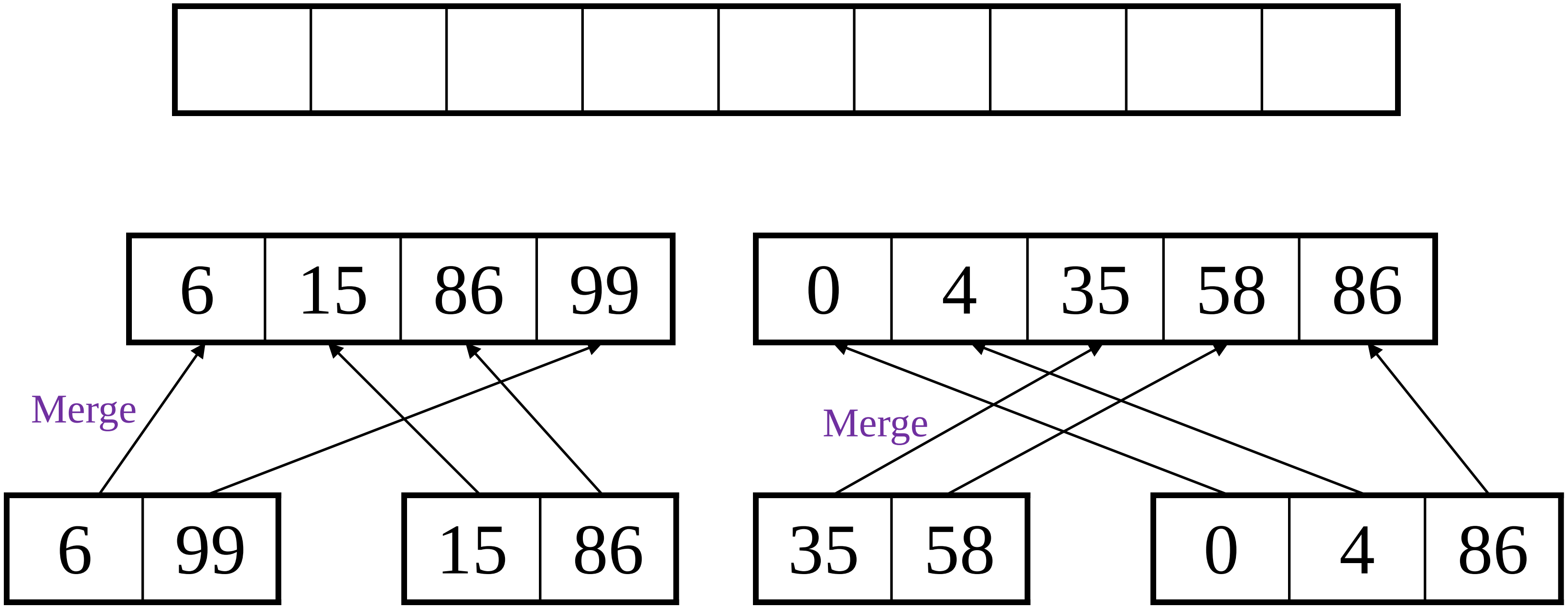
Mergesort Example



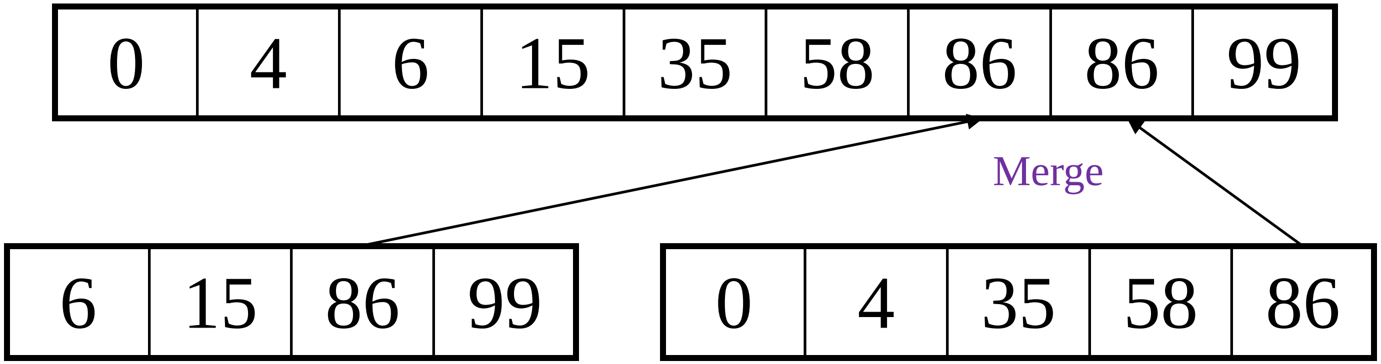
Mergesort Example



Mergesort Example



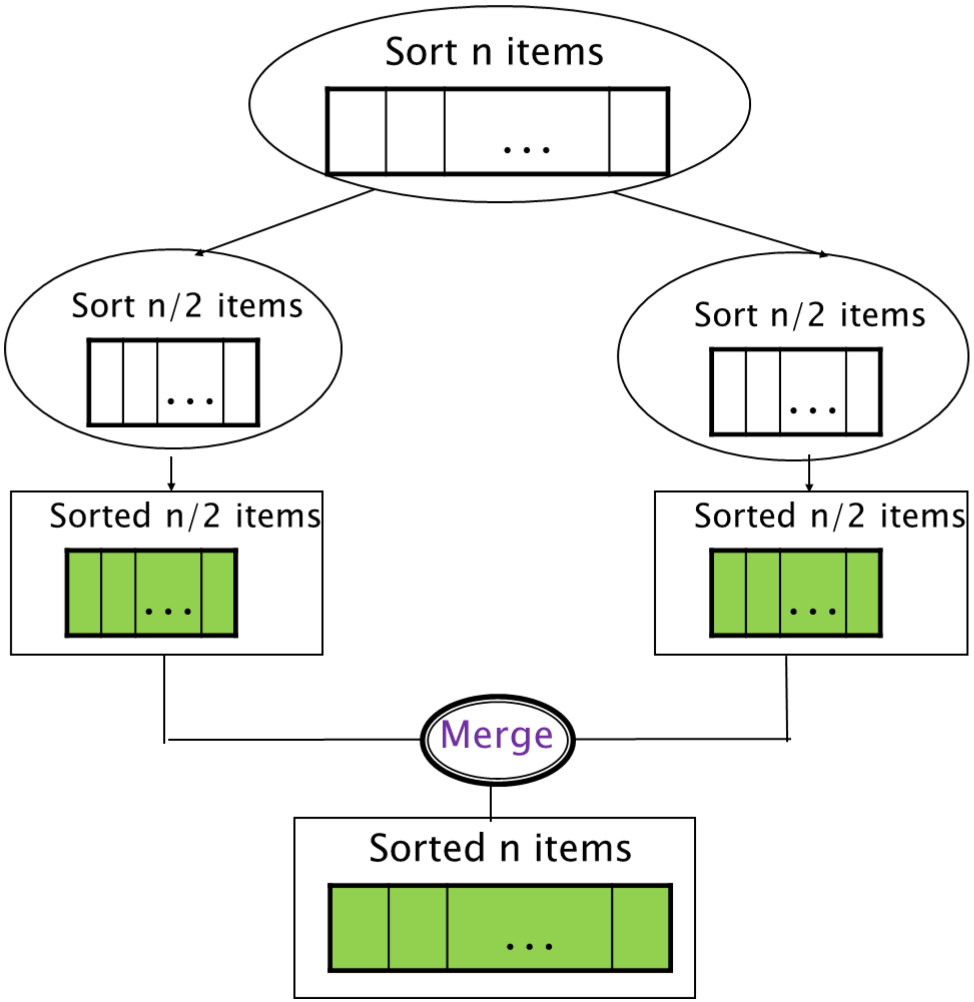
Mergesort Example



Mergesort Example

0	4	6	15	35	58	86	86	99

Mergesort running time



$T(n) = 2\ T(n/2) + n$

$n^{\log_b a} = n^{\log_2 2} = n$

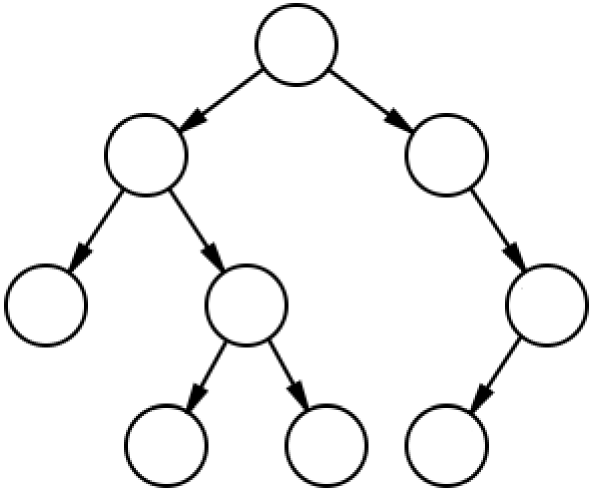
$T(n) \in O(n \log n)$

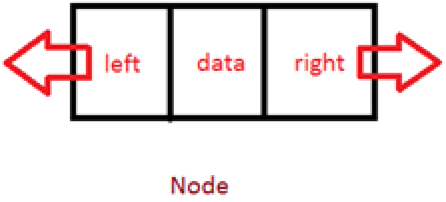
Binary trees

Binary tree structure

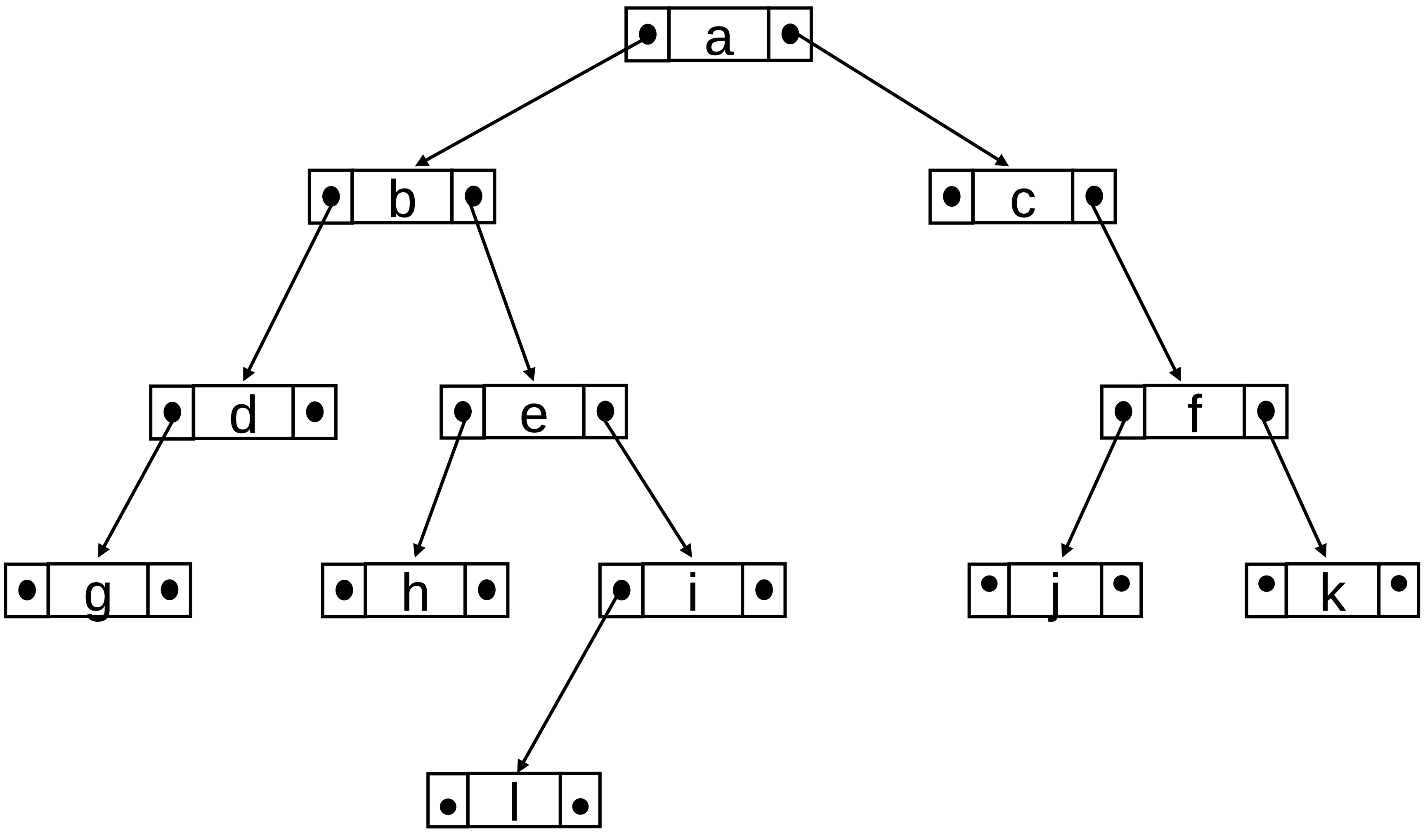
```
public class Node {
        public char data;
        public Node left;
        public Node right;

        public Node(char d) {
                data = d;
        }
}
```

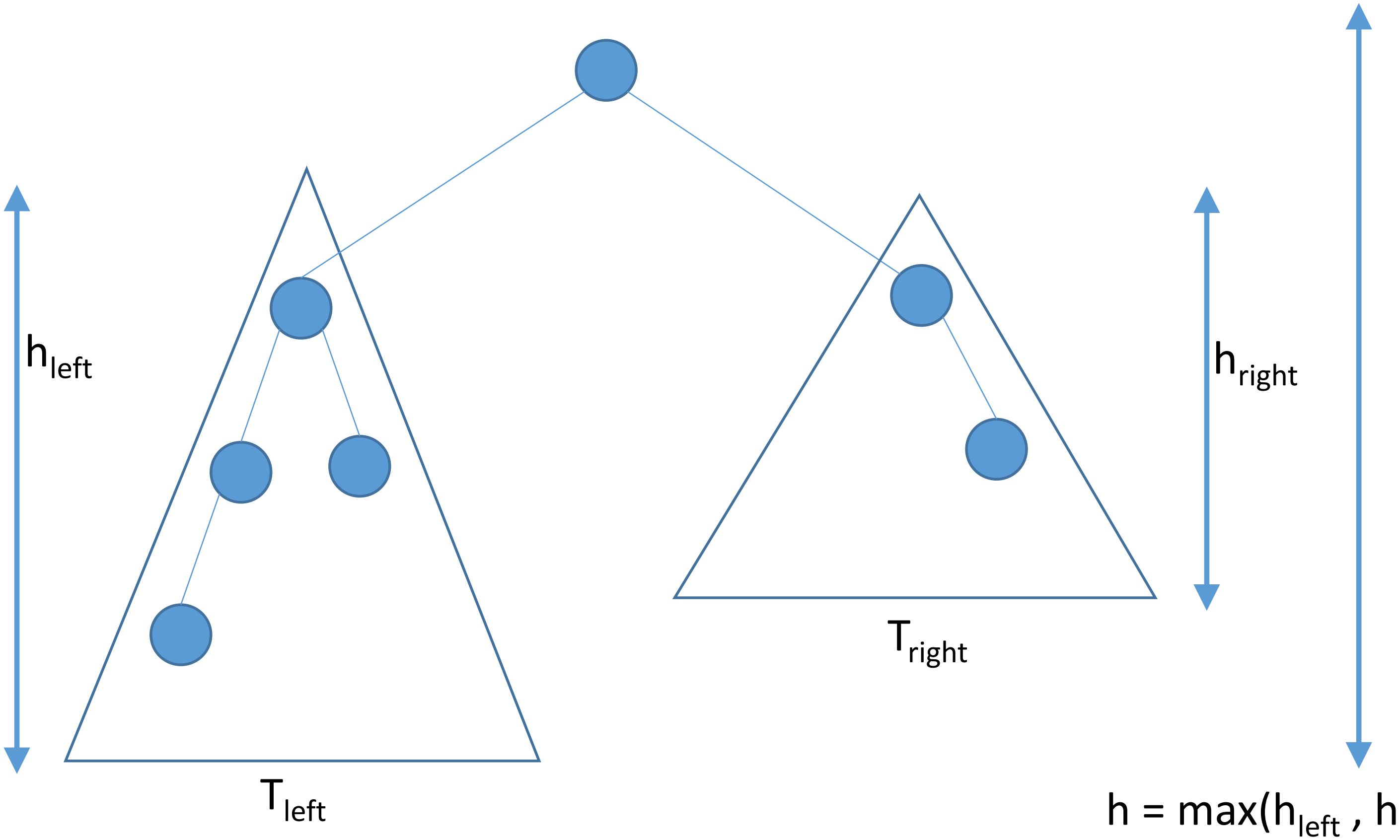




Binary tree implementation



Computing the height of a binary tree



$h = \max(h_{left}, h_{right}) + 1$

Computing the height of a binary tree

ALGORITHM $Height(T)$

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

Compute the number of leaves

Algorithm $LeafCounter(T)$
//Computes recursively the number of leaves in a binary tree
//Input: A binary tree T
//Output: The number of leaves in T
if $T = \varnothing$ **return** 0 //empty tree
else if $T_L = \varnothing$ **and** $T_R = \varnothing$ **return** 1 //one-node tree
else return $LeafCounter(T_L) + LeafCounter(T_R)$ //general case

Practice problems

1. Chapter 5.1, page 174, questions 1, 2, 6
2. Chapter 5.3, page 185, question 2
3. Implement a function to check if a tree is balanced. A balanced tree is defined to be a tree such that no two leaf nodes differ in distance from the root by more than one.