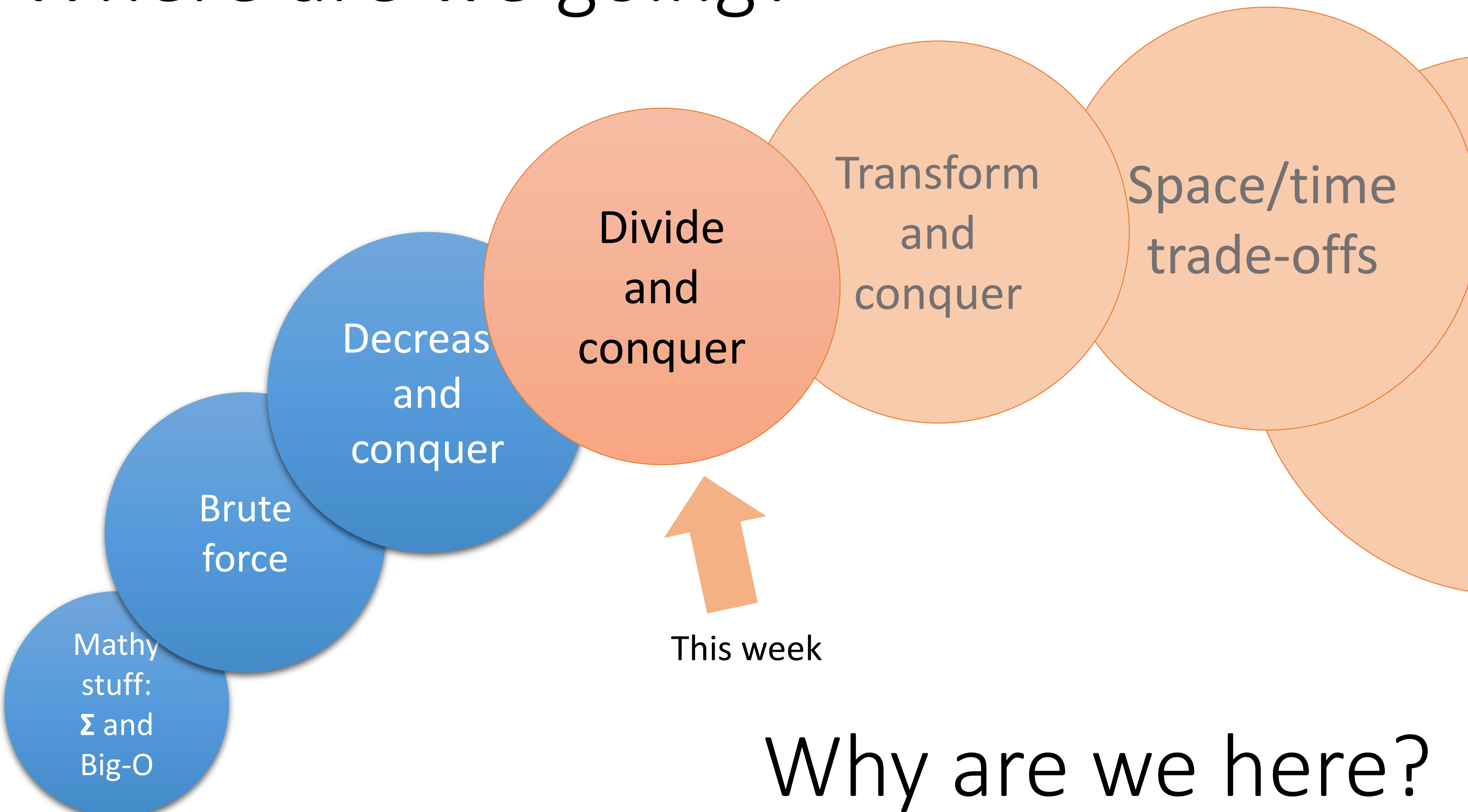
# Where have we been?
# Where are we going?

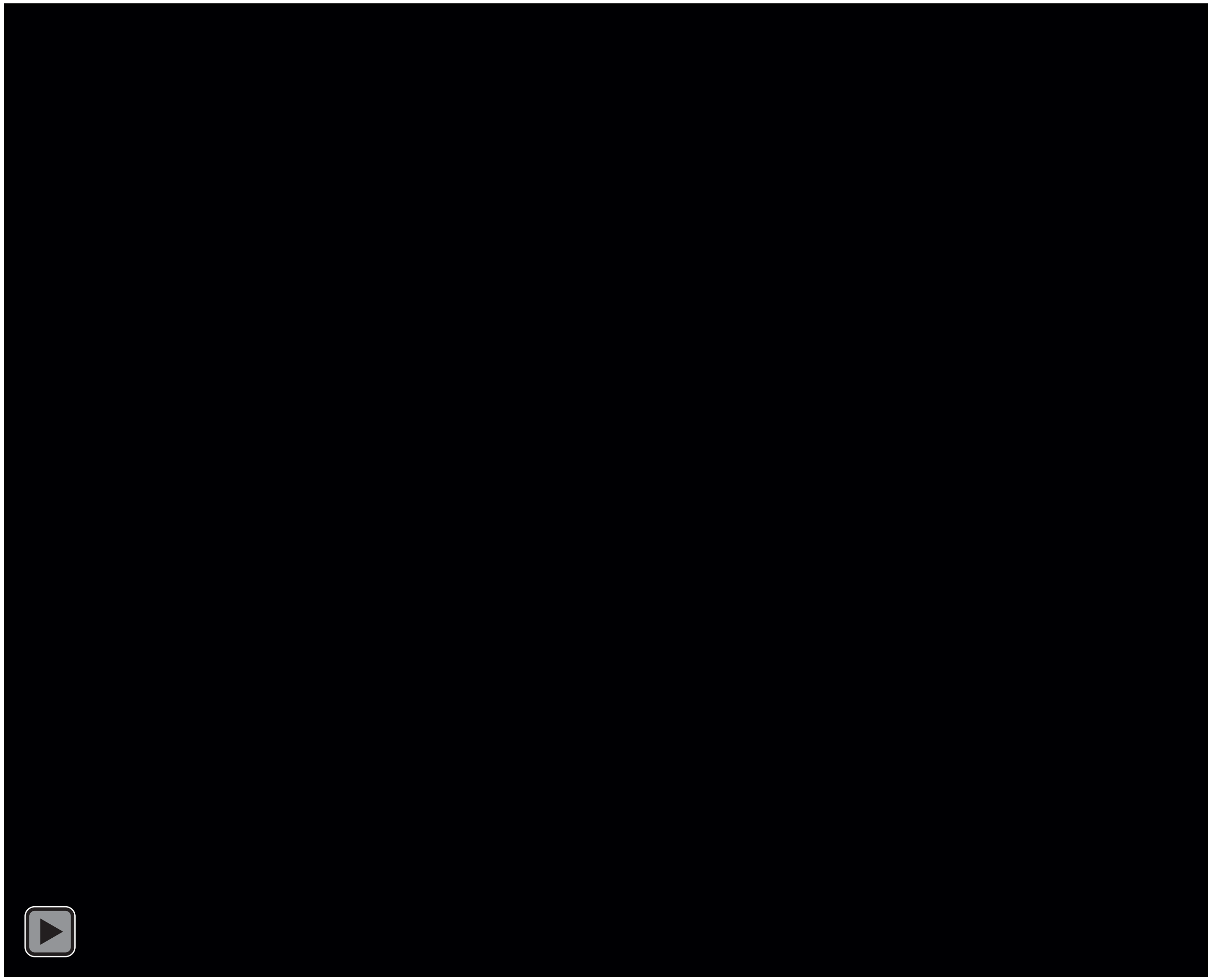



## Why are we here?
No. Too deep.

# 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

# First a bit about recursion



# Our old friend Factorial()

```
Factorial(n)
 if n=0 or n=1 then
   return 1
 else
   return n * Factorial(n - 1)
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return
     4 *
```

```
Factorial(3)
 if n=0 or n=1 then
   return 1
 else
   return
     3 *
```

```
Factorial(2)
 if n=0 or n=1 then
   return 1
 else
   return
     2 *
```

```
Factorial(1)
 if n=0 or n=1 then
   return 1
 else
   return 1 * Factorial(0)
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return
     4 *
```

```
Factorial(3)
 if n=0 or n=1 then
   return 1
 else
   return
     3 *
```

```
Factorial(2)
 if n=0 or n=1 then
   return 1
 else
   return
     2 *
```

```
Factorial(1)
 if n=0 or n=1 then
   return 1
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return
     4 *
```

```
Factorial(3)
 if n=0 or n=1 then
   return 1
 else
   return
     3 *
```

```
Factorial(2)
 if n=0 or n=1 then
   return 1
 else
   return
     2 * 1
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return
     4 *
```

```
Factorial(3)
 if n=0 or n=1 then
   return 1
 else
   return
     3 *
```

```
Factorial(2)
 if n=0 or n=1 then
   return 1
 else
   return 2
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return
     4 *
```

```
Factorial(3)
 if n=0 or n=1 then
   return 1
 else
   return
     3 * 2
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return
     4 *
```

```
Factorial(3)
 if n=0 or n=1 then
   return 1
 else
   return 6
```

```
Factorial(5)
  if n=0 or n=1 then
    return 1
  else
    return 5 *
```

```
Factorial(4)
  if n=0 or n=1 then
    return 1
  else
    return
      4 * 6
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 *
```

```
Factorial(4)
 if n=0 or n=1 then
   return 1
 else
   return 24
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 5 * 24
```

```
Factorial(5)
 if n=0 or n=1 then
   return 1
 else
   return 120
```

```
Factorial(5) ==> 120
```

# Example: permutations

```
generatePerms (a1, a2, ..., an)
   if n > 1
      smallerPerms = generatePerms (a1, a2, ..., an-1)
      initialize allPerms to {}
      for each p in smallerPerms
         insert an before a1 and add to allPerms
         for i = 1 to n-1
            insert an after ai and add to allPerms
      return allPerms
```

# Permutations algorithm

```
generatePerms (a1, a2, ..., an)
   if n > 1
      smallerPerms =
             generatePerms (...)

                /* it does whatever it does */


      initialize allPerms to {}
      for each p in smallerPerms
         insert an before a1 and add to allPerms
         for i = 1 to n-1
            insert an after ai and add to allPerms
      return allPerms
```

# Permutations algorithm

```
generatePerms (a1, a2, ..., an)
   if n > 1
      smallerPerms =

              {perm, perm, perm, perm, ..., perm}




      initialize allPerms to {}
      for each p in smallerPerms
         insert an before a1 and add to allPerms
         for i = 1 to n-1
            insert an after ai and add to allPerms
      return allPerms
```

# Lecture 4

COMP 3760

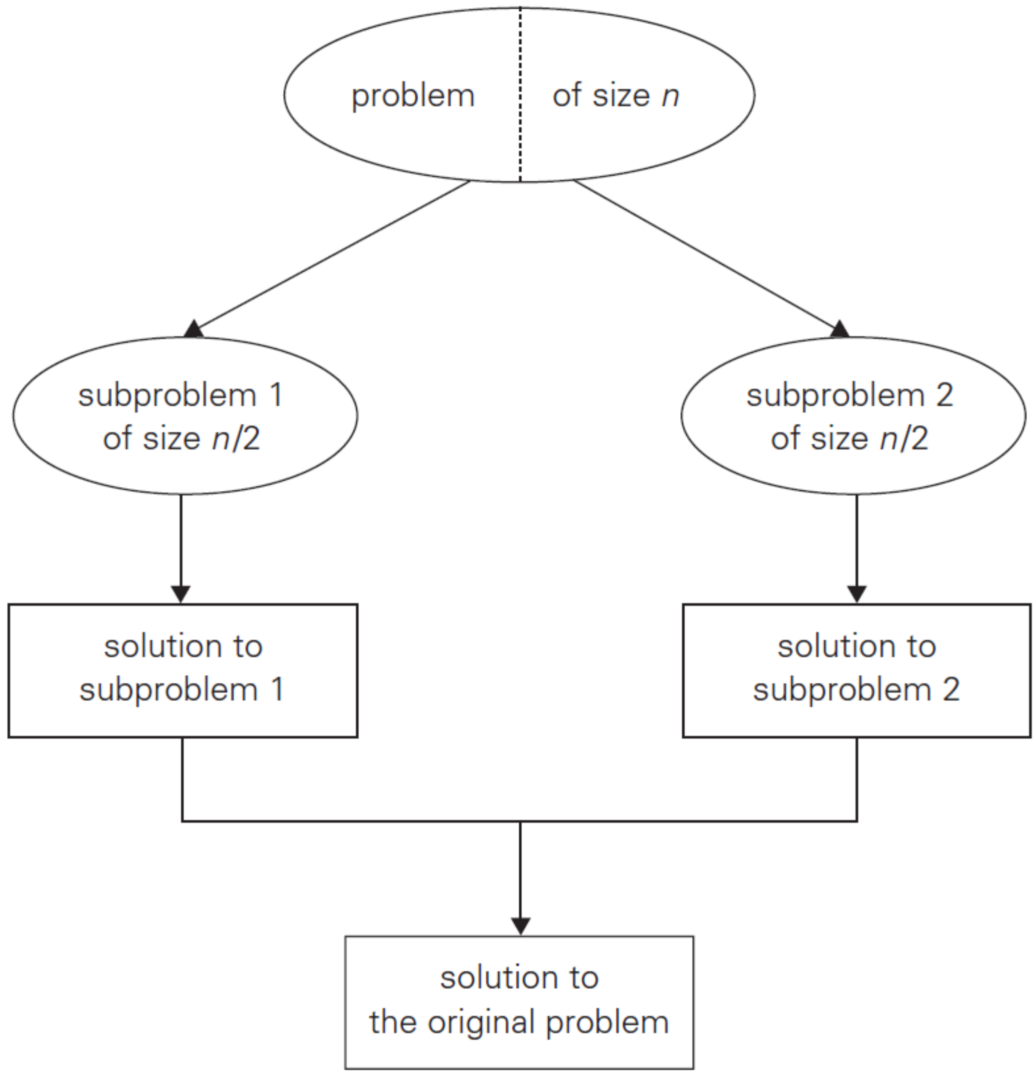
Divide and Conquer algorithms

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 only worked on one “subproblem”

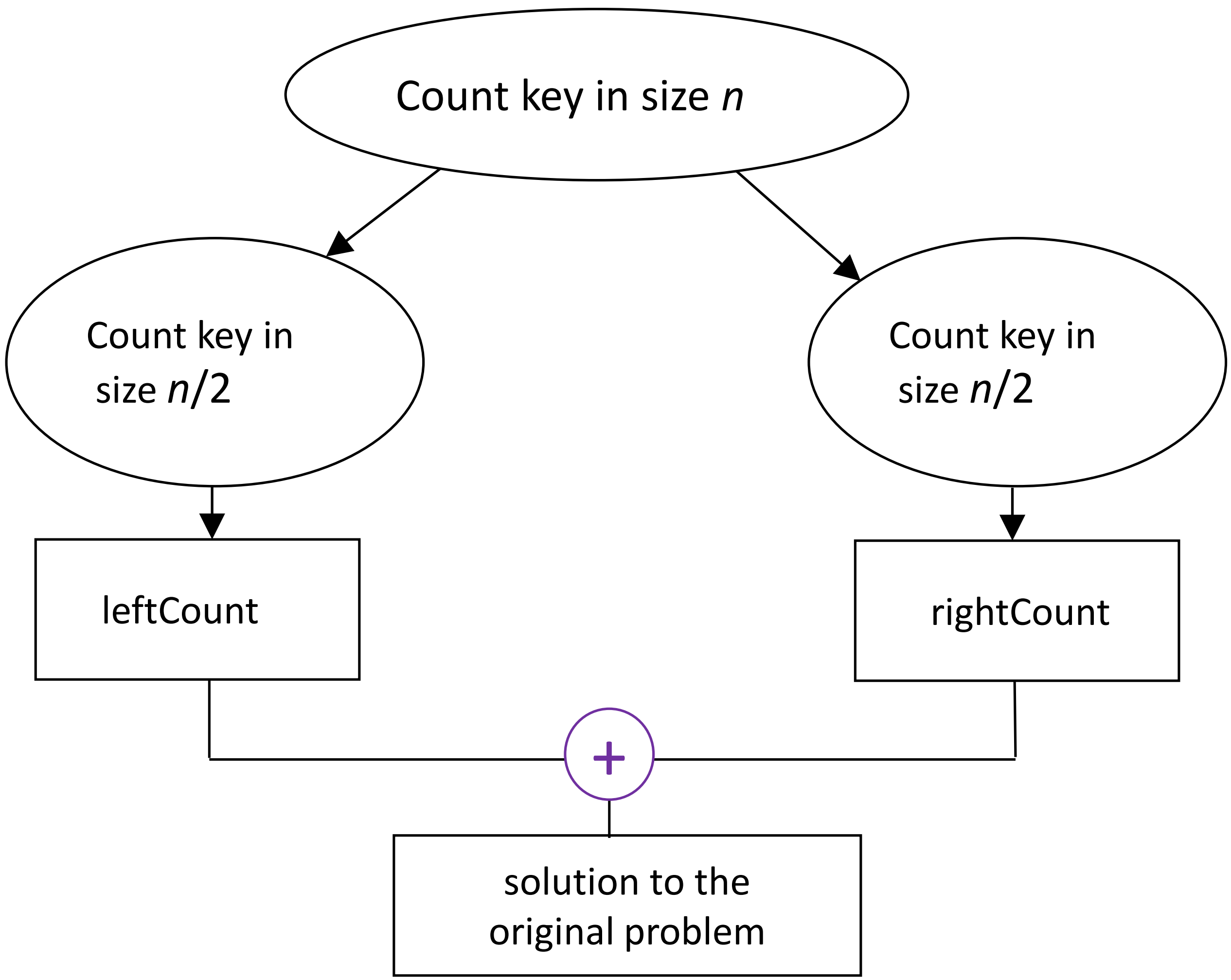
- For divide and conquer…
  - You need to solve all of the subproblems

# 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 that uses divide and conquer

# Count a key in an array



# Count a key in an array

```
Algorithm CountKeys(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

- Superficially, CountKeys resembles Binary Search
  - Similar arguments (array bounds)
  - Finding a midpoint
  - What’s the difference?

- We have to process both sides
  - In CountKeys, 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:

$$n^{\log_b a}$$

# Count a key in an array

```
Algorithm CountKeys(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
```

2 subproblems
*i.e.,* 2 recursive calls
a = 2

# Count a key in an array

```
Algorithm CountKeys(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
```

each subproblem is
half the size (n/2)
b = 2

# Count a key in an array

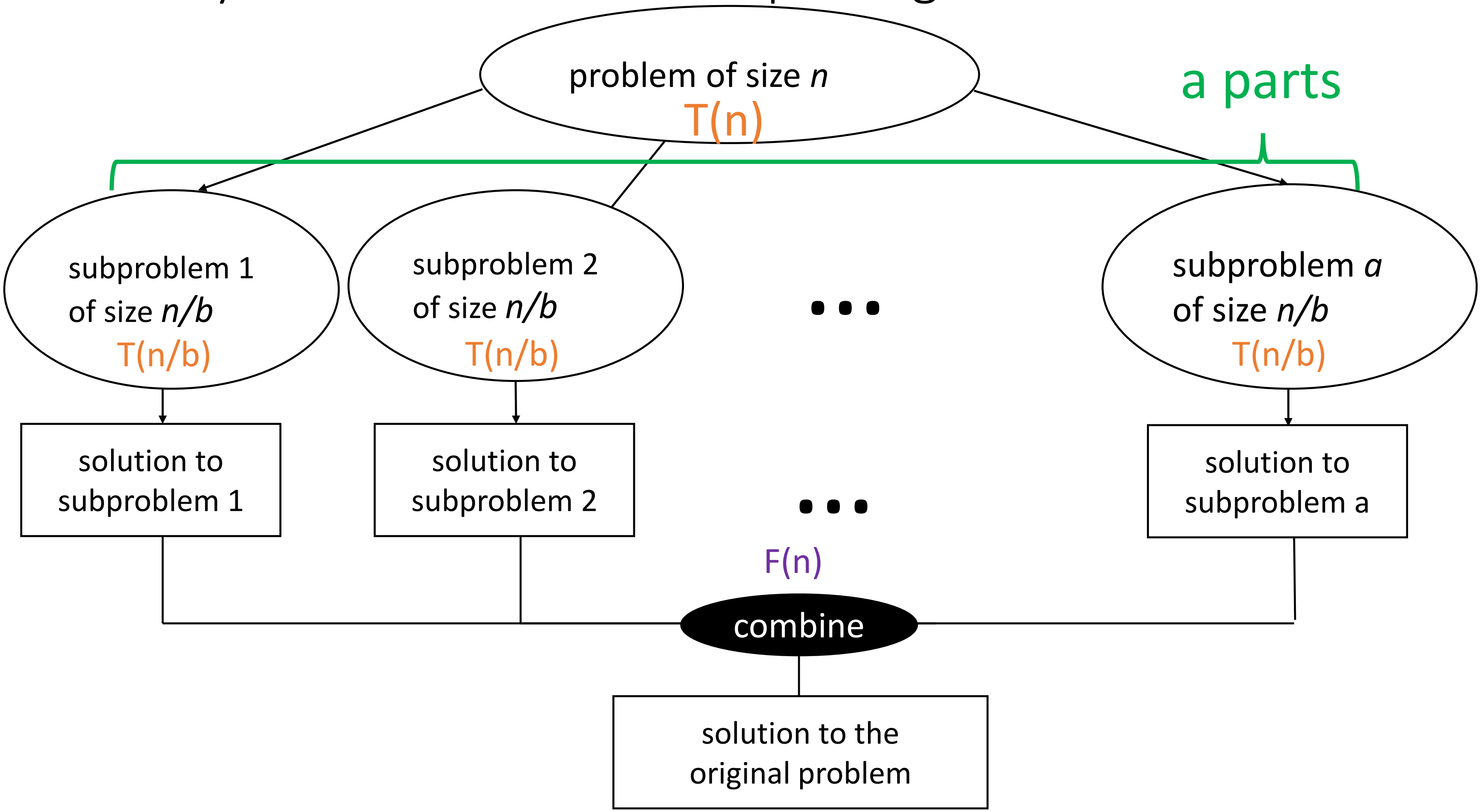
```
Algorithm CountKeys(A[], Key, L, R)

//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
```

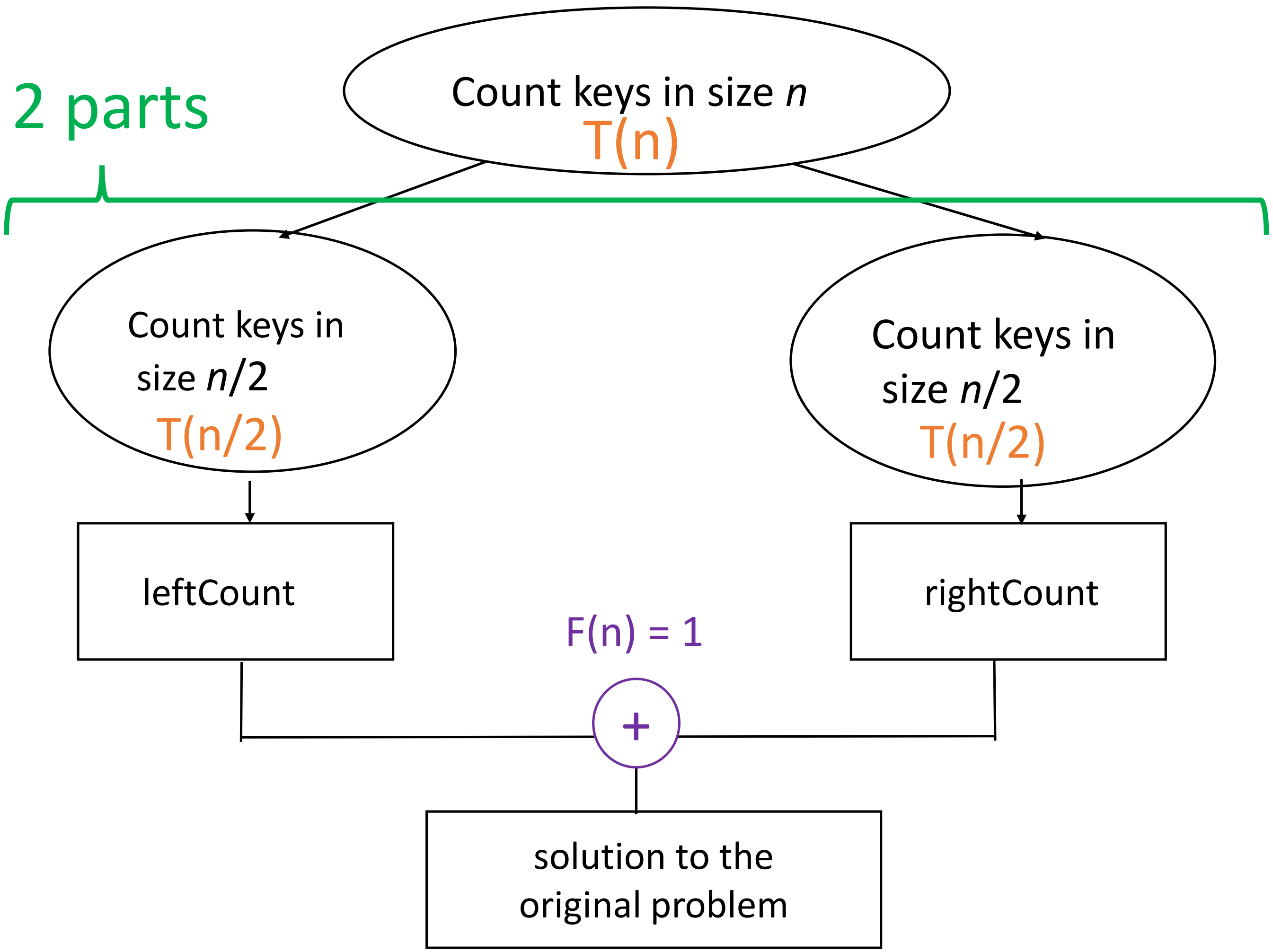
additional computation
time is O(1)
F(n) = 1

# Analysis of a divide and conquer algorithm




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

# Example: analysis of CountKeys




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

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 $n^{\log_b a} = F(n)$, $\quad T(n) \in O(n^{\log_b a} \log n)$

# Another version

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

# Master Theorem examples

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

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:** $\boldsymbol{T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?}$

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:** $\boldsymbol{T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?}$

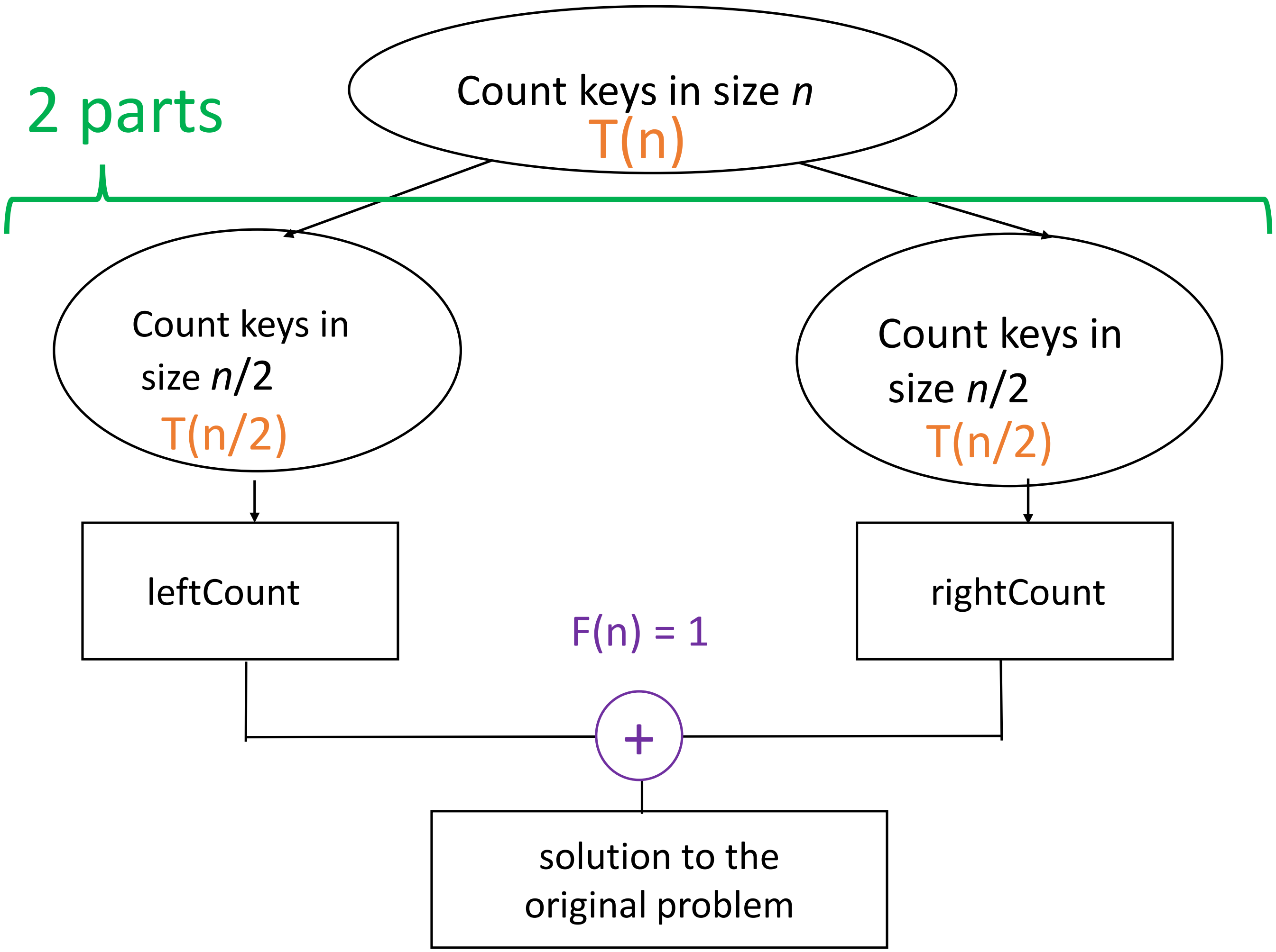
a = 4
b = 2
F(n) = $n^3$

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

F(n) = $n^3$

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

# Back to CountKeys again



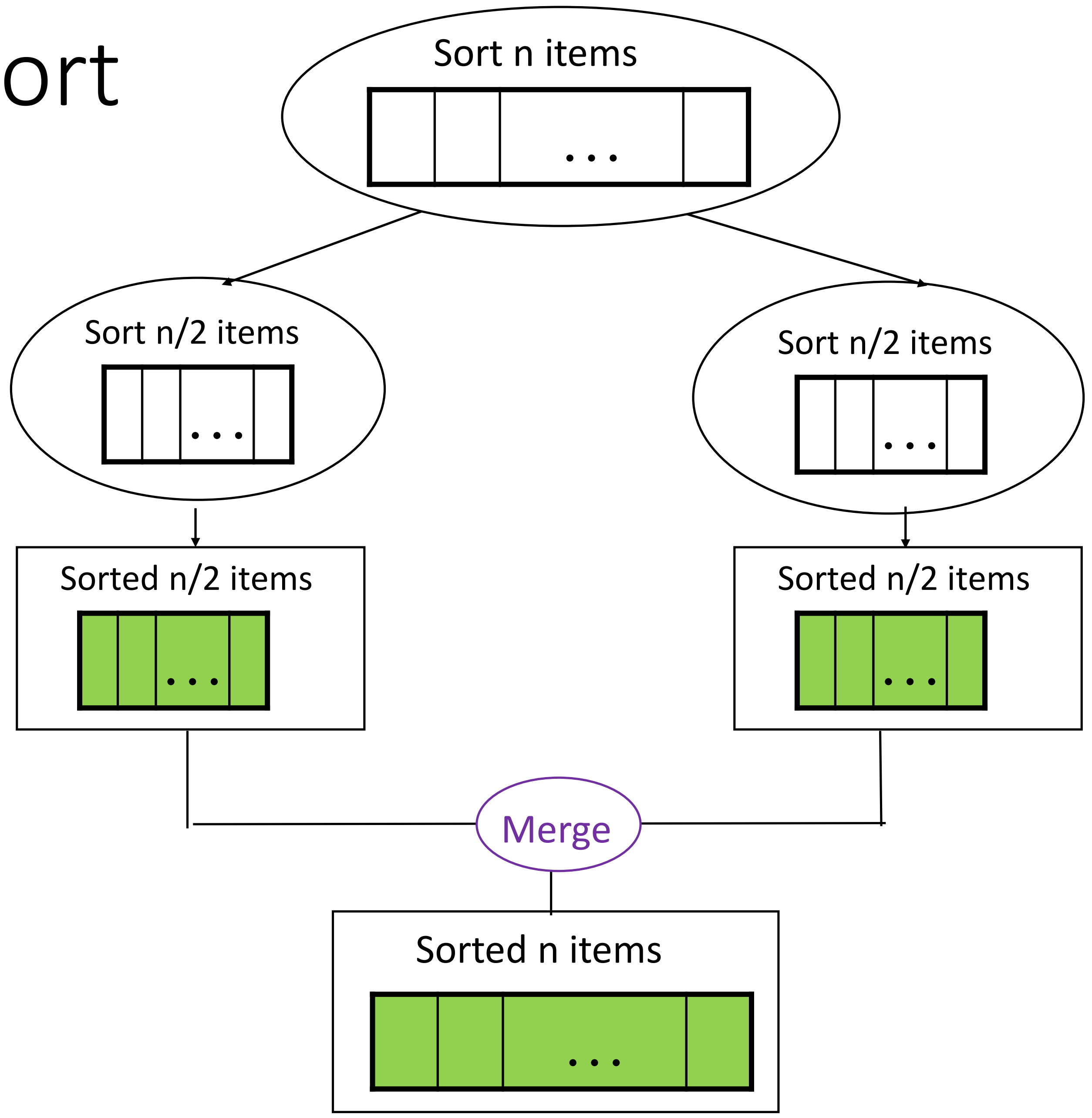

$$T(n) = 2\ T(n/2) + 1 \quad \Rightarrow \quad T(n) \in O(n)$$

# Analyzing a *decrease and conquer* algorithm

- Can use the same Master Theorem
- Fake coin problem:
  - a=1 (we only solve ONE sub-part)
  - b=2 (each part is n/2)
  - F(n)=1 (no combination step = constant time)
- → Running time is T(n) = 1T(n/2) + 1
  - $n^{\log_b a} == n^{\log_2 1} == n^0 == 1$
  - F(n) = 1
  - They are equal
- → Final answer is $O(n^{\log_b a} \log n) == O(\log n)$

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

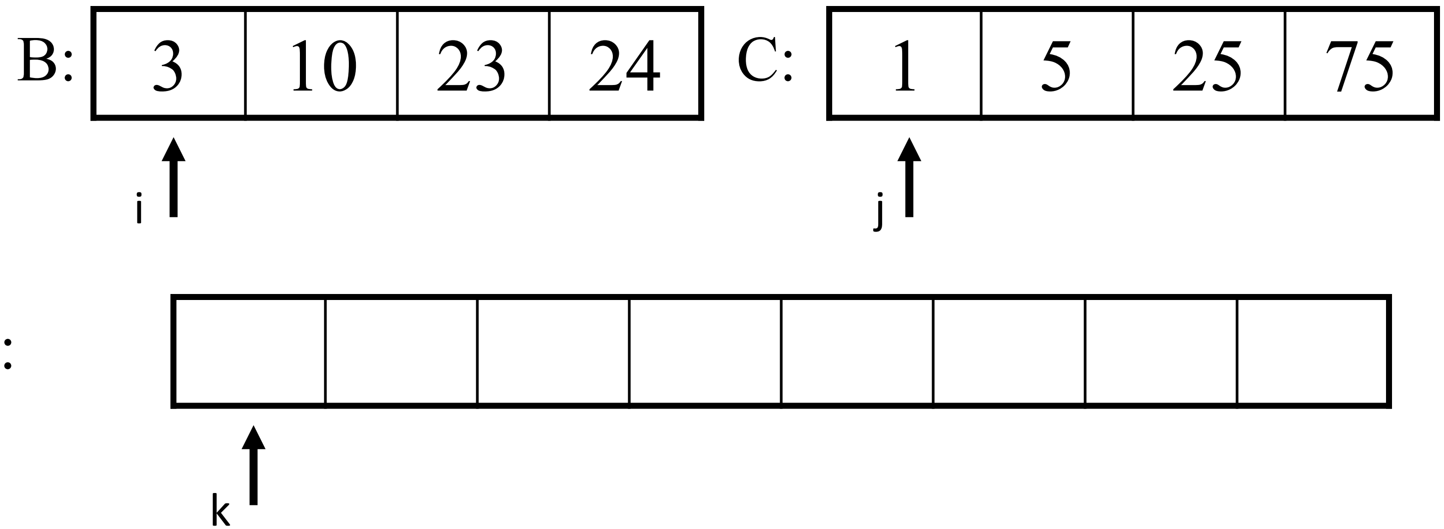
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
**if** $n > 1$
  copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
  copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil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

# Mergesort

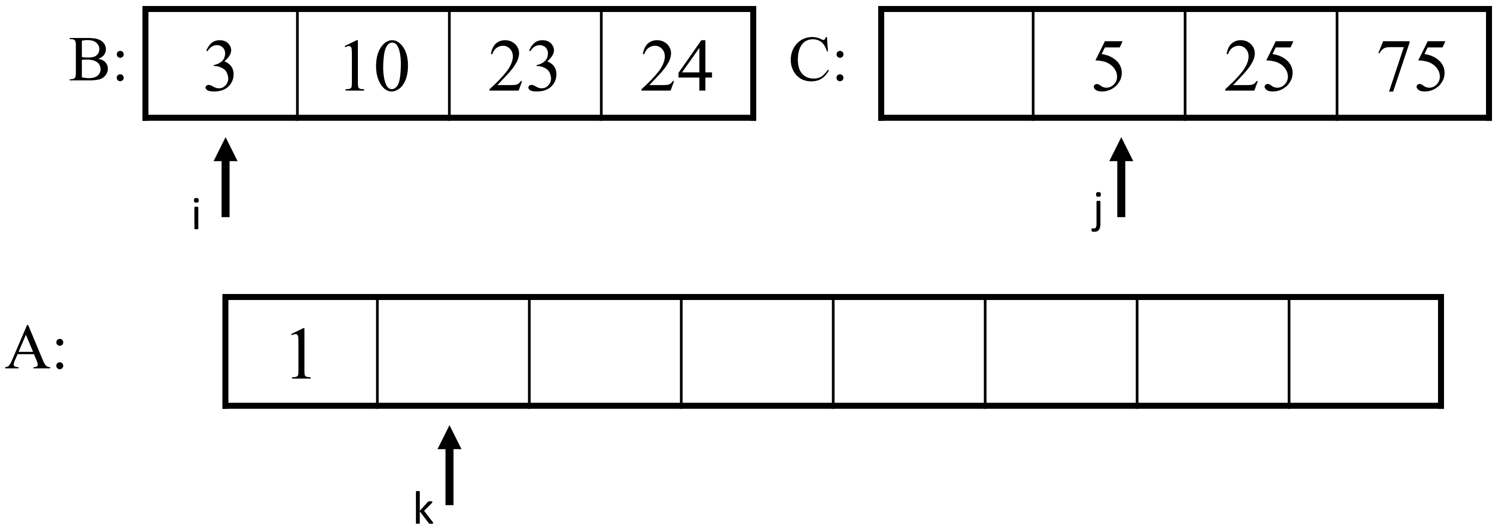
- The “combine partial solutions” part of mergesort is *merging two sorted arrays into one*

- Example:
  - B = { 3 8 9 }   C = { 1 5 7 }
  - merge(B, C) = { 1 3 5 7 8 9 }

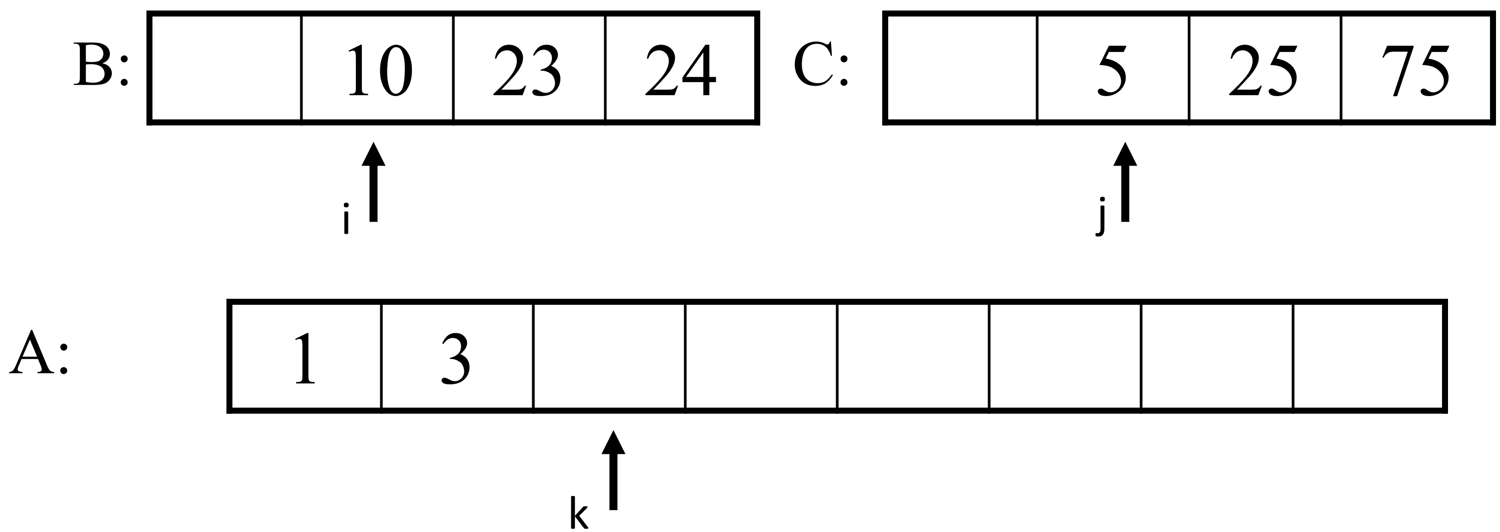
# Merging



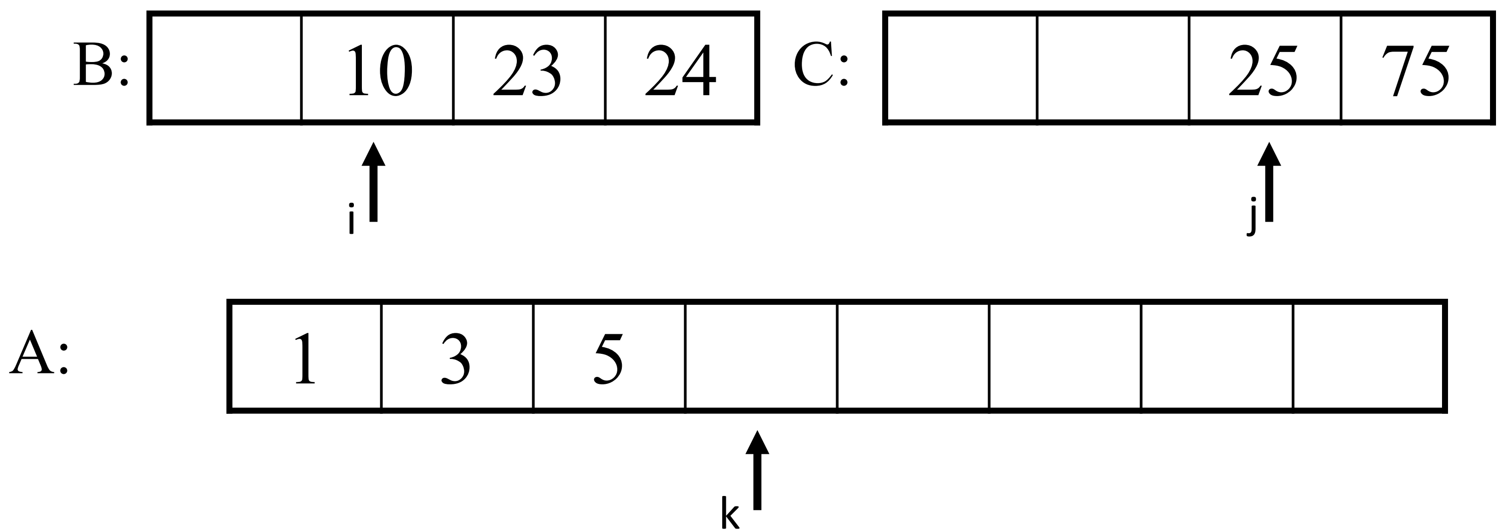
# Merging (cont.)



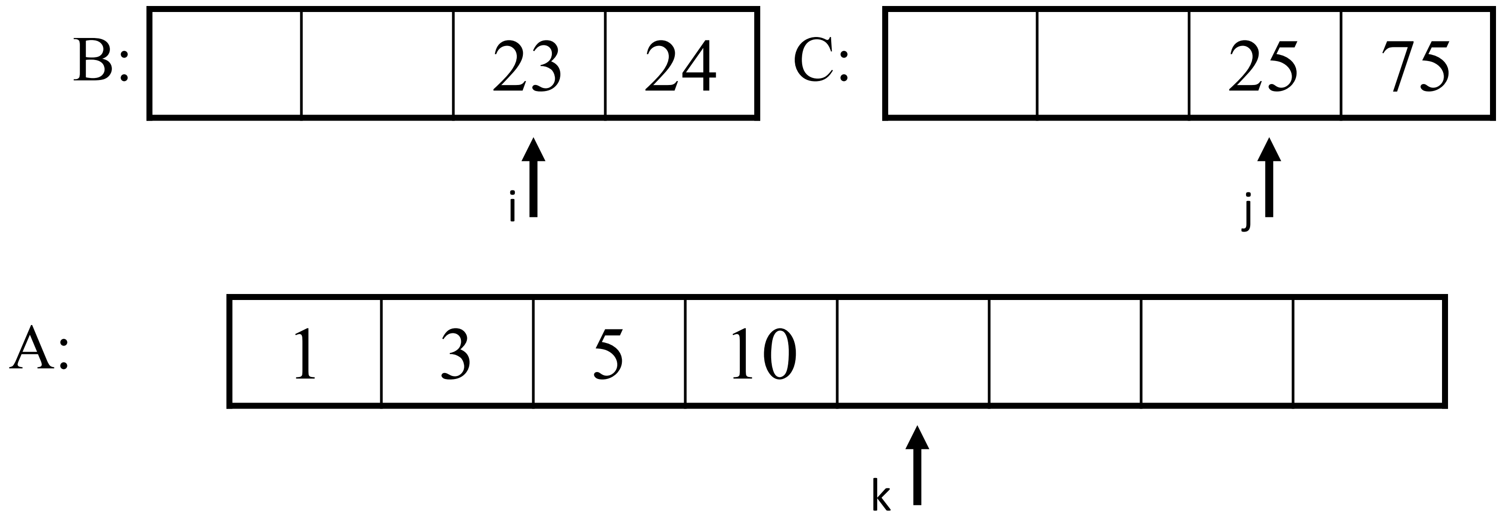
# Merging (cont.)



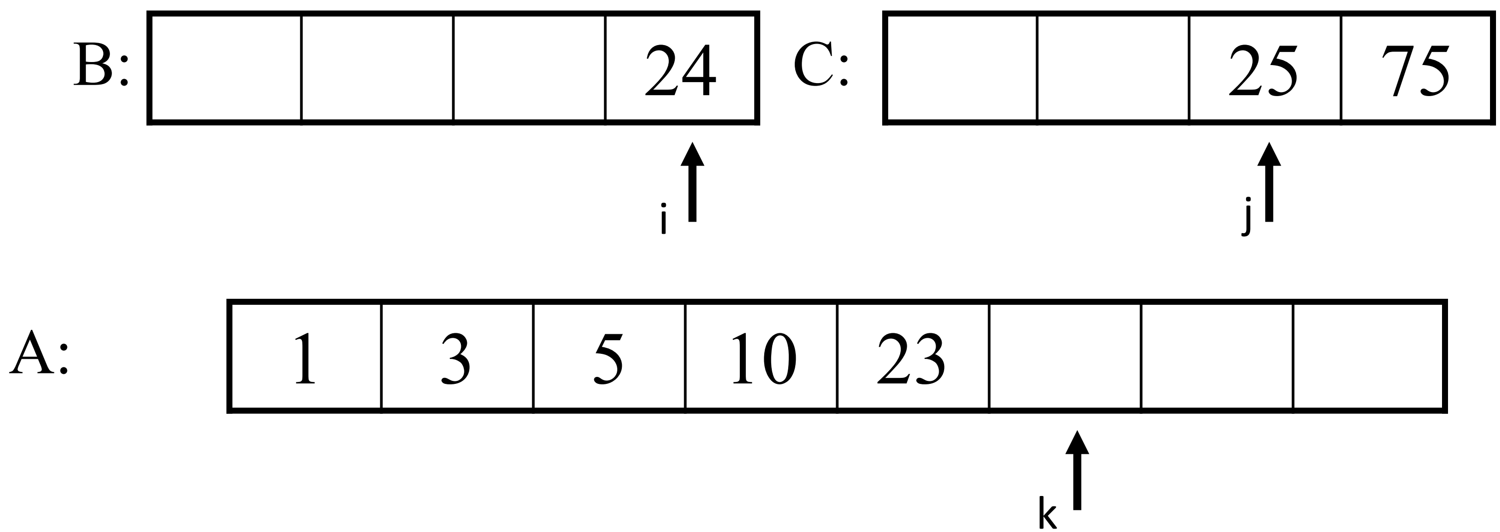
# Merging (cont.)



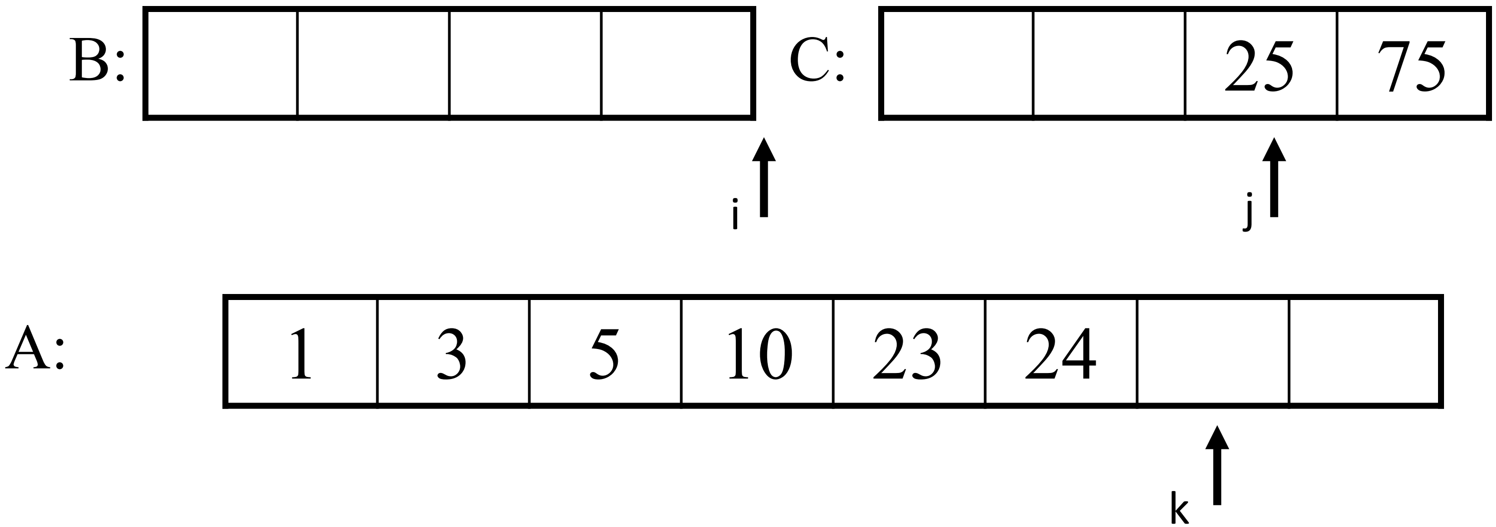
# Merging (cont.)



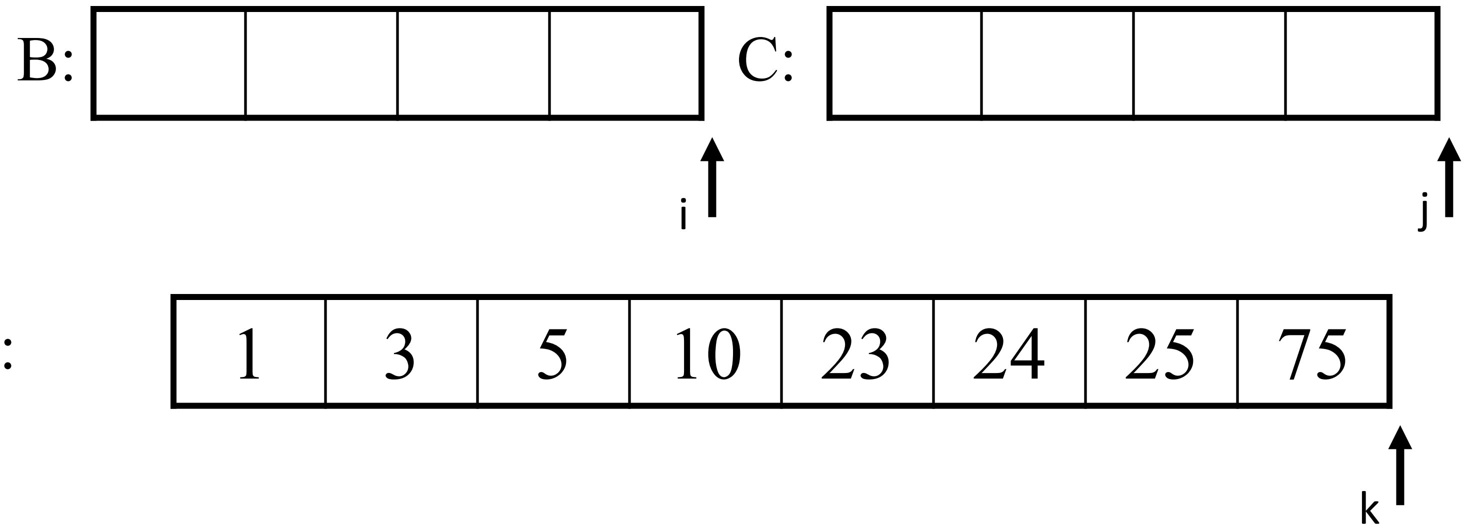
# Merging (cont.)



# Merging (cont.)



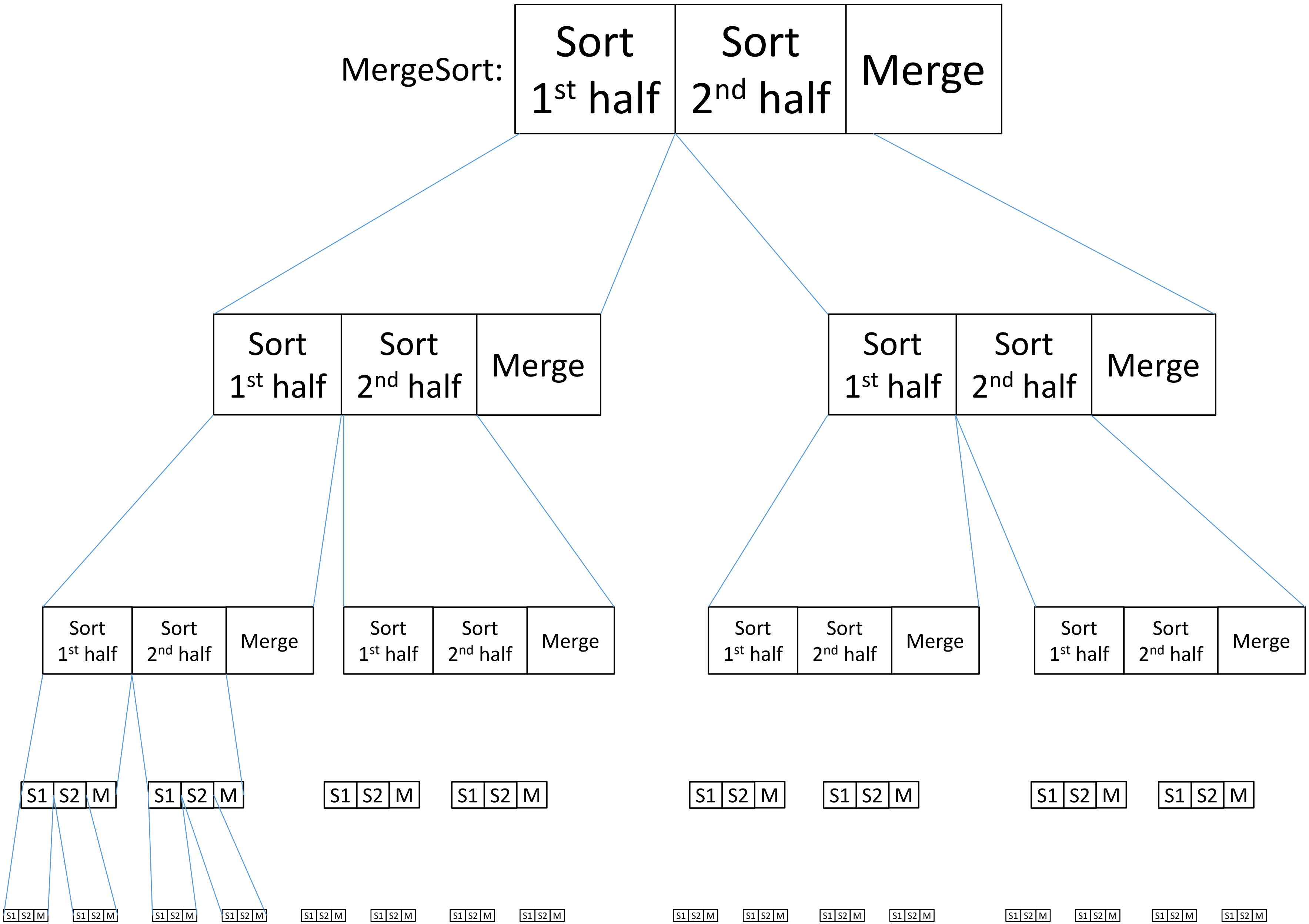
# Merging (cont.)



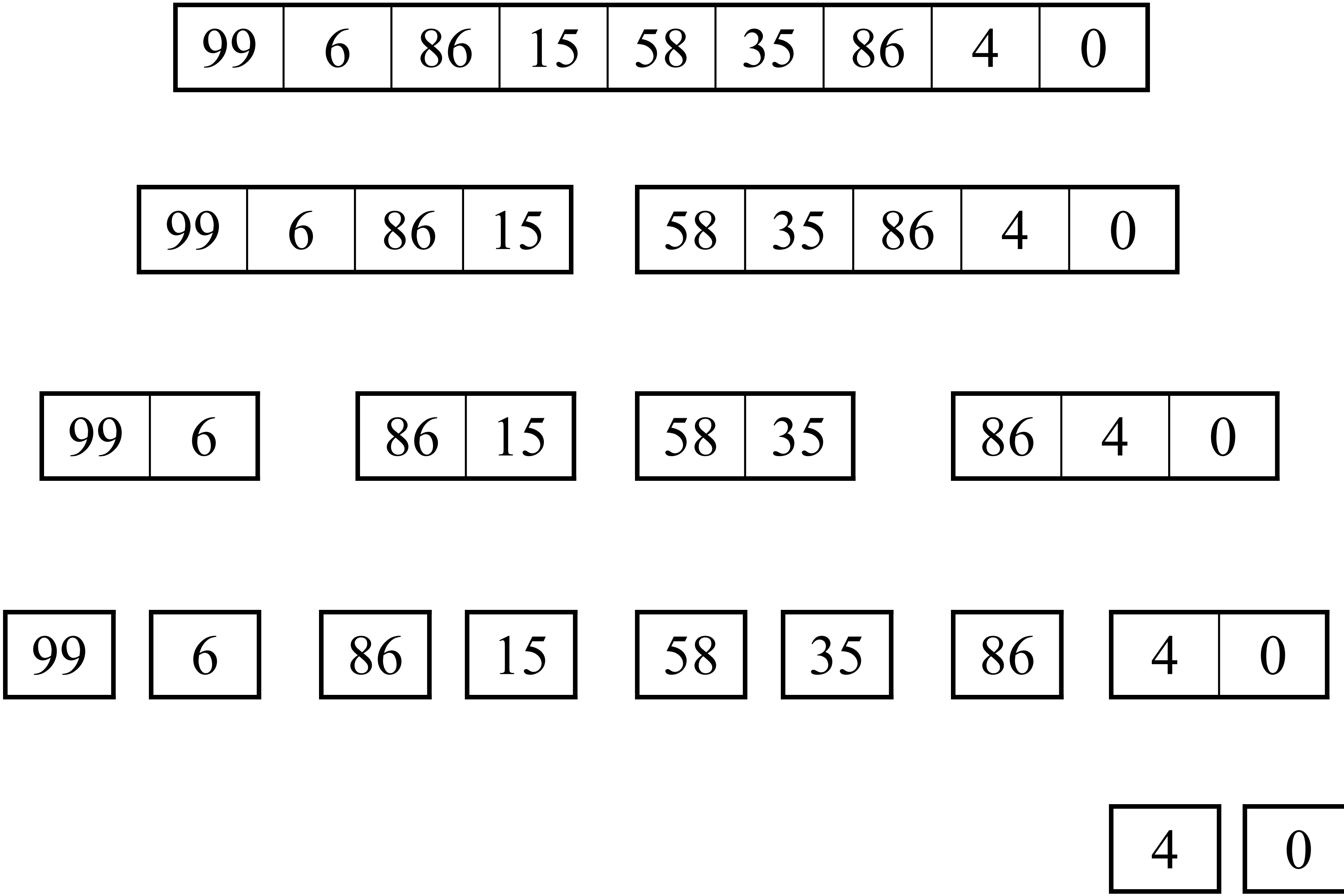
# Pseudocode of Merge

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

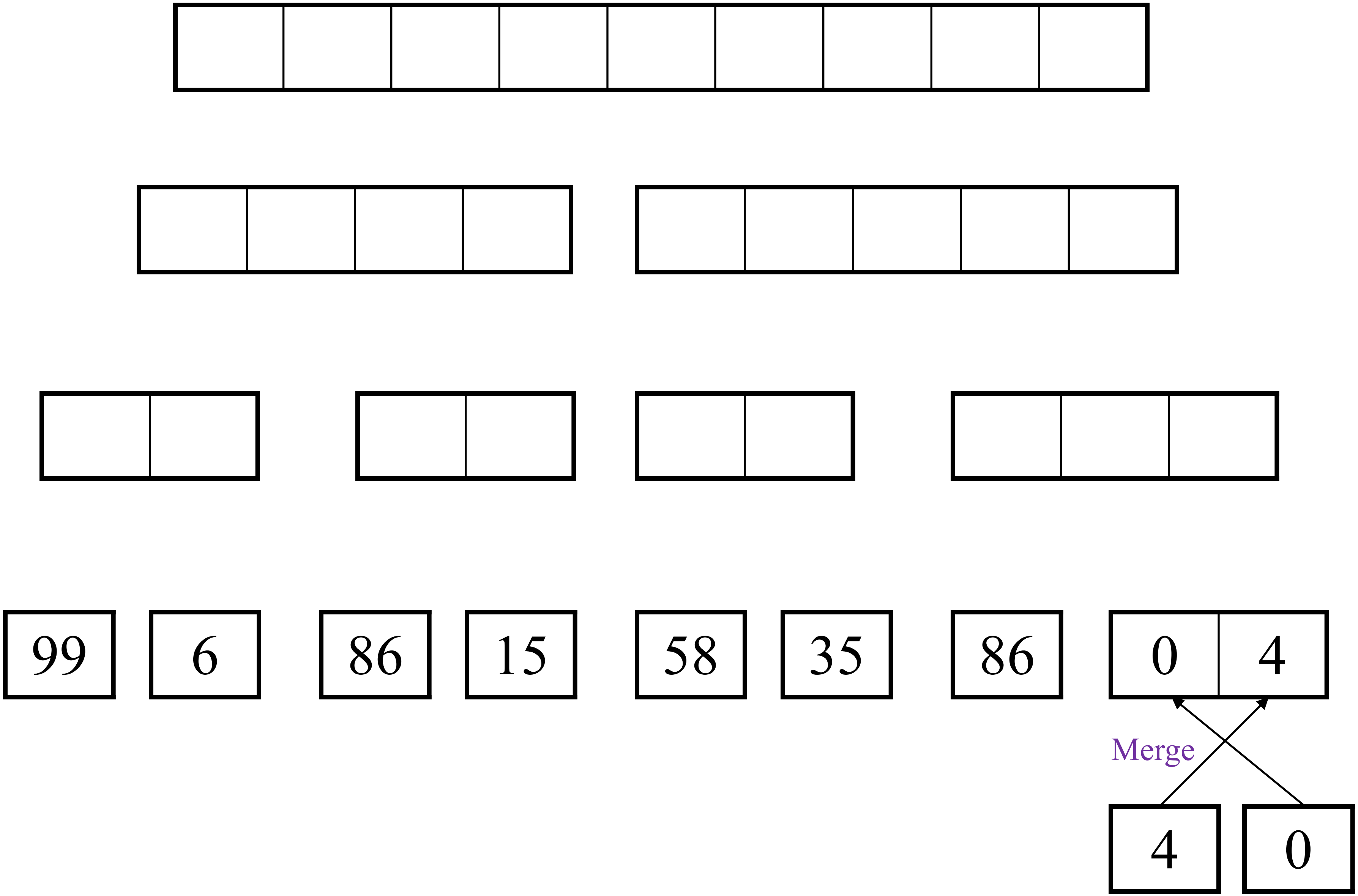
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
$i \leftarrow 0;\ \ j \leftarrow 0;\ \ k \leftarrow 0$
**while** $i < p$ **and** $j < q$ **do**
    **if** $B[i] \leq C[j]$
   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


MergeSort:
Sort 1st half
Sort 2nd half
Merge
Sort 1st half
Sort 2nd half
Merge
Sort 1st half
Sort 2nd half
Merge
S1 S2 M

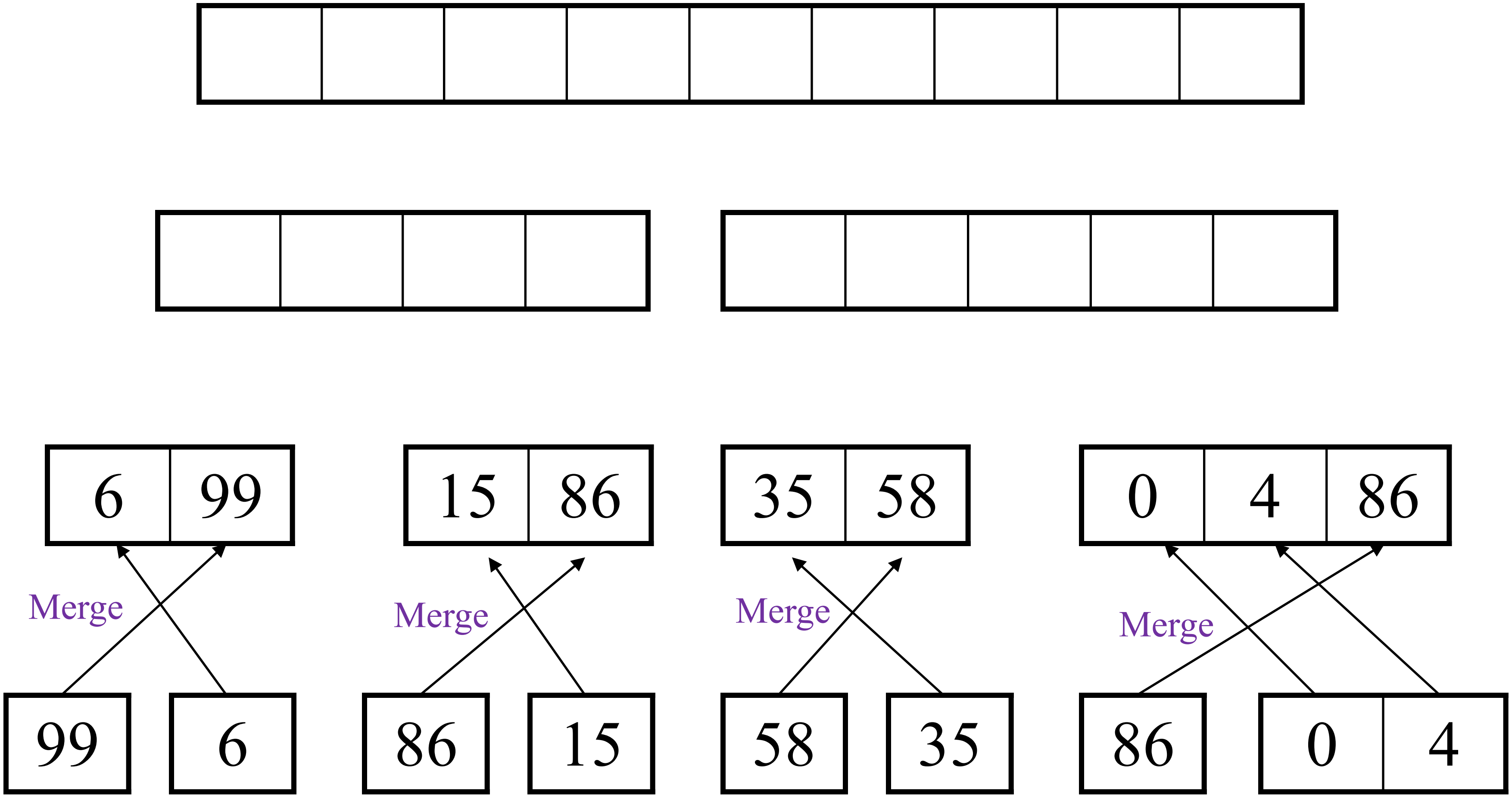
# Mergesort Example



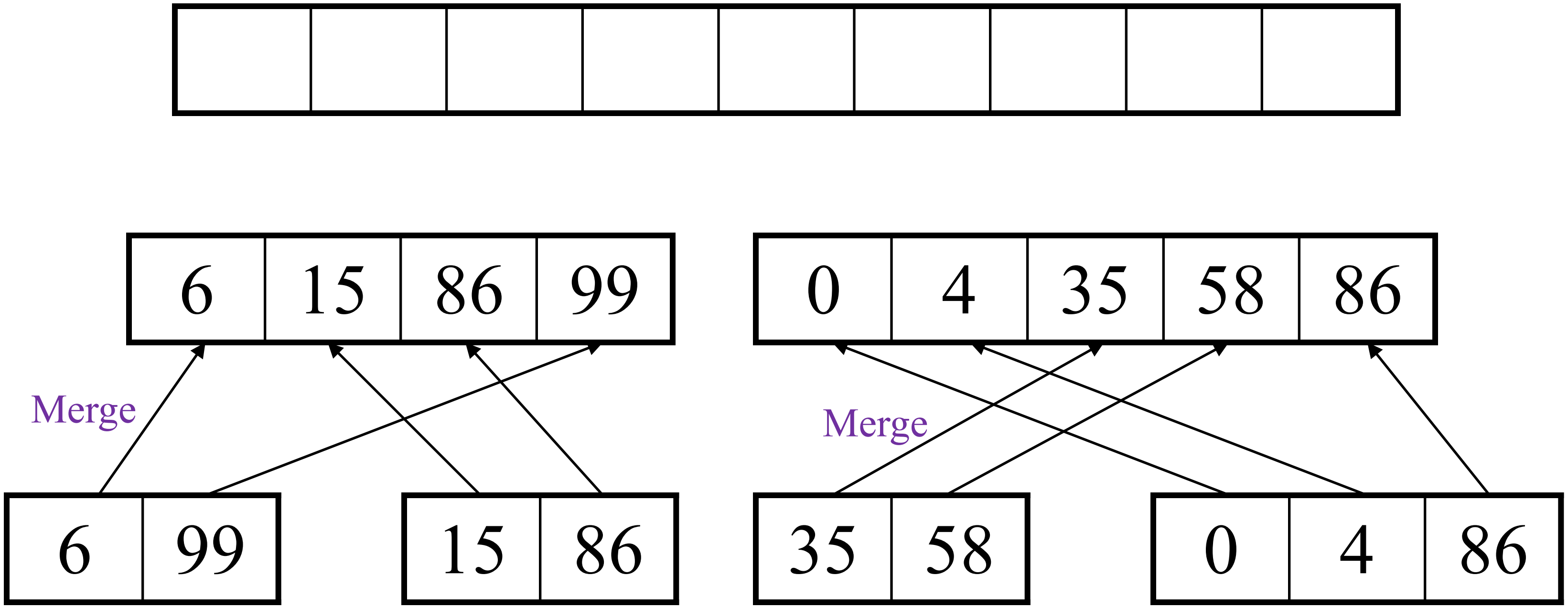
# Mergesort Example



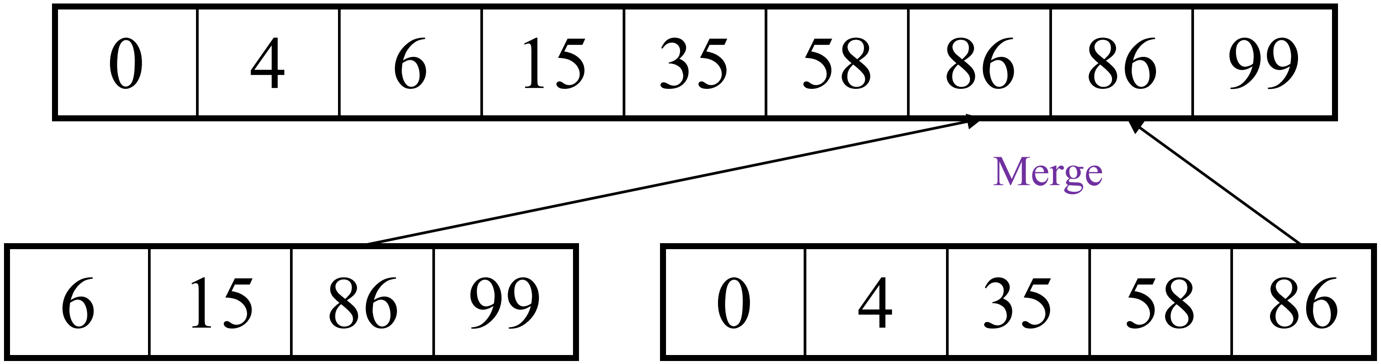
# Mergesort Example



# Mergesort Example



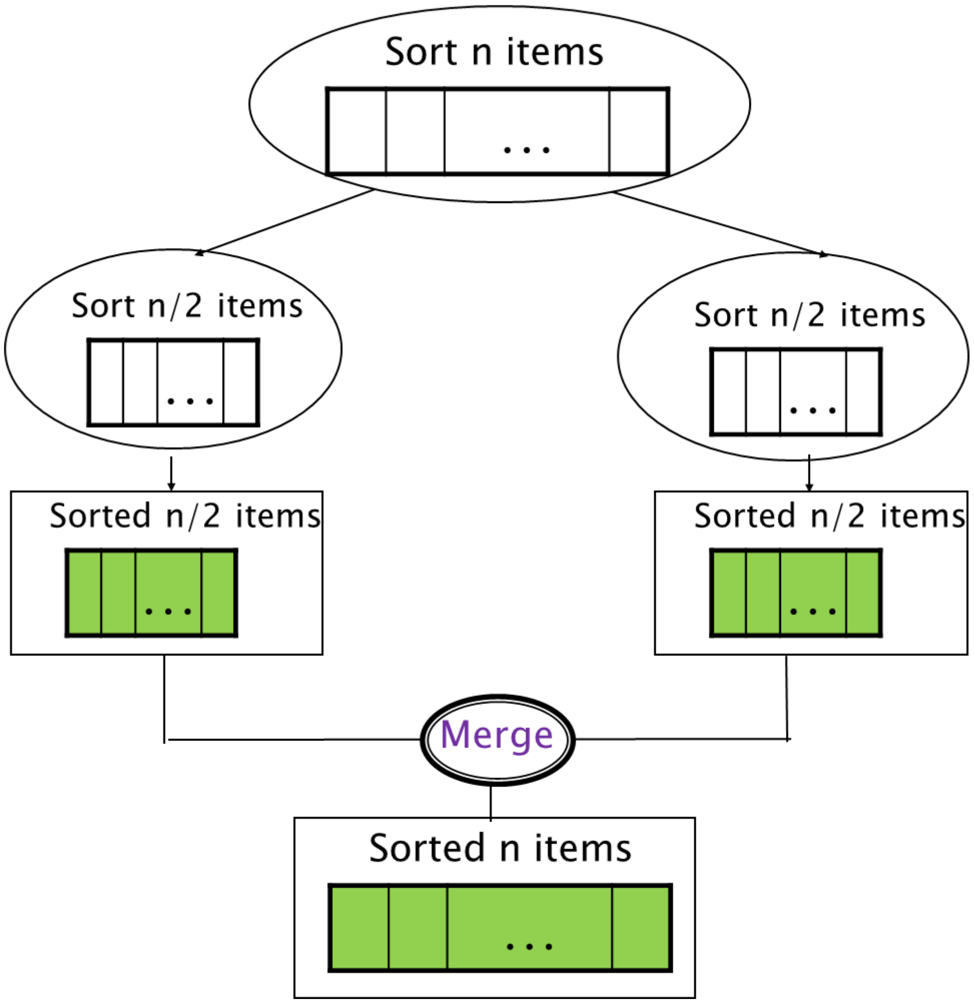
# Mergesort Example



# Mergesort Example

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 6 | 15 | 35 | 58 | 86 | 86 | 99 |

# Mergesort running time




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

Compare the two key expressions:

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

$F(n) = n$

They are equal!

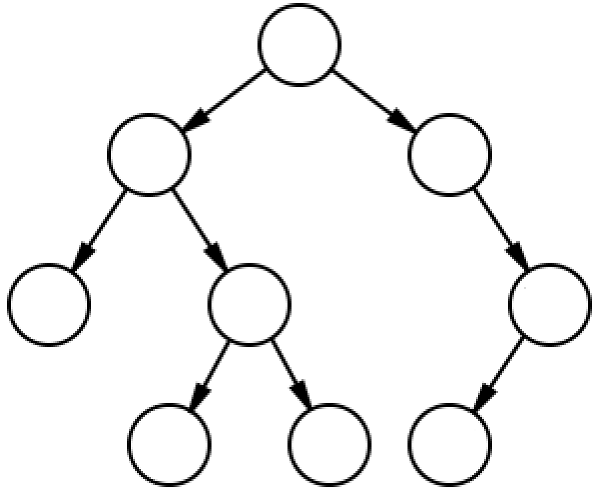
$\rightarrow T(n) \in O(n \log n)$

# Binary trees

# Binary tree structure

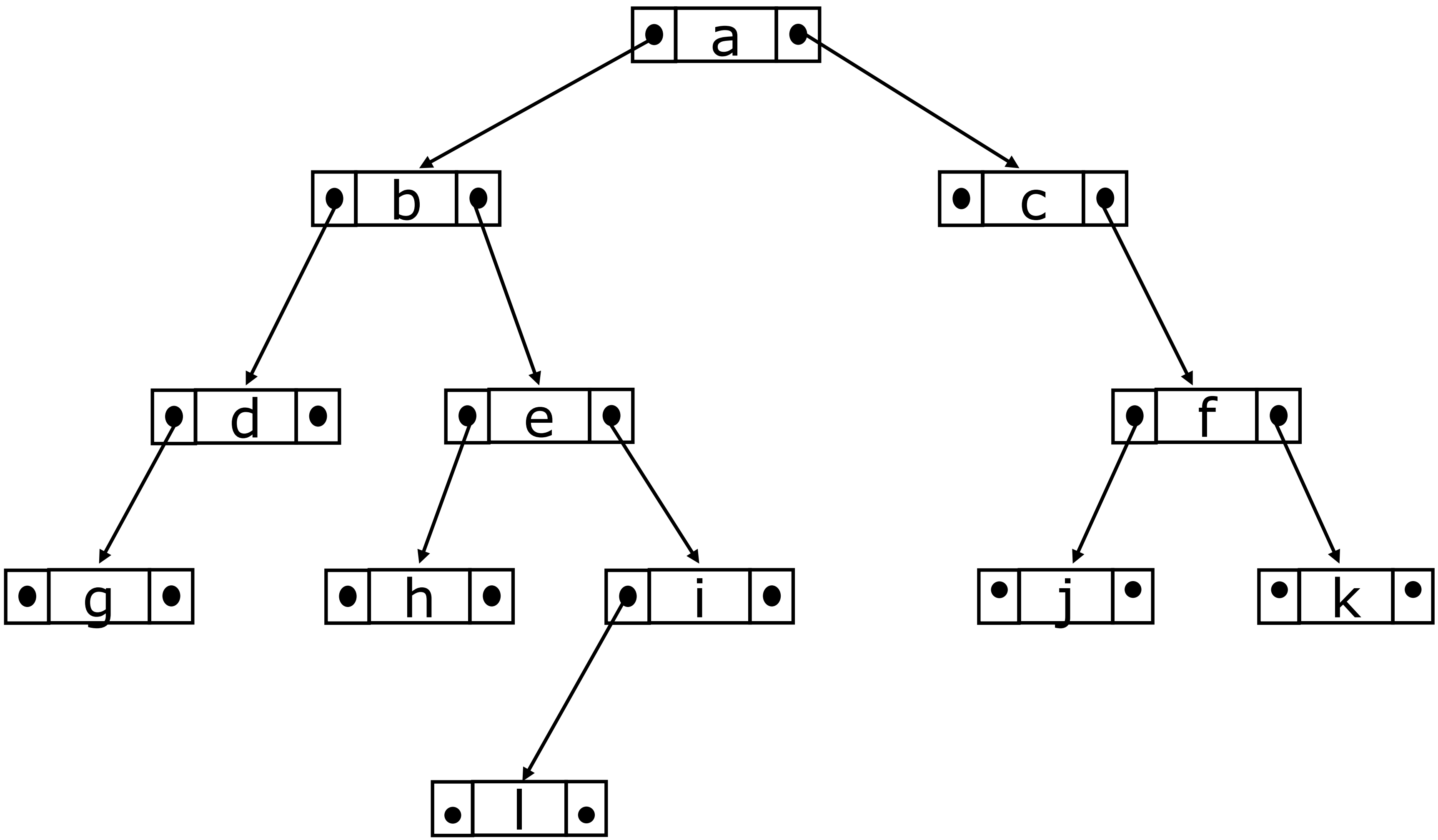
```
public class Node {
        public char data;
        public Node left;
        public Node right;

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

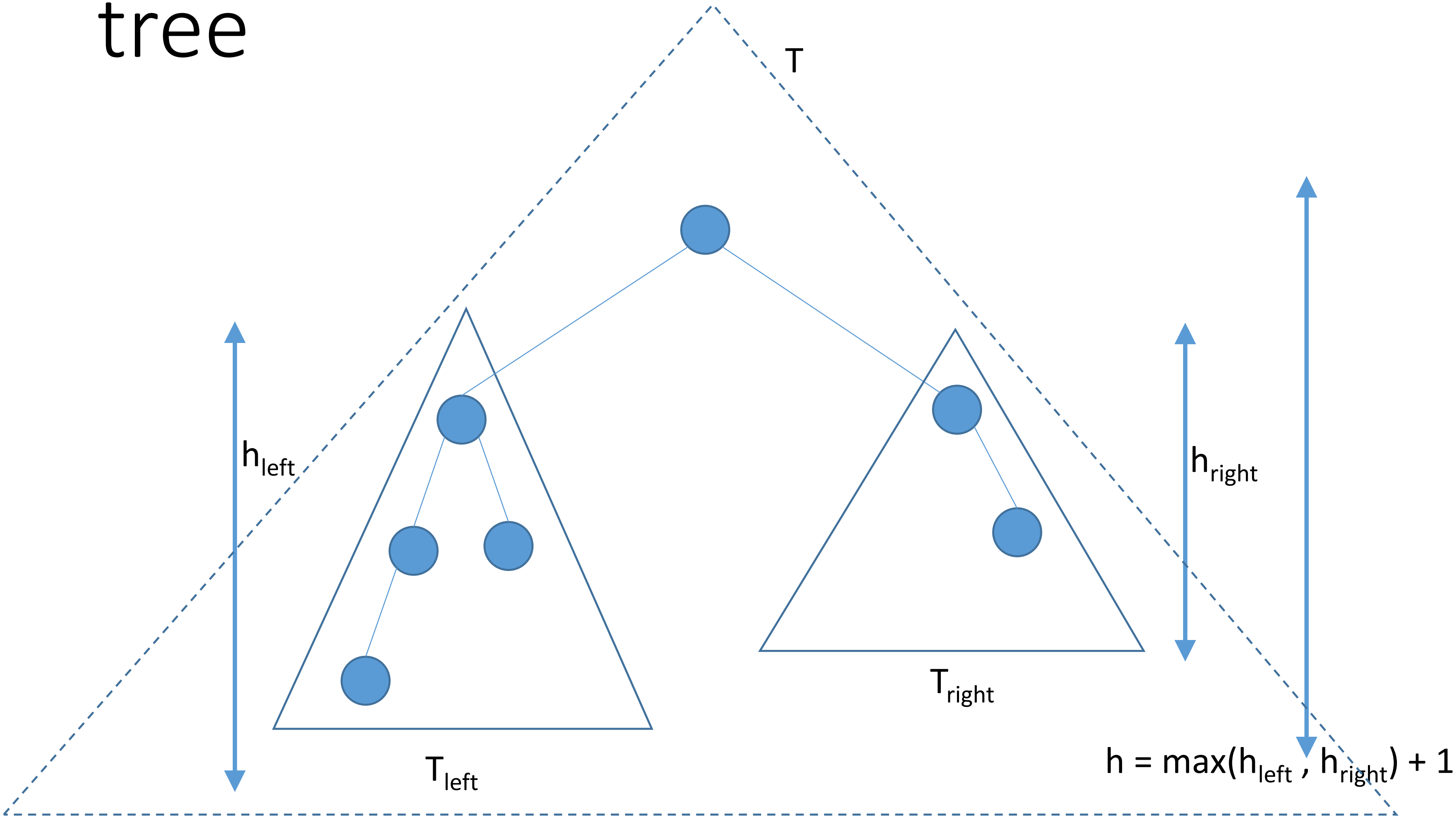



Node

# Binary tree implementation



# Computing the height of a binary tree



# 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.