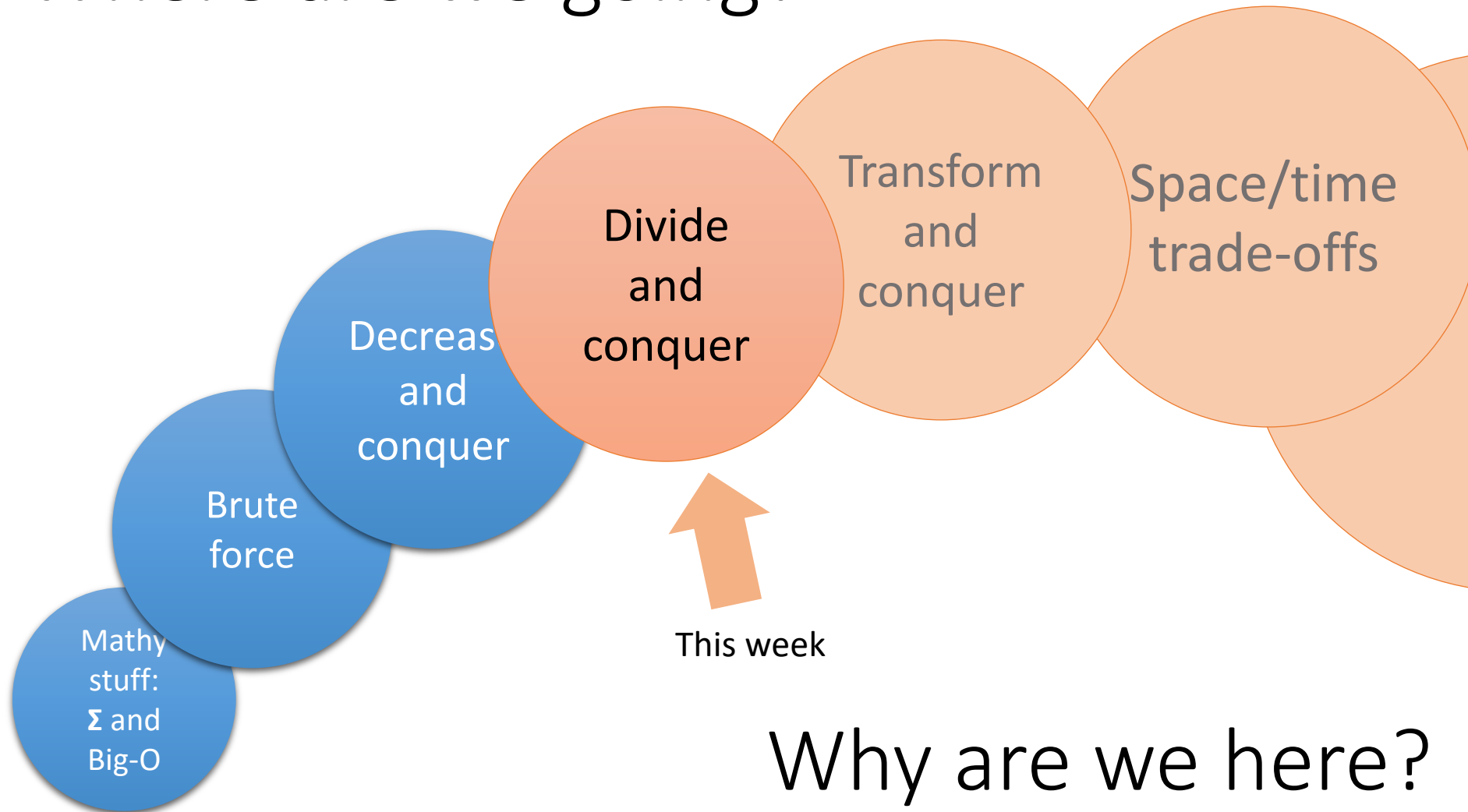


Where have we been?

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

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

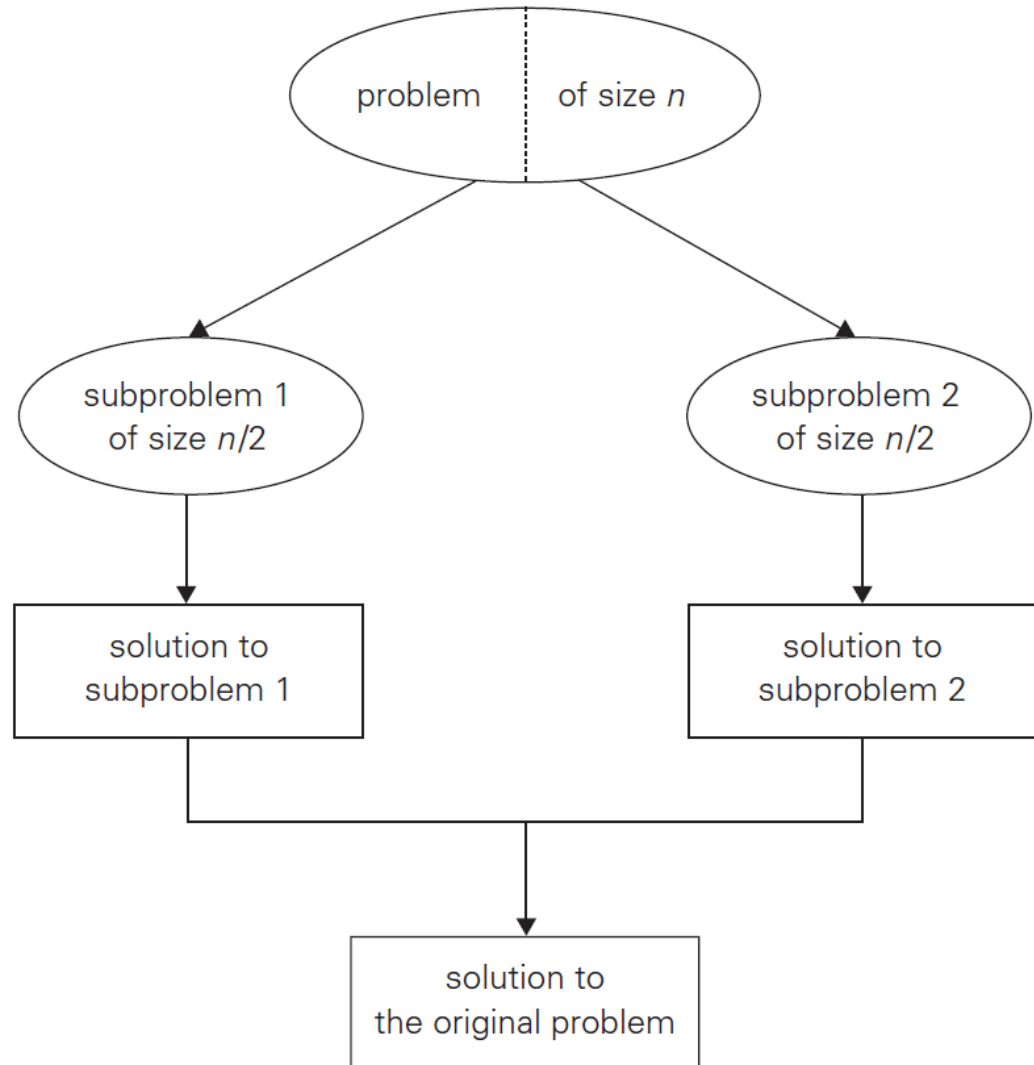
Divide and Conquer algorithms

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 the subproblems**

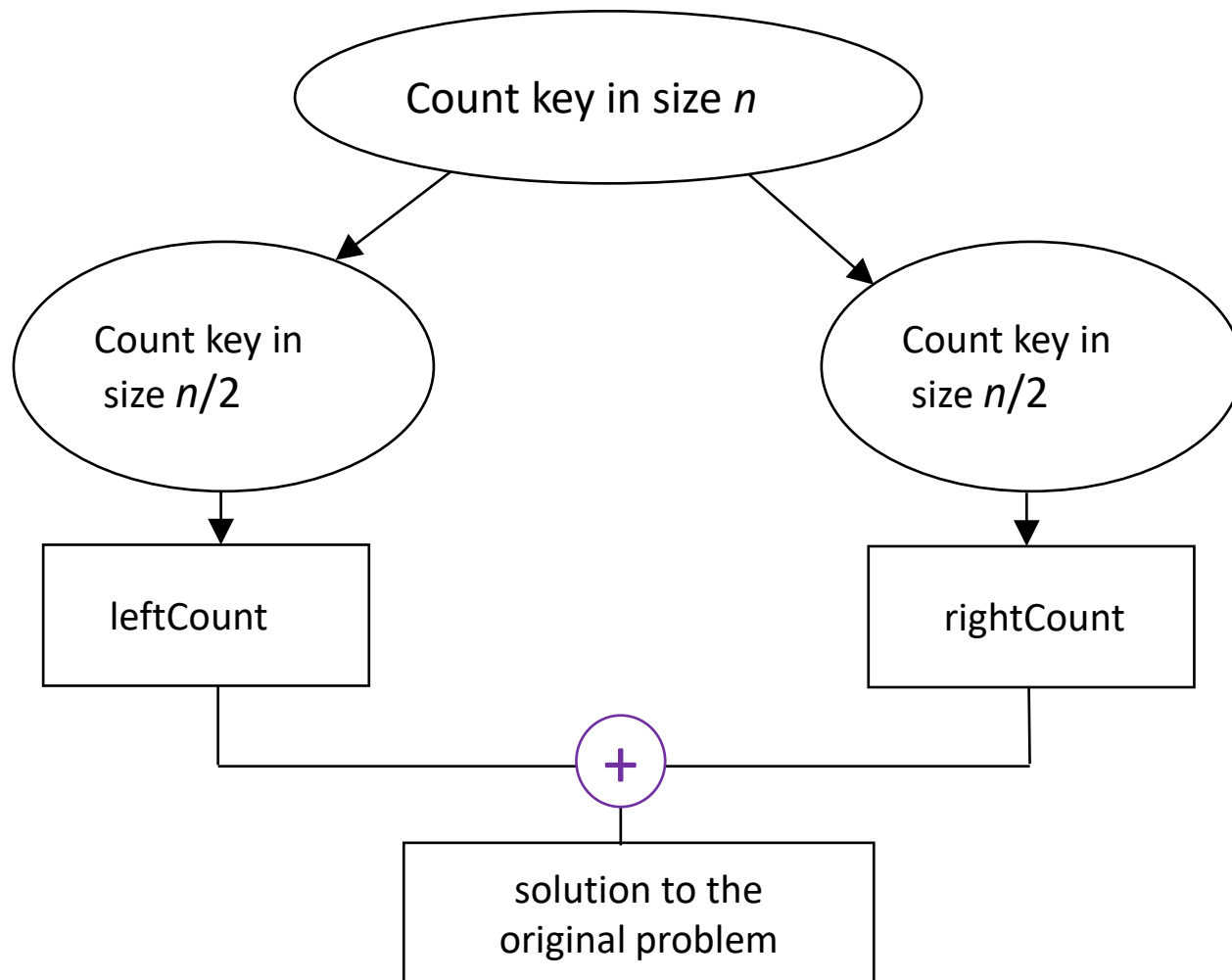
Example:

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 $\text{key}=6$...
 - ... 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 ($L \leq R$) are boundaries of the current search

//Output: The number of times Key exists in A[L..R]

```
1.  if L = R
2.      if (A[L] = Key) return 1
3.      else return 0
4.  else
5.      leftCount = CountKey(A[], Key, L,  $\lfloor (L+R)/2 \rfloor$ )
6.      rightCount = CountKey(A[], Key,  $\lfloor (L+R)/2 \rfloor + 1$ , R)
7.      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 ($L \leq R$) are boundaries of the current search

//Output: The number of times Key exists in A[L..R]

```
1.  if L = R
2.      if (A[L] = Key) return 1
3.      else return 0
4.  else
5.      leftCount = CountKey(A[], Key, L, ⌊(L+R)/2⌋)
6.      rightCount = CountKey(A[], Key, ⌊(L+R)/2⌋+1, R)
7.      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 ($L \leq R$) are boundaries of the current search
//Output: The number of times Key exists in A[L..R]

```
1.  if L = R
2.      if (A[L] = Key) return 1
3.      else return 0
4.  else
5.      leftCount = CountKey(A[], Key,
6.      rightCount = CountKey(A[], Key,
7.      return leftCount + rightCount
```

L, $\lfloor (L+R)/2 \rfloor$
 $\lfloor (L+R)/2 \rfloor + 1$, R)

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 ($L \leq R$) are boundaries of the current search
//Output: The number of times Key exists in A[L..R]

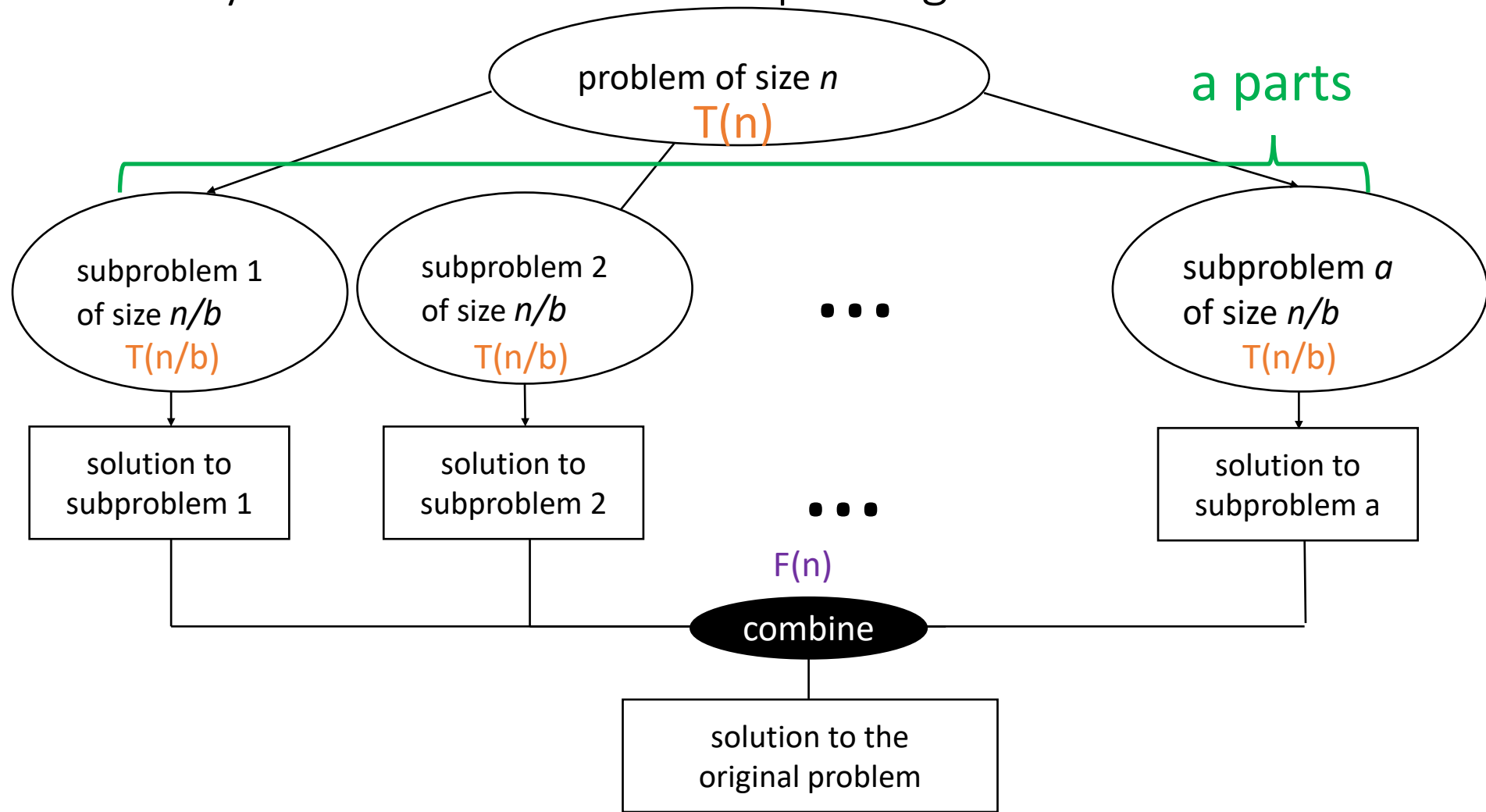
```
1.  if L = R
2.      if (A[L] = Key) return 1
3.      else return 0
4.  else
5.      leftCount = CountKey(A[], Key, L,      ⌊(L+R)/2⌋)
6.      rightCount = CountKey(A[], Key,  ⌊(L+R)/2⌋+1, R)
7.      return leftCount + rightCount
```

additional computation

time is $O(1)$

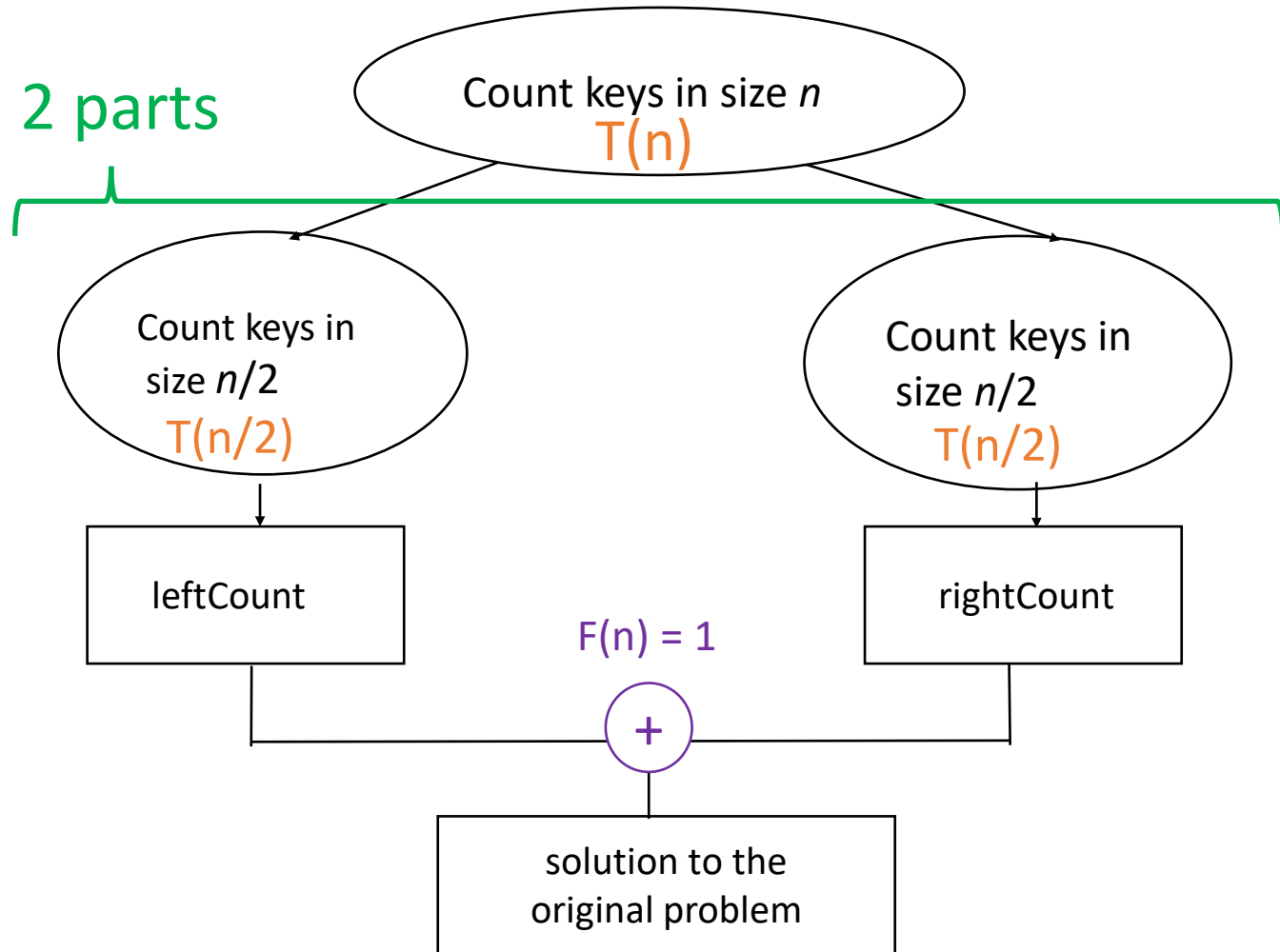
$F(n) = 1$

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

Another version

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

Master Theorem examples

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

$$\begin{array}{l} a = 4 \\ b = 2 \\ F(n) = n \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \begin{array}{c} n^{\log_b a} \\ n^{\log_2 4} \\ n^2 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \left. \begin{array}{c} n^2 \\ F(n) = n \end{array} \right\} \rightarrow T(n) \in O(n^2)$$

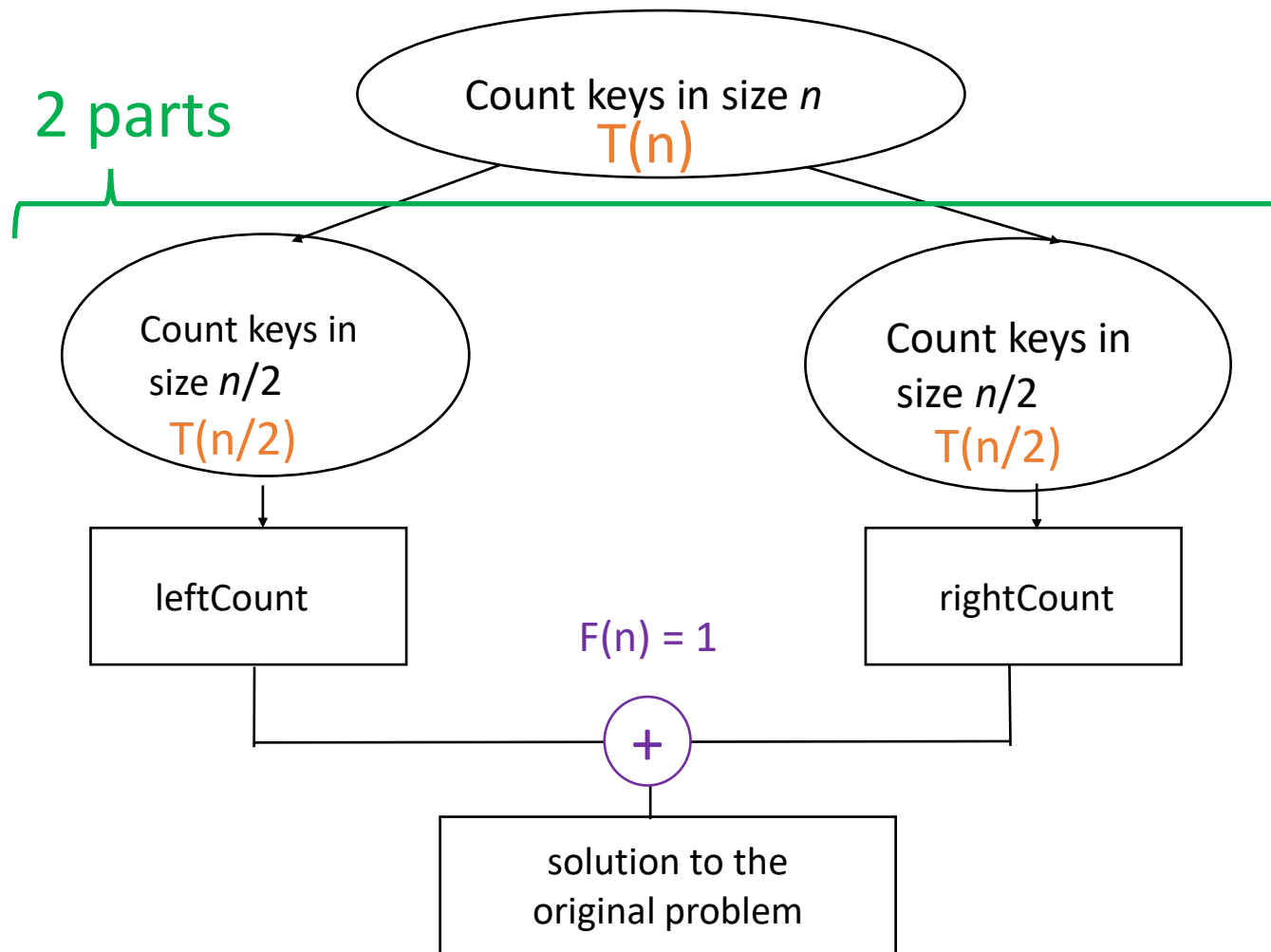
Example 2: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$$\begin{array}{l} a = 4 \\ b = 2 \\ F(n) = n^2 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \begin{array}{c} n^{\log_b a} \\ n^{\log_2 4} \\ n^2 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \left. \begin{array}{c} n^2 \\ F(n) = n^2 \end{array} \right\} \rightarrow T(n) \in O(n^2 \log n)$$

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

$$\begin{array}{l} a = 4 \\ b = 2 \\ F(n) = n^3 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \begin{array}{c} n^{\log_b a} \\ n^{\log_2 4} \\ n^2 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \left. \begin{array}{c} n^2 \\ F(n) = n^3 \end{array} \right\} \rightarrow T(n) \in O(n^3)$$

Back to CountKeys again



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

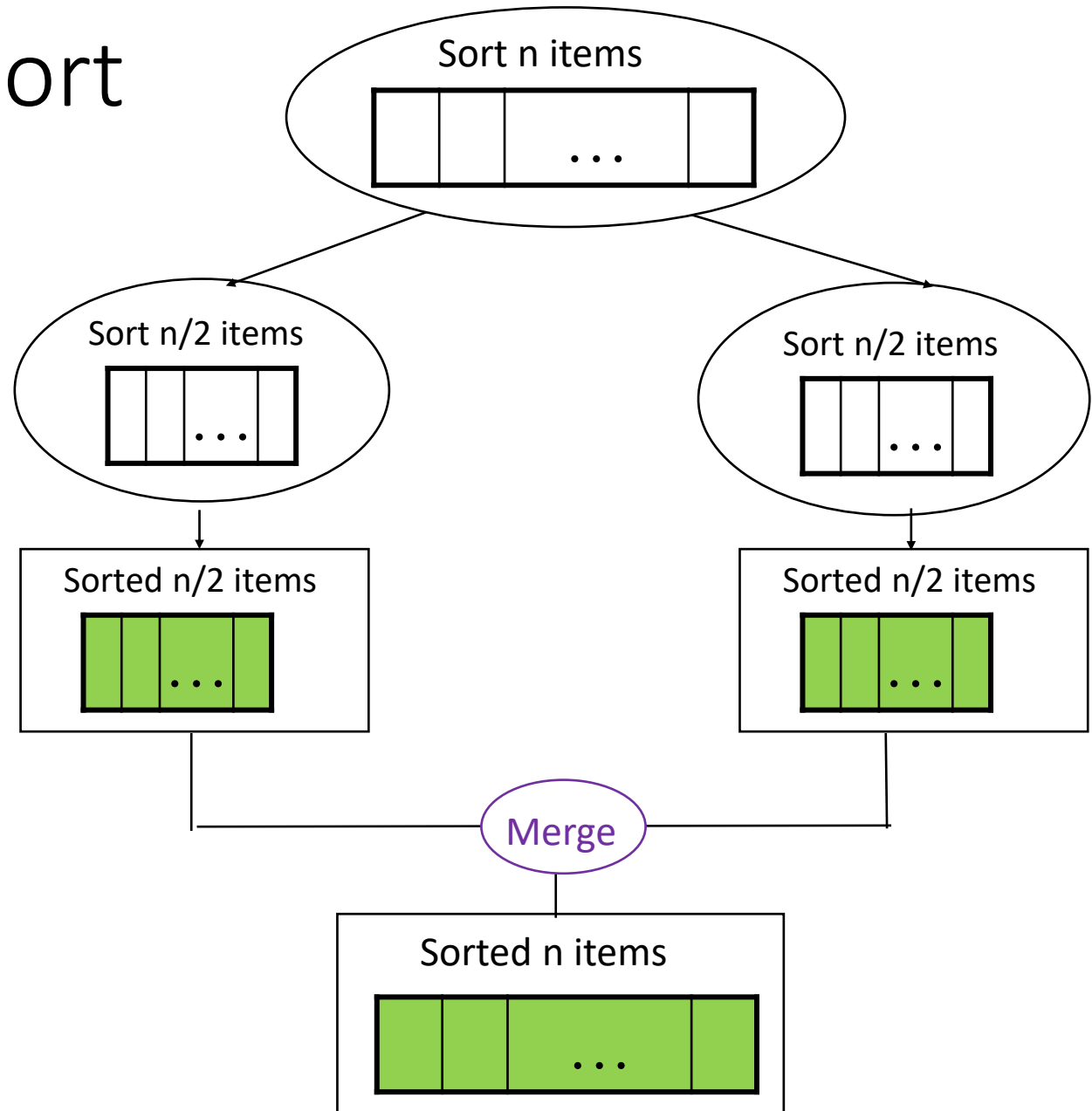
$$\Rightarrow T(n) \in O(n)$$

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



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($B[0..\lfloor n/2 \rfloor - 1]$)

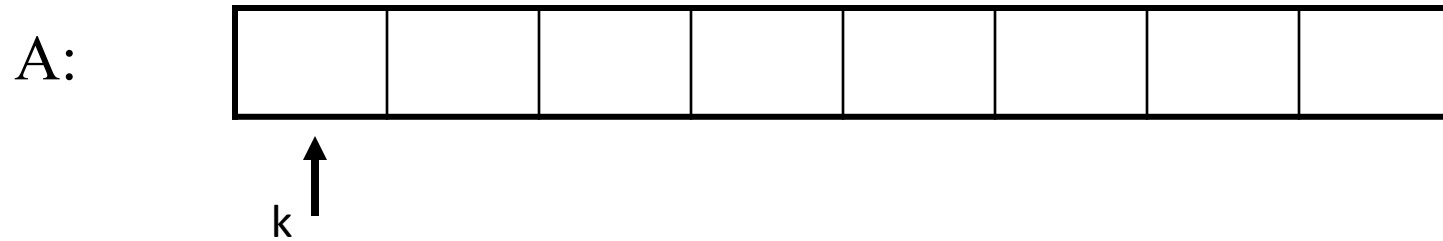
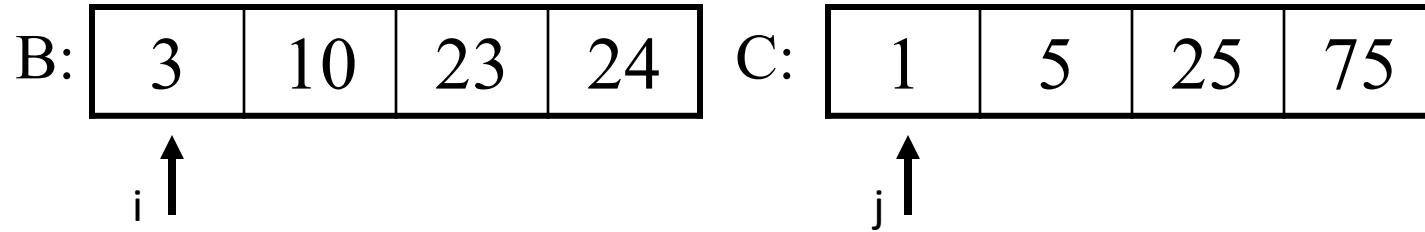
Mergesort($C[0..\lceil n/2 \rceil - 1]$)

Merge(B, C, A)

Mergesort

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

Merging



Merging (cont.)

B:

3	10	23	24
---	----	----	----



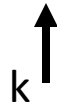
C:

	5	25	75
--	---	----	----



A:

1							
---	--	--	--	--	--	--	--



Merging (cont.)

B:

	10	23	24
--	----	----	----



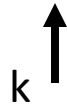
C:

	5	25	75
--	---	----	----



A:

1	3						
---	---	--	--	--	--	--	--



Merging (cont.)

B:

	10	23	24
--	----	----	----



C:

		25	75
--	--	----	----

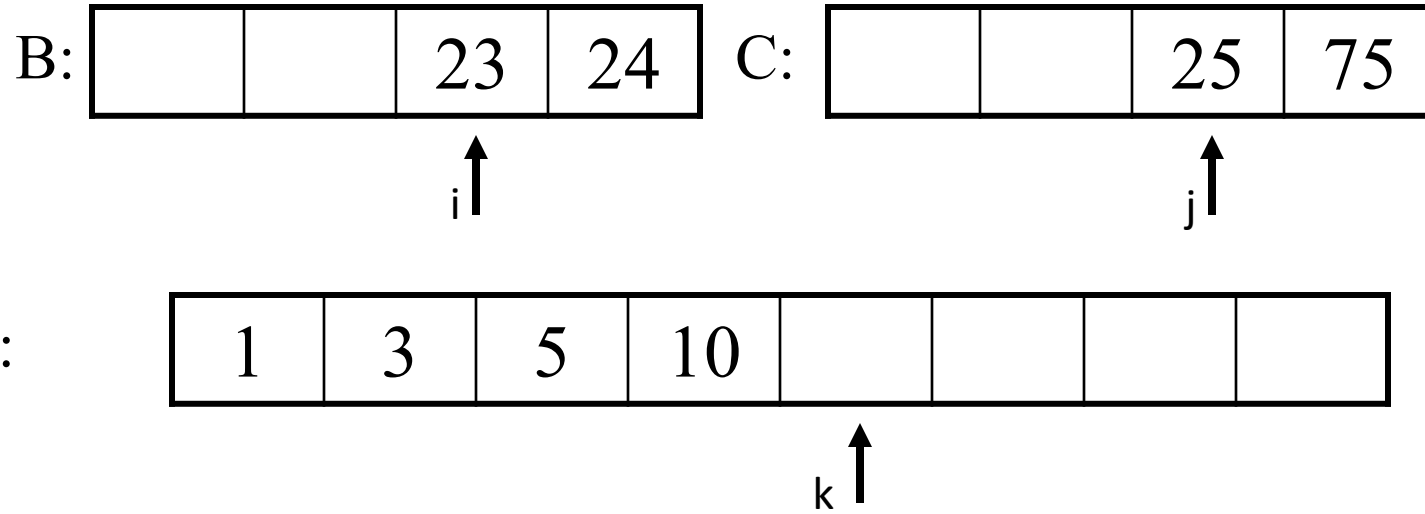


A:

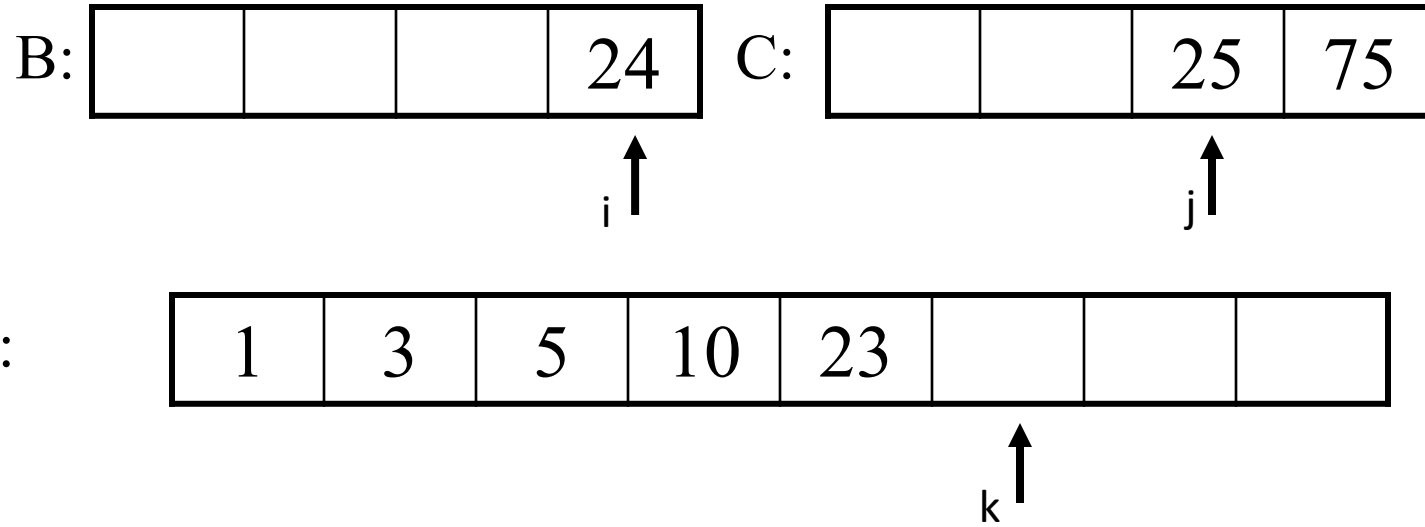
1	3	5					
---	---	---	--	--	--	--	--



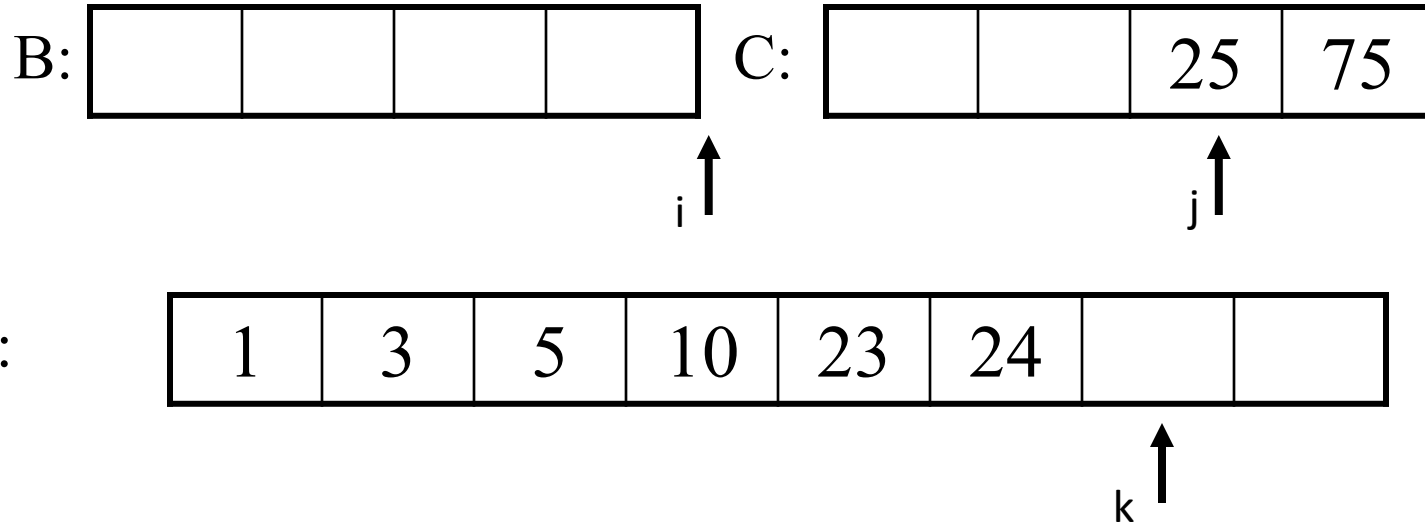
Merging (cont.)



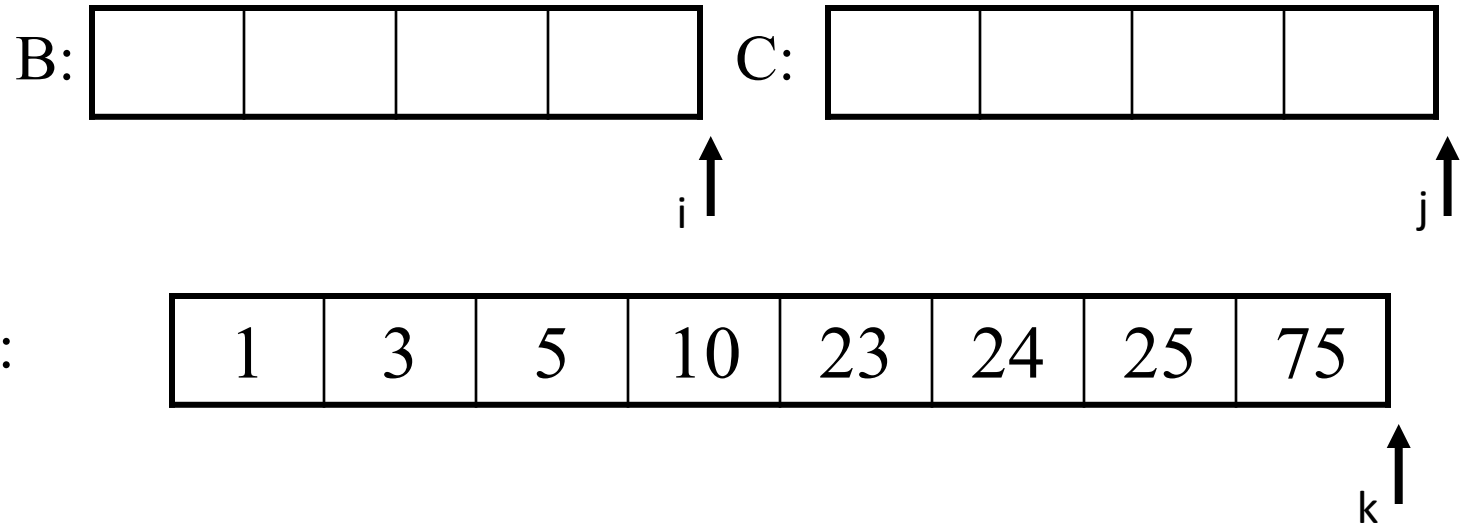
Merging (cont.)



Merging (cont.)



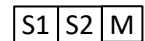
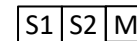
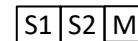
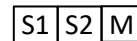
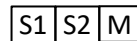
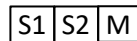
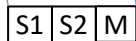
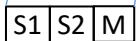
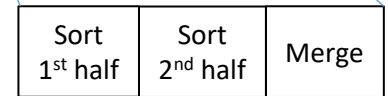
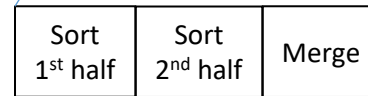
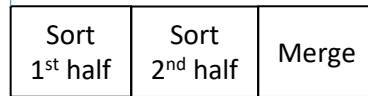
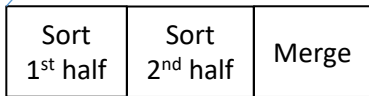
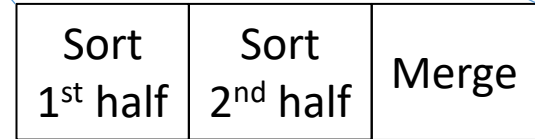
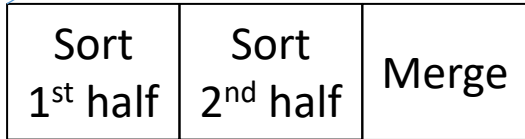
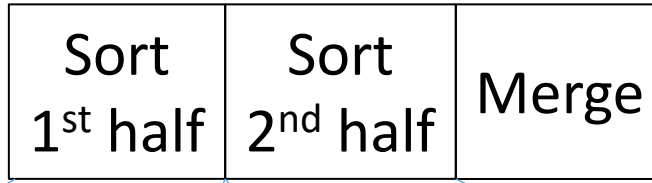
Merging (cont.)



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:



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

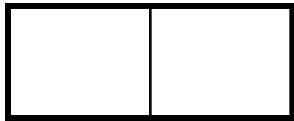
86

4	0
---	---

4

0

Mergesort Example



99

6

86

15

58

35

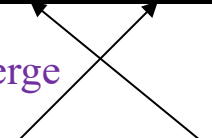
86

0 4

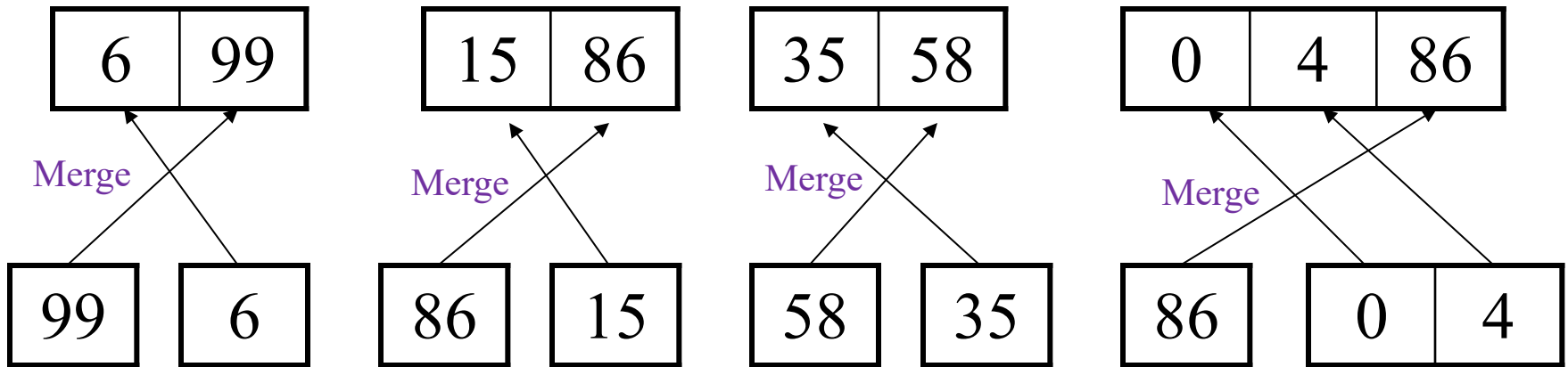
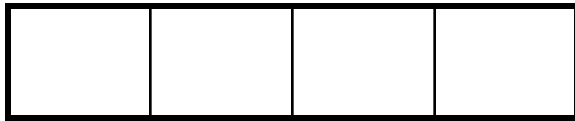
Merge

4

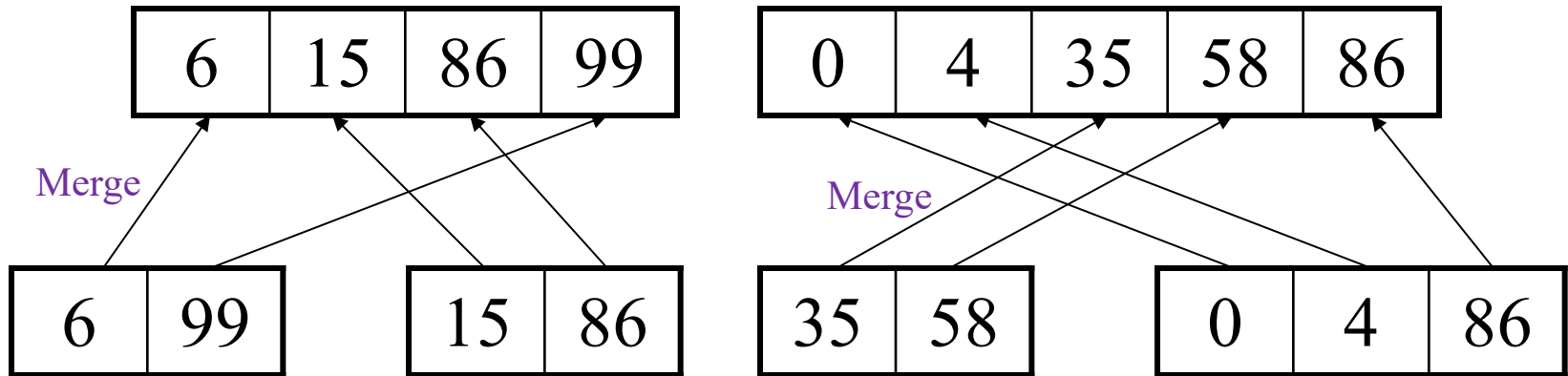
0



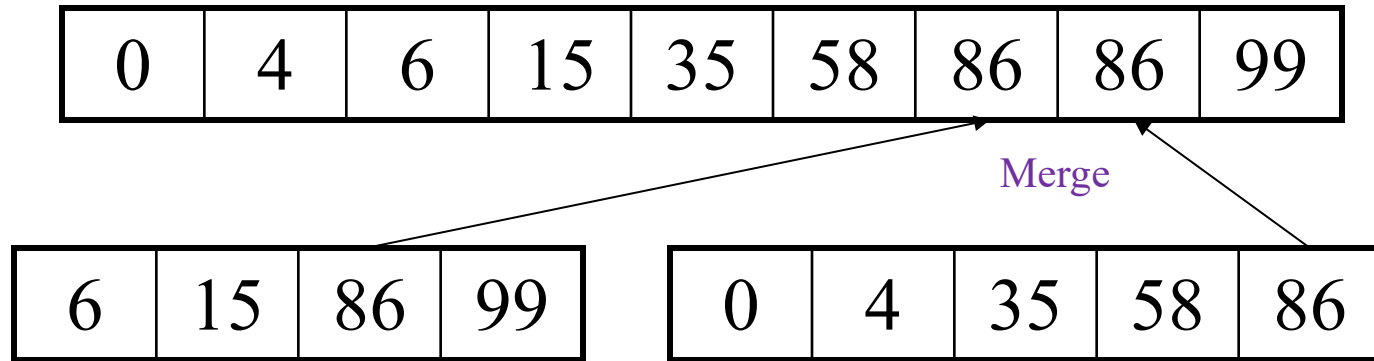
Mergesort Example



Mergesort Example



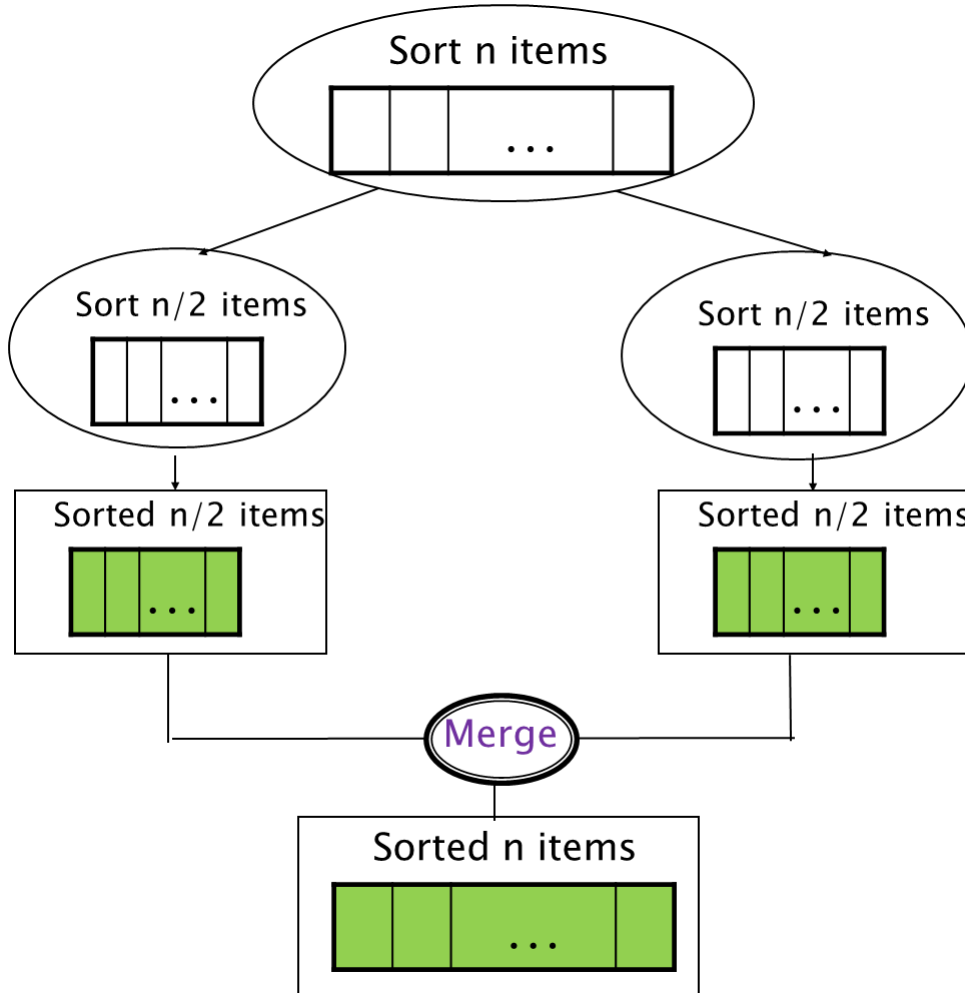
Mergesort Example



Mergesort Example

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

Mergesort running time



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

Compare the two key expressions:

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

$$F(n) = n$$

They are equal!

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

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.