# CS430-01
# Introduction to Algorithms
# 2. Divide & Conquer Algorithm

Michael Choi
Dept. of Computer Science
IIT

## Algorithm Design

- Different design approaches depend on the problem
- Think of insertion sort as “incremental”
  - Why?
- We will next discuss another design approach: **divide-and-conquer** for sorting
- Has anyone heard about divide-and-conquer?

## Divide and Conquer Principle

- **DIVIDE** the problem into $\boxed{\text{smaller}}$ and similar sub-problems
- **CONQUER** the sub-problems by
  - dividing them $\boxed{\text{recursively,}}$ or
  - solve them in a directly when trivial
- **COMBINE** the solutions of the sub-problems

## How does this work for sorting? Merge Sort

- **INPUT:** $<a_1,a_2,\ldots a_n>$
- **DIVIDE:** the n-element sequence in two n/2-element sequences
- **CONQUER:** sort the two sequences
  - By dividing them recursively, or
  - Simply return a singleton element when the subsequence is of size 1 (bottom of recursion)
- **COMBINE:** *Merge* two sorted lists into a single sorted list (hence the name)

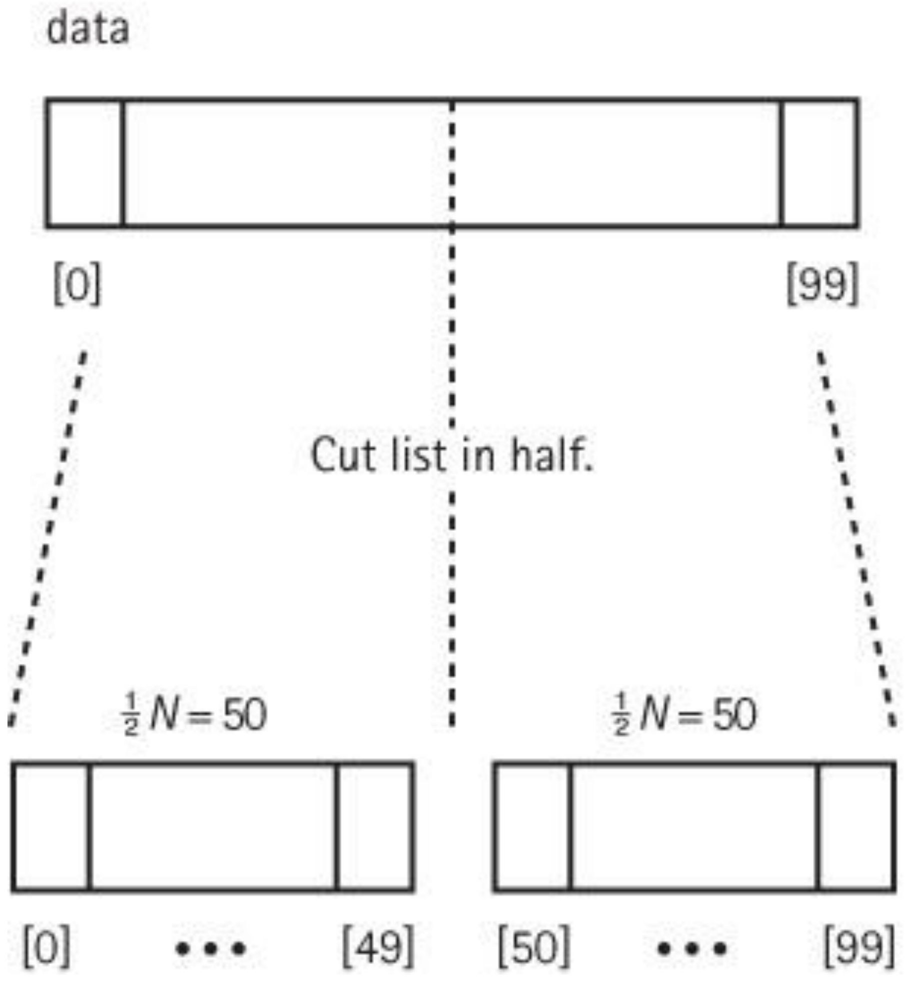
## Divide and Conquer: Sorting

How should we split the data?

What are the sub-problems we need to solve?

How do we combine the results from these sub-problems?

# Rationale for Divide and Conquer



$N = 100$

$N^2 = (100)^2 = 10{,}000$

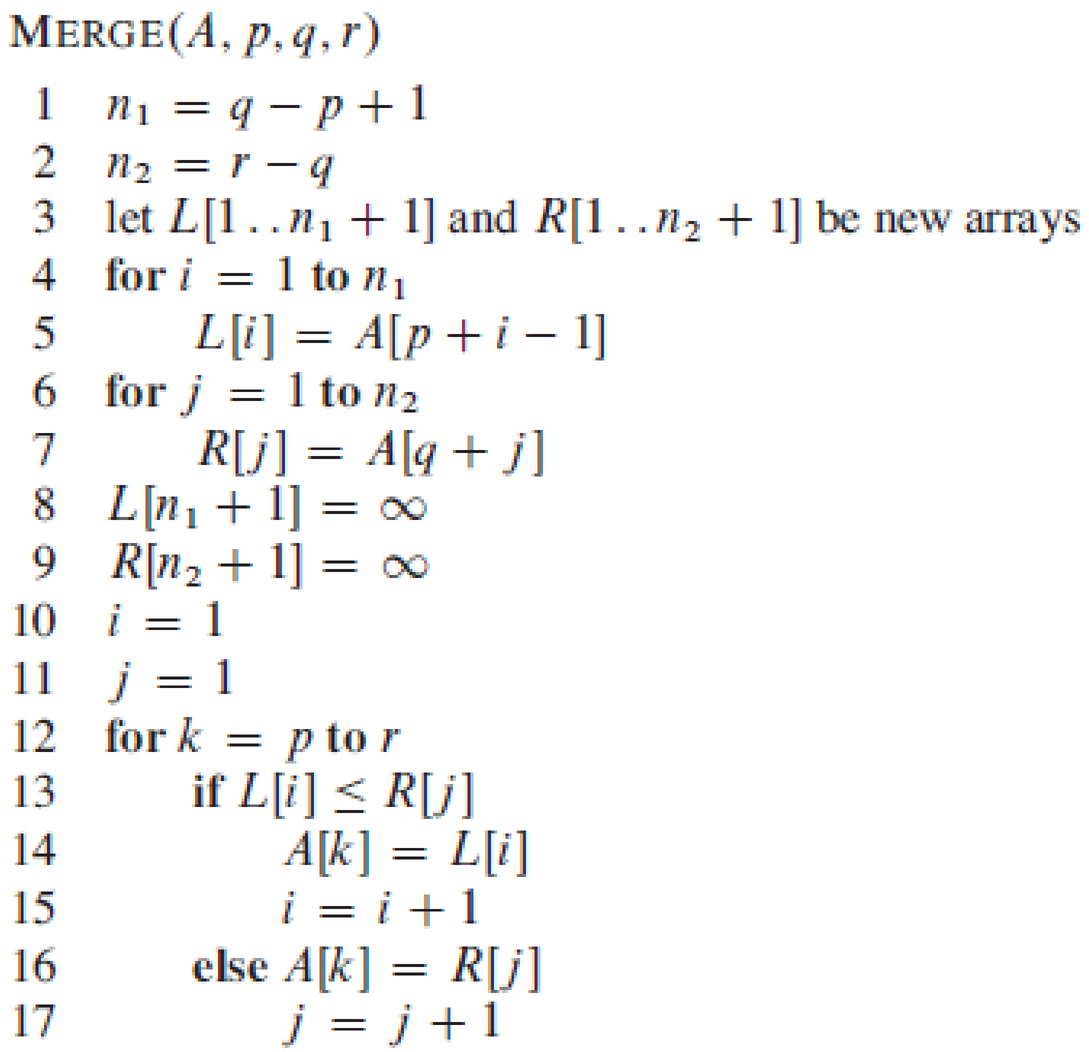
$(\frac{1}{2}N)^2 + (\frac{1}{2}N)^2$ To sort

$+ \quad N$ To merge

$= (50)^2 + (50)^2 + 100$

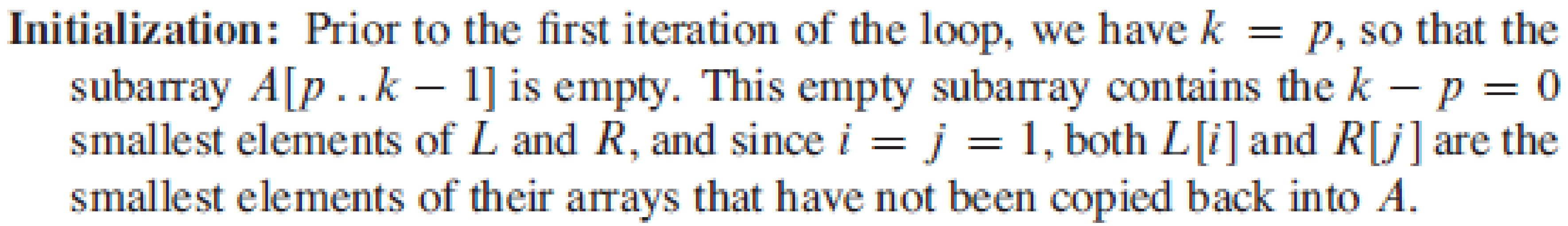
$= 5100$

# Merge algorithm in textbook

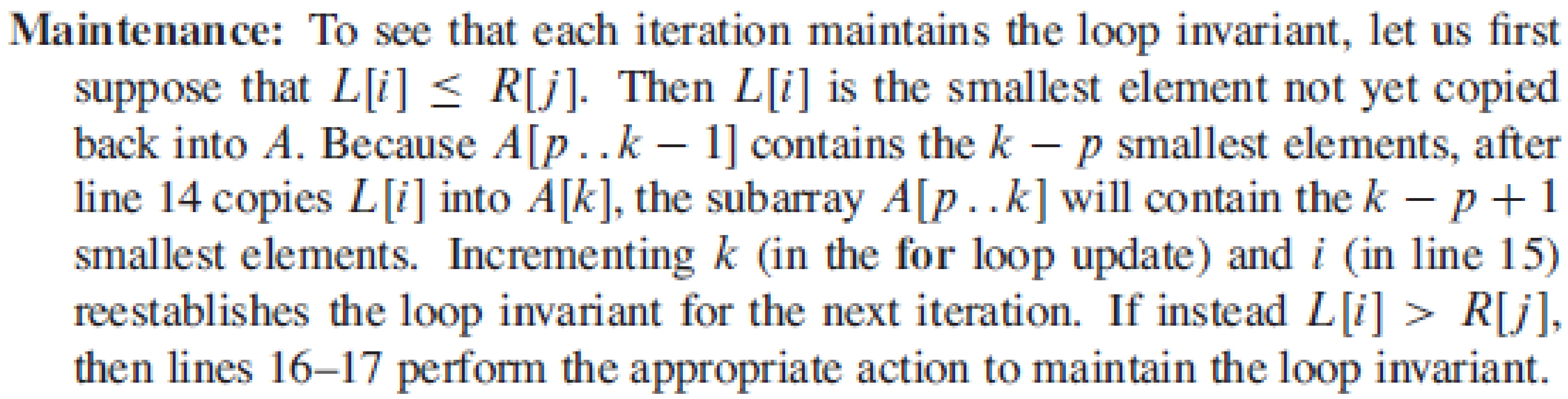


```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

# Merge algorithm loop invariant



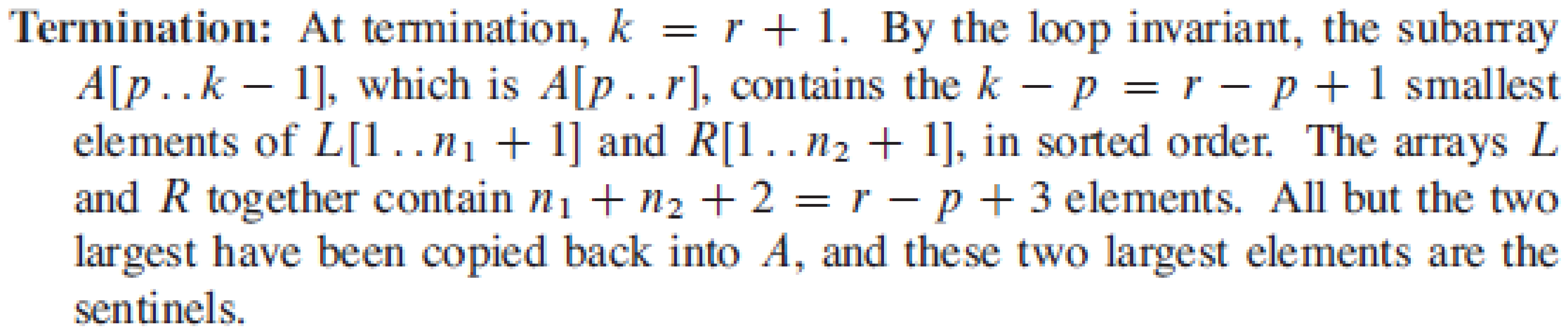
**Initialization:** Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \, . \, . \, k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of $L$ and $R$, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.



**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into $A$. Because $A[p \, . \, . \, k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \, . \, . \, k]$ will contain the $k - p + 1$ smallest elements. Incrementing $k$ (in the **for** loop update) and $i$ (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

*$n_1$ +1*

## Merge algorithm loop invariant



**Termination:** At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \mathinner{.\,.} k - 1]$, which is $A[p \mathinner{.\,.} r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \mathinner{.\,.} n_1 + 1]$ and $R[1 \mathinner{.\,.} n_2 + 1]$, in sorted order. The arrays $L$ and $R$ together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into $A$, and these two largest elements are the sentinels.

*$n_1$ +1*

# Merge algorithm in textbook

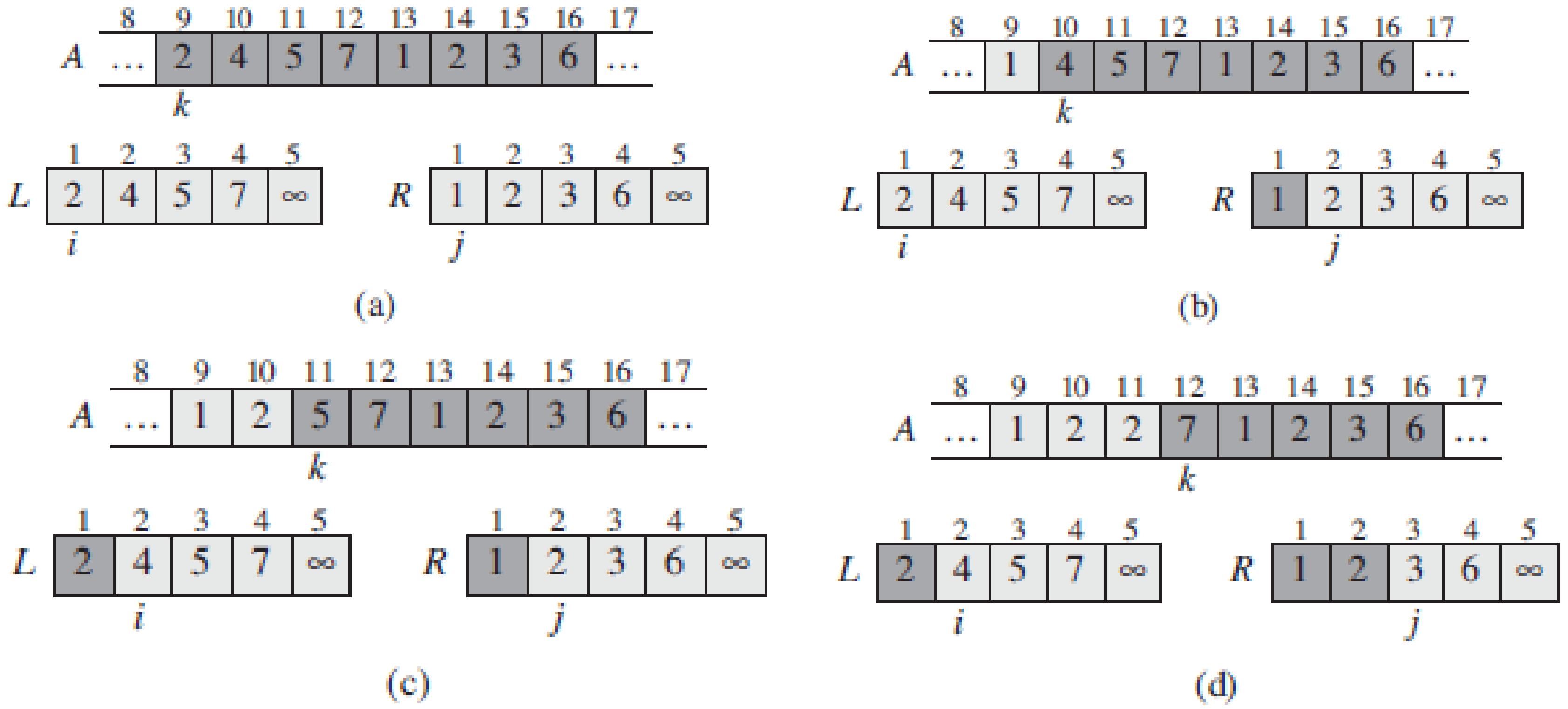
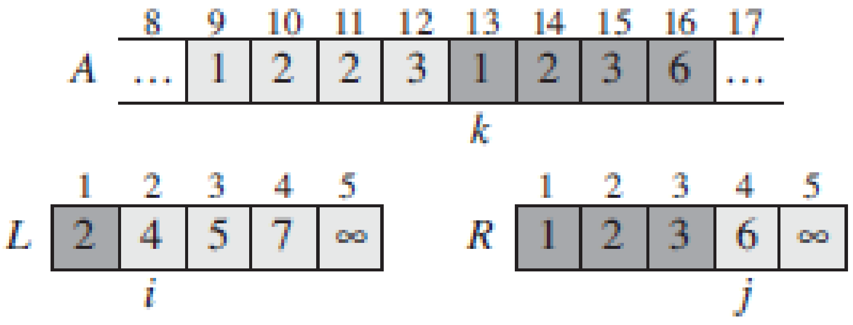
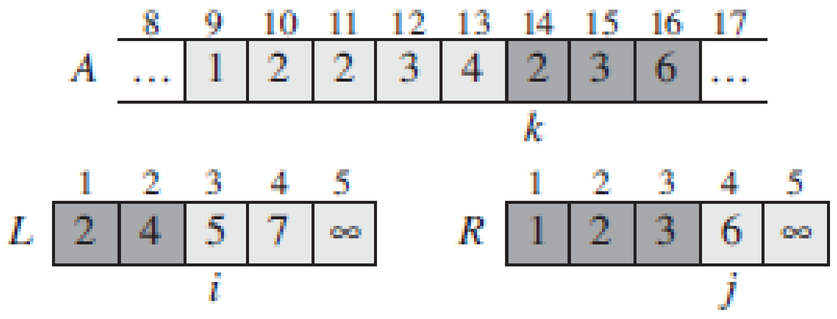


**Figure 2.3** The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array $L$ contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array $R$ contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in $A$ contain their final values, and lightly shaded positions in $L$ and $R$ contain values that have yet to be copied back into $A$. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in $A$ contain values that will be copied over, and heavily shaded positions in $L$ and $R$ contain values that have already been copied back into $A$. **(a)–(h)** The arrays $A$, $L$, and $R$, and their respective indices $k$, $i$, and $j$ prior to each iteration of the loop of lines 12–17.

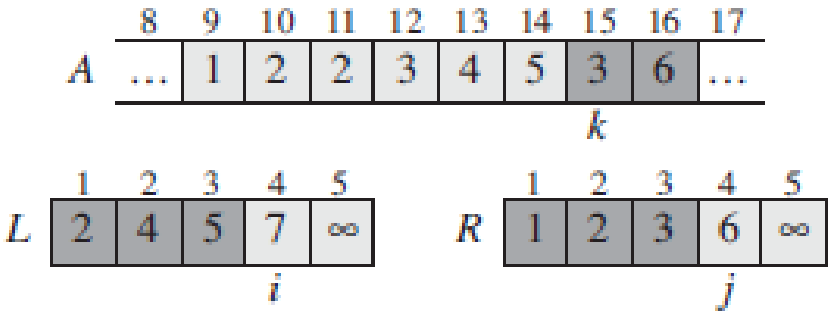
# Merge algorithm in textbook



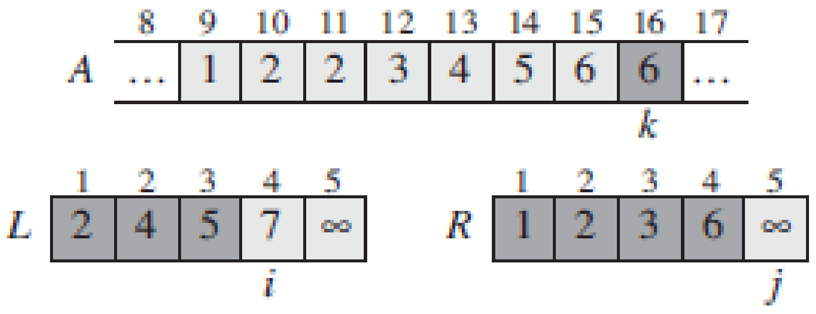
(e)



(f)

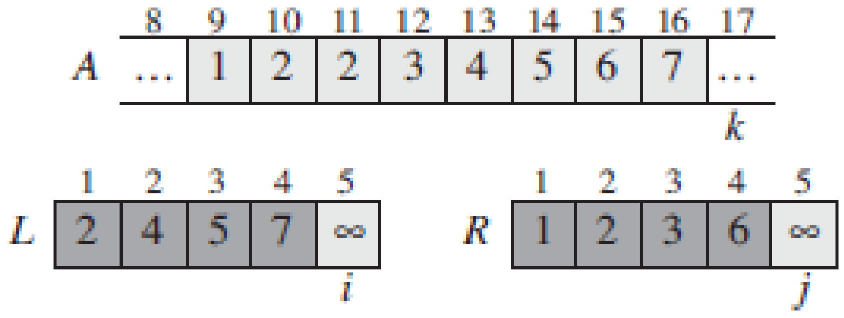


(g)



(h)

Running time?



(i)

MERGE procedure runs Ɵ(n) time
where n = r – p + 1

# Merge Sort algorithm in textbook

```
MERGE-SORT(A, p, r)        // initial call MERGE-SORT(A, 1, A.length)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

# Merge Sort algorithm in textbook

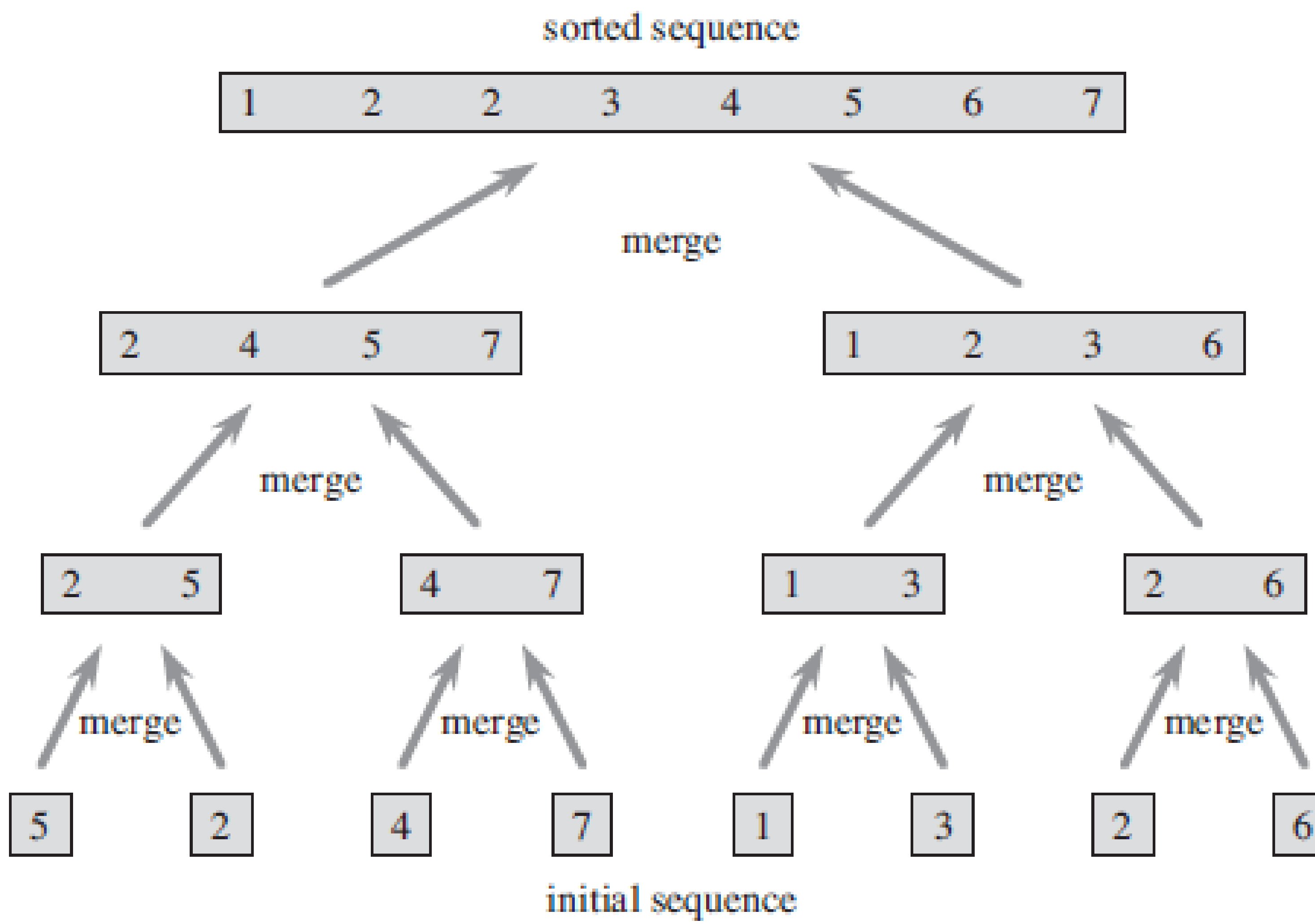


**Figure 2.4** The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

# MergeSort

```
MERGE-SORT($A$)
if $length[A] == 1$
        return A
else
        $q \leftarrow \lfloor length[A]/2 \rfloor$
        create arrays $L[1..q]$ and $R[q+1..length[A]]$
        copy $A[1..q]$ to $L$
        copy $A[q+1..length[A]]$ to $R$
        $LS \leftarrow$ MERGE-SORT(L)
        $RS \leftarrow$ MERGE-SORT(R)
        return MERGE(LS, RS)
```

## MergeSort: Merge

Assuming L and R are sorted already, merge the two to create a single sorted array

```
MERGE($L, R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

## Merge

L: 1 3 5 8 R: 2 4 6 7

```
MERGE(L, R)
 1  create array B of length $length[L] + length[R]$
 2  $i \leftarrow 1$
 3  $j \leftarrow 1$
 4  for $k \leftarrow 1$ to $length[B]$
 5          if $j > length[R]$ or $(i \leq length[L]$ and $L[i] \leq R[j])$
 6                  $B[k] \leftarrow L[i]$
 7                  $i \leftarrow i + 1$
 8          else
 9                  $B[k] \leftarrow R[j]$
10                  $j \leftarrow j + 1$
11  return B
```

# Merge

L: 1 3 5 8 R: 2 4 6 7

B:

```
MERGE($L, R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

# Merge

L: 1 3 5 8

R: 2 4 6 7

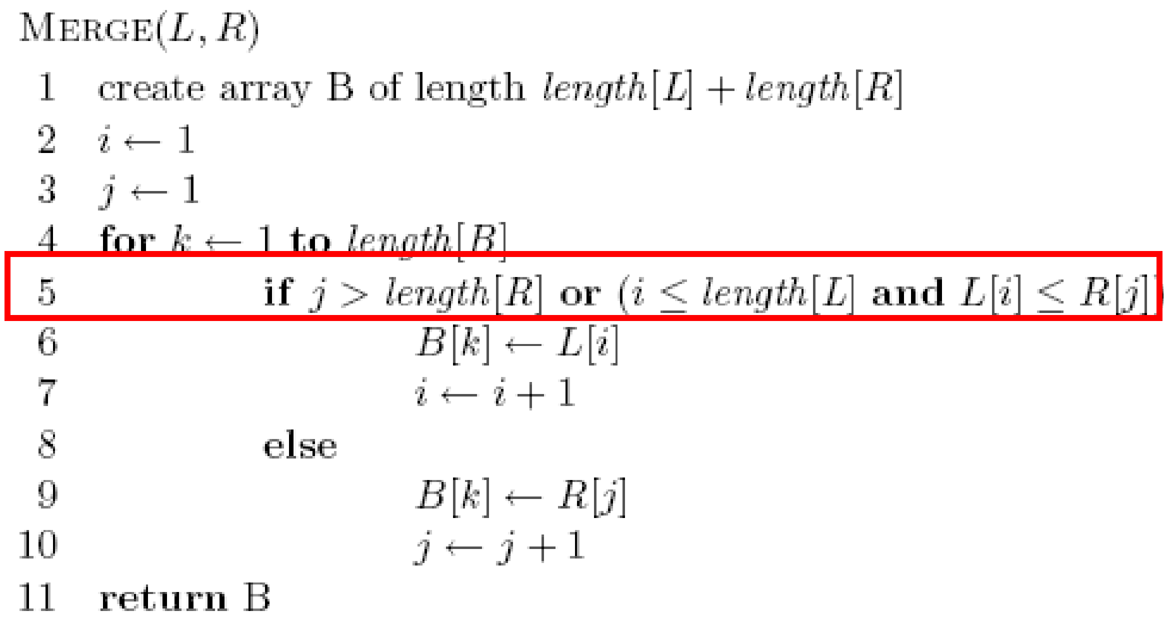
B:

```
MERGE($L, R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or $(i \leq length[L]$ and $L[i] \leq R[j])$
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

# Merge

L: $\overset{\downarrow}{1}$ 3 5 8　　　R: $\overset{\downarrow}{2}$ 4 6 7

B:



```
MERGE(L, R)
 1  create array B of length $length[L] + length[R]$
 2  $i \leftarrow 1$
 3  $j \leftarrow 1$
 4  for $k \leftarrow 1$ to $length[B]$
 5          if $j > length[R]$ or $(i \leq length[L]$ and $L[i] \leq R[j])$
 6                  $B[k] \leftarrow L[i]$
 7                  $i \leftarrow i + 1$
 8          else
 9                  $B[k] \leftarrow R[j]$
10                  $j \leftarrow j + 1$
11  return B
```

# Merge

L: 1 $\overset{\downarrow}{3}$ 5 8

R: $\overset{\downarrow}{2}$ 4 6 7

B: 1

```
MERGE($L, R$)
1  create array B of length $length[L] + length[R]$
2  $i \leftarrow 1$
3  $j \leftarrow 1$
4  for $k \leftarrow 1$ to $length[B]$
5          if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
6                  $B[k] \leftarrow L[i]$
7                  $i \leftarrow i + 1$
8          else
9                  $B[k] \leftarrow R[j]$
10                 $j \leftarrow j + 1$
11 return B
```

# Merge

L: 1 $\overset{\downarrow}{3}$ 5 8 R: $\overset{\downarrow}{2}$ 4 6 7

B: 1

```
Merge($L, R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

# Merge

L: 1 3 5 8        R: 2 4 6 7

B: 1 2

```
MERGE($L, R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

# Merge

L: 1 $\overset{\downarrow}{3}$ 5 8 R: 2 $\overset{\downarrow}{4}$ 6 7

B: 1 2

```
MERGE($L, R$)
    create array B of length $length[L] + length[R]$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
            $B[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else
            $B[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
    return B
```

# Merge

L: 1 3 5 8　　　R: 2 4 6 7

B: 1 2 3

```
MERGE($L, R$)
    create array B of length $length[L] + length[R]$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
            $B[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else
            $B[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
    return B
```

# Merge

```
        ↓              ↓
L: 1  3  5  8     R: 2  4  6  7
```

B: 1 2 3

```
MERGE($L$, $R$)
1   create array B of length $length[L] + length[R]$
2   $i \leftarrow 1$
3   $j \leftarrow 1$
4   for $k \leftarrow 1$ to $length[B]$
5           if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
6                   $B[k] \leftarrow L[i]$
7                   $i \leftarrow i + 1$
8           else
9                   $B[k] \leftarrow R[j]$
10                  $j \leftarrow j + 1$
11  return B
```

# Merge

L: 1 3 5 8    R: 2 4 6 7

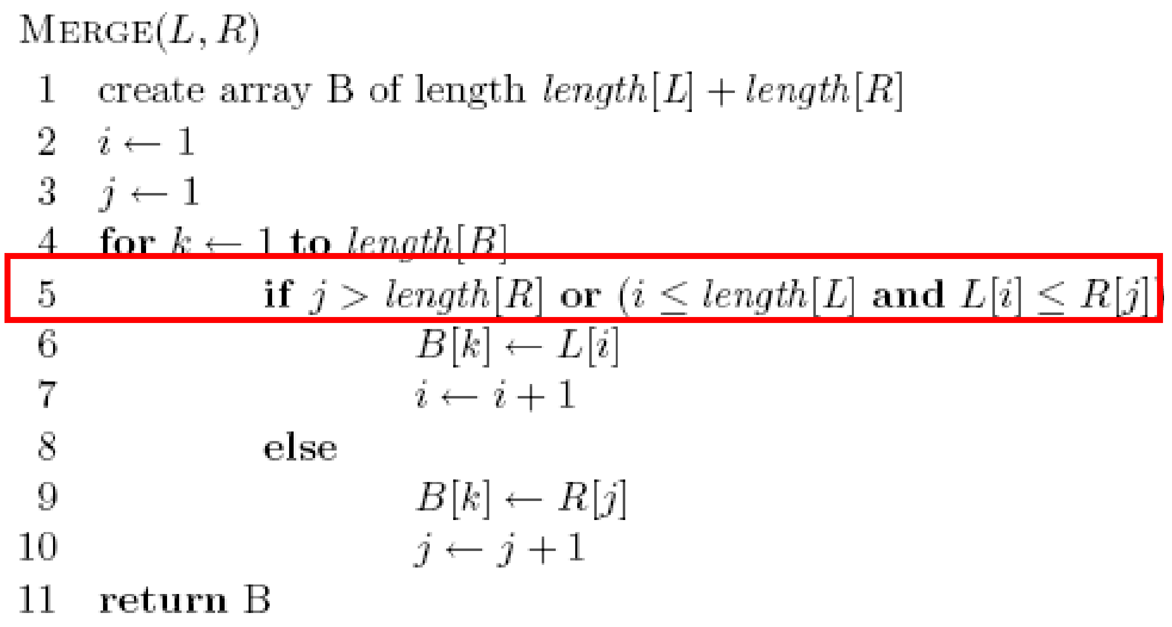
B: 1 2 3 4

```
MERGE($L, R$)
    create array B of length $length[L] + length[R]$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
            $B[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else
            $B[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
    return B
```

# Merge

L: 1 3 5 8          R: 2 4 6 7

B: 1 2 3 4



```
MERGE($L, R$)
  create array B of length $length[L] + length[R]$
  $i \leftarrow 1$
  $j \leftarrow 1$
  for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
              $B[k] \leftarrow L[i]$
              $i \leftarrow i + 1$
        else
              $B[k] \leftarrow R[j]$
              $j \leftarrow j + 1$
  return B
```

# Merge

L: 1 3 5 8 R: 2 4 6 7

B: 1 2 3 4 5

```
MERGE($L, R$)
1  create array B of length $length[L] + length[R]$
2  $i \leftarrow 1$
3  $j \leftarrow 1$
4  for $k \leftarrow 1$ to $length[B]$
5          if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
6                  $B[k] \leftarrow L[i]$
7                  $i \leftarrow i + 1$
8          else
9                  $B[k] \leftarrow R[j]$
10                 $j \leftarrow j + 1$
11 return B
```

# Merge

L: 1 3 5 $\overset{\downarrow}{8}$        R: 2 4 $\overset{\downarrow}{6}$ 7

B: 1 2 3 4 5

```
MERGE($L, R$)
 1  create array B of length $length[L] + length[R]$
 2  $i \leftarrow 1$
 3  $j \leftarrow 1$
 4  for $k \leftarrow 1$ to $length[B]$
 5          if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
 6                  $B[k] \leftarrow L[i]$
 7                  $i \leftarrow i + 1$
 8          else
 9                  $B[k] \leftarrow R[j]$
10                  $j \leftarrow j + 1$
11  return B
```

# Merge

L: 1 3 5 8        R: 2 4 6 7

B: 1 2 3 4 5 6

```
MERGE($L$, $R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
    if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
        $B[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $B[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
return B
```

# Merge

L: 1 3 5 8  R: 2 4 6 7

B: 1 2 3 4 5 6

```
MERGE(L, R)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

# Merge

L: 1 3 5 8 R: 2 4 6 7

B: 1 2 3 4 5 6 7

```
MERGE($L$, $R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

# Merge

L: 1 3 5 8        R: 2 4 6 7

B: 1 2 3 4 5 6 7

```
MERGE$(L, R)$
  create array B of length $length[L] + length[R]$
  $i \leftarrow 1$
  $j \leftarrow 1$
  for $k \leftarrow 1$ to $length[B]$
          if $j > length[R]$ or $(i \leq length[L]$ and $L[i] \leq R[j])$
                  $B[k] \leftarrow L[i]$
                  $i \leftarrow i + 1$
          else
                  $B[k] \leftarrow R[j]$
                  $j \leftarrow j + 1$
  return B
```

# Merge

↓ ↓

L: 1 3 5 8        R: 2 4 6 7

B: 1 2 3 4 5 6 7 8

```
MERGE($L, R$)
1   create array B of length $length[L] + length[R]$
2   $i \leftarrow 1$
3   $j \leftarrow 1$
4   for $k \leftarrow 1$ to $length[B]$
5           if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
6                   $B[k] \leftarrow L[i]$
7                   $i \leftarrow i + 1$
8           else
9                   $B[k] \leftarrow R[j]$
10                  $j \leftarrow j + 1$
11  return B
```

## Merge

Does the algorithm terminate?

```
MERGE($L, R$)
create array B of length $length[L] + length[R]$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow 1$ to $length[B]$
        if $j > length[R]$ or ($i \leq length[L]$ and $L[i] \leq R[j]$)
                $B[k] \leftarrow L[i]$
                $i \leftarrow i + 1$
        else
                $B[k] \leftarrow R[j]$
                $j \leftarrow j + 1$
return B
```

## Is it correct?

Loop invariant: At the beginning of the **for** loop of lines 4-10 the first k-1 elements of B are the smallest *k-1* elements from L and R in sorted order.

```
MERGE($L, R$)
1   create array B of length $length[L] + length[R]$
2   $i \leftarrow 1$
3   $j \leftarrow 1$
4   for $k \leftarrow 1$ to $length[B]$
5           if $j > length[R]$ or $(i \leq length[L]$ and $L[i] \leq R[j])$
6                   $B[k] \leftarrow L[i]$
7                   $i \leftarrow i + 1$
8           else
9                   $B[k] \leftarrow R[j]$
10                  $j \leftarrow j + 1$
11  return B
```

# Analyze D-and-C algorithms

Running time?

- three steps of the basic algorithm
- Let T(n) be the running time on a problem of size n
- if problem size is too small, n ≤ c, solution takes constant time, ɵ(1)
- If yields *a* subproblems, 1/b size of the original problem

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

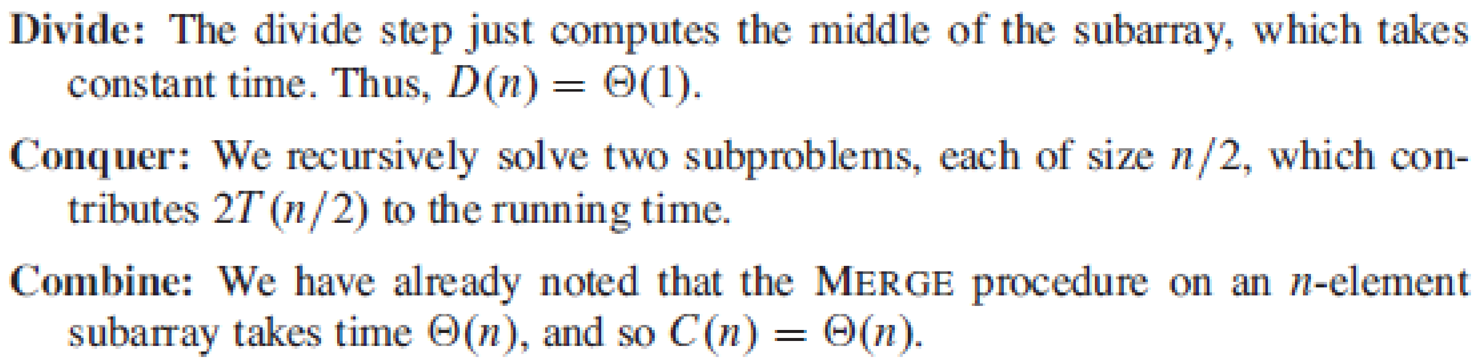
*D(n)*: cost of splitting (dividing) the data

*C(n)*: cost of merging/combining the data

# Merge-Sort

Running time?

- each divide step then yields two subsequences of size exactly *n*/2



**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

**Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

**Combine:** We have already noted that the MERGE procedure on an $n$-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + D(n) + C(n) & \text{otherwise} \end{cases}$$

*D(n)*: cost of splitting (dividing) the data - linear Θ(n)

*C(n)*: cost of merging/combining the data - linear Θ(n)

# Merge-Sort

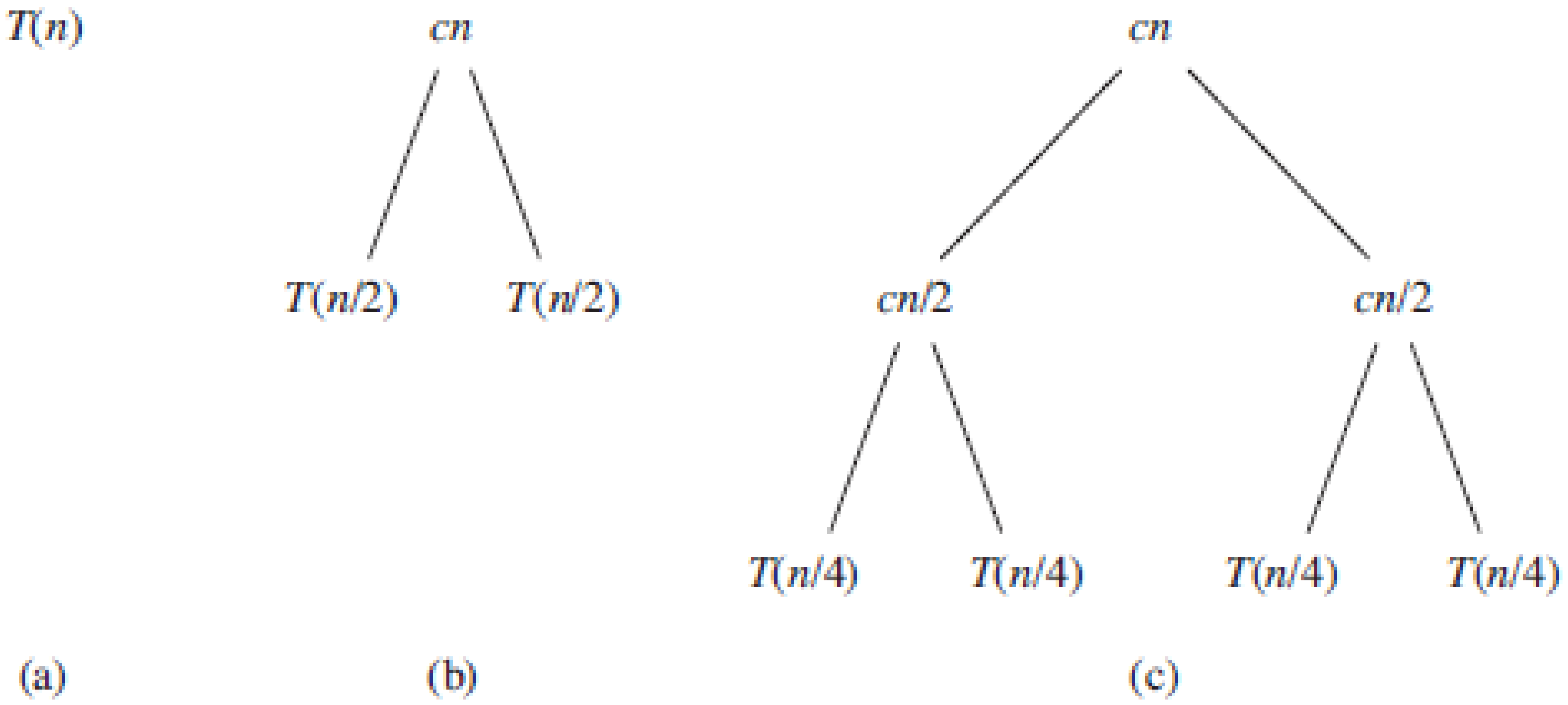
When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of $n$, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

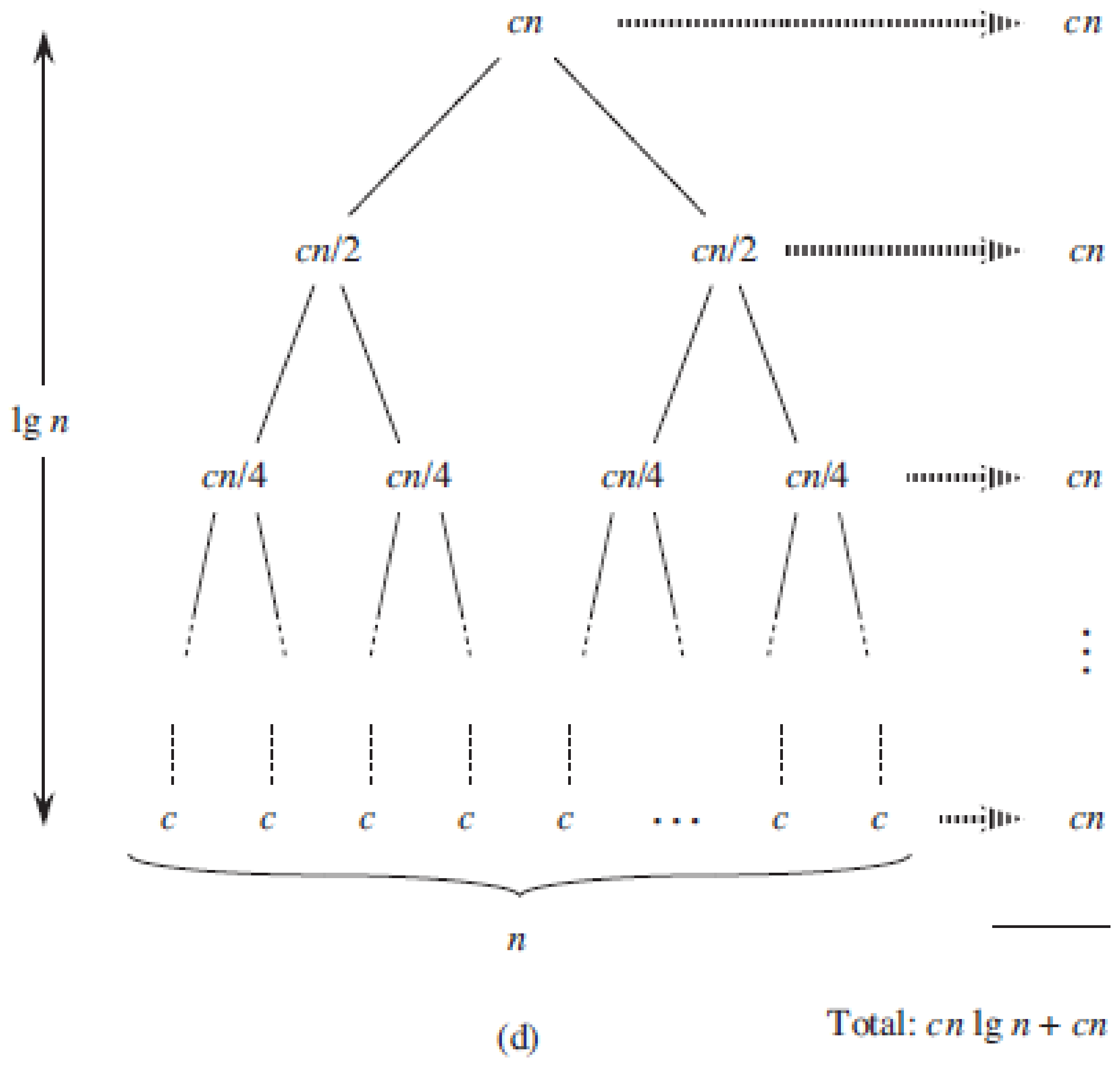
that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$

# Merge-Sort



# Merge-Sort



$cn$
$cn$
$cn/2$
$cn/2$
$cn$
$\lg n$
$cn/4$
$cn/4$
$cn/4$
$cn/4$
$cn$
$c$ $c$ $c$ $c$ $c$ $\cdots$ $c$ $c$
$cn$
$n$
(d)
Total: $cn \lg n + cn$

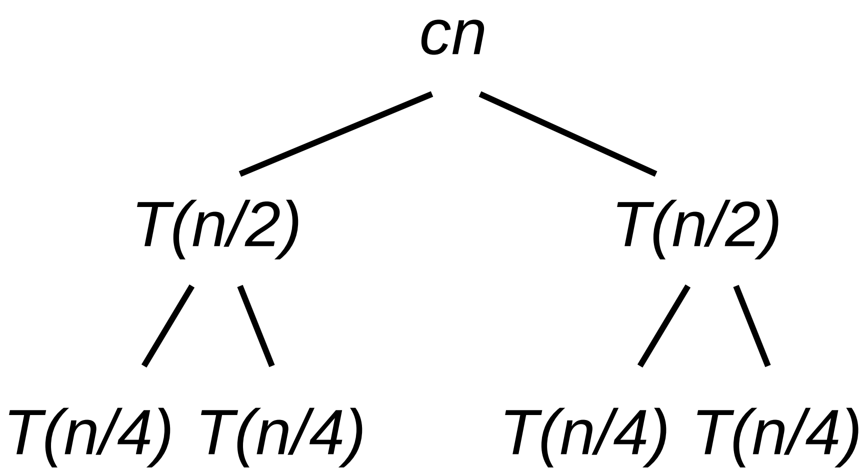
# Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

$$\begin{tikzcd}
 & cn \arrow[dl, dash] \arrow[dr, dash] & \\
T(n/2) & & T(n/2)
\end{tikzcd}$$

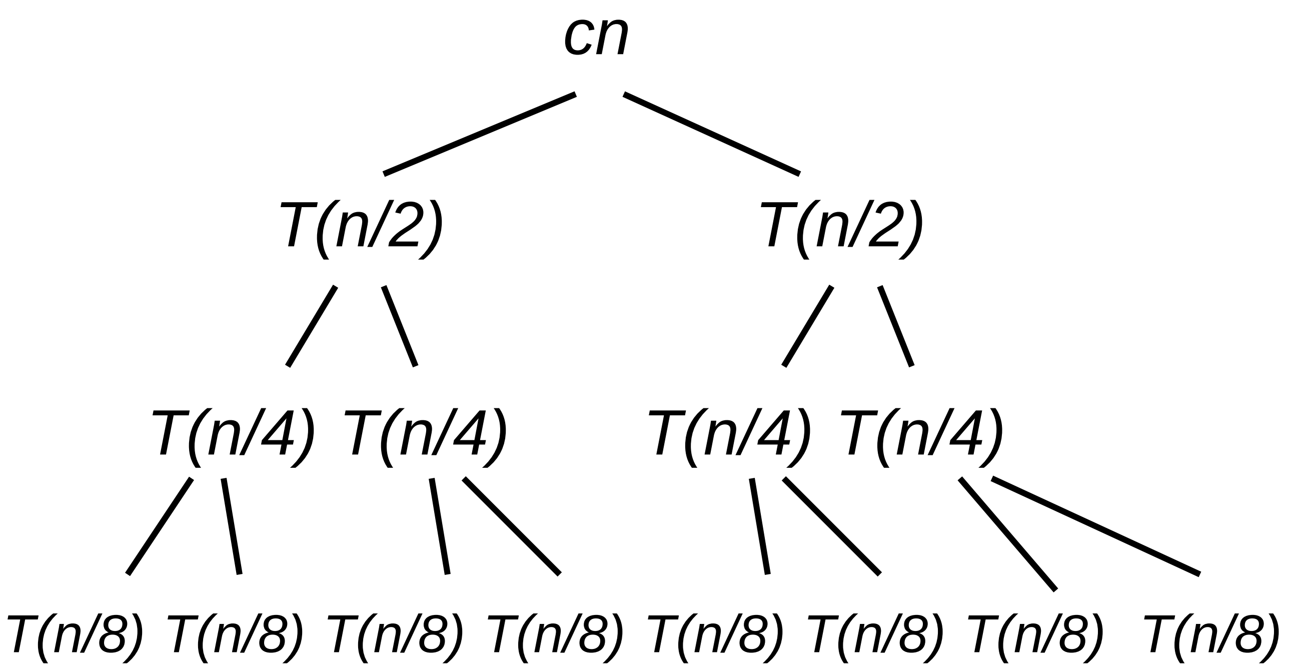
# Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



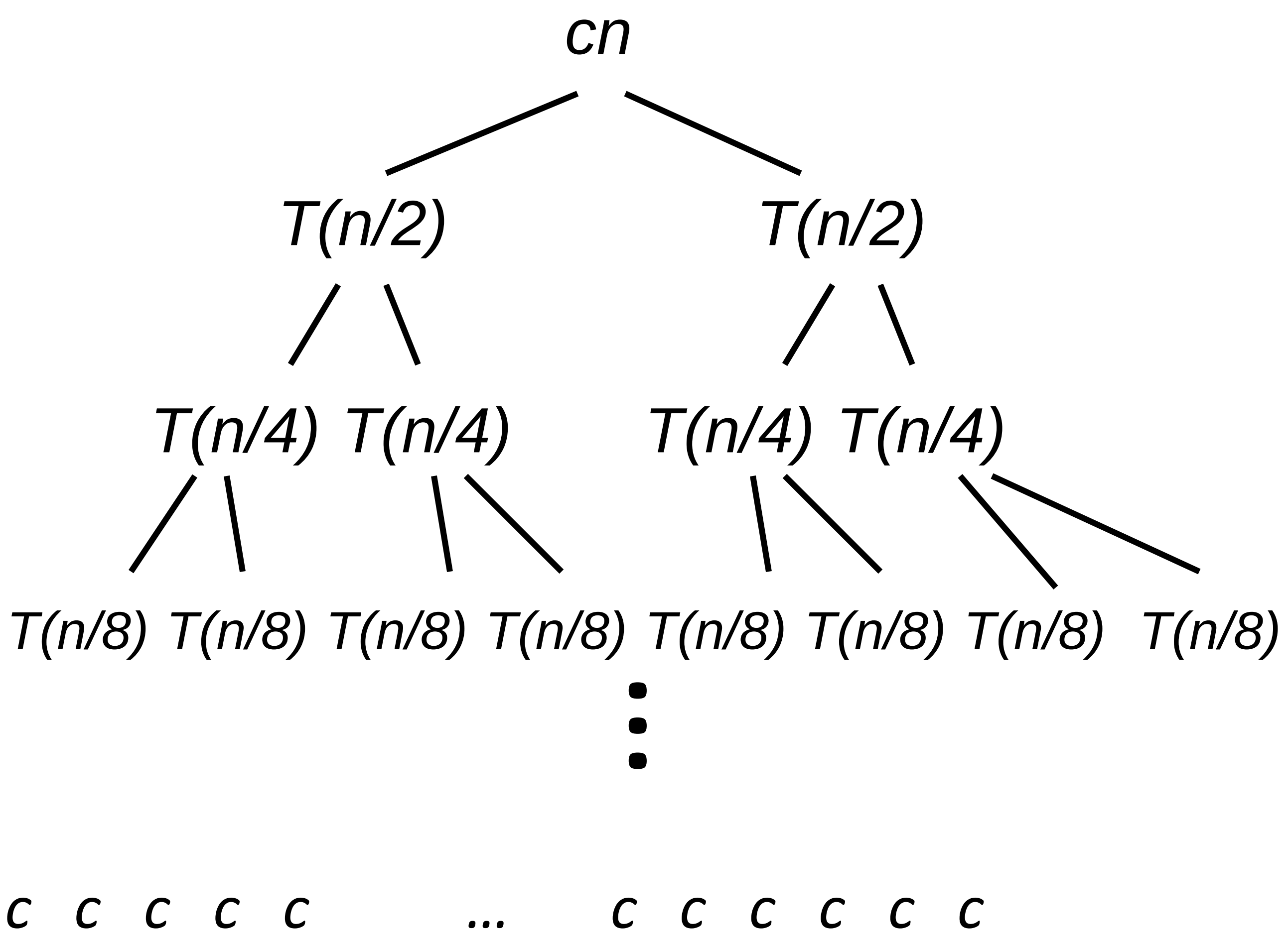
# Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



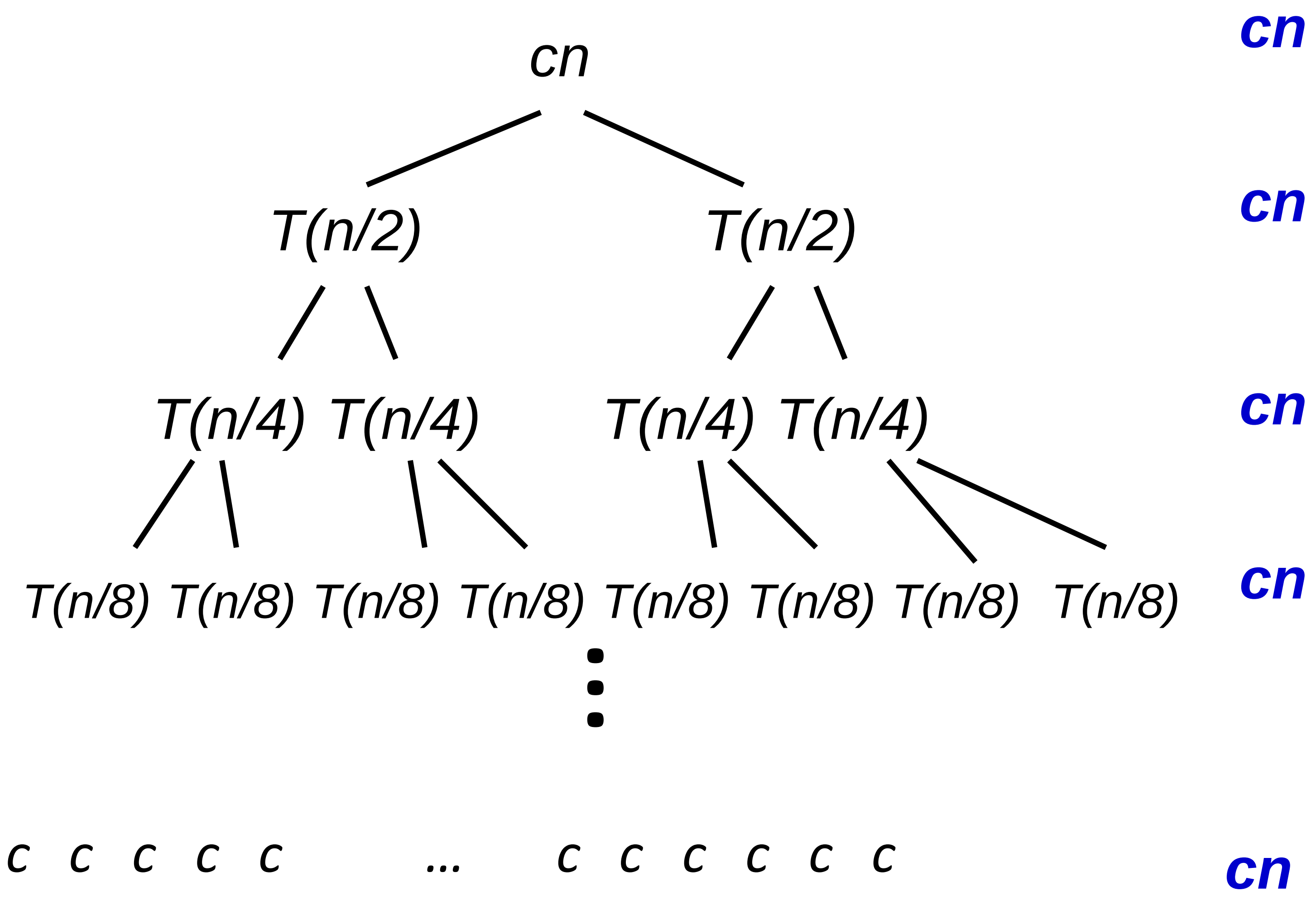
# Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



# Merge-Sort

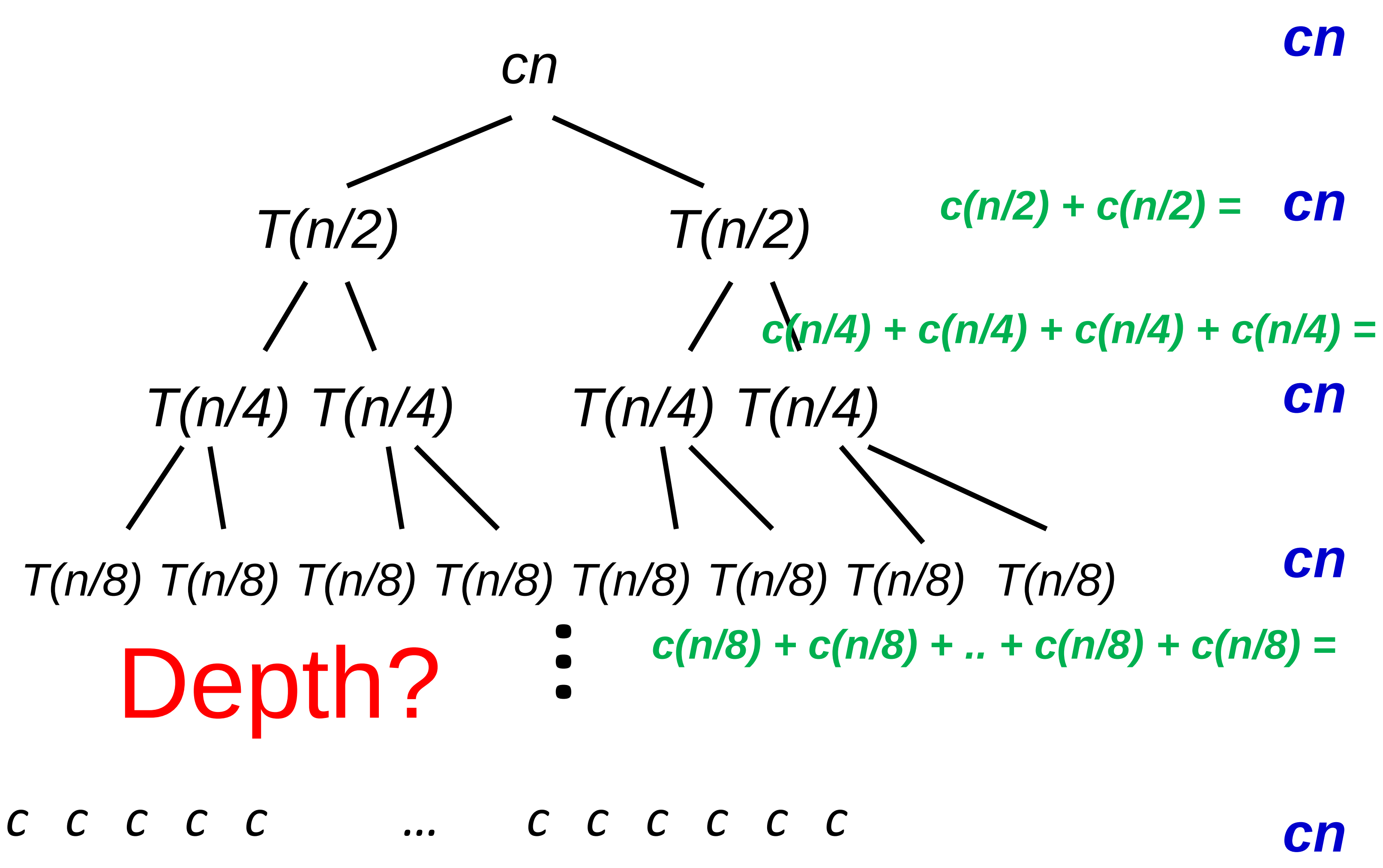
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



cn
T(n/2) T(n/2)
T(n/4) T(n/4) T(n/4) T(n/4)
T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8)
⋮
c c c c c … c c c c c c
cn
cn
cn
cn
cn

# Merge-Sort

$$T(n)=\begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2)+cn & \text{otherwise} \end{cases}$$



Depth?

## Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

We can calculate the depth, by determining when the recursion gets to down to a small problem size, e.g. 1

At each level, we divide by 2

$$\frac{n}{2^d} = 1$$

$$2^d = n$$

$$\log 2^d = \log n$$

$$d \log 2 = \log n$$

$$d = \log_2 n$$

# Merge Sort Algorithm

***mergeSort***

```
Cut the array in half
Sort the left half
Sort the right half
Merge the two sorted halves into one sorted array
```

Because `mergeSort` is itself a sorting algorithm, we might as well use it to sort the two halves.

We can make `mergeSort` a recursive method and let it call itself to sort each of the two subarrays:

***mergeSort—Recursive***

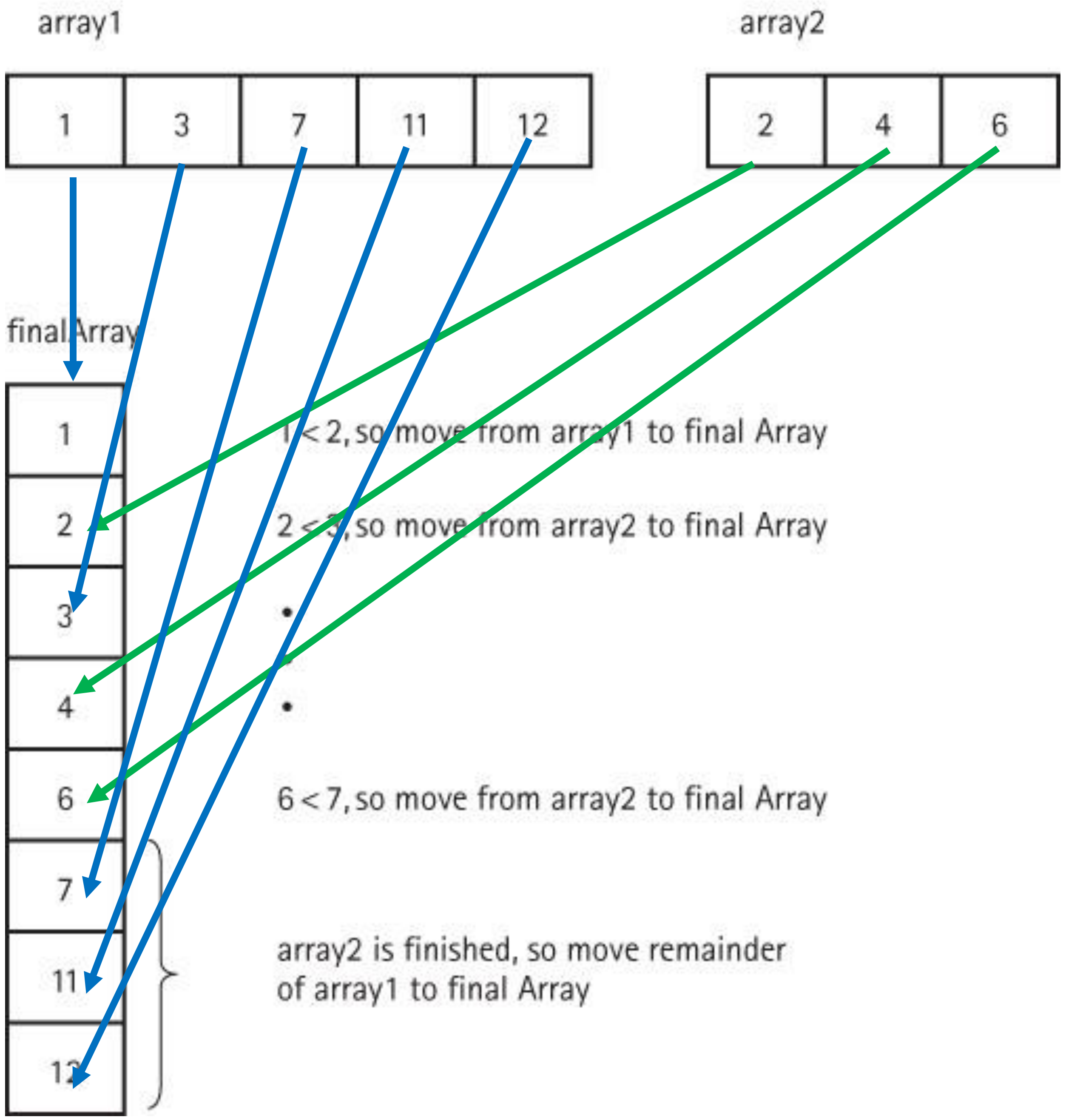
```
Cut the array in half
mergeSort the left half
mergeSort the right half
Merge the two sorted halves into one sorted array
```

## Merge Sort Summary

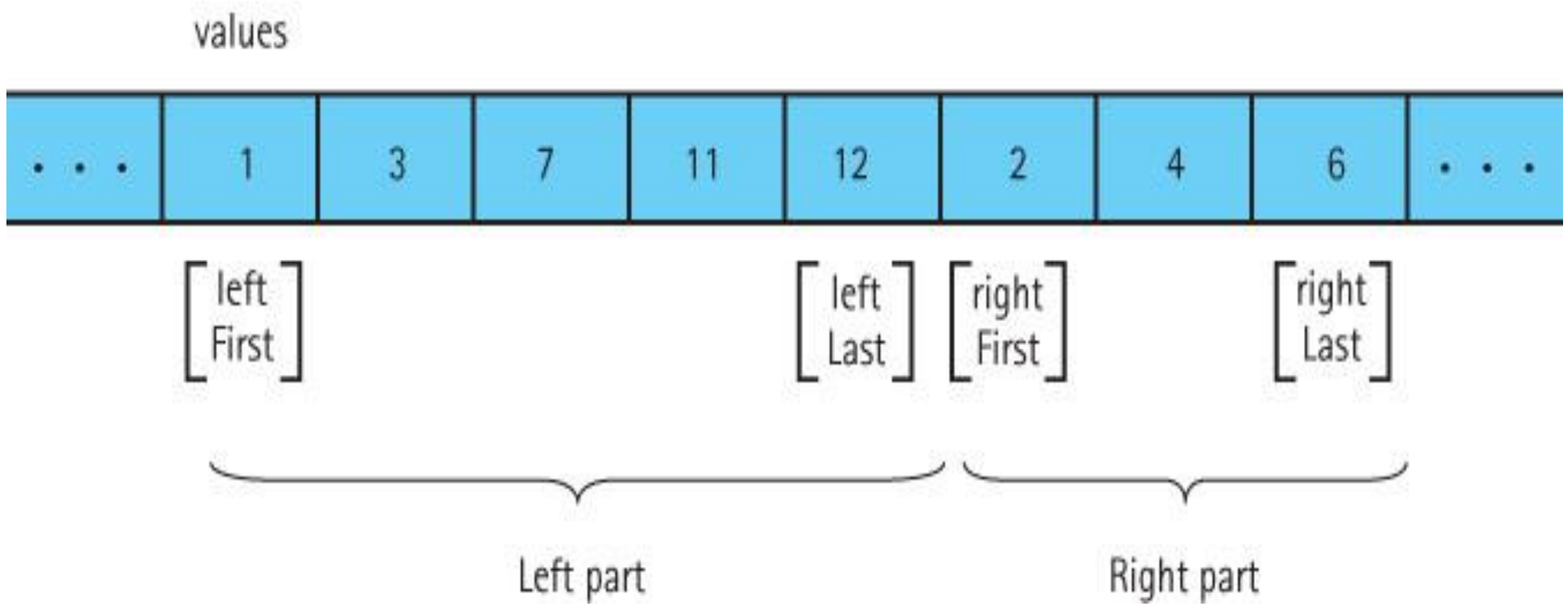
**Method mergeSort(first, last)**

| | |
|---|---|
| *Definition:* | Sorts the array elements in ascending order. |
| *Size:* | $\underset{\text{index}}{\text{last}} - \underset{\text{index}}{\text{first}} + 1$ |
| *Base Case:* | If size less than 2, do nothing. |
| *General Case:* | Cut the array in half. |
| | mergeSort the left half. |
| | mergeSort the right half. |
| | Merge the sorted halves into one sorted array. |

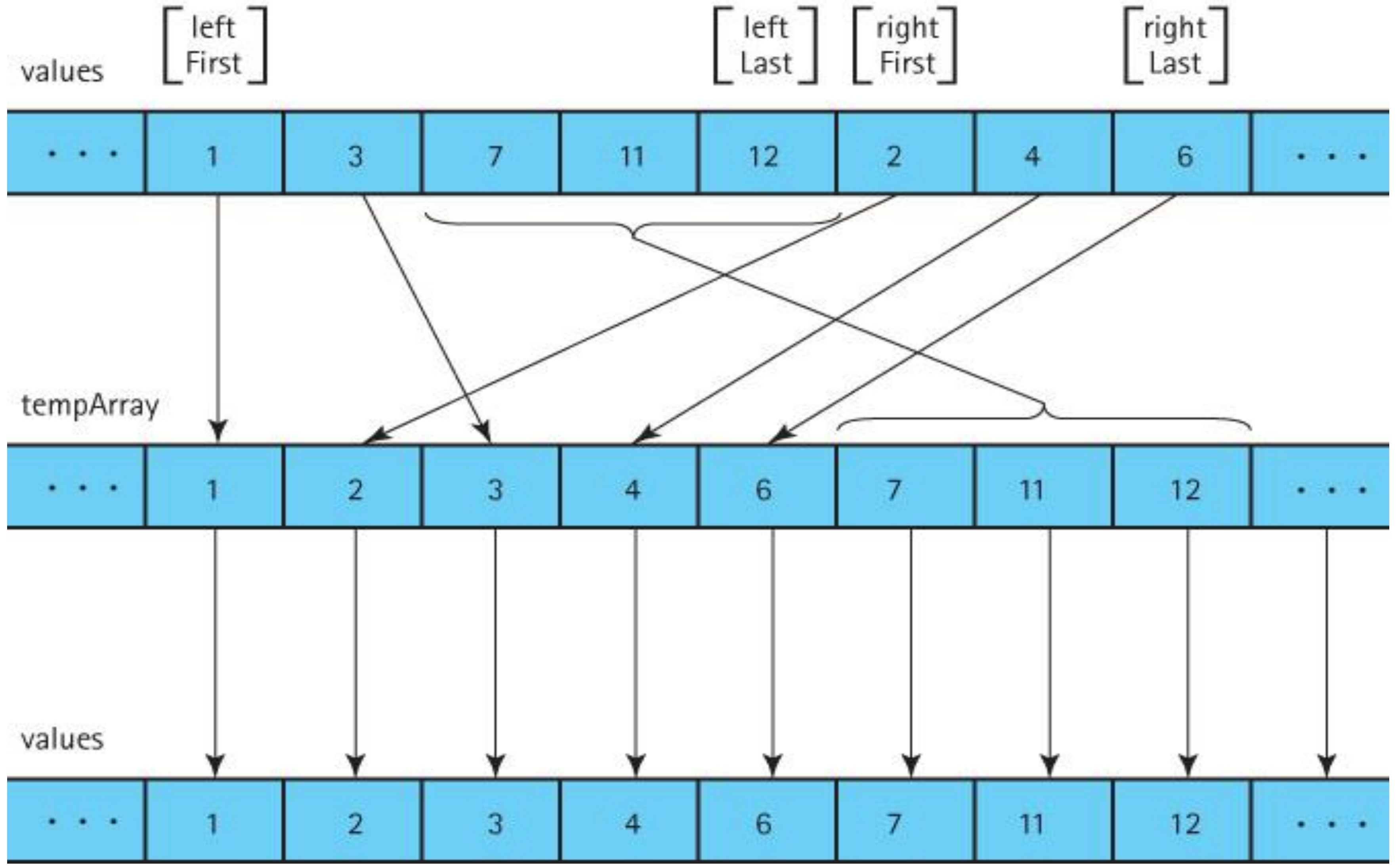
## Strategy for merging two sorted arrays


array1
1
3
7
11
12
array2
2
4
6
finalArray
1
2
3
4
6
7
11
12
1 < 2, so move from array1 to final Array
2 < 3, so move from array2 to final Array
•
•
•
6 < 7, so move from array2 to final Array
array2 is finished, so move remainder of array1 to final Array

# Our actual merge problem



# Our solution



## The `merge` algorithm

```
merge (leftFirst, leftLast, rightFirst, rightLast)

(uses a local array, tempArray)

Set index to leftFirst
while more elements in left half AND more elements in right half
    if values[leftFirst] < values[rightFirst]
        Set tempArray[index] to values[leftFirst]
        Increment leftFirst
    else
        Set tempArray[index] to values[rightFirst]
        Increment rightFirst
    Increment index
Copy any remaining elements from left half to tempArray
Copy any remaining elements from right half to tempArray
Copy the sorted elements from tempArray back into values
```

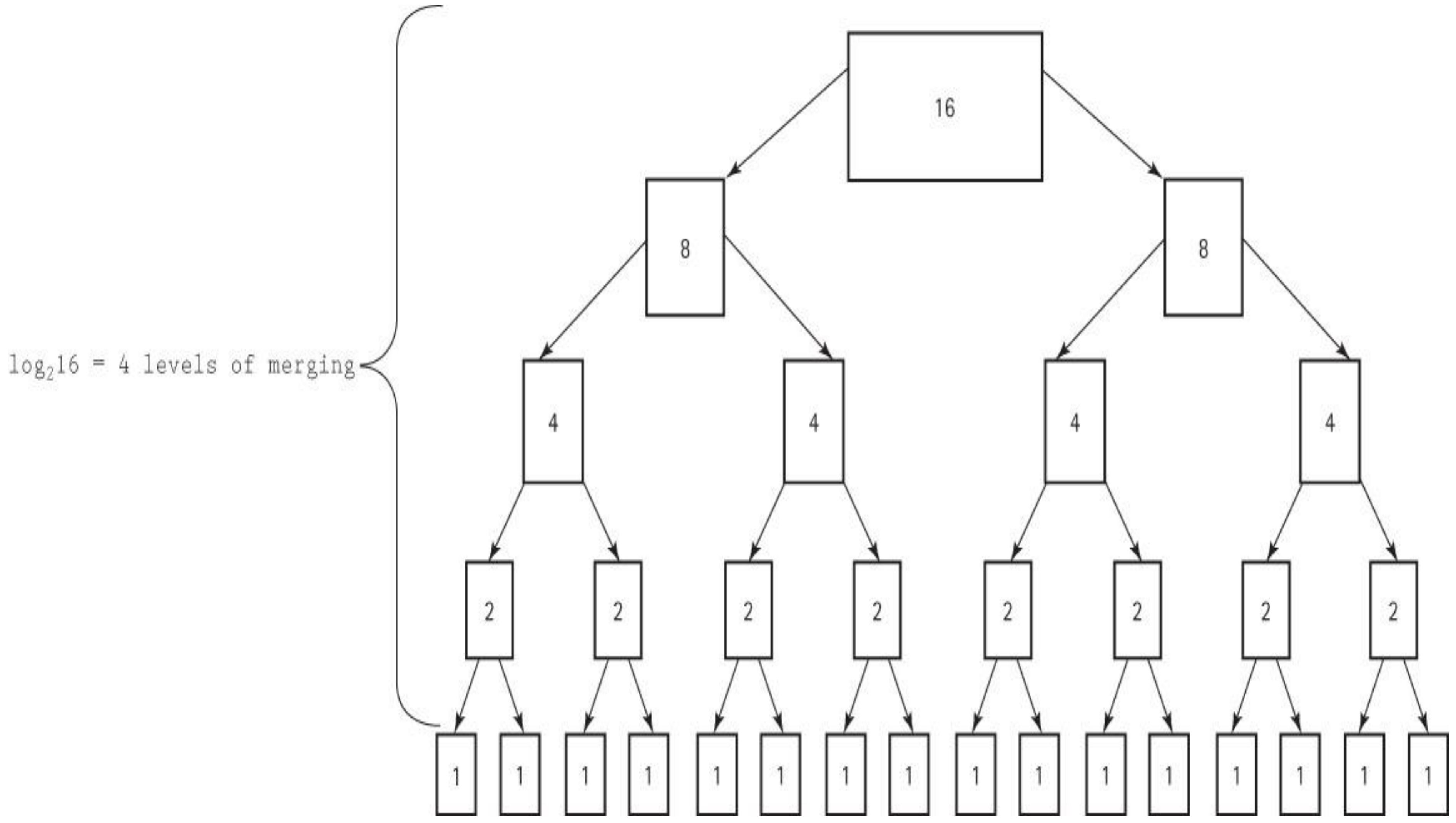
## The `mergeSort` method

The code for `merge` follows the algorithm on the previous slide.
`merge` does most of the work!

Here is `mergeSort`:

```
static void mergeSort(int first, int last)
// Sorts the values array using the merge sort algorithm.
{
  if (first < last)
  {
    int middle = (first + last) / 2;
    mergeSort(first, middle);
    mergeSort(middle + 1, last);
    merge(first, middle, middle + 1, last);
  }
}
```

# Analysing Merge Sort



## Comparing $N^2$ and N $\log_2 N$

| **N** | **$\log_2 N$** | **$N^2$** | **N $\log_2 N$** |
|---|---|---|---|
| 32 | 5 | 1,024 | 160 |
| 64 | 6 | 4.096 | 384 |
| 128 | 7 | 16,384 896 | |
| 256 | 8 | 65,536 2,048 | |
| 512 | 9 | 262,144 | 4,608 |
| 1024 | 10 | 1,048,576 | 10,240 |
| 2048 | 11 | 4,194,304 | 22,528 |
| 4096 | 12 | 16,777,216 | 49,152 |

# Drawback of Merge Sort

- A disadvantage of `mergeSort` is that it requires an auxiliary array that is as large as the original array to be sorted.
- If the array is large and space is a critical factor, this sort may not be an appropriate choice.

# Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Running time?

- Each level costs *cn*
- log *n* levels

$cn \log n = \Theta(n \log n)$

# Recurrence

A function that is defined with respect to itself on smaller inputs

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

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

$$T(n) = 2T(n-1) + n^2$$