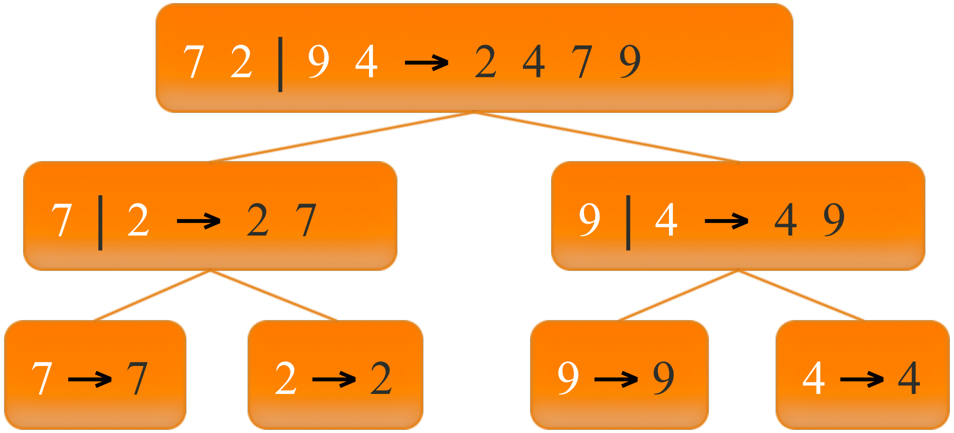
Fall 2019

Fang Yu

Software Security Lab.
Dept. Management Information Systems,
National Chengchi University

Data Structures
Lecture 9



7 2 | 9 4 → 2 4 7 9

7 | 2 → 2 7

9 | 4 → 4 9

7 → 7

2 → 2

9 → 9

4 → 4

Fundamental Algorithms

Divide and Conquer: Merge-sort, Quick-sort, and Recurrence Analysis

Divide-and-Conquer

A general algorithm design paradigm

- **Divide**: divide the input data $\boldsymbol{S}$ in two or more disjoint subsets $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …
- Recursion: solve the sub problems recursively
- **Conquer**: combine the solutions for $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …, into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of a constant size
- Analysis can be done using recurrence equations

Merge-sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-sort

Merge-sort on an input sequence ***S*** with ***n*** elements consists of three steps:

- Divide: partition ***S*** into two sequences S_1 and S_2 of about $n/2$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S, C)
    Input sequence S with n
        elements, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1
        (S_1, S_2) ← partition(S, n/2)
        mergeSort(S_1, C)
        mergeSort(S_2, C)
        S ← merge(S_1, S_2)
```

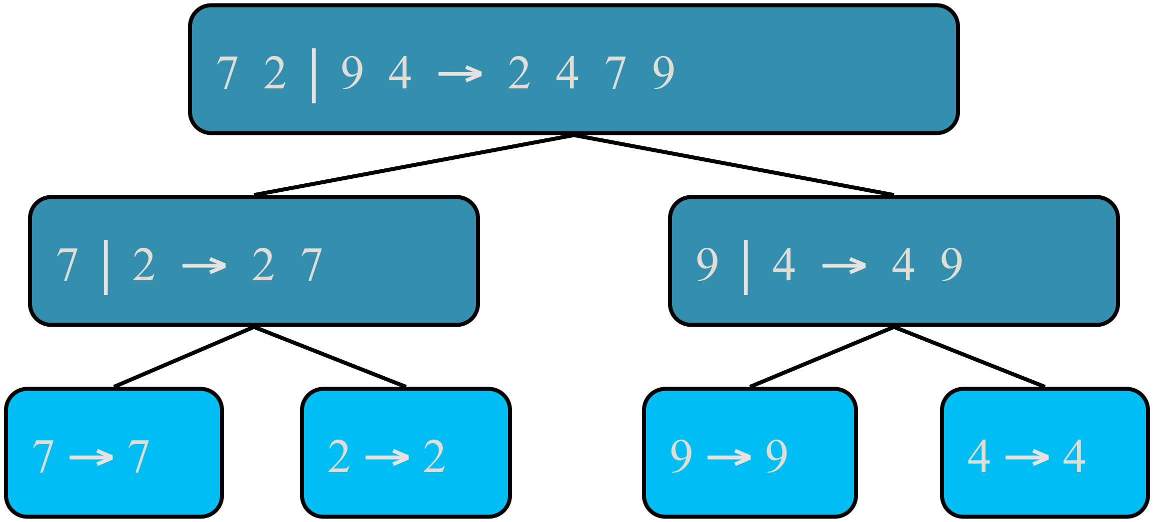
Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***
- Merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes ***O***(***n***) time

```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of $A \cup B$

    $S \leftarrow$ empty sequence
    while $\neg$A.isEmpty() $\wedge$ $\neg$B.isEmpty()
        if A.first().element() < B.first().element()
            S.addLast(A.remove(A.first()))
        else
            S.addLast(B.remove(B.first()))
    while $\neg$A.isEmpty()
        S.addLast(A.remove(A.first()))
    while $\neg$B.isEmpty()
        S.addLast(B.remove(B.first()))
    return S
```

Merge-Sort Tree



7 2 | 9 4 → 2 4 7 9

7 | 2 → 2 7

9 | 4 → 4 9

7 → 7

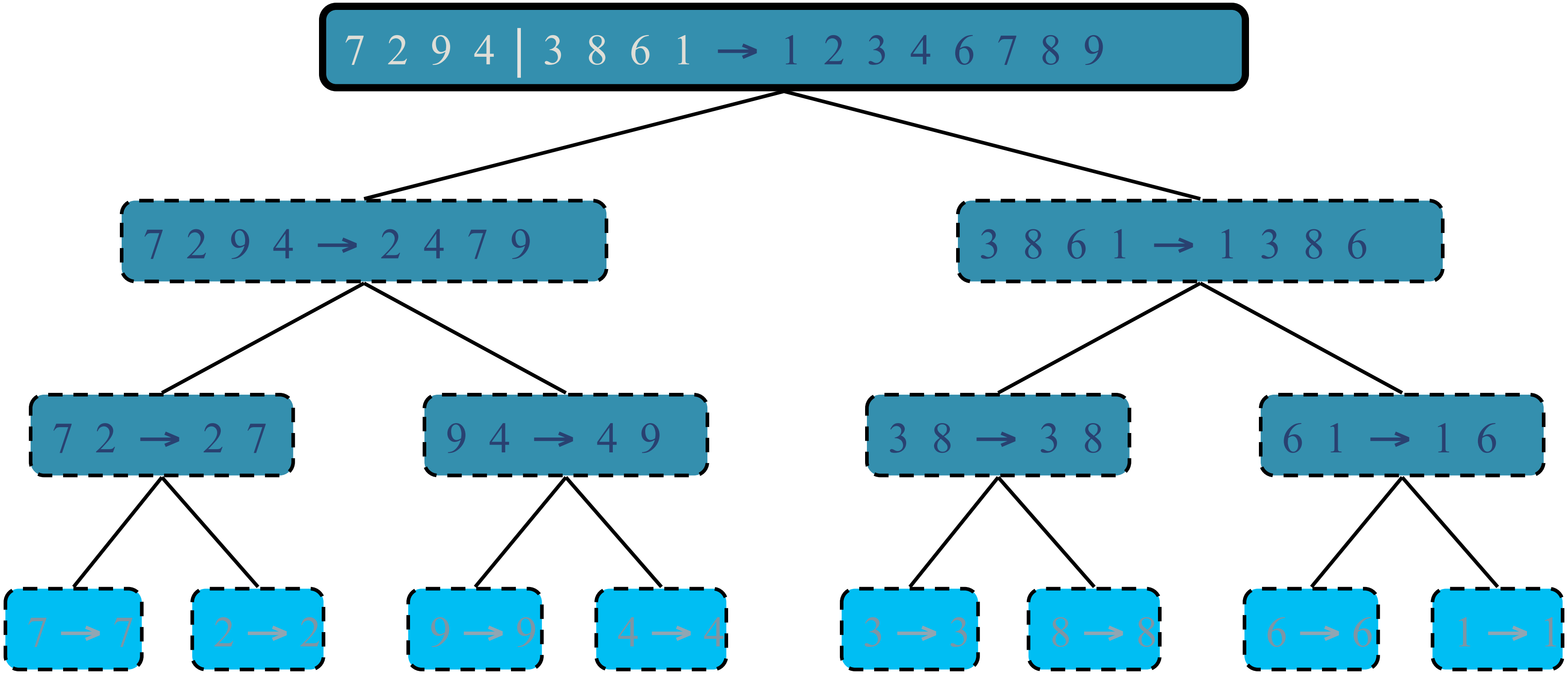
2 → 2

9 → 9

4 → 4

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1

An execution example



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

7 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

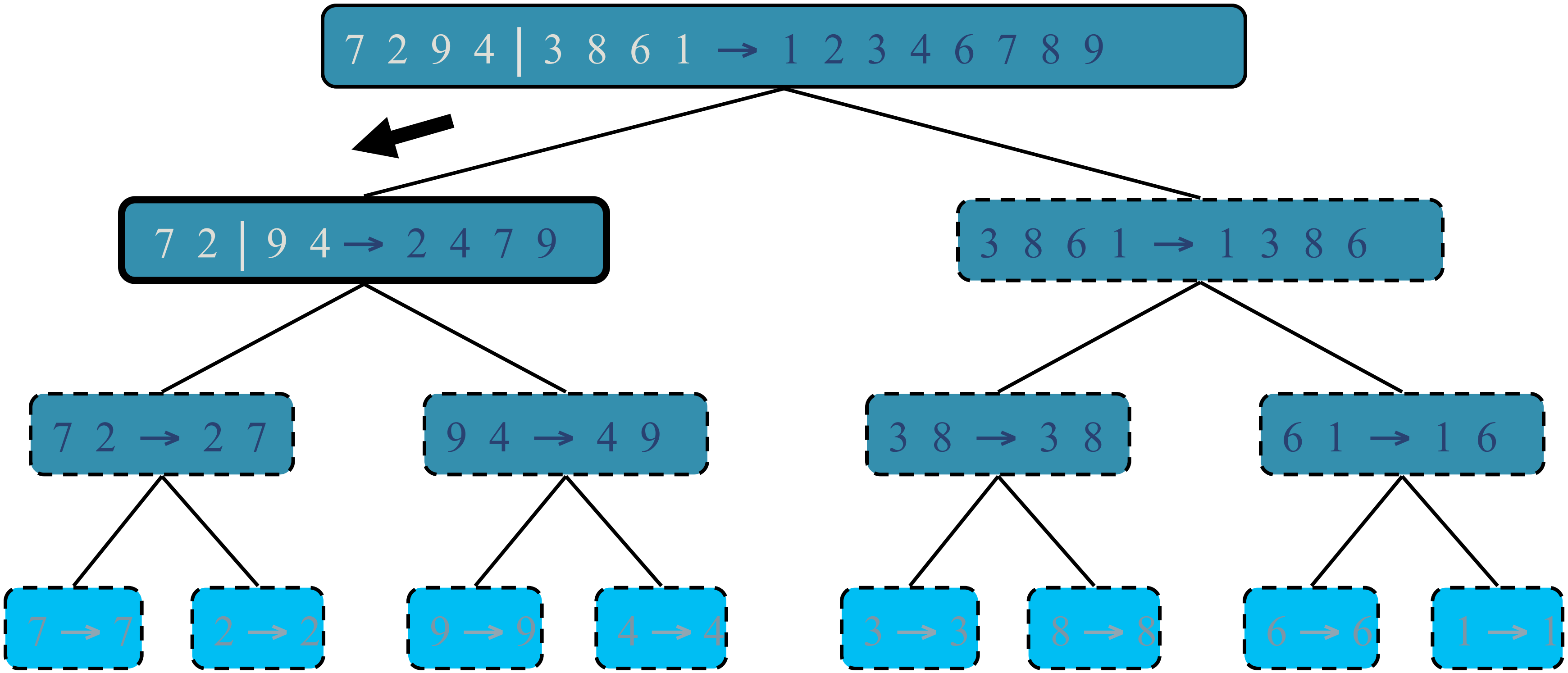
3 → 3

8 → 8

6 → 6

1 → 1

Partition



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

7 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

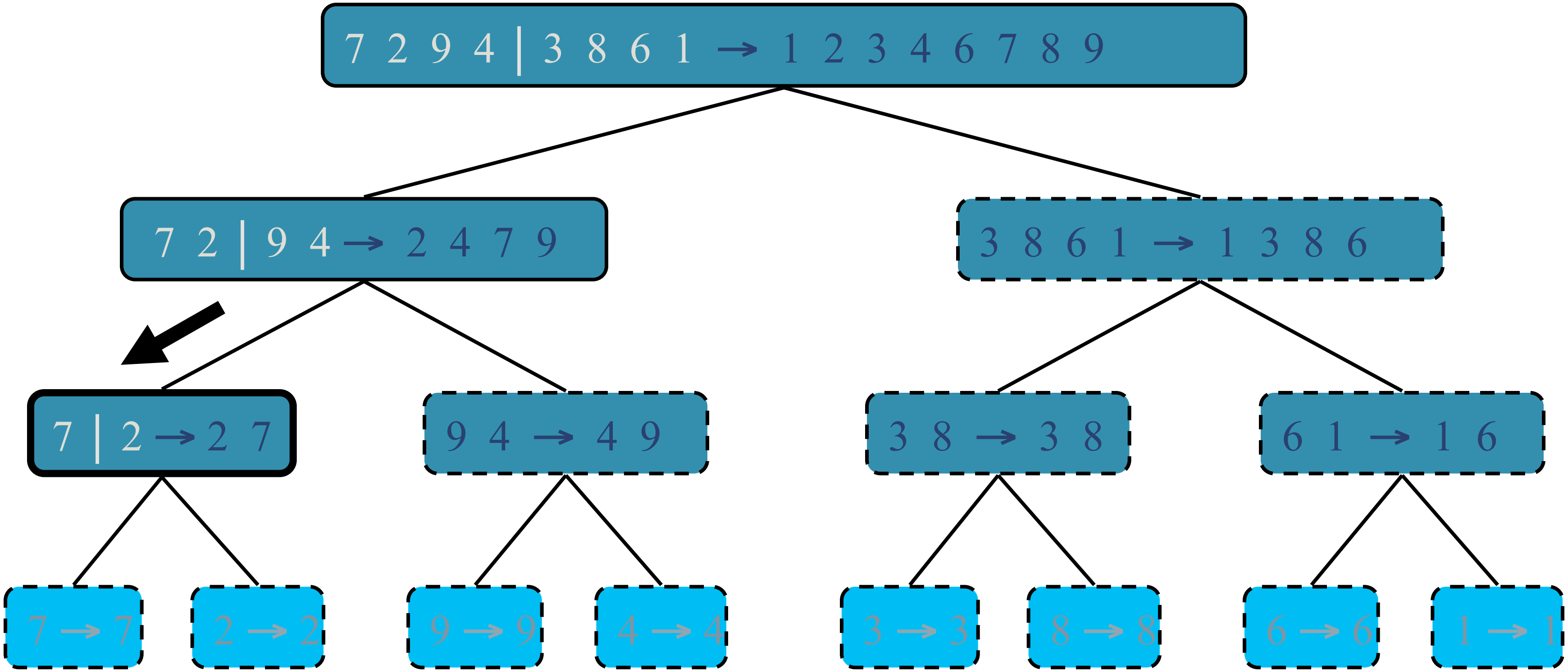
3 → 3

8 → 8

6 → 6

1 → 1

Partition



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

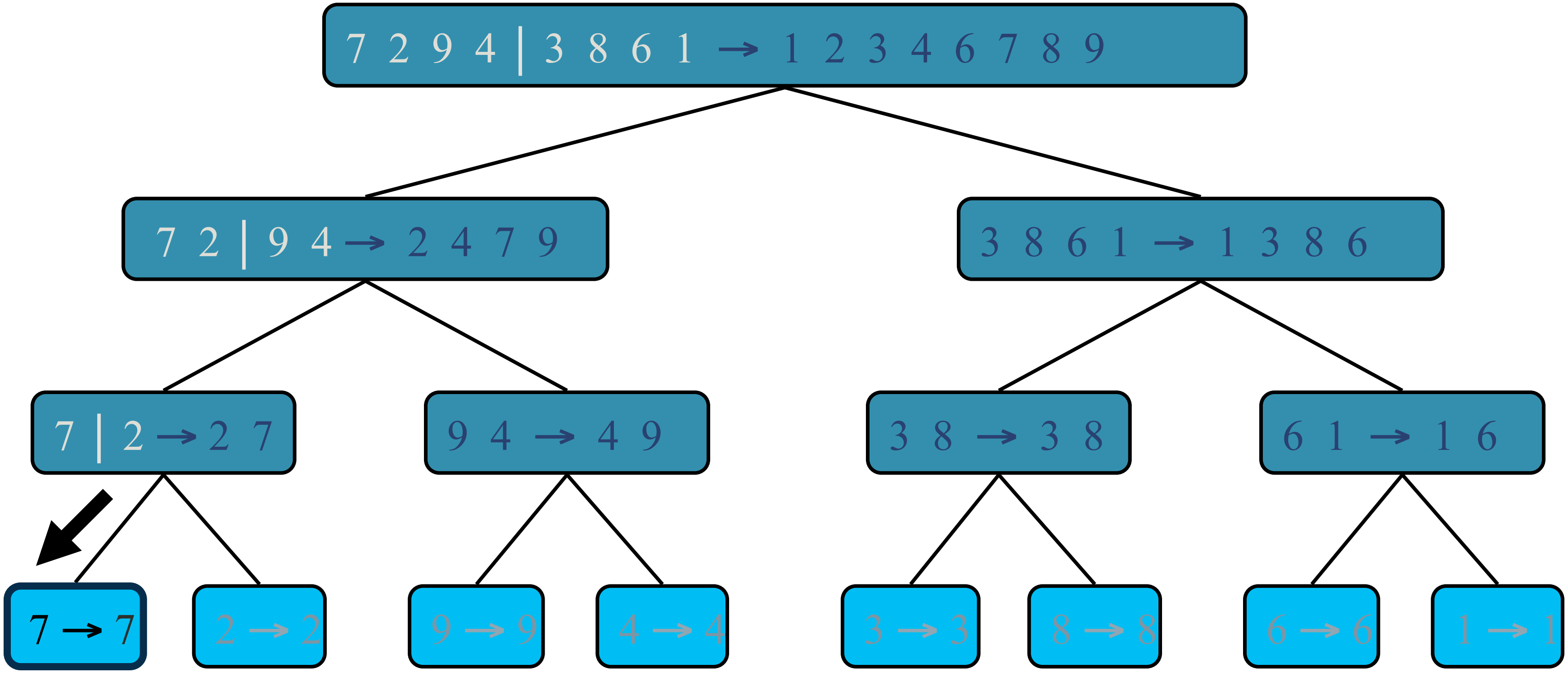
3 → 3

8 → 8

6 → 6

1 → 1

Recur: base case



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

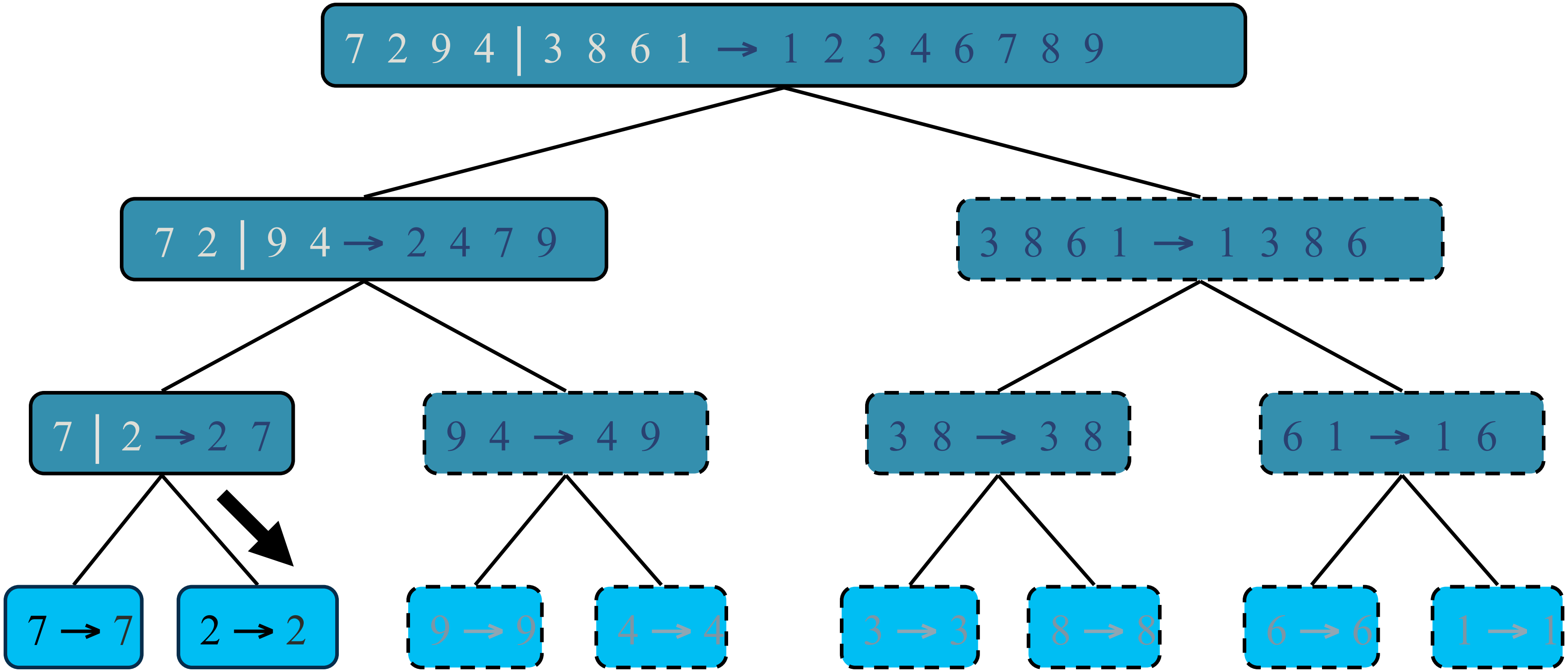
3 → 3

8 → 8

6 → 6

1 → 1

Recur: Base case



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

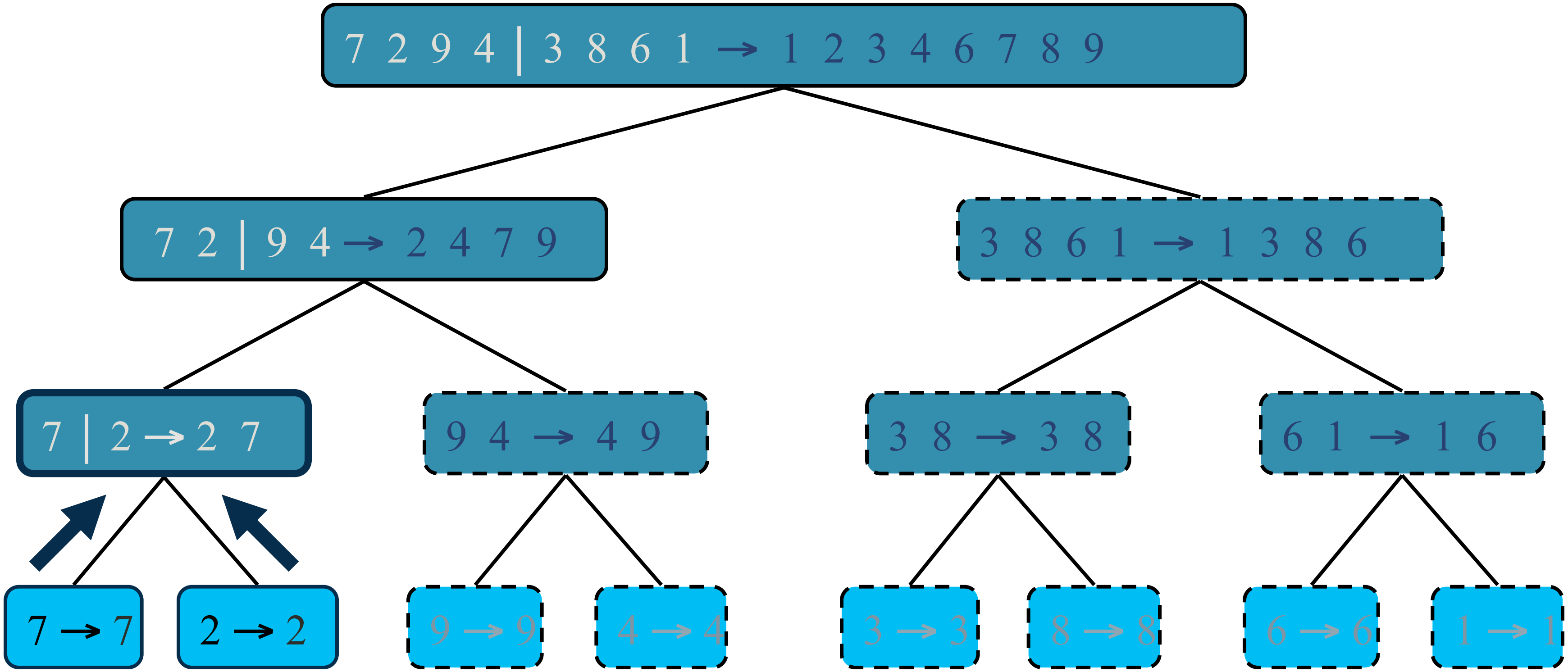
3 → 3

8 → 8

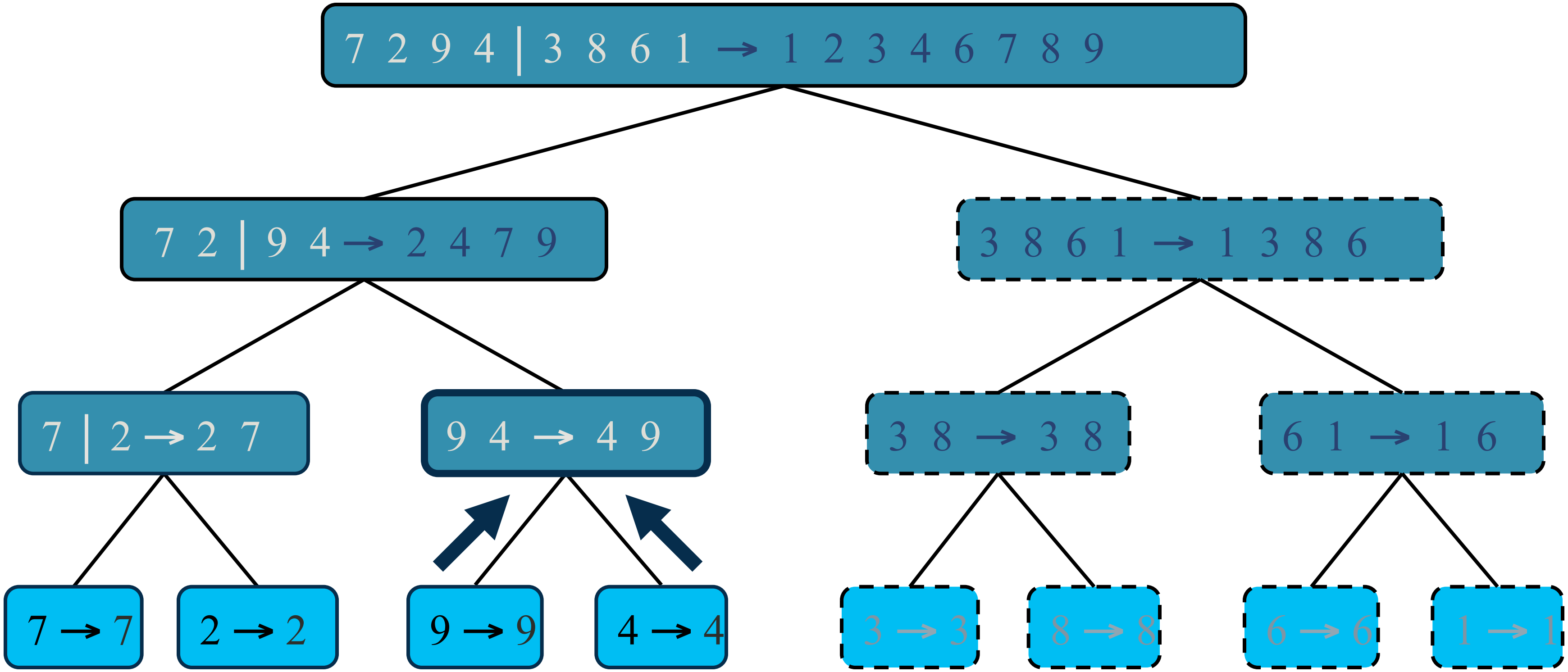
6 → 6

1 → 1

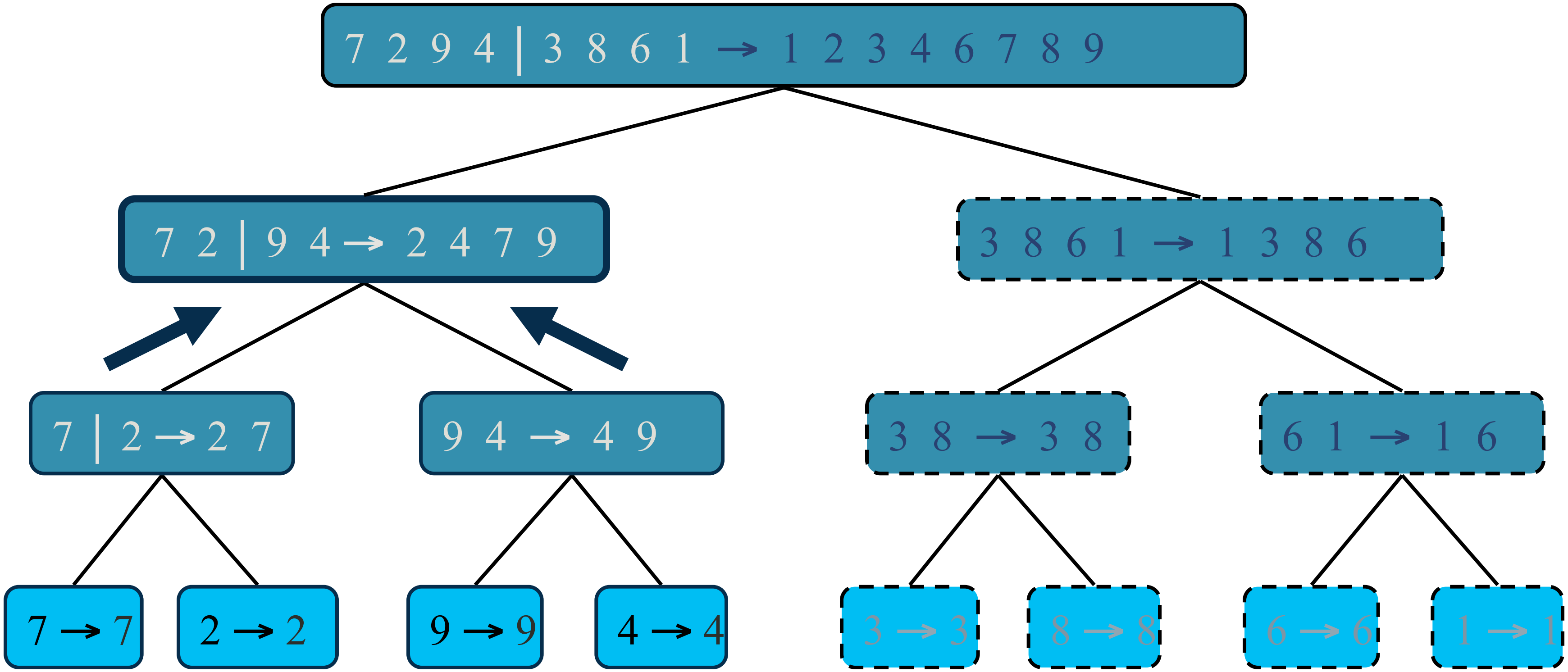
Merge


7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 8 6
7 | 2 → 2 7
9 4 → 4 9
3 8 → 3 8
6 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1

Recursive call,…, merge



Merge



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

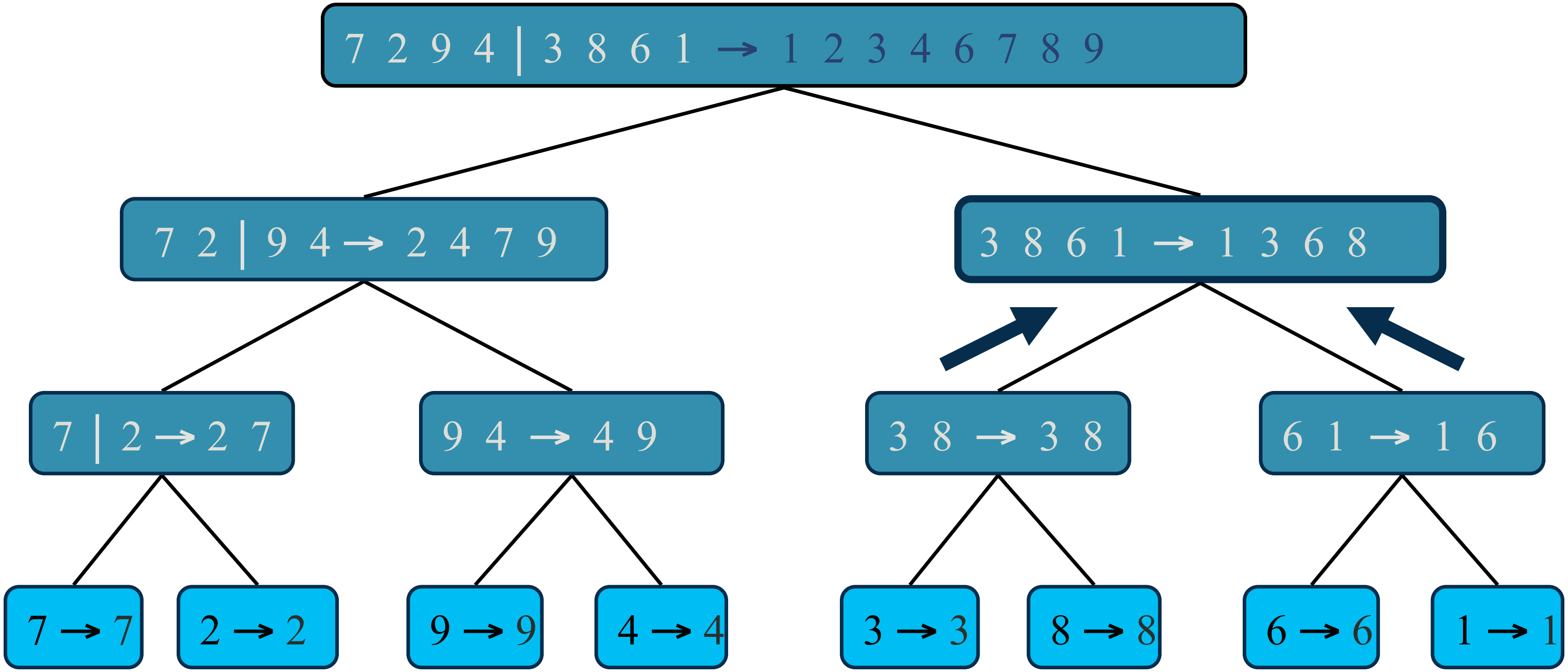
3 → 3

8 → 8

6 → 6

1 → 1

Recursive call, …, merge, merge



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 6 8

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

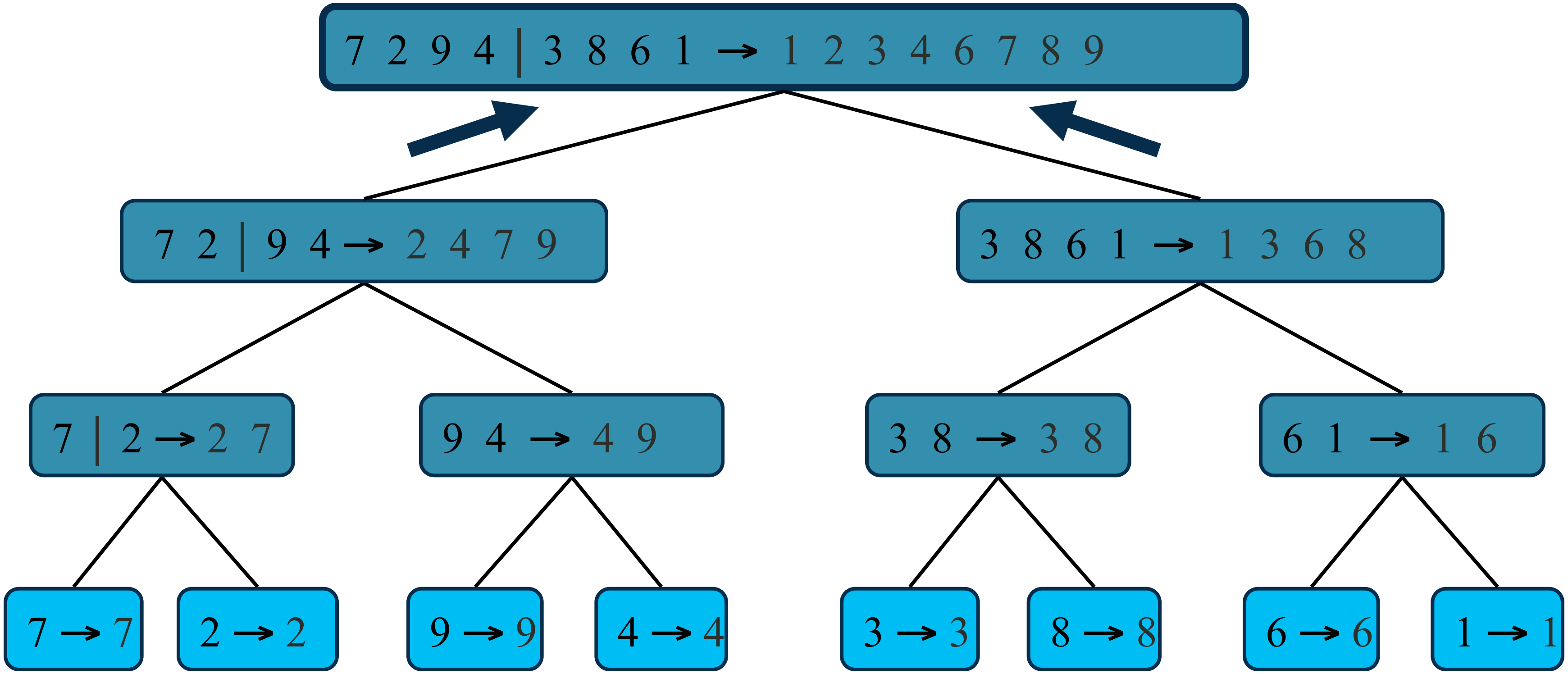
3 → 3

8 → 8

6 → 6

1 → 1

Merge



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 6 8

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

3 → 3

8 → 8

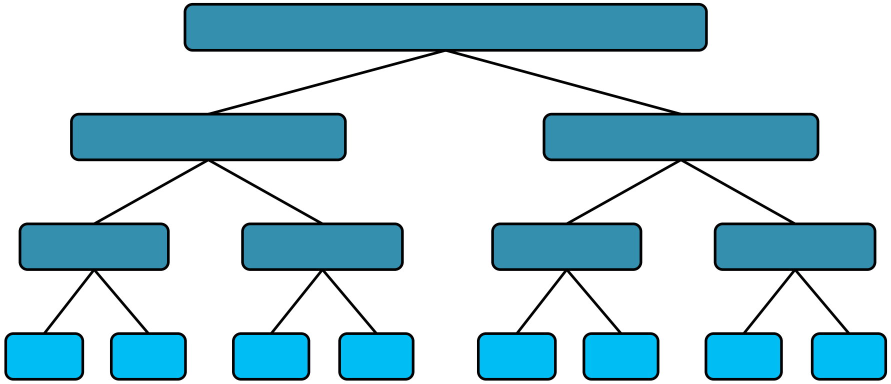
6 → 6

1 → 1

Analysis of Merge-sort

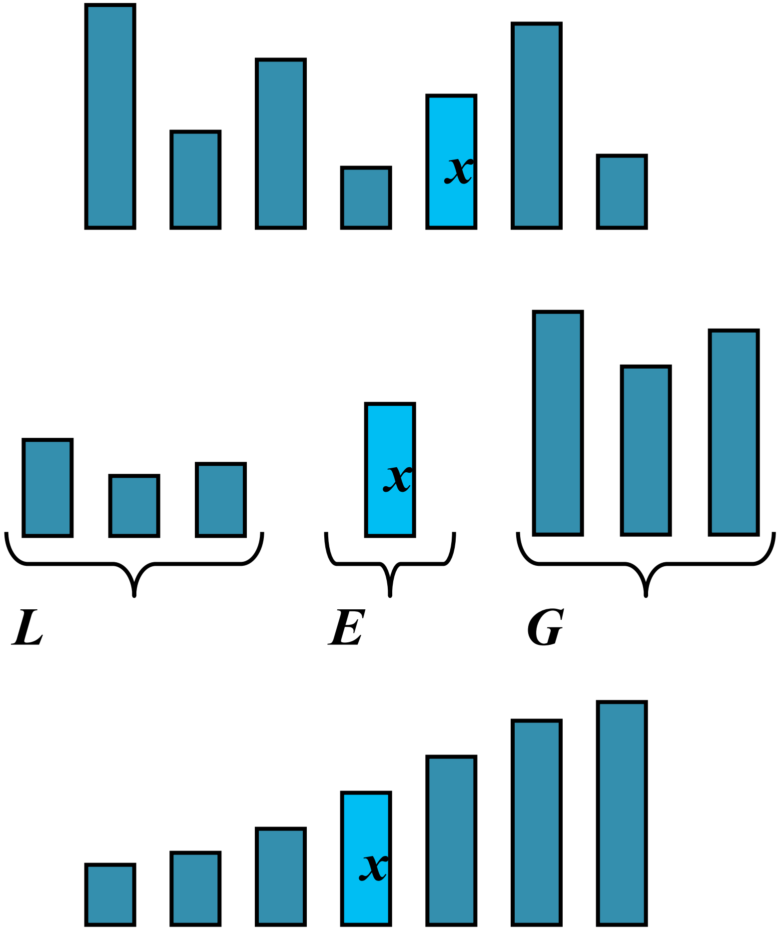
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Quick-sort

A randomized sorting algorithm based on the divide-and-conquer paradigm:



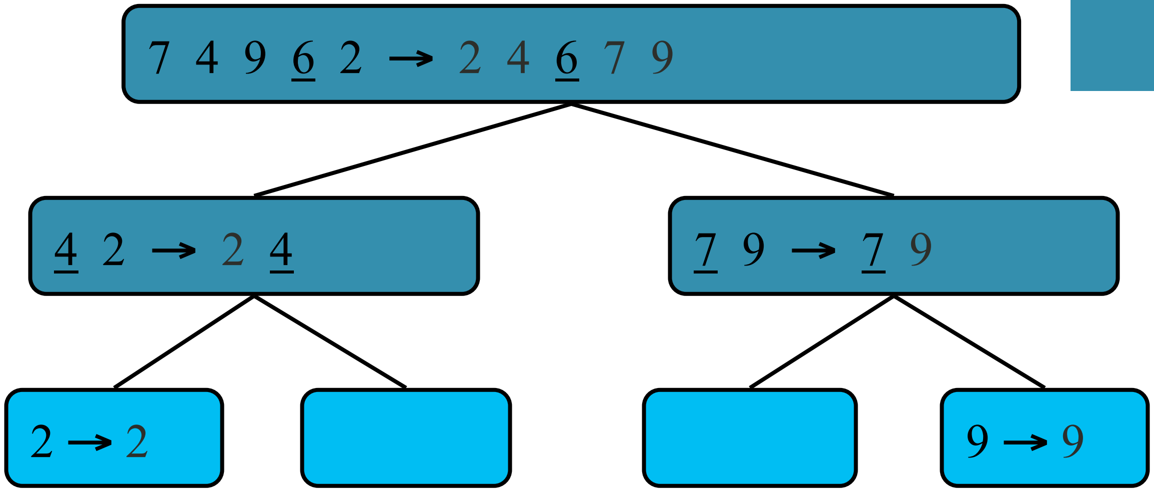
- Divide: pick a random element ***x*** (called pivot) and partition ***S*** into
  - ***L*** elements less than ***x***
  - ***E*** elements equal ***x***
  - ***G*** elements greater than ***x***
- Recur: sort ***L*** and ***G***
- Conquer: join ***L***, ***E*** and ***G***

Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element *y* from *S* and
  - We insert *y* into *L*, *E* or *G*, depending on the result of the comparison with the pivot *x*
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes *O*(1) time
- Thus, the partition step of quick-sort takes *O*(*n*) time

```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

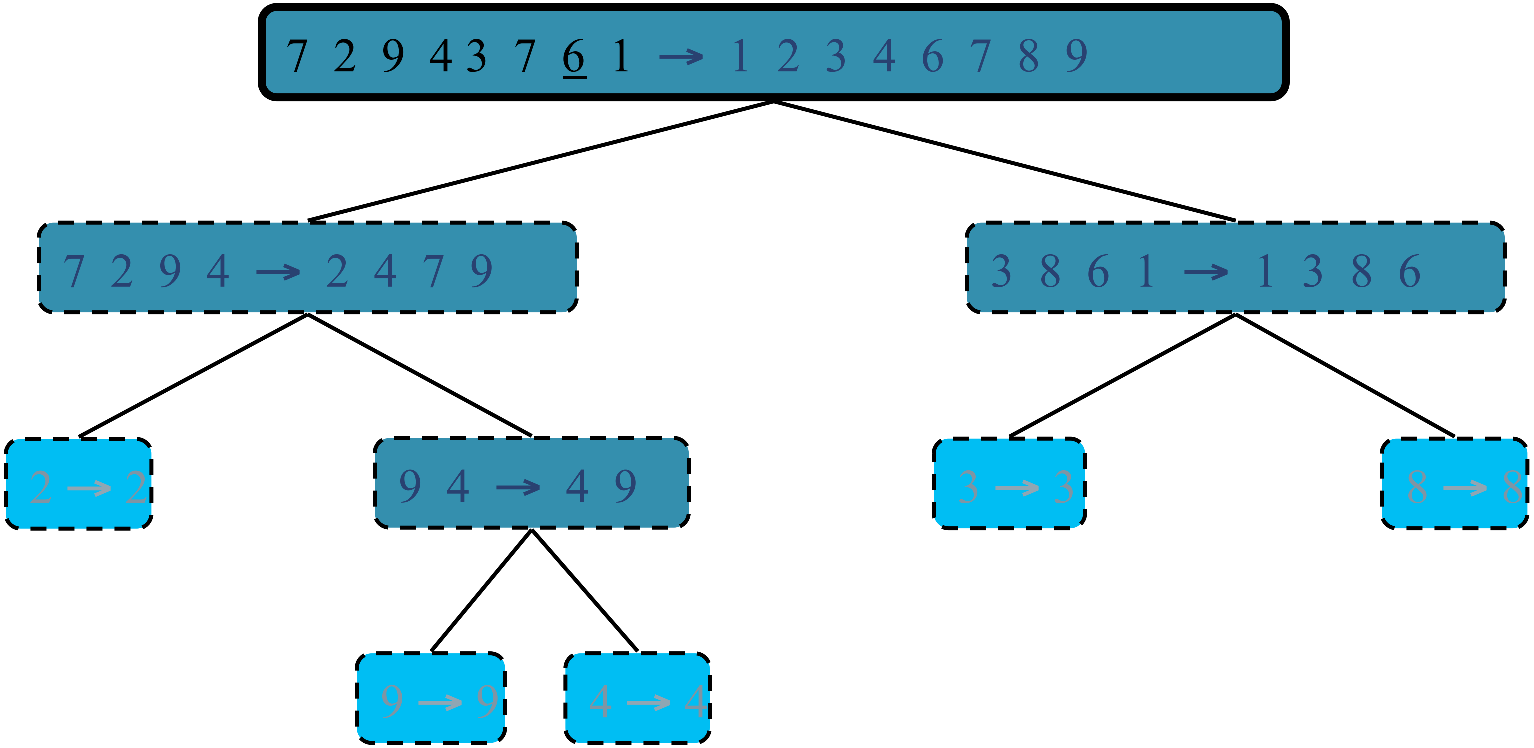
Quick-Sort Tree



- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1

Execution Example

- Pivot selection



7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9

7 2 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

2 → 2

9 4 → 4 9

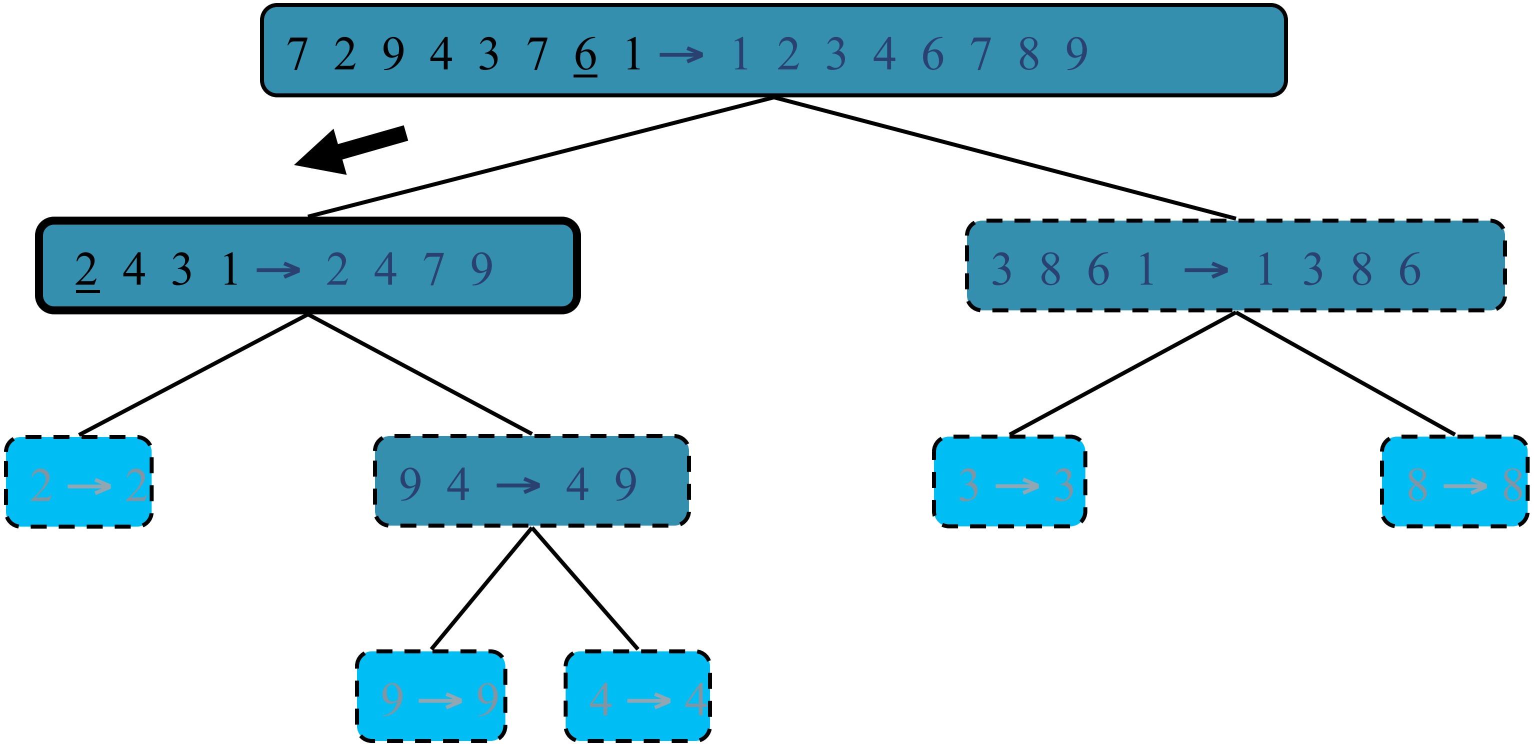
3 → 3

8 → 8

9 → 9

4 → 4

- Partition, recursive call, pivot selection



7 2 9 4 3 7 $\underline{6}$ 1 → 1 2 3 4 6 7 8 9

$\underline{2}$ 4 3 1 → 2 4 7 9

3 8 6 1 → 1 3 8 6

2 → 2

9 4 → 4 9

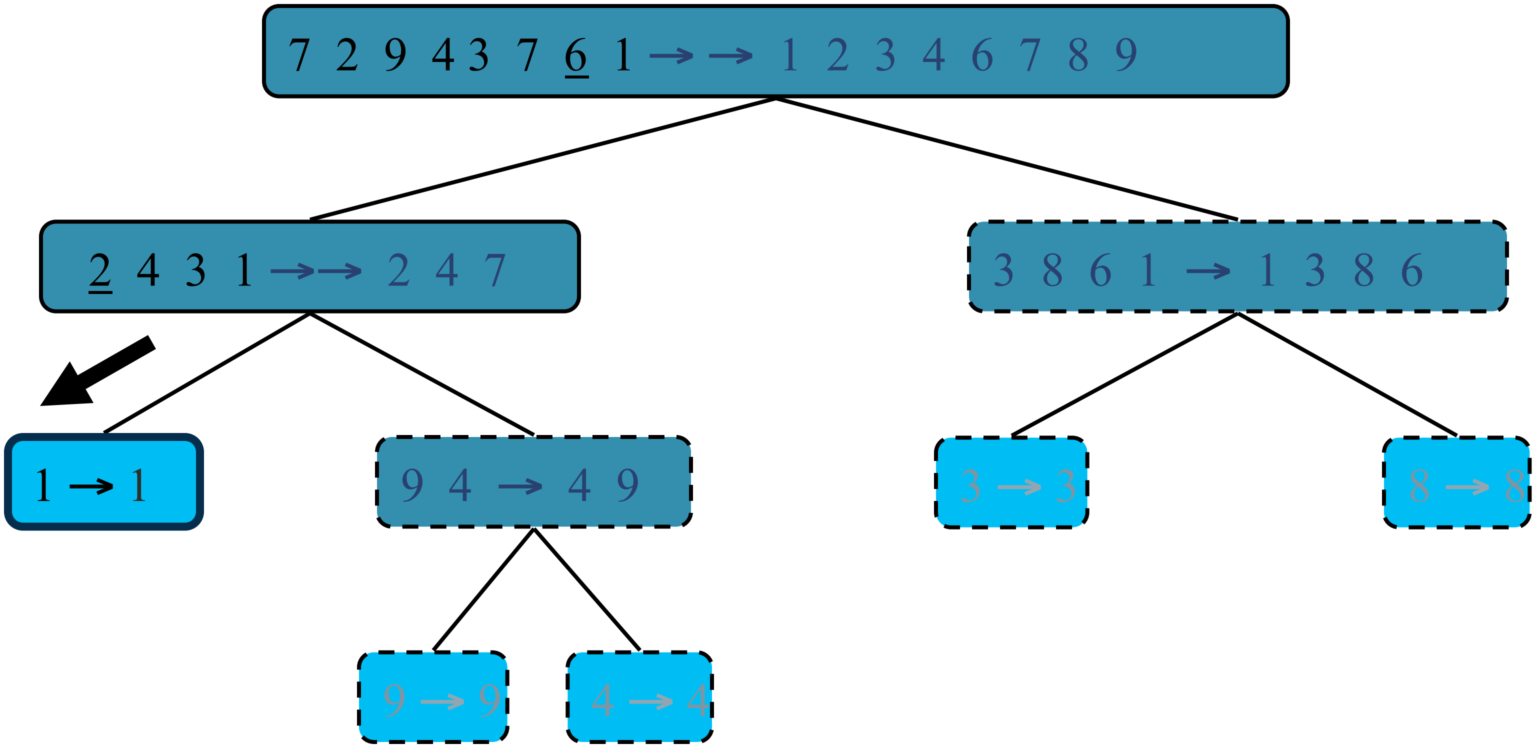
3 → 3

8 → 8

9 → 9

4 → 4

- Partition, recursive call, base case



7 2 9 4 3 7 6 1 → → 1 2 3 4 6 7 8 9

2 4 3 1 → → 2 4 7

3 8 6 1 → 1 3 8 6

1 → 1

9 4 → 4 9

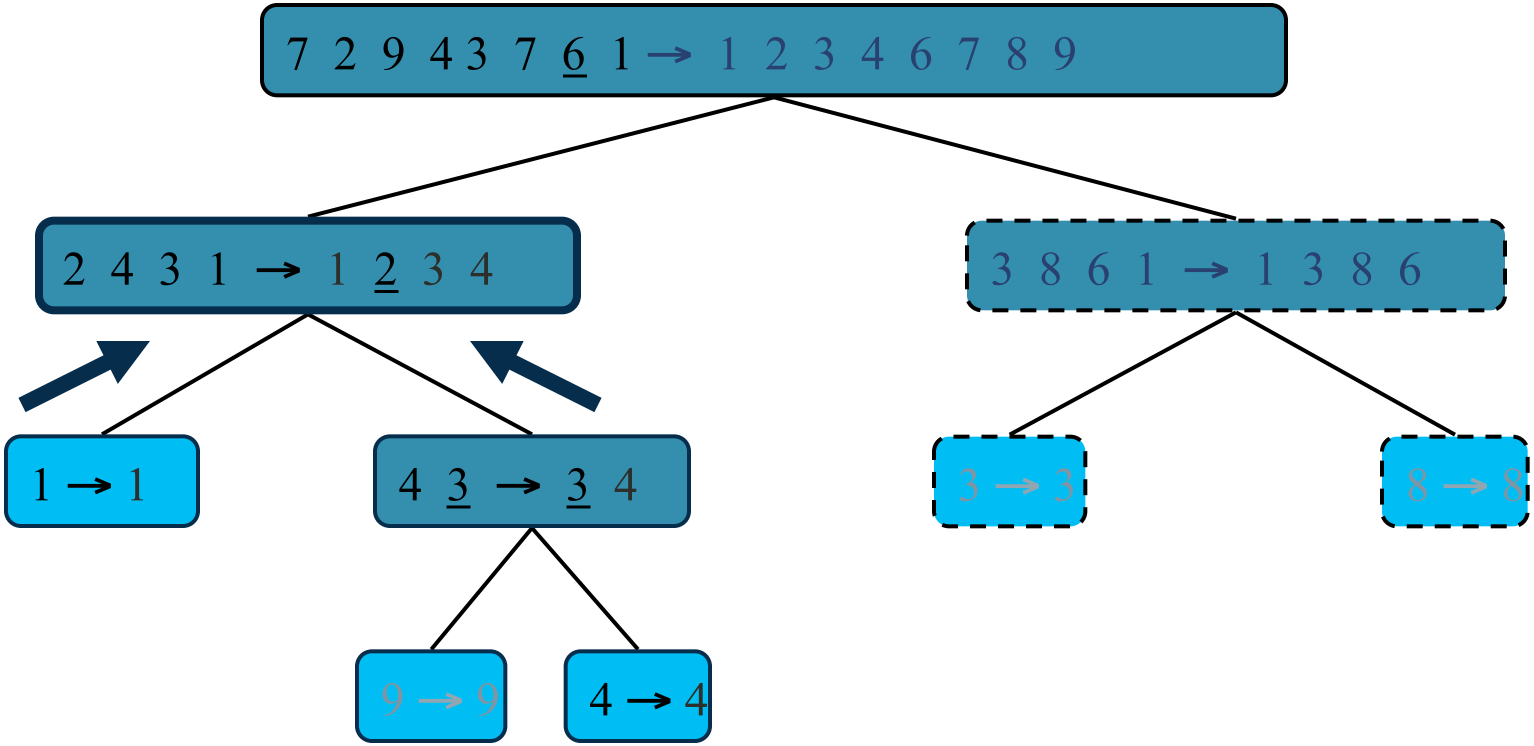
3 → 3

8 → 8

9 → 9

4 → 4

- Recursive call, …, base case, join



7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9

2 4 3 1 → 1 2 3 4

3 8 6 1 → 1 3 8 6

1 → 1

4 3 → 3 4

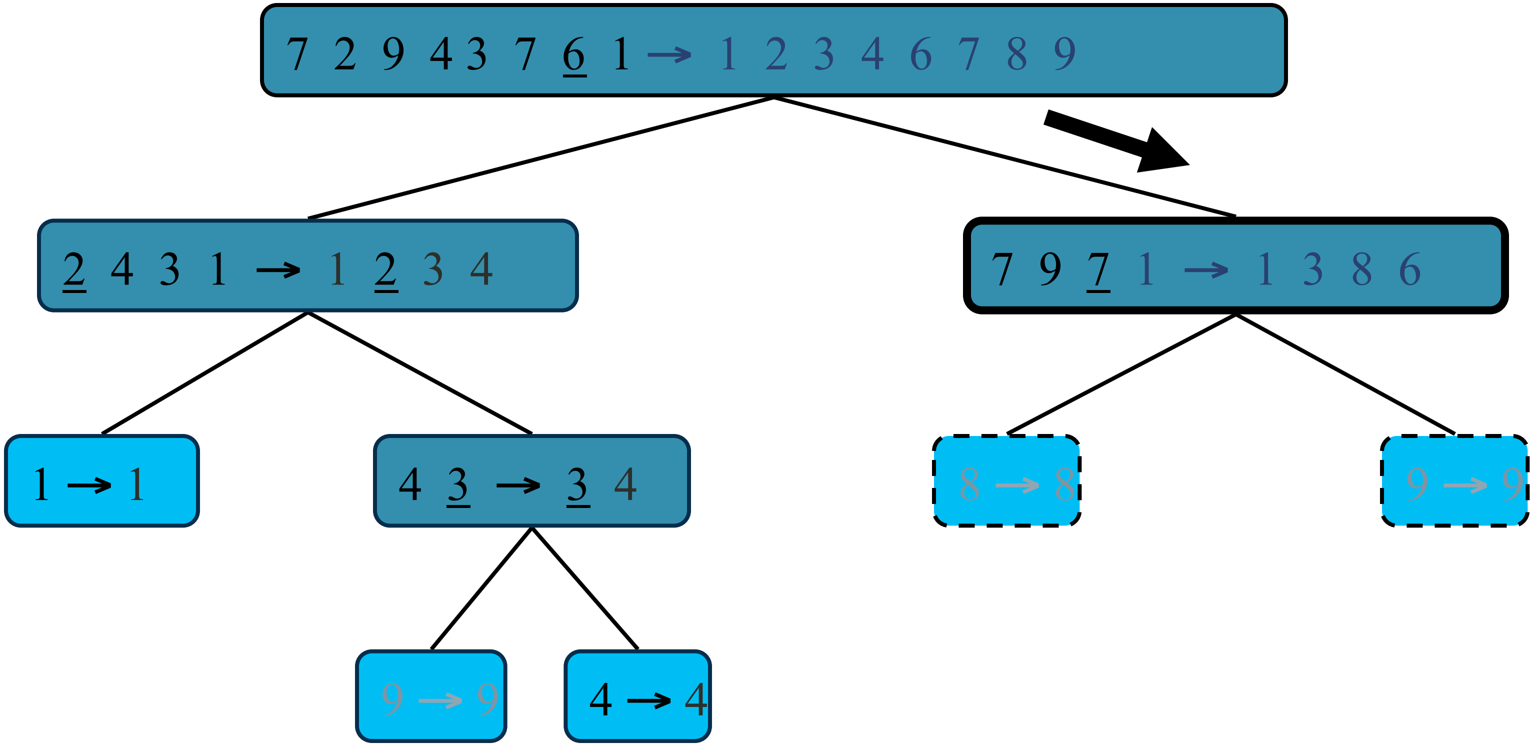
3 → 3

8 → 8

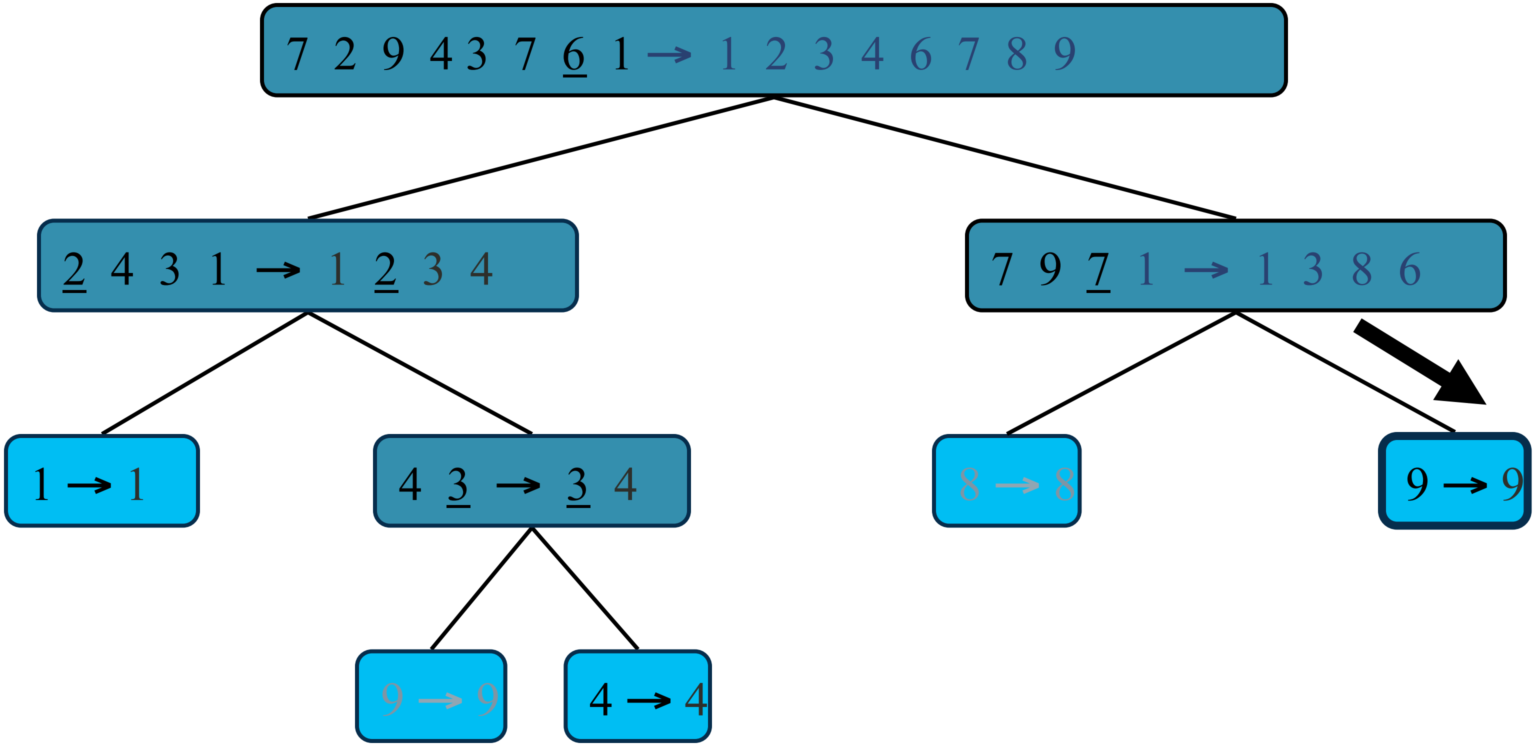
9 → 9

4 → 4

- Recursive call, pivot selection



- Partition, …, recursive call, base case



7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 8 9

2 4 3 1 → 1 2 3 4

7 9 7 1 → 1 3 8 6

1 → 1

4 3 → 3 4

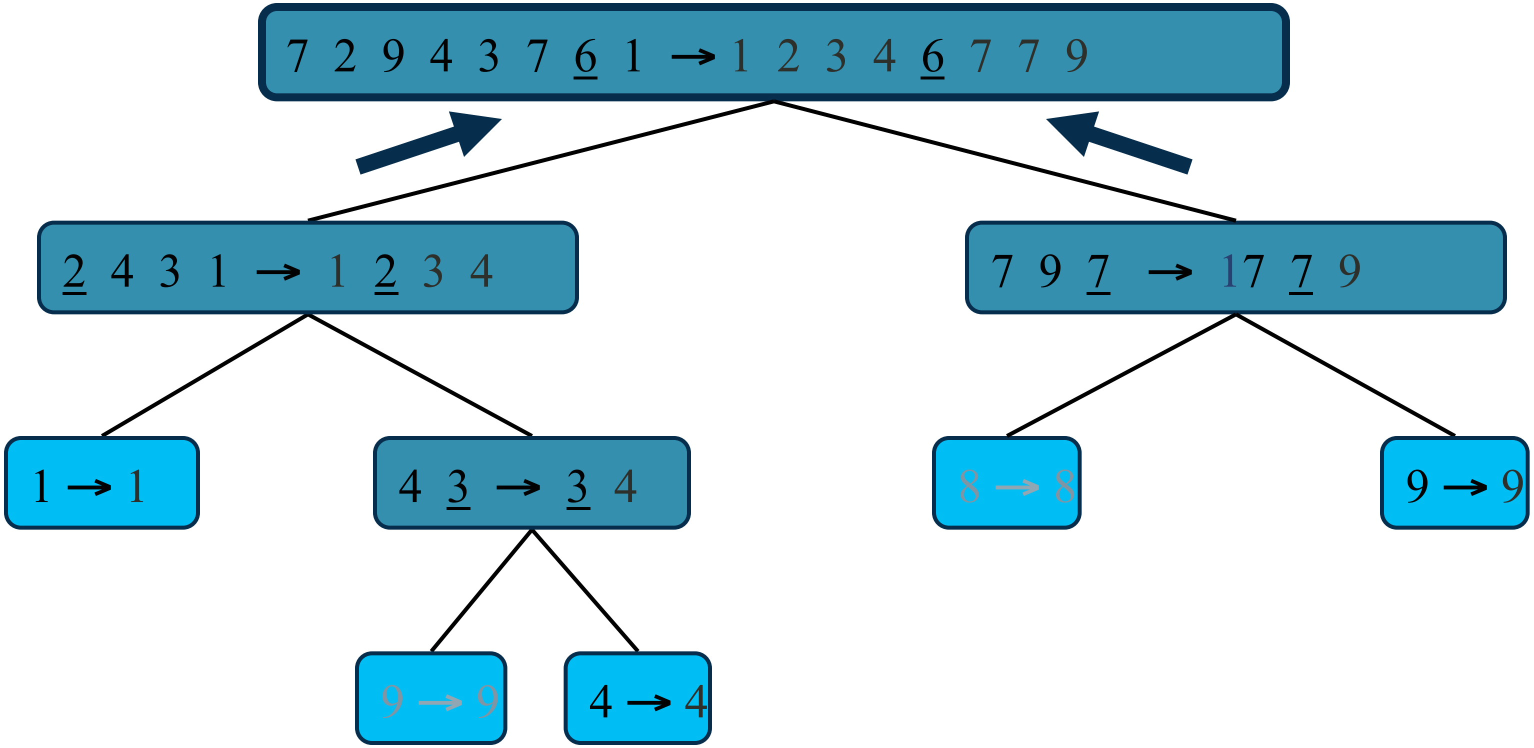
8 → 8

9 → 9

9 → 9

4 → 4

- Join, join



7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9

2 4 3 1 → 1 2 3 4

7 9 7 → 17 7 9

1 → 1

4 3 → 3 4

8 → 8

9 → 9

9 → 9

4 → 4

In-place Quick-sort

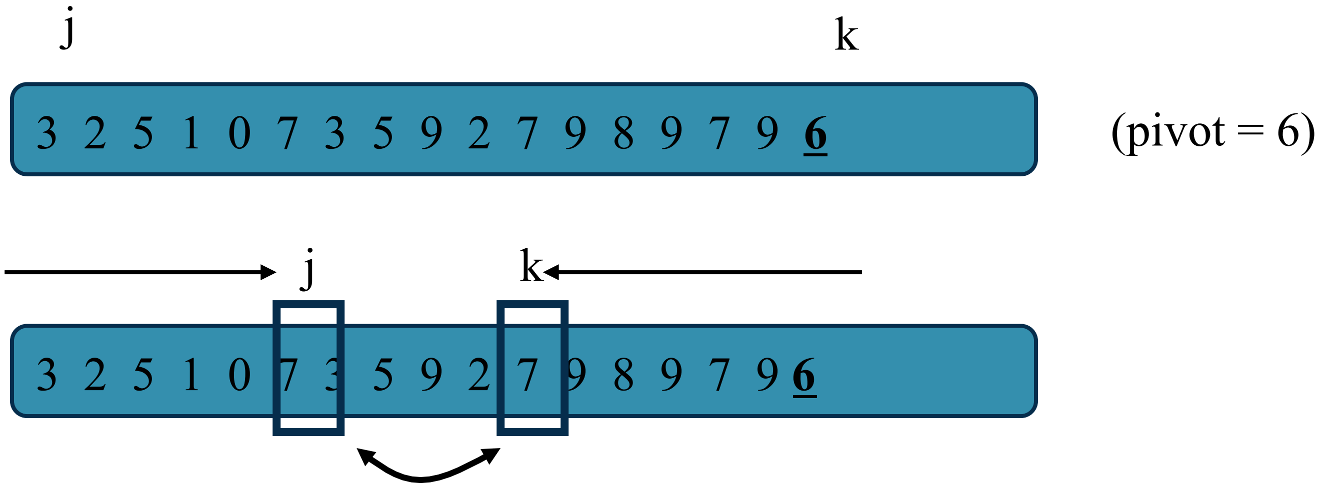
- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements
- The recursive calls consider
  - elements with rank less than ***h***
  - elements with rank greater than ***k***

```
Algorithm inPlaceQuickSort($S$, $l$, $r$)
    Input sequence $S$, ranks $l$ and $r$
    Output sequence $S$ with the
        elements of rank between $l$ and $r$
        rearranged in increasing order
    if $l \geq r$
        return
    $i \leftarrow$ a random integer between $l$ and $r$
    $x \leftarrow S.elemAtRank(i)$
    $(h, k) \leftarrow inPlacePartition(x)$
    $inPlaceQuickSort(S, l, h - 1)$
    $inPlaceQuickSort(S, k + 1, r)$
```

In-Place Quick-Sort

- Perform the partition using two indices to split S into L, E, G
- Algorithm Quicksort(leftBound, rightBound, S)
  - If(leftBound>=rightBound) return;
  - Set rightBound as the pivot (x = S[righBound])
  - Set j = leftBound; k = rightBound-1;
  - When j<k:
    - Scan j to the right (j++) until j >= k or the element S[j] > x.
    - Scan k to the left (k--) until j>=k or the element S[k]<=x.
    - Swap elements if j < k
  - Swap pivot with j
  - Quicksort(leftBound, j-1, S); Quicksort(j+1, rightBound, S)

In-Place Quick-Sort



j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 9 **<u>6</u>**

(pivot = 6)

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 9 **<u>6</u>**

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	in-place slow (good for small inputs)
insertion-sort	$O(n^2)$	in-place slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	in-place, randomized fastest (good for large inputs)
heap-sort	$O(n \log n)$	in-place fast (good for large inputs)
merge-sort	$O(n \log n)$	sequential data access fast (good for huge inputs)

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

Recurrence Equation Analysis

- We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
- That is, a solution that has ***T*(*n*)** only on the left-hand side.
- We can achieve this by iterative substitution:
- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern

Iterative Substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2T(n/2^2) + 2bn \\
&= 2^3T(n/2^3) + 3bn \\
&= 2^4T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^iT(n/2^i) + ibn
\end{aligned}
$$

- Note that base, T(n)=b, case occurs when 2^i=n.
- That is, i = log n. So,

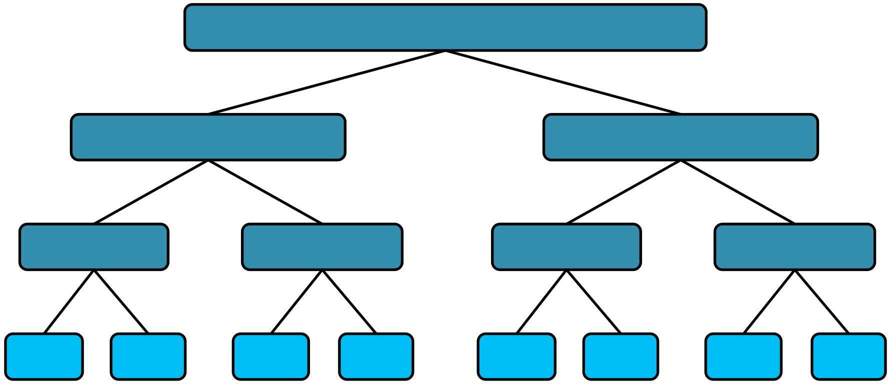
$$T(n) = bn + bn\log n$$

- Thus, T(n) is O(n log n).

The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$



depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…

Total time = $bn + bn \log n$

(last level plus all previous levels)

Guess-and-Test Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:
- For example:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: T(n) < cn log n

Guess-and-Test Method

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn\log n \\
&< 2(c(n/2)\log(n/2)) + bn\log n \\
&= cn(\log n - \log 2) + bn\log n \\
&= cn\log n - cn + bn\log n \\
&< cn\log n(?)
\end{aligned}
$$

- Wrong!
- We cannot make this last line be less than cn log n

Guess-and-Test Method, (cont.)

- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess #2: $T(n) < cn \log^2 n$.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n \qquad \text{(if c>b)} \end{aligned}$$

So, $T(n)$ is $O(n \log^2 n)$.

In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Master Method

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Master Method, Example 1

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

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Solution:
  - a = 4, b =2, f(n) is n
  - $\log_b a = 2$, so case 1 says T(n) is $O(n^2)$

Master Method, Example 2

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

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

     provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Solution:
  - a = 2, b =2
  - Solution: $\log_b a=1$, so case 2 says T(n) is $O(n \log^2 n)$.

Master Method, Example 3

$$T(n) = T(n/3) + n\log n$$

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Solution:
  - a = 1, b =3
  - $\log_b a=0$, so case 3 says T(n) is O(n log n).

Master Method, Example 4

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

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Solution:
  - a = 8, b =2
  - $\log_b a=3$, so case 1 says T(n) is $O(n^3)$.

HW8 (Due on Nov. 28)

Quick sort keywords!

- Implement a quick sort algorithm for keywords
- Add each keyword into an array/linked list inorder
- Sort the keywords upon request
- Output all the keywords

Operations

Given a sequence of operations in a txt file, parse the txt file and execute each operation accordingly

operations	**description**
add(Keyword k)	Insert a keyword k to an array
sort()	Sort the keywords using quick sort
output()	Output all keywords in the array

An input file

Similar to HW7,

1. You need to read the sequence of operations from a txt file
2. The format is firm
3. Raise an exception if the input does not match the format

```
add Fang 3
add Yu 5
add NCCU 2
add UCSB 1
output
add MIS 4
Sort
output
```

```
[Fang, 3][Yu, 5][NCCU, 2][UCSB, 1]

[UCSB, 1][NCCU, 2][Fang, 3][MIS, 4] [Yu, 5]
```

Midterm on Dec. 5 (9:10-12:00am, 大勇樓106)

- Lec 1-10, TextBook Ch1-8, 11-12 (part)
- How to prepare your midterm:
  - Understand “ALL” the materials mentioned in the slides
    - Discuss with me, your TAs, or classmates
    - Read the text book to help you understand the materials
- You are allowed to bring an A4 size note
  - Prepare your own note; write whatever you think that may help you get better scores in the midterm