Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University
a.khan@innopolis.ru

Recap

- Algorithmic Strategies
  - Brute-Force
  - Divide-and-Conquer
  - Dynamic Programming

Disclaimer!!!

- You will be asked to do plenty of self-reading this week
- All topics will be considered a part of the materials covered in the course (unless explicitly stated)

Objectives

- What is sorting?
- Why must one learn about sorting algorithms in this course?
- Properties of sorting algorithms
- Sorting Algorithms
    - Bubble Sort, Selection Sort, Insertion Sort
    - Merge-sort, Quick-sort
    - Time complexity of comparison-based sort

Sorting

- Arranging items of the same kind, class or nature, in some ordered sequence

- Sorting Algorithm: an algorithm that arranges elements of a collection in a certain order

- Input: $a_1, a_2, \cdots, a_n$

- Output: $a'_1, a'_2, \cdots a'_n$ such that $a'_1 \leq a'_2, \leq \cdots \leq a'_n$

Reasons to Study Sorting Algorithms

- Almost all of the ideas used in design of algorithms appear in the context of sorting
    - Time Analysis, Algorithmic Strategies, Data Structures
- Computers have spent and will keep spending more time sorting than doing anything else
- Most thoroughly studied problem in computer science

Applications

- Punch Line: Sorting takes $O(n \log n)$

- So many important algorithms can be reduced to sorting

    - Closest pair

    - Element uniqueness

    - Frequency distribution

Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.

Try Yourself!

- Punch Line: $O(n \log n)$

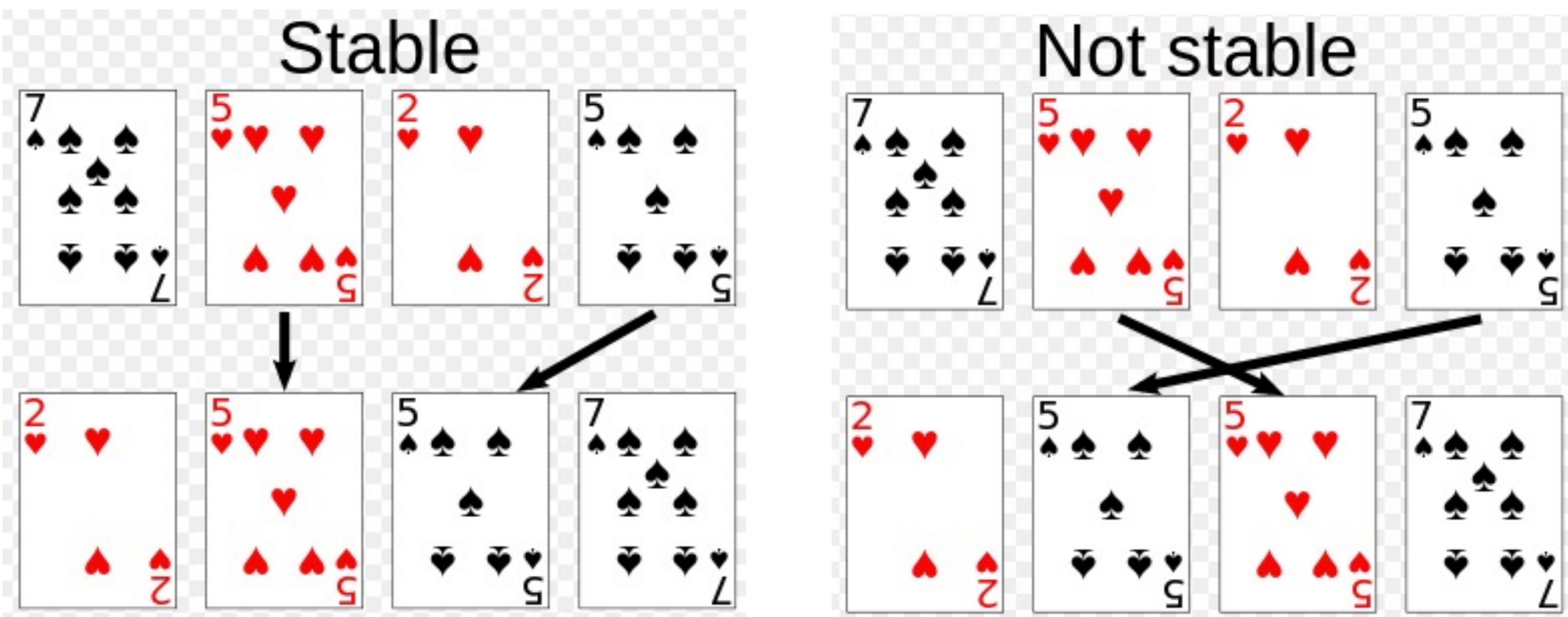
❖ Give an algorithm to determine whether two sets (of size m and n, respectively) are *disjoint*.

Sorting Algorithms

- Many ways to classify sorting algorithms
  - ❖ Time Complexity
  - ❖ Stable vs. Unstable
  - ❖ In place sorting or not
  - ❖ Whether it works by comparison or not

Stability

- Stable sort is one which preserves the original order of the input sent whenever it encounters items of the same rank it



http://en.wikipedia.org/wiki/Sorting_algorithm#/media/File:Sorting_stability_playing_cards.svg

In-place Sorting

- When the algorithm uses a small fixed amount of extra space to perform sorting

Sorting Algorithms

- ➢ Bubble Sort
- ➢ Selection Sort
- ➢ Insertion Sort
- ➢ Merge Sort
- ➢ Quick Sort
- ➢ Heap Sort
- …

Bubble Sort

Simple rules

1. Start from the left most position in the unsorted sequence
2. Compare two adjacent keys
3. If one on the left is bigger, swap the keys
4. Move on to the next key

As elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda

Bubble Sort

29	10	14	37	13

Pass 1: Compare Item at position 1 and 2, and swap if needed!

Bubble Sort

10	29	14	37	13

Pass 1: Compare Item at position 2 and 3, and swap if needed!

Bubble Sort

10	14	29	37	13

Pass 1: Compare Item at position 3 and 4, and swap if needed!

Bubble Sort

10	14	29	37	13

Pass 1: Compare Item at position 4 and 5, and swap if needed!

Bubble Sort

10	14	29	13	37

Unsorted

Sorted

After Pass 1

Bubble Sort

10	14	29	13	37

Sorted

Pass 2: Compare Item at position 1 and 2, and swap if needed!

Bubble Sort

10	14	29	13	37

Sorted

Pass 2: Compare Item at position 2 and 3, and swap if needed!

Bubble Sort

10	14	29	13	37

Sorted

Pass 2: Compare Item at position 3 and 4, and swap if needed!

Bubble Sort

10	14	13	29	37

Unsorted

Sorted

After Pass 2

Bubble Sort

10	14	13	29	37

Sorted

Pass 3: Compare Item at position 1 and 2, and swap if needed!

Bubble Sort

10	14	13	29	37

Sorted

Pass 3: Compare Item at position 2 and 3, and swap if needed!

Bubble Sort

10	13	14	29	37

Unsorted

Sorted

After Pass 3

Bubble Sort

10	13	14	29	37

Sorted

Pass 4: Compare Item at position 1 and 2, and swap if needed!

Bubble Sort

10	13	14	29	37

Unsorted

Sorted

After Pass 4

Bubble Sort

10	13	14	29	37

Sorted

The sequence is sorted.

Bubble Sort

The complexity of bubble sort is $O(n^2)$

Merge-Sort

Divide-and-Conquer

- Divide-and-conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two (or more) disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- It has $O(n \log n)$ running time

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***$(\boldsymbol{S})$

    Input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$ elements

    Output sequence $\boldsymbol{S}$ sorted according to $\boldsymbol{C}$

    if ***S.size***() > 1

        $(\boldsymbol{S}_1, \boldsymbol{S}_2) \leftarrow$ ***partition***$(\boldsymbol{S}, \boldsymbol{n}/2)$

        mergeSort$(\boldsymbol{S}_1)$

        mergeSort$(\boldsymbol{S}_2)$

        $\boldsymbol{S} \leftarrow$ ***merge***$(\boldsymbol{S}_1, \boldsymbol{S}_2)$

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 linked list, takes $O(n)$ time

Merging Two Sorted Sequences

Algorithm *merge*(*A*, *B*)

  Input sequences A and B with
    $n/2$ elements each

  Output sorted sequence of $A + B$

```
S ← empty sequence
while ¬A.isEmpty() && ¬B.isEmpty()
 if A.first().element() < B.first().element()
   S.addLast(A.remove(A.first()))
 else
   S.addLast(B.remove(B.first()))
while ¬A.isEmpty()
 S.addLast(A.remove(A.first()))
while ¬B.isEmpty()
 S.addLast(B.remove(B.first()))
return S
```

Merging Two Sorted Sequences

Algorithm *merge*(*A*, *B*)

  Input sequences *A* and *B* with
    n/2 elements each

  Output sorted sequence of *A* + *B*

  S ← empty sequence
  while ¬*A.isEmpty*() && ¬*B.isEmpty*()
   if *A.first*().*element*() < *B.first*().*element*()
     S.addLast(*A.remove*(*A.first*()))
   else
     S.addLast(*B.remove*(*B.first*()))
  while ¬*A.isEmpty*()
   S.addLast(*A.remove*(*A.first*()))
  while ¬*B.isEmpty*()
   S.addLast(*B.remove*(*B.first*()))
  return *S*

Merging Two Sorted Sequences

Algorithm *merge*(*A*, *B*)
  Input sequences *A* and *B* with
    n/2 elements each
  Output sorted sequence of *A* + *B*

  $S \leftarrow$ empty sequence
  while $\neg$*A.isEmpty*() && $\neg$*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*

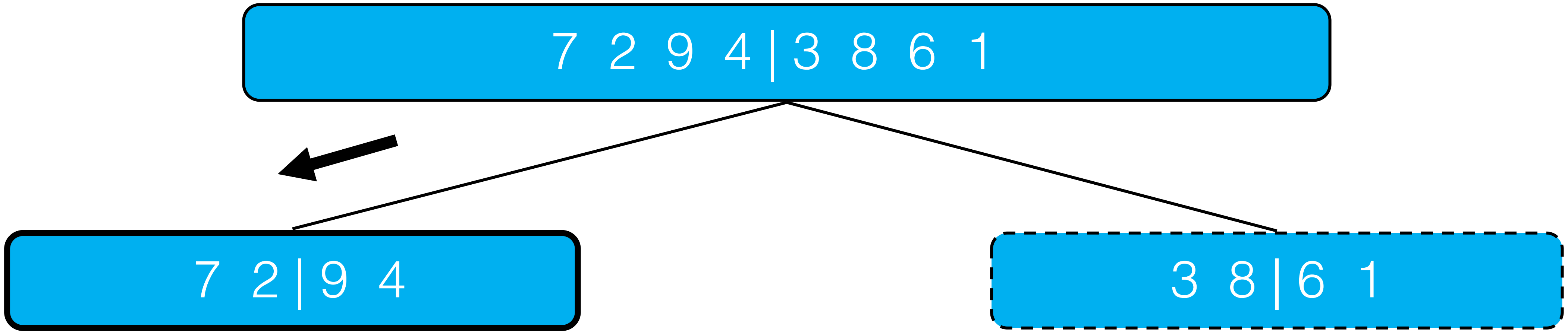
Execution Example

- Partition

7 2 9 4 | 3 8 6 1

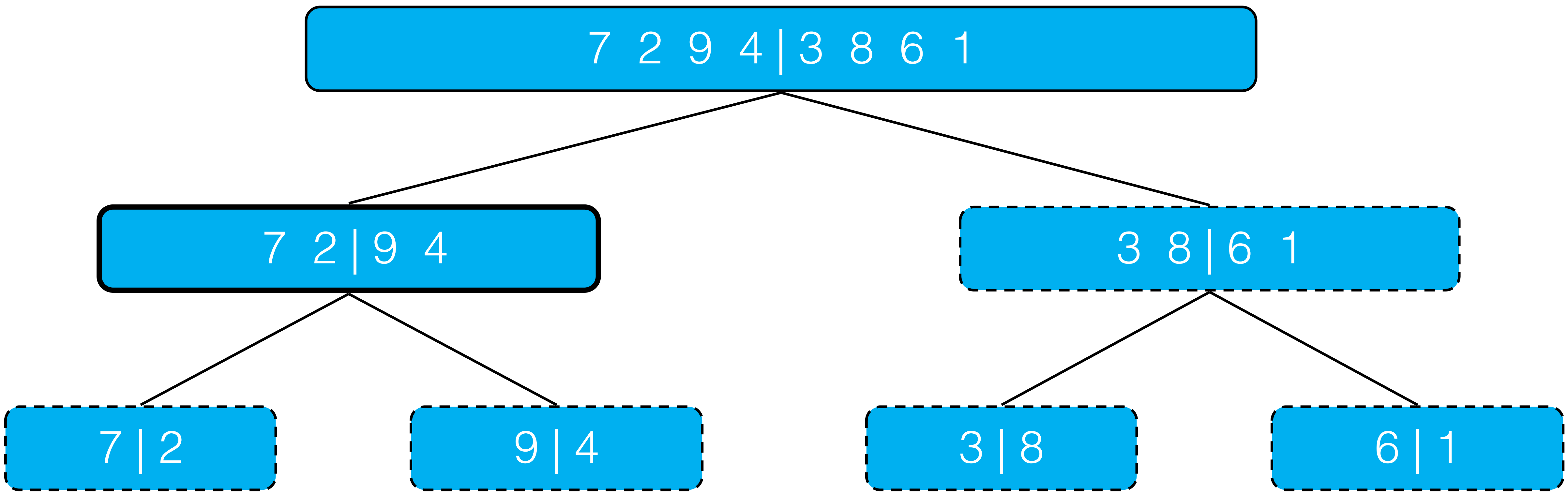
Execution Example

- Recursive call, partition



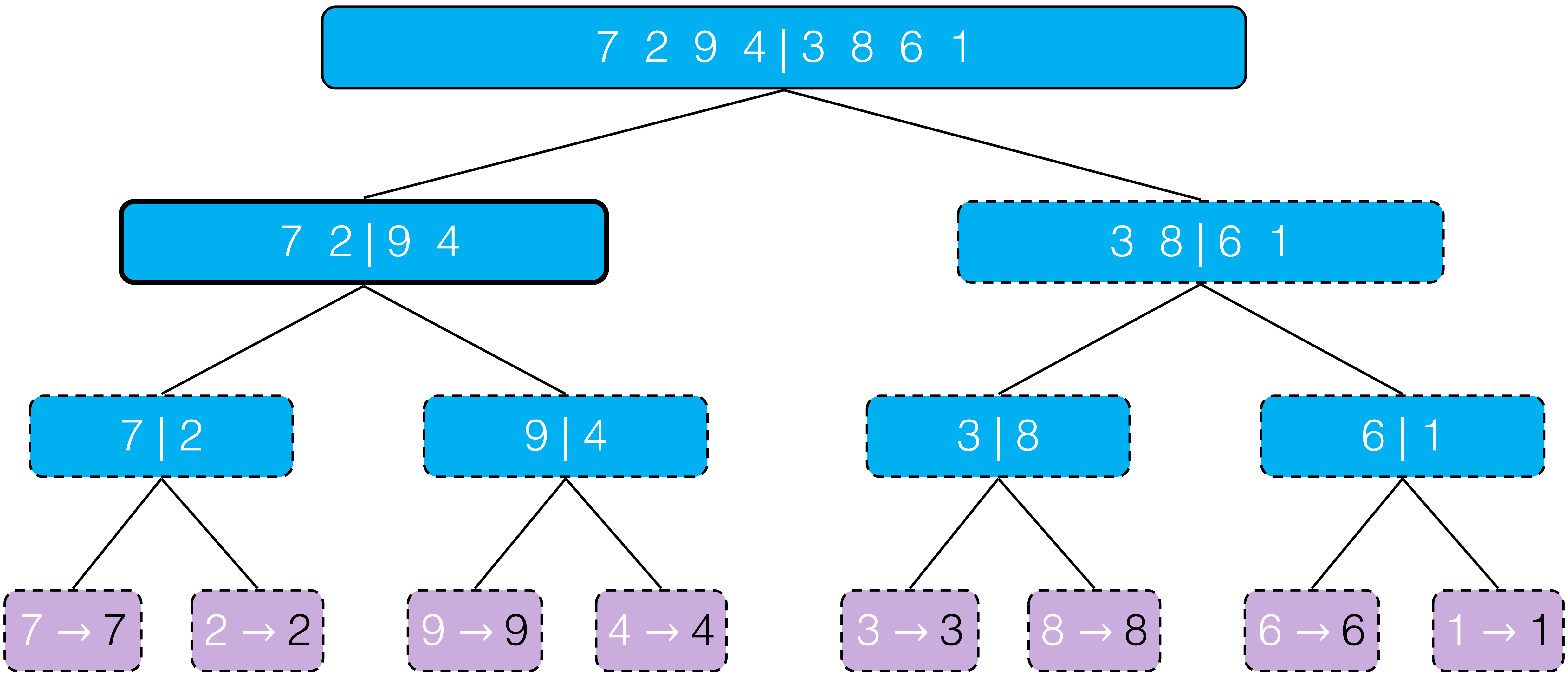
Execution Example

- Recursive call, partition



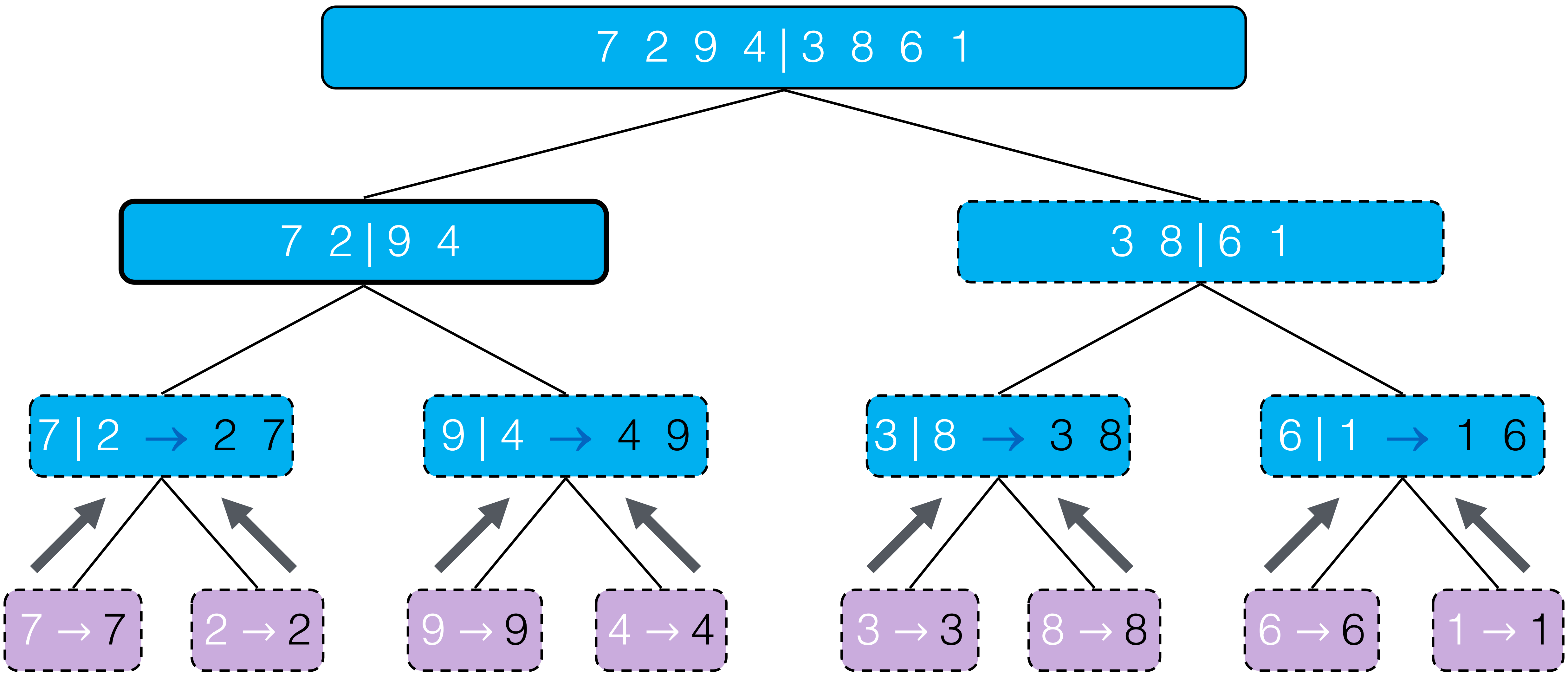
Execution Example

- Recursive call, Base Case



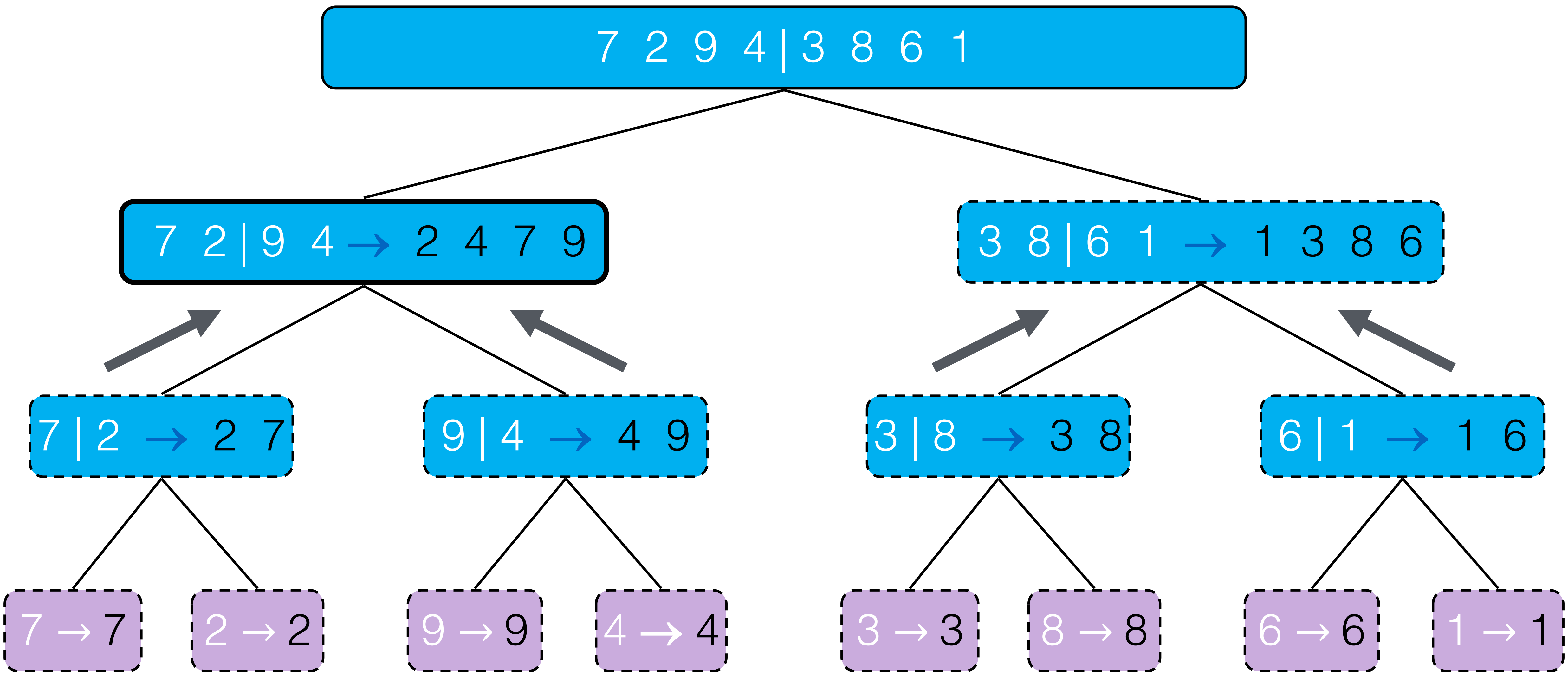
Execution Example

- Merge



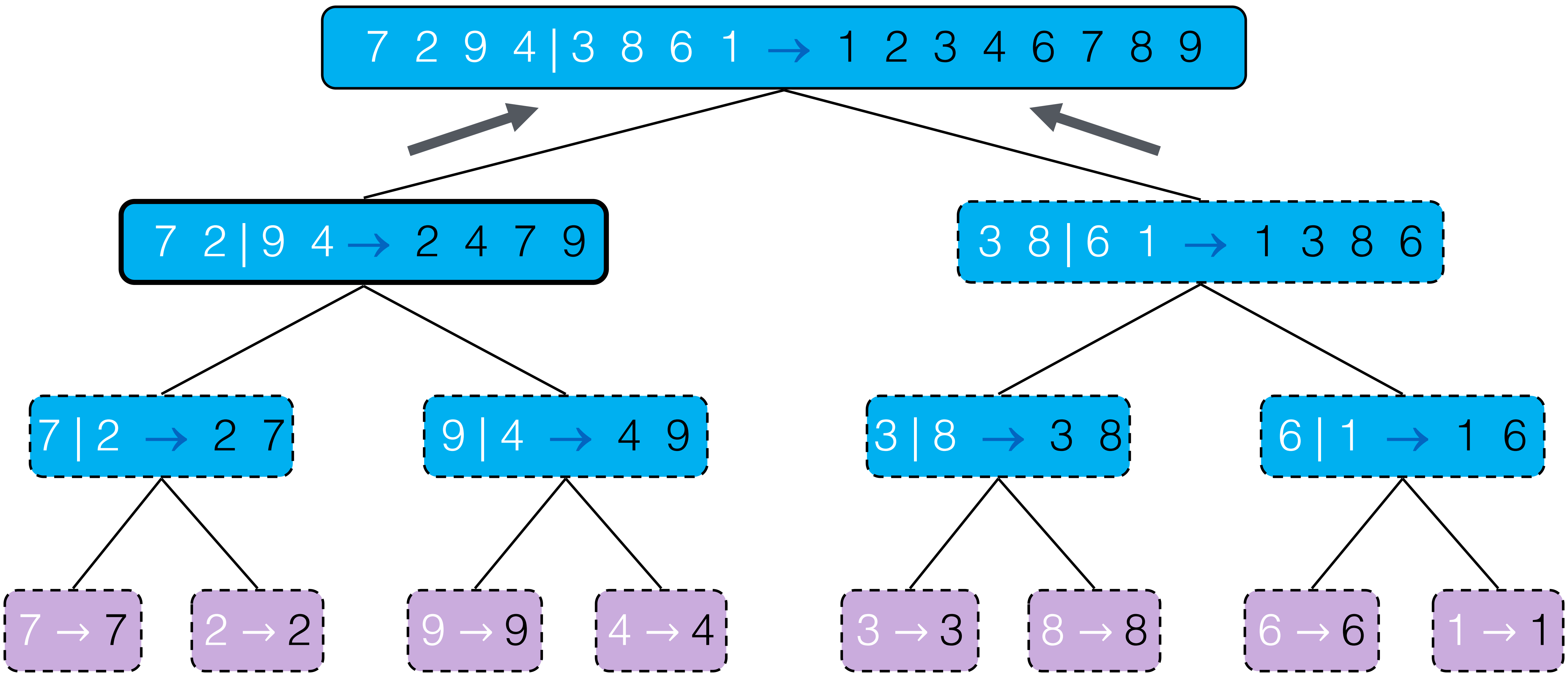
Execution Example

- Merge



Execution Example

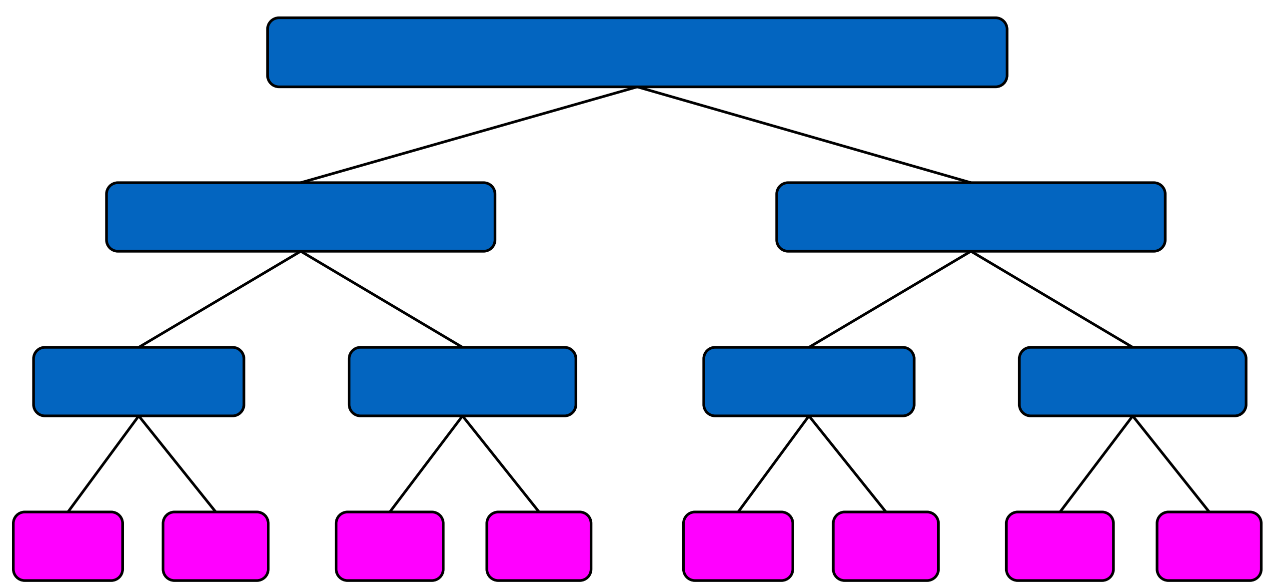
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $\log n$
- The overall amount or work done at the nodes of depth i is $\Theta(n)$
- Thus, the total running time of merge-sort is ?

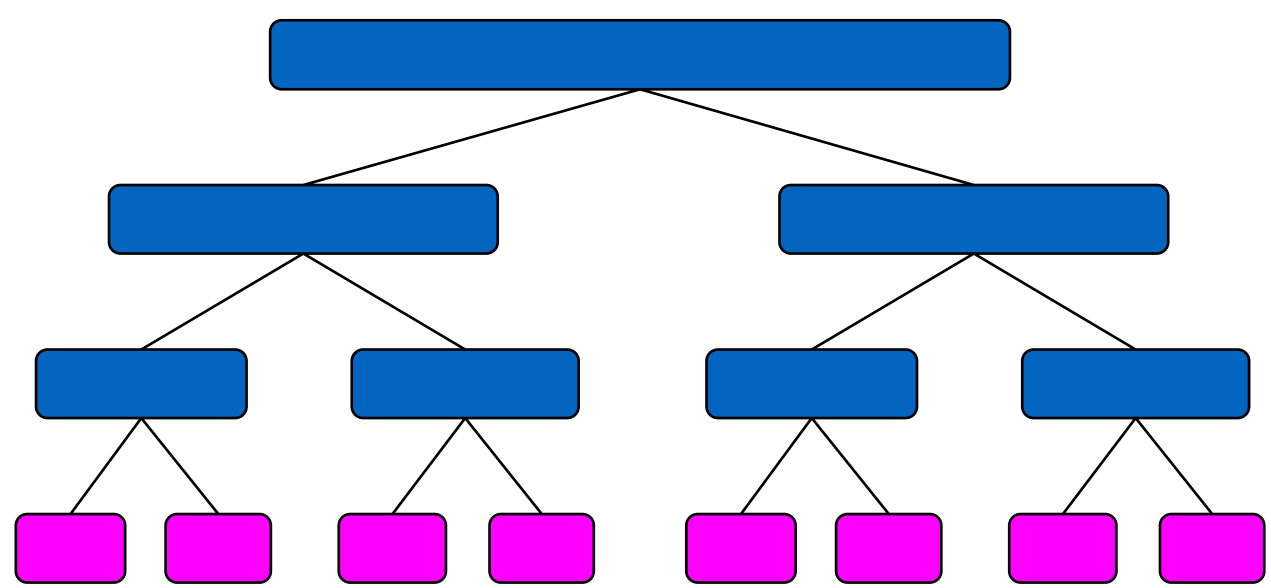
depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Analysis of Merge-Sort

- The height *h* of the merge-sort tree is $\log n$
- The overall amount or work done at the nodes of depth i is $\Theta(n)$
- Thus, the total running time of merge-sort is $\Theta(n \log n)$

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



Analysis of Merge-Sort

- Using Master Theorem

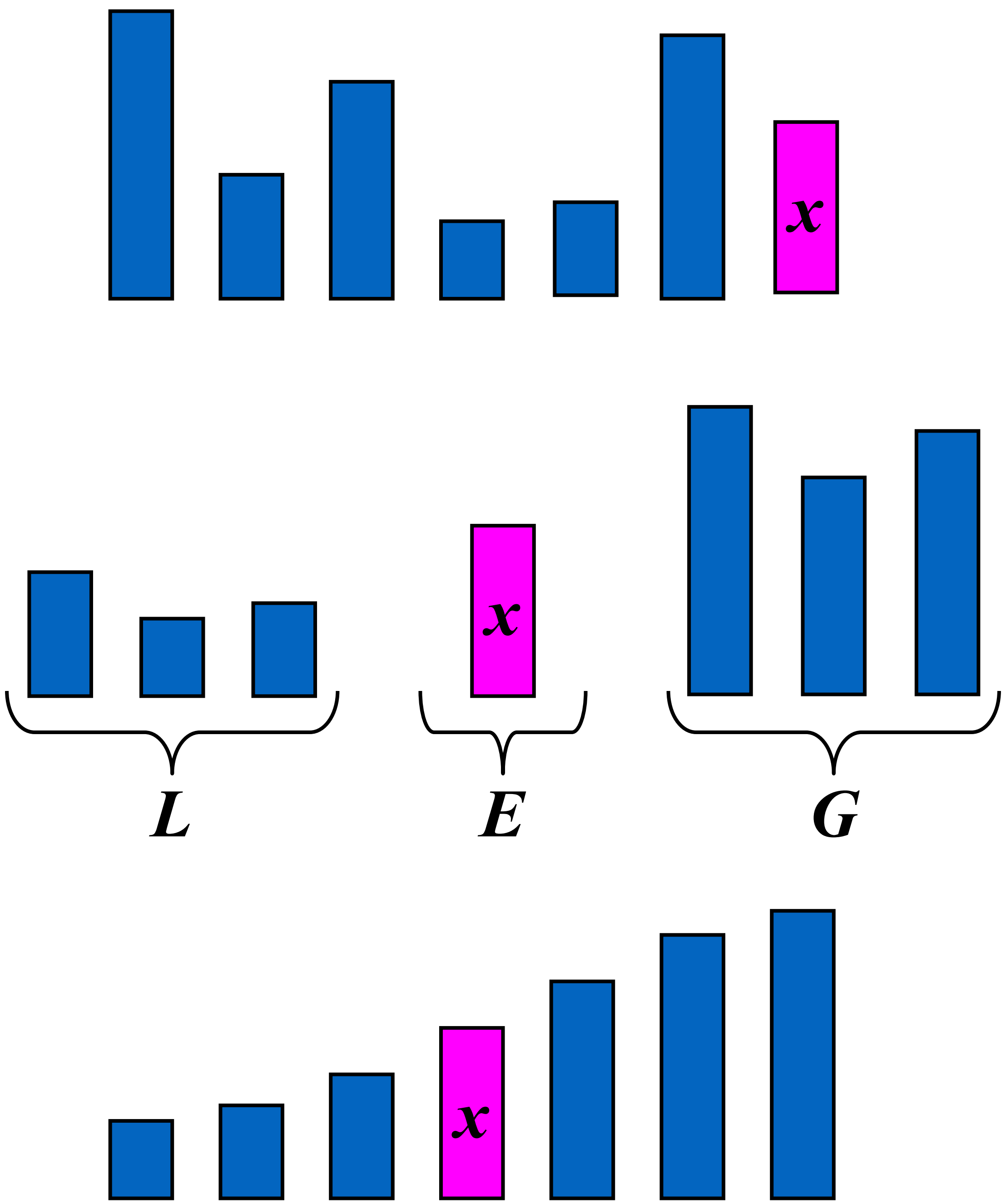
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 , \end{cases}$$

- Case 2 applies
- Thus, the total running time of merge-sort is $\boldsymbol{\Theta(n \log n)}$

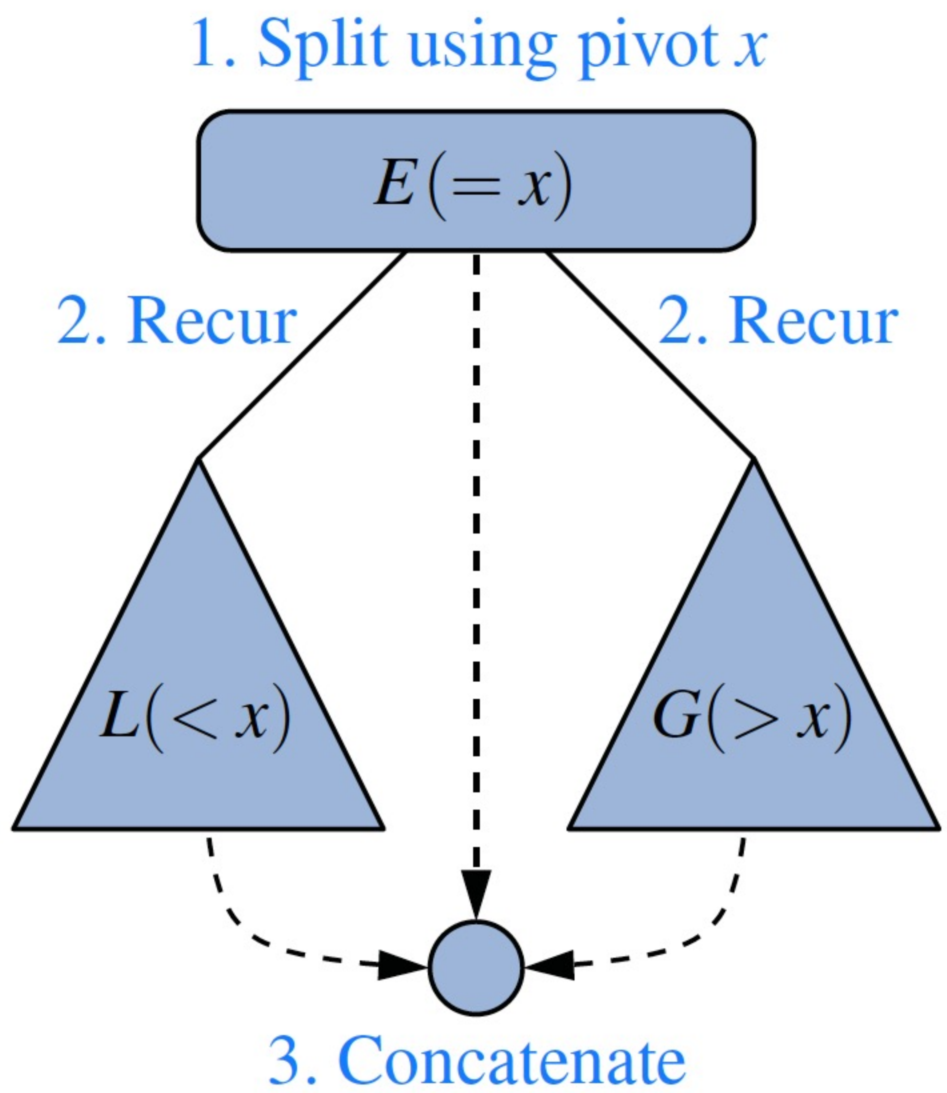
Quick-Sort

Quick-sort

- Quick-sort is a sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick the last element x (called pivot) and
  - partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: Quicksort L and G
  - Conquer: join L, E and G



Quick-sort



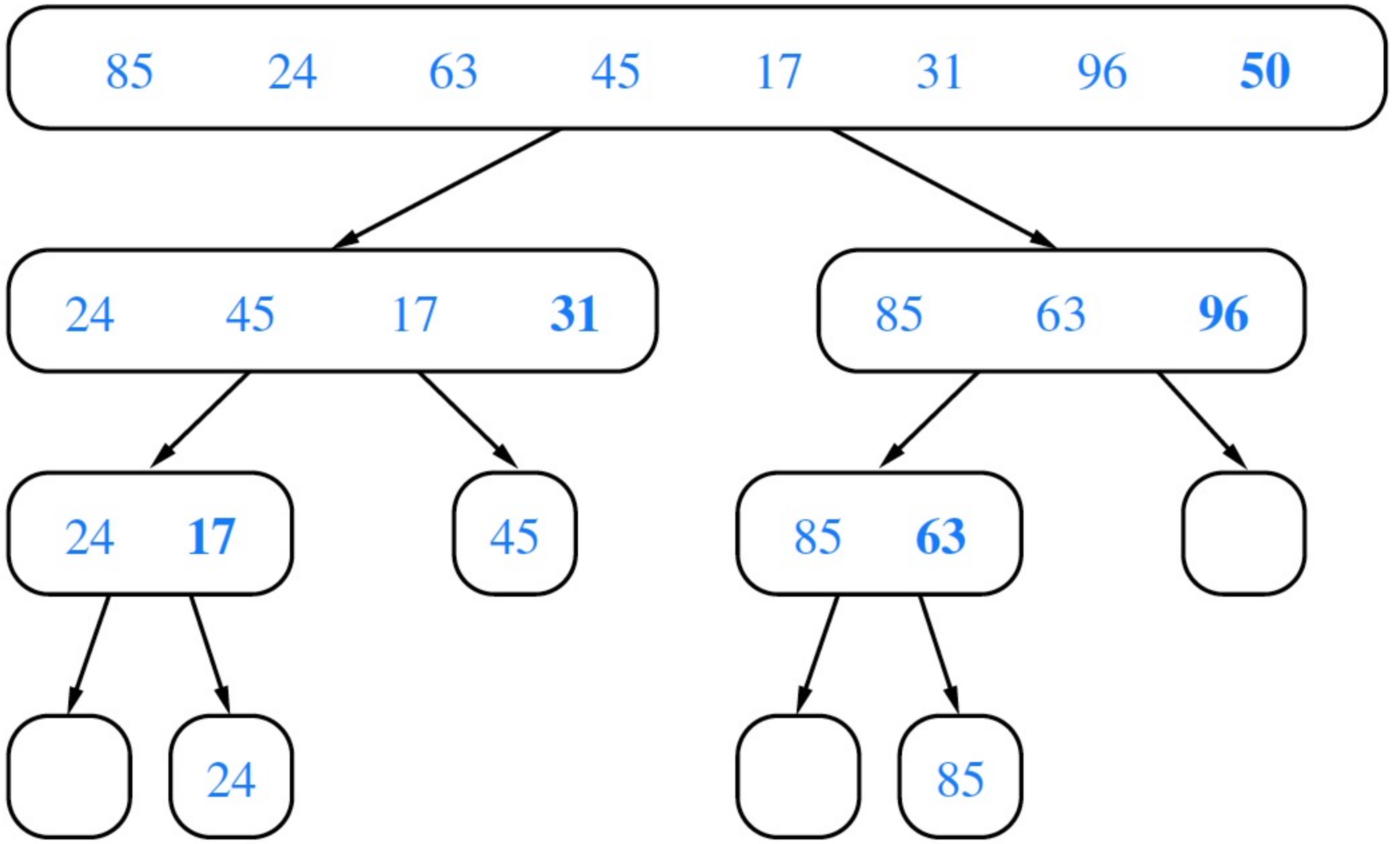
A visual schematic of the quick-sort algorithm – Goodrich, Ch: 12

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 ***constant*** time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm *partition*(S)
  Input sequence S
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.removeLast()$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.addLast(y)$
    else if $y = x$
      $E.addLast(y)$
    else $\{ y > x \}$
      $G.addLast(y)$
  return L, E, G

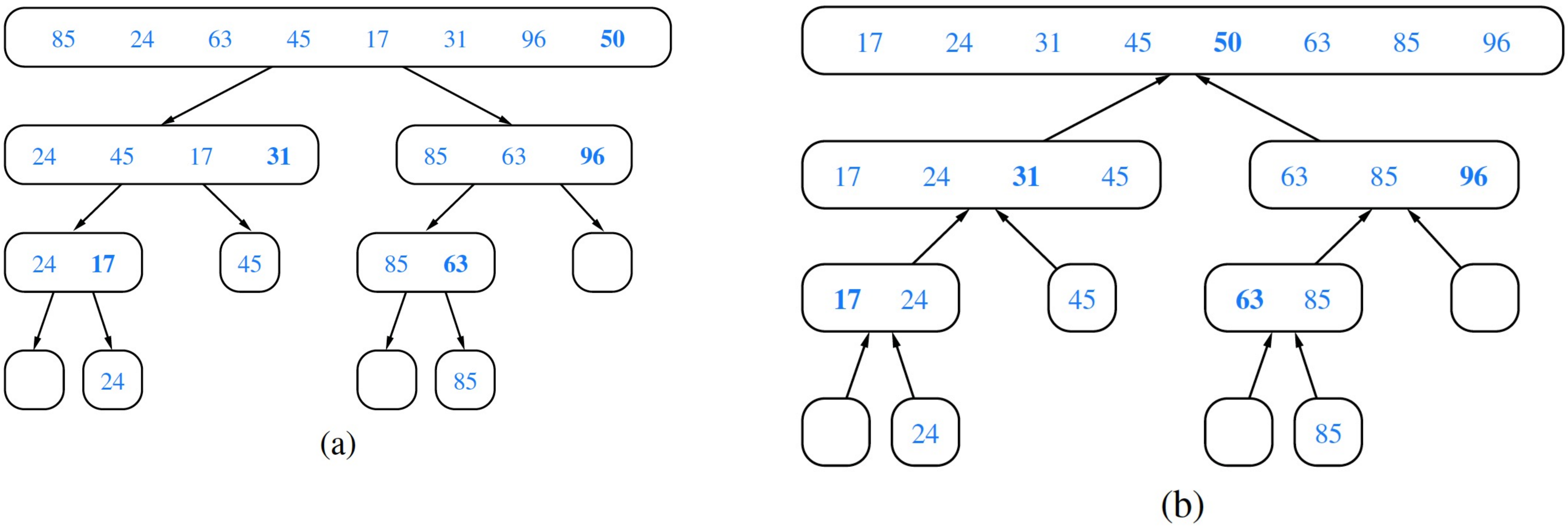
Quick-sort



(a)

Quick-sort tree T for an execution of the quick-sort algorithm on a sequence with 8 elements: (a) input sequences processed at each node of T – Goodrich, Ch: 12;

Quick-sort



(b) output
sequences generated at each node of T.
– Goodrich, Ch: 12;

Java Implementation

```java
/** Quick-sort contents of a queue. */
public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
  int n = S.size();
  if (n < 2) return;                                // queue is trivially sorted
  // divide
  K pivot = S.first();                              // using first as arbitrary pivot
  Queue<K> L = new LinkedQueue<>();
  Queue<K> E = new LinkedQueue<>();
  Queue<K> G = new LinkedQueue<>();
  while (!S.isEmpty()) {                            // divide original into L, E, and G
    K element = S.dequeue();
    int c = comp.compare(element, pivot);
    if (c < 0)                                      // element is less than pivot
      L.enqueue(element);
    else if (c == 0)                                // element is equal to pivot
      E.enqueue(element);
    else                                            // element is greater than pivot
      G.enqueue(element);
  }
  // conquer
  quickSort(L, comp);                               // sort elements less than pivot
  quickSort(G, comp);                               // sort elements greater than pivot
  // concatenate results
  while (!L.isEmpty())
    S.enqueue(L.dequeue());
  while (!E.isEmpty())
    S.enqueue(E.dequeue());
  while (!G.isEmpty())
    S.enqueue(G.dequeue());
}
```

Goodrich, Ch: 12

Analysis of Quick Sort

- Worst-case Partitioning

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

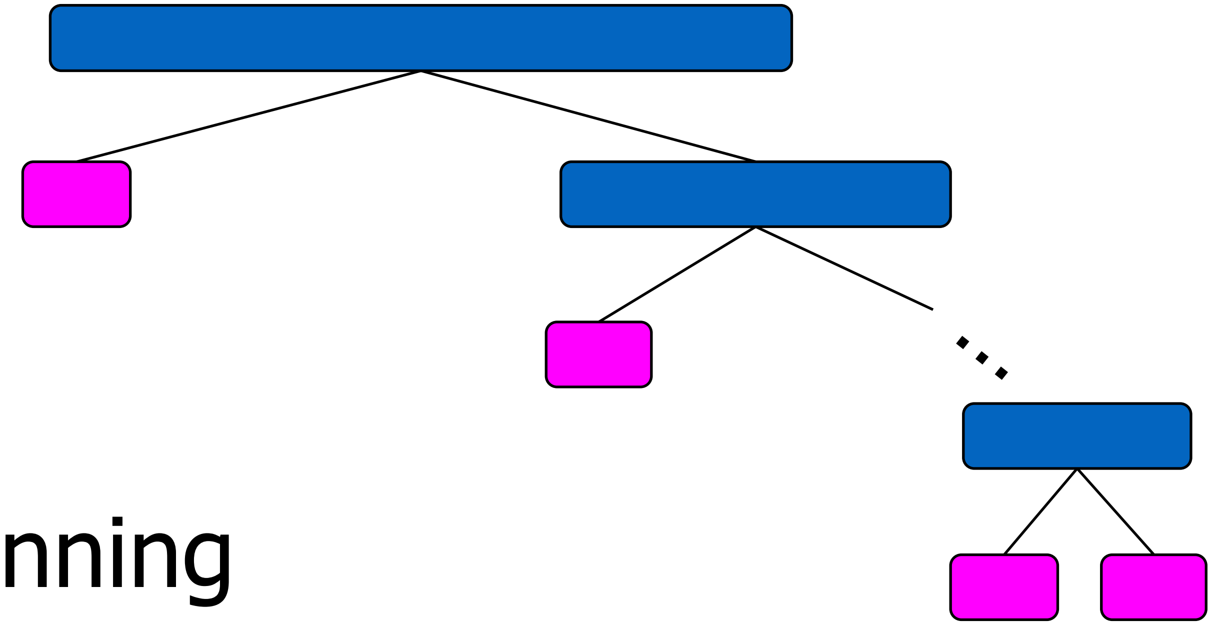
One of L and G has size $n - 1$ and the other has size 0

And this happens at every recursive call

Analysis of Quick Sort

- Worst-case Partitioning

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) .
\end{aligned}
$$



Thus, the worst-case running time of quick-sort is $O(n^2)$

Analysis of Quick Sort

- Best-case Partitioning

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

The best case for quick-sort occurs when the pivot is the middle element

Both of $\boldsymbol{L}$ and $\boldsymbol{G}$ has size $\boldsymbol{n/2}$

And this happens at every recursive call

Summary Analysis of Quick Sort

- Worst-case Partitioning

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

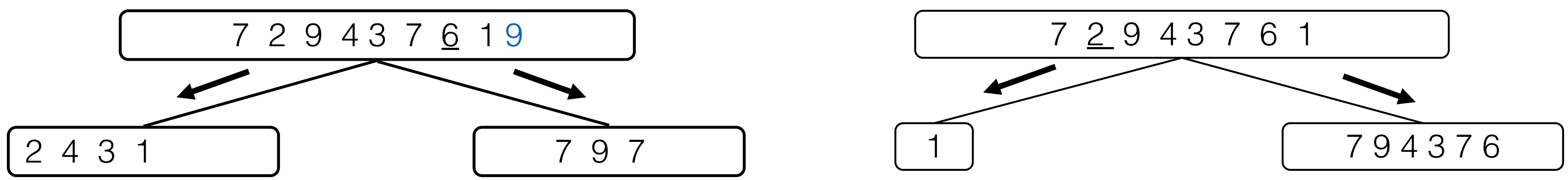
$O(n^2)$

- Best-case Partitioning

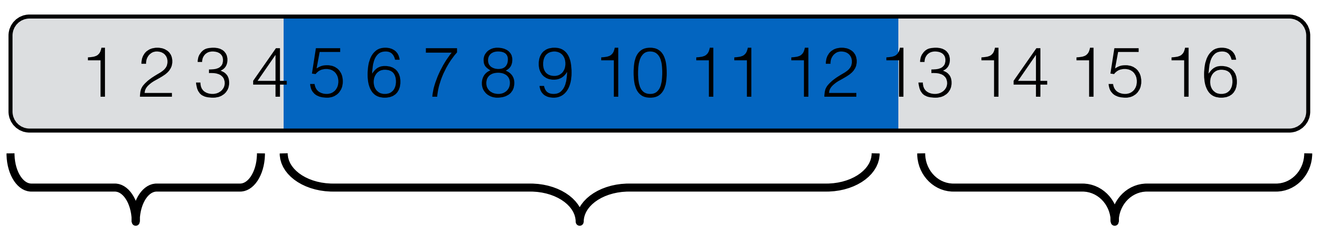
$$T(n) = 2T(n/2) + \Theta(n) ,$$

$O(nlogn)$

Expected Running Time



Good call **Bad call**



Bad pivots **Good pivots** **Bad pivots**

A call is good with probability $1/2$
1/2 of the possible pivots cause good calls

The expected running time of randomized quick-sort on a sequence S of size n is $O(n \log n)$.

Analysis of Randomized Quick-sort

- Proof is available in Cormen's book
- Chapter 7, Section 7.4
- Based on probability theory (Indicator Random Variable and Expectation)
- Interested students can read it there
- Won't be part of the evaluation

Comparison Sorts

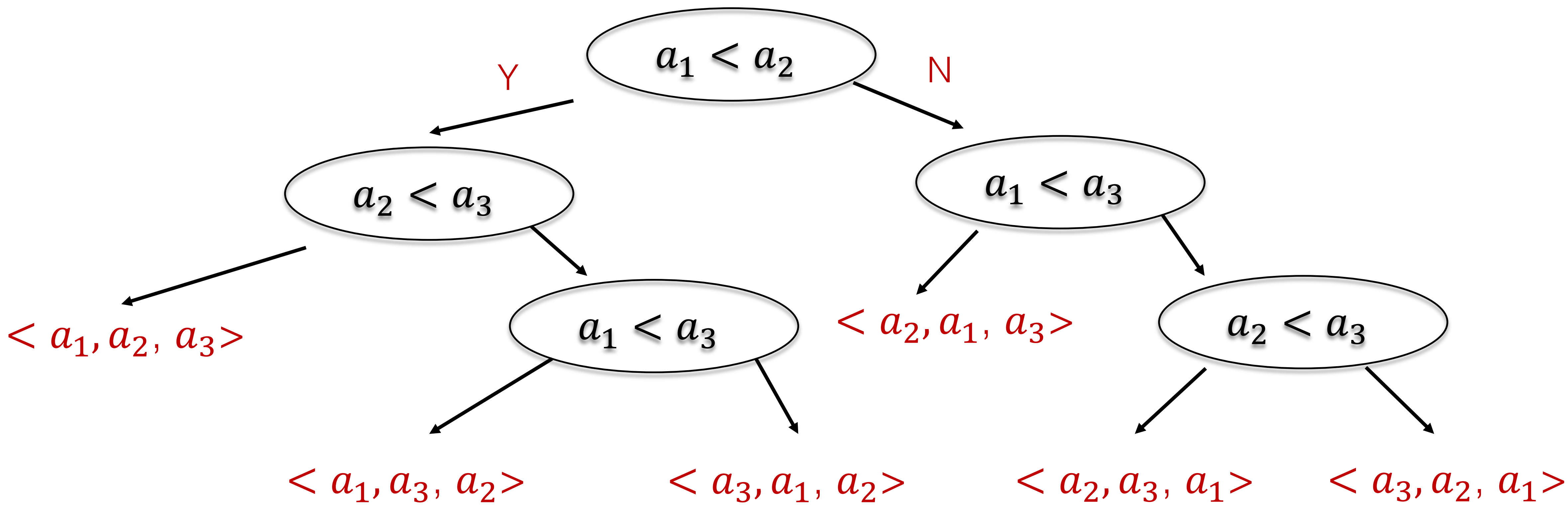
- All sorting algorithms that we have seen so far use only comparisons to gain information about the input.

- We will now see that such algorithms have to do $\Omega(nlogn)$ comparisons

- To do this, we will use a formal model

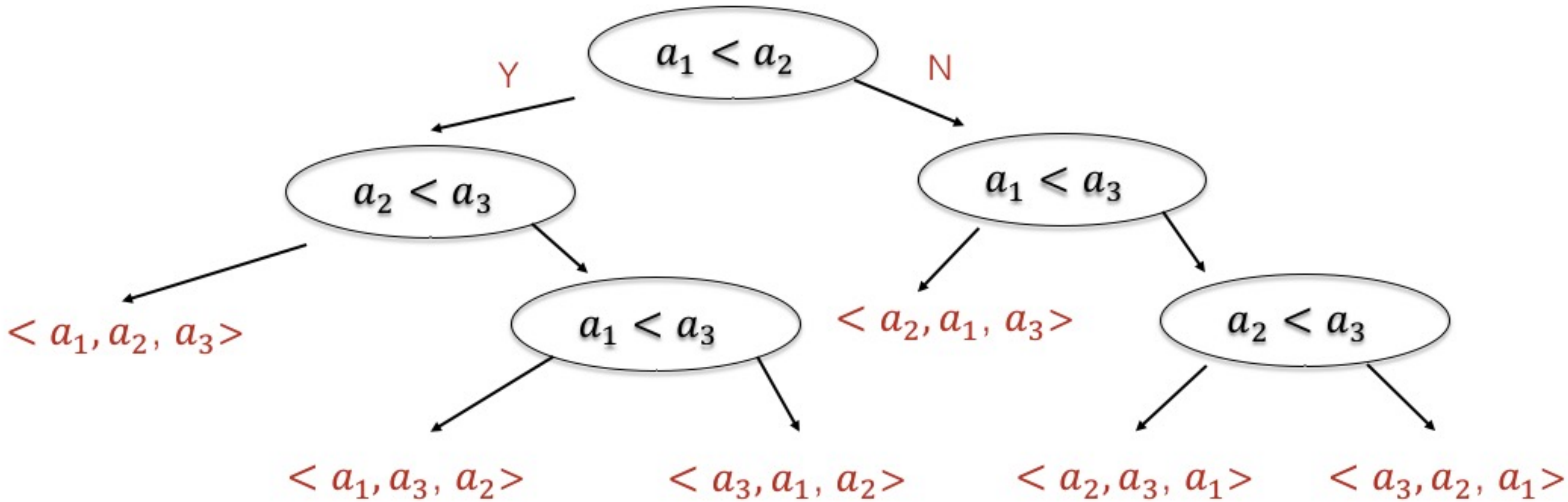
The Decision Tree Model

- This example is for three elements a_1, a_2, a_3



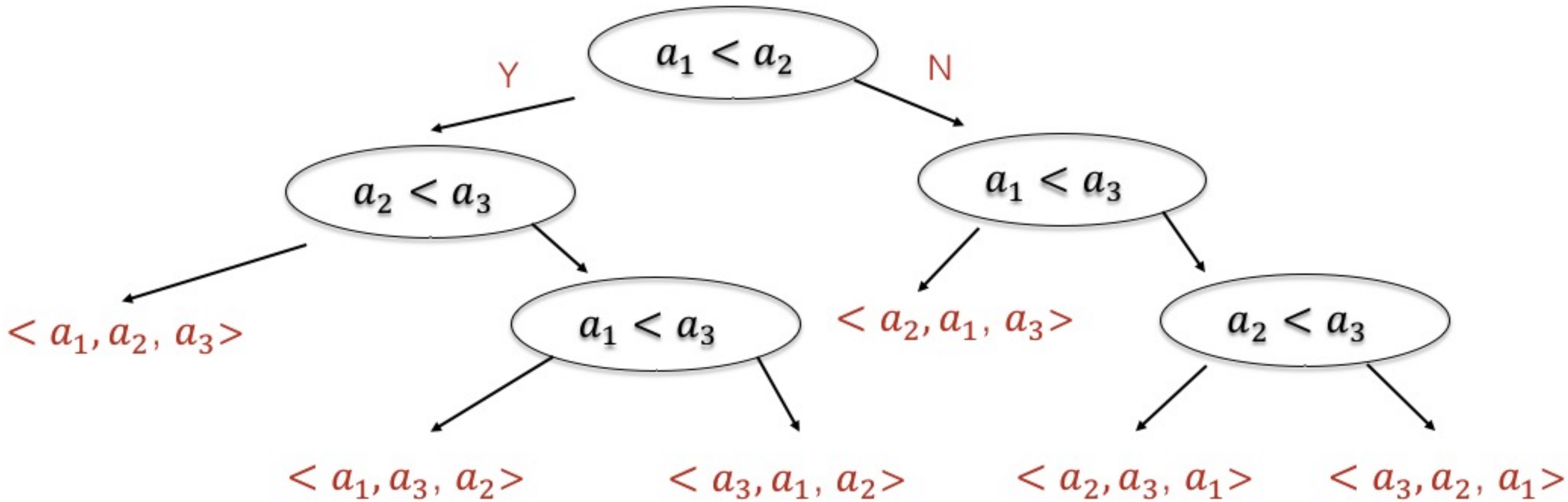
Therefore, lower bound on height $\Rightarrow$ lower bound on sorting.

Comparison Sorts



- For n distinct elements, how many permutations are there?

Comparison Sorts

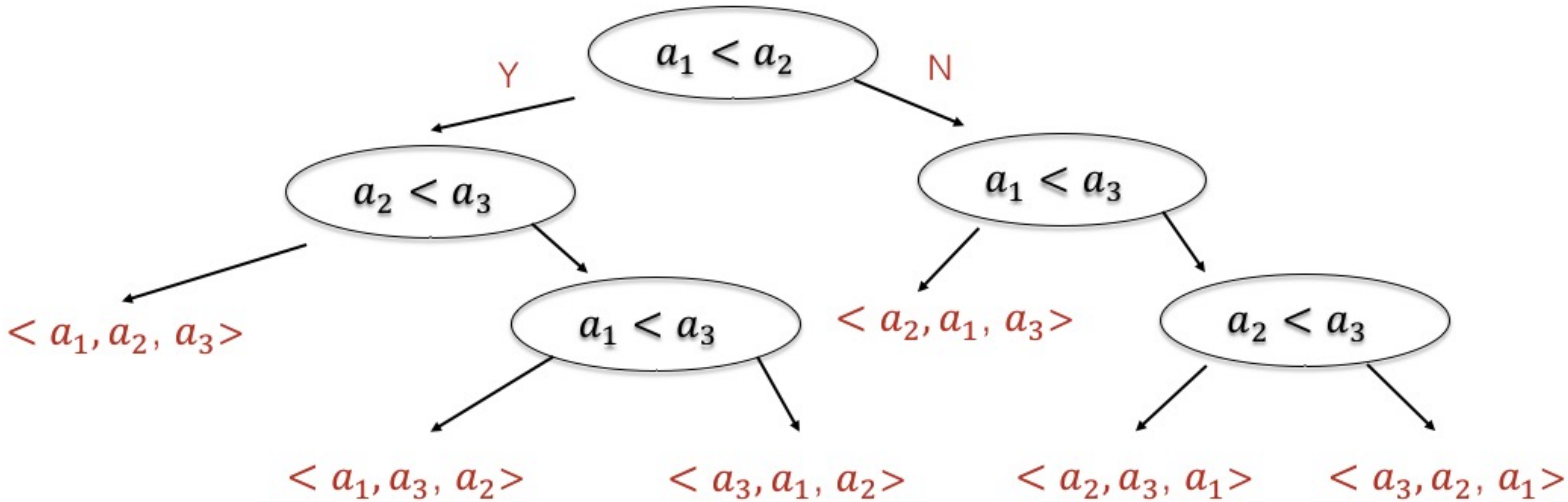


- Thus, there must be at least $n!$ leaves

Another Fact!

- A binary tree of height h has no more than 2^h leaves

Comparison Sorts



- Therefore, $2^h \geq n!$

Comparison Sorts

$$2^h \geq n! \Rightarrow h \geq log(n!)$$

$$= log(n(n-1)(n-2)\cdots(2))$$

$$= log(n) + log(n-1) + \cdots + log(2)$$

$$= \sum_{i=2}^{n} log i$$

$$= \sum_{i=2}^{\frac{n}{2}-1} log i + \sum_{i=\frac{n}{2}}^{n} log i$$

Comparison Sorts

$$2^h \geq n! \Rightarrow h \geq log(n!)$$

$$= log(n(n-1)(n-2)\cdots(2))$$

$$= log(n) + log(n-1) + \cdots + log(2)$$

$$= \sum_{i=2}^{n} logi$$

$$= \sum_{i=2}^{\frac{n}{2}-1} logi + \sum_{i=\frac{n}{2}}^{n} logi$$

$$\geq \sum_{i=\frac{n}{2}}^{n} log\frac{n}{2}$$

$$= \frac{n}{2}.log\frac{n}{2}$$

$$= \Omega(nlogn)$$

Did we achieve today's objectives?

- What is sorting?
- Why must one learn about sorting algorithms in this course?
- Properties of sorting algorithms
- Sorting Algorithms
  - ❖ Bubble Sort, Selection Sort , Insertion Sort
  - ❖ Merge Sort, Quick Sort