



# CPTS 223 Advanced Data Structure C/C++

---

Sorting

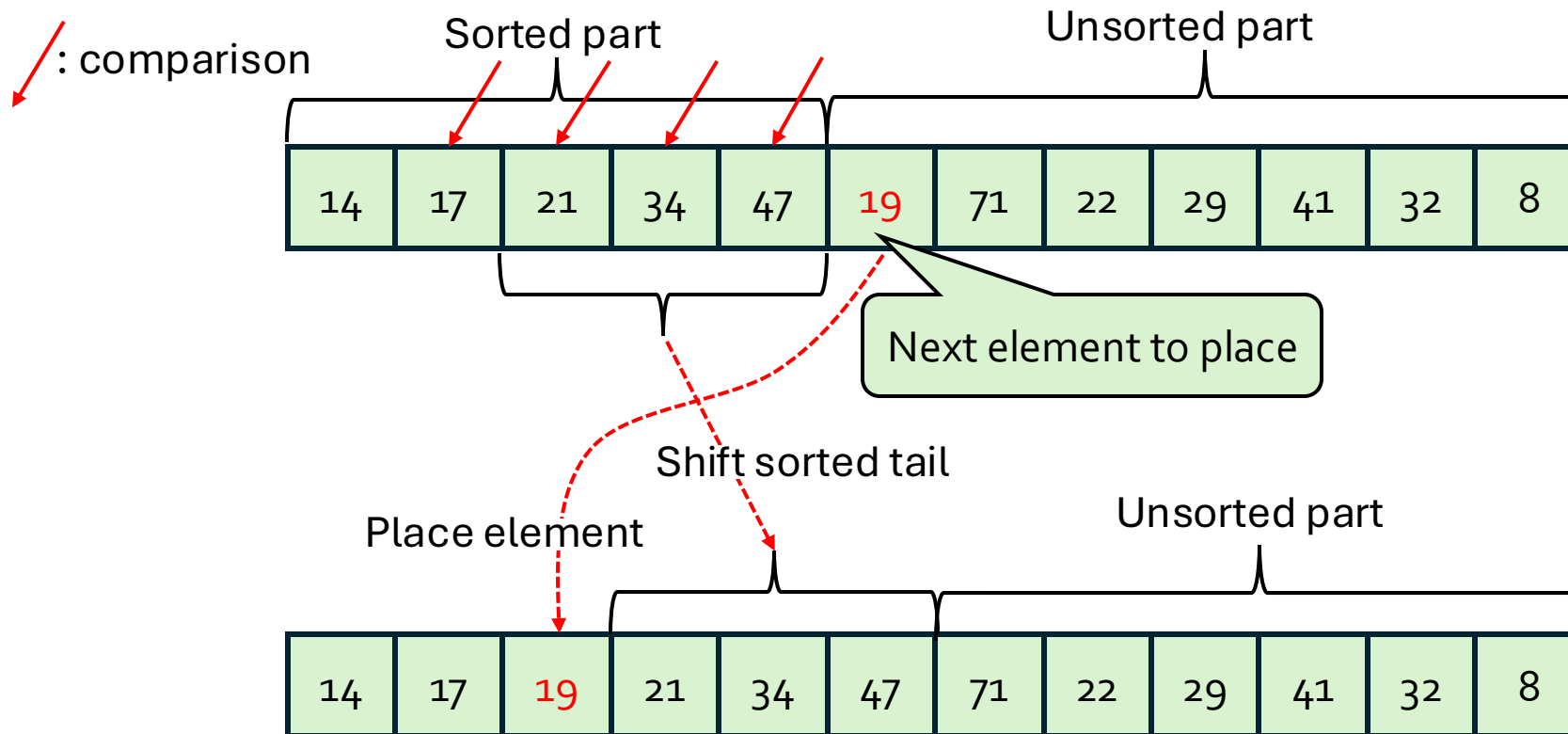
# Overview: sorting methods

---

- Comparison based sorting
  - $O(n^2)$  methods
    - e.g., insertion sort, bubble sort
  - Average time  $O(n \log(n))$  methods
    - e.g., quick sort
  - $O(n \log(n))$  methods
    - e.g., merge sort, heap sort
- Non-comparison-based sorting
  - Integer sorting: linear time
    - e.g., counting sort
  - Radix sort, bucket sort
- Stable v.s. non-stable sorting

Sorting

# Insertion sort: at a give iteration



Insertion sort:  
Worst-case time  
complexity:  $\Theta(n^2)$   
Best-case time  
complexity:  $\Theta(n)$

# Divide and conquer technique

---

- Input: A problem of size  $n$
- Recursive
- At each level of recursion:
  - (Divide)
    - Split the problem of size  $n$  into a fixed number of sub-problems of smaller sizes, and solve each sub-problem recursively
  - (Conquer)
    - Merge the answers to the sub-problems

Sorting

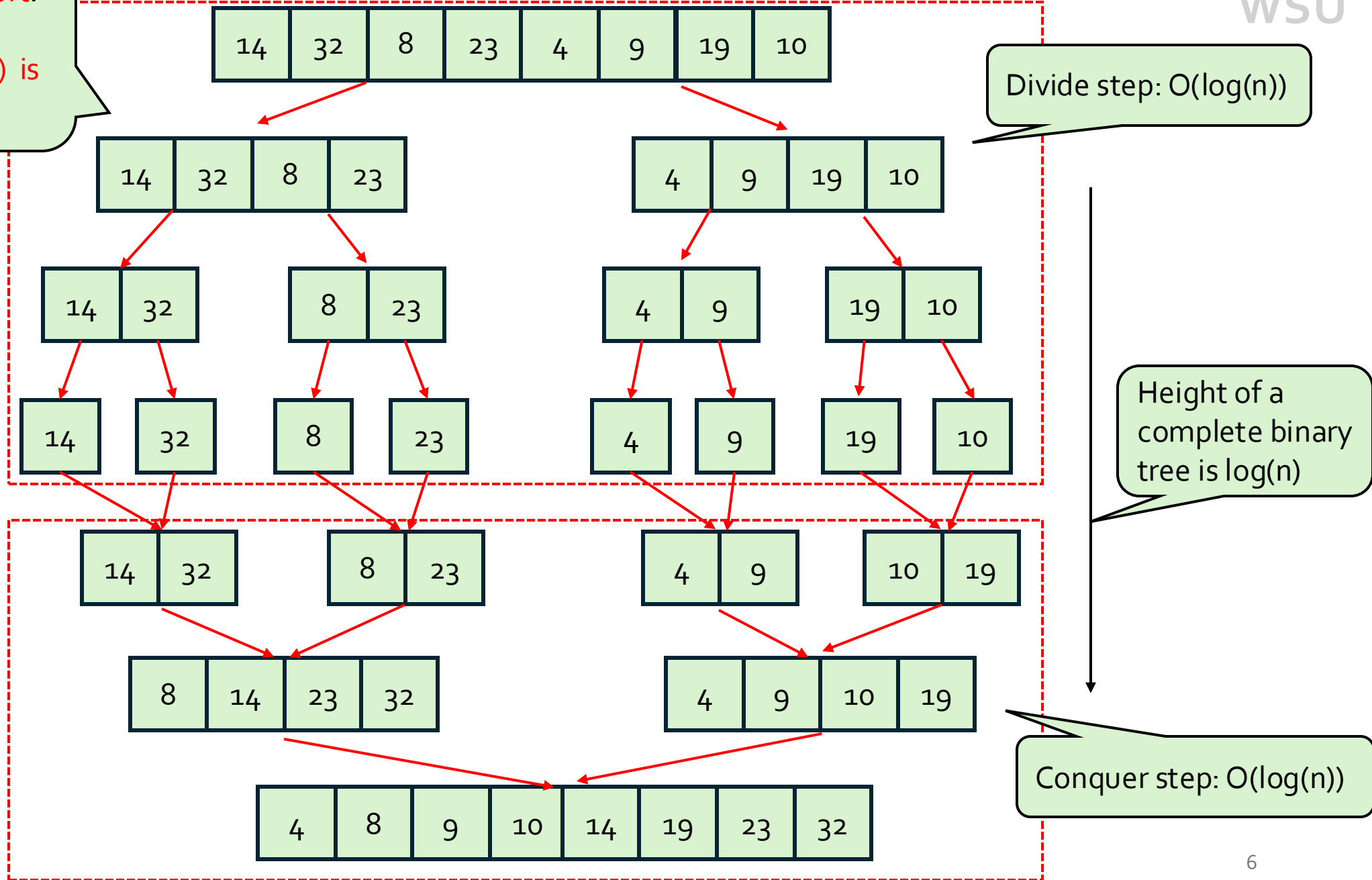
# Two divide and conquer sorts

---

- Mergesort
  - Divide is trivial
  - Merge (i.e, conquer) does all the work
- Quicksort
  - Partition (i.e, Divide) does all the work
  - Merge (i.e, conquer) is trivial

Main idea of **mergesort**:

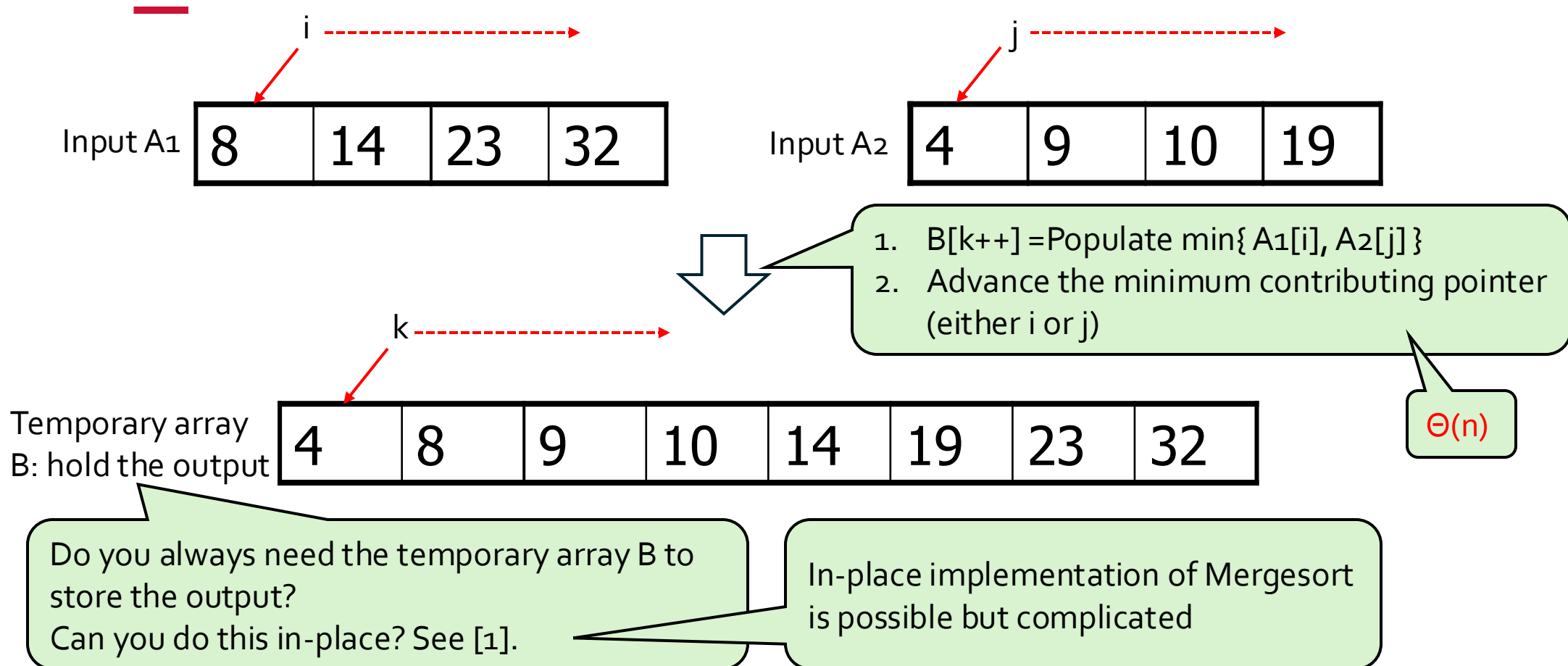
- Dividing is trivial
- **Merging (conquer) is non-trivial**



Sorting

we assume the **two input arrays are already sorted**

# Mergesort: merge two arrays



# Mergesort: analysis

- Mergesort takes  $\Theta(n \log(n))$  time
- Proof:
  - Let  $T(n)$  be the time taken to merge sort  $n$  elements
  - Time for each comparison operation:  $T(1) = O(1)$
  - Main observation:
    - To merge two sorted arrays of size  $n/2$ , it takes  $n$  comparisons at most.
  - Therefore:
    - $T(n) = 2 T(n/2) + n$
  - Final result:  $T(n) = \Theta(n \log(n))$

Solving the recurrence:

$$\begin{aligned}
 T(n) &= T(n/2) + T(n/2) + n \\
 &= 2T(n/2) + n \\
 &= 2 ( 2 T(n/4) + n/2 ) + n \\
 &= 4T(n/4) + n + n \\
 &= 8 T(n/8) + (1+1+1) n \\
 &= 2^K T\left(\frac{n}{2^K}\right) + n K \\
 &\quad \text{Let } \frac{n}{2^K} = 1 \\
 &= n T(1) + n * \log(n)
 \end{aligned}$$



## Sorting

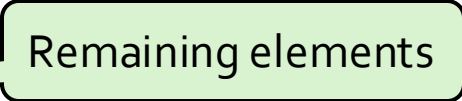
# Quicksort

---

- Divide-and-conquer approach to sorting
- Like MergeSort, except:
  - Not divide the array in half
  - Partition the array-based elements being less than or greater than some element of the array (the **pivot**)
  - i.e., **divide phase does all the work**; merge phase is trivial.
- **Worst case running time  $O(n^2)$**
- **Average case running time  $O(n \log(n))$**
- Fastest generic sorting algorithm in practice
- Even faster if use simple sort (e.g., InsertionSort) when array becomes small

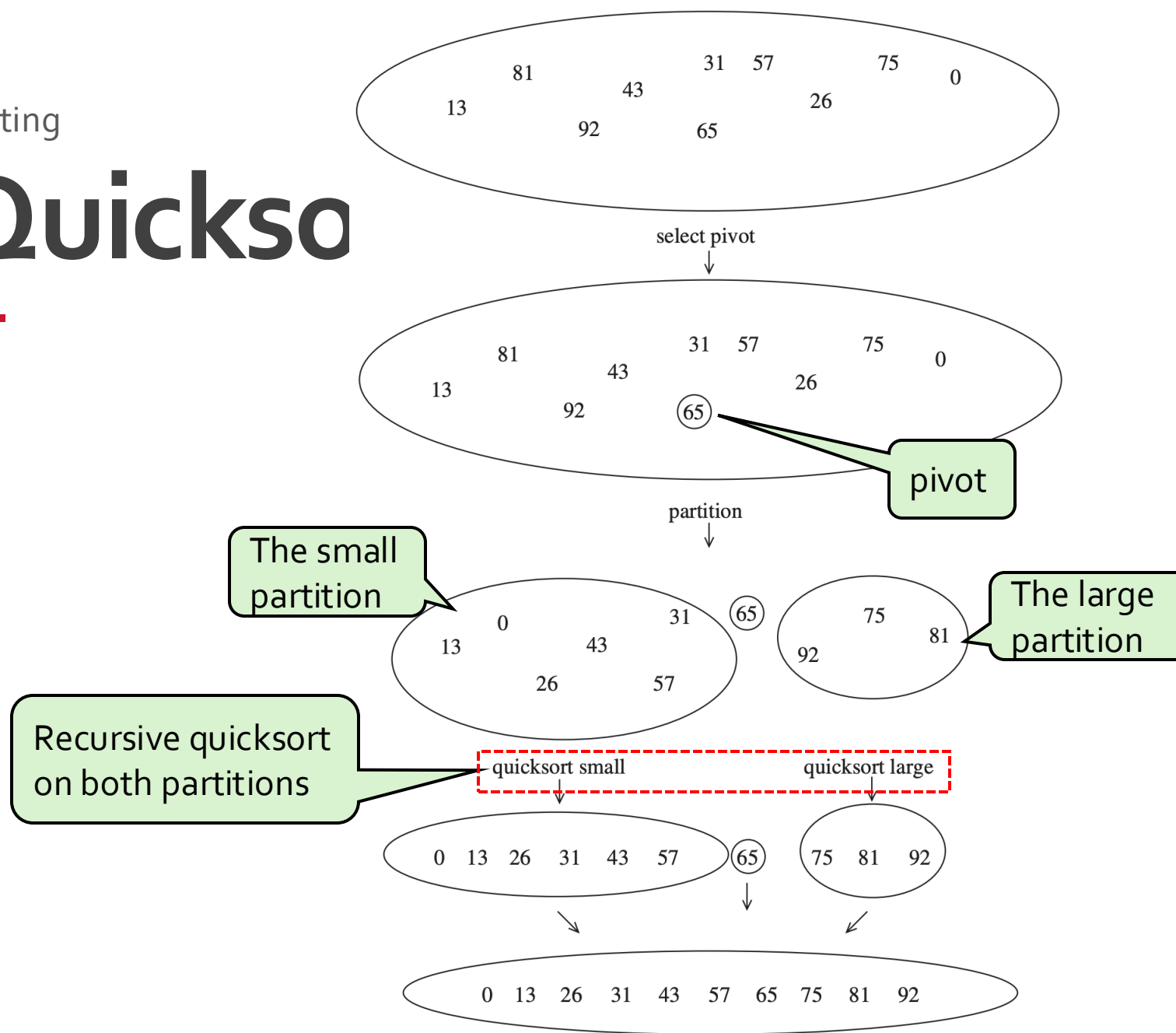
# Quicksort algorithm

---

- **QuickSort**( Array: S)
  1. If size of S is 0 or 1, return
  2. **Pivot** = Pick an element v in S 
  3. Partition  **$S - \{v\}$**  into two disjoint groups
    - $S_1 = \{x \in (S - \{v\}) \mid x < v\}$
    - $S_2 = \{x \in (S - \{v\}) \mid x > v\}$
  4. Return:  
**QuickSort**(S<sub>1</sub>), v, **QuickSort**(S<sub>2</sub>)

Sorting

# Quicksort



**Figure 7.14** The steps of quicksort illustrated by example

# Mergesort v.s. quicksort

---

- Main problem with quicksort:
  - **Worst-case**: dividing the input array into subproblems of **size 1 and N-1** in the worst case at every recursive step
  - unlike merge sort which always divides into **two halves**
  - Leading to  **$O(N^2)$**  performance
- → Need to choose pivot wisely (but efficiently)
- MergeSort is typically implemented using a temporary array (for merge step)
  - QuickSort can partition the array “in place”

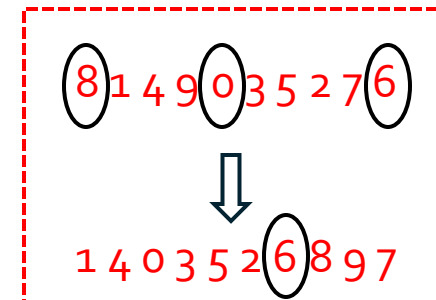
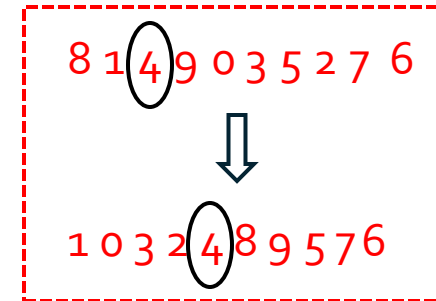
# Quicksort: picking the pivot

---

- How about choosing the first element?
  - What if array already or nearly sorted?
  - Good for a randomly populated array
- How about choosing a random element?
  - Good in practice if “truly random”
  - Still possible to get some bad choices
  - Requires execution of random number generator

# Quicksort: picking the pivot

- Best choice of pivot:
  - Median of array
- But **median is expensive to calculate**
- A practical strategy: Approximate the median
  - Estimate median as the median of any three elements
  - Median = **median {first, middle, last}** Has been shown to reduce
  - running time (comparisons) by 14%



# Quicksort: partition strategy

---

- Goal of partitioning:
  - i) Move all (elements < pivot) to the left of pivot
  - ii) Move all (elements > pivot) to the right of pivot
- Partitioning is conceptually straightforward, but easy to do inefficiently
- One bad way:
  - Do one pass to figure out how many elements should be on either side of pivot
  - Then create a temp array to copy elements relative to pivot

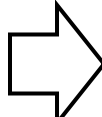
# Quicksort: partition strategy

- A good strategy to do partition : do it **in place**

all operations are done in place of the input array (i.e., without creating a temporary array)

```
// Swap pivot with last element S[right]
i = left
j = (right - 1) While (i < j) {
// advance i until first element > pivot
// decrement j until first element < pivot
// swap S[i] & S[j] (only if i<j)
Swap ( pivot , S[i] )
```

Designed algorithm



```
Swap pivot with last element S[right]
i = left
j = (right - 1)
while (i < j) {
    { i++; } until S[i] > pivot
    { j--; } until S[j] < pivot
    If (i < j), then swap( S[i] , S[j] )
}
Swap ( pivot , S[i] )
```

Pseudo code



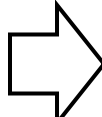
# Quicksort: partition strategy

- A good strategy to do partition : do it **in place**

all operations are done in place of the input array (i.e., without creating a temporary array)

```
// Swap pivot with last element S[right]
i = left
j = (right - 1) While (i < j) {
// advance i until first element > pivot
// decrement j until first element < pivot
// swap S[i] & S[j] (only if i<j)
Swap ( pivot , S[i] )
```

Designed algorithm



```
Swap pivot with last element S[right]
i = left
j = (right - 1)
while (i < j) {
    { i++; } until S[i] > pivot
    { j--; } until S[j] < pivot
    If (i < j), then swap( S[i] , S[j] )
}
Swap ( pivot , S[i] )
```

Pseudo code

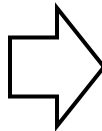
# Quicksort: partition strategy

- A good strategy to do partition : do it **in place**

all operations are done in place of the input array (i.e., without creating a temporary array)

```
// Swap pivot with last element S[right]
i = left
j = (right - 1) While (i < j) {
// advance i until first element > pivot
// decrement j until first element < pivot
// swap S[i] & S[j] (only if i<j)
Swap ( pivot , S[i] )
```

Designed algorithm



```
Swap pivot with last element S[right]
i = left
j = (right - 1)
while (i < j) {
    { i++; } until S[i] > pivot
    { j--; } until S[j] < pivot
    If (i < j), then swap( S[i] , S[j] )
}
Swap ( pivot , S[i] )
```

Pseudo code

Sorting

# Quicksort: partition strategy

- A good strategy to do partition : do it **in place**

(6) 1 4 9 0 3 5 2 7 8



1 4 0 3 5 2 (6) 9 7 8

```
// Swap pivot with last element S[right]
i = left
j = (right - 1)  While (i < j) {
// advance i until first element > pivot
// decrement j until first element < pivot
// swap S[i] & S[j] (only if i<j)
Swap ( pivot , S[i] )
```

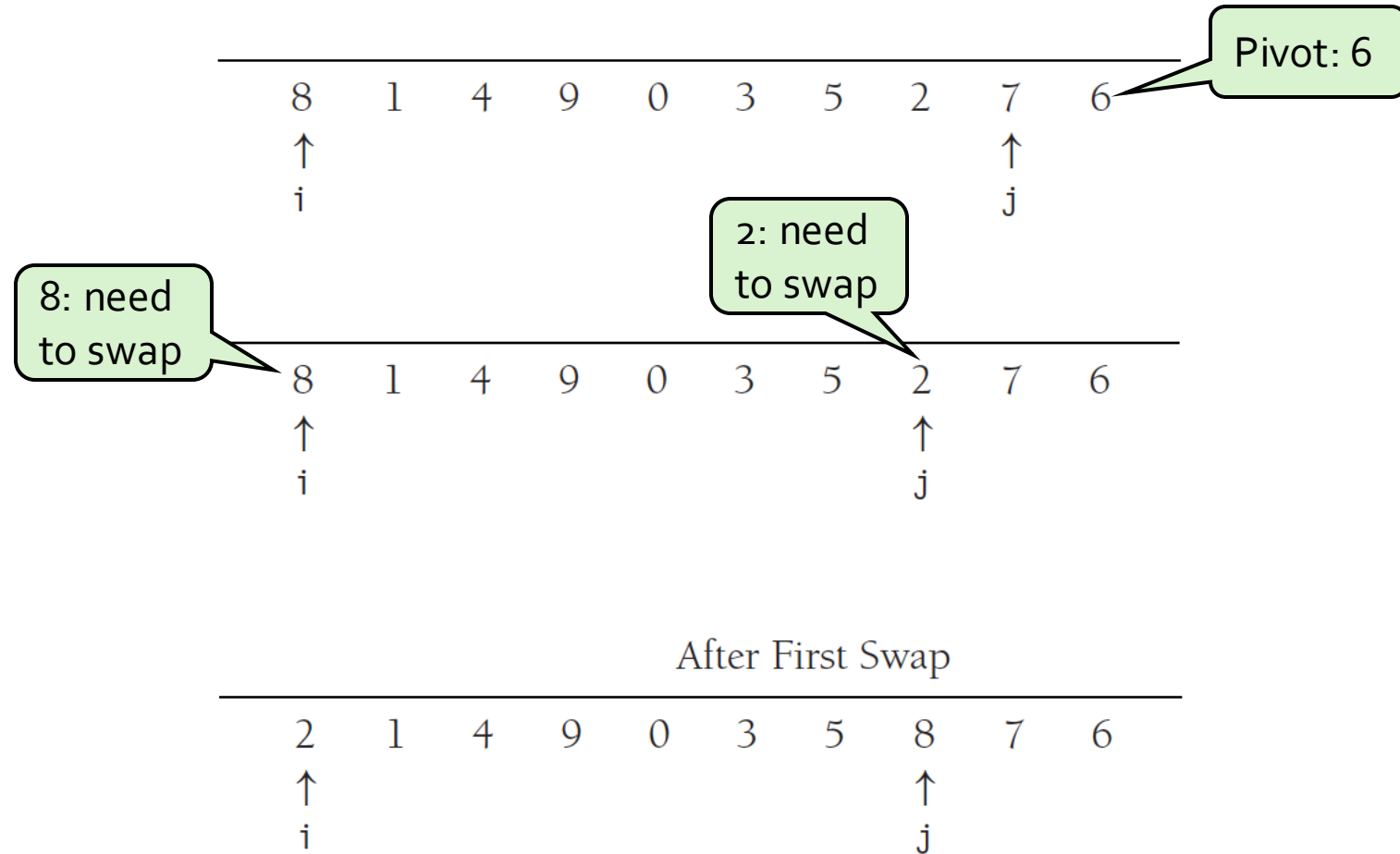
Designed algorithm

```
Swap pivot with last element S[right]
i = left
j = (right - 1)
while (i < j) {
    { i++; } until S[i] > pivot
    { j--; } until S[j] < pivot
    If (i < j), then swap( S[i] , S[j] )
}
Swap ( pivot , S[i] )
```

Pseudo code

Sorting

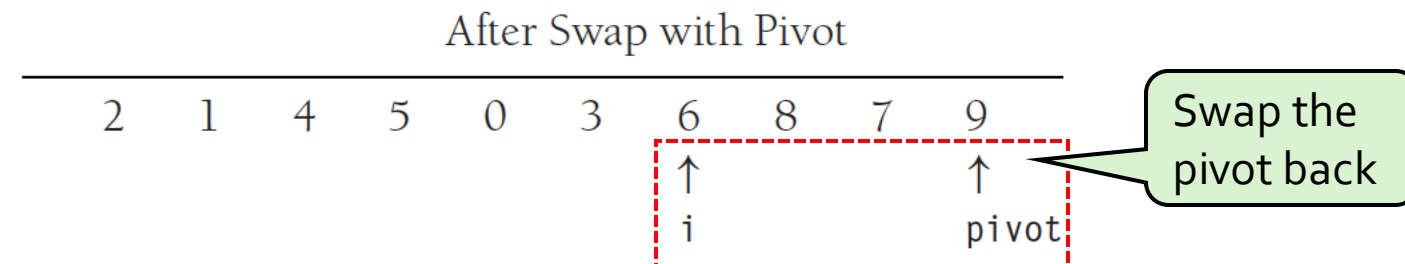
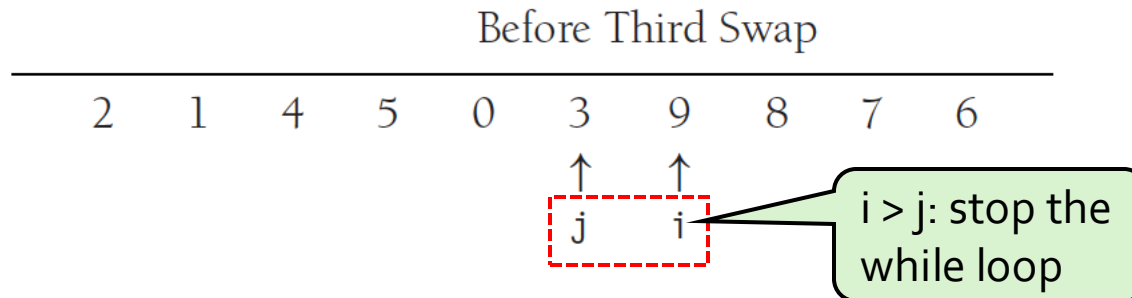
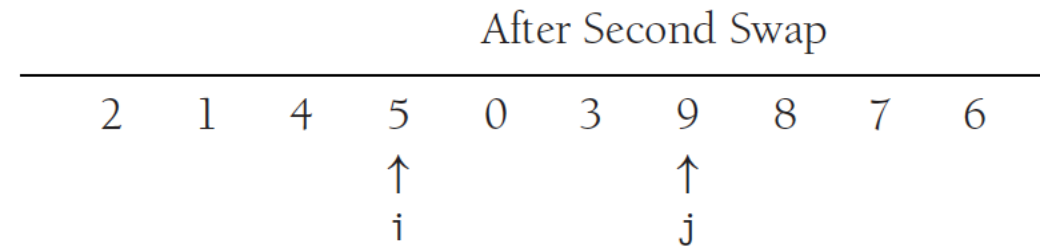
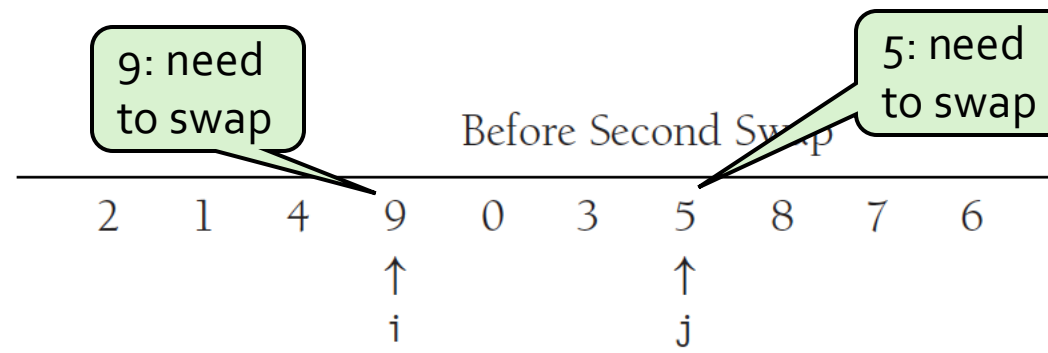
# Quicksort: partition strategy



Sorting

# Quicks

ategy



# Quicksort: handling duplicates

- What if all input elements are equal:  
{6, 6, 6, 6, 6, 6, 6, 6, }
- Current approach:
  - { i++; } until S[i] > pivot
  - { j--; } until S[j] < pivot
- What will happen?
  - i will advance all the way to the right end
  - j will advance all the way to the left end
  - pivot will remain in the right position, creating the left partition to contain N-1 elements and empty right partition
  - → Worst case  $O(N^2)$  performance

```
Swap pivot with last element S[right]
i = left
j = (right - 1)
while (i < j) {
    { i++; } until S[i] > pivot
    { j--; } until S[j] < pivot
    If (i < j), then swap( S[i] , S[j] )
}
Swap ( pivot , S[i] )
```

Biased partition

# Quicksort: handling duplicates

---

- A better code
- Not skip elements equal to pivot
  - { i++; } until  $S[i] \geq \text{pivot}$
  - { j--; } until  $S[j] \leq \text{pivot}$
- Adds some unnecessary swaps
- But results in **perfect partitioning** for array of identical elements

```
Swap pivot with last element S[right]
i = left
j = (right - 1)
while (i < j) {
    { i++; } until S[i] ≥ pivot
    { j--; } until S[j] ≤ pivot
    If (i < j), then swap( S[i] , S[j] )
}
Swap ( pivot , S[i] )
```

# Quicksort: small arrays

---

- When  $S$  is small, recursive calls become expensive (overheads)
- General strategy
  - When  $\text{size} < \text{threshold}$ , use a sort more efficient for small arrays (e.g., InsertionSort)
  - Good thresholds range from 5 to 20
  - Has been shown to reduce running time by 15%



# Quicksort: implementation

---

```
1  /**
2   * Quicksort algorithm (driver).
3   */
4  template <typename Comparable>
5  void quicksort( vector<Comparable> & a )
6  {
7      quicksort( a, 0, a.size( ) - 1 );
8  }
```

Left index

Right index

**Figure 7.15** Driver for quicksort

Sorting

Q1

```

1  /**
2   * Return median of left, center, and right.
3   * Order these and hide the pivot.
4   */
5  template <typename Comparable>
6  const Comparable & median3( vector<Comparable> & a, int left, int right )
7  {
8      int center = ( left + right ) / 2;
9
10     if( a[ center ] < a[ left ] )
11         std::swap( a[ left ], a[ center ] );
12     if( a[ right ] < a[ left ] )
13         std::swap( a[ left ], a[ right ] );
14     if( a[ right ] < a[ center ] )
15         std::swap( a[ center ], a[ right ] );
16
17     // Place pivot at position right - 1
18     std::swap( a[ center ], a[ right - 1 ] );
19     return a[ right - 1 ];
20 }

```



8	1	4	9	6	3	5	2	7	0
L				C					R
6	1	4	9	8	3	5	2	7	0
L				C					R
0	1	4	9	8	3	5	2	7	6
L				C					R
0	1	4	9	6	3	5	2	7	8
L				C					R
0	1	4	9	7	3	5	2	6	8
L				C				P	R

**Figure 7.16** Code to perform median-of-three partitioning

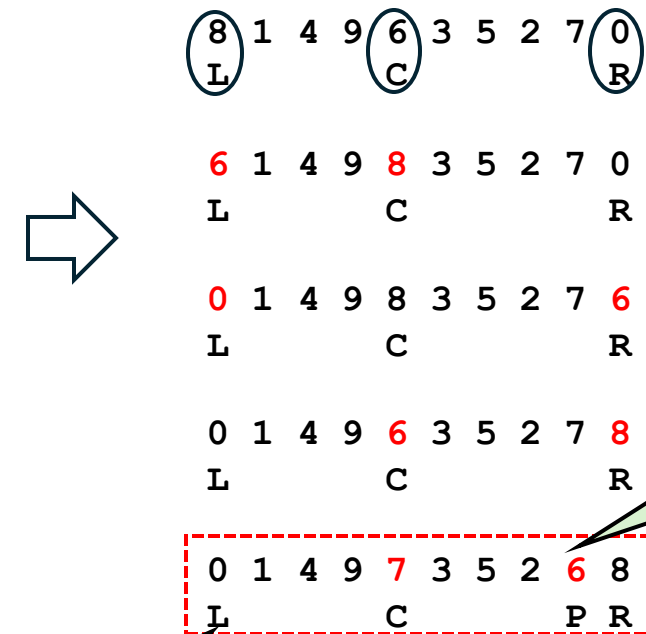
Sorting

Q1

```

1  /**
2   * Return median of left, center, and right.
3   * Order these and hide the pivot.
4   */
5  template <typename Comparable>
6  const Comparable & median3( vector<Comparable> & a, int left, int right )
7  {
8      int center = ( left + right ) / 2;
9
10     if( a[ center ] < a[ left ] )
11         std::swap( a[ left ], a[ center ] );
12     if( a[ right ] < a[ left ] )
13         std::swap( a[ left ], a[ right ] );
14     if( a[ right ] < a[ center ] )
15         std::swap( a[ center ], a[ right ] );
16
17     // Place pivot at position right - 1
18     std::swap( a[ center ], a[ right - 1 ] );
19     return a[ right - 1 ];
20 }

```

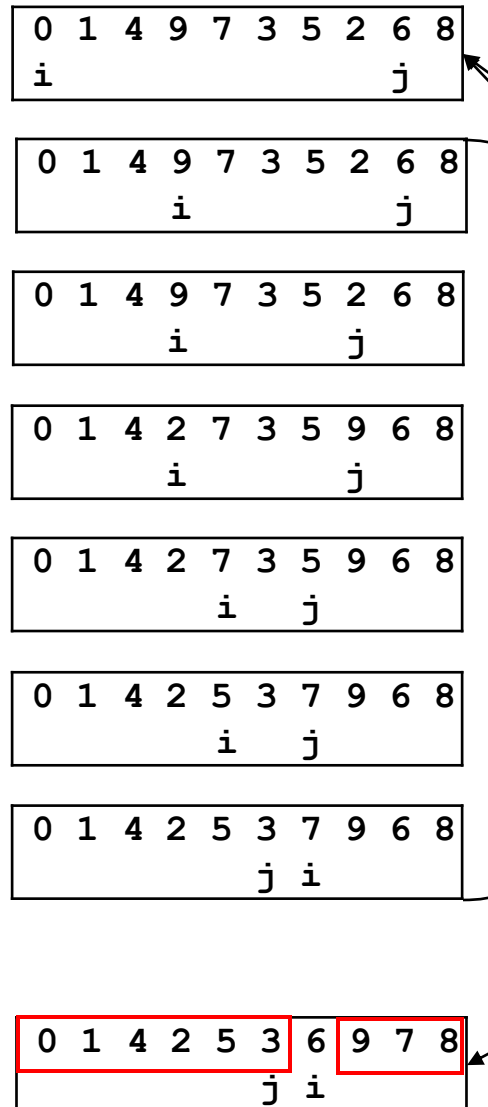


Pivot:  
median of 3

Smallest:  
at left

largest:  
at left

**Figure 7.16** Code to perform median-of-three partitioning



```

1  /**
2   * Internal quicksort method that makes recursive calls.
3   * Uses median-of-three partitioning and a cutoff of 10.
4   * a is an array of Comparable items.
5   * left is the left-most index of the subarray.
6   * right is the right-most index of the subarray.
7   */
8  template <typename Comparable>
9  void quicksort( vector<Comparable> & a, int left, int right )
10 {
11     if( left + 10 <= right )
12     {
13         const Comparable & pivot = median3( a, left, right );
14
15         // Begin partitioning
16         int i = left, j = right - 1;
17         for( ; ; )
18         {
19             while( a[ ++i ] < pivot ) { }
20             while( pivot < a[ --j ] ) { }
21             if( i < j )
22                 std::swap( a[ i ], a[ j ] );
23             else
24                 break;
25         }
26
27         std::swap( a[ i ], a[ right - 1 ] ); // Restore pivot
28
29         quicksort( a, left, i - 1 ); // Sort small elements
30         quicksort( a, i + 1, right ); // Sort large elements
31     }
32     else // Do an insertion sort on the subarray
33         insertionSort( a, left, right );
34 }

```

Assign pivot as  
median of 3

Partition based  
on pivot

Recursively  
sort partitions

**Figure 7.17** Main quicksort routine

# Quicksort: analysis

- Let  $T(N)$  = time to quicksort  $N$  elements
- Let  $L$  = #elements in left partition
- $\Rightarrow$  #elements in right partition =  $N-L-1$
- Base:  $T(0) = T(1) = O(1)$
- $T(N) = T(L) + T(N - L - 1) + O(N)$

Time to sort  
left partition

Time to sort  
right partition

Time to partition at  
current recursive step

# Quicksort: analysis

---

- **Worst-case** analysis: pivot is the smallest element ( $L=0$ )
- $T(N) = T(0) + T(N - 1) + O(N)$   
 $= O(1) + T(N - 1) + O(N)$   
 $= T(N - 2) + O(N - 1) + O(N)$   
 $= \sum_{i=1}^N O(i)$   
 $= O(N^2)$

# Quicksort: analysis

- **Best-case analysis:** pivot is the median ( $L=N/2$ )

- $T(N) = T(N/2) + T(N/2) + O(N)$   
 $= 2T(N/2) + O(N)$   
 $= O(N \log(N))$

- **Average-case analysis:** assuming each partition **equally likely**

- $T(N) = \frac{1}{N} \sum_{j=0}^{N-1} (T(j) + T(N-j-1) + O(N))$   
 $= \frac{2}{N} \sum_{j=0}^{N-1} T(j) + \frac{1}{N} \sum_{j=0}^{N-1} O(N)$   
 $= O(N \log N)$

So  $L$  can be  $0, 1, \dots, N-1$  with the same probability (indexed by  $j$ )

All proof in  
Chapter 7.7.5

## Sorting

# Comparison sorting algorithms

---

Sorting algorithm	Worst-case	Average-case	Best-case	comments
InsertionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	Fast for small N
MergeSort	$\Theta(N \log(N))$	$\Theta(N \log(N))$	$\Theta(N \log(N))$	Requires memory
HeapSort	$\Theta(N \log(N))$	$\Theta(N \log(N))$	$\Theta(N \log(N))$	Large constants in complexity
QuickSort	$\Theta(N^2)$	$\Theta(N \log(N))$	$\Theta(N \log(N))$	Small constants in complexity



# Comparison sorting algorithms

$N$	Insertion Sort $O(N^2)$	Shellsort $O(N^{7/6})(?)$	Heapsort $O(N \log N)$	Quicksort $O(N \log N)$	Quicksort (opt.) $O(N \log N)$
10	0.000001	0.000002	0.000003	0.000002	0.000002
100	0.000106	0.000039	0.000052	0.000025	0.000023
1000	0.011240	0.000678	0.000750	0.000365	0.000316
10000	1.047	0.009782	0.010215	0.004612	0.004129
100000	110.492	0.13438	0.139542	0.058481	0.052790
1000000	NA	1.6777	1.7967	0.6842	0.6154

# Lower bound on sorting

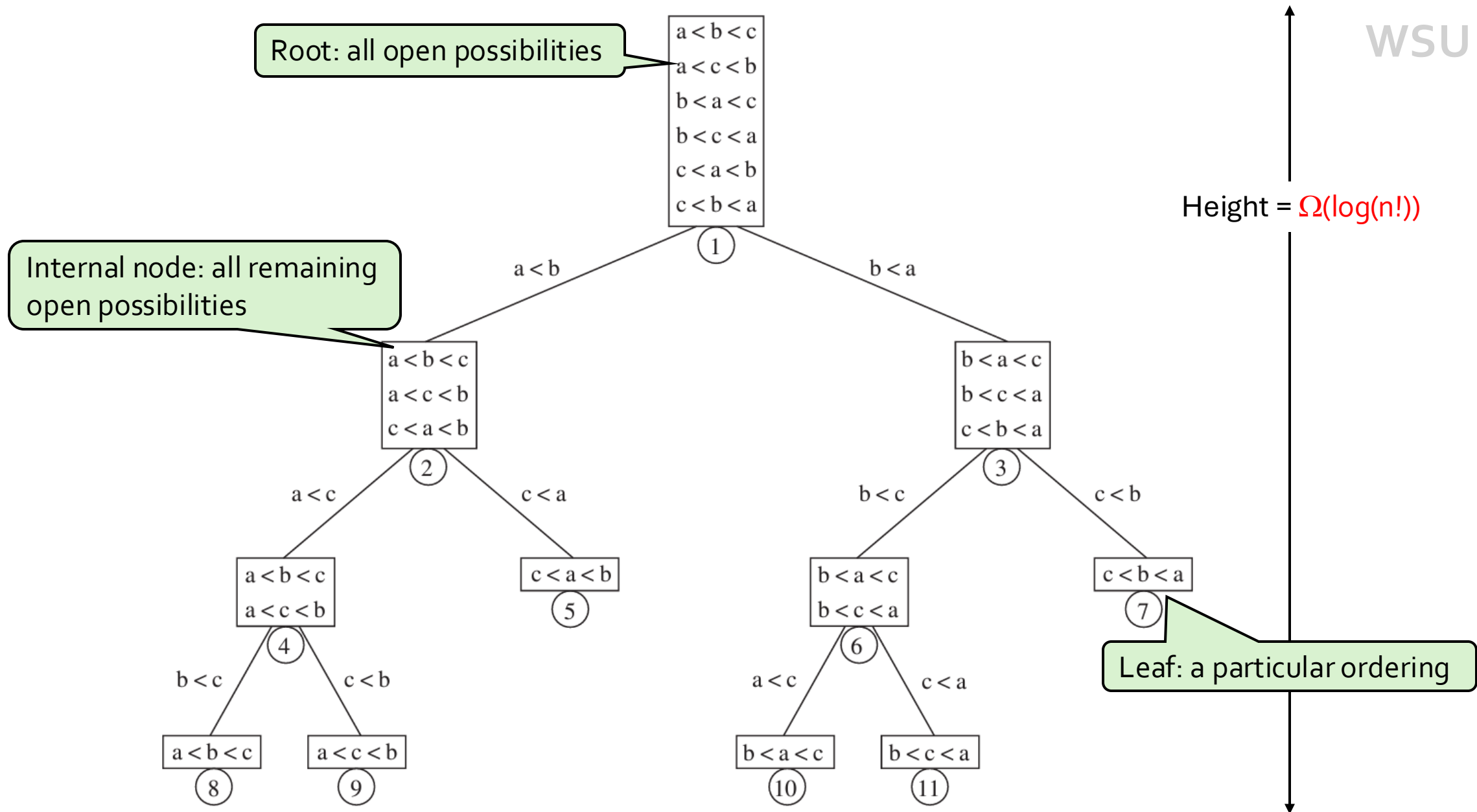
---

- What is the best we can do on comparison-based sorting?
- Best worst-case sorting algorithm (so far) is  $O(N \log(N))$ 
  - Can we do better?
- Can we prove a lower bound on the sorting problem, independent of the algorithm?
  - For comparison sorting: we cannot do better than  $O(N \log(N))$
  - Can show lower bound of  $\Omega(N \log N)$

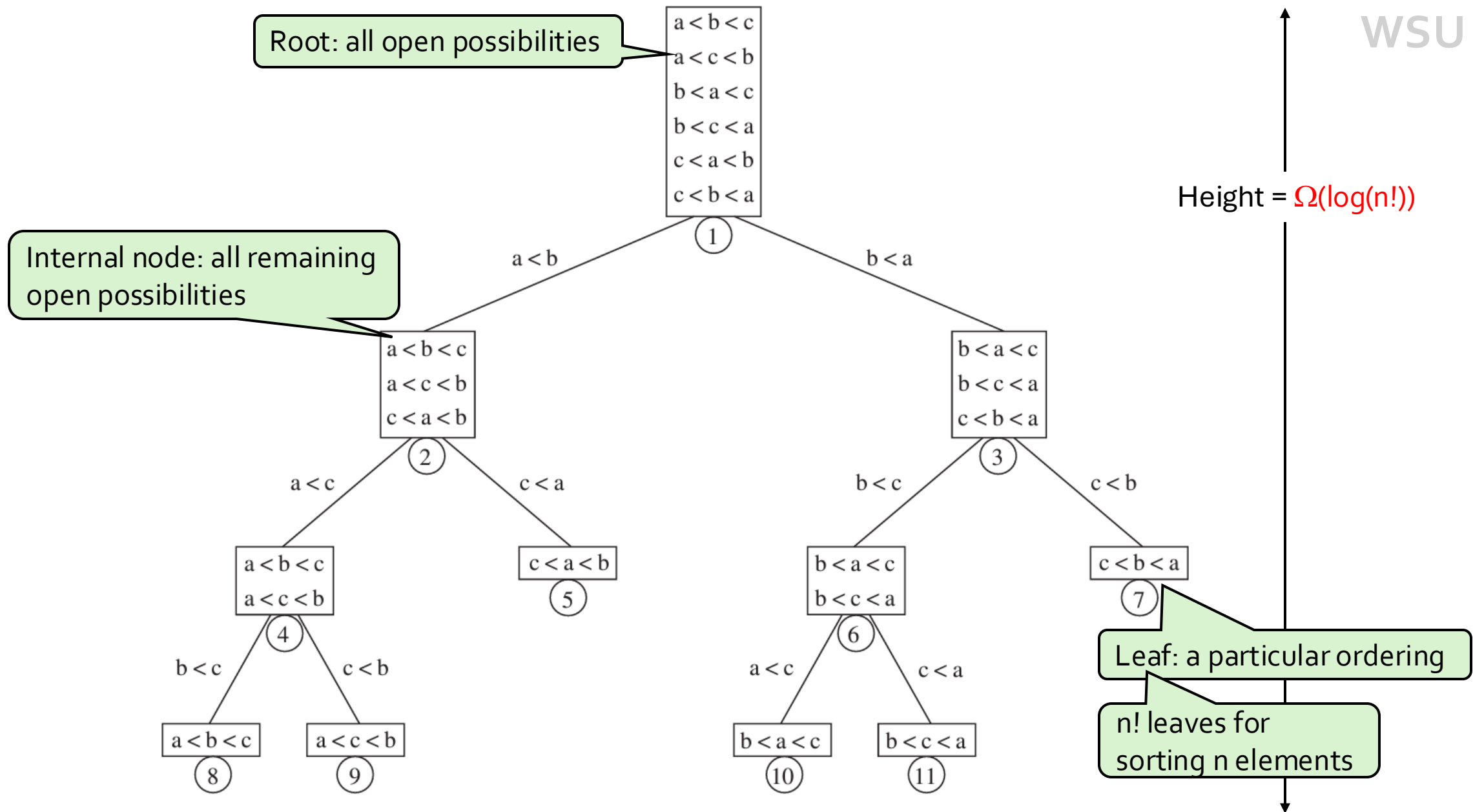
# Lower bound on sorting

---

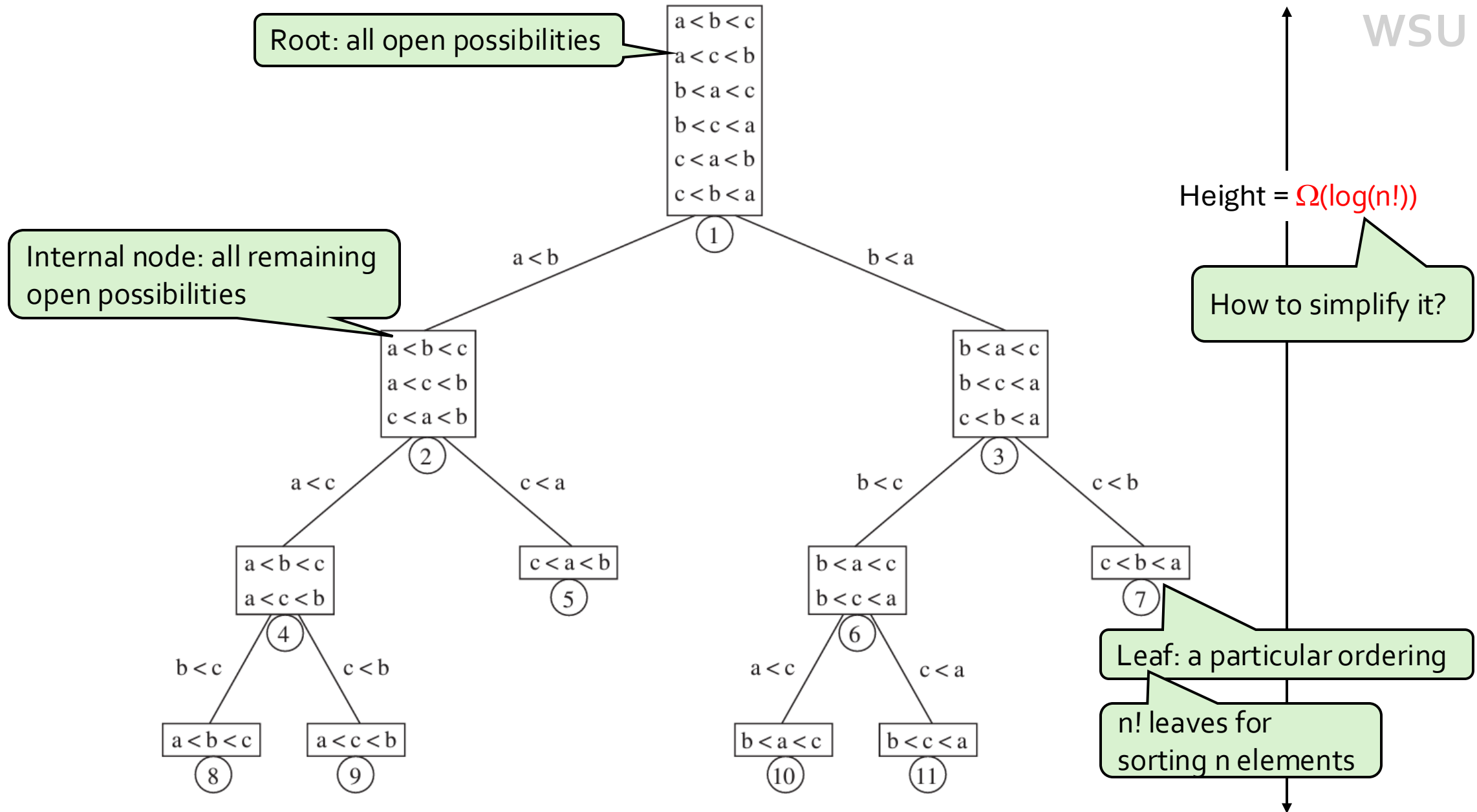
- A **decision tree** is a binary tree where:
  - Each node
    - lists all left-out **open possibilities** (for deciding)
  - Path of each node
    - represents a **decided sorted prefix** of elements
  - Each branch
    - represents **an outcome of a particular comparison**
  - Each leaf
    - Represents a **particular ordering** of the original array elements



**Figure 7.20** A decision tree for three-element sort



**Figure 7.20** A decision tree for three-element sort



**Figure 7.20** A decision tree for three-element sort

# Decision tree for sorting

---

- The logic of any sorting algorithm that uses comparisons can be **represented by a decision tree**
- In the worst case, the **number of comparisons** used by the algorithm equals the **HEIGHT OF THE DECISION TREE**
- In the average case, the number of comparisons is the **average of the depths of all leaves**
- There are  **$n!$  different orderings of  $n$  elements**

# Lower bound on sorting

---

- Lemma: A binary tree with  $L$  leaves must have depth at least:
  - $\lceil \log(L) \rceil$
- Sorting's decision tree has  $N!$  leaves
- Theorem: Any comparison sort may
  - require at least  $\lceil \log(n!) \rceil$  comparisons in the worst case



# Lower bound on sorting

---

- Theorem: Any comparison sort requires  $\Omega(N \log(N))$  comparisons
- Proof sketch (uses Stirling's approximation)

$$N! \approx 2 N (N / e)^N (1 + \Theta(1 / N))$$

Full proof in Chapter 7.8

$$N! > (N / e)^N$$

$$\log(N!) > N \log N - N \log e = \Theta(N \log N) \quad \log(N!) > \Theta(N \log N)$$

$$\therefore \log(N!) = \Omega(N \log N)$$

# Implications of the lower bound

---

- **Comparison-based sorting** cannot be achieved in less than  $O(n \log(n))$  steps in the worst case
- **Mergesort, Heapsort** are optimal
- **Quicksort** is not optimal but pretty good as optimal in practice
- Insertion sort, Bubble sort are clearly sub-optimal, even in practice

# Non-comparison-based sorting

---

- Integer sorting:
  - Counting sort
- Bucket sort
- Radix sort

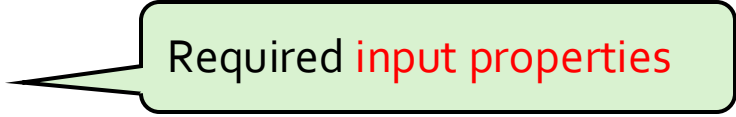
Some **input properties** allow to eliminate the need for comparison

E.g., sorting an employee database by **age (integers in some range)** of employees

## Sorting

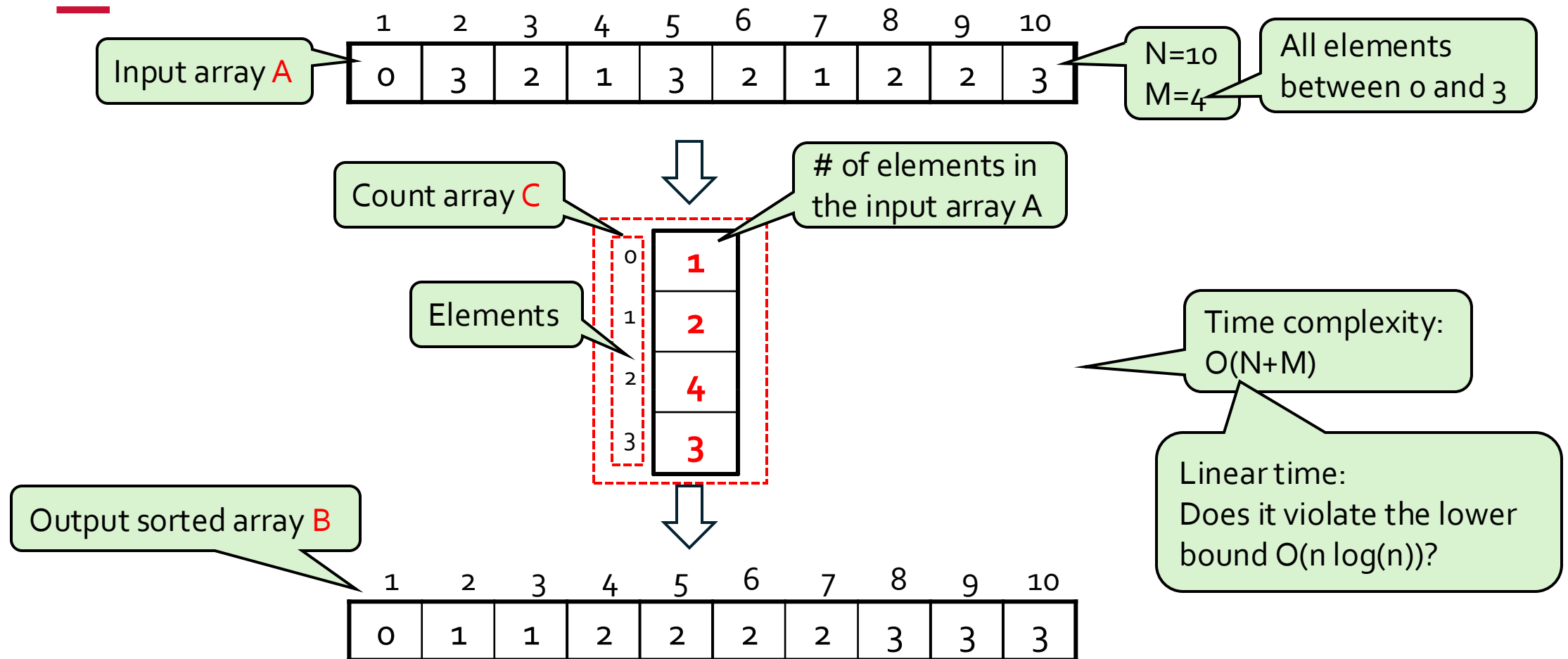
# Counting sort

---

- Given array  $A[1..N]$ , where  $1 \leq A[i] \leq M$   Required input properties
- Create array  $C$  of size  $M$ , where  $C[j]$  is the number of  $j$ 's in  $A$
- Use  $C$  to place elements into new sorted array  $B$
- Running time  $\Theta(N+M) = \Theta(N)$  if  $M = \Theta(N)$

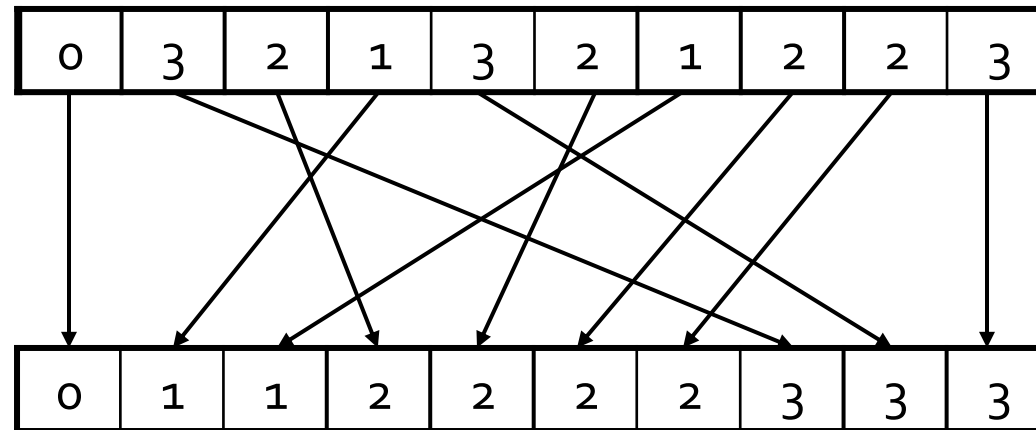
Sorting

# Counting sort: example



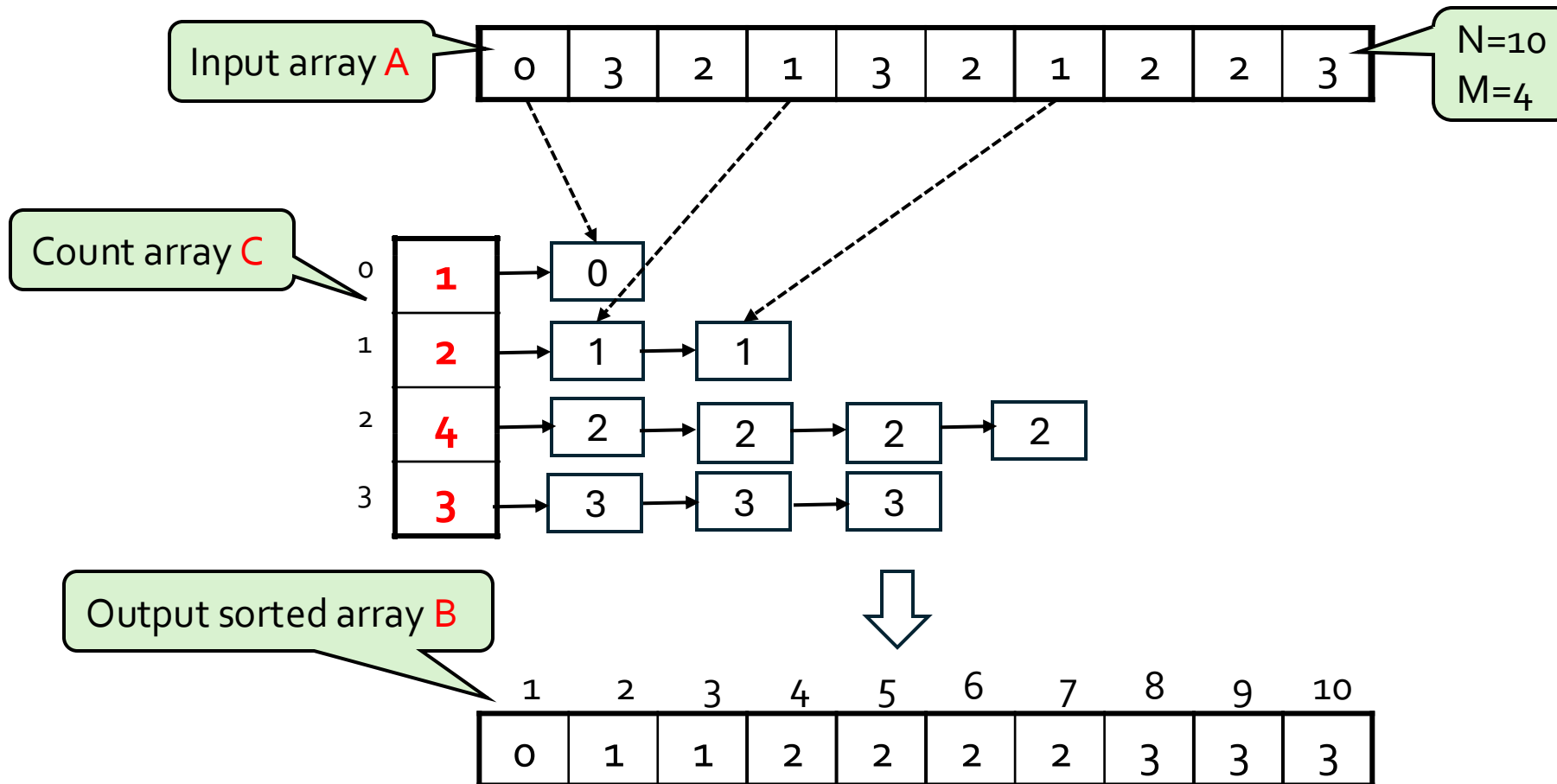
# Stable and nonstable sorting

- A “stable” sorting method:
  - preserves the original input order among duplicates in the output



Sorting

# Make counting sort stable



# Bucket sort

---

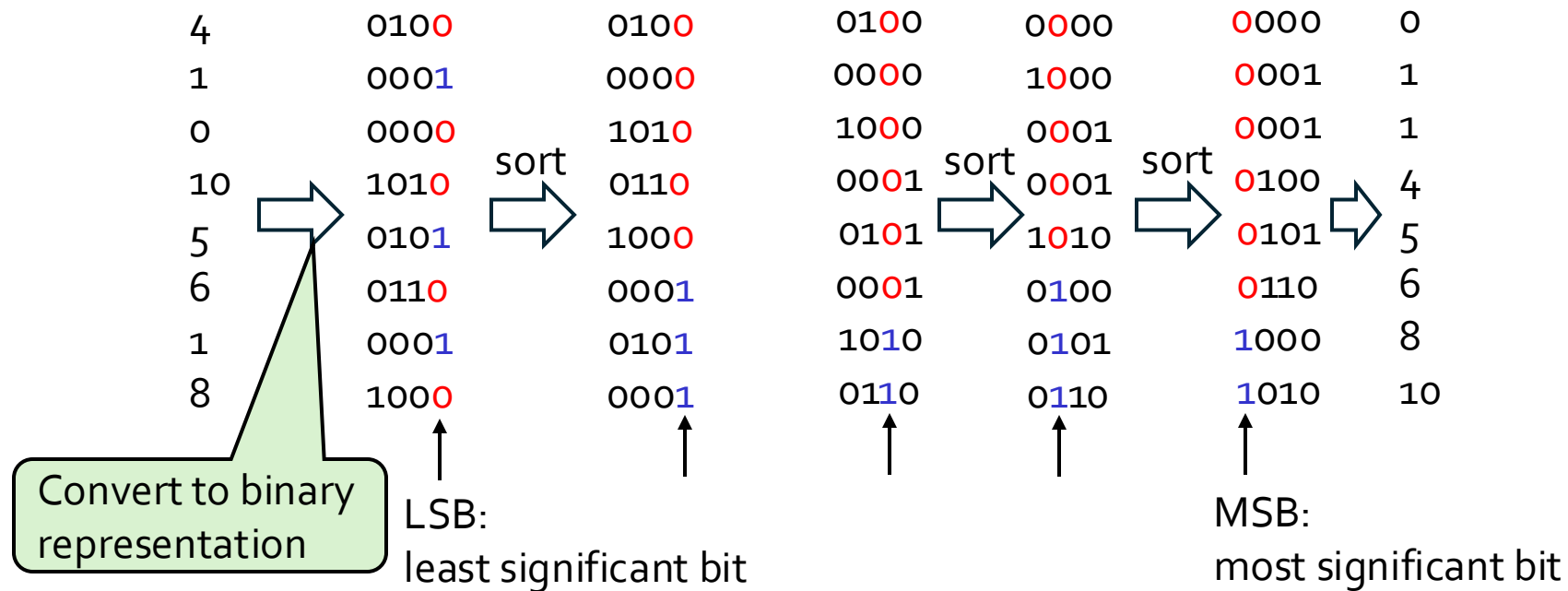
- Assume  $N$  elements of  $A$  uniformly distributed over the range  $[0,1]$
- Create  $M$  equal-sized buckets over  $[0,1]$ , s.t.,  $M \leq N$
- Add each element of  $A$  into appropriate bucket
- Sort each bucket internally
  - Can use recursion, or
  - Can use something like InsertionSort
- Return concatenation of buckets
- Average case running time  $\Theta(N)$ 
  - assuming each bucket will contain  $\Theta(1)$  elements



Sorting

# Radix sort

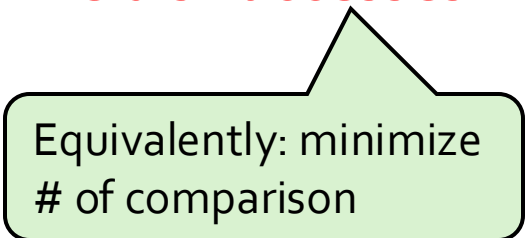
- Sort  $N$  numbers, each with  $k$  bits
- E.g, input {4, 1, 0, 10, 5, 6, 1, 8}



# External sorting

---

- What if the number of elements  $N$  **do not fit in memory**?
- Obviously, our existing sort algorithms are inefficient
- **Each comparison** potentially requires a **disk access**
- Once again, we want to **minimize disk accesses**

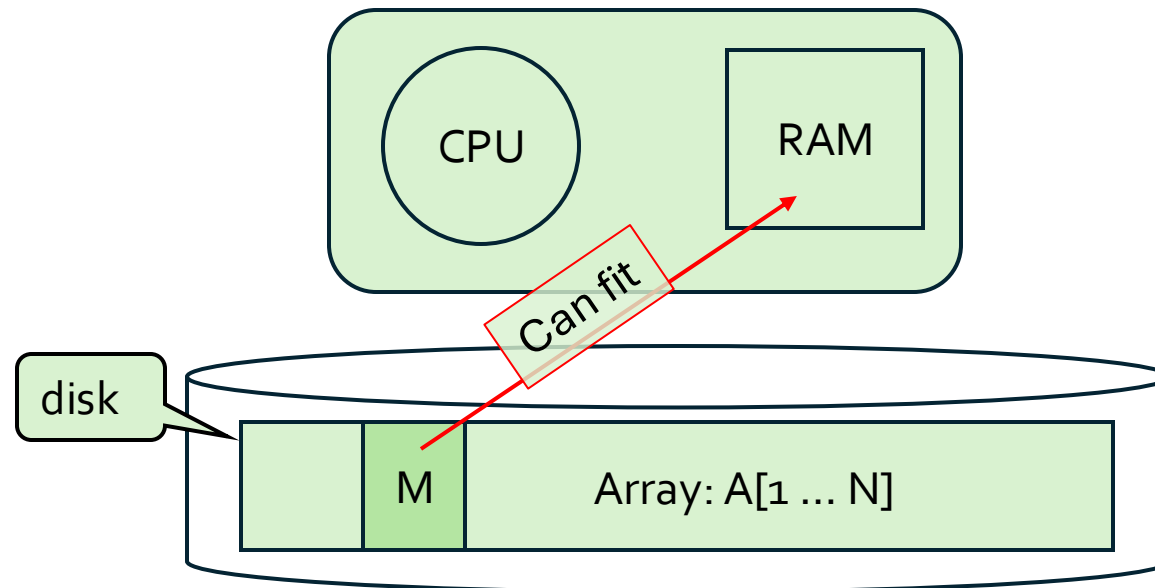


Equivalently: minimize  
# of comparison

Sorting

# External MergeSort

- $N$  = number of elements in array  $A[1..N]$  to be sorted
- $M$  = number of elements that **can fit in memory** at any given time
- $K = \lceil N/M \rceil$



# External MergeSort

- 
1. Read in **M amount of A**, sort it using local sorting (e.g., quicksort), and write it back to disk.   
Callout: Step 1:  $O(M \log(M))$   
Callout:  $K = \lceil N/M \rceil$
2. Repeat above **K** times until all of A processed.   
Callout: Step 2 goes over all elements in the array:  $O(K M \log(M))$   
Callout: K chunks: each sorted internally
3. Create **K input buffers** and **1 output buffer**, each of size  $r = M/(K+1)$
4. Perform a **K-way merge**:  
• Update **input buffers** one disk-page at a time  
• Write **output buffer** one disk-page at a time

# External MergeSort

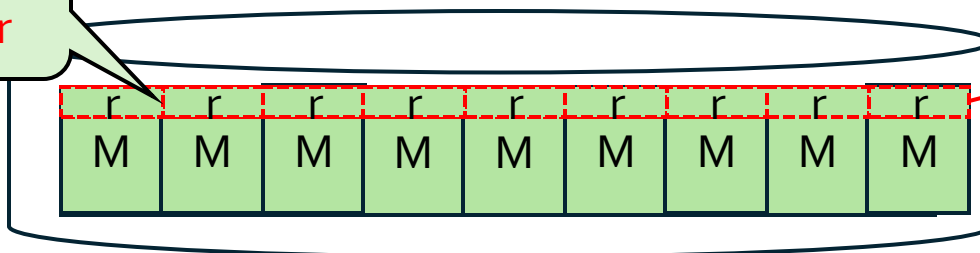
$$O(K M \log(M)) = O(N \log(M))$$

1. Read in **M amount of A**, sort it using local sorting (e.g., quicksort), and write it back to disk
2. Repeat above **K** times until all of A processed
3. Create **K input buffers** and **1 output buffer**, each of size  $r = M/(K+1)$
4. Perform a **K-way merge**:
  - Update **input buffers** one disk-page at a time
  - Write **output buffer** one disk-page at a time

# External MergeSort

1. Read in **M amount of A**, sort it using local sorting (e.g., quicksort), and write it back to disk
2. Repeat above **K** times until all of A processed
3. Create **K input buffers** and **1 output buffer**, each of size  **$r = M/(K+1)$**
4. Perform a **K-way merge**:
  - Update **input buffers** one disk-page at a time
  - Write **output buffer** one disk-page at a time

Input buffers:  
Total size =  $M-r$



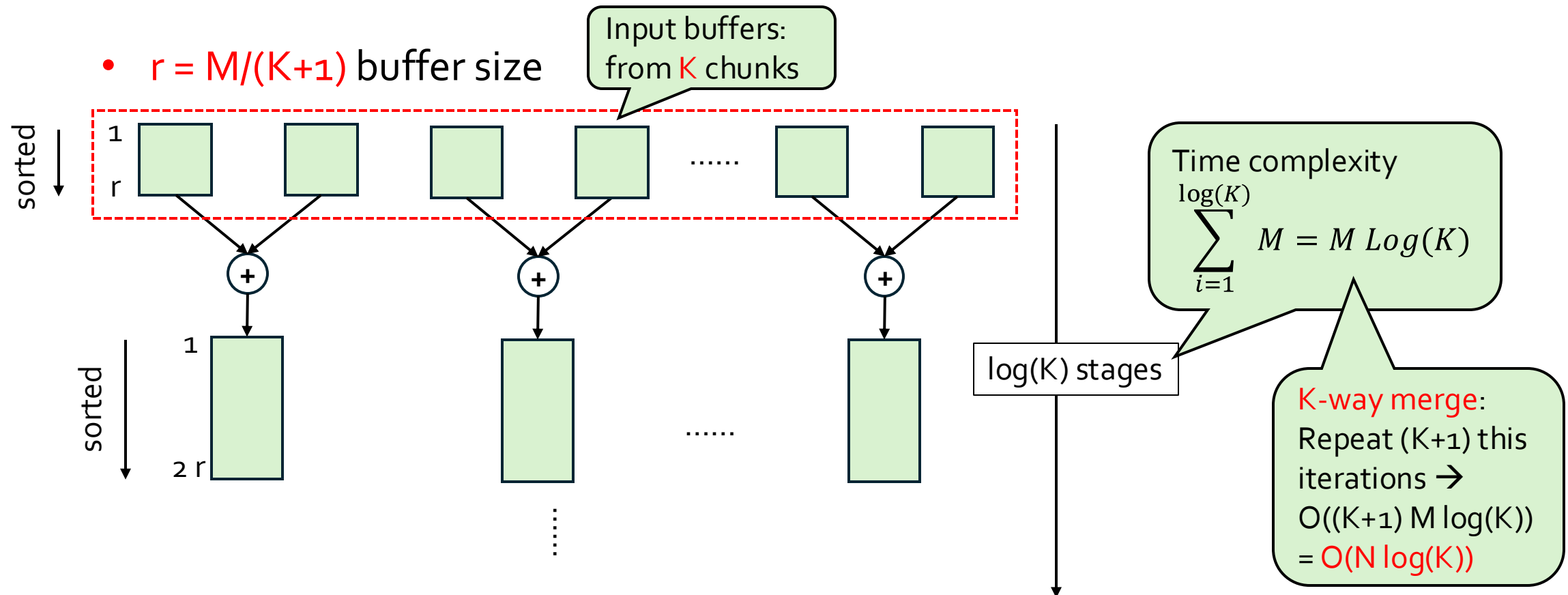
So that we can fit  $(K+1)$  buffers in the memory at the same time

K-way merge: Iteratively read **r size** of data from each disk block to each input buffer

Sorting

# K-way merge

- $r = M/(K+1)$  buffer size



# External MergeSort: analysis

---

- Computational time
- $T(N, M)$ 
  - $= O(K * M \log(M)) + O((K+1) * M \log(K))$
  - $= O(\textcolor{red}{N} \log(M) + \textcolor{red}{N} \log(K))$
  - $= O(N \log(M))$
- Disk accesses (all sequential)
  - $P$  = page size
  - # of accesses =  $O(N/P)$



# Non-comparison-based sorting

---

- Need for sorting is ubiquitous in software
- Optimizing the sort algorithm to the domain is essential
- Good general-purpose algorithms available
  - QuickSort
- Optimizations continue...
  - Sort benchmarks: <http://sortbenchmark.org>