Session 04 - Sorting Algorithms

Instructor:
Dr. LE Thanh Tung

Content

1. Merge Sort
2. Quick Sort
3. Radix Sort
4. Counting Sort

Review

	Best	Average	Worst
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$

- What’s the worst-case runtime complexity of heapsort

Sorting

- How can we design better, more efficient sorting algorithms?
- So far, we've seen $O(N^2)$ sorting algorithms. How can we start to do better?
  - Divide-and-Conquer

- Assume that it takes t seconds to run insertion sort on the following array:

14	6	3	9	16	7	2	15	10	8

- Approximately how many seconds will it take to run insertion sort on each of the following arrays?

14	6	3	9	16

7	2	15	10	8

- Assume that it takes **t** seconds to run insertion sort on the following array:

14	6	3	9	16	7	2	15	10	8

- Approximately how many seconds will it take to run insertion sort on each of the following arrays?

14	6	3	9	16

7	2	15	10	8

- Each array should only take about **t/4** seconds to sort

- Motivating Divide-and-Conquer
  - Sorting **N** elements directly takes total time t
  - Sorting two sets of **N/2** elements (total of **N** elements) takes total time **t/2**
  - We got a speedup just by sorting smaller sets of elements at a time!

- Our general approach when designing a divide-and-conquer algorithm is to decide how to make the problem smaller and how to unify the results of these solved, smaller problems
- Both sorting algorithms we explore today will have both of these components
  - **Divide** Step: Make the problem smaller by splitting up the input list
  - **Join** Step: Unify the newly sorted sublists to build up the overall sorted result

Merge Sort

This technique can be divided into the following three parts:

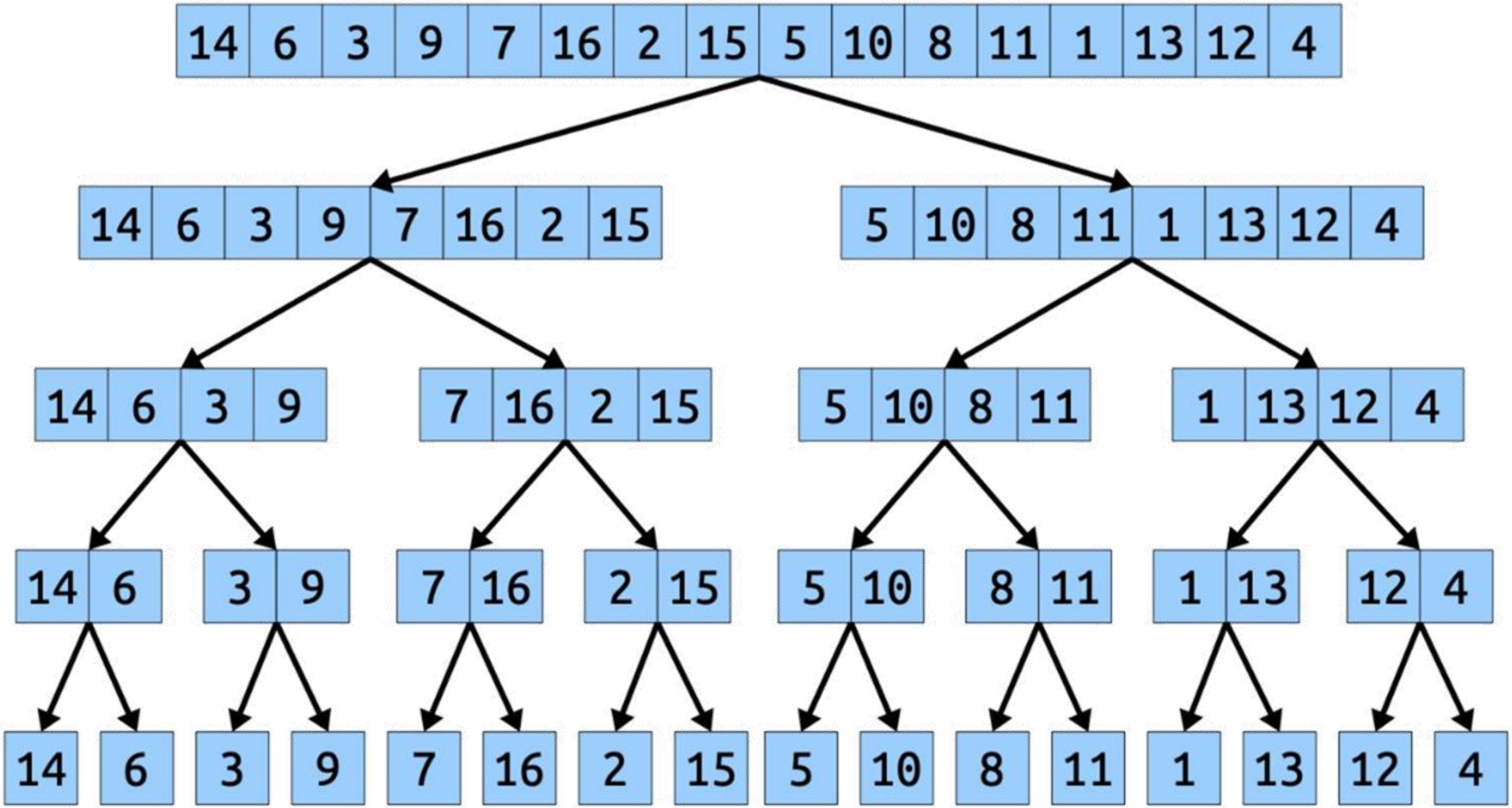
- **Divide**: This involves dividing the problem into smaller sub-problems.
- **Conquer**: Solve sub-problems by calling **recursively** until solved.
- **Join**: **Combine** the sub-problems to get the final solution of the whole problem

Merge Sort

- Merge Sort algorithm is one of two important **divide-and-conquer** sorting algorithms.
- It is a recursive sorting algorithm
  - **Base case**: An empty or single-element list is already sorted
  - **Recursive step**:
    - Break the list in half and recursively sort each part
    - Use merge to combine them back into a single sorted list

- Idea: It is a recursive algorithm.
    - Divides the list into halves,
    - Sort each halve separately, and
    - Then merge the sorted halves into one sorted array.

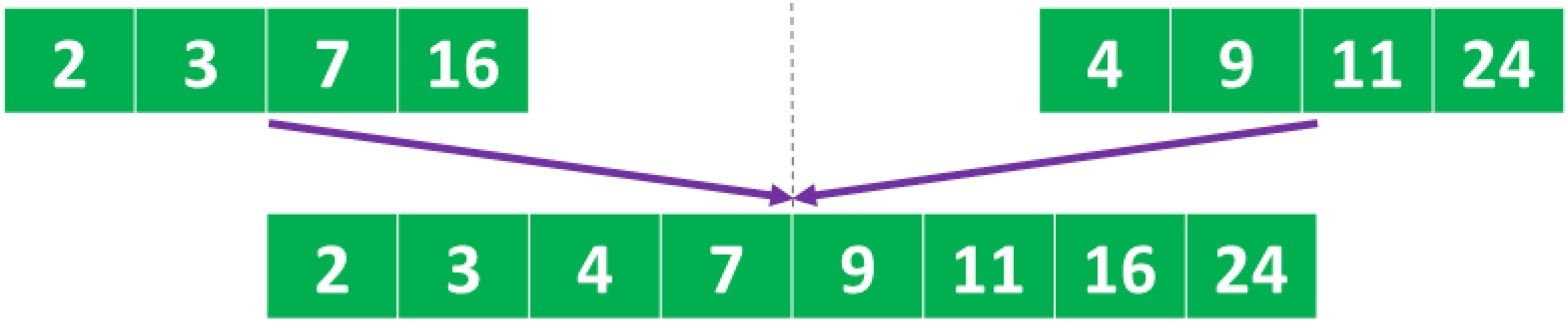
- **Note**:
    - A list with 0 or 1 element is a sorted list.


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

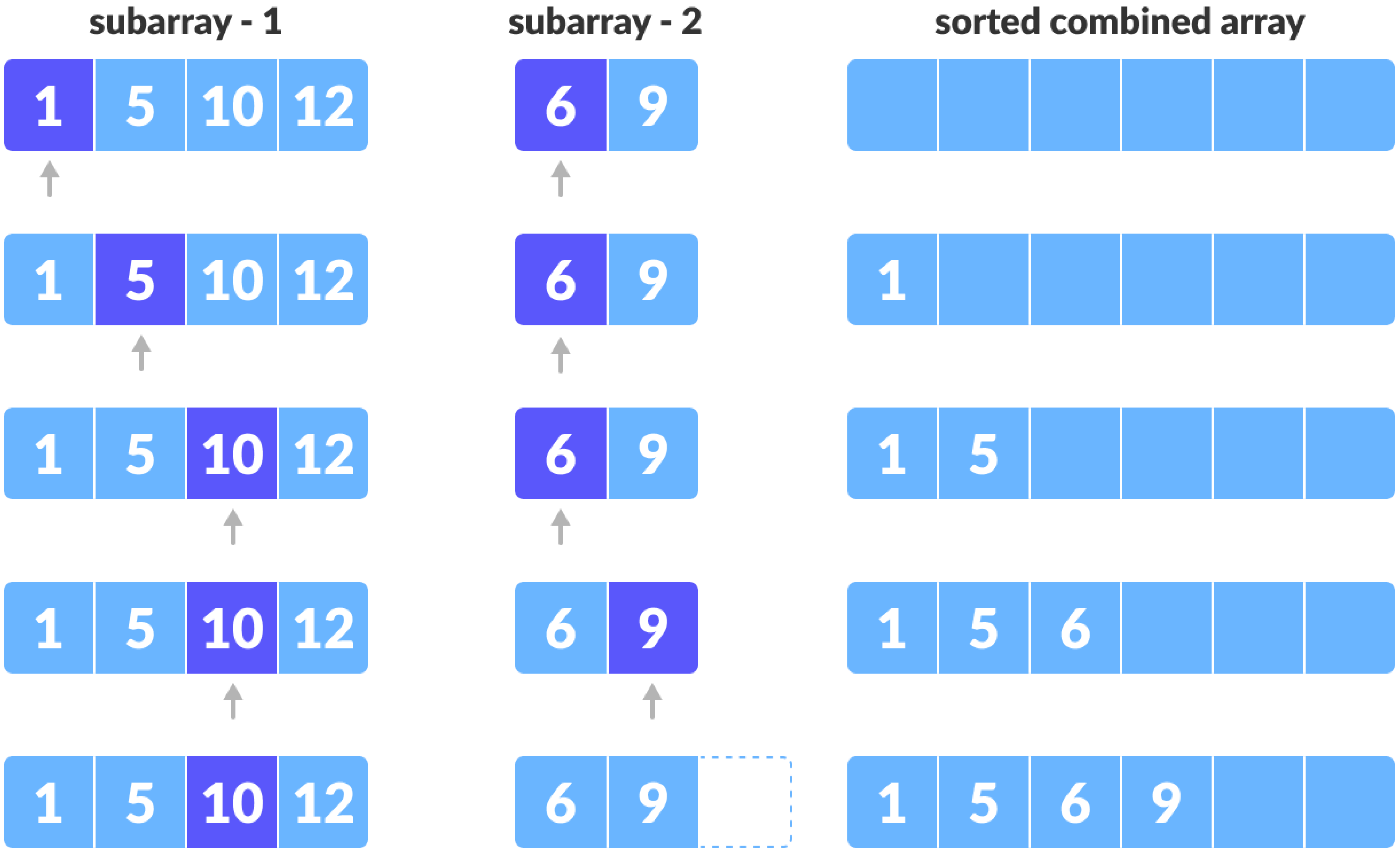
Merge Sort

- Merge procedure:
  - **Goal**: Merge two ordered lists into an order list.
  - Input: two **ordered** lists A[] (n elements), B[] (m elements)
  - Output: a new **ordered** list C[] (n + m elements) (containing all elements of A and B).

- Merge procedure:
  - **Input**:
    - A = {2, 3, 7, 16},
    - B = {4, 9, 11, 24};
  - **Output**:
    - C = {2, 3, 4, 7, 9, 11, 16, 24}



- **Propose the efficient algorithm**



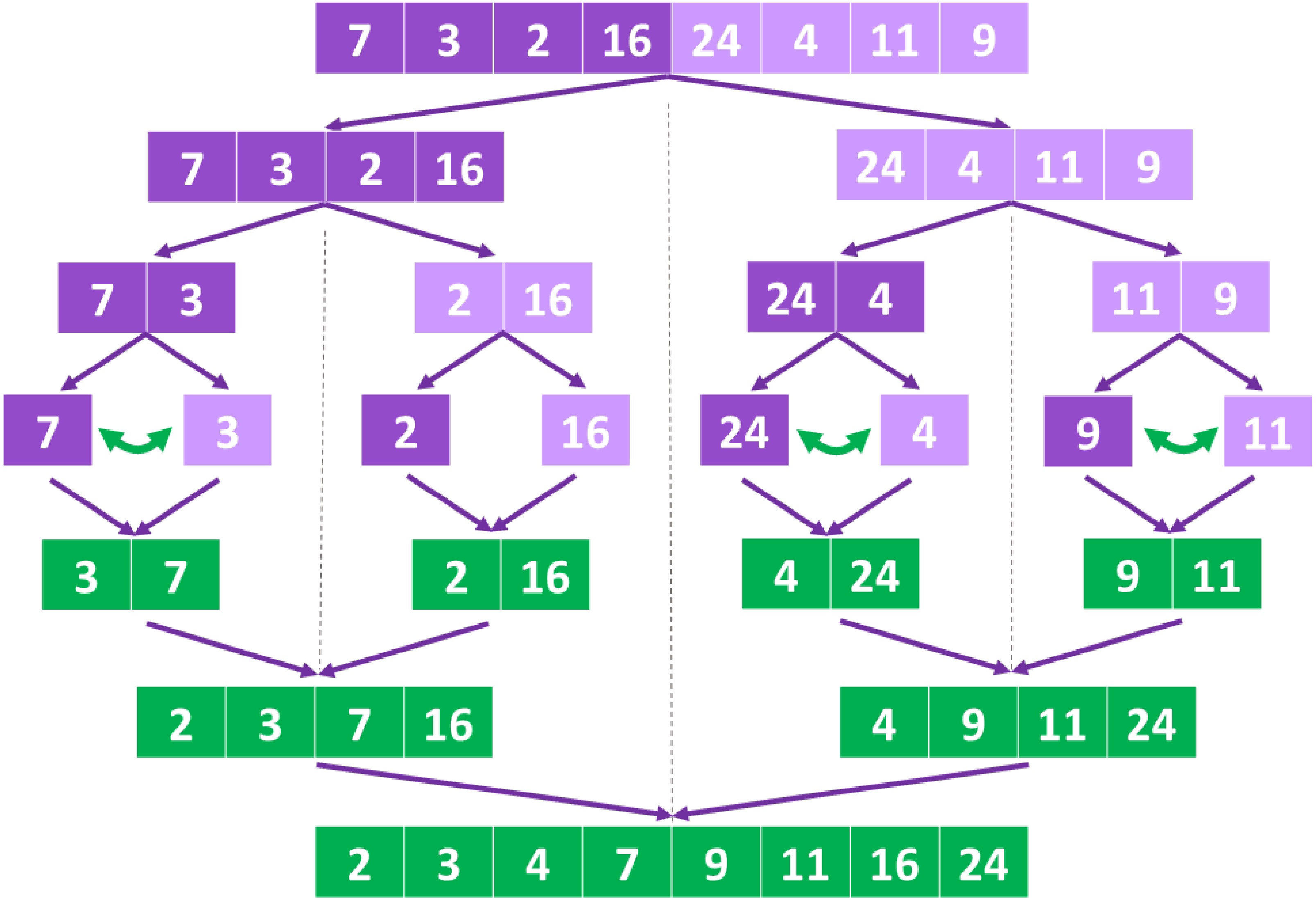
subarray - 1
subarray - 2
sorted combined array
1 5 10 12
6 9
1
1 5
1 5 6
1 5 6 9

Merge Sort

- Input: A[], left, right (list A from index left to right).
- Output: (Ordered) A[] (from left, to right)

```
MergeSort(A[], left, right)
{
   if (left < right) {
      mid = (left + right)/2;
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
   }
}
```

Merge Sort



Step 1:
Split sub-lists in two until you reach pair of values.

Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Merge Sort

- Exercise: employs the Merge Sort algorithm to effectively organize a given set of integers listed below

6 3 9 1 5 4 7 2

Merge Sort: Analysis

- Merge Sort is extremely efficient algorithm with respect to time.
  - Both worst case and average case are $O(n * \log_2 n)$

- Merge Sort requires an extra array whose size equals to the size of the original array.

- If we use a linked list, we do not need an extra array
  - But we need space for the links
  - And, it will be difficult to divide the list into half $O(n)$

Merge Sort: Analysis

- Number of division stages is $\log_2 n$ (Divide Phase)
- On each merge step, n elements are merged:
  - `Step 1: n × 1`
  - `Step 2: n/2 x 2`
  - `Step 3: n/4 x 4`
  - …
- Best case: occurs when the elements are already sorted in ascending order
- The time complexity of merge sort is **`O(nlog(n))`**

Merge Sort: Pros and Cons

- Advantages:
  - Stability
  - Guaranteed worst-case performance
  - Parallelizable: take advantage of multiple processors or threads
- Drawbacks
  - Space complexity
  - Not in-place
  - Not always optimal for small datasets

Merge Sort: Pros and Cons

- Merge sort runs in time $O(n \log n)$, which is faster than insertion sort's $O(n^2)$?

Quick Sort

- Like Merge Sort, Quick Sort is also based on the **divide-and-conquer paradigm**.

- It works as follows:
  - First, it partitions an array into two parts, *(hard divide)*
  - Then, it sorts the parts independently,
  - Finally, it combines the sorted subsequences by a simple concatenation. *(easy join)*

Quick Sort

- The algorithm consists of the following three steps:
- Divide: Partition the list.
  - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the **pivot**.
  - Then we partition the elements so that all those with values **less than** the pivot come **in one sub-list** and all those with greater values come in another.

$$\underbrace{A[0]\ldots A[p-1]}_{<A[p]} \quad \boxed{A[p]} \quad \underbrace{A[p+1]\ldots A[n-1]}_{\geq A[p]}$$

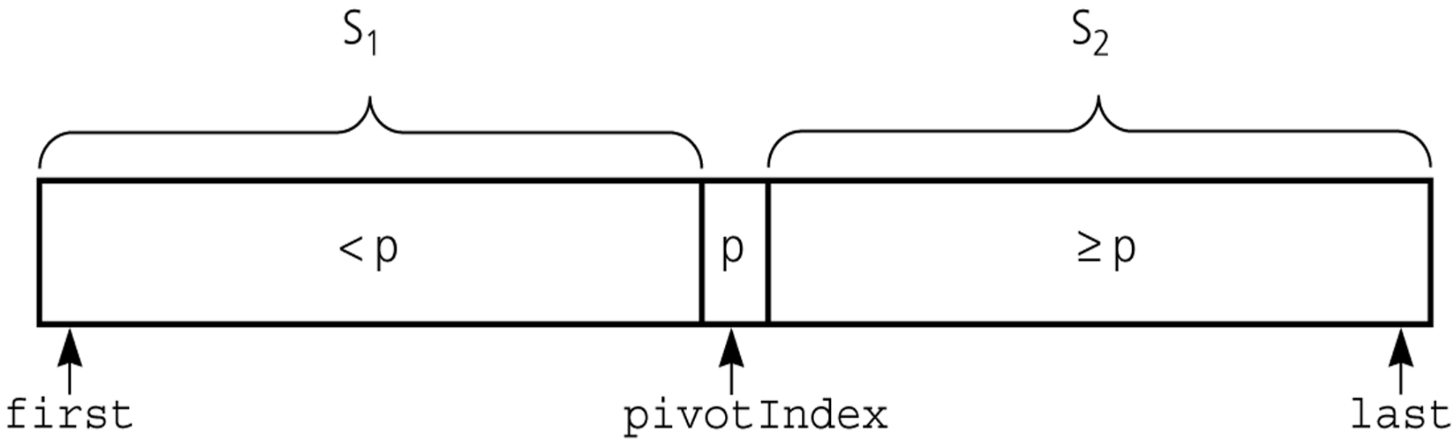
- Recursion: Recursively sort the sub-lists separately.
- Conquer: Put the sorted sub-lists together.

- Input: A[], first, last **(Sort the list A[] from index *first* to *last*)**
- Output: Ordered list A[first … last]

```
QuickSort(A[], first, last)
   if (first < last) {
      // Select a pivot p from A[].
      pivotIndex = Partition(A, first, last)
      //Partition A[] into 2 sub-lists
      // S1(first … pivotIndex-1), S2 (pivotIndex+1 … last)
      QuickSort (A, first, pivotIndex-1) //Sort S1
      QuickSort (A, pivotIndex + 1, last) //Sort S2
   }
```

Quick Sort - Partition

- Partitioning places the pivot in its correct place position within the array.

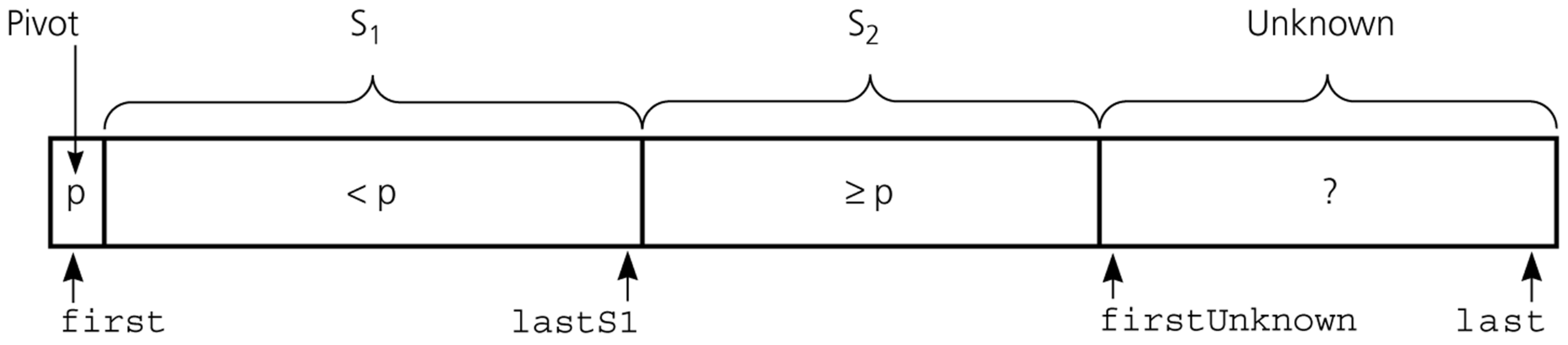


- Arranging the array elements **around the pivot p** generates two smaller sorting problems.
  - sort the **left section** of the array and sort the **right section** of the array.
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

- Selecting the pivot
  - Select a pivot element among the elements of the given array
  - We put this pivot into the first location of the array before partitioning

Quick Sort - Partition

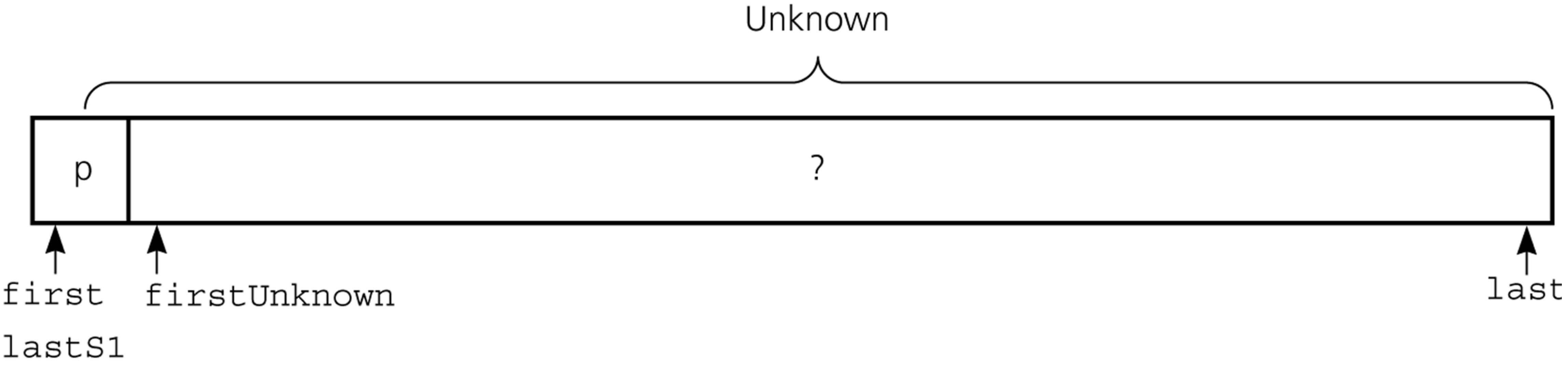
- Partitioning uses two more variables:
  - `lastS1`: the last index of S1 (the elements in A less than p).
  - `firstUnknown`: the first index of Unknown.
- Partitioning takes place when `firstUnknown` ≤ `last`.



Quick Sort - Partition

- Initialize
  - `lastS1 = first`
  - `firstUnknown = first + 1`

- Initial state

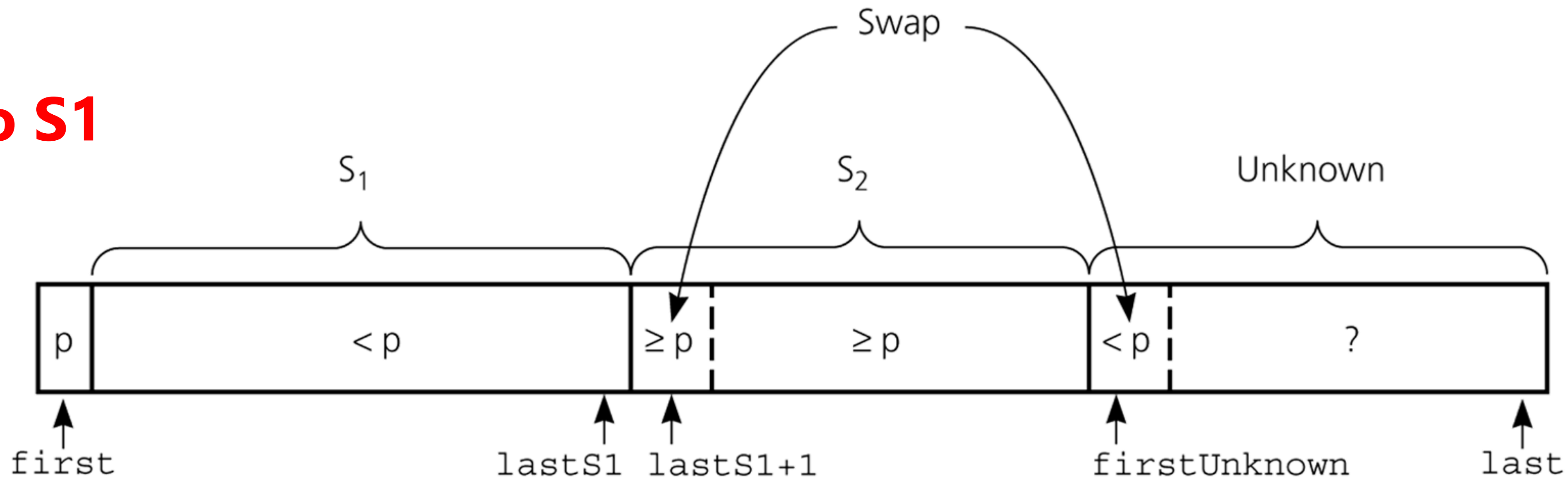


Quick Sort - Partition

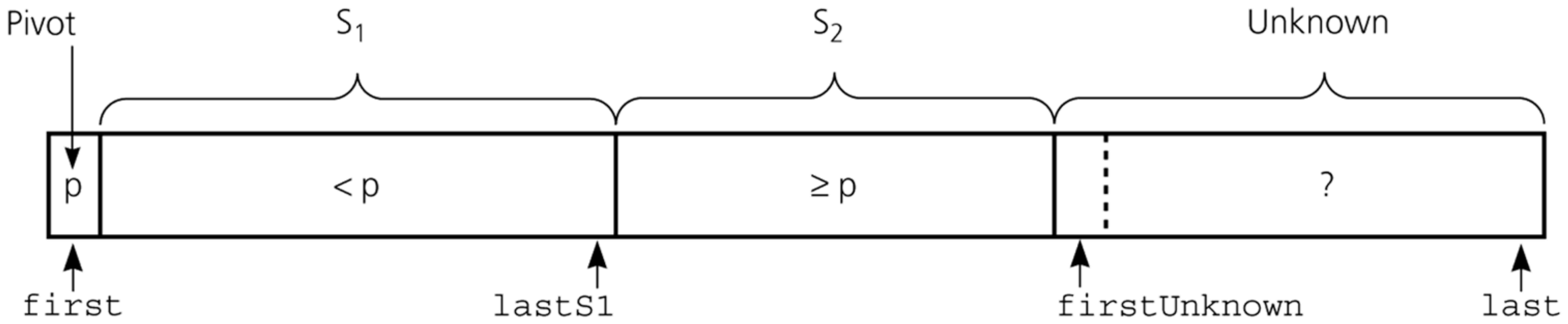
```
Partition(A[], first, last, pivot) -> pivotIndex

Step 1. while (firstUnknown <= last) //not finish
   1.1 If a[firstUnknown] < a[pivot]
        then move that element to S1
        Otherwise, move that element to S2
   1.2 firstUnknown = firstUnknown + 1 //next element
Step 2. Move pivot to the correct position
(between S1 and S2):
   Swap two elements at lastS1 and first.
Step 3. pivotIndex = lastS1
```

Move to S1



Move to S2



Quick Sort - Partition

- Partition this list: 27, 38, 12, 39, 27, 16

Pivot	Unknown				
27	38	12	39	27	16

firstUnknown = 1

firstS1 = 0

lastS1 = 0

Move to S2

Pivot	S2	Unknown			
27	38	12	39	27	16

firstUnknown = 2

firstS1 = 0

lastS1 = 0

- Partition this list: 27, 38, 12, 39, 27, 16

Pivot	S2	Unknown			
27	38	12	39	27	16

firstUnknown = 2

firstS1 = 0

lastS1 = 0

Pivot	S1	S2	Unknown		
27	12	38	39	27	16

Swap(a[lastS1 + 1], a[firstUnknown])

lastS1 = 1

firstUnknown = 3

Quick Sort - Partition

- Partition this list: 27, 38, 12, 39, 27, 16

Pivot	S2	Unknown			
27	38	12	39	27	16

firstUnknown = 2

firstS1 = 0

lastS1 = 0

Pivot	S1	S2	Unknown		
27	12	38	39	27	16

Move to S1

Swap(a[lastS1 + 1], a[firstUnknown]

lastS1 = 1

firstUnknown = 3

- Partition this list: 27, 38, 12, 39, 27, 16

Pivot	S1	S2	Unknown		
27	12	38	39	27	16

Pivot	S1	S2			U.K
27	12	38	39	27	16

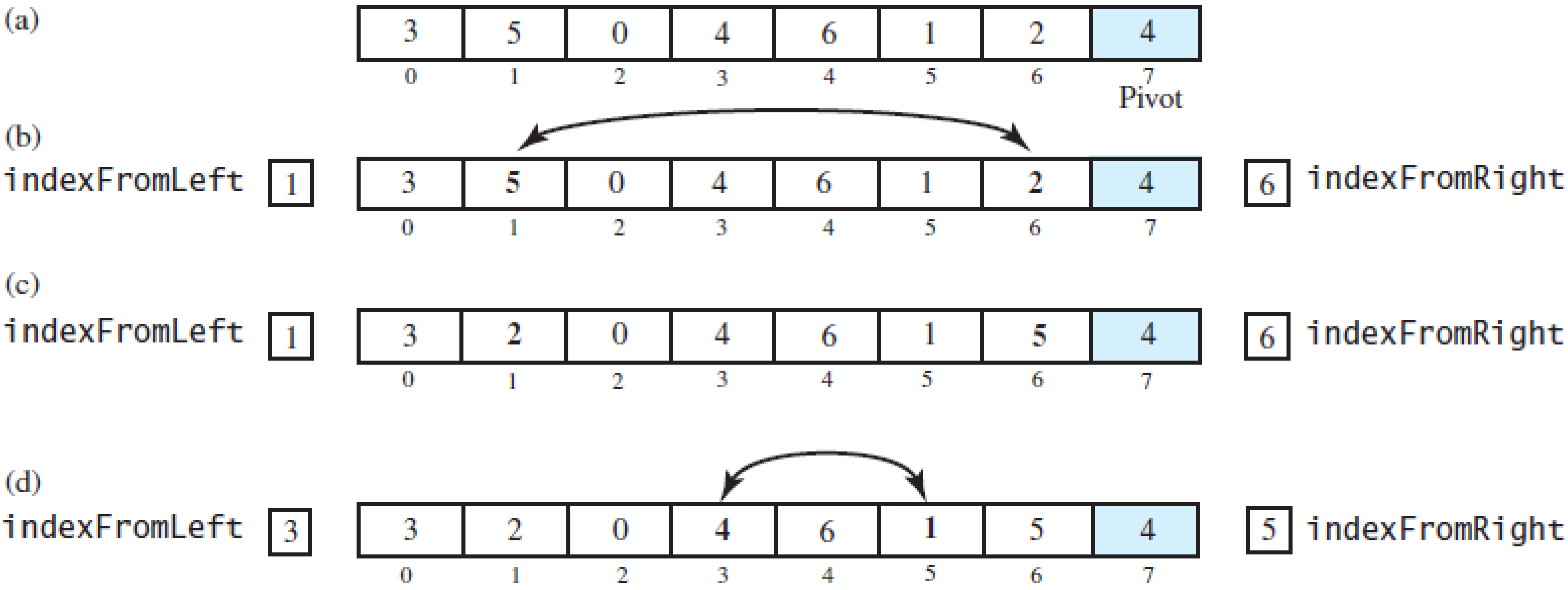
Pivot	S1		S2		
27	12	16	39	27	38

S1		Pivot	S2		
16	12	27	39	27	38

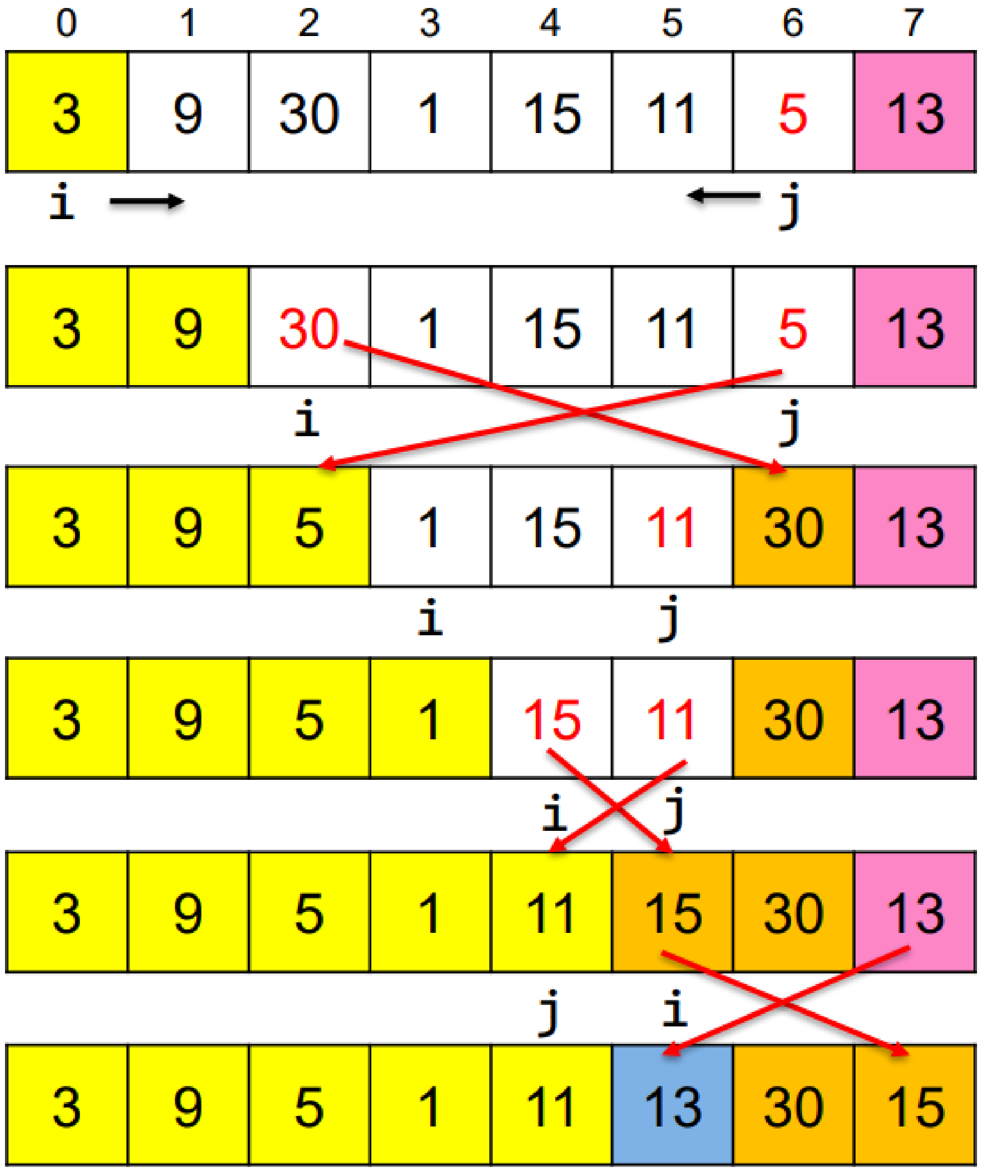
Swap a[0], a[lastS1]

Return pivotIndex

- Another technique:



- Another technique:



An inversion occurs:
- A[j] < pivot
- A[i] > pivot

→ Swap A[i], A[j]

X	A[i] <= pivot
X	A[j] > pivot
X	Pivot
X	Sorted item

Finally, swap A[i] and pivot

→ *pivot is at correct position*

- **Which array item should be selected as pivot?**
  - The first element
  - The last element
  - The middle element
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).

→ We can use **different techniques** to select the pivot

→ E.g: **Median-of-three pivot selection**

Quick Sort: Analysis

- **What is the worst/best case of Quick Sort ?**
  - Best case:
  - Worst case:

Quick Sort: Analysis

- **What is the worst/best case of Quick Sort ?**
  - Best case: all the splits happen in the middle of corresponding subarrays
    - Pivot is chosen as the median of two sub-lists
  - Worst case: 1 of the 2 subarrays is empty
    - Pivot is chosen as the largest or smallest element in sub-list

Quick Sort: Analysis

- **Time Complexity**
  - Best case/Average Case: $O(n\log_2(n))$
  - Worst case: $O(n^2)$
- **Notes:**
  - Quick Sort is slow when the array is sorted and we choose the first element as the pivot
  - Although the worst case behavior is not so good, its average case behavior is much better than its worst case
  - Quick Sort is one of best sorting algorithms using key comparisons.

Quick Sort: Pros and Cons

- **Pros:**
  - The performance in average case is far better than its worst case
  - In-place Algorithm
  - In most situation, quick sort is better than merge sort
    - This is because its innermost loop is so efficient
- **Cons:**
  - Not stable
  - In the worst case, quick sort is significant slower than merge sort.

Radix Sort

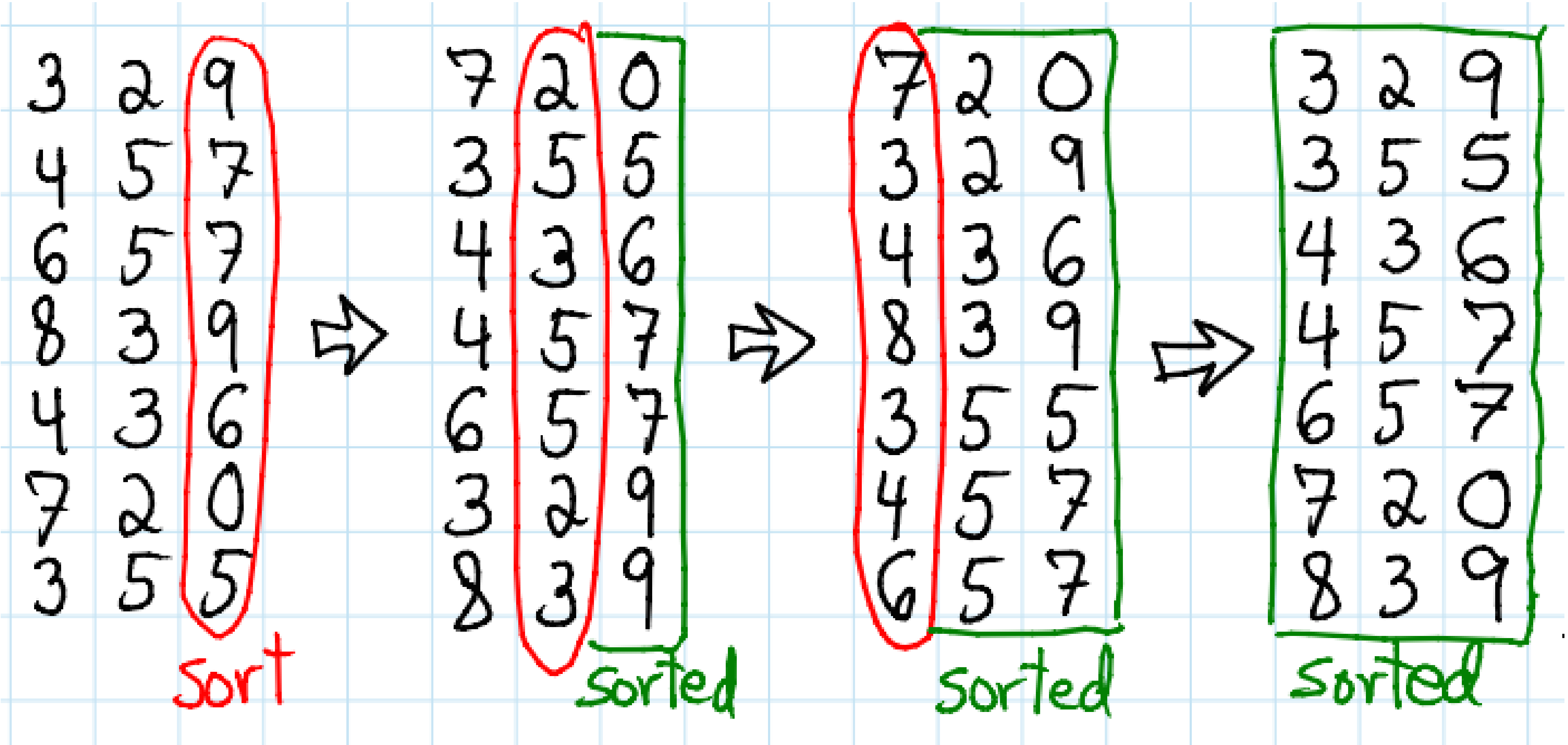
- Radix Sort algorithm different than other sorting algorithms that we talked.
- It **DOES NOT** use key **comparisons** to sort an array.

- Treats each data item as a character string.

- Repeat *(for all character positions from the rightmost to the leftmost)*
  - Groups data items according to their rightmost character
  - Put these groups into order with respect to this rightmost character.
  - Combine all the groups.
  - Move to the next left position.

- At the end, the sort operation will be completed.

```
RadixSort(A[], n, d) // sort n d-digit integers in the array A
  for (j = d down to 1) {
        Initialize 10 groups to empty
        Initialize a counter for each group to 0
        for (i = 0 through n-1) {
              k = jth digit of A[i]
              Place A[i] at the end of group k
              Increase kth counter by 1
        }
        Replace the items in A with all the items in group 0,
        followed by all the items in group 1, and so on.
  }
```


3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
Sort
7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9
sorted
7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7
sorted
3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9
sorted

Radix Sort - Example

- Sort the following list ascendingly using Radix Sort:

  27, 78, 52, 39, 17, 46

- Base: 10, Number of digits: 2
- First Pass. The rightmost digit

0	1	2	3	4	5	6	7	8	9
							1**7**		
		5**2**				4**6**	2**7**	7**8**	3**9**

Combine after first pass: **52, 46, 27, 17, 78, 39**

Radix Sort - Example

- Second Pass.
  - The second rightmost digit of : 52, 46, 27, 17, 78, 39

0	1	2	3	4	5	6	7	8	9
	17	**2**7	**3**9	**4**6	**5**2		**7**8		

Resulting list: **17, 27, 39, 46, 52, 78**

Radix Sort - Analysis

- Radix Sort is O(n)

- What are the strength and weakness of this algorithm?

Radix Sort - Analysis

- Pros:
  - linear time complexity
  - stable sorting algorithm
  - efficient for sorting large numbers of integers/strings
  - easily parallelized
- Cons:
  - Not efficient for sorting floating-point numbers
  - requires a significant amount of memory
  - not efficient for small data sets

Radix Sort - Analysis

- Although the radix sort is O(n), it is NOT appropriate as a general-purpose sorting algorithm.
  - Memory needed?

- The Radix Sort is more appropriate for a linked list than an array.

Counting Sort

Counting Sort - Idea

- Counting sort sorts elements by storing the count of unique elements
- The count is stored in an auxiliary array
- The sorting is done by mapping the count as an index of the auxiliary array

Counting Sort - Idea

- Counting Sort – Assumption: data is going to be in the specific range [a, b]
- Sort the array A in Counting sort:
  - Step 1: Create the auxiliary array C from [a, b]
  - Step 2: Count the frequency of element C in A
  - Step 3: Create the expecting position of element from C
  - Step 4: Copy the element from A via the position array

Counting Sort - Idea

fit@hcmus

- Step 1, 2: Get the frequency of element

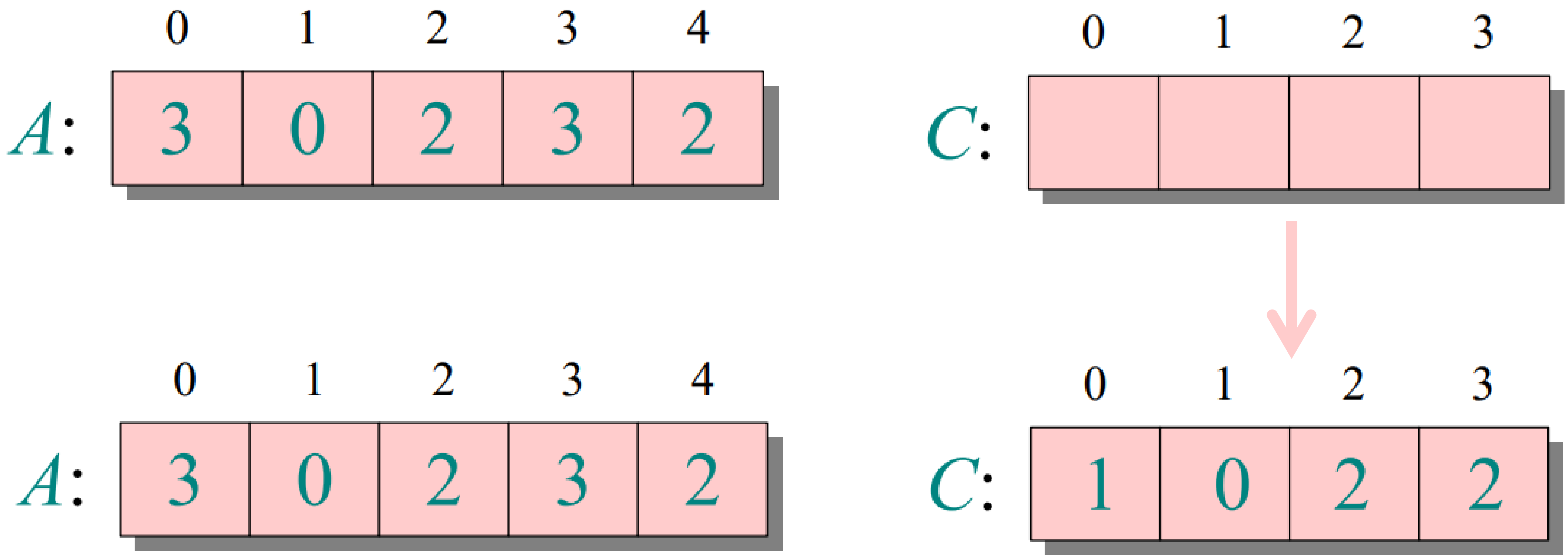
A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3

- Step 1, 2: Get the frequency of element

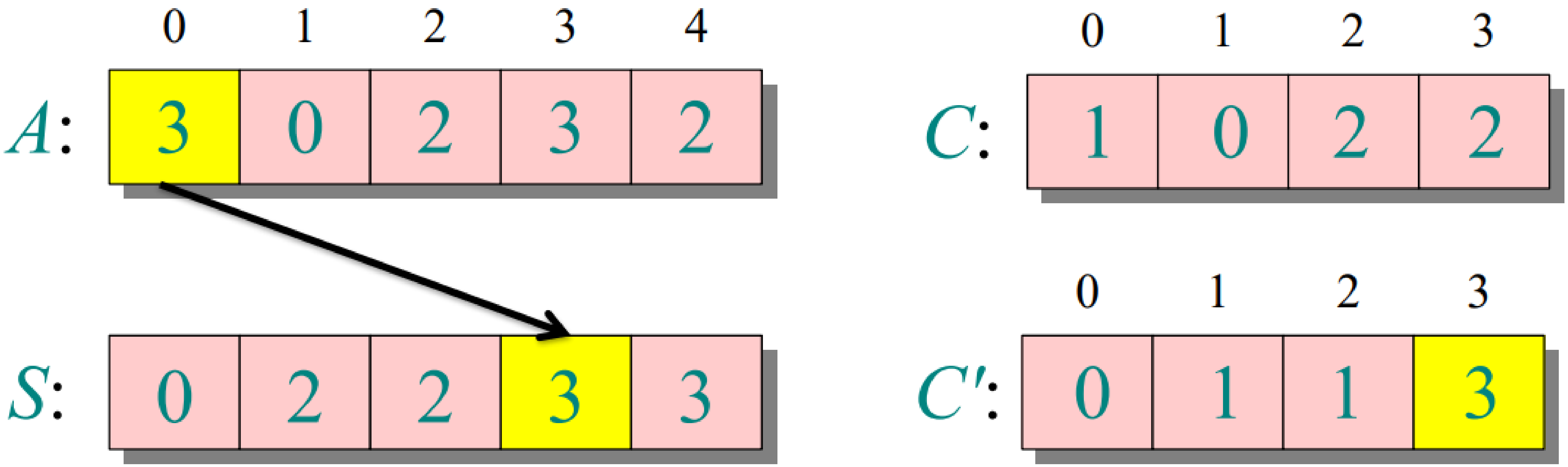


- Step 3: Calculate the expecting position

	0	1	2	3
C:	1	0	2	2

	0	1	2	3
C':	1	1	3	5

- Step 4: Copy the elements in A to result S



Counting Sort - Analysis

- Time Complexity: O(n + k)
  - n: number of original array
  - k: number of elements in the range [a, b]
- Space Complexity: O(k)
- Worst case: when data is skewed and range is large
- Best Case: When all elements are same
- Average Case: O(N+K) (N & K equally dominant)

Counting Sort – Pros and Cons

- Pros:
  - Stability
  - No comparison operation
  - Effectiveness as the range of the input is small compared to the number of elements
- Cons:
  - Limited to sorting integers or similar ones
  - Not in-place

Comparison of Sorting Algorithms

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)

Summary

- **Selection Sort** is $O(n^2)$ algorithm. Good in some particular case but it is slow for large problems.
- **Heap Sort** converts an array into a heap to locate the array's largest items, enabling to sort more efficient.
- **Quick Sort** and **Merge Sort** are efficient recursive sorting algorithms.
- **Quick Sort** is $O(n^2)$ in worst case but rarely occurs.
- **Merge Sort** requires additional storage.
- **Radix Sort** is $O(n)$ but not always applicable as not a general-purpose sorting algorithm.

Exercise

- Demonstrate the steps to sort the following list of integers ascendingly by Merge Sort and Quick Sort

5 2 7 6 1 9 4

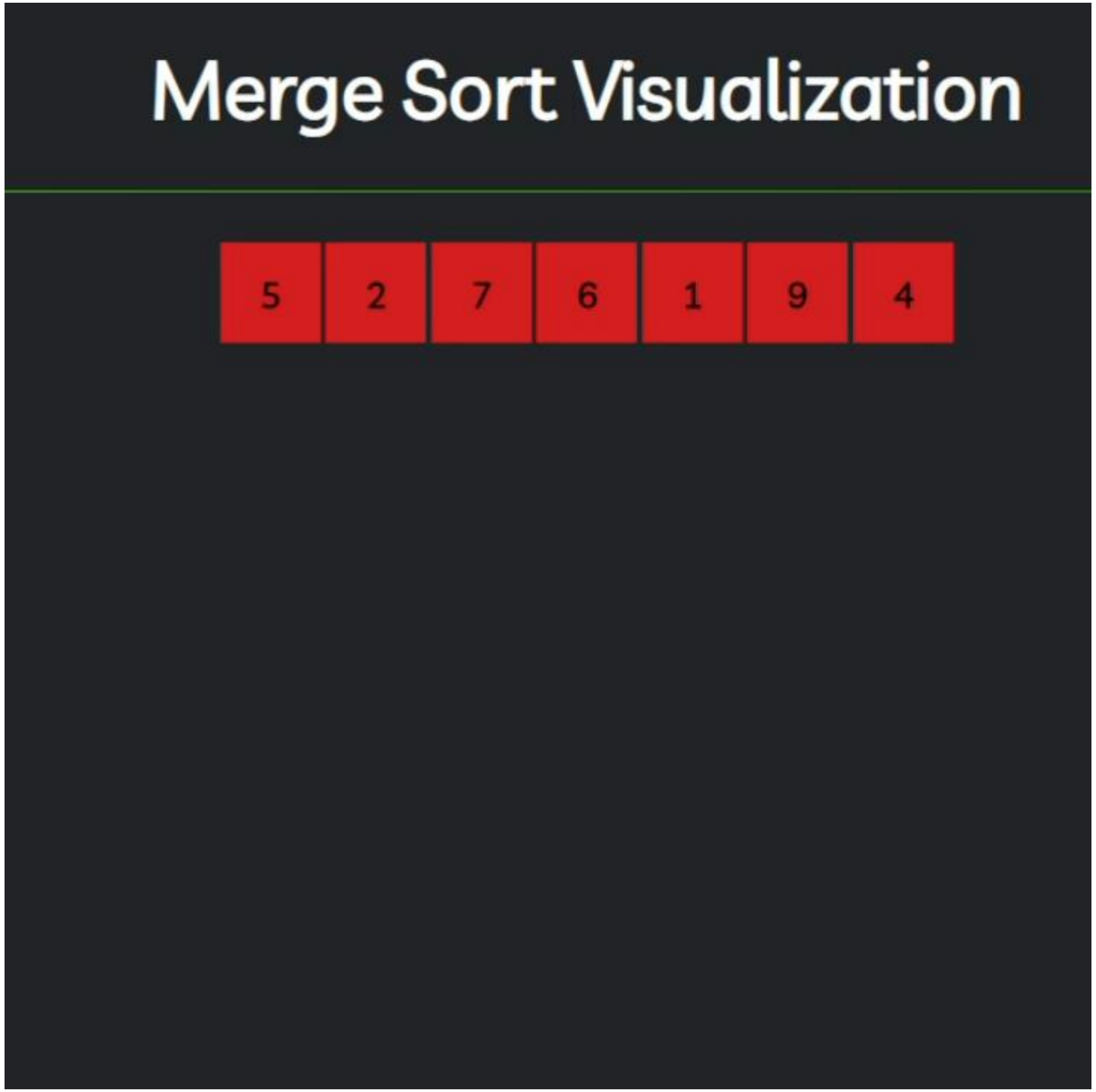
Exercise: Quick Sort

Exercise: Quick Sort

Exercise: Quick Sort

Exercise: Quick Sort

Exercise: Merge Sort



THANK YOU for YOUR ATTENTION