Sorting Algorithms

Contents

- Selection Sort
- Heap Sort
- Merge Sort
- Quick Sort
- Radix Sort

Overview

Sorting

- Sorting is:
  - A process that organizes a list of data into ascending/descending order

- Example:
  - List before sorting:

    {1, 25, 6, 5, 2, 37, 40}
  - List after sorting:

    {1, 2, 5, 6, 25, 37, 40}

Sorting

- Sort key: data item which determines order
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of some algorithms.
- Majority of programming projects use sorting, and, in many cases, the sorting cost determines the running time.

Sorting

- Some popular sorting algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort
  - Radix Sort

Selection Sort

Selection Sort - Idea

- Sort naturally the same as in real-life:
  - The list is divided into two sub-lists, *sorted* and *unsorted*, which are divided by an imaginary wall.
  - Find the **smallest element** from the unsorted sub-list and move to the correct position (swap it with the element at the beginning of the unsorted data.)
  - After each selection and swapping, increase the number of sorted elements and decrease the number of unsorted ones.
  - Loop those steps until the unsorted list has only 1 element.

Selection Sort

Input: (unsorted) a[] (*n* elements)

Output: (sorted) a[] (*n* elements)

- Step 1. Initialize *i* = 0.
- Step 2. Loop:
  - 2.1. Find the **smallest value** a[min] in the list with index from *i* to *n*-1 (a[i], …, a[n-1]).
  - 2.2. Swap a[min] and a[i]
- Step 3. Compare *i* with *n*:
  - If *i* < *n* then increase *i* by 1, back to step 2.
  - Otherwise, Stop.

Example

i = 0	15	2	8	7	3	6	9	17
i = 1	2	15	8	7	3	6	9	17
i = 2	2	3	8	7	15	6	9	17
i = 3	2	3	6	7	15	8	9	17
i = 4	2	3	6	7	15	8	9	17
i = 5	2	3	6	7	8	15	9	17
i = 6	2	3	6	7	8	9	15	17
i = 7	2	3	6	7	8	9	15	17

Analysis

- Which operation should be used for analysis?
- How many operations are there with size of the problem *n*?
- Best case? Worst case? Average case?

- We compare keys and move items in the algorithm
- Number of comparisons and the number of moves is used for analysis

Analysis

- Outer loop executes n-1 times

  Total Swaps: n-1

  Total Moves: 3*(n-1)

- Inner loop executes the size of unsorted part minus 1 (from 1 to n-1)

  Comparisons = 1+2+...+n-1 = n*(n-1)/2

- Time complexity of the algorithm is $O(n^2)$ in all cases

- Step 1. Initialize *i* = 0.
- Step 2. Loop:
  - 2.1. Find the **smallest value** a[min] in the list with index from *i* to *n*-1 (a[i],.., a[n-1]).
  - 2.2. Swap a[min] and a[i]
- Step 3. Compare *i* with *n*:
  - If *i* < *n* then increase *i* by 1, back to step 2.
  - Otherwise, Stop.

Analysis

- If sorting a very large array, selection sort algorithm is inefficient to use.
- The behavior of selection sort algorithm does not depend on the initial order of data.

- What is the advantage of this algorithm?

Heap Sort

Heap Structure

- Heap is a collection of *n* elements (a_0, a_1, …, a_{n-1}) in which
  - For every *i* ($0 \le i \le n/2-1$)

    $a_i \ge a_{2i+1}$

    $a_i \ge a_{2i+2}$
  - If $2i+2 \ge n$, just $a_i \ge a_{2i+1}$ need to hold
  - Condition does not apply to the second half as 2i+1 and 2i+2 are out of array

- Heap in above definition is called **max-heap** *(we also have **min-heap**)*

Heap Structure

- Examples:
  - A max-heap: 9, 5, 6, 4, 5, 2, 3, 3
  - A min-heap: 8, 15, 10, 20, 17, 12, 18, 21, 20

Heap Structure

- Property:
  - The first element of the max-heap is always the largest.

Heap Structure - Heap Construction

- Input: An array a[], *n* elements
- Output: A heap a[], *n* elements

Step 1. Start from the middle of the array (first half). Initialize *index* = (n - 1)/2

Step 2. while (*index* >= 0)

```
{
    heapRebuild at position index //heapRebuild(index, a, n)
    index = index - 1
}
```

Heap Structure – heapRebuild (pos, A, n)

- **Step 1.** Initialize *k* = pos, *v* = A[*k*], *isHeap* = false
- **Step 2.** while not *isHeap* and 2**k*+1 < *n* do

```
j = 2*k + 1 //first element
if j < n - 1 //has enough 2 elements
    if A[j] < A[j + 1] then j = j + 1 //position of the larger
    between A[2*k+1] and A[2*k+2]
if A[k] >= A[j] then isHeap = true
else
    swap between A[k] and A[j]
    k = j
```

Heap Construction - An Example

- Construct a heap from the following list:

2, 9, 7, 6, 5, 8

Heap Sort

- Idea is the same as Selection Sort.
- It has two stages:
  - Stage 1: **(heap construction)**. Construct a heap for a given array.
  - Stage 2: **(move max to right place)**. Move the first element to the last position and rebuild heap
    - Swap the first and the last element of the heap.
    - Decrease the heap size by 1.
    - Rebuild the heap at the first position.

Heap Sort

```
HeapSort(a[], n)
{
	heapConstruct(a, n);
	r = n - 1;
	while (r > 0)
	{
		swap(a[0], a[r]);
		heapRebuild(0, a, r);
		r = r - 1;
	}
}
```

Heap Sort - Analysis

- Best case, Worst case, Average case are the same.
- The order of this algorithm: $O(n\log_2 n)$

Merge Sort

Merge Sort - Idea

- Merge Sort follows **divide-and-conquer** strategy.
- It is a recursive algorithm that
  - Divides the list into halves,
  - Sorts each halve separately, and
  - Then merges the sorted halves into one sorted array.

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

Merge Sort - Idea

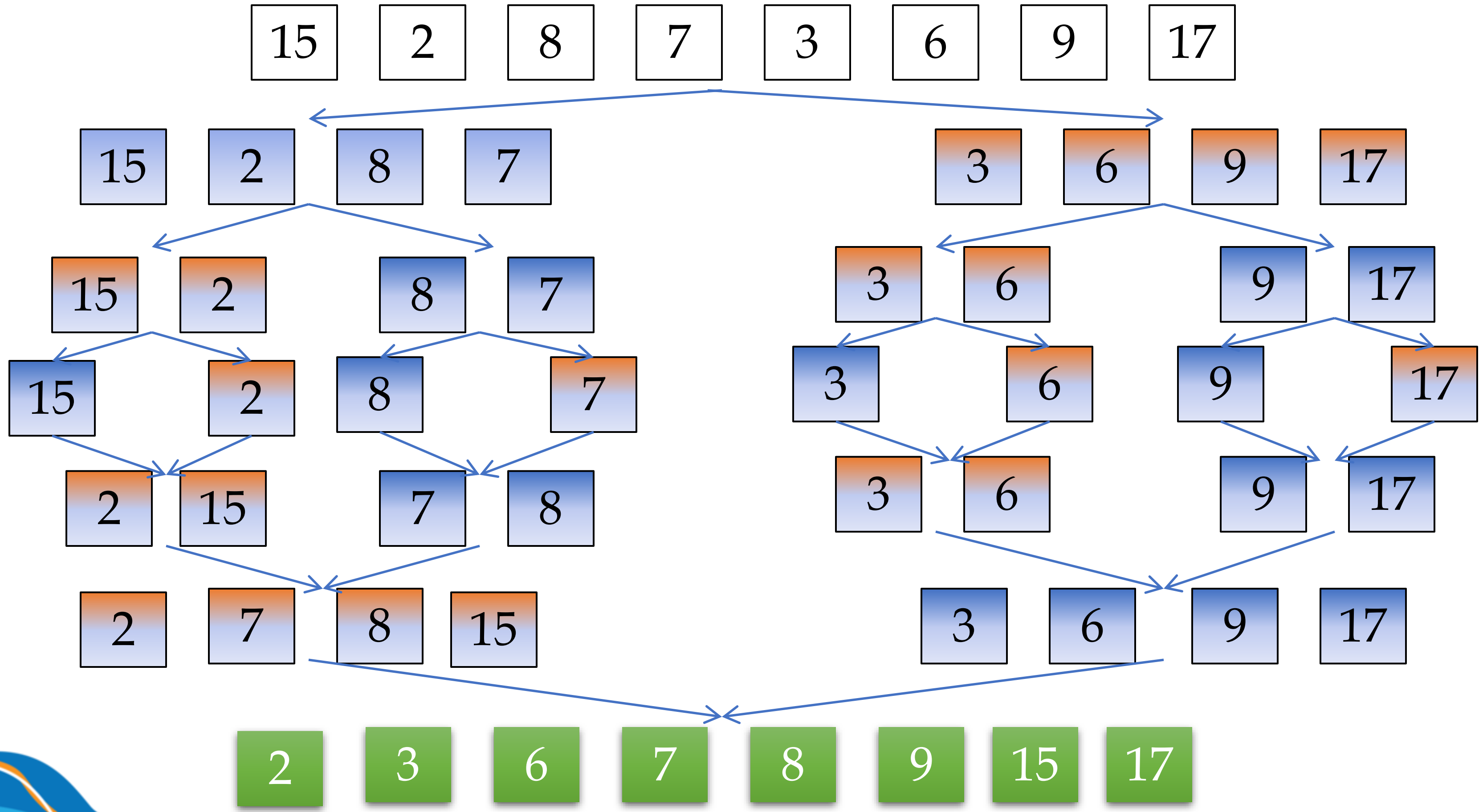
- **Merge** procedure:
  - Goal: Merge two ordered lists into an ordered list.
  - Input: two ordered lists A[] (*n* elements), B[] (*m* elements)
  - Output: a new ordered list C[] (*n* + *m* elements) (containing all elements of A and B).
  - Example:
    - A = {1, 5, 7, 9}, B = {2, 9, 10, 12, 17, 26}; C = {1, 2, 5, 7, 9, 9, 10, 12, 17, 26}

Merge Sort

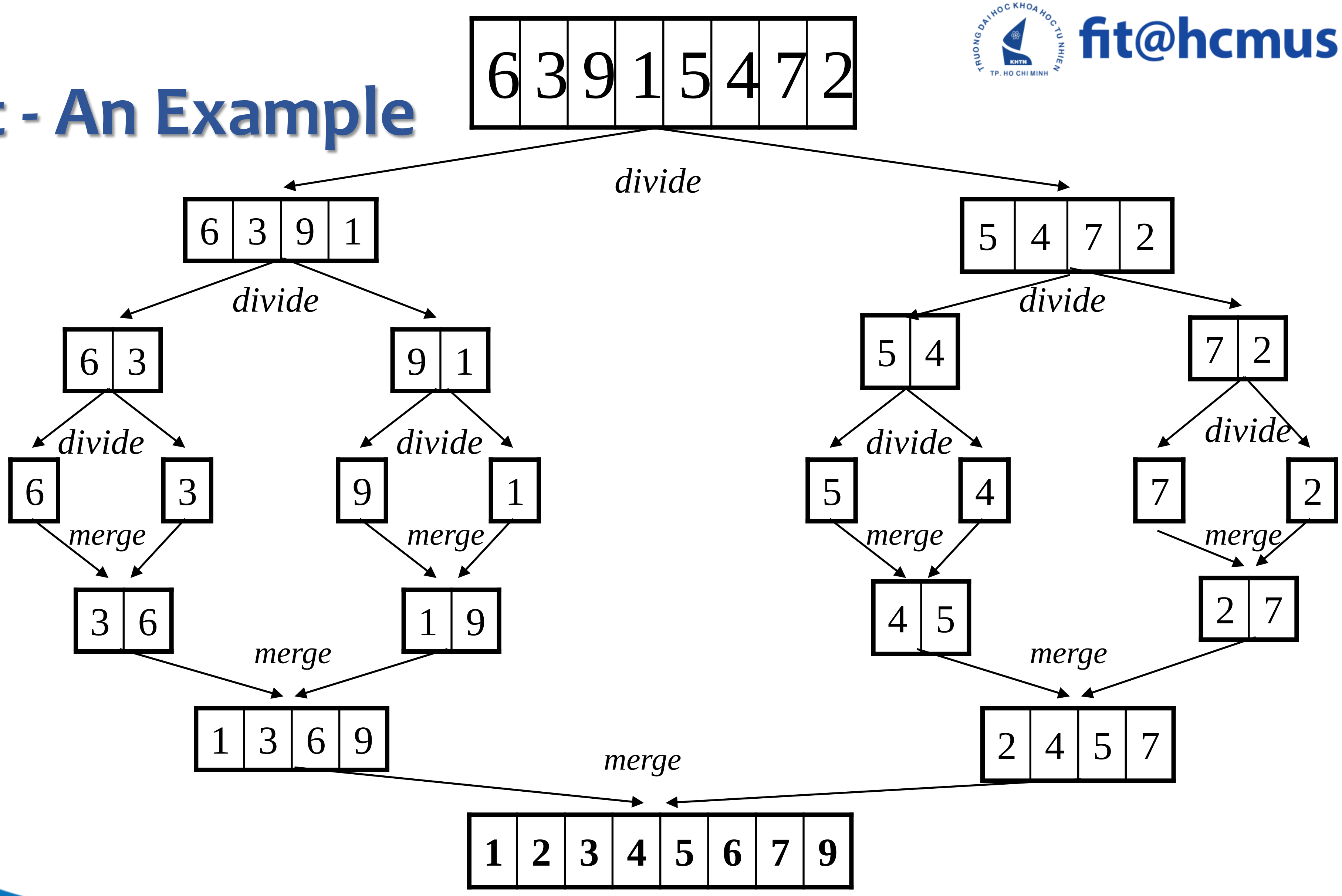
- Input: A[], left, right (list A from index left to right).
- Output: (sorted) A[] (from left, to right)

```
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);
   }
}
```

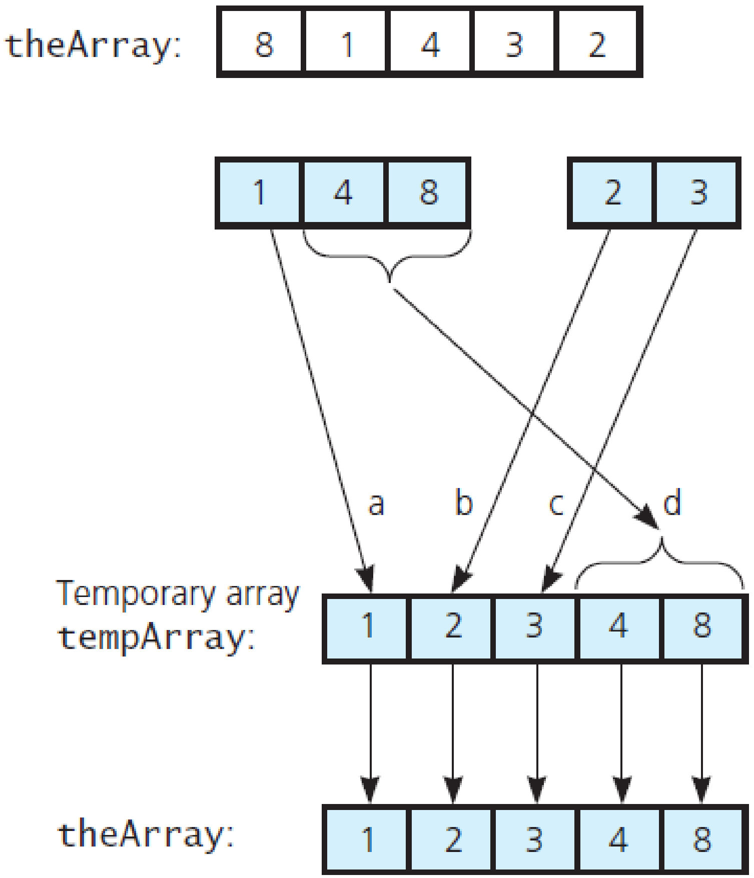
Merge Sort - An Example



Merge Sort - An Example



Merge Sort - An Example



Divide the array in half

Sort the halves

Merge the halves:

a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

Analysis

- Merge Sort is an 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 (complexity of O(n))

Quick Sort

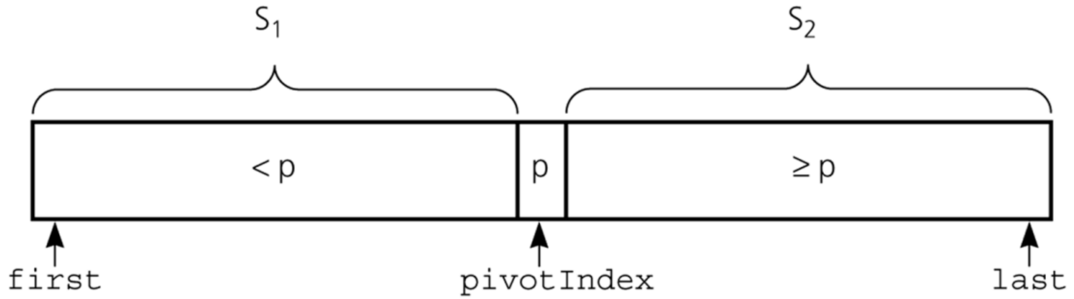
Quick Sort - Idea

- Like Merge Sort, Quick Sort is based on **divide-and-conquer** strategy.

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

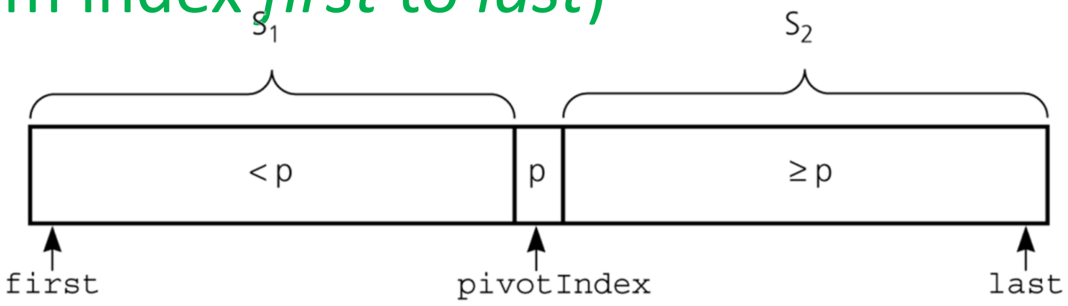
Quick Sort - Idea

- The algorithm consists of the following three steps:
  - **Divide:** ***partition*** the list.
    - To partition the list, we first choose an element from the list, namely ***pivot***, for which we expect a half of the list will come before and the other half after.
    - Then we partition the elements so that ones less than the pivot go to one sub-list and ones greater than the pivot go to the other.
  - **Conquer:** recursively sort the sub-lists.
  - **Combine** the sorted sub-lists together.



Quick Sort

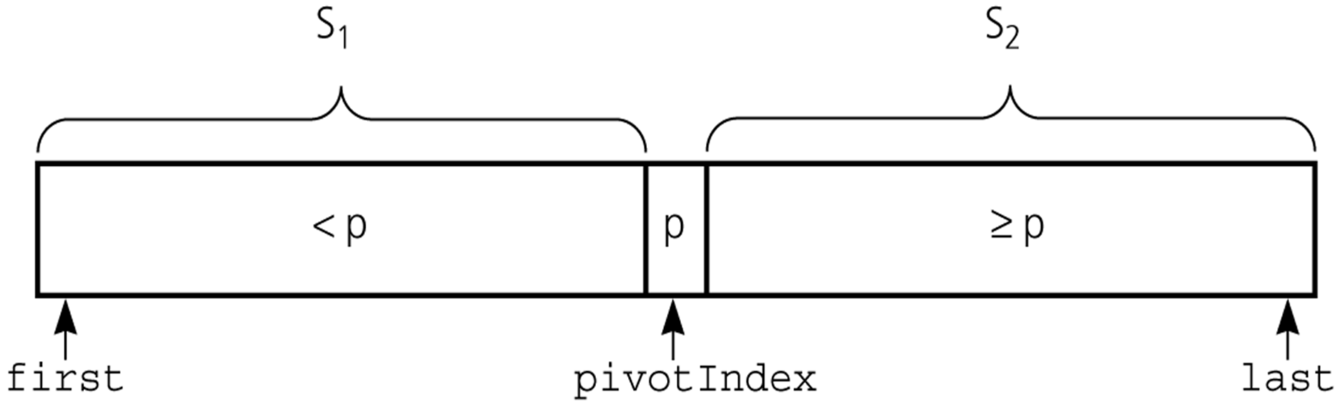
- Input: A[], first, last (sort the list A[] from index *first* to *last*)
- Output: sorted list A[first..last]




```
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 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 problems are solved recursively, our bigger sorting problem is solved.

Quick Sort - Partition

- 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.

- Which array item should be selected as pivot?
  - We hope that we will get a good partitioning.

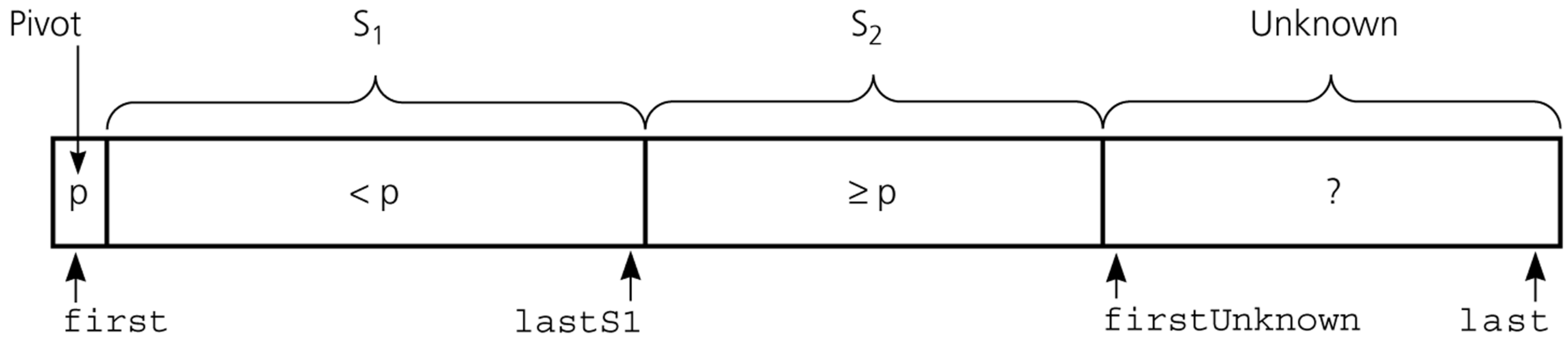
Quick Sort - Partition

- 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.

- Which array item should be selected as pivot?
  - 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).

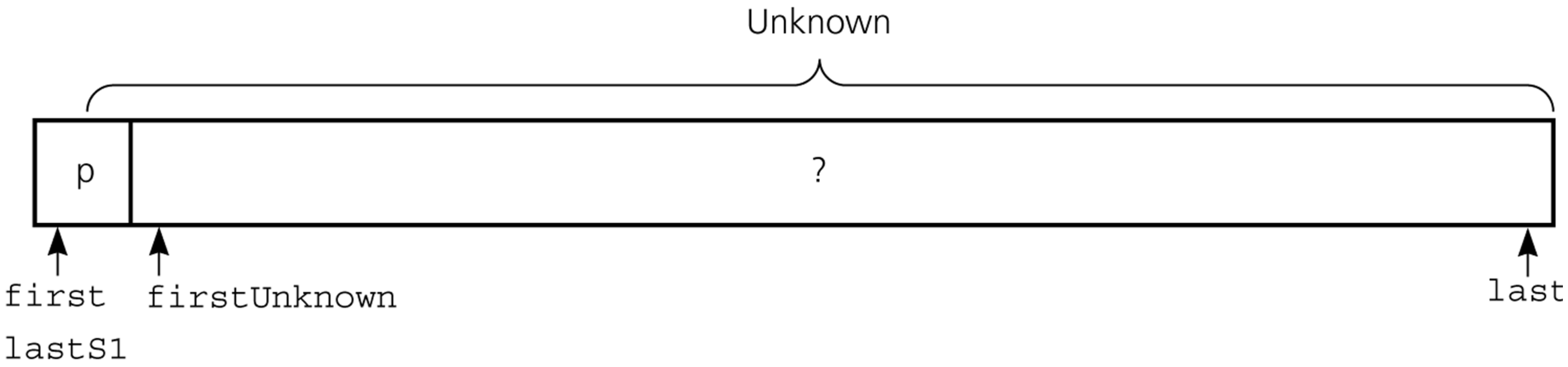
Quick Sort - Partition

- 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 the element at position *firstUnknown* is **less than** *pivot* then move that element to *S1*

Otherwise, move that element to *S2*

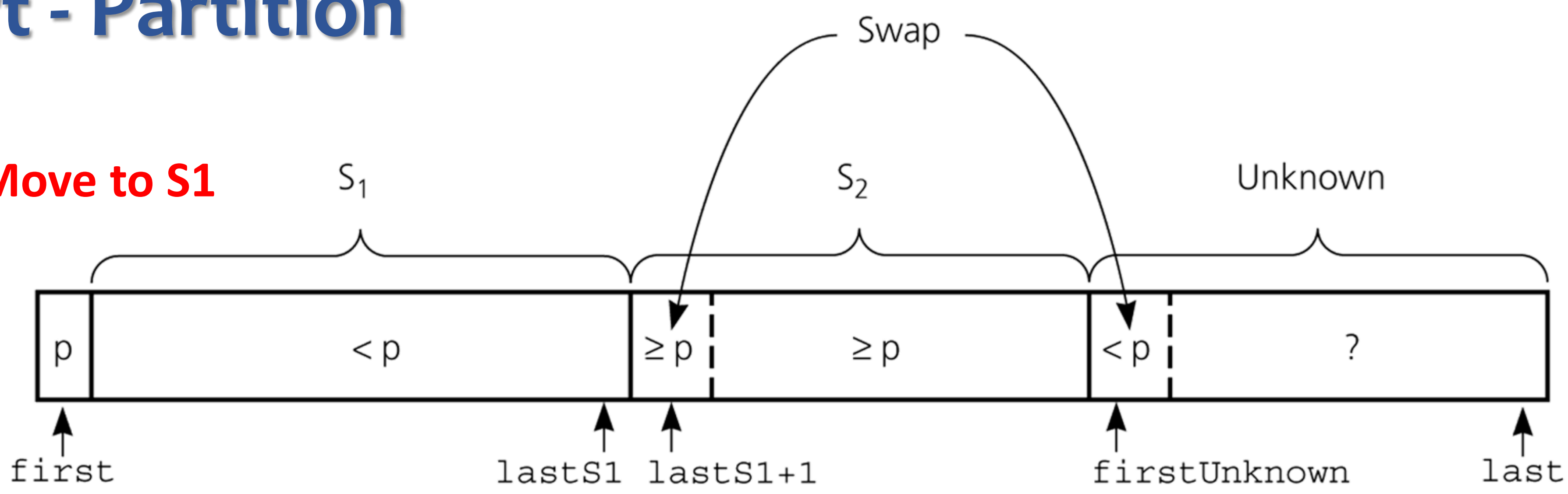
1.2 *firstUnknown* = *firstUnknown* + 1 //next element

Step 2. Move *pivot* to the correct position (between *S1* and *S2*): Swap two elements at *lastS1* and *first*.

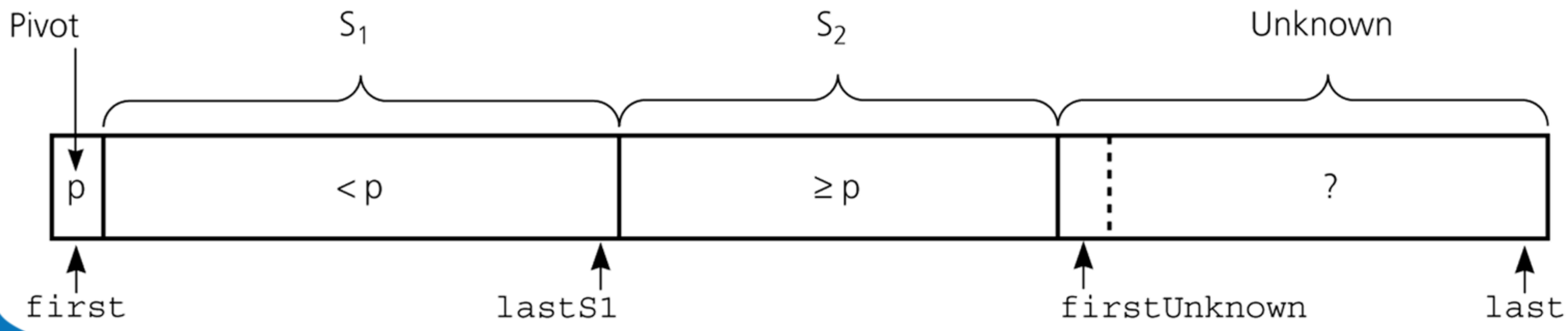
Step 3. pivotIndex = lastS1

Quick Sort - Partition

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

Pivot	S2	Unknown			
27	38	12	39	27	16

Pivot	S1	S2	Unknown		
27	12	38	39	27	16

Quick Sort - Partition

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

Pivot	S1	S2	Unknown		
27	12	38	39	27	16

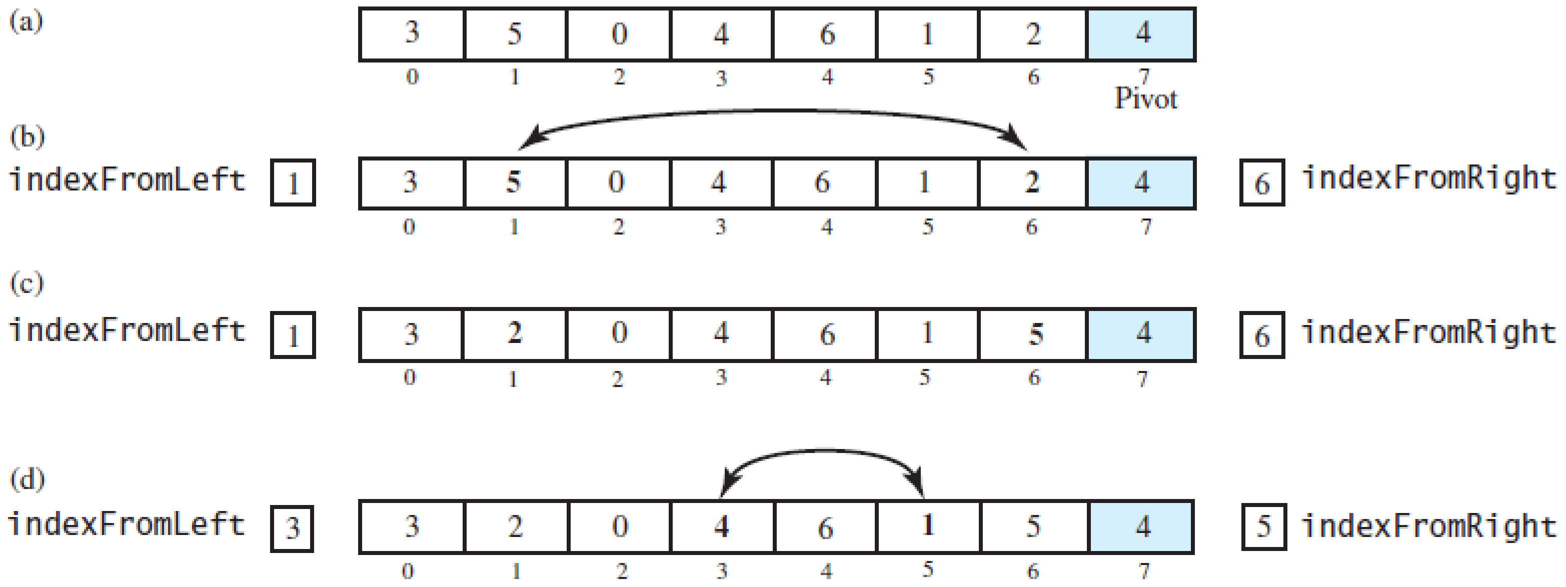
Pivot	S1	S2			U.K
27	12	38	39	27	16

Pivot	S1		S2		
27	12	16	39	27	38

S1		Pivot	S2		
16	12	27	39	27	38

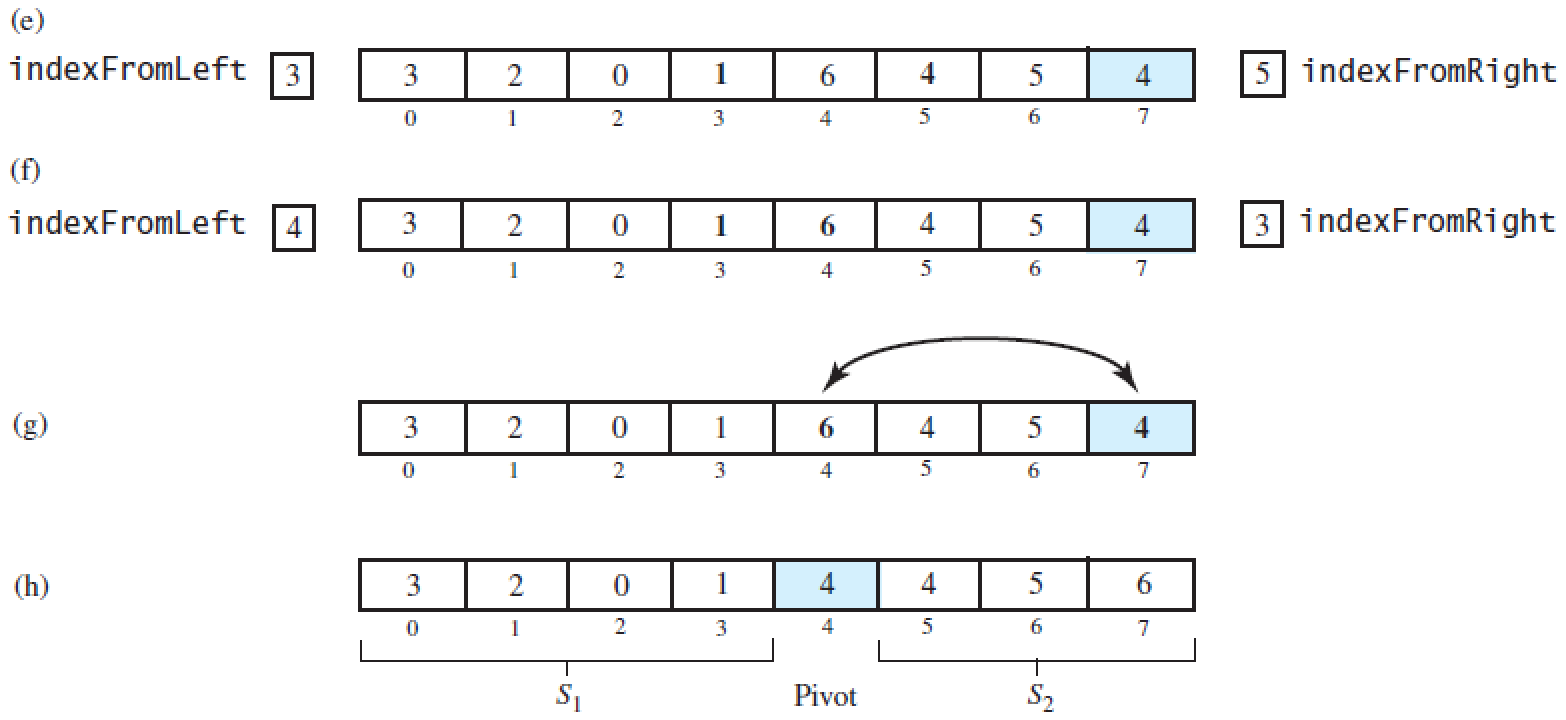
Quick Sort - Partition

- Another technique



Quick Sort - Partition

- Another technique



Quick Sort - Partition

- Median-of-three pivot selection

Analysis

- Worst case: **$O(n^2)$**
- Quick Sort is **$O(n\log_2 n)$** in the best case and average case.
- 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.

Radix Sort

Radix Sort

- Radix Sort algorithm is different from the other sorting algorithms that we have learnt.
  - It DOES NOT use key comparisons to sort an array.

Radix Sort - Idea

- Treats each element as a character string.
- Repeat (*for all characters from the rightmost to the leftmost*)
  - Groups elements 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 sorting process will be completed.

Radix Sort

```
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 = j^th digit of A[i]
          Place A[i] at the end of group k
          Increase counter for group k by 1
      }
      Replace the items in A with all the items in group 0,
  group 1, …, group k in orders.
  }
```

Radix Sort - An 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
							17		
		52				46	27	78	39

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

Radix Sort - An 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	27	39	46	52		78		

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

Analysis

- Time complexity of radix Sort is O(n)

- 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.

Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Radix sort	n	n
Treesort	n^2	$n * \log n$
Heapsort	$n * \log n$	$n * \log n$

Summary

- Selection Sort is $O(n^2)$ algorithm. Good in some particular cases 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.

Summary

- 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.

Thank you for your listening

Exercises

- Using the Selection Sort, demonstrate the steps to sort the following list of integers DESCENDENLY:

5, 9, 2, 7, 4, 3, 6, 2, 3, 8

- Applying the Heap Construction algorithm, demonstrate the steps to create a max-heap from the above list.
- Using the Merge Sort, demonstrate the steps to sort the following list of integers ASCENDENLY.
- When using the “median-of-three” pivot selection technique on the above list, what is value of the pivot?