

# Sorting Algorithms

1

## Contents

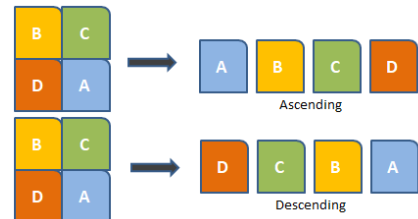
- Selection Sort
- Heap Sort
- Merge Sort
- Quick Sort
- Radix Sort

2

## Overview

## Sorting

- Sorting is:
  - A process organizes a collection 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
- Internal: data **fits in** memory
- External: data must reside on **secondary** storage
- In-place (algorithm): sorts the data **without using any additional** memory.
- Stable (algorithm): **preserves** the relative order of data elements.

## Classification of Sorting Algorithms

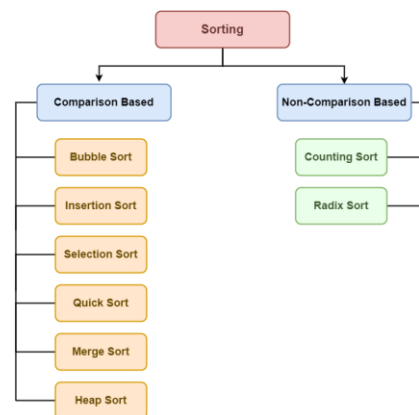
- **Memory usage:** in-place sort / not in-place sort
  - **In-place** (algorithm): sorts the data without using any additional memory.
- **By stability:** maintain the relative order of the records with equal keys
- **Comparison:** utilize comparison or not?
- **Adaptability:** whether the pre-sorted-ness of the input affects the running time or not
- **Data Location:** Internal or external
  - **Internal:** data **fits in** memory
  - **External:** data must reside on **secondary storage**

## Sorting

- We will analyze only **internal** sorting algorithms.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- A comparison-based sorting algorithm makes ordering decisions only based on comparisons.

## Sorting

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

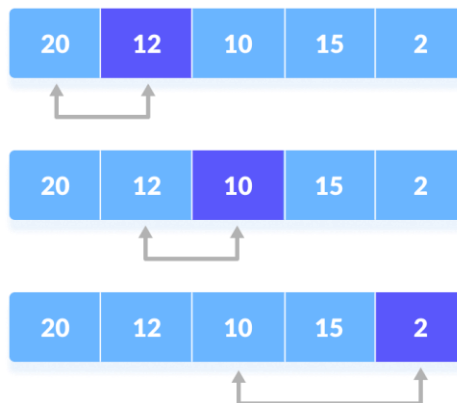
Input: (unsorted)  $a[]$  ( $n$  elements)

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

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

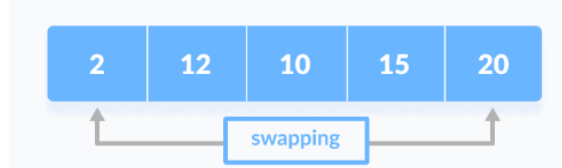
## Selection Sort

- Step 1: find the smallest value of the (unsorted) list

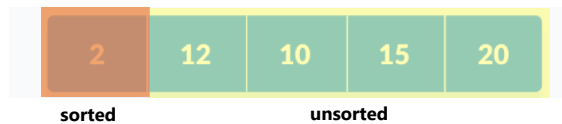


## Selection Sort

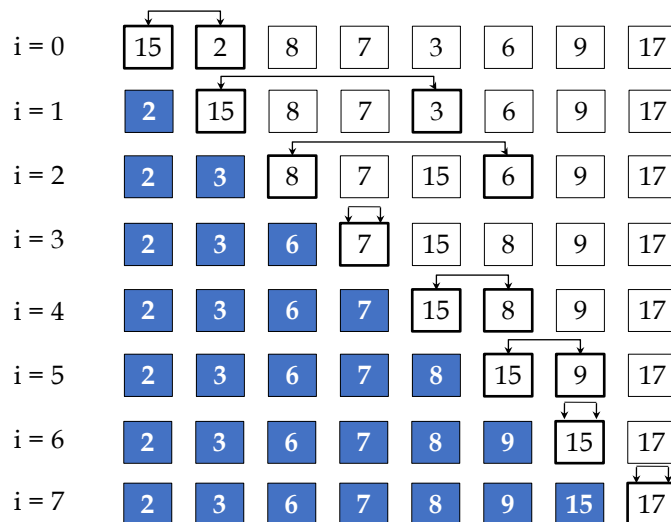
- Step 2: the smallest value is placed in the front of the unsorted list



- Step 3: repeatedly step 1-2 for the unsorted parts



## Example



## Analysis

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

## Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).
- To analyze a sorting algorithm we should count the number of key comparisons and the number of moves.
  - Ignoring other operations does not affect our result.
- The outer for loop executes  $n-1$  times. We invoke **swap** function once at each iteration.

Total Swaps:  $n-1$

Total Moves:  $3*(n-1)$

(Each swap has three moves)



## Analysis

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to  $n-1$ ), and in each iteration we make one key comparison.  
number of key comparisons =  $1+2+\dots+n-1 = n*(n-1)/2$

## Analysis

- The best case, the worst case, and the average case of the selection sort algorithm are same.
- Order of the algorithm:  $O(n^2)$ .

## Analysis

- If sorting a very large array, selection sort algorithm probably too inefficient to use.
- What is the advantage of this algorithm?

## Analysis

- The behavior of the selection sort algorithm does not depend on the initial organization of data.
- Although the selection sort algorithm requires  $O(n^2)$  key comparisons, it only requires  $O(n)$  moves.
- A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

# Heap Sort

# Heap Structure

- Definition (*array-based representation*):

- Heap is a collection of  $n$  elements ( $a_0, a_1, \dots, a_{n-1}$ ) in which every element (at position  $i$ ) in the **first half** is greater than or equal to the elements at position  $2i+1$  and  $2i+2$ .

(if  $2i+2 \geq n$ , just  $a_i \geq a_{2i+1}$  satisfied).

- i.e., for every  $i$  ( $0 \leq i \leq n/2-1$ )

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

$$a_i \geq a_{2i+2}$$

- Heap in above definition is called **max-heap**. (We also have **min-heap** structure).

## 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
- Give some more examples of:
  - A max-heap with 8 elements.
  - A max-heap with 11 elements.
  - A min-heap with 7 elements.

## 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  $index = n / 2 - 1$

**Step 2.** while ( $index \geq 0$ )

```
{
    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  $A[k] \geq A[j]$  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

- An interesting sorting algorithm discovered by J.W.J. Williams (in 1964).
- Idea is same as Selection Sort.
- It has two stages:
  - Stage 1: **(heap construction)**. Construct a heap for a given array.
  - Stage 2: **(maximum deletion)**. Apply the maximum key deletion  $n-1$  times to the remaining heap
    - Exchange 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); //heapConstruct(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

## Divide-and-Conquer

- 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.
  - **Combine:** **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.

## Merge Sort - Idea

- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each half separately, and
  - Then merge 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 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  $B$ ).
- Example:
  - $A = \{1, 5, 7, 9\}$ ,  $B = \{2, 9, 10, 12, 17, 26\}$ ;  $C = \{1, 2, 5, 7, 9, 9, 10, 12, 17, 26\}$
- Propose the efficient algorithm.

## Merge Sort - Idea

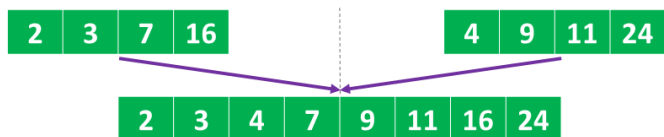
### ▪ Merge procedure:

#### ▪ Input:

- $A = \{2, 3, 7, 16\}$ ,
- $B = \{4, 9, 11, 24\}$ ;

#### ▪ Output:

- $C = \{2, 3, 4, 7, 9, 11, 16, 24\}$



## Merge Sort

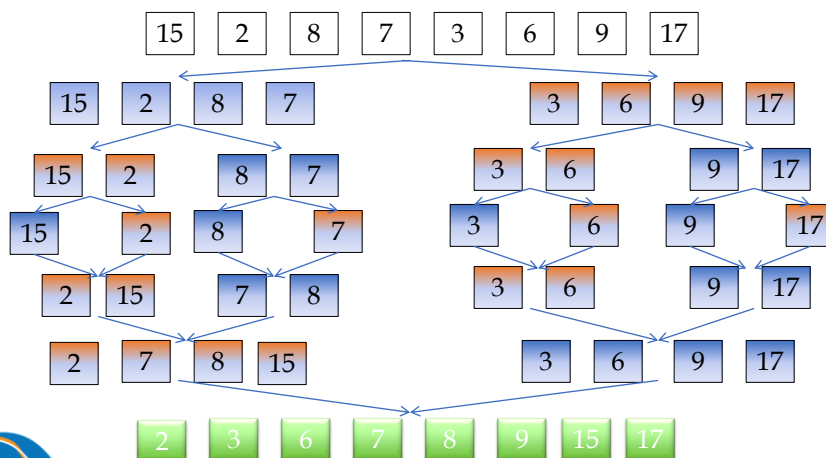
- Input: A[], left, right (list A from index left to right).
- Output: (Ordered) A[] (from left, to right)

```

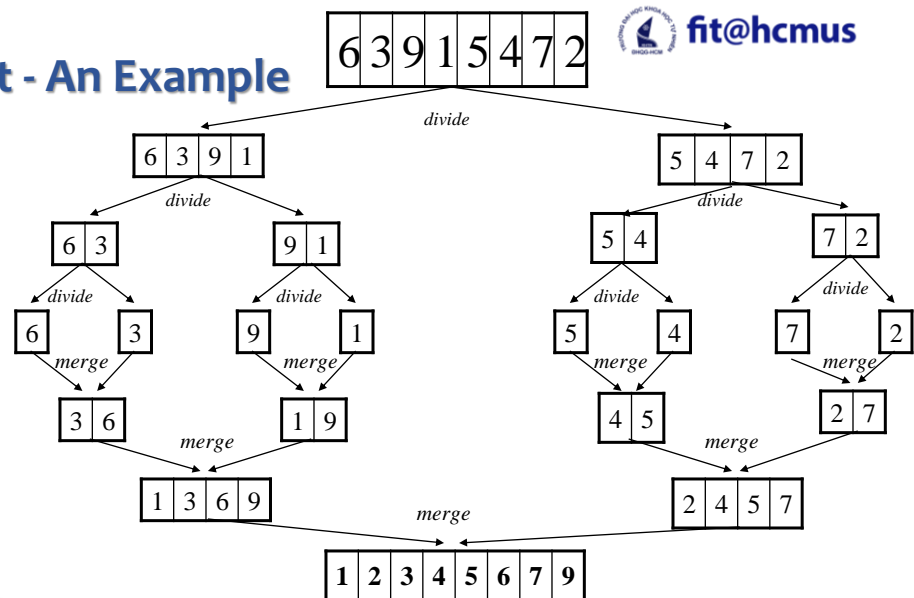
MergeSort(A[], left, right)
{
    if (left < right) {
        mid = (left + right)/2;
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
    }
}

```

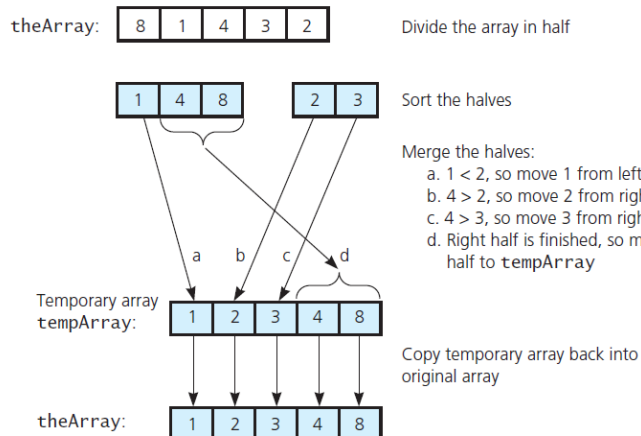
## Merge Sort - An Example



## Merge Sort - An Example



## Merge Sort - An Example



## 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 ( $O(n)$ )

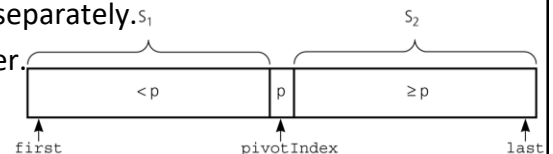
## Quick Sort

## Quick Sort - Idea

- 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,
  - Then, it **sorts** the parts **independently**,
  - Finally, it **combines** the sorted subsequences 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 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.
  - **Recursion:** Recursively sort the sub-lists separately.
  - **Conquer:** Put the sorted sub-lists together.



## Quick Sort

- Input:  $A[]$ , first, last (Sort the list  $A[]$  from index *first* to *last*)
- Output: Ordered list  $A[\text{first}..\text{last}]$

**QuickSort**( $A[]$ , first, last)

```
if (first < last) {
```

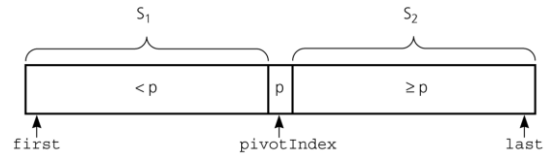
```
    Select a pivot  $p$  from  $A[]$ .
```

```
    pivotIndex = Partition( $A$ , first, last) //Partition  $A[]$  into 2  
    sub-lists  $S_1$  (first ... pivotIndex-1),  $S_2$  (pivotIndex+1...last)
```

```
    QuickSort ( $A$ , first, pivotIndex-1) //Sort  $S_1$ 
```

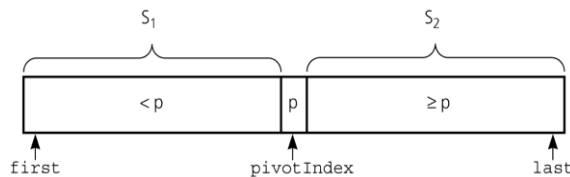
```
    QuickSort ( $A$ , pivotIndex + 1, last) //Sort  $S_2$ 
```

```
}
```



## 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.

## 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.

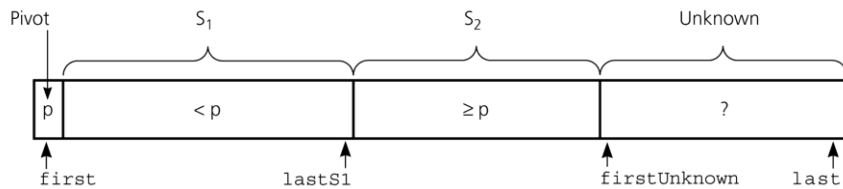
## Quick Sort - Partition

- 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).
  - We can use different techniques to select the pivot.



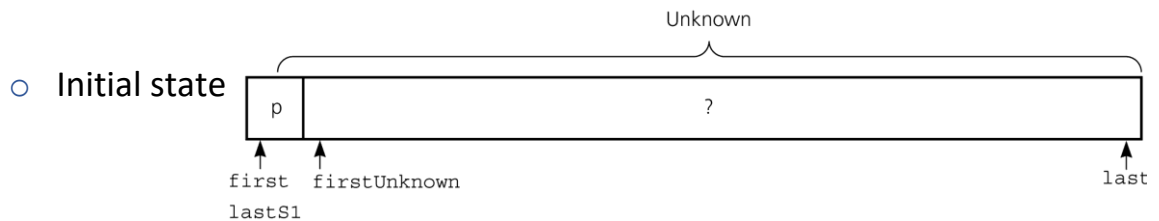
## Quick Sort - Partition

- Partitioning uses two more variables:
  - `lastS1`: the last index of  $S_1$  (the elements in  $A$  less than  $p$ ).
  - `firstUnknown`: the first index of Unknown.
- Partitioning takes place when **`firstUnknown`  $\leq$  `last`**.



## Quick Sort - Partition

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





## 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

↑                                      ↑

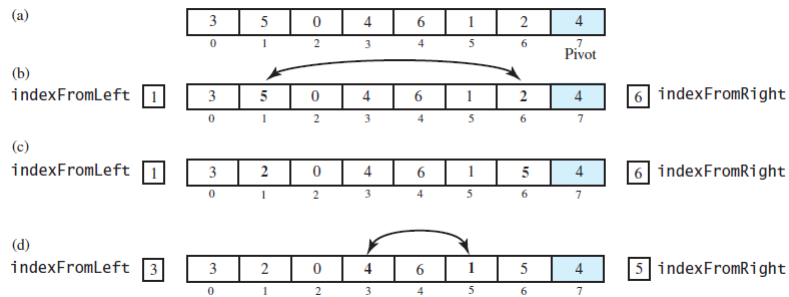
Pivot	S1		S2		
27	12	16	39	27	38

↑                                      ↑

S1		Pivot	S2		
16	12	27	39	27	38

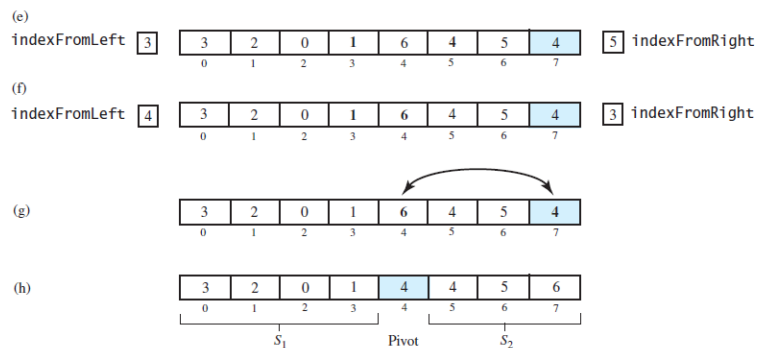
## Quick Sort - Partition

- Another technique



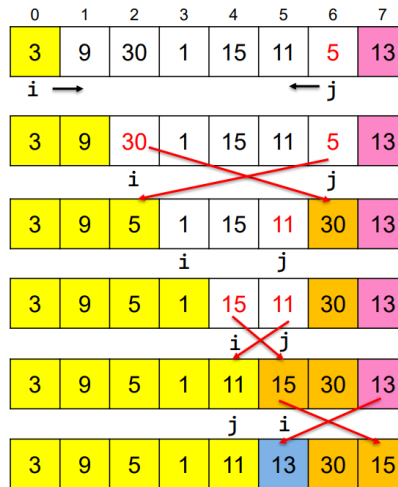
## Quick Sort - Partition

- Another technique



## Quick Sort - Partition

- Another technique



An inversion occurs:

- $A[j] < \text{pivot}$
  - $A[i] > \text{pivot}$
- Swap  $A[i], A[j]$

- X  $A[i] \leq \text{pivot}$
- X  $A[j] > \text{pivot}$
- X Pivot
- X Sorted item

Finally, swap  $A[i]$  and pivot  
→ *pivot is at correct position*

## 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.
- **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.



## Radix Sort

## Radix Sort

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

## Radix Sort - Idea

- 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.

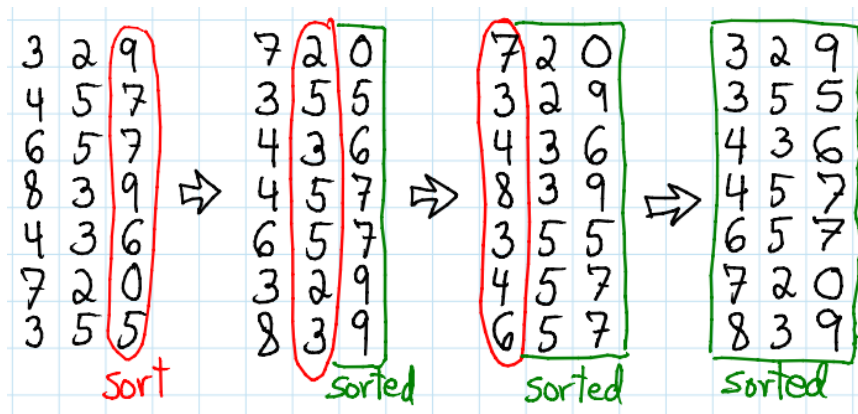
## Radix Sort

```

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 = 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.
  }

```

## Radix Sort





## 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

- Radix Sort is  $O(n)$
- What are the strength and weakness of this algorithm?

## 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.

## Comparison of Sorting Algorithms

	<u>Worst case</u>	<u>Average case</u>
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 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.

# Questions and Answers

fit@hcmus | DSA | 2023

79