AAI2007 Introduction to Algorithms

Week 10: Sorting Algorithms

Instructor: Jinyoung Han (jinyounghan@skku.edu)

In This Lecture

Outline

1. Sorting
2. Simple Sorting Algorithms
3. Advanced Sorting Algorithms

01. Sorting

Sorting

- Enumerate data in an ascending or descending order
  - Sorting is one of the most important algorithms in computer science as well as all the other science & technology areas

- Essential in searching!
  - An example, what if words are not ordered in a dictionary?

01. Sorting

Record

- A record
  - To be sorted
  - Consists of multiple fields
  - Key field: identifier of a record

- Example
  - A student's record consists of
    - Name, id, address, phone number, …
    - "id" can be a key field

01. Sorting

Sorting Algorithms

- No omniscient and optimal sorting algorithm
  - Depending on situations
- Different applications need to consider appropriate sorting algorithms
  - # records
  - Size of records
  - Characteristics of keys (e.g., character, integer, complex number, …)
  - (Memory) internal or external sorting
- Evaluation criteria
  - # comparisons
  - # moves

01. Sorting

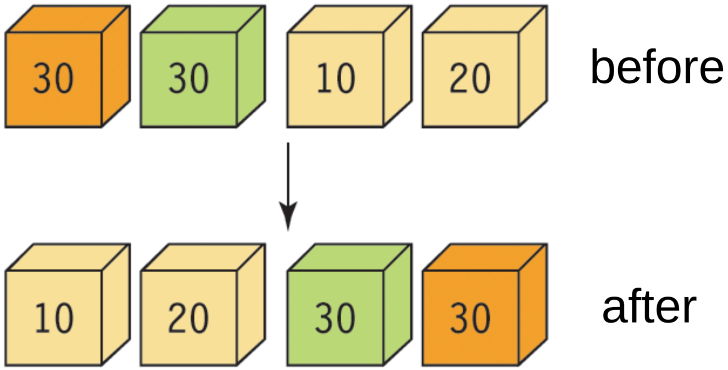
Sorting Algorithms

- Simple and inefficient: insertion sorting, selection sorting, bubble sorting, etc.
- Complex but efficient: quick sorting, heap sorting, merge sorting, radix sorting, etc.

- Internal sorting: all the data, stored in main memory, are sorted
- External sorting: most of data are stored in external devices, and main memory partly

Stability

- Stability of sorting
  - If there are multiple records having same key values, after sorting, the relative order of them does not change
  - An example of low stability



To pursue a stability of sorting, insertion sorting or merge sorting can be used!

Simple Sorting Algorithms

02. Simple Sorting Algorithms

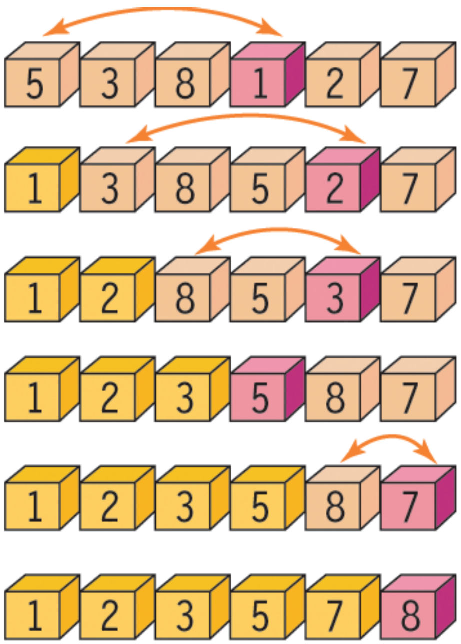
Selection Sorting

- Algorithm
  - Assuming, a left list as sorted and a right list as non-sorted
  - Initially, the left list is empty, and all the data to be sorted belong to the right list
  - Select the minimum value in the right list, and put it in the left list
  - Increase the size of the left list / decrease the size of the right list
  - If the right list becomes empty, done

Left list	Right list	Description
()	(5,3,8,1,2,7)	Initial state
(1)	(5,3,8,2,7)	Select 1
(1,2)	(5,3,8,7)	Select 2
(1,2,3)	(5,8,7)	Select 3
(1,2,3,5)	(8,7)	Select 5
(1,2,3,5,7)	(8)	Select 7
(1,2,3,5,7,8)	()	Select 8

Section Sorting

- In-place sorting
    - Just use an input array, i.e., do not use additional space
    - If a minimum value is found, exchange it with the first data
    - Among the remaining data, except the first data, select the next minimum value, and exchange it with the second data
    - Iterate until it is done



Section Sorting

- Pseudo code

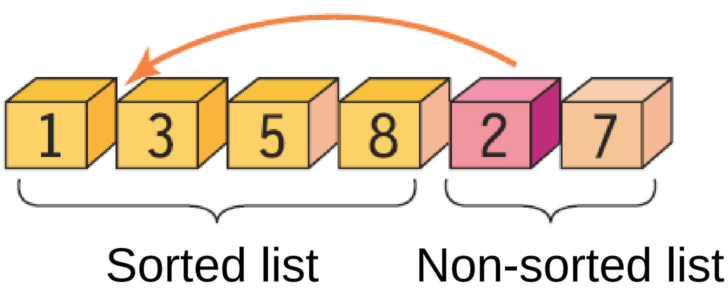
```
selection_sort(A, n)

for i←0 to n-2 do:
        least ← index of the smallest data among A[i], A[i+1],..., A[n-1];
        exchange A[i] and A[least];
        i++;
```

Section Sorting

- Time complexity
  - O(n^2)
- Not stable
  - For the records with same keys, the relative order may change

Insertion Sorting

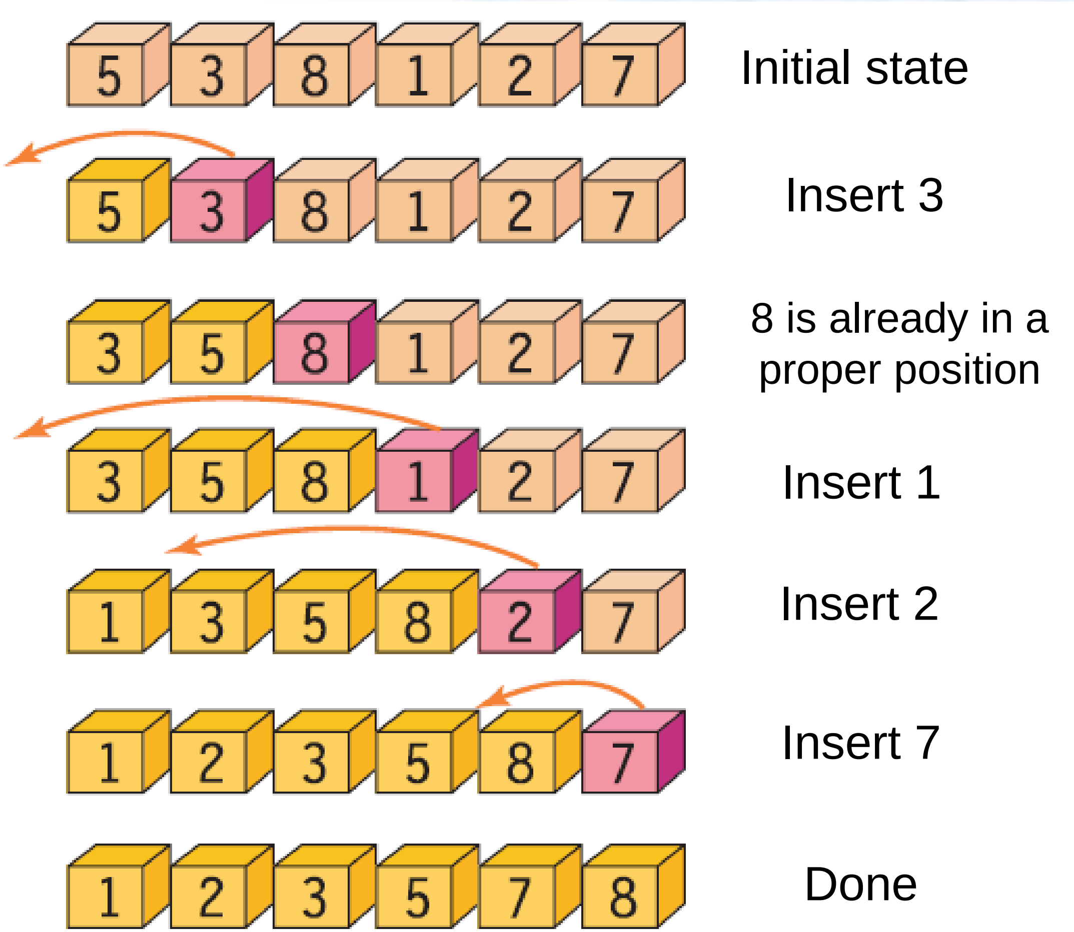
- Idea
  - Similar to sort cards in a hand
  - Insert a new card into the appropriate position among the existing cards
- Insertion sorting
  - Iterate to insert a new record into the appropriate position among the sorted existing list



02. Simple Sorting Algorithms

Insertion Sorting

- Algorithm



02. Simple Sorting Algorithms

Insertion Sorting

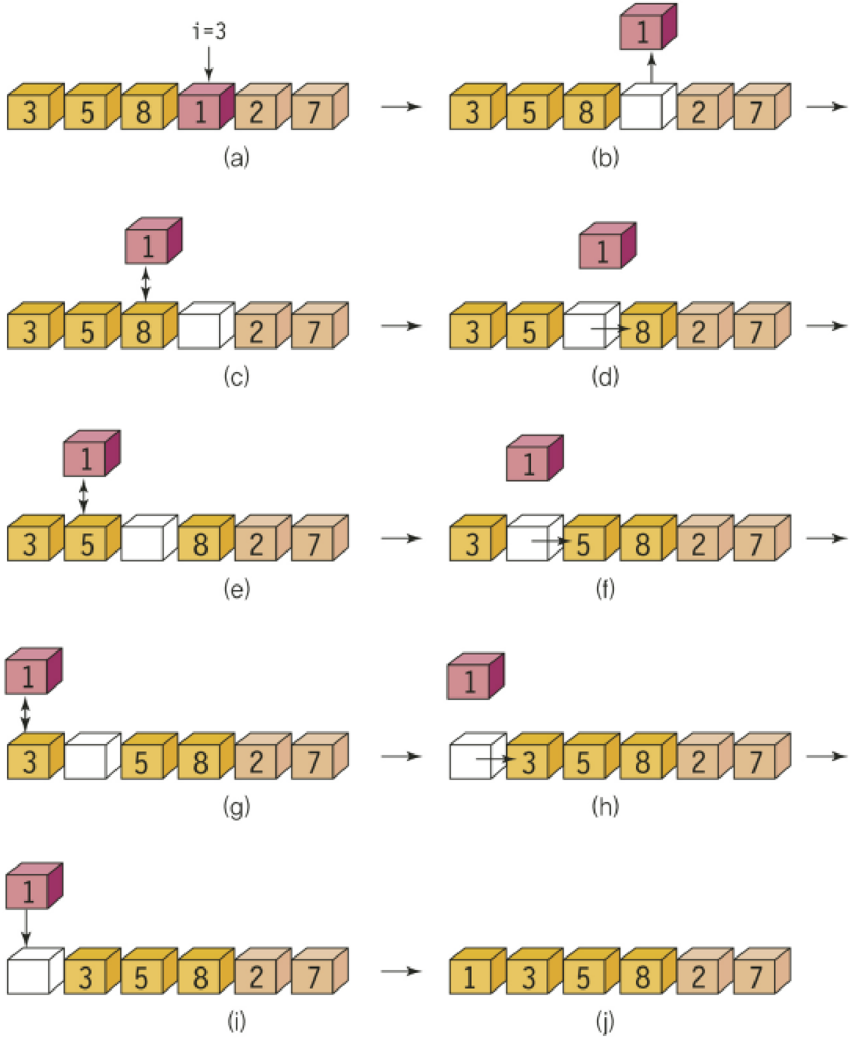
- Pseudo code

```
insertion_sort(A, n)

1. for i ← 1 to n-1 do
2.         key ← A[i];                      // key is the value to be inserted
3.         j ← i-1;
4.         while j≥0 and A[j]>key do        // investigate from i-1 to 1
5.                 A[j+1] ← A[j];
6.                 j ← j-1;
7.         A[j+1] ← key                     // since A[j] < key, j+1 is the position
                                              where key is inserted
```

02. Simple Sorting Algorithms

Insertion Sorting

- Example



02. Simple Sorting Algorithms

Insertion Sorting

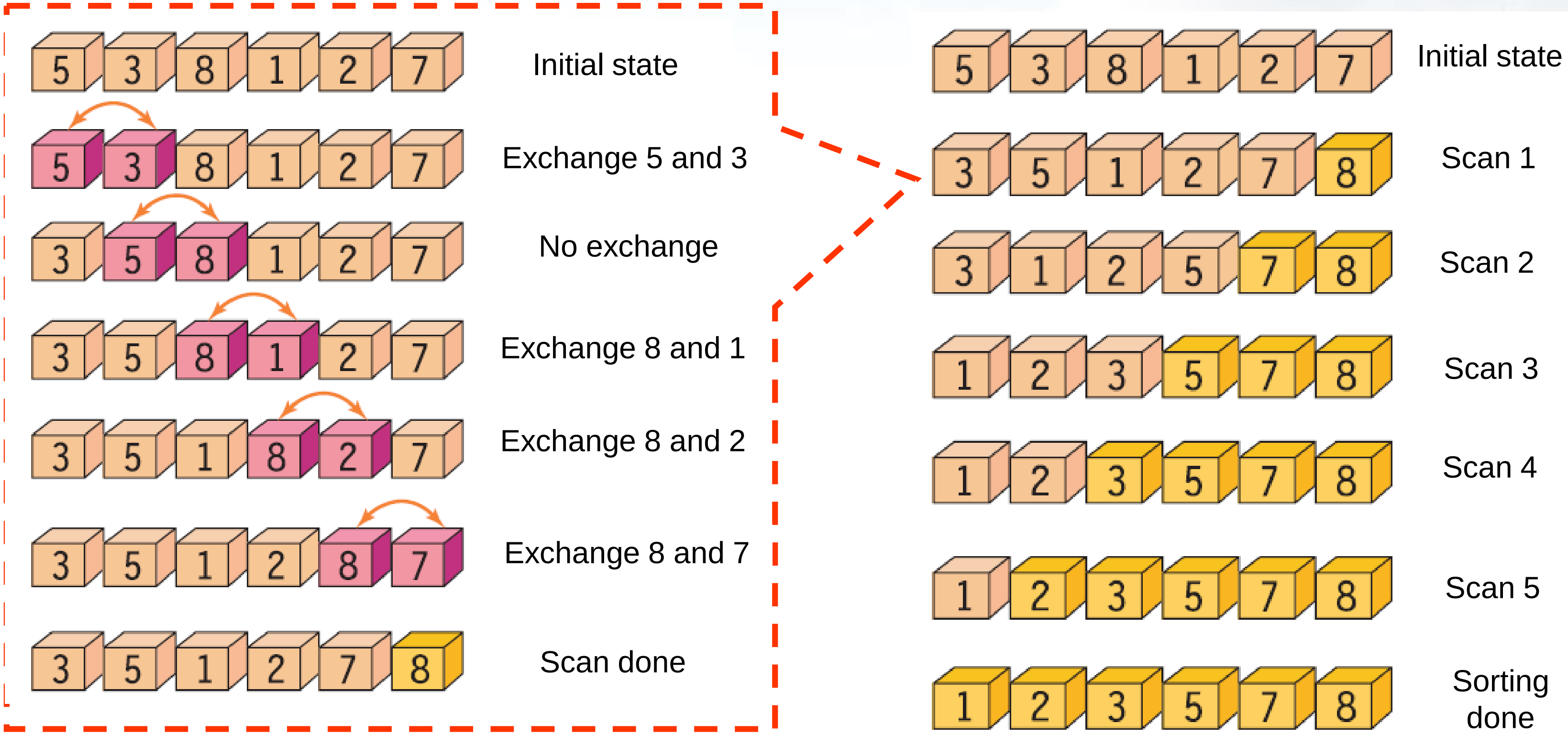
- Time complexity
  - Best case: O(n)
    - The case where already sorted
    - n-1 comparisons
  - Worst case: $O(n^2)$
    - The case where reversely sorted
    - For each step, all the data in front should move
  - Average case: $O(n^2)$

- Characteristics
  - Require many moves
    - If size of record is large, not efficient
  - Stable
  - If most of data are sorted, very efficient

02. Simple Sorting Algorithms

Bubble Sorting

- Idea
  - Compare 2 adjacent records, and exchange if they are not in order
  - For each ‘scan’, conduct compares-exchanges throughout the list



02. Simple Sorting Algorithms

Bubble Sorting

- Pseudo code

```
BubbleSort(A, n)

for i←n-1 to 1 do
   for j←0 to i-1 do        // for a scan
           if j and j+1 is not in order, exchange
           j++;
   i--;
```

02. Simple Sorting Algorithms

Bubble Sorting

- Time complexity
  - Number of comparisons (best, worst, average are all constant)

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- Number of moves
  - Worst case (reversely ordered): $O(n^2)$
  - Best case (already ordered): 0
  - Average case: $O(n^2)$

- Many moves of records
  - Move operations take much longer time than comparison operations

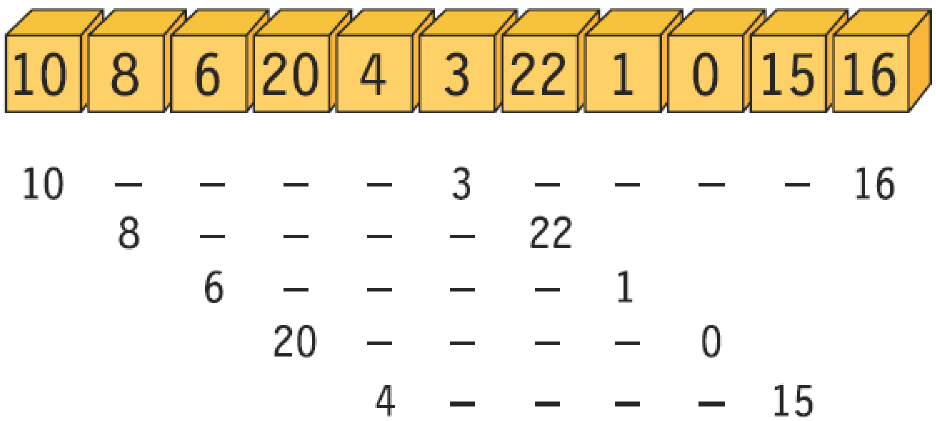
Advanced Sorting Algorithms

Shell Sorting

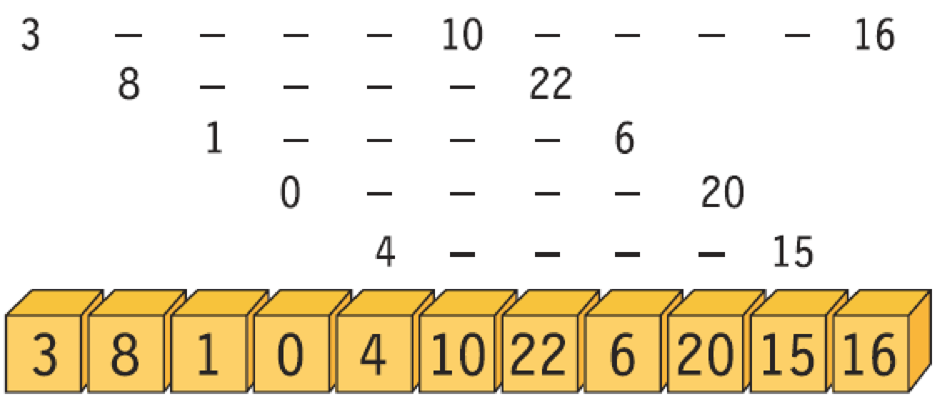
- Idea
  - Insertion sorting is fast in case of the (mostly) sorted list
  - In original insertion sorting, data moves to its neighbor position, hence resulting in many moves
  - By moving data distantly, # moves can be reduced
- Algorithm
  - Divide a list into sub-lists with a specific interval
    - Insertion sorting for each sub-list
  - Reduce the interval
    - # sub-lists decreases, and the size of each sub-list increases
  - Insertion sorting for each sub-list
    - Iterate until the interval becomes 1

Shell Sorting

- Example



(a) sub-lists with interval 5



(b) After sorting each sub-list

Shell Sorting

- Example

Input	10	8	6	20	4	3	22	1	0	15	16
Sub-lists with interval 5	10					3					16
		8					22				
			6					1			
				20					0		
					4					15	
After sorting each sub-list (#5)	3					10					16
		8					22				
			1					6			
				0					20		
					4					15	
	3	8	1	0	4	10	22	6	20	15	16
Sub-lists with interval 3	3			0			22			15	
		8			4			6			16
			1			10			20		
After sorting each sub-list (#3)	0			3			15			22	
		4			6			8			16
			1			10			20		
	0	4	1	3	6	10	15	8	20	22	16
After sorting each sub-list (#1)	0	1	3	4	6	8	10	15	16	20	22

Shell Sorting

- Pros
  - Moving distantly, a data may find a proper position with a small number of moves (compared to original insertion sorting)
  - As sub-lists gradually become sorted, insertion sorting becomes faster

- Time complexity
  - Worst case: O(n^2)
  - Average case: O($n^{1.5}$)

Merge Sorting

- Idea
  - Divide a list into two same-sized sub-lists
  - Sort two sub-lists (in a recursive way)
  - Merge two sub-lists into a sorted final list
- Divide and Conquer (D&C) method
  - A big problem is divided into two smaller problems, and then solve each problem, and combine it so that the original big problem can be solved
  - If a small problem is also difficult to be solved, apply the D&C to the small problem recursively

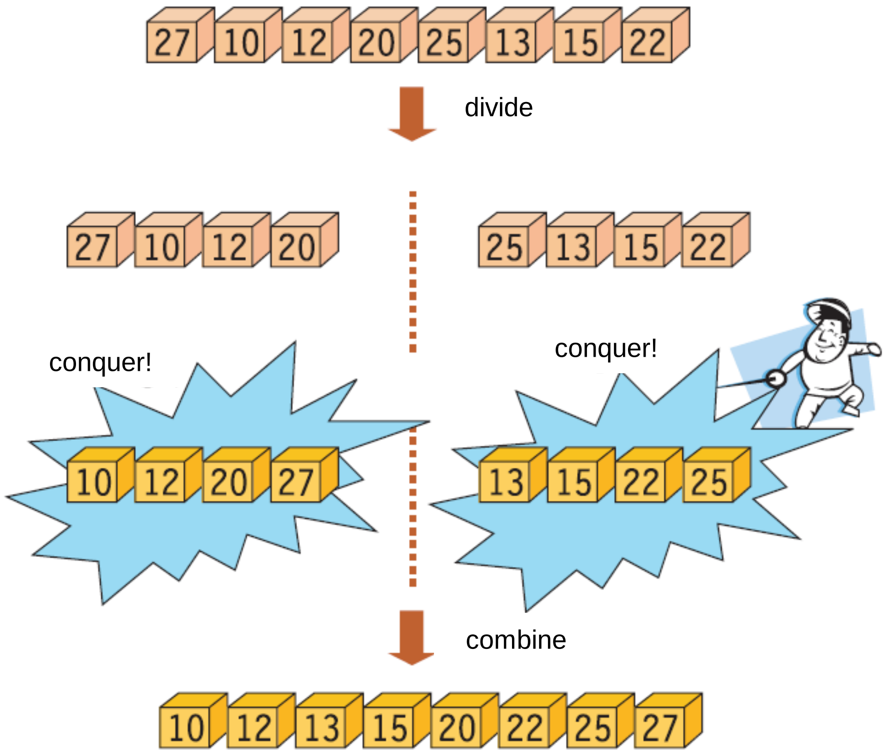
1. Divide: divide a list into two sub-lists
2. Conquer: sort each sub-list. If the size of sub-list is not small-enough (i.e., atomically solvable), apply the D&C to the sub-list recursively
3. Combine: merge the two sorted sub-lists into the final output

03. Advanced Sorting Algorithms

Merge Sorting

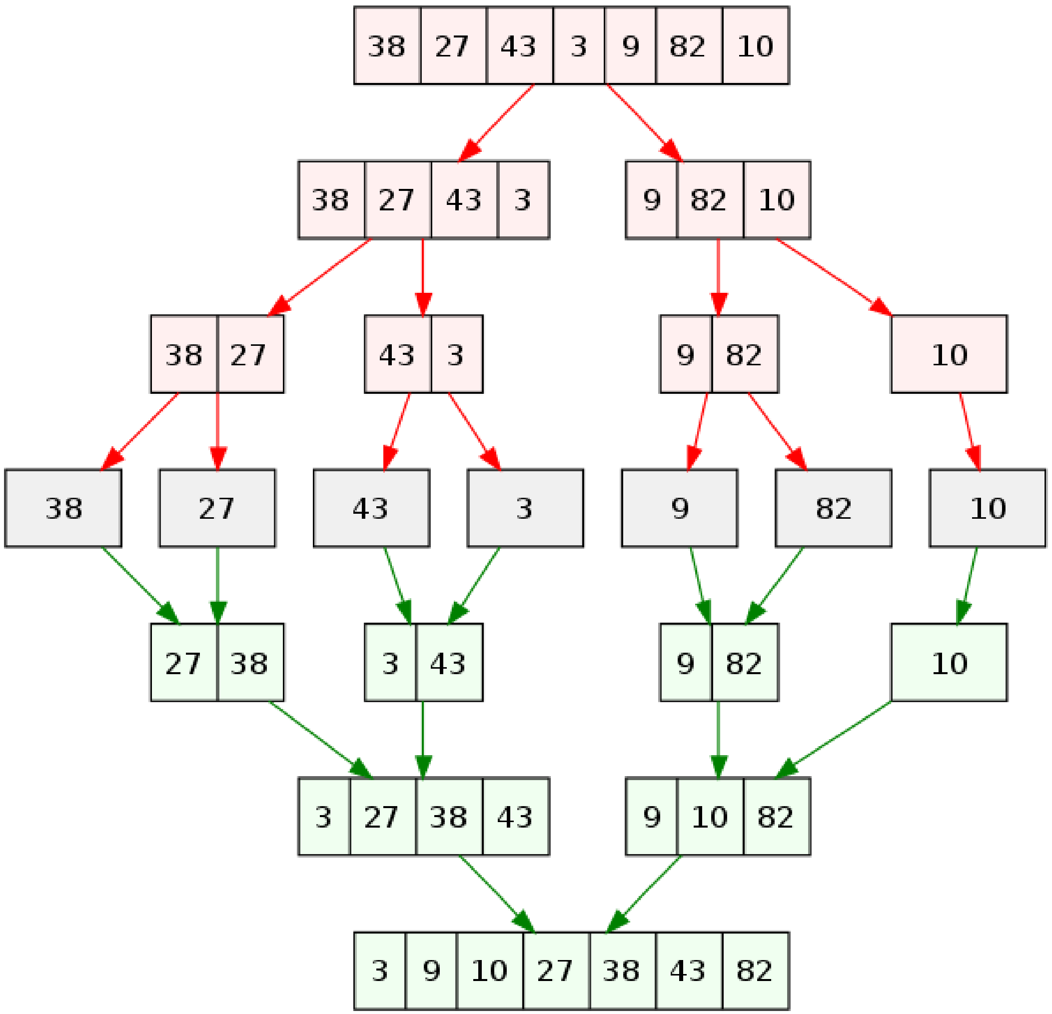
Input: (27 10 12 20 25 13 15 22)

1. Divide: Divide into two lists: (27 10 12 20), (25 13 15 22)
2. Conquer: Sort each sub-list, (10 12 20 27), (13 15 22 25)
3. Combine: Merge two sub-lists into the final output, (10 12 13 15 20 22 25 27)



Merge Sorting

- An illustration



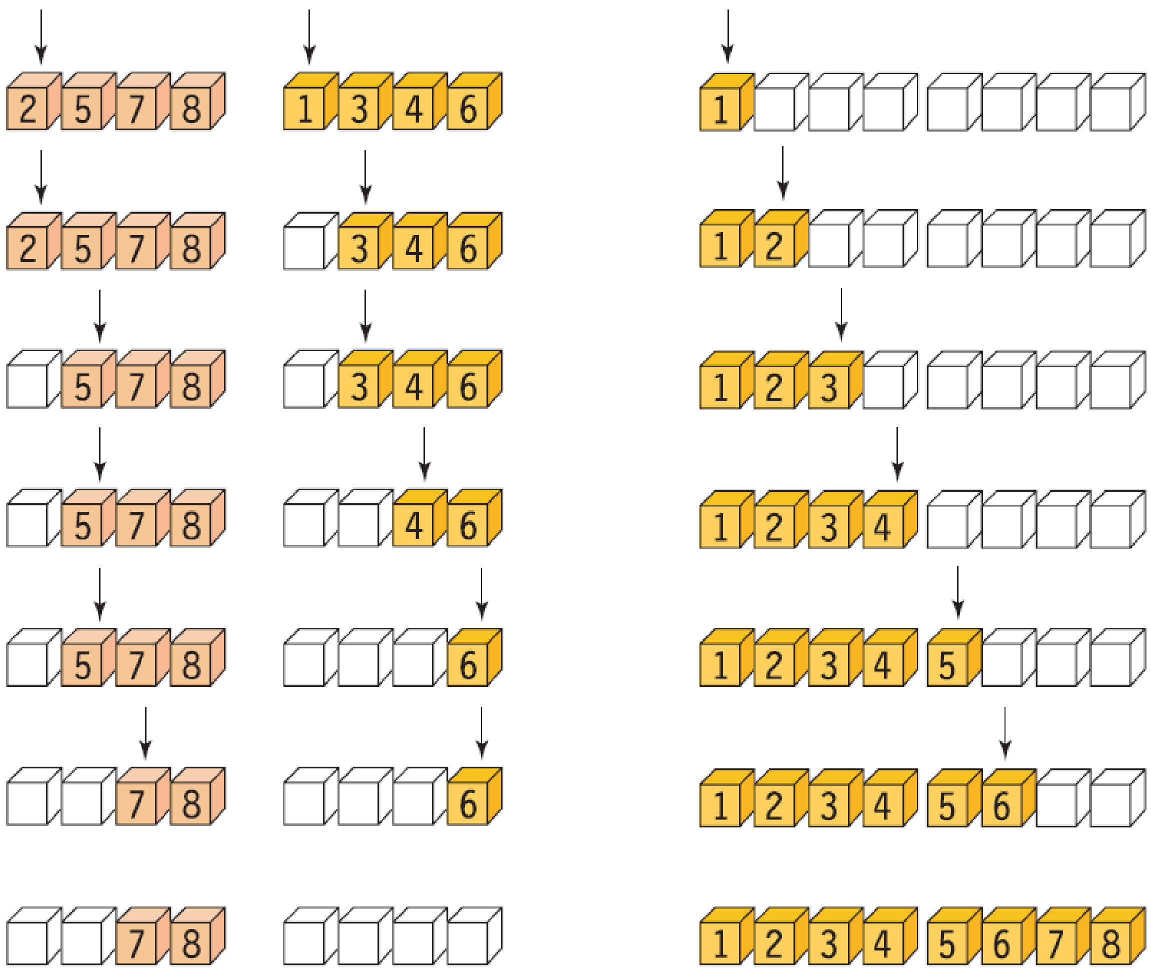
Merge Sorting

- Algorithm

```
merge_sort(list, left, right)

1. if left < right
2.  mid = (left+right)/2;                   // find the mid point
3.  merge_sort(list, left, mid);            // sort the left part
4.  merge_sort(list, mid+1, right);         // sort the right part
5.  merge(list, left, mid, right);          // merge
```

Merge Sorting

- A merging process



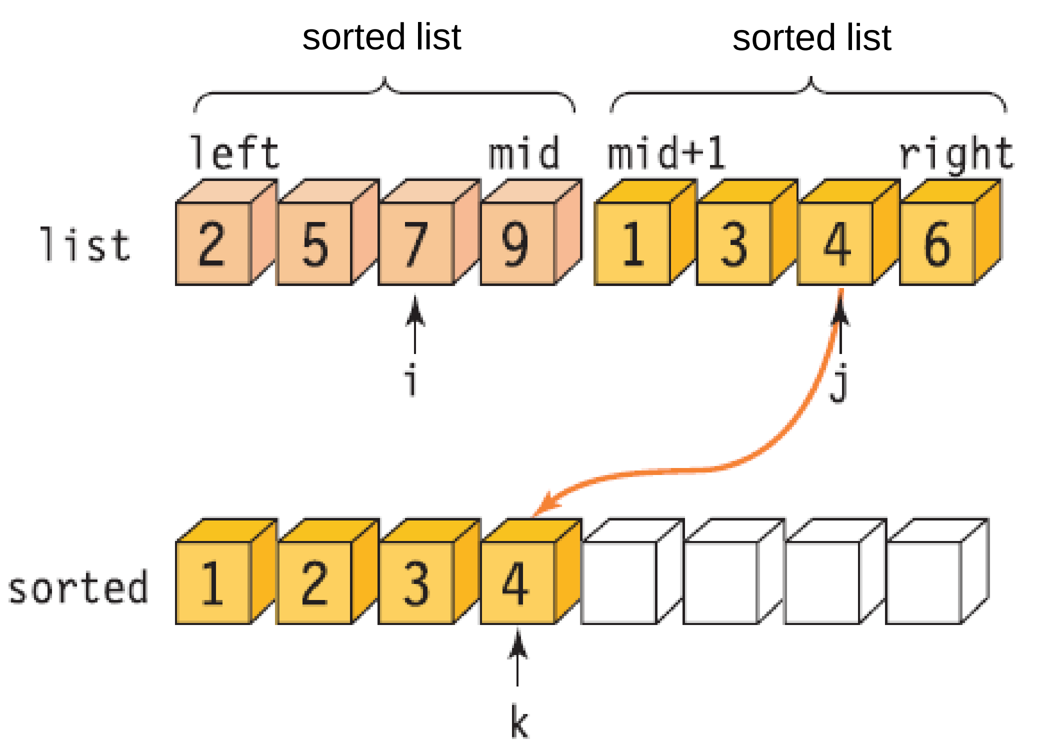
Merge Sorting

- Pseudo code of merge

```
merge(list, left, mid, right)
i←left;
j←mid+1;
k←left;
create a sorted array;
 while i≤left and j≤right do
       if(list[i]<list[j])
                 then
                   sorted[k]←list[i];
                   k++;
                   i++;
                 else
                   sorted[k]←list[j];
                   k++;
                   j++;
copy remaining one into the sorted;
copy sorted to list;
```

Merge Sorting

- An example of a merge



03. Advanced Sorting Algorithms

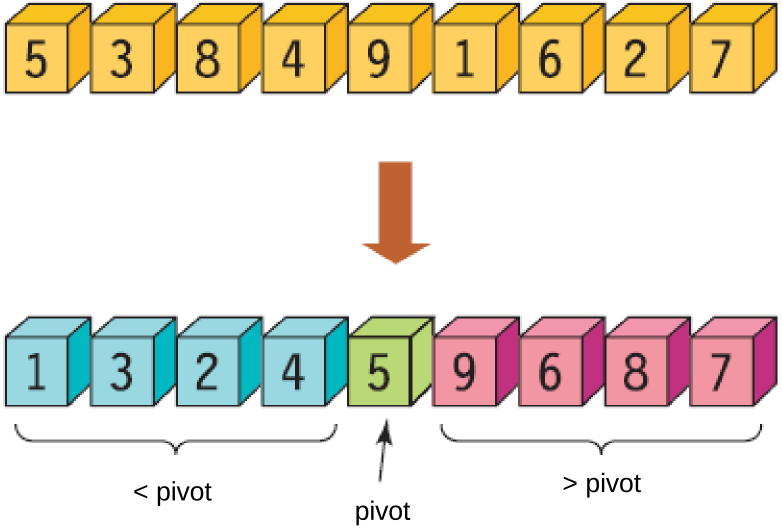
Merge Sorting

- Time complexity
  - Merge sorting uses the recursion
  - Assuming $n = 2^k$, the depth of recursion can be k where $k = \log_2 n$
  - Comparison operations
    - For each pass, n comparisons are needed
    - For k merges, $n * k = n \log_2 n$ comparisons are needed
  - Move operations
    - For each pass, 2n moves are needed
    - For k merges, $2n * k = 2n \log_2 n$ moves are needed
    - If the size of record is large, it takes much time
      - Using a linked list can be a way of reducing # moves
  - Best, worst, and average cases: O(n log n)
- Stable, and less influenced by the initial data distribution

03. Advanced Sorting Algorithms

Quick Sorting

- Known as a fast sorting algorithm on average
- Use the divide and conquer method
- Basic idea
  - Divide a list into two sub-lists based on the pivot value
  - For each sub-list, quick sorting recursively



Quick Sorting

- Algorithm

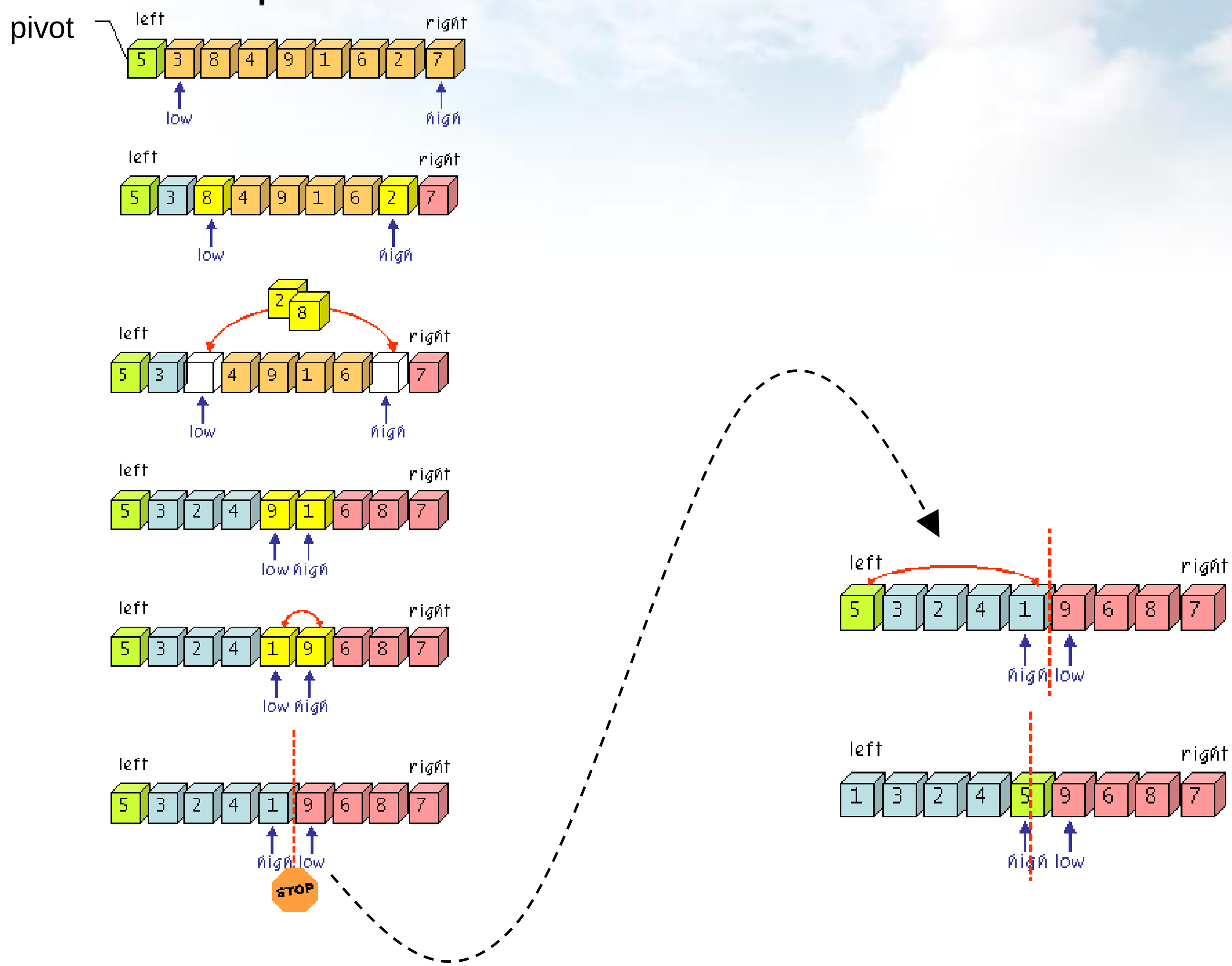
```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);        // Before pi
        quickSort(arr, pi + 1, high);     // After pi
    }
}
```

Quick Sorting

- Algorithm – partition
  - Based on the pivot, divide an in put into two sub-lists
  - Move to left for < pivot, whereas move to right for > pivot
- Algorithm description
  - Pivot: assume the first data
  - From the low index, if it is smaller than pivot, pass to right, otherwise stop
  - From the right index, if it is larger than pivot, pass to left, otherwise stop
  - When low meets high, done

Quick Sorting

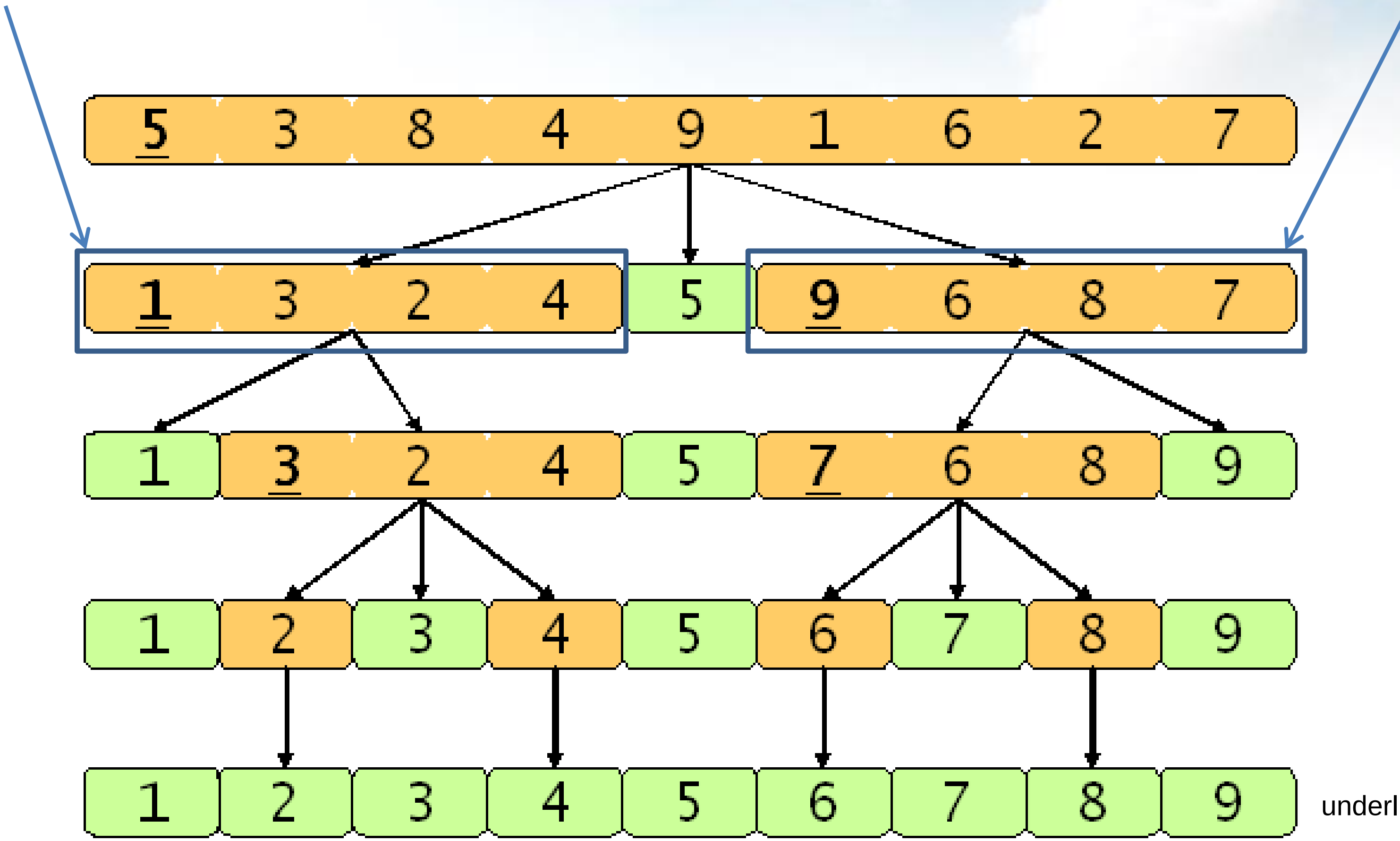
- An illustration – partition



Quick Sorting

- An illustration – quick sorting

Except the pivot, left list (1 3 2 4) and right list (9 6 8 7) are sorted independently, respectively

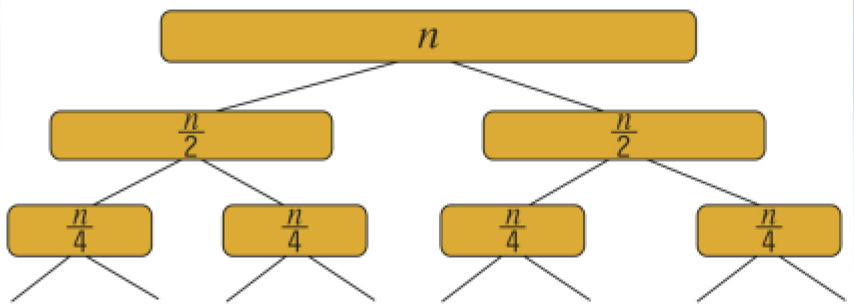


underline: pivot

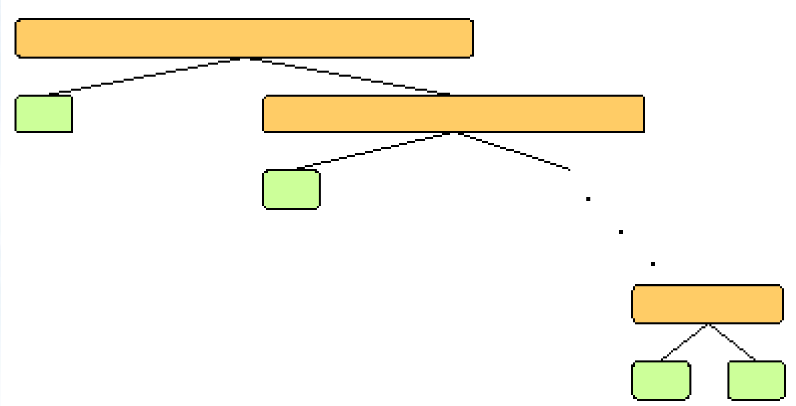
03. Advanced Sorting Algorithms

Quick Sorting

- Complexity
  - Best case (evenly divided)
    - # passes: log n
      - 2->1
      - 4->2
      - 8->3
      - n->log n
    - For each pass, # comparisons: n
    - Total comparisons: n * log n



Quick Sorting



- Complexity
  - Worst case (skewedly divided)
    - # passes: n
    - For each pass, # comparisons: n
    - Total comparisons: n^2
    - Example

```
(1 2  3  4  5  6  7  8  9)
1 (2  3  4  5  6  7  8  9)
1  2 (3  4  5  6  7  8  9)
1  2  3 (4  5  6  7  8  9)
1  2  3  4 (5  6  7  8  9)
                ...
1  2  3  4  5  6  7  8  9
```

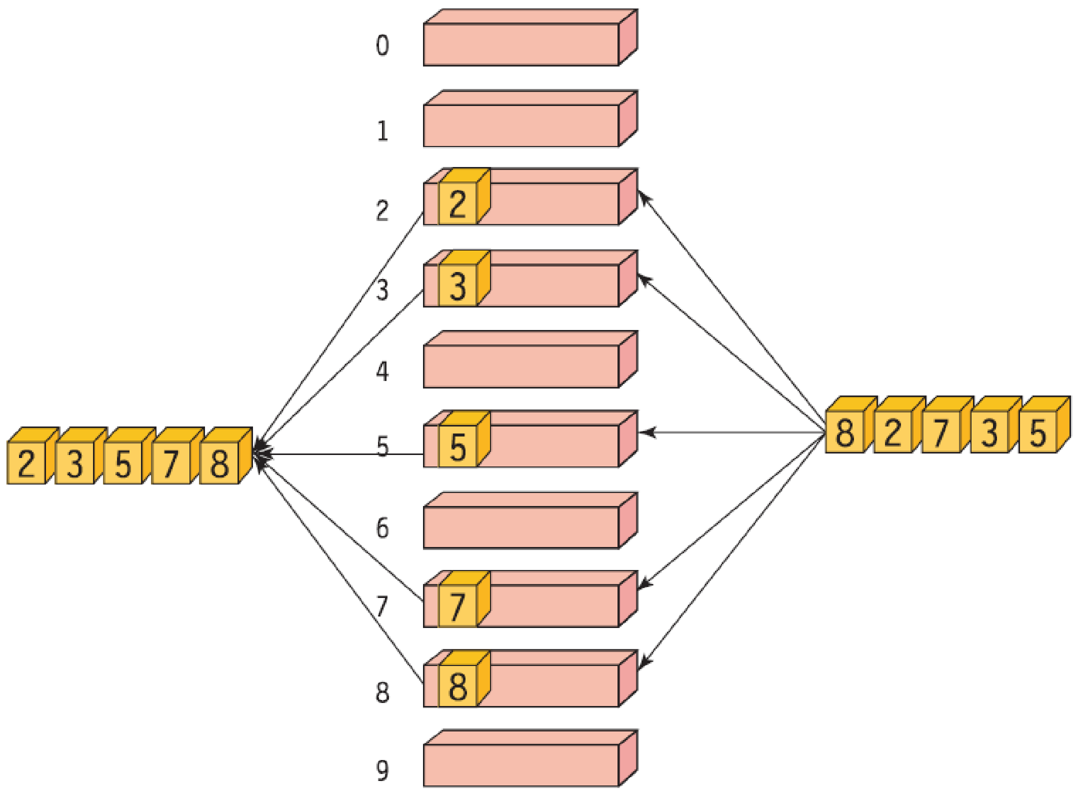
  - Choosing a good pivot, e.g., medium value, would reduce the imbalance partitioning

Radix Sorting

- Idea
  - Most of sorting algorithms ‘compare’ data (or records)
  - How about sorting without comparison?
- Radix sort
  - May perform better than sorting algorithms with comparisons (lower bound: O (n log n))
  - Complexity: O(dn) where d < 10
  - Cons
    - Limited type of data to be sorted
      - Floating numbers, Korean, Chinese letters may not be applicable
      - Key with same length such as number or simple letter like alphabet can be applicable
    - Require additional memory

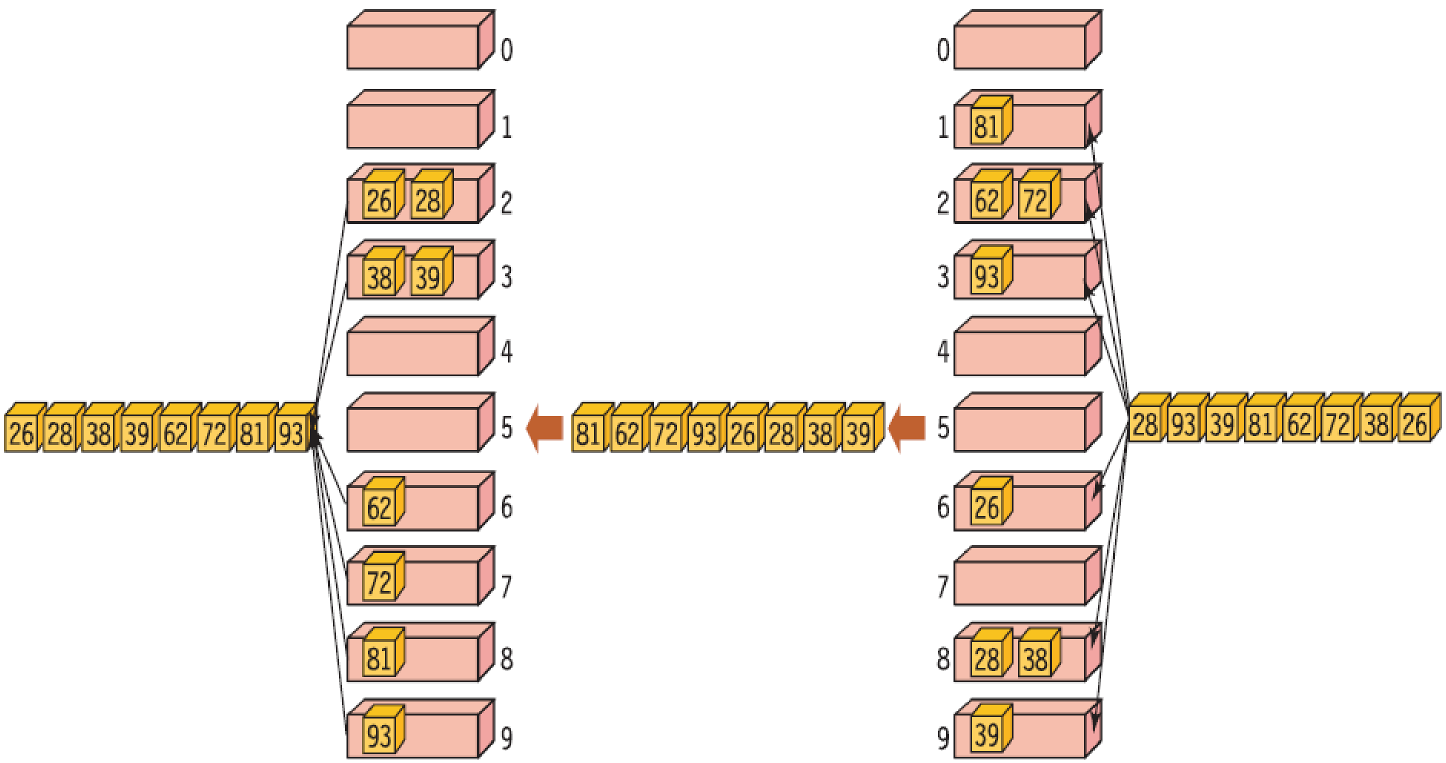
Radix Sorting

- Example: (8, 2, 7, 3, 5) by radix sorting
  - Uses 10 buckets for single digit data



Radix Sorting

- Example: (28, 93, 39, 81, 62, 72, 38, 26) by radix sorting
  - Uses 10 buckets for double digits data
  - First sort for the low digit, and then sort for the high digit



Radix Sorting

- Algorithm

```
RadixSort(list, n):

for d←LSD to MSD do
{
   for the dth digit, enqueue from 0 to 9 buckets
   read from each bucket to generate a list
  d++;
}
```

Radix Sorting

- Design consideration
  - Each bucket is implemented as a queue
  - # buckets links to the representation of key
    - Binary notation -> 2 buckets
    - Alphabet -> 26 buckets
    - Decimal notation -> 2 buckets
  - A trade-off between # bucket vs. # passes

03. Advanced Sorting Algorithms

Radix Sorting

- Time complexity
  - n data, d digits key
  - d * n enqueues
  - O(d n)
    - As d is mostly smaller than 10, sorting is done quickly

What You Need to Know

Summary

Algorithm	Best	Average	Worst
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
Shell	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
Quick	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Heap	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Merge	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Radix	$O(dn)$	$O(dn)$	$O(dn)$

What You Need to Know

Summary

Algorithm	Execution time (sec) with 60,000 data
Insertion	7.438
Selection	10.842
Bubble	22.894
Shell	0.056
Quick	0.014
Heap	0.034
Merge	0.026

Thanks

Week 10: Sorting Algorithms
Instructor: Jinyoung Han (jinyounghan@skku.edu)