Data Structures

Lecture 9: Sort

Dongbo Min
Department of Computer Science and Engineering
Ewha Womans University, Korea
E-mail: dbmin@ewha.ac.kr

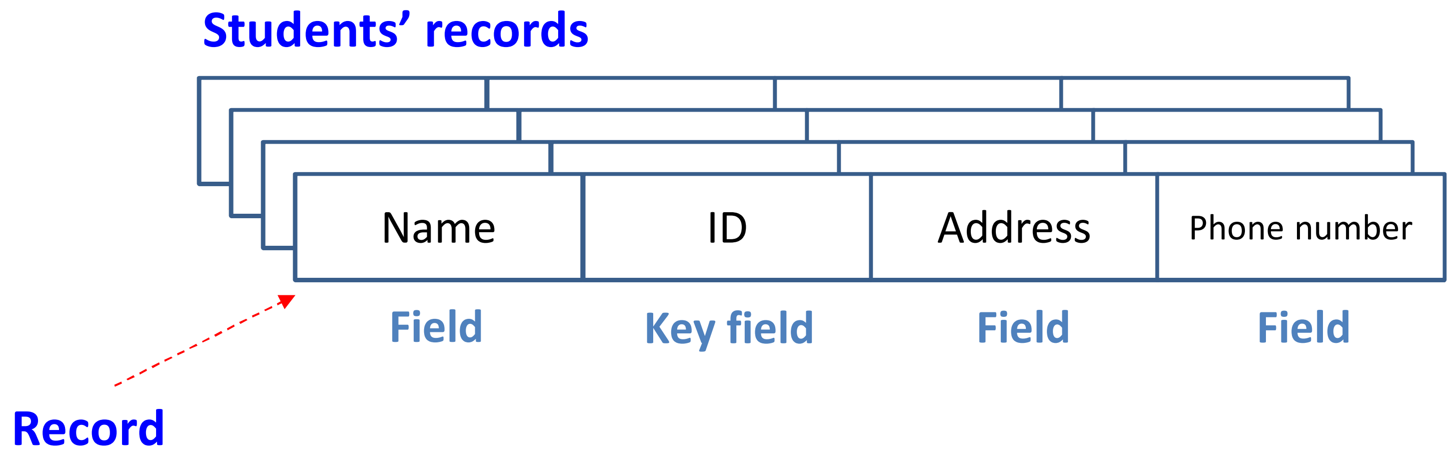
Sorting

- Sorting
  lists a set of data in ascending or descending order

- Sorting is essential when searching data
  Ex) What if words in the English dictionary are not sorted alphabetically?

Target of Sorting

- Record
  - Data to be sorted
  - Consists of multiple fields
  - Key field is used to identify records



Sorting Algorithm

- No golden solution that works for all cases perfectly!

- Soring algorithm must be chosen by considering the following cases
  - The number of records (data)
  - Record size
  - Key characteristics (letter, integer, floating number)
  - Internal / external memory sorting

- Evaluation criteria of sorting algorithm
  - The number of comparisons
  - The number of moves of data

Summary of Sorting Algorithms

	Best	Average	Worst	Stability
Insert sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes
Shell sort	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$	No
Quick sort	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	No
Heap sort	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	No
Merge sort	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	Yes
Count sort	$O(n)$	$O(n)$	$O(n)$	Yes
Radix sort	$O(dn)$	$O(dn)$	$O(dn)$	Yes

Sorting Algorithm

- Simple but inefficient algorithms
  - Insert sort, selection sort, bubble sort

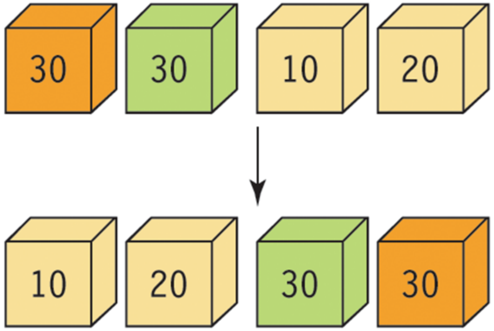
- Complex but efficient algorithms
  - Quick sort, heap sort, merge sort, radix sort

- Internal sort
  - Sort a set of data that is stored in main memory

- External sort
  - Sort a set of data, when most of the data is stored in the external storage device and only a part of the data is stored in the main memory

Sorting Algorithm

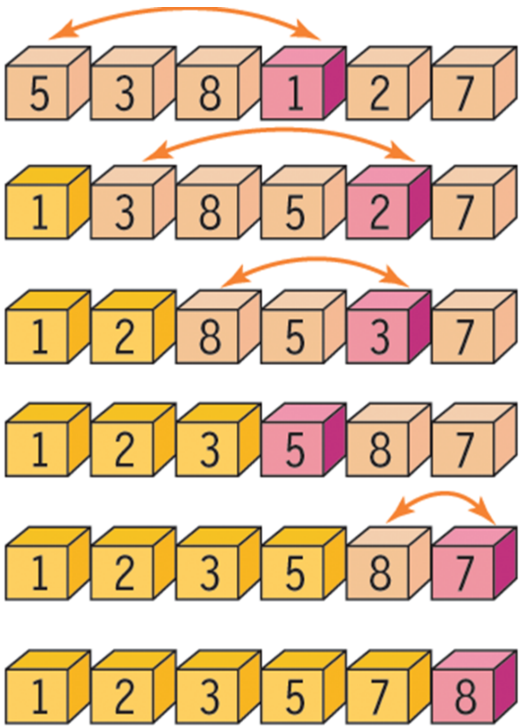
- *Stability* of sorting algorithm
  - The relative position of records with the same key value does not change after sorting
  - Example of unstable sort



Selection Sort

- An input array: left and right lists
  - Left list: sorted data
  - Right list: unordered data
  - Initially, the left list is empty, and all the numbers to sort are in the right list
- Procedure
  1. Select the minimum value in the right list and exchange it with the first number in the right list.
  2. Increase the left list size
  3. Decrease the right list size

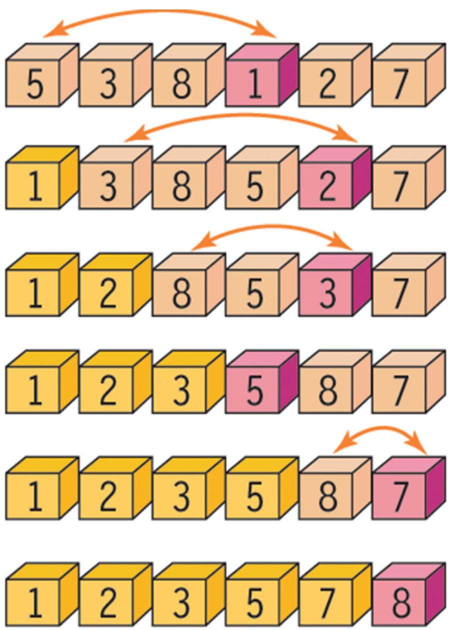
  Iterate 1-3 until the right list is empty



Selection Sort

```
selection_sort(A, n)

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

```c
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void selection_sort(int list[], int n)
{
        int i, j, least, temp;
        for (i = 0; i < n-1; i++) {
                least = i;
                for (j = i + 1; j<n; j++)
                        if (list[j]<list[least]) least = j;
                SWAP(list[i], list[least], temp);
        }
}
```



Selection Sort

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void selection_sort(int list[], int n)
{
         int i, j, least, temp;
         for (i = 0; i < n-1; i++) {
                  least = i;
                  for (j = i + 1; j<n; j++)
                           if (list[j]<list[least]) least = j;
                  SWAP(list[i], list[least], temp);
         }
}
```

# of comparisons
(n - 1) + (n - 2) + … + 1 = n(n - 1)/2 = $O(n^2)$

# of moves: 3(n - 1)

Time complexity: $O(n^2)$

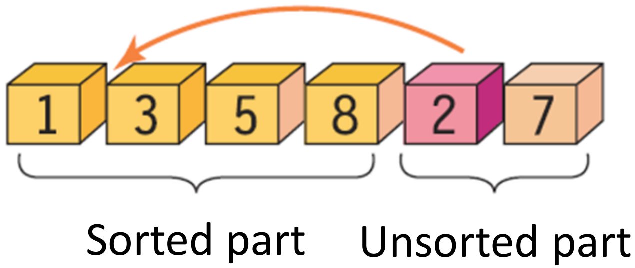
The selection sort is not stable. Ex) Sort [2 5 3 2 1]

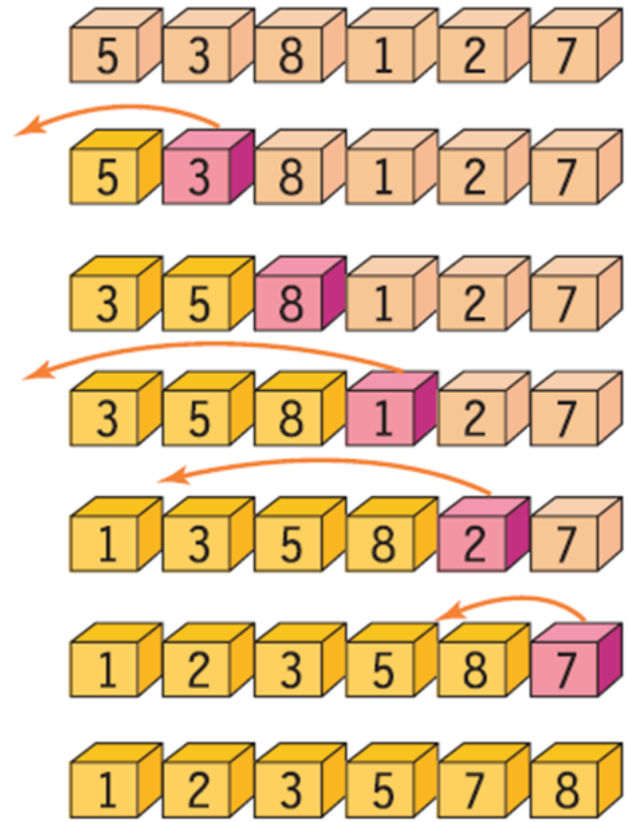
Selection Sort

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10000
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
int list[MAX_SIZE];
int n;
//
void selection_sort(int list[], int n) {
         //...
}

void main()
{
         int i;
         n = MAX_SIZE;
         // Generate input data using random numbers (0~n)
         for (i = 0; i<n; i++)
                  list[i] = rand() % n;

         selection_sort(list, n);

         for (i = 0; i<n; i++)       printf("%d\n", list[i]);
}
```

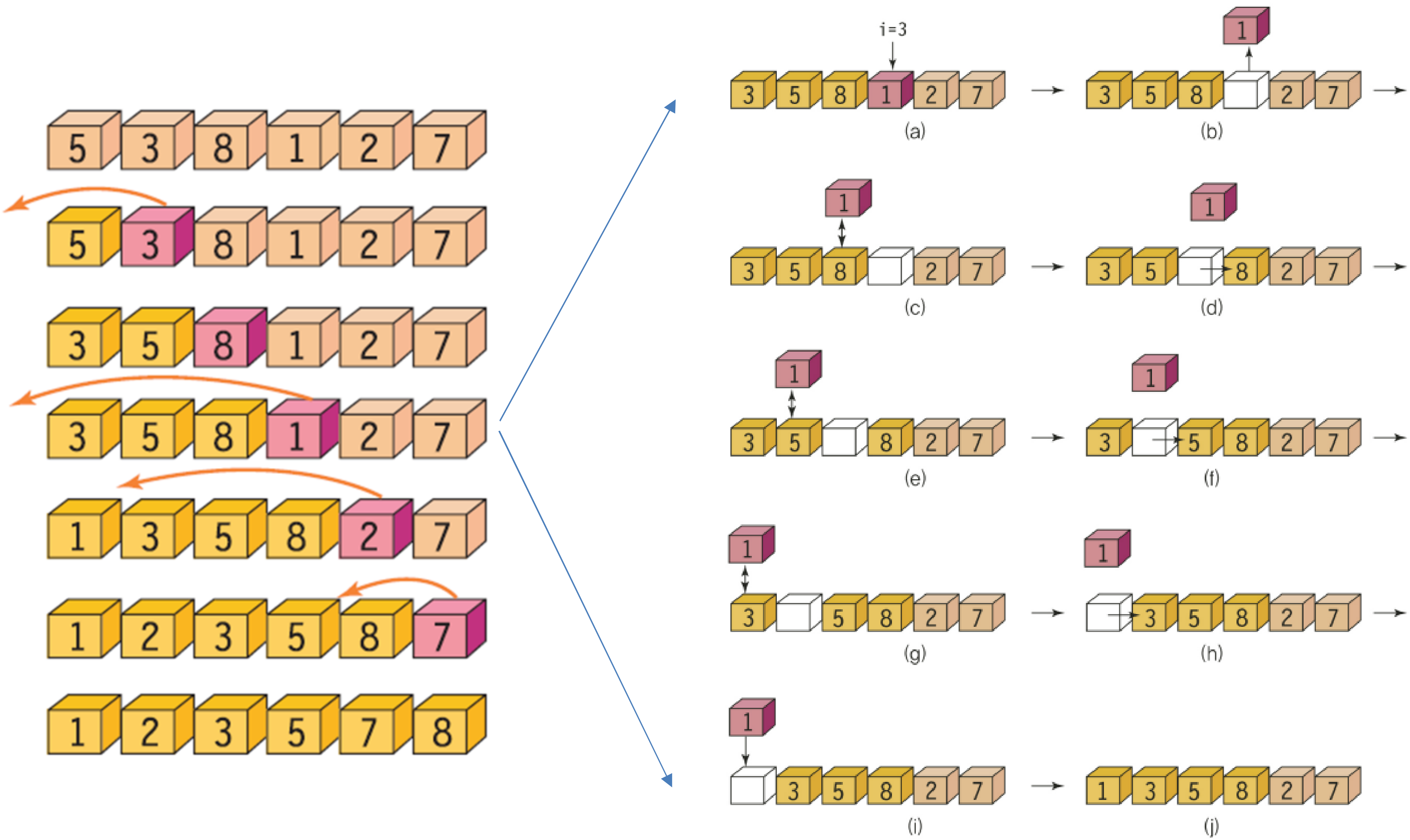
Insertion Sort

- Iteratively insert a new record in the right place of the sorted part

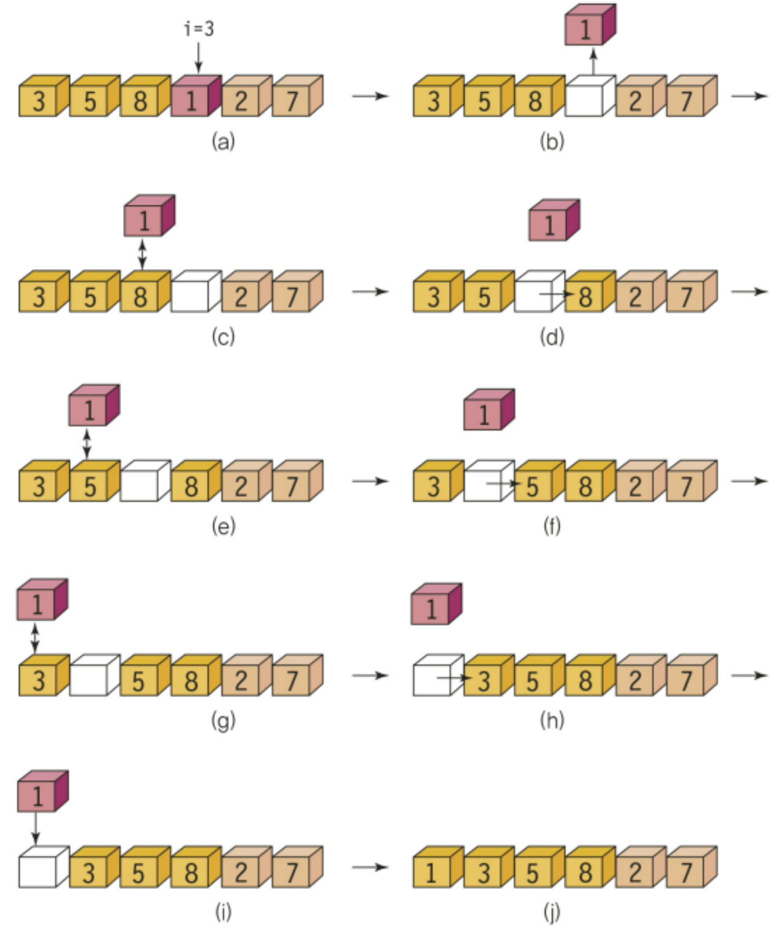




Insertion Sort



```
insertion_sort(A, n)

1. for i ← 1 to n-1 do
2.       key ← A[i];
3.       j ← i-1;
4.       while j≥0 and A[j]>key do
5.             A[j+1] ← A[j];
6.             j ← j-1;
7.       A[j+1] ← key
```



1. Starting from i=1
2. Copy the i-th integer, which is the current number to insert, into the key variable.
3. Start at (i-1)-th position, since the currently sorted array is up to i-1
4. If the value in the sorted array is greater than the key value,
5. Move the j-th data to the (j+1)-th data
6. Decrease j
7. Because the j-th integer is less than key, copy the key into the (j+1) position

Insertion Sort

```
void insertion_sort(int list[], int n)
{
        int i, j, key;
        for (i = 1; i<n; i++) {
                key = list[i];
                for (j = i - 1; j >= 0 && list[j]>key; j--)
                        list[j + 1] = list[j];
                list[j + 1] = key;
        }
}
```

Time Complexity of Insert Sort

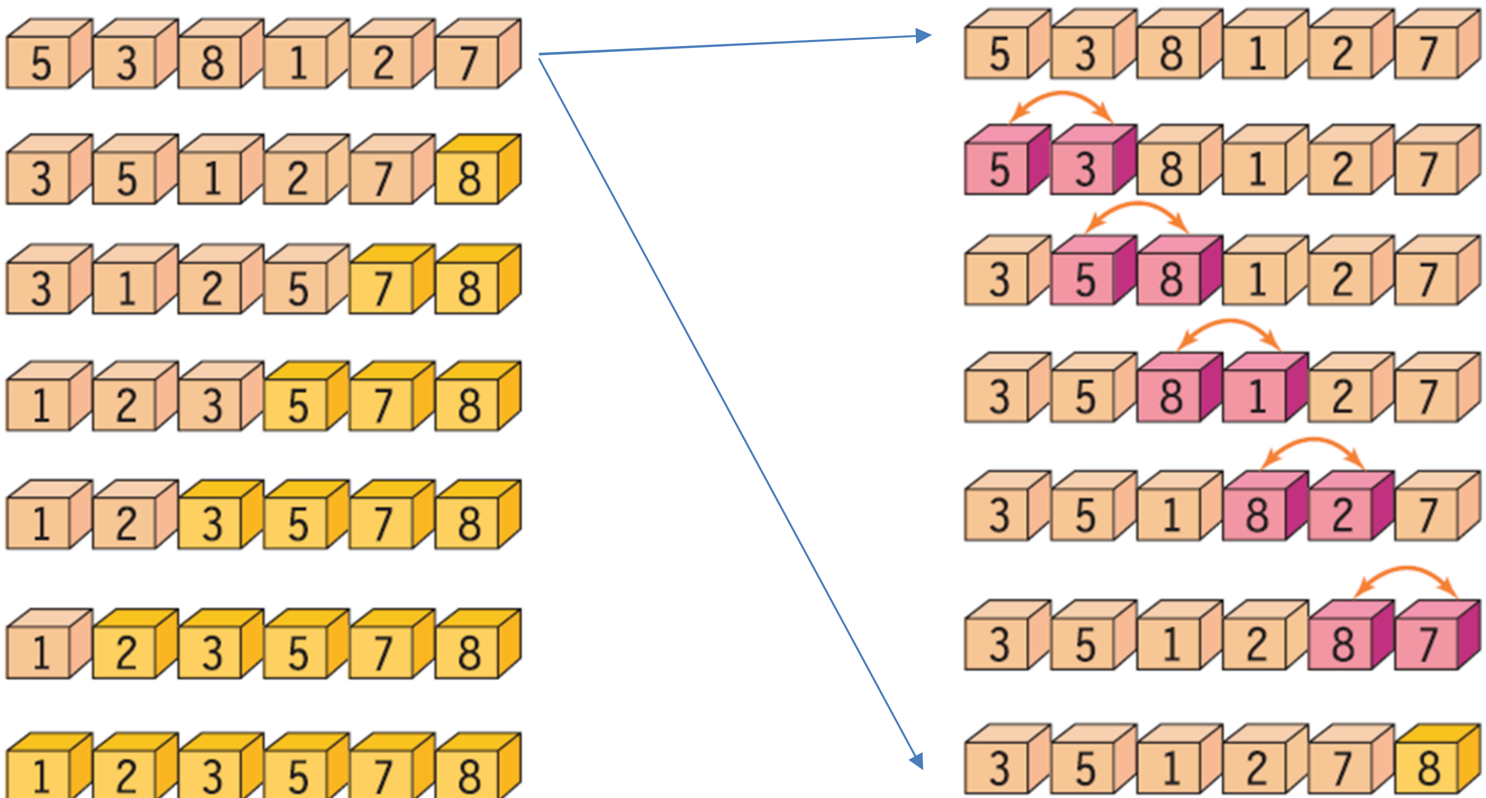
- Best case: when the data is already sorted
  - # of comparisons: n-1
  - # of moves: 0

- Worst case: when sorted in reverse order
  - # of comparisons: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - # of moves: $\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$

- Property
  - When the data size is large, the time complexity increases significantly.
  - Stable sort
  - It is very efficient when the data is mostly sorted.

Bubble Sort

- Procedure
  - Swap two adjacent data, when they are not in order
  - This comparison-exchange process is repeated from the left to the right.



Bubble Sort

```
BubbleSort(A, n)

for i←n-1 to 1 do
        for j←0 to i-1 do
                Swap if when j and j+1 elements are not in order
                j++;
        i--;
```

```c
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void bubble_sort(int list[], int n)
{
        int i, j, temp;
        for (i = n - 1; i>0; i--) {
                for (j = 0; j<i; j++)
                        if (list[j]>list[j + 1])
                                SWAP(list[j], list[j + 1], temp);
        }
}
```

Time Complexity of Bubble Sort

- Best case: when the data is already sorted
  - Comparison: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
  - Movement: 0
- Worst case: when sorted in reverse order
  - Comparison: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
  - Move: 3*comparison count = $\frac{3n(n-1)}{2}$
- It is stable sort.

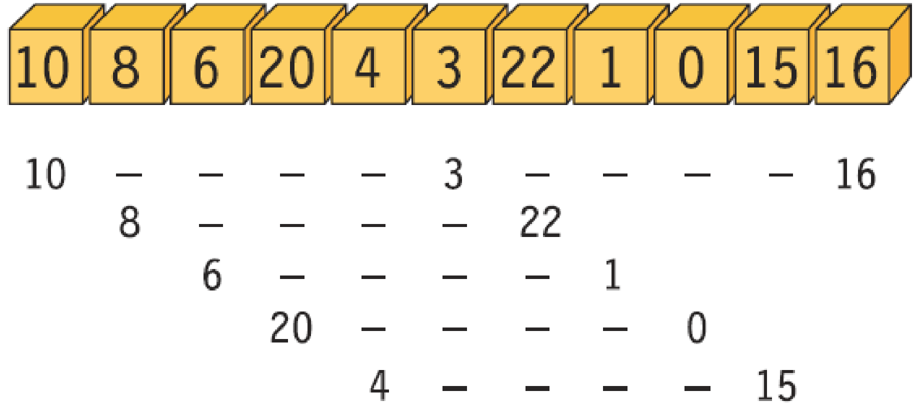
Shell Sort

- Motivation
  – Insert sort is very efficient when the data is mostly sorted.
  – The insertion sort moves elements only to neighboring positions
  – By allowing elements to move to a remote location, they can be sorted with a smaller amount of moves.

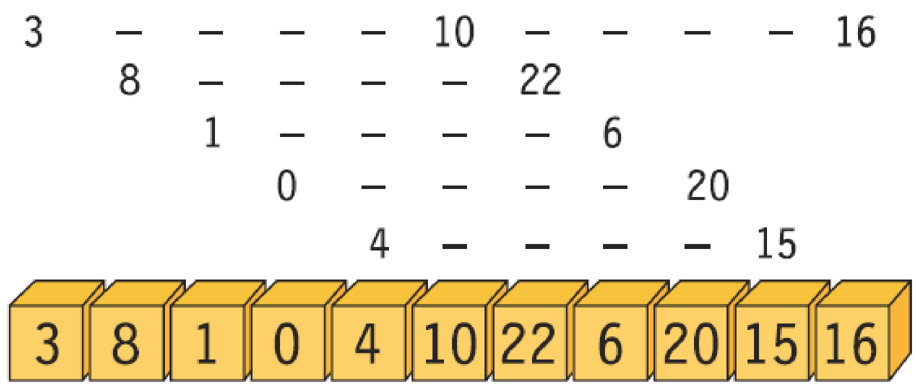
- Procedure
  1. Divide the array into a set of sub-arrays with an interval.
  2. Apply the insert sort to each sub-array
  3. Decrease the interval

  Iterate 1-3 until the interval becomes 1

Shell Sort



(a) A set of sub-arrays with an interval of 5



(b) Each sub-array is sorted using the insert sort

Shell Sort

Input array	10	8	6	20	4	3	22	1	0	15	16
A set of sub-arrays when interval=5	10					3					16
		8					22				
			6					1			
				20					0		
					4					15	
After the insert sort	3					10					16
		8					22				
			1					6			
				0					20		
					4					15	
	3	8	1	0	4	10	22	6	20	15	16
A set of sub-arrays when interval=3	3			0			22			15	
		8			4			6			16
			1			10			20		
After the insert sort	0			3			15			22	
		4			6			8			16
			1			10			20		
	0	4	1	3	6	10	15	8	20	22	16
After the insert sort when interval = 1	0	1	3	4	6	8	10	15	16	20	22

Shell Sort

```c
// Insert and sort elements apart by interval
// The range of sort is first to last
inc_insertion_sort(int list[], int first, int last, int gap)
{
        int i, j, key;
        for (i = first + gap; i <= last; i = i + gap) {
                key = list[i];
                for (j = i - gap; j >= first && key<list[j]; j = j - gap)
                        list[j + gap] = list[j];
                list[j + gap] = key;
        }
}
//
void shell_sort(int list[], int n)   // n = size
{
        int i, gap;
        for (gap = n / 2; gap>0; gap = gap / 2) {
                if ((gap % 2) == 0) gap++;
                        for (i = 0; i<gap; i++)
                                inc_insertion_sort(list, i, n - 1, gap);
        }
}
```

Shell Sort

- Advantages of shell short
  - Increases the likelihood of completing the sort with less amount of moves by moving remote data in a sub-array
  - Since the sub-arrays become progressively sorted, the insertion sort becomes faster when the interval decreases.

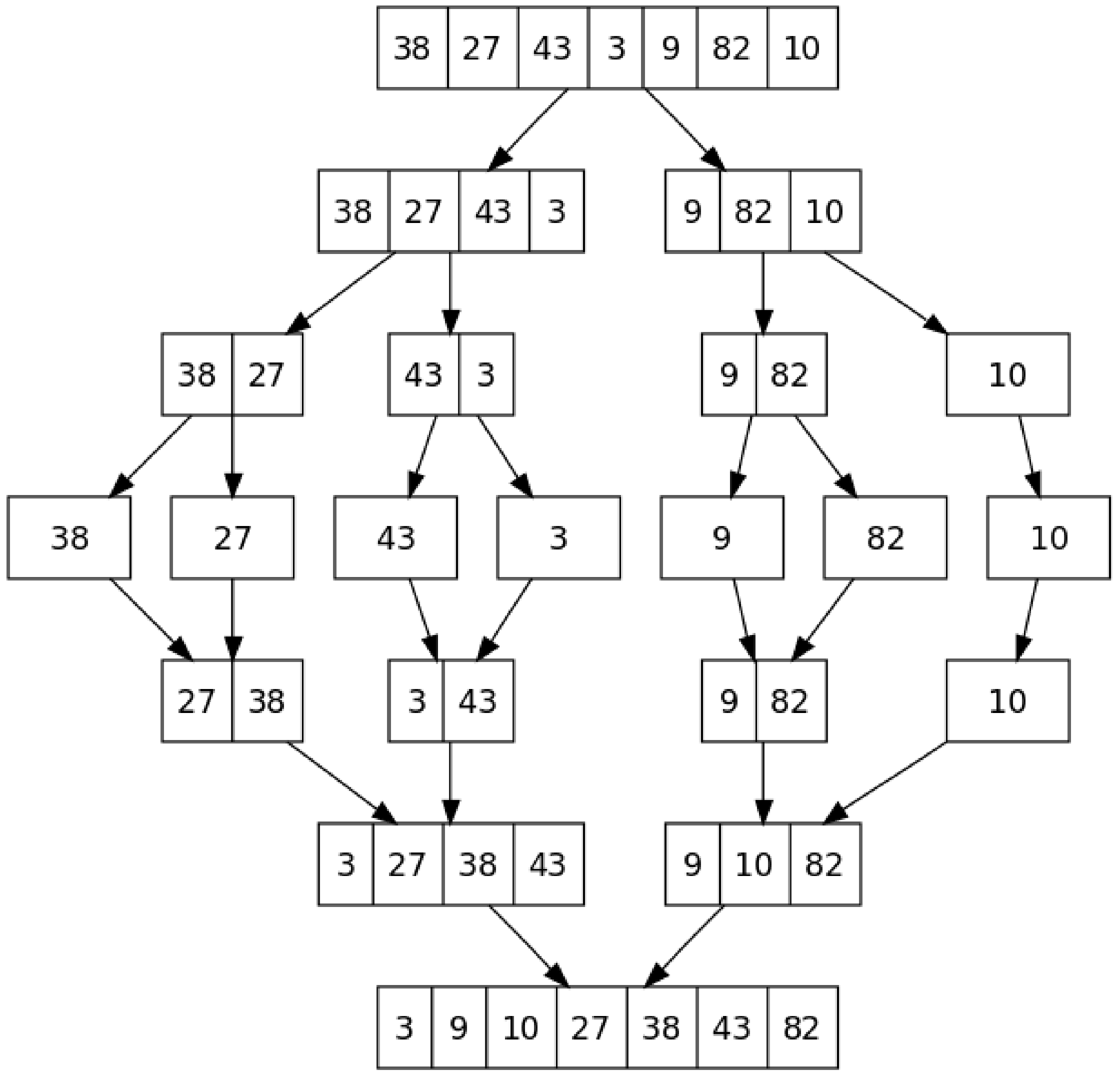
- Time complexity
  - Worst case: $O(n^2)$
  - Average case: $O(n^{1.5})$

Merge Sort

- Using the divide and conquer method
  1. Divide the array into two *equal* sizes and sort the split arrays
  2. Sort the entire array by summing the two sorted arrays.

```
Input: (27 10 12 20 25 13 15 22)
1. Divide: divide it into (27 10 12 20) and (25 13 15 22)
2. Conquer: Sort the sub-arrays - (10 12 20 27) and (13 15 22 25)
3. Combine: Merge the sorted sub-arrays:
            (10 12 13 15 20 22 25 27)
```

Merge Sort



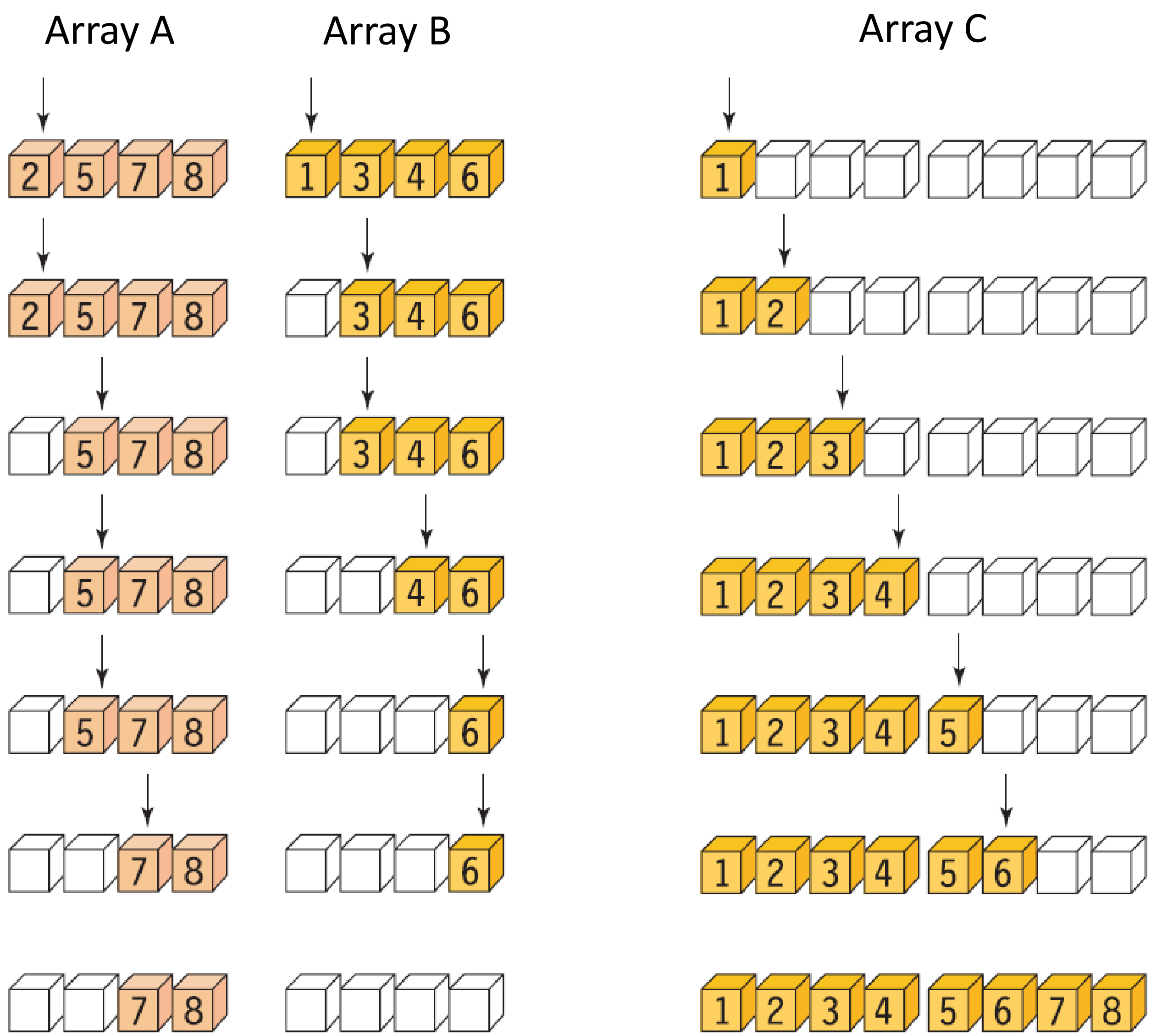
38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort

```
merge_sort(list, left, right)
1       if left < right
2               mid = (left+right)/2;
3               merge_sort(list, left, mid);
4               merge_sort(list, mid+1, right);
5               merge(list, left, mid, right);
```

1. If the size of the segment is greater than 1
2. Calculate intermediate position
3. Sort the left-side array (recursive call)
4. Sort the right sub-array (recursive call)
5. Merge the two sorted subarrays

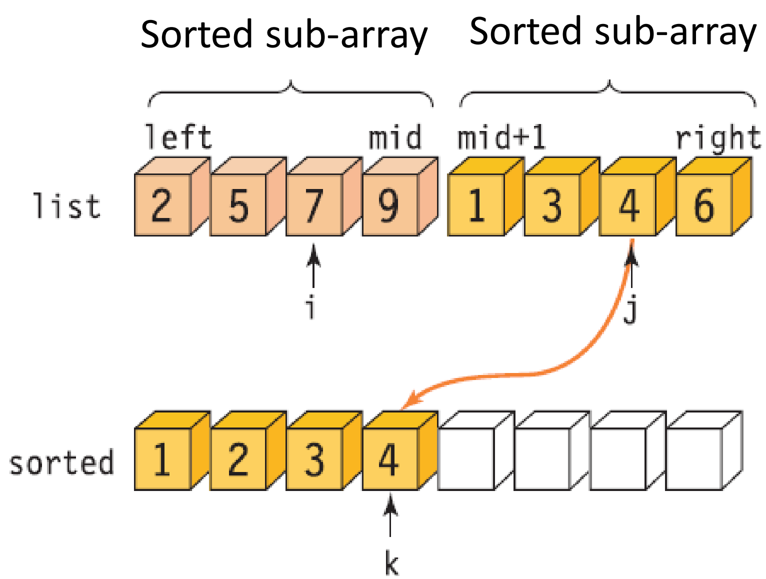
Merge Sort



Merge function

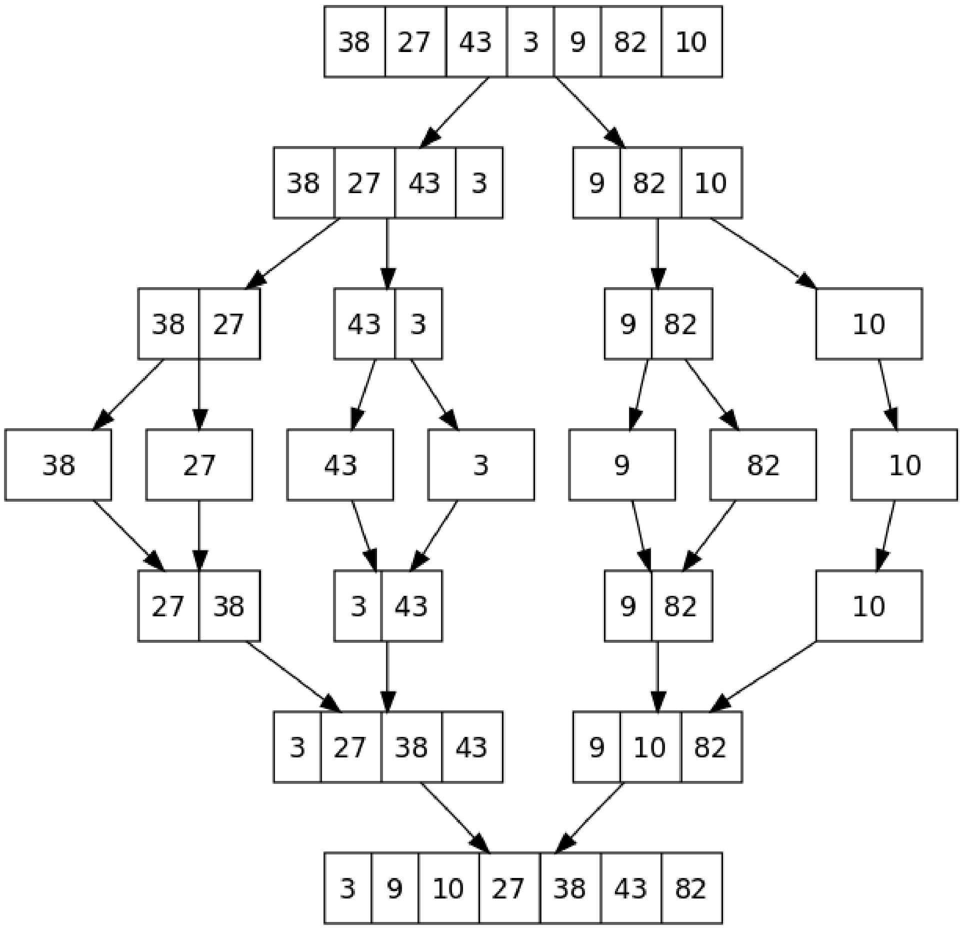
Merge Sort

```
merge(list, left, mid, right) :
// merge two adjacent arrays list [left..mid] and list [mid + 1..right]

i ← left;
j ← mid + 1;
k ← left;
while i ≤left and j ≤right do
         if (list[i] <= list[j])
                  sorted[k] ← list[i];
                  k++;
                  i++;
         else
                  sorted[k] ← list[j];
                  k++;
                  j++;

Copies the remaining elements to ‘sorted’;
copy ‘sorted’ to list;
```



```c
int sorted[MAX_SIZE];
// i, j: the index to the sorted left and right arrays
// k is the index for the list to be sorted
void merge(int list[], int left, int mid, int right)
{
        int i, j, k, l;
        i = left; j = mid + 1; k = left;
        // Merge split-sorted arrays
        while (i <= mid && j <= right) {
                if (list[i] <= list[j]) sorted[k++] = list[i++];
                else sorted[k++] = list[j++];
        }
        if (i> mid) // Copy remaining data
                for (l = j; l<=right; l++)
                        sorted[k++] = list[l];
        else // Copy remaining data
                for (l = i; l<=mid; l++)
                        sorted[k++] = list[l];
        // Copy list of array sorted [] to array list []
        for (l = left; l<=right; l++)
                list[l] = sorted[l];
}
void merge_sort(int list[], int left, int right){
        int mid;
        if (left<right) {
                mid = (left + right) / 2;
        merge_sort(list, left, mid);
                merge_sort(list, mid + 1, right);
                merge(list, left, mid, right);
        }
}
```

Time Complexity of Merge Sort



Totally, $log_2 n$ recursion

For each recursion,
\# of comparisons = n
\# of moves = $2n$

For best and worst case,
$O(nlog_2 n)$ and $\Omega(nlog_2 n)$

Note)
- If the input is implemented using the linked list, the move becomes a simple address update.
- It is a stable sort and is less influenced by the initial order of data

Time Complexity of Merge Sort

```
merge_sort(list, left, right)
      if left < right
            mid = (left+right)/2;
            merge_sort(list, left, mid);
            merge_sort(list, mid+1, right);
            merge(list, left, mid, right);
```

$$T(1) = c$$

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \quad n \geq 2$$

$$\Rightarrow \quad \Theta(nlog_2 n)$$

Note)

$$T(1) = c$$

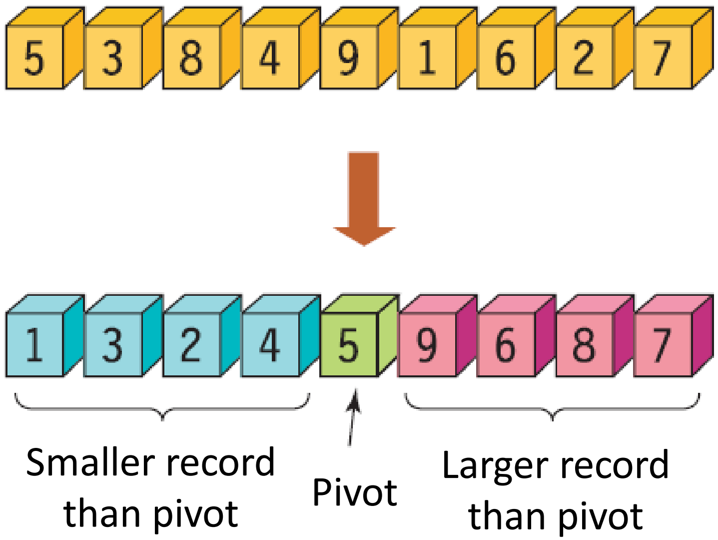
$$T(n) = 2T\left(\frac{n}{2}\right) + c \quad n \geq 2$$

$$\Rightarrow \quad \Theta(log_2 n)$$

Quick Sort

- Property
  - In average, the fastest sorting algorithm
  - Divide-and-conquer method
  - It is unstable sort

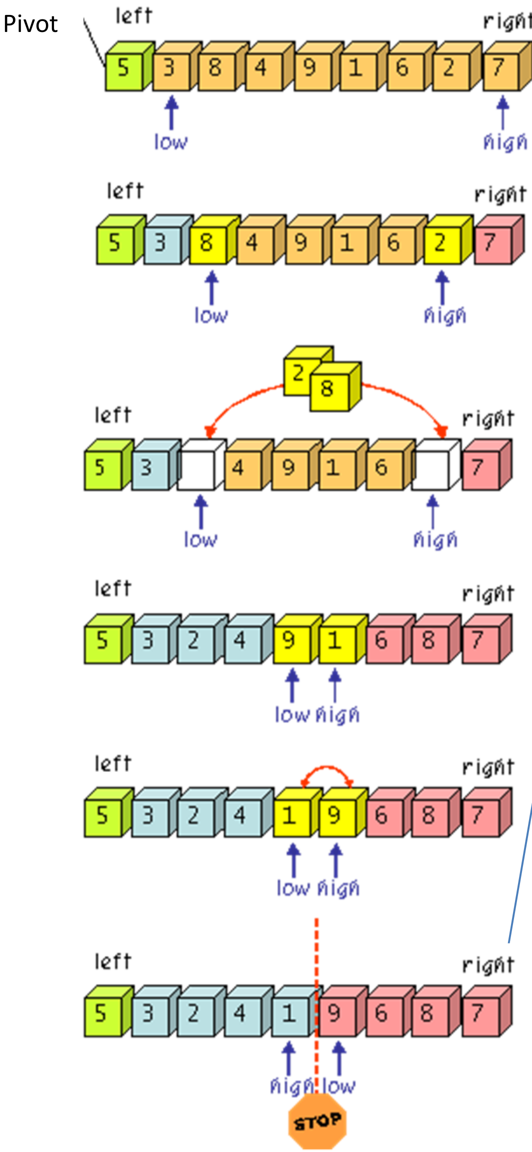
- Method
  1. Divide the array into two sizes using pivot
  2. Call the quick sort recursively.

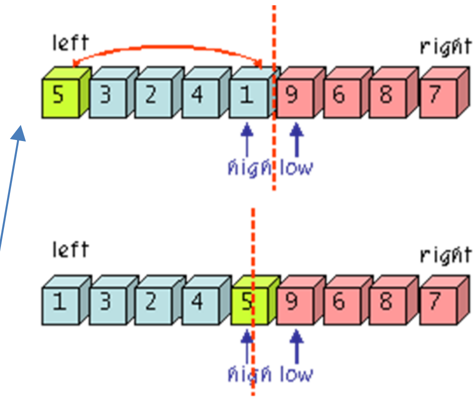


Quick Sort

```
void quick_sort(int list[], int left, int right)
{
1  if(left<right){
2     int q=partition(list, left, right);
3     quick_sort(list, left, q-1);
4     quick_sort(list, q+1, right);
   }
}
```

1. If the size of the segment is greater than 1
2. Partition into two lists based on pivot.
   The ‘partition’ function returns the position of the pivot.
3. Recursive call from left to right before the pivot (except pivot)
4. Recursive call from left next the pivot to right (except pivot)





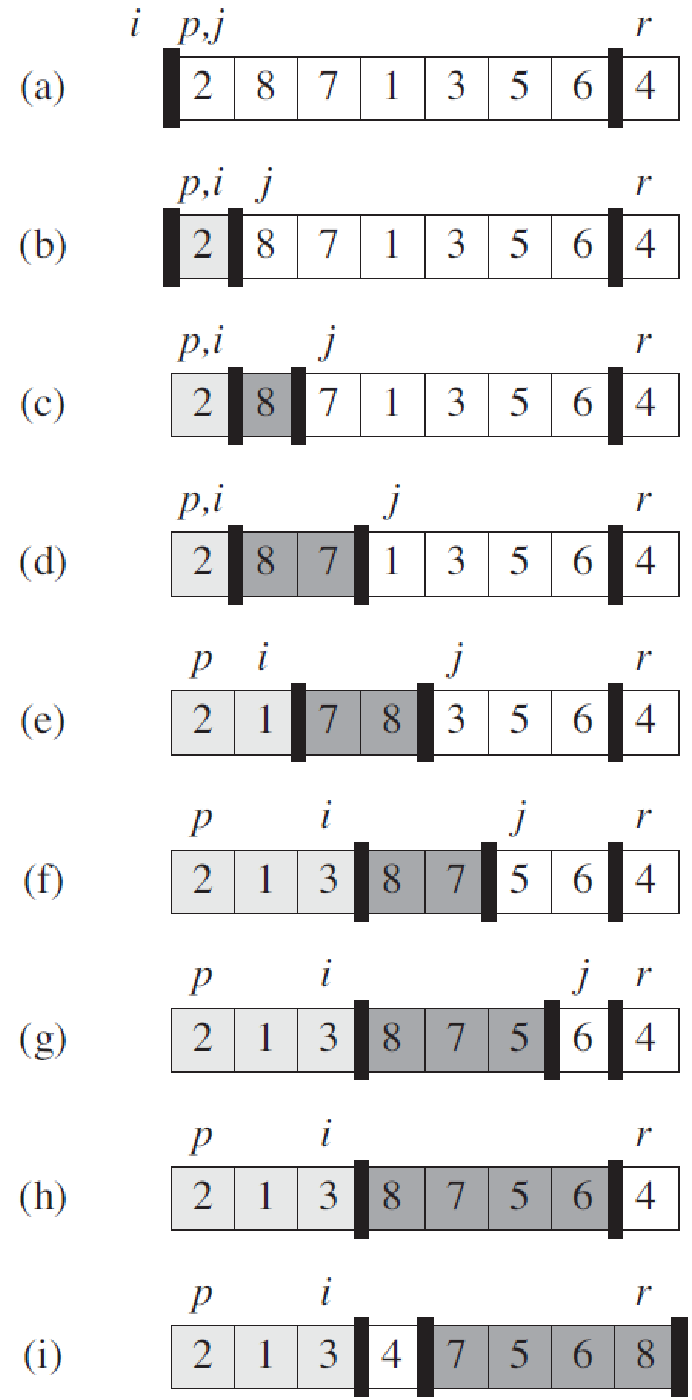
Goal:
put the data smaller (or larger) than a pivot on the left (or right) side

Pivot can be selected using any element in the current array.

```
int partition(int list[], int left, int right)
{
            int pivot, temp;
            int low, high;

            low = left + 1;
            high = right;
            pivot = list[left];
            do {
                        do
                                    low++;
                        while (low <= right &&list[low]<pivot);
                        do
                                    high--;
                        while (high >= left && list[high]>pivot);
                        if (low<high) SWAP(list[low], list[high], temp);
            } while (low<high);

            SWAP(list[left], list[high], temp);
            return high;
}
```

Another Solution for Partition

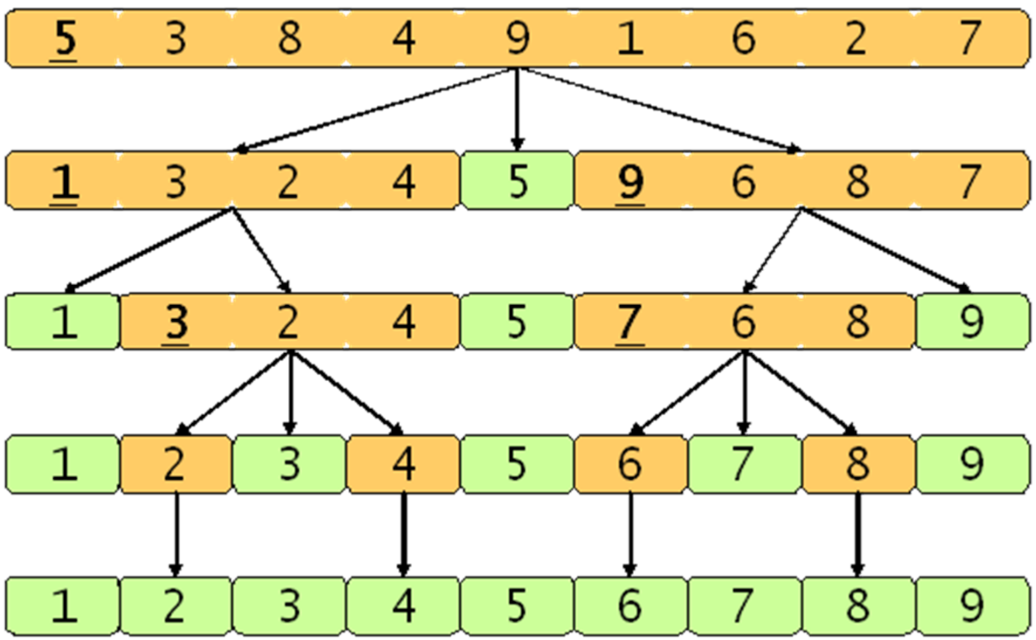
```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



What is the running time of **`partition()`**?

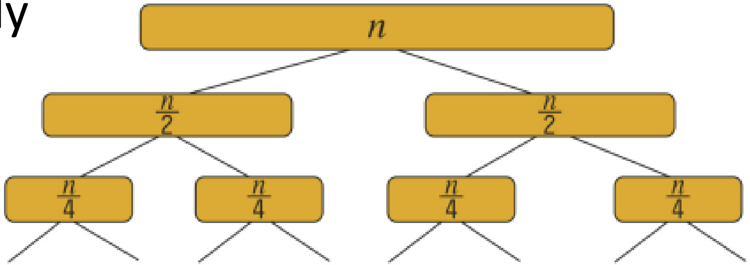
`partition()` runs in $\Theta(n)$ time

Quick Sort



Time Complexity of Quick Sort

- Best case
  - When the array is partitioned almost equally
  - # of recursions: $log_2 n$
  - # of comparisons for each recursion: n



⇨ $\Theta(nlog_2 n)$

$$T(1) = c$$

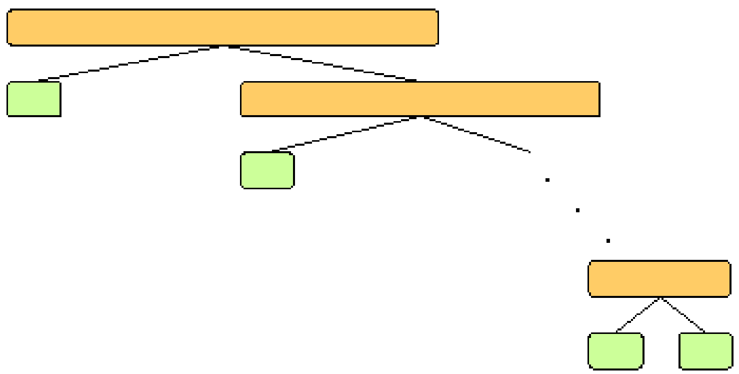
$$T(n) = 2T\left(\frac{n}{2}\right) + cn \quad n \geq 2$$

# of moves for each recursion
(<n) is negligible

Time Complexity of Quick Sort

- Worst case
  - When the array is partitioned unequally
  - # of recursions: n
  - # of comparisons for each recursion: n

$\Rightarrow$ $\Theta(n^2)$



$$T(1) = c$$

$$T(n) = T(n-1) + cn \quad n \geq 2$$

# of moves for each recursion (<n) is negligible

Ex) An already ordered input

(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

Solution

Selecting the pivot randomly or medium

Ex) Randomized quick sort

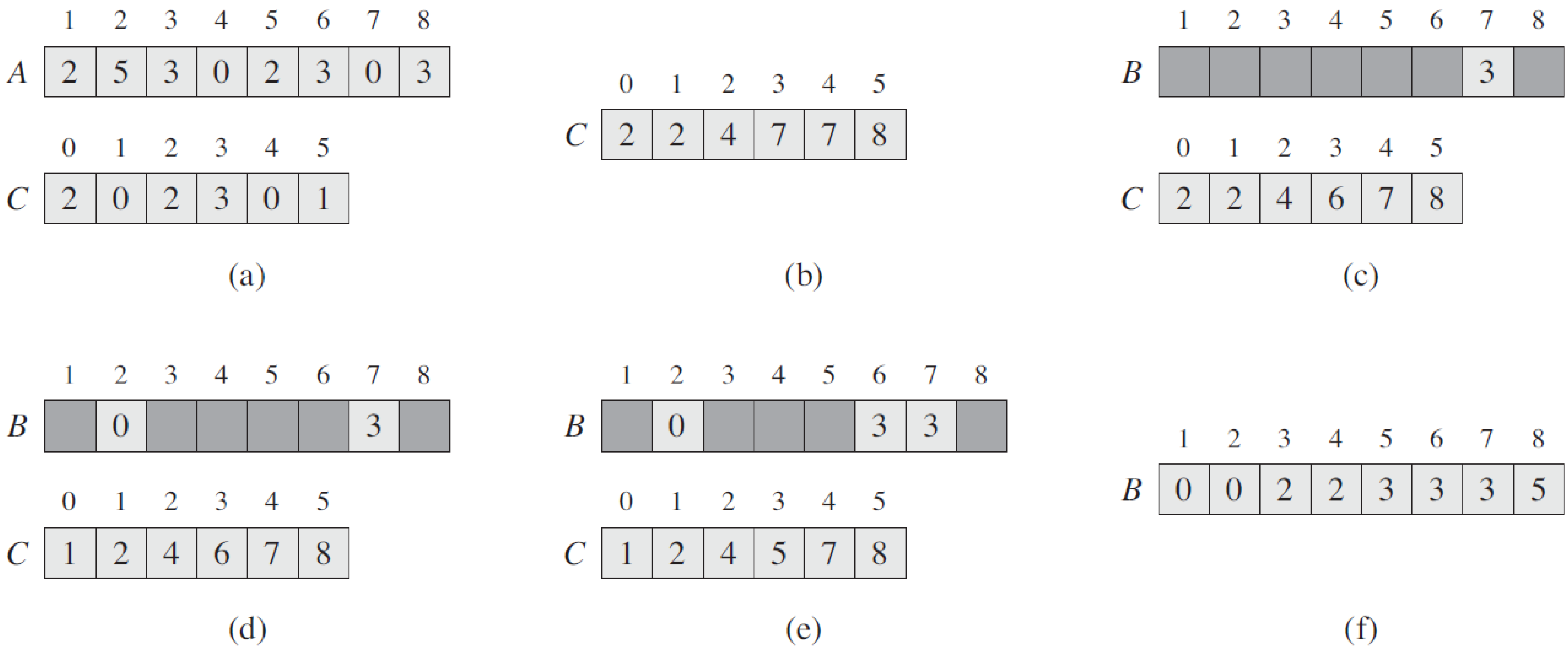
Sorting So Far

- Comparison based sorting
  - Selection sort, insert sort, bubble sort, shell sort
  - Heap sort, merge sort, quick sort
  - Time complexity: n^2 or $nlog_2n$

- Non-comparison sorting is also possible
  - Counting sort, radix sort
  - Time complexity: n

Counting Sort

`Counting-Sort(A, B, k)`



Counting Sort

```
Counting-Sort(A, B, k)
        for i=0 to k-1
                C[i]= 0;
        for j=0 to n-1
                C[A[j]] += 1;
        for i=1 to k-1
                C[i] = C[i] + C[i-1];
        for j=n-1 downto 1
                B[C[A[j]]] = A[j];
                C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

Counting Sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$

  → counting sort runs in $O(n)$ time
  - There are **<u>no comparisons at all</u>**

- Notice that this algorithm is *stable*
  - Numbers with the same value appear in the output array in the same order as in the input array

Counting Sort

- Good! *Why don't we always use counting sort?*
  - Because it depends on range *k* of elements
    (Counting sort works well only when k is rather small)
- *Could we use counting sort to sort 32 bit integers?*
  - Answer: no, *k* is too large (2^{32} = 4,294,967,296)
- *How can we sort them when k is large?*
  - Answer: Radix sort!

Radix Sort

- <u>Key idea</u>: sort the number for each digit
  - Sorting order does matter!
  - Most significant digit (MSD) vs. Least significant digit (LSD)
  - Sort the *least* significant digit (LSD) first

```
RadixSort(A, d)
       for i=1 to d
          StableSort(A) on digit i
```

Sort these for each digit!

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

- Inductive argument
  - When sorting the i-th digit, assume lower-order digits are sorted
  - Show that sorting the i-th digit leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a *stable* sort, the numbers stay in the right order

Time Complexity of Radix Sort

- Total complexity
  - For n numbers with d digits, assume each pass ranges 0 to k
  - Time complexity of each pass: $O(n+k)$
  - Total time: $O(d(n+k))$
  - When d is constant and $k = O(n)$, takes $O(n)$ time

Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time if the stable sort it uses takes $\Theta(n+k)$ time for inputs in the range 0 to k.

When r bits are used for integer,
b/r: # of digits
$k = [0, 2^r - 1]$

Time Complexity of Radix Sort

- Example: sort 1 million 64-bit numbers
  - Treat the data as four-digit radix
  - Each digit ranges $0 \sim 2^{16} - 1$

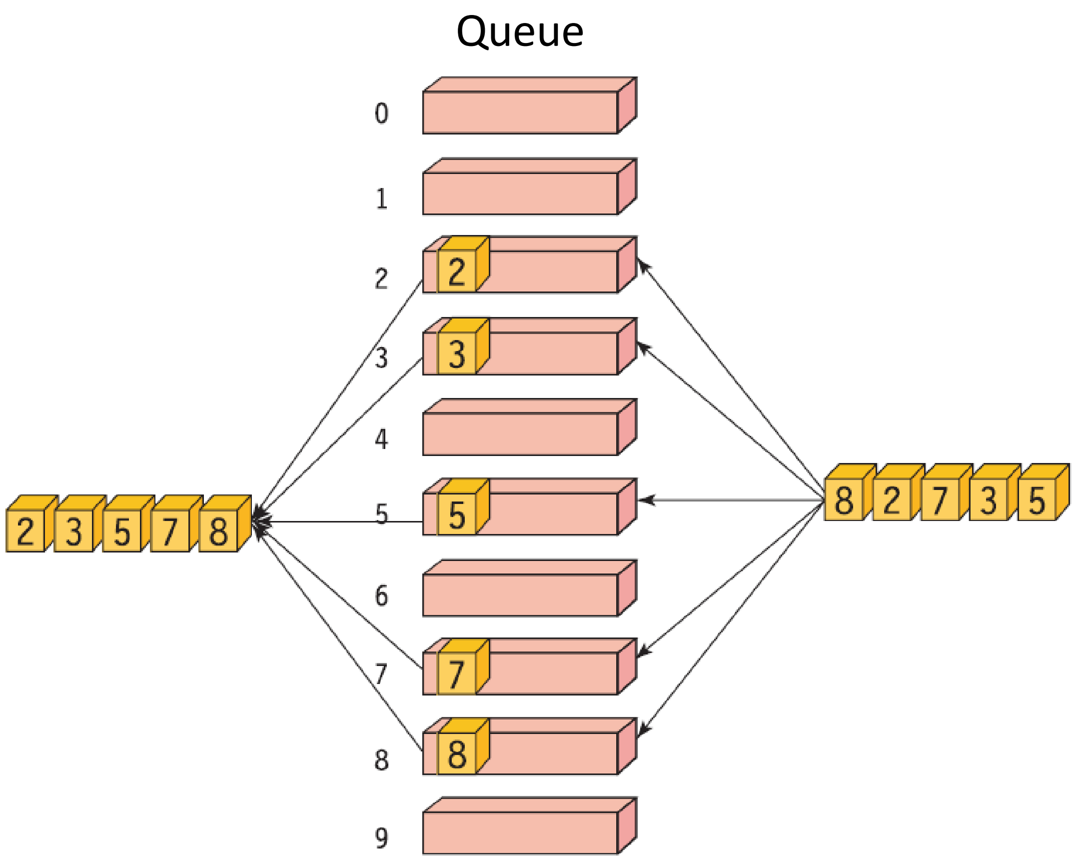
    $\Rightarrow$ $\Theta\left(4\left(n + 2^{16}\right)\right)$

  - Note) Typical comparison-based sort: $O(n log_2 n)$
    - Requires about $log_2 n = 20$ operations per number being sorted

Radix Sort using Queue

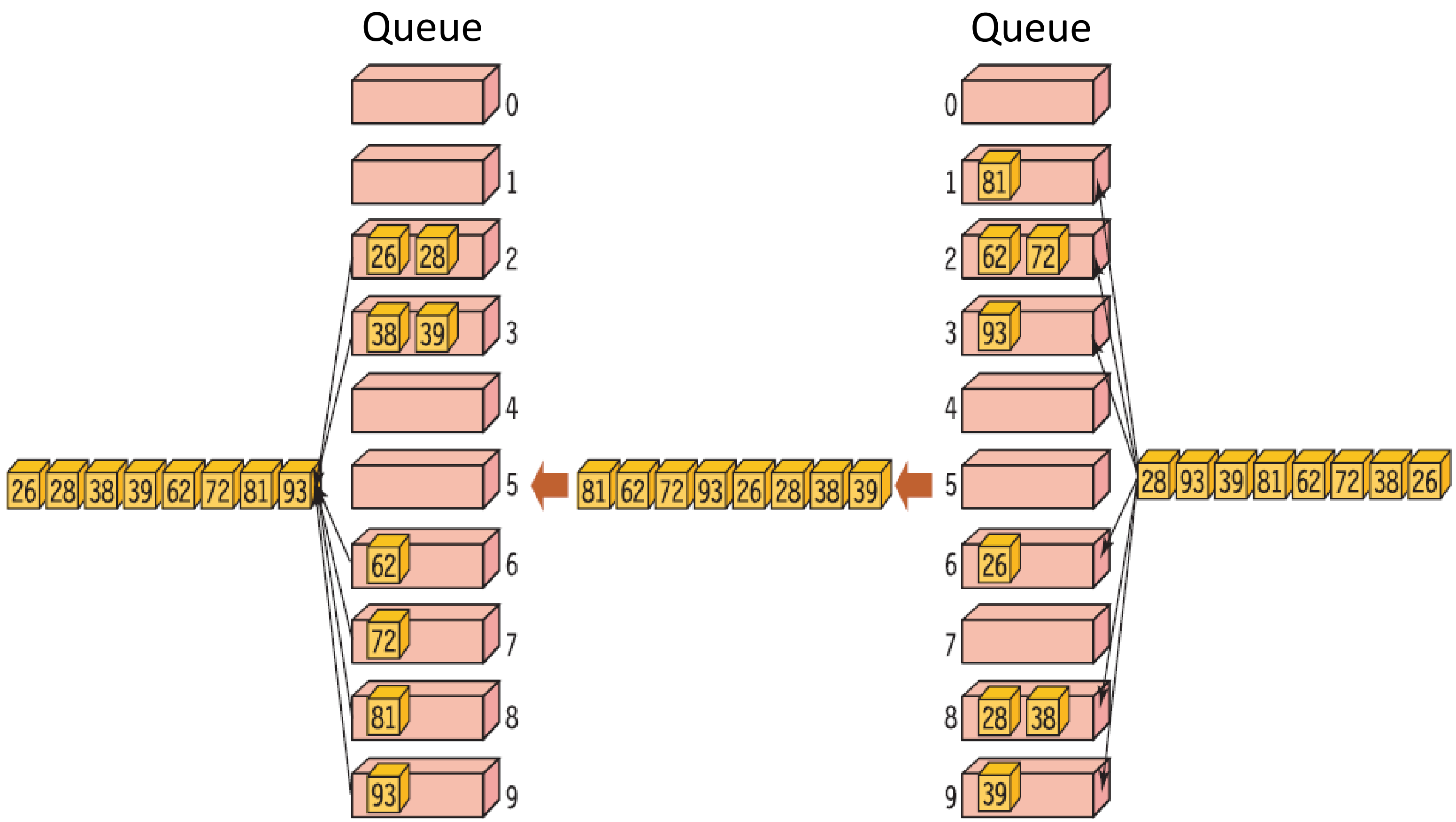
Ex) sorting 1 digit numbers (range = 0~9)

1. Set a set of queues for range 0~9
2. Execute 'Enqueue' and 'Dequeue'



Radix Sort using Queue

Ex) sorting 2 digit numbers (range of each pass = 0~9)
1. Set a set of queues for range 0~9
2. Execute ‘Enqueue’ and ‘Dequeue’
3. Iterate 1-2 for each pass (e.g. digit 1 -> digit 2)



Radix Sort using Queue

```
#define BUCKETS 10
#define DIGITS 4
void radix_sort(int list[], int n)
{
int i, b, d, factor = 1;
QueueType queues[BUCKETS];

for (b = 0; b<BUCKETS; b++) init(&queues[b]); // Initialize queues

for (d = 0; d<DIGITS; d++) {
        for (i = 0; i<n; i++)        // Add the data into queues
                enqueue(&queues[(list[i] / factor) % BUCKETS], list[i]);

        for (b = i = 0; b<BUCKETS; b++)        // Extract from queues
                while (!is_empty(&queues[b]))
                        list[i++] = dequeue(&queues[b]);
                factor *= BUCKETS; // Process next digit
        }
}
```

Time complexity: $O(d(n+k))$

Summary of Sorting Algorithms

	Best	Average	Worst	Stability
Insert sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes
Shell sort	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$	No
Quick sort	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	No
Heap sort	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	No
Merge sort	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	Yes
Count sort	$O(n)$	$O(n)$	$O(n)$	Yes
Radix sort	$O(dn)$	$O(dn)$	$O(dn)$	Yes

Runtime Measure Example

	Runtime (sec)
Insert sort	7.438
Selection sort	10.842
Bubble sort	22.894
Shell sort	0.056
Quick sort	0.014
Heap sort	0.034
Merge sort	0.026