

# 3장 분할 정복 알고리즘 (Divide-and-Conquer)

# 지난 시간에는...

## ▶ 분할 정복 알고리즘

- 주어진 문제의 입력을 분할하여 문제를 해결(정복)하는 방식의 알고리즘
- 문제를 나누어서 풀면 더 효율적인 경우 사용

## ▶ Merge sort

- 입력이 2개의 부분문제로 분할되고, 부분문제의 크기가  $1/2$ 로 감소하는 분할 정복 알고리즘

# 학습목표 및 내용

분할 정복 알고리즘의 이해

- ▶ 합병 정렬 (Merge Sort)
- ▶ 퀵 정렬 (Quick Sort)
- ▶ 선택 문제

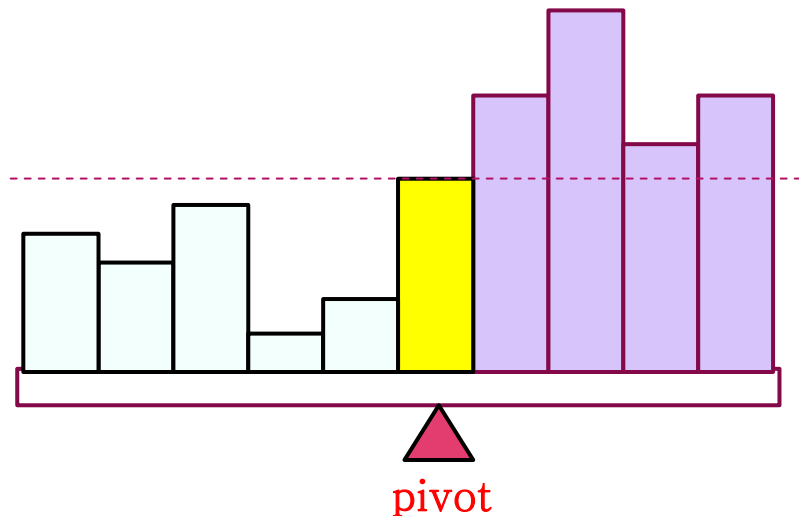
## 2. 퀵 정렬 (Quick Sort)

- ▶ 퀵 정렬은 분할 정복 알고리즘으로 분류
  - 사실 알고리즘이 수행되는 과정을 살펴보면 정복 후 분할하는 알고리즘
- ▶ 퀵 정렬 알고리즘은 문제를 2개의 부분문제로 분할
  - 각 부분문제의 크기가 일정하지 않은 형태의 분할 정복 알고리즘

# 퀵 정렬 (Quick Sort)

## ▶ 퀵 정렬의 아이디어

- 피벗 (pivot)이라 일컫는 배열의 원소(숫자)를 기준
- 피벗보다 작은 숫자들은 왼쪽, 피벗보다 큰 숫자들은 오른쪽에 위치하도록 분할
- 분할된 부분문제들에 대해서도 위와 동일한 과정을 재귀적으로 수행하여 정렬



# 퀵 정렬 (Quick Sort)

- ▶ 피벗은 분할된 왼편이나 오른편 부분에 포함되지 않음
  - 피벗이 60이라면, 60은 [20 40 10 30 50]과 [70 90 80] 사이에 위치한다.

30	80	90	70	50	20	60	10	40
----	----	----	----	----	----	----	----	----



분할 전

# How to find a pseudo code?

1. Clarify inputs/outputs
2. Clarify how to iterate them
3. Clarify actions to do in an iteration

# 퀵 정렬 알고리즘

**QuickSort**(A, left, right)

입력: 배열 A[left]~A[right]

출력: 정렬된 배열 A[left]~A[right]

1. if (left < right) {
2. 피벗을 A[left]~A[right] 중에서 선택하고, 피벗을 A[left]와 자리를 바꾼 후, 피벗과 각 원소를 비교하여 피벗보다 작은 숫자들은 A[left]~A[p-1]로 옮기고, 피벗보다 큰 숫자들은 A[p+1]~A[right]로 옮기며, 피벗은 A[p]에 둠
3. **QuickSort**(A, left, p-1)     // 피벗보다 작은 그룹
4. **QuickSort**(A, p+1, right)    // 피벗보다 큰 그룹
- }



# 퀵 정렬 알고리즘

## ▶ Line 1

- 배열 A의 가장 왼쪽 원소의 인덱스(left)가 가장 오른쪽 원소의 인덱스(right)보다 작으면, line 2~4에서 정렬을 수행
- 그렇지 않으면(즉, 1개의 원소를 정렬하는 경우), line 2~4의 정렬 과정을 수행할 필요 없이 그대로 호출을 마칩

## ▶ Line 2

- 임의의 피벗을 선택후  $A[\text{left}]$ 와 위치를 교환, 배열  $A[\text{left}+1] \sim A[\text{right}]$ 의 원소들을 피벗과 비교
- 피벗보다 작은 그룹인  $A[\text{left}] \sim A[p-1]$ 과 피벗보다 큰 그룹인  $A[p+1] \sim A[\text{right}]$ 로 분할하고  $A[p]$ 에 피벗을 위치

## ▶ Line 3

- 피벗보다 작은 그룹인  $A[\text{left}] \sim A[p-1]$ 을 재귀적으로 호출

## ▶ Line 4

- 피벗보다 큰 숫자들은  $A[p+1] \sim A[\text{right}]$ 를 재귀적으로 호출

# 퀵 정렬 알고리즘의 수행 과정

## ▶ QuickSort(A,0,11) 호출


0	1	2	3	4	5	6	7	8	9	10	11
6	3	11	9	12	2	8	15	18	10	7	14

- 피벗  $A[6]=8$ 이라면, 먼저 피벗을 가장 왼쪽으로 이동시킨다.

0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

# 어떻게 피벗을 기준으로 분류할 것인가?

## ▶ Linear search?



0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

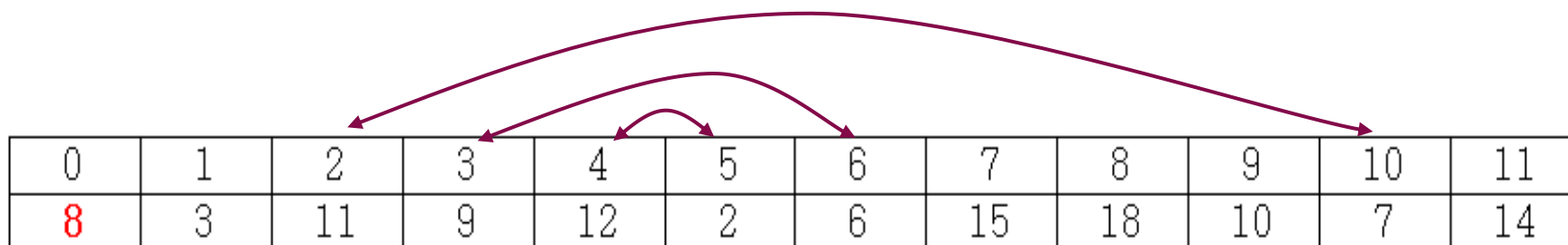
- 우리의 관심은 피벗의 위치이기 때문에 일일이 비교할 필요가 없음

## ▶ Key idea

- 배열의 앞, 뒤의 값을 비교해서 피벗의 위치를 찾음

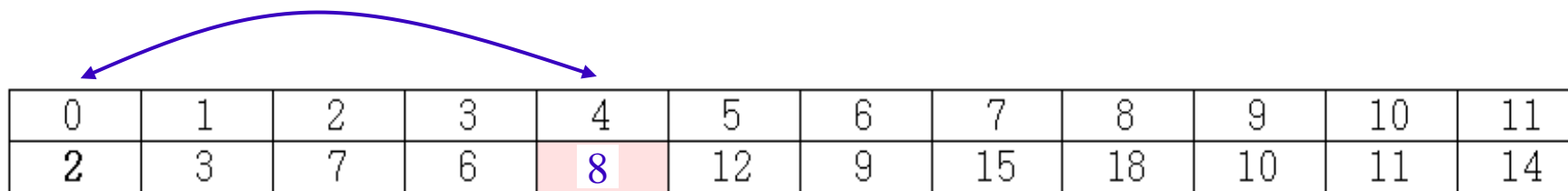
# 퀵 정렬 알고리즘의 수행 과정

- 피벗보다 큰 수와 피벗보다 작은 수를 다음과 같이 각각 교환



0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

- 피벗을 피벗보다 작으면서 가장 오른쪽에 있는 숫자와 교환(마지막 교환한 위치)



0	1	2	3	4	5	6	7	8	9	10	11
2	3	7	6	8	12	9	15	18	10	11	14

- line 3에서  $\text{QuickSort}(A, 0, 4-1) = \text{QuickSort}(A, 0, 3)$  호출
- line 4에서  $\text{QuickSort}(A, 4+1, 11) = \text{QuickSort}(A, 5, 11)$  호출

# 퀵 정렬 알고리즘의 수행 과정

## ▶ QuickSort(A,0,3) 호출

0	1	2	3
2	3	7	6

- 피벗 A[3]=6이라면, line 2에서 아래와 같이 원소들의 자리를 바꾼다.

0	1	2	3
6	3	7	2

0	1	2	3
2	3	6	7

# QuickSort(A,0,1) 호출

- line 3에서 QuickSort(A,0,2-1) = QuickSort(A,0,1) 호출
- line 4에서 QuickSort(A,2+1,3) = QuickSort(A,3,3) 호출


0	1
2	3

## ▶ QuickSort(A,0,1) 호출

- 피벗 A[1]=3이라면, line 2에서 아래와 같이 원소들의 자리를 바꾼다.

0	1
3	2

0	1
2	3



# 퀵 정렬 알고리즘의 수행 과정

- line 3에서  $\text{QuickSort}(A,0,1-1) = \text{QuickSort}(A,0,0)$  호출
- line 4에서  $\text{QuickSort}(A,1+1,1) = \text{QuickSort}(A,2,1)$  호출
- ▶  $\text{QuickSort}(A,0,0)$  호출
  - Line 1의 if-조건이 '거짓'이 되어서 알고리즘을 더 이상 수행하지 않는다.
- ▶  $\text{QuickSort}(A,2,1)$  호출
  - Line 1의 if-조건이 '거짓'이므로 알고리즘을 수행하지 않는다.
- ▶  $\text{QuickSort}(A,3,3)$ 은  $A[3]$  자체로 정렬된 것이므로,  $\text{QuickSort}(A,0,3)$ 은 아래와 같이 완성됨

0	1	2	3
2	3	6	7

# Pseudo Code

**QuickSort**(A, left, right)

입력: 배열 A[left]~A[right]

출력: 정렬된 배열 A[left]~A[right]

1. if (left < right) {
2.   피봇을 A[left]~A[right] 중에서 선택하고
3.   피봇을 A[left]와 자리를 바꾼 후
4.   A[p+1]~A[right]를 A[left]와 비교
5.   

?
6.   **QuickSort**(A, left, p-1)   // 피봇보다 작은 그룹
7.   **QuickSort**(A, p+1, right)   // 피봇보다 큰 그룹
8. }



# 시간복잡도(Worst)

- ▶ 퀵 정렬의 성능은 피벗 선택이 좌우
  - 피벗으로 가장 작은 숫자 또는 가장 큰 숫자가 선택되면, 치우치는 분할 발생
  - 피벗으로 항상 가장 작은 숫자가 선택되는 경우

피벗

1	17	42	9	18	23	31	11	26
---	----	----	---	----	----	----	----	----



1	9	42	17	18	23	31	11	26
---	---	----	----	----	----	----	----	----



...

1	9	11	17	18	23	26	31	42
---	---	----	----	----	----	----	----	----



# 시간복잡도(Worst)

- 피봇=1일 때: 8회 - [17 42 9 18 23 31 11 26]과 각각 1회씩 비교
- 피봇=9일 때: 7회 - [42 17 18 23 31 11 26]과 각각 1회씩 비교
- 피봇=11일 때: 6회 - [17 18 23 31 42 26]과 각각 1회씩 비교
- .....
- 피봇=31일 때: 1회 - [42]와 1회 비교
- 총 비교 횟수는  $8+7+6+\dots+1 = 36$ 이다.

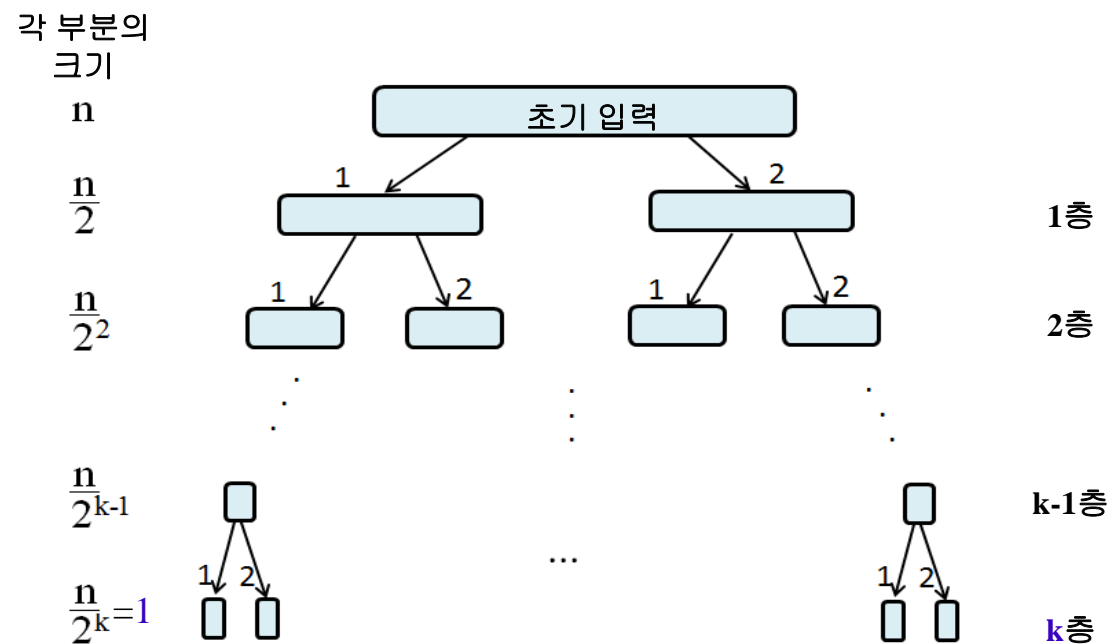
## ▶ 퀵 정렬의 최악 경우 시간복잡도

$$= (n-1)+(n-2)+(n-3)+\dots+2+1 = n(n-1)/2 = O(n^2)$$

## 시간복잡도 (Best)

## ▶ 최선 경우의 분할

- **중간의 값을 피봇으로 선택한 경우**



# 시간복잡도(Average)

## ▶ 평균 경우 시간복잡도

- 피벗을 항상 랜덤하게 선택한다고 가정하면, 퀵 정렬의 평균 경우 시간복잡도를 계산할 수 있다.
- 이때의 시간복잡도도 역시 최선 경우와 동일하게  $O(n\log_2 n)$ 이다.

# 피벗 선정 방법

- ▶ 전체를 다 비교해서 중간값 찾기
- ▶ 랜덤하게 선정
- ▶ 세개의 숫자의 중앙값으로 선정하는 방법 (Approximately sampling)
  - 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중간값으로 피벗을 선정

31	17	42	9	1	23	18	11	26
----	----	----	---	---	----	----	----	----

# 성능 향상 방법

- ▶ 입력의 크기가 매우 클 때, 퀵 정렬의 성능을 더 향상시키기 위해서, 삽입 정렬을 동시에 사용
  - 입력의 크기가 작을 때에는 퀵 정렬이 삽입 정렬보다 빠르지만은 않다.
    - 피벗 선정의 오버헤드 발생
    - 재귀 호출로 수행
  - 부분문제의 크기가 작아지면, 더 이상의 분할(재귀 호출)을 중단하고 삽입 정렬을 사용한다.

# 응용

- ▶ 쿼 정렬은 커다란 크기의 입력에 대해서 가장 좋은 성능을 보이는 정렬 알고리즘
- ▶ 쿼 정렬은 실질적으로 어느 정렬 알고리즘보다 좋은 성능
- ▶ 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는 데 접미 배열(suffix array)과 함께 쿼 정렬이 활용된다.