2021 KCA Algorithm Study

Day 3: Divide and Conquer

KU201711424

KCA Mentor **Pt J**

목차

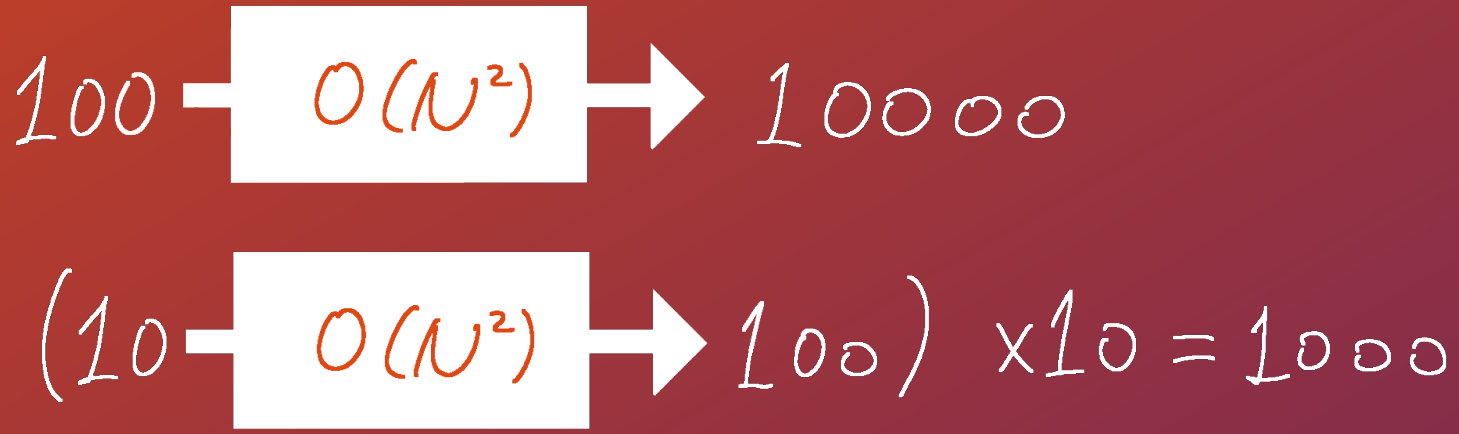
- ✔ Divide and Conquer란?
- ✔ 고려해야 할 사항
- ✔ 예시 – 합병 정렬
- ✔ 예시 – 퀵 정렬
- ✔ 예시 – k번째 수 구하기
- ✔ 주의사항

Divide and Conquer란?

- ✔ **Divide [분할]**
  주어진 문제를 더 작은 수준의 하위 문제(sub-problem)로 분할하는 과정
- ✔ **Conquer [정복]**
  하위 문제의 답을 구하고 그것을 통해 더 큰 문제의 답을 이끌어 내는 과정

Divide and Conquer란?

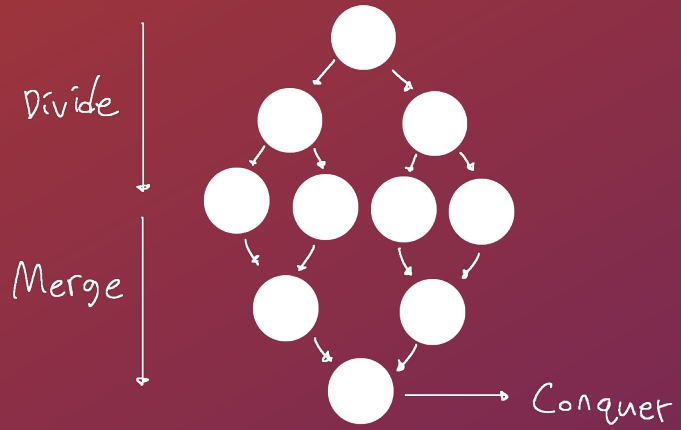
- ✔ 거대한 input에 취약한 알고리즘의 경우
  작은 단위로 나누어 해결하고 합병하는 것이
  cost를 줄일 수 있다



고려해야 할 사항

- ✔ 문제를 어떻게 분할할 것인가
- ✔ 하위 문제를 어떻게 해결할 것인가
- ✔ 하위 해결책을 어떻게 합병하여 상위 해결책을 이끌어낼 것인가

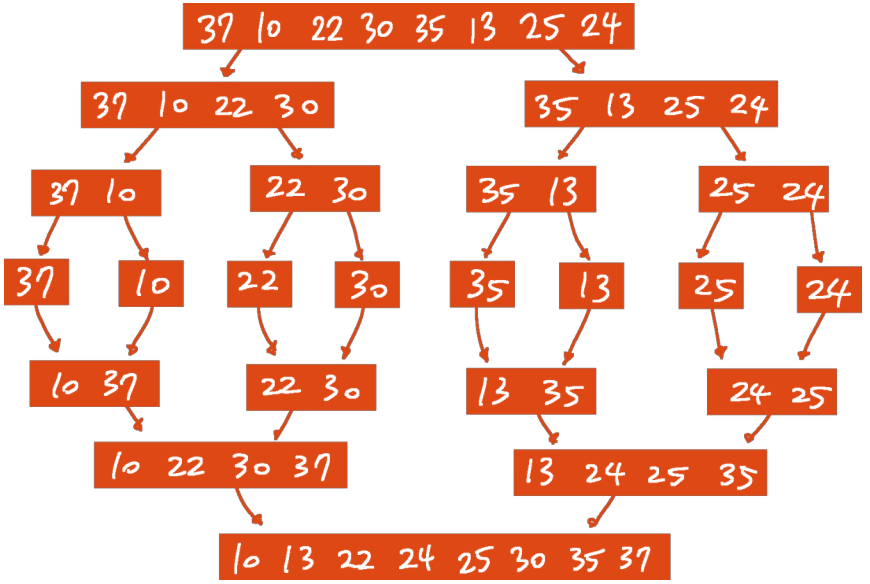
- ✔ 각 단계마다 효율성을 따져 보아야 한다



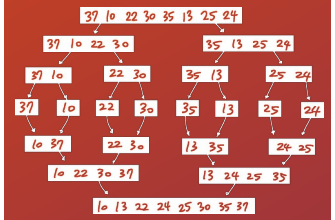
예시

첫번째 예시: 합병 정렬

- ✔ 배열을 반으로 나누어 문제의 크기를 반으로 줄인다
- ✔ 각각의 하위 문제에 대하여 합병 정렬을 수행한다



예시



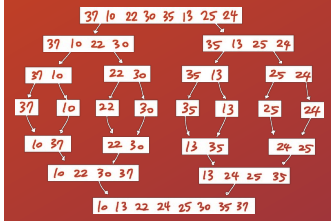
첫번째 예시: 합병 정렬

Part 1: Division

- ✔ 현재의 input size N에 대하여,
  N/2 개의 원소를 선택하여 하나의 group으로 설정
- ✔ 나머지 N/2개의 원소를 또 하나의 group으로 설정

- ✔ group의 원소가 하나 남을 때까지 반복적으로 분할

예시



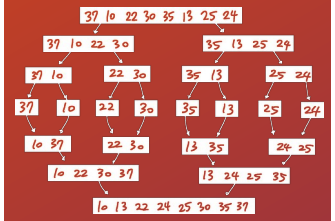
첫번째 예시: 합병 정렬

Part 2: Conquer

- ✔ group size가 1이라면 그대로 반환

- ✔ 두 개의 group을 합병해야 한다면
  각각의 group 내부는 정렬된 상태이므로
  각각의 group의 첫번째 원소끼리 비교하여
  더 작은 값을 꺼내 새 group에 넣고
  모든 원소가 새 group에 들어갈 때까지 반복한 뒤
  새 group을 반환

예시



첫번째 예시: 합병 정렬

Pseudo-code

함수 MergeSort

Input: 배열 A, 시작 인덱스 s, 끝 인덱스 e
Output: 정렬된 배열 A[s]~A[e]

```
if (s < e)
    m = (s + e) / 2
    MergeSort(A, s, m)
    MergeSort(A, m+1, e)
    B = Merge(A, s, m, e)
    copy B to A
```

함수 Merge

Input: 배열 A, 시작 인덱스 s, 중간 인덱스 m, 끝 인덱스 e
Output: 정렬된 배열 A[s]~A[e]

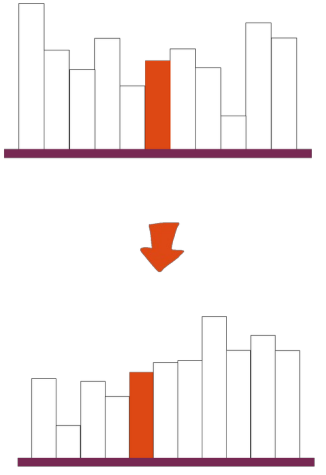
```
s2 = m + 1, i = 0
while (s <= m && s2 <= e)
    if (A[s] < A[s2])
        B[i++] = A[s++]
    else
        B[i++] = A[s1++]
if (s <= m)
    append A[s:m] to B
else
    append A[s1:e] to B
return B
```

예시

두번째 예시: 퀵 정렬

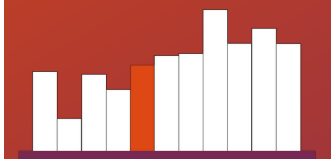
- ✔ pivot을 사용한 정렬
- ✔ division 결과 나뉘는 두 하위 문제의 size는 유동적

- ✔ 모든 숫자를 pivot 값과 비교하여
  pivot보다 크면 pivot보다 오른쪽으로,
  pivot보다 작으면 pivot보다 왼쪽으로 이동



- ✔ pivot 좌우의 하위 문제에 대해 작업 반복
- ✔ 각 step에서 결정된 pivot의 위치는 최종 위치

예시



두번째 예시: 퀵 정렬

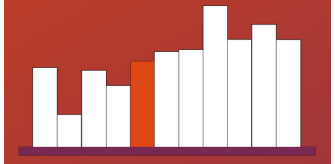
Pseudo-code

```
함수  QuickSort

Input: 배열 A, 시작 인덱스 s, 끝 인덱스 e
Output: 정렬된 배열 A[s]~A[e]

if (s < e)
    A[s:e] 구간에서 pivot을 선택하여 p = pivot index
    A[s:e] 구간에서 A[p]보다 큰 것은 왼쪽, 작은 것은 오른쪽으로 이동
    m := 이동 후 최종적인 p의 값
    QuickSort(A, s, m - 1)
    QuickSort(A, m + 1, e)
```

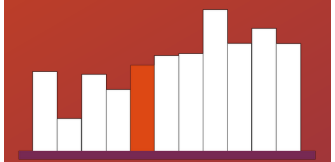
예시



두번째 예시: 퀵 정렬

- ✔ “A[s:e] 구간에서 pivot을 선택”이 불명확
  ⇒ random selection
- ✔ “A[s:e] 구간에서 A[p]보다 큰 것은 왼쪽, 작은 것은 오른쪽으로 이동”이 불명확
  ⇒ 가장 단순한 방법은 A[p]를 맨 앞으로 보내고
  이보다 작은 값을 A[p]보다 앞으로 옮기는 것
  BUT! 배열에서 하기엔 Read/Write이 너무 많다

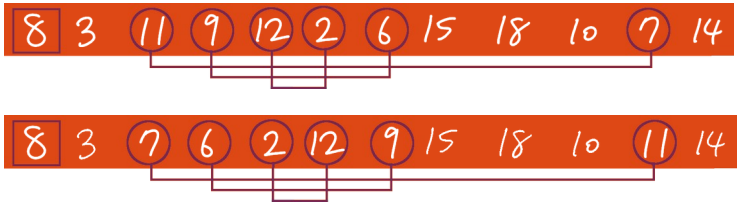
예시



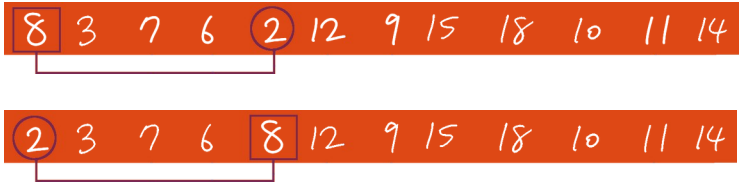
두번째 예시: 퀵 정렬

보다 효율적인 방법으로 옮기기

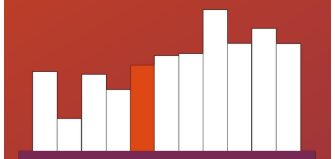
- ✔ pivot을 왼쪽 끝의 값과 서로 교환
- ✔ 왼쪽에서부터 pivot값보다 큰 값을 찾고 오른쪽에서부터 pivot 값보다 작은 값을 찾아 서로 교환



- ✔ 왼쪽에서 출발한 index가 오른쪽에서부터 출발한 index보다 커지면 오른쪽에서 출발한 index가 나타내는 원소와 pivot값을 서로 교환



예시



두번째 예시: 퀵 정렬

- ✔ pivot을 어떻게 설정하냐에 따라 보다 효율적인 퀵 정렬

- ✔ 랜덤하게?
- ✔ 어차피 왼쪽 끝으로 보내야 하니 항상 왼쪽 끝 선택?
- ✔ 혹은 그 외의 어떤 규칙 이용?

예시

세번째 예시: k번째 수 구하기

Q. N개의 숫자가 주어졌을 때 N번째로 작은 수 찾기

- ✔ 가장 단순한 방법들
  - ✔ 가장 작은 수를 찾아 삭제하는 연산을 k번 반복하여 마지막으로 나온 값을 반환
  - ✔ 정렬 후 k번째 수를 반환
- ✔ Divide and Conquer 방식을 사용한다면?

예시

세번째 예시: k번째 수 구하기

Q. N개의 숫자가 주어졌을 때 N번째로 작은 수 찾기

✔ 우리는 하위 k개의 숫자 중 가장 큰 수를 원한다
⇒ 나머지 구간은 알 필요 없다

✔ quick sort에서처럼 pivot값을 기준으로 이동하여
pivot값의 최종 위치가 k보다 크다면 왼쪽 부분을 남기고,
k보다 작다면 오른쪽 부분을 남기며,
k와 같다면 그것이 k번째 수이므로 반환

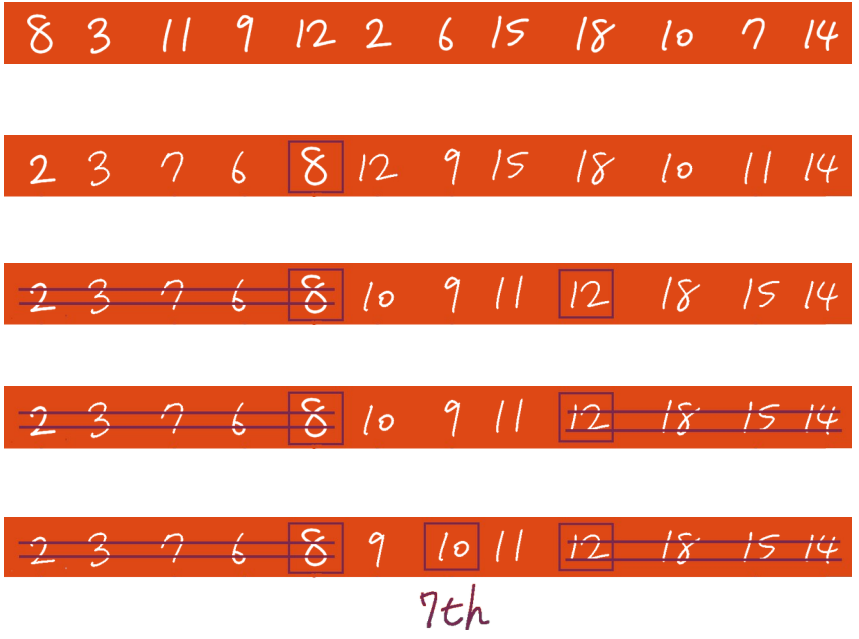
✔ 정확히는, index는 0부터 시작하므로
pivot index p에 대하여 p와 k−1 비교

예시

세번째 예시: k번째 수 구하기

Q. N개의 숫자가 주어졌을 때 N번째로 작은 수 찾기

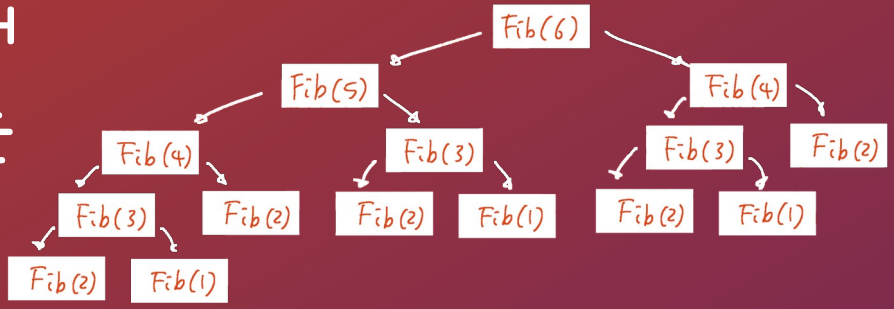
✔ 7번째 수 구하기



주의사항

- ✔ 하위 문제의 개수가 기하급수적으로 증가하지 않는가
- ✔ 하위 문제의 크기가 기하급수적으로 증가하지 않는가

- ✔ 가령, 불필요한 중복 연산으로 인해 하위 문제의 개수가 너무 많아지는 피보나치 수열이라던가?



다음 시간에는

다양한 알고리즘 (3) Dynamic Programming

에 대해 배워보도록 하겠습니다★