Professional 사전 과정

그리디 & 분할 정복

차례

탐욕 알고리즘

- 거스름돈 줄이기
- Knapsack
- 회의실 배정하기
- Review baby-gin

분할 정복

- 병합 정렬 / 퀵 정렬 / 이진 탐색

탐욕 알고리즘

문제 제시 : 거스름돈 줄이기

- 손님이 지불한 금액에서 물건값을 제한 차액(거스름돈)을 지불하는 문제를 생각해보자.

- “어떻게 하면 손님에게 거스름돈으로 주는 지폐와 동전의 개수를 최소한으로 줄일 수 있을까?”

탐욕 (Greedy) 알고리즘

- 탐욕 알고리즘은 최적해를 구하는 데 사용되는 근시안적인 방법
- 일반적으로, 머리속에 떠오르는 생각을 검증 없이 바로 구현하면 Greedy 접근이 된다.

- 여러 경우 중 하나를 선택 할 때마다 그 순간에 최적이라고 생각되는 것을 선택해 나가는 방식으로 진행하여 최종적인 해답에 도달한다.
- 각 선택 시점에서 이루어지는 결정은 지역적으로는 최적이지만, 그 선택들을 계속 수집하여 최종적인 해답을 만들었다고 하여, **그것이 최적이라는 보장은 없다.**

- 일단, 한번 선택된 것은 번복하지 않는다. 이런 특성 때문에 대부분의 탐욕 알고리즘들은 단순하며, 또한 제한적인 문제들에 적용된다.

- 최적화 문제(optimization)란 가능한 해들 중에서 가장 좋은(최대 또는 최소) 해를 찾는 문제이다.

탐욕 알고리즘의 동작 과정

1) 해 선택 : 현재 상태에서 부분 문제의 최적 해를 구한 뒤, 이를 부분해 집합(Solution Set)에 추가한다.

2) 실행 가능성 검사 : 새로운 부분 해 집합이 실행가능한지를 확인한다. 곧, 문제의 제약 조건을 위반하지 않는 지를 검사한다.

3) 해 검사 : 새로운 부분 해 집합이 문제의 해가 되는지를 확인한다. 아직 전체 문제의 해가 완성되지 않았다면 1의 해 선택부터 다시 시작한다.

탐욕 기법을 적용한 거스름돈 줄이기

1) 해 선택 : 여기에서는 멀리 내다볼 것 없이 가장 좋은 해를 선택한다. 단위가 큰 동전으로만 거스름돈을 만들면 동전의 개수가 줄어들므로 현재 고를 수 있는 가장 단위가 큰 동전을 하나 골라 거스름돈에 추가한다.

2) 실행 가능성 검사 : 거스름돈이 손님에게 내드려야 할 액수를 초과하는지 확인한다. 초과한다면 마지막에 추가한 동전을 거스름돈에서 빼고, 1로 돌아가서 현재보다 한 단계 작은 단위의 동전을 추가한다.

3) 해 검사 : 거스름돈 문제의 해는 당연히 거스름돈이 손님에게 내드려야 하는 액수와 일치하는 셈이다. 더 드려도, 덜 드려도 안되기 때문에 거스름돈을 확인해서 액수에 모자라면 다시 1로 돌아가서 거스름돈에 추가할 동전을 고른다.

최적해를 반드시 구한다는 보장이 없다.

물건 값: 1200원

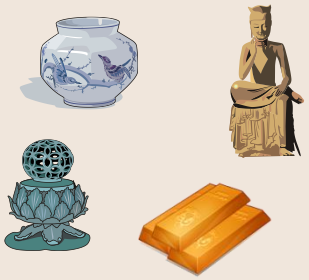
받은 돈: 2000원

거스름돈: 800원

Case 1	Case 2
500, 100, 50, 10	500, 400, 100, 50, 10
500원 : 1개 100원 : 3개 50원 : 0개 10원 : 0개	500원 : 1개 400원 : 0개 100원 : 3개 50원 : 0개 10원 : 0개

배낭 짐싸기(Knapsack)

- 도둑은 부자들의 값진 물건들을 훔치기 위해 보관 창고에 침입하였다.
- 도둑은 훔친 물건을 배낭에 담아 올 계획이다. 배낭은 담을 수 있는 물건의 총 무게(W)가 정해져 있다.
- 창고에는 여러 개(n개)의 물건들이 있고 각각의 물건에는 무게와 값이 정해져 있다.
- 경비원들에 발각되기 전에 배낭이 수용할 수 있는 무게를 초과하지 않으면서, 값이 최대가 되는 물건들을 담아야 한다.



배낭(30kg)		
물건	무게	값
물건1	25kg	10천만원
물건2	10kg	9천만원
물건3	10kg	5천만원
...	...	...

Knapsack 문제의 정형적 정의

- S = { $item_1$, $item_2$, . . ., $item_n$}, 물건들의 집합
- w_i : $item_i$ 의 무게, P_i = $item_i$ 의 값
- W : 배낭이 수용가능 한 총 무게

- 문제 정의
  - $\sum_{item_i \in A} w_i \leq W$ 를 만족하면서 $\sum_{item_i \in A} P_i$ 가 최대가 되도록 $A \subseteq S$ 가 되는 A 를 결정하는 문제

Knapsack 문제 유형

- 0-1 Knapsack
  - 배낭에 물건을 통째로 담아야 하는 문제.
  - 물건을 쪼갤 수 없는 경우.

- Fractional Knapsack
  - 물건을 부분적으로 담는 것이 허용되는 문제.
  - 물건을 쪼갤 수 있는 경우.

0-1 Knapsack에 대한 완전 검색 방법

- 완전 검색으로 물건들의 집합 S에 대한 모든 부분집합을 구한다.

- 부분집합의 총무게가 W를 초과하는 집합들은 버리고, 나머지 집합에서 총 값이 가장 큰 집합을 선택할 수 있다.

- 물건의 개수가 증가하면 시간 복잡도가 지수적으로 증가한다.
  - 크기 n인 부분합의 수 2^n

0-1 Knapsack에 대한 탐욕적 방법1

- 값이 비싼 물건부터 채운다.
- W = 30kg

	무게	값
물건1	25kg	10만원
물건2	10kg	9만원
물건3	10kg	5만원

- 탐욕적 방법의 결과
  - (물건1), 25kg, 10만원
- 최적해
  - (물건2, 물건3), 20kg, 14만원
- 최적이 아니다.

0-1 Knapsack에 대한 탐욕적 방법2

- 무게가 가벼운 물건부터 채운다.
- W = 30kg

	무게	값
물건1	25kg	15만원
물건2	10kg	9만원
물건3	10kg	5만원

- 탐욕적 방법의 결과
  - (물건2 + 물건3), 14만원
- 최적해
  - (물건1), 15만원
- 역시 최적해를 구할 수 없다.

0-1 Knapsack에 대한 탐욕적 방법3

- 무게 당 (예> kg당) 값이 높은 순서로 물건을 채운다.
- W = 30kg

	무게	값	값/kg
물건1	5kg	50만원	10만원/kg
물건2	10kg	60만원	6만원/kg
물건3	20kg	140만원	7만원/kg

- 탐욕적 방법
  - (물건1, 물건3), 190만원
- 최적해
  - (물건2, 물건3), 200만원

- 역시, 탐욕적 방법으로 최적해를 구하기 어렵다.

Fractional Knapsack 문제

- 물건의 일부를 잘라서 담을 수 있다.

	무게	값	값/kg
물건1	5kg	50만원	10만원/kg
물건2	10kg	60만원	6만원/kg
물건3	20kg	140만원	7만원/kg

- 탐욕적인 방법
  - (물건1 + 물건3 + 물건2의 절반), 30kg, 220만원

회의실 배정하기

- 김대리는 소프트웨어 개발팀들의 회의실 사용 신청을 처리하는 업무를 한다. 이번 주 금요일에 사용 가능한 회의실은 하나만 존재하고 다수의 회의가 신청된 상태이다.
- 회의는 시작 시간과 종료 시간이 있으며, 회의 시간이 겹치는 회의들은 동시에 열릴 수 없다.
- 가능한 많은 회의가 열리기 위해서는 회의들을 어떻게 배정해야 할까?
- 입력 예
  - 회의 개수
  - (시작시간, 종료 시간)

```
10
1 4   1 6   6 10   5 7   3 8   5 9   3 5   8 11   2 13   12 14
```

활동 선택(Activity-selection problem) 문제

- 시작시간과 종료시간(s_i, f_i)이 있는 n개의 활동들의 집합 A = {A_1, A_2, . . ., A_n}, $1 \leq i \leq n$ 에서 서로 겹치지 않는(non-overlapping) 최대갯수의 활동들의 집합 S를 구하는 문제

↓

- 양립 가능한 활동들의 크기가 최대가 되는 $S_{0,n+1}$ 의 부분집합을 선택하는 문제
  - 종료 시간 순으로 활동들을 정렬한다.
  - $S_{0,n+1}$ 는 a_0의 종료 시간부터 a_{n+1}의 시작시간 사이에 포함된 할동들
  - $S_{0,n+1}$ = {a_1, a_2, a_3, a_4 a_5, a_6, a_7, a_8, a_9, a_{10}} = S

i	0	1	2	3	4	5	6	7	8	9	10 (=n)	11
시작		1	3	1	5	3	5	6	8	2	12	∞
종료	0	4	5	6	7	8	9	10	11	13	14	

탐욕 기법의 적용

- 공집합이 아닌 하위 문제(subproblem) $S_{i,j}$ 가 있고 $S_{i,j}$ 에 속한 활동 a_m 은 종료 시간이 가장 빠른 활동이다.
- 그렇다면,

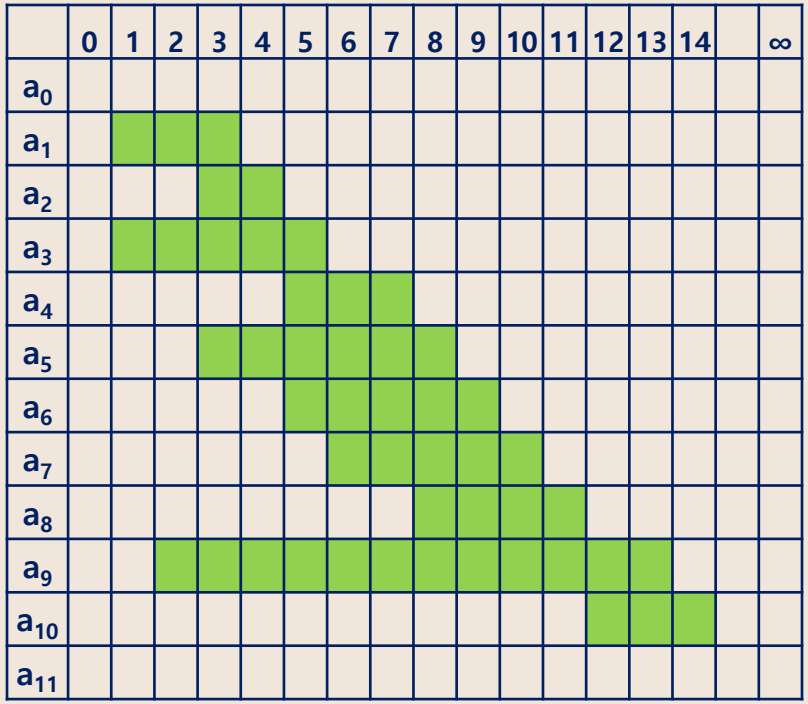
1. 하위문제(subproblem) $S_{i,j}$ 에서 종료 시간이 가장 빠른 활동 a_m을 선택한다.
2. $S_{i,m}$ 은 공집합이므로, a_m을 선택하면 공집합이 아닌 하위 문제 $S_{m,j}$ 가 남는다.

1, 2 과정을 반복한다.

탐욕 기법의 적용

- $S_{i,j}$ 를 풀기 위해

1. 종료 시간이 가장 빠른 a_m 선택
2. $S_{i,j} = \{a_m\} \cup S_{m,j}$ 의 해집합



$S_{0,11}$에 대해, a_1 선택하고 , $S_{1,11}$을 푼다.

$S_{1,11}$에 대해, a_4 선택하고, $S_{4,11}$을 푼다.

$S_{4,11}$에 대해, a_8 선택하고, $S_{8,11}$을 푼다.

탐욕적 선택
(locally optimal choice)

탑다운 방식의 문제 해결

탐욕 기법을 적용한 반복 알고리즘

```
A : 활동들의 집합,  S: 선택된 활동(회의)들 집합
s_i: 시작시간,  f_i: 종료시간,  1 ≤ i ≤ n

Sort A by finish time
S ← {A_1}
j ← 1
FOR i in 2 → n
        IF s_i ≥ f_j
                S ← S ∪ {A_i}
                j ← i
```

- 종료 시간이 빠른 순서로 활동들을 정렬한다.
- 첫 번째 활동(A_1)을 선택한다.
- 선택한 활동(A_1)의 종료시간보다 빠른 시작 시간을 가지는 활동을 모두 제거한다.
- 남은 활동들에 대해 앞의 과정을 반복한다.

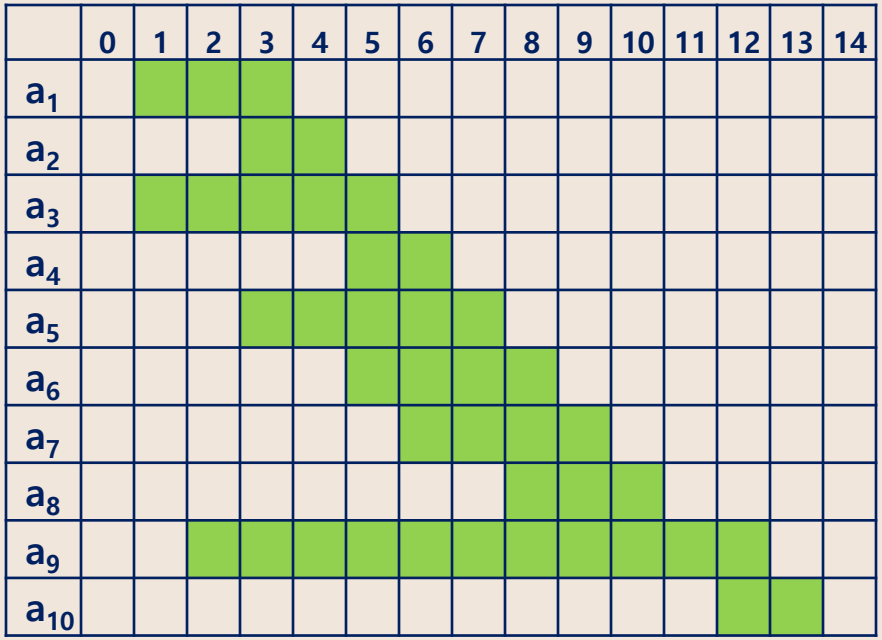
예제>

종료시간으로 정렬된 10개의 회의들

```
(1,4), (3,5), (1,6), (5,7), (3,8), (5,9), (6,10), (8,11), (2,13), (12,14)
```

가능한 활동 집합

$\{a_1, a_4, a_8, a_{10}\}$
$\{a_2, a_4, a_8, a_{10}\}$
$\{a_3, a_7, a_{10}\}$
$\{a_4, a_8, a_{10}\}$
. . .



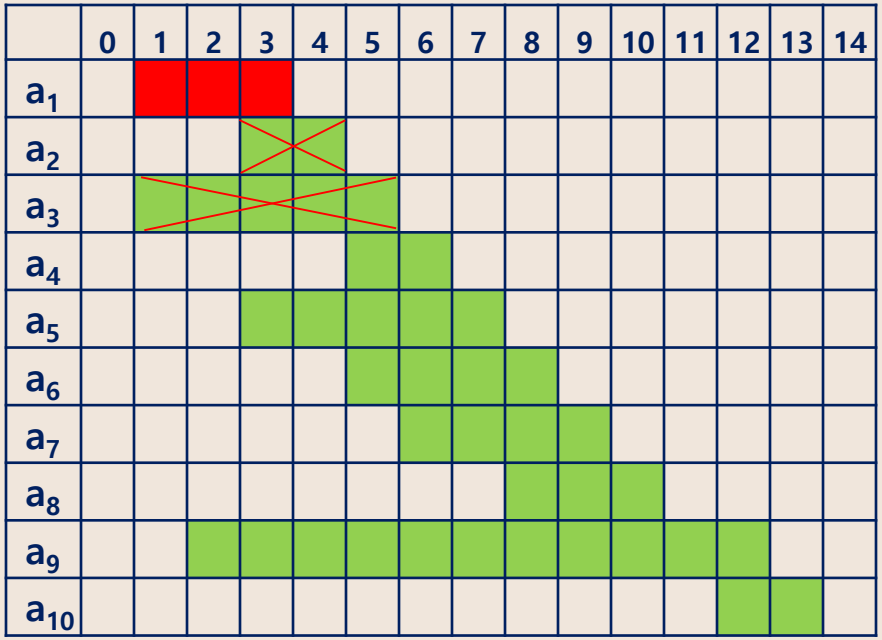
예제>

종료시간으로 정렬된 10개의 회의들

(1,4), ~~(3,5)~~, ~~(1,6)~~, (5,7), (3,8), (5,9), (6,10), (8,11), (2,13), (12,14)

선택된 회의

a_1



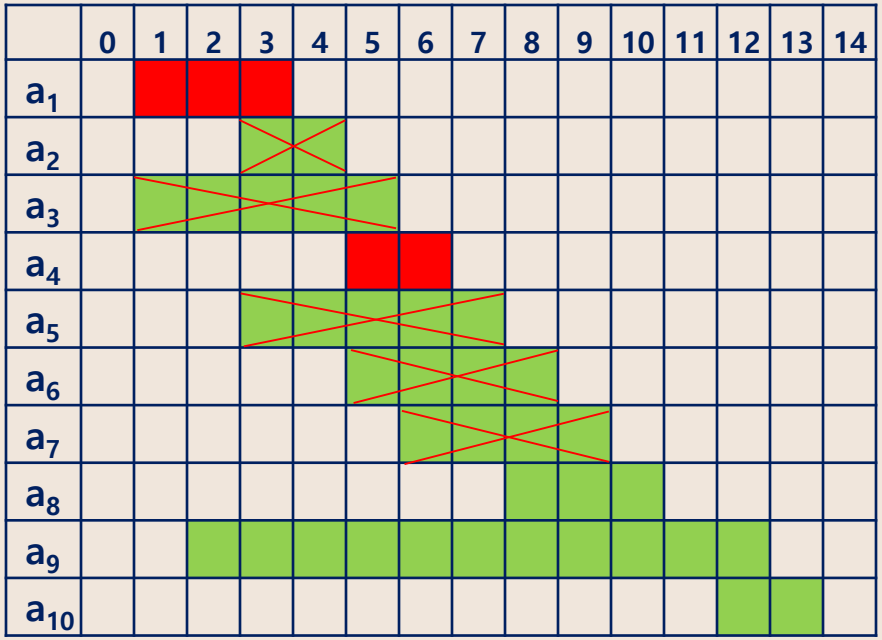
예제>

종료시간으로 정렬된 10개의 회의들

(1,4), ~~(3,5)~~, ~~(1,6)~~, (5,7), ~~(3,8)~~, ~~(5,9)~~, ~~(6,10)~~, (8,11), (2,13), (12,14)

선택된 회의

a_1
a_4



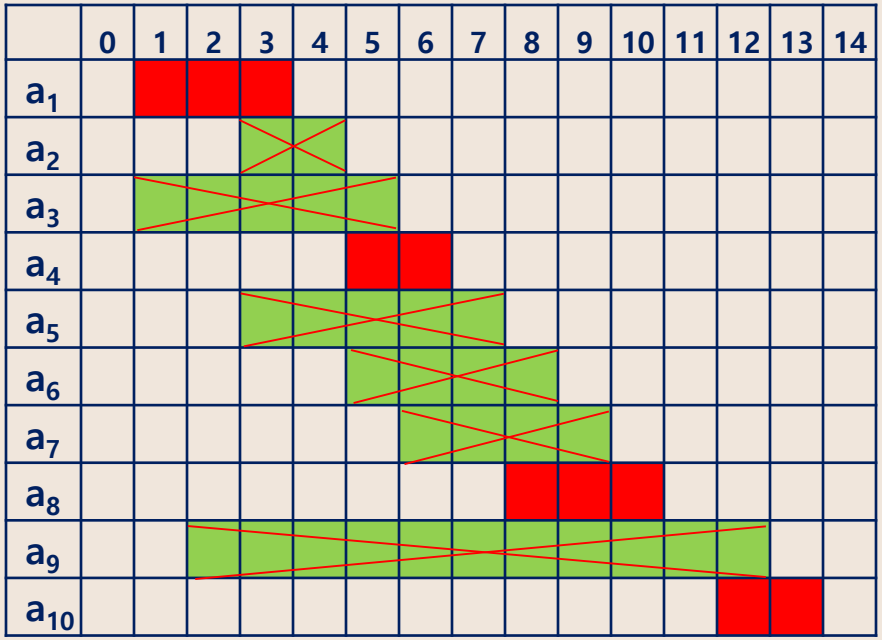
예제>

종료시간으로 정렬된 10개의 회의들

(1,4), ~~(3,5)~~, ~~(1,6)~~, (5,7), ~~(3,8)~~, ~~(5,9)~~, ~~(6,10)~~, (8,11), ~~(2,13)~~, (12,14)

선택된 회의

a_1
a_4
a_8
a_{10}



재귀 알고리즘

A: 정렬된 활동 들의 집합
S: 선택된 활동(회의)들 집합
s_i: 시작시간, f_i: 종료시간, $0 \le i \le n+1$

```
Recursive_Selection(i, j)
    m ← i + 1

    WHILE  m < j  AND s_m < f_i     // 종료 시간이 가장 빠른 활동 선택
        m ← m + 1

    IF m < j  :  RETURN {a_m} ∪ Recursive_Selection (m, j)
    ELSE      :  RETURN {}        // 공집합
```

탐욕 알고리즘의 필수 요소

- **탐욕적 선택 속성(greedy choice property)**
  - 탐욕적 선택은 최적해로 갈수 있음을 보여라. → 즉, 탐욕적 선택은 항상 안전하다.

- **최적 부분 구조(optimal substructure property)**
  - 최적화 문제를 정형화하라 → 하나의 선택을 하면 풀어야 할 하나의 하위 문제가 남는다.

- **[원문제의 최적해 = 탐욕적 선택 + 하위 문제의 최적해] 임을 증명하라.**

탐욕 기법과 동적 계획법의 비교

탐욕 기법	동적 계획법
매 단계에서, 가장 좋게 보이는 것을 빠르게 선택한다. → 지역 최적 선택(local optimal choice)	매 단계의 선택은 해결한 하위 문제의 해를 기반으로 한다.
하위 문제를 풀기 전에 (탐욕적) 선택이 먼저 이루어진다.	하위 문제가 우선 해결된다.
Top-down 방식	Bottom-up 방식
일반적으로, 빠르고 간결하다.	좀더 느리고, 복잡하다.

대표적인 탐욕 기법의 알고리즘들

알고리즘	목적	설명	
Prim	N 개의 노드에 대한 최소 신장 트리(MST)를 찾는다.	서브 트리를 확장하면서 MST를 찾는다.	그래프
Kruskal		싸이클이 없는 서브 그래프를 확장하면서 MST를 찾는다.	그래프
Dijkstra	주어진 정점에서 다른 정점들에 대한 최단 경로를 찾는다.	주어진 정점에서 가장 가까운 정점을 찾고, 그 다음을 정점을 반복해서 찾는다.	그래프
Huffman tree & code	문서의 압축을 위해 문자들의 빈도수에 따라 코드 값을 부여한다.	출현 빈도가 낮은 문자부터 선택해서 이진 트리를 완성하고 코드 값을 부여한다.	문자열

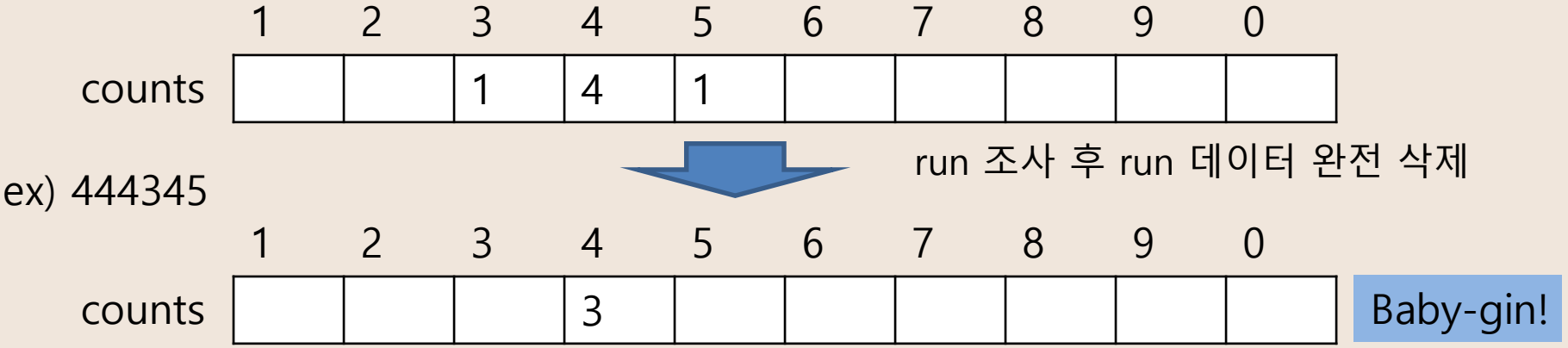
탐욕 기법을 통한 Baby-gin 문제 해결

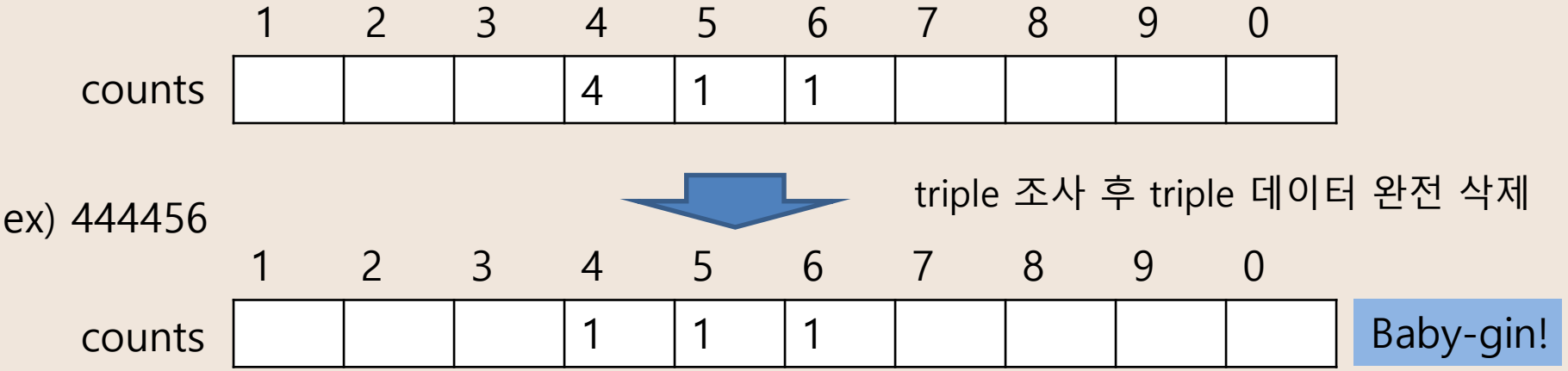
탐욕 기법을 통한 Baby-gin 문제 해결

완전검색 아닌 방법으로 풀어보자.

- 6개의 숫자는 6자리의 정수 값으로 입력된다.
- counts 배열의 각 원소를 체크하여 run과 triplet 및 baby-gin 여부를 판단한다.

풀이


1 2 3 4 5 6 7 8 9 0
counts
1 4 1
ex) 444345
run 조사 후 run 데이터 완전 삭제
1 2 3 4 5 6 7 8 9 0
counts
3
Baby-gin!


1 2 3 4 5 6 7 8 9 0
counts
4 1 1
ex) 444456
triple 조사 후 triple 데이터 완전 삭제
1 2 3 4 5 6 7 8 9 0
counts
1 1 1
Baby-gin!

알고리즘 예

```
i ← 0, inp ← 0, tri ← 0, run ← 0
inp ← input_6_numbers()
c[12] ← {0,}

WHILE  i < 6
     c[inp % 10] ← c[inp % 10] + 1
     inp ← inp / 10
     i++

i ← 0
WHILE i < 10
     IF c[i] ≥ 3
          c[i] ← c[i] - 3
          tri++
          continue
     IF c[i] ≥ 1 AND c[i + 1] ≥ 1 AND c[i + 2] ≥ 1
          c[i] ← c[i] - 1
          c[i + 1] ← c[i + 1] - 1
          c[i + 2] ← c[i + 2] - 1
          run++
          continue
     i++
```

```
IF  run + tri == 2 :  print(“Baby Gin")
ELSE               :  print("Lose")
```

분할정복

문제 제시 : 가짜 동전 찾기

- n 개의 동전들 중에 가짜 동전이 하나 포함되어 있다. 가짜 동전은 진짜 동전에 비해 아주 조금 가볍다. 진짜 동전들의 무게가 동일하다고 할 때 양팔 저울을 이용해서 가짜 동전을 찾아보자.
- 양팔 저울을 최소로 사용해서 가짜 동전을 찾는 방법은 무엇인가?

- 예를 들어 동전이 24(진짜 23, 가짜 1)개 있다면?

분할 정복 기법

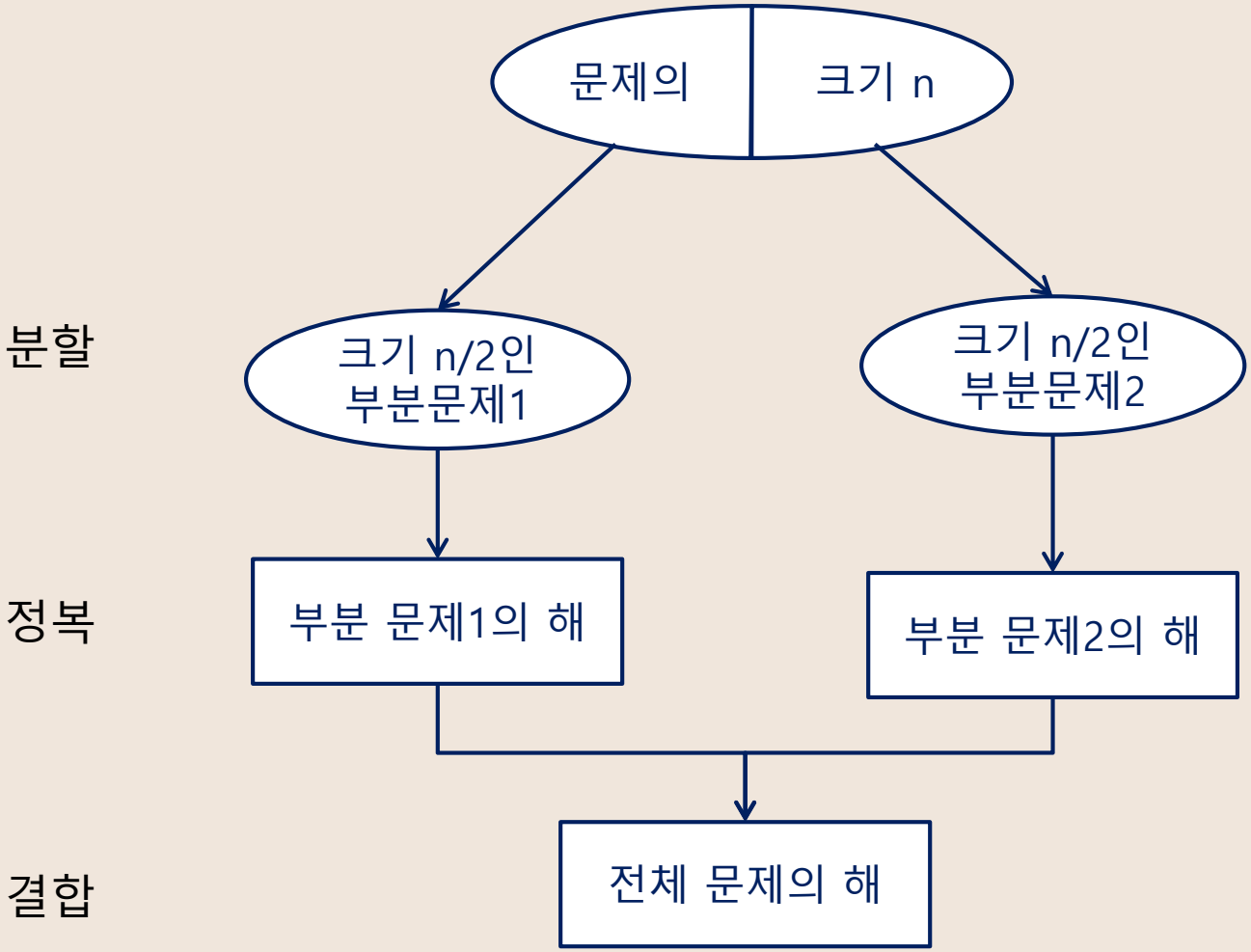
유래

- 1805년 12월 2일 아우스터리츠 전투에서 나폴레옹이 사용한 전략
- 전력이 우세한 연합군을 공격하기 위해 나폴레옹은 연합군의 중앙부로 쳐들어가 연합군을 둘로 나눔.
- 둘로 나뉜 연합군을 한 부분씩 격파함.

설계 전략

- 분할(Divide): 해결할 문제를 여러 개의 작은 부분으로 나눈다.
- 정복(Conquer): 나눈 작은 문제를 각각 해결한다.
- 통합(Combine): (필요하다면) 해결된 해답을 모은다.

Top-down approach



거듭 제곱

- 반복(Iterative) 알고리즘: O(n)

$$C^2 = C \times C$$

$$C^3 = C \times C \times C$$

$$\ldots$$

$$C^n = C \times C \times C \ldots C$$

```
Iterative_Power(x, n)
    result ← 1

    FOR i in 1 → n
        result ← result * x

    RETURN result
```

분할 정복 기반의 알고리즘 : $O(log_2 n)$

$$C^8 = C \times C \times C \times C \times C \times C \times C \times C$$
$$C^8 = C^4 \times C^4 = (C^4)^2 = ((C^2)^2)^2$$
$$C^n = C^{\frac{n-1}{2}} \times C^{\frac{n-1}{2}} \times C = (C^{\frac{n-1}{2}})^2 \times C$$

$$C^n = \begin{cases} C^{n/2} \bullet C^{n/2} & n\text{은짝수} \\ C^{(n-1)/2} \bullet C^{(n-1)/2} \bullet C & n\text{은홀수} \end{cases}$$

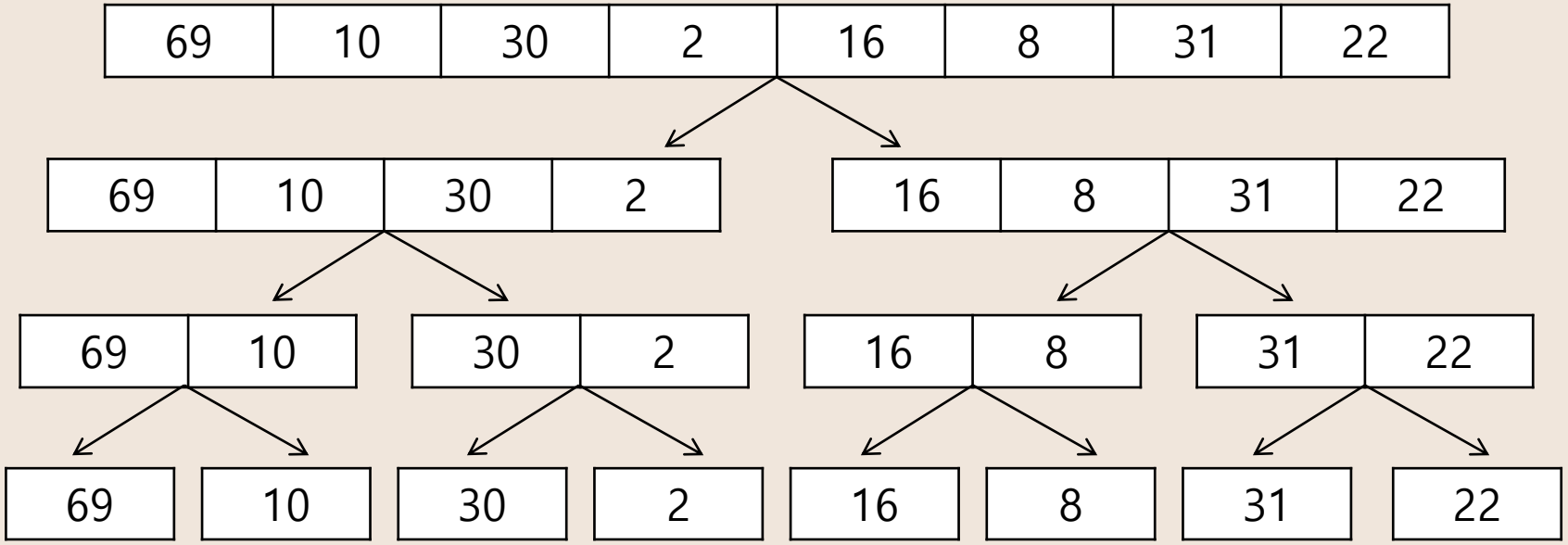
```
Recursive_Power(x, n)
    IF n == 1 : RETURN x
    IF n is even
        y ← Recursive_Power(x, n/2)
        RETURN y * y
    ELSE
        y ← Recursive_Power(x, (n-1)/2)
        RETURN y * y * x
```

병합 정렬 (Merge Sort)

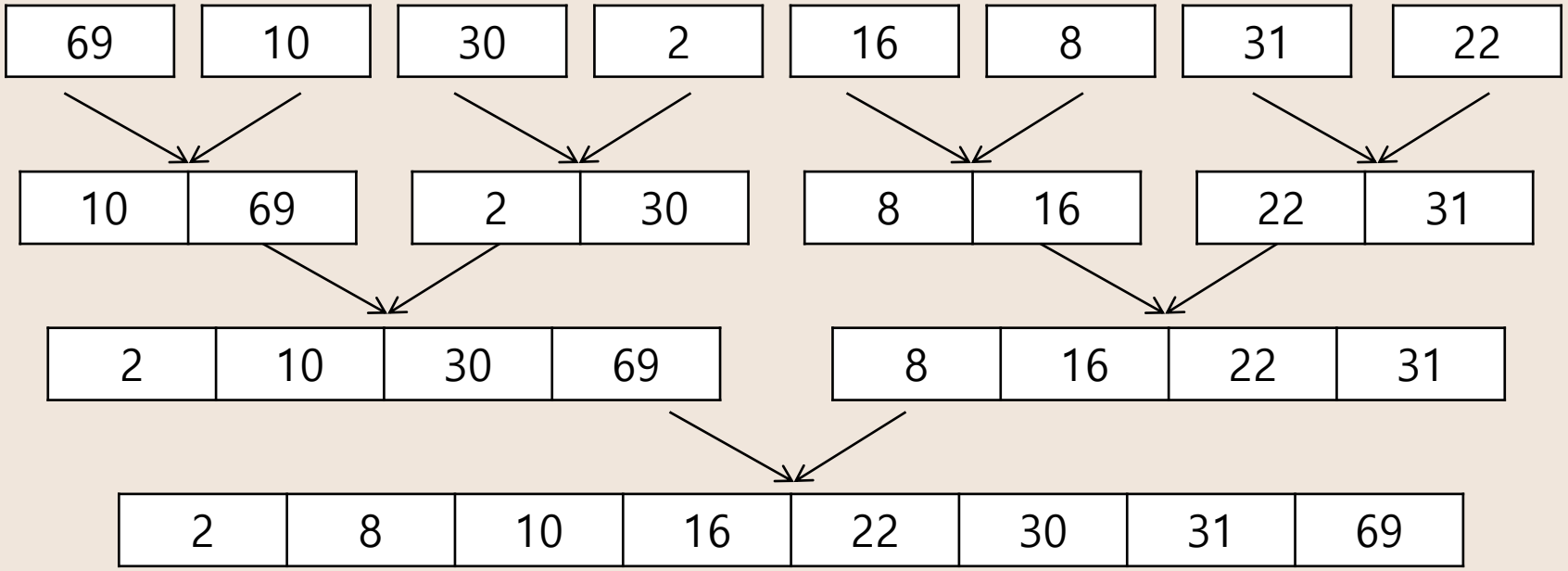
- **여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방식**
- **분할 정복 알고리즘 활용**
  - 자료를 최소 단위의 문제까지 나눈 후에 차례대로 정렬하여 최종 결과를 얻어냄.
  - top-down 방식
- **시간 복잡도**
  - O(n log n)

병합 정렬 과정

- {69, 10, 30, 2, 16, 8, 31, 22}를 병합 정렬하는 과정
- 분할 단계: 전체 자료 집합에 대하여, 최소 크기의 부분집합이 될 때까지 분할 작업을 계속한다.



- 병합 단계: 2개의 부분집합을 정렬하면서 하나의 집합으로 병합
- 8개의 부분집합이 1개로 병합될 때까지 반복함



알고리즘 : 분할 과정

```
merge_sort(LIST m)
    IF length(m) == 1 : RETURN m

    LIST left, right
    middle ← length(m) / 2
    FOR x in m before middle
         add x to left
    FOR x in m after or equal middle
         add x to right

    left ← merge_sort(left)
    right ← merge_sort(right)

    RETURN merge(left, right)
```

알고리즘 : 병합 과정

```
merge(LIST left, LIST right)
    LIST result

    WHILE length(left) > 0 OR length(right) > 0
        IF length(left) > 0 AND length(right) > 0
            IF first(left) <= first(right)
                append popfirst(left) to result
            ELSE
                append popfirst(right) to result
        ELIF length(left) > 0
            append posfirst(left) to result
        ELIF length(right) > 0
            append popfirst(right) to result

    RETURN result
```

퀵 정렬

- 주어진 배열을 두 개로 분할하고, 각각을 정렬한다.
  - 합병정렬과 동일?

- 다른점 1: 병합정렬은 그냥 두 부분으로 나누는 반면에, 퀵정렬은 분할할 때, 기준 아이템(pivot item) 중심으로, 이보다 작은 것은 왼편, 큰 것은 오른편에 위치시킨다.

- 다른점 2: 각 부분 정렬이 끝난 후, 병합정렬은 “병합” 이란 후처리 작업이 필요하나, 퀵정렬은 필요로 하지 않는다.

알고리즘

```
quickSort(A[], l, r)
        if l < r
            s ← partition(a, l, r)
            quickSort(A[], l, s - 1)
            quickSort(A[], s + 1, r)
```

Hoare-Partition 알고리즘

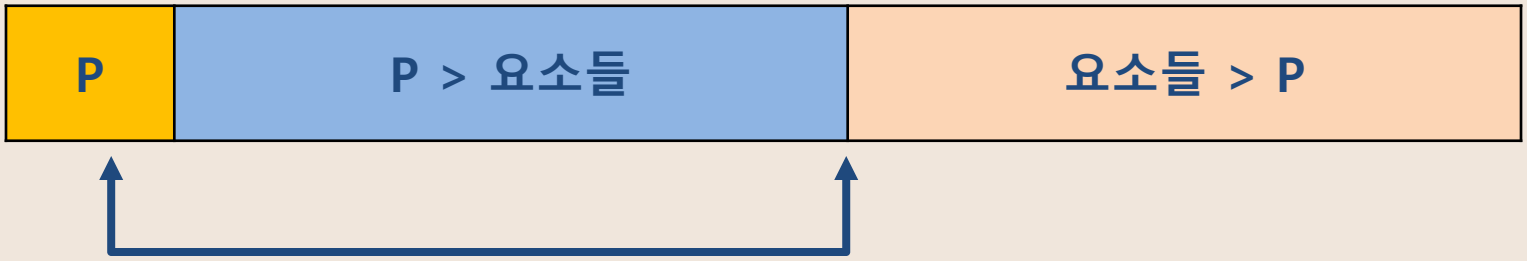
```
partition(A[], l, r)
        p ← A[l]                    // p: 피봇 값
        i ← l,  j ← r
        WHILE i ≤ j
            WHIEL A[i] ≤ p : i++
            WHILE A[j] ≥ p : j--
            IF i < j : swap(A[i], A[j])

        swap(A[l], A[j])
        RETURN j
```

아이디어

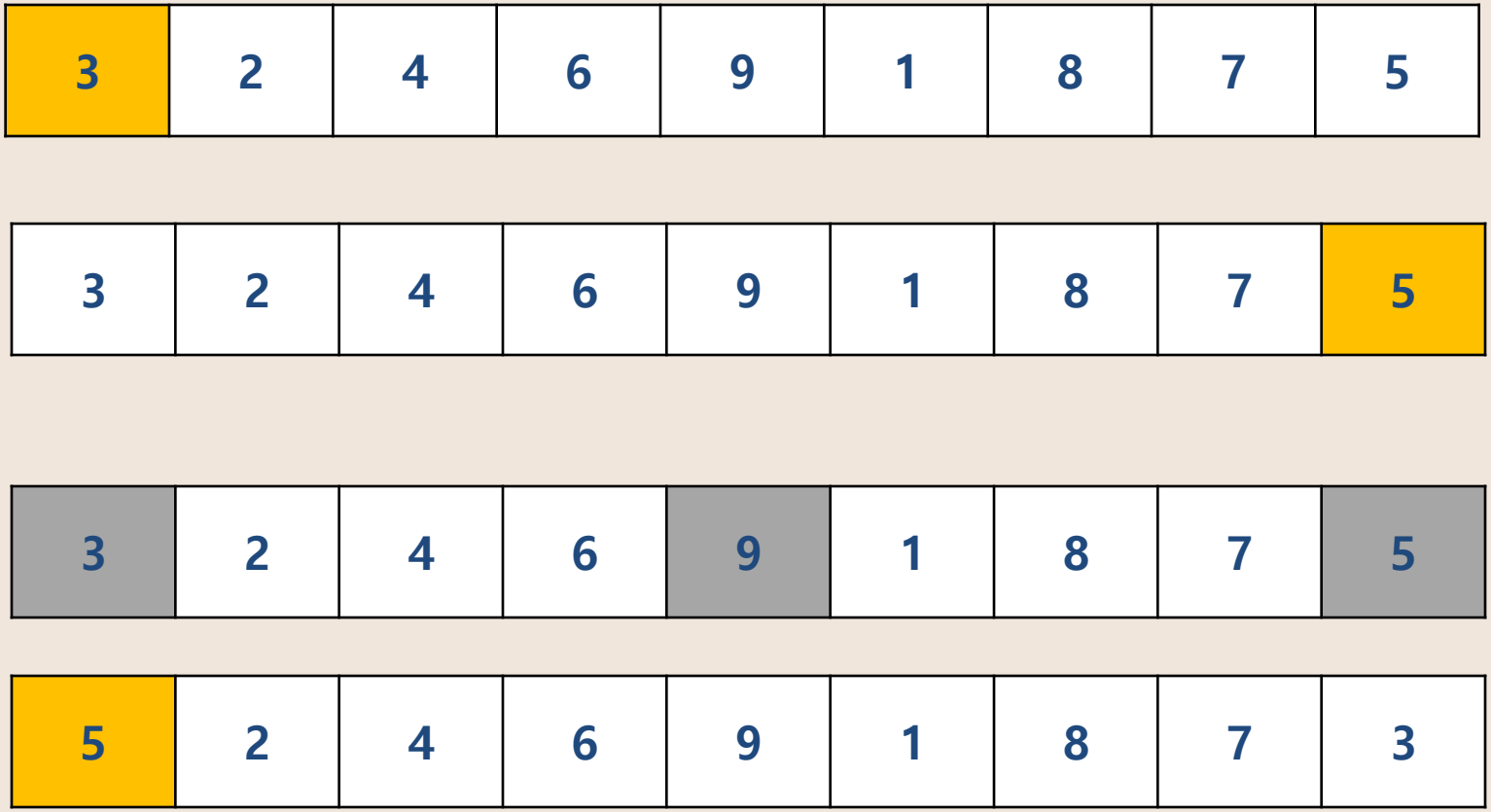
- P(피봇)값들 보다 큰 값은 오른쪽, 작은 값들은 왼쪽 집합에 위치하도록 한다.

- 피봇을 두 집합의 가운데에 위치시킨다.

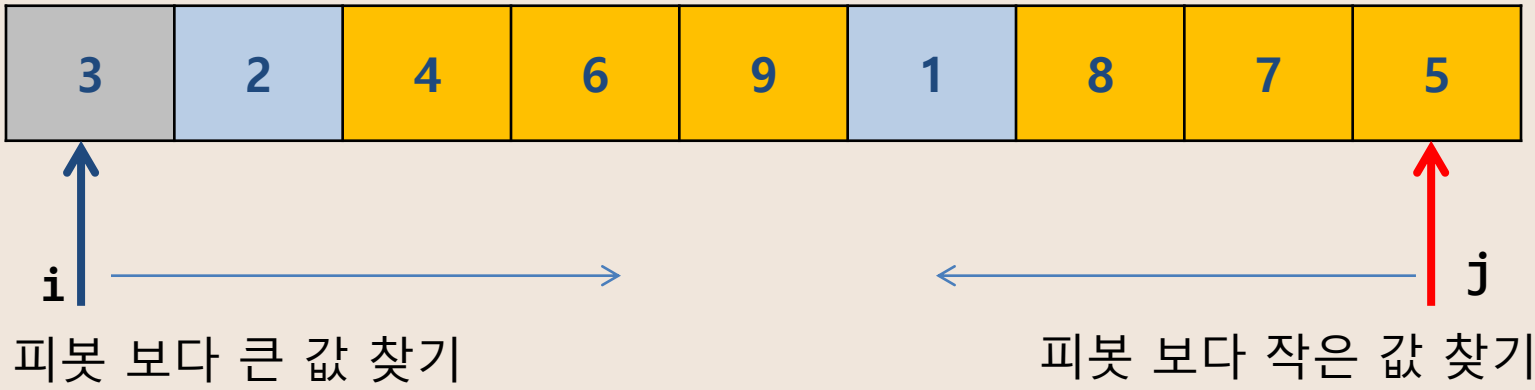


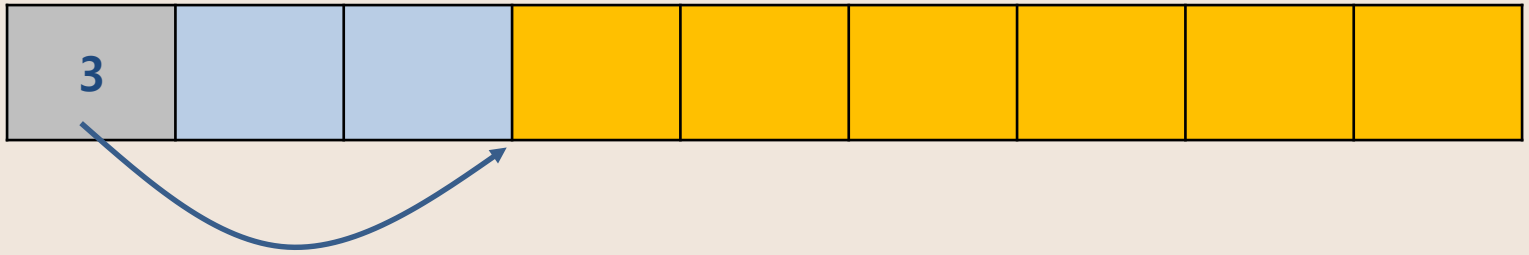
피봇 선택

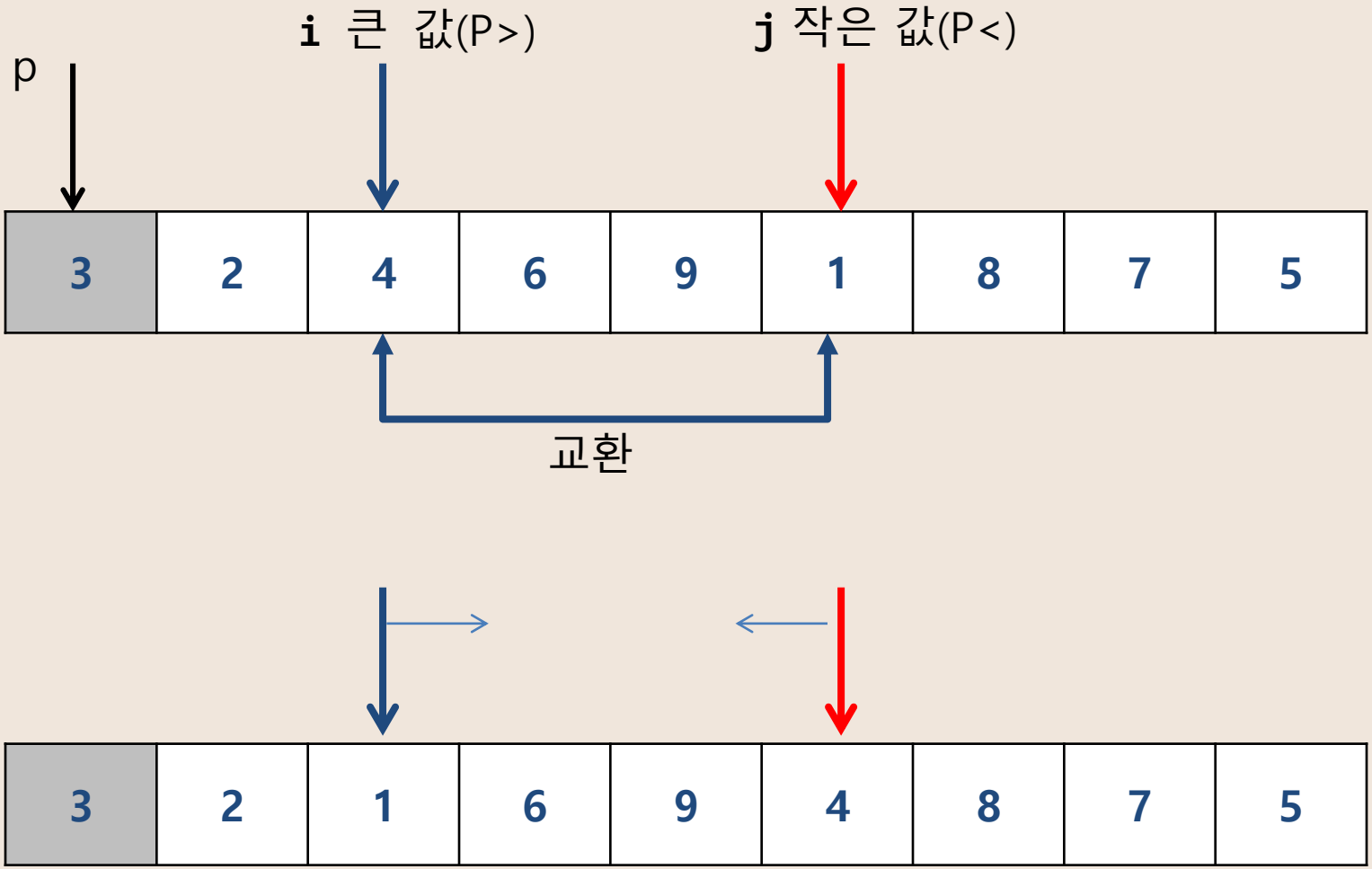
- 왼쪽 끝/오른쪽 끝 / 임의의 세 개 값 중에 중간 값

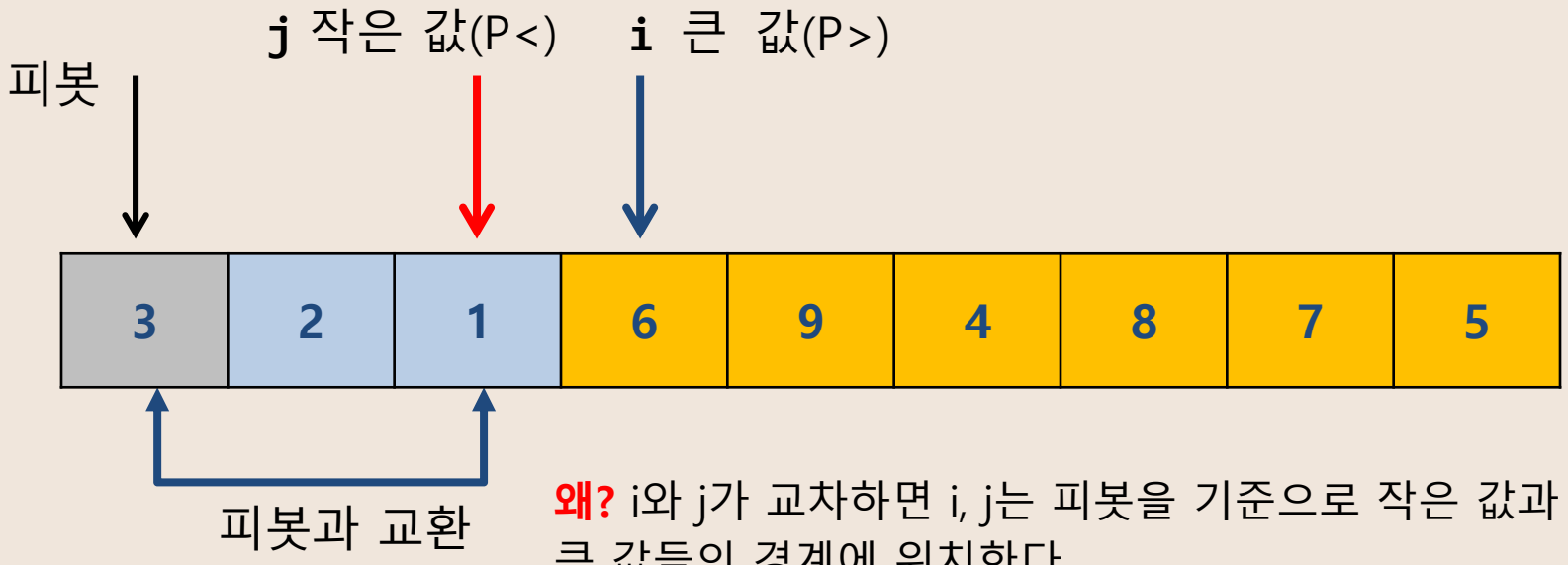


피봇

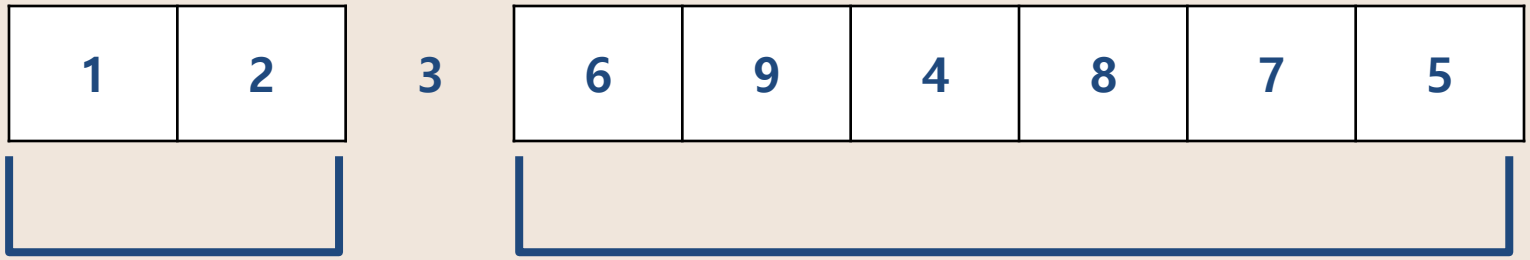


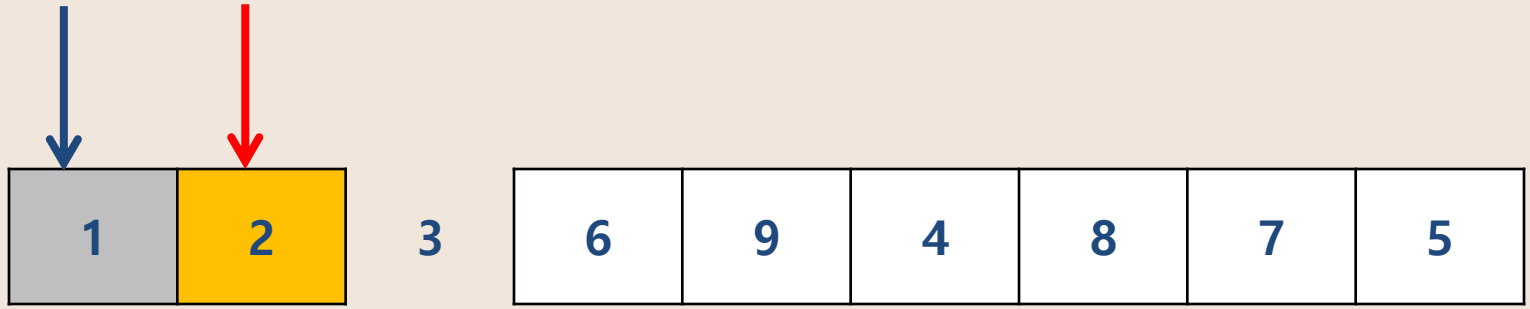


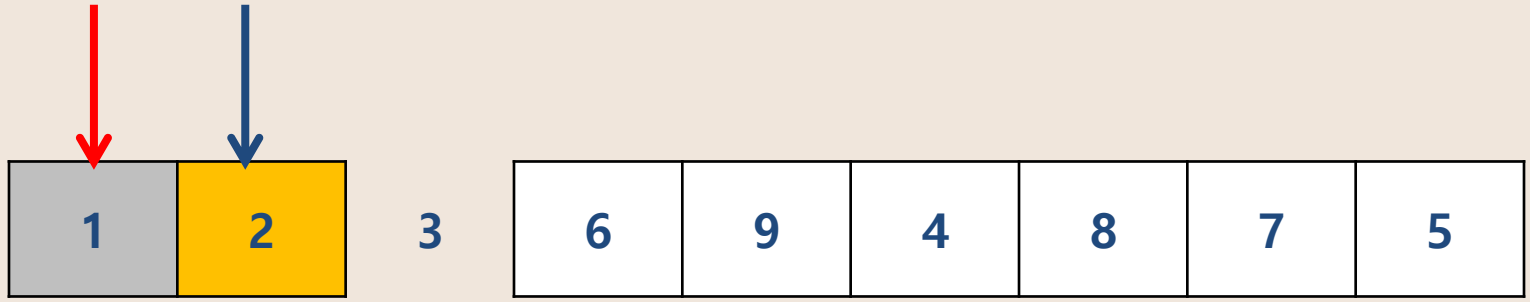

i 큰 값(P>)
j 작은 값(P<)
p
3
2
4
6
9
1
8
7
5
교환
3
2
1
6
9
4
8
7
5

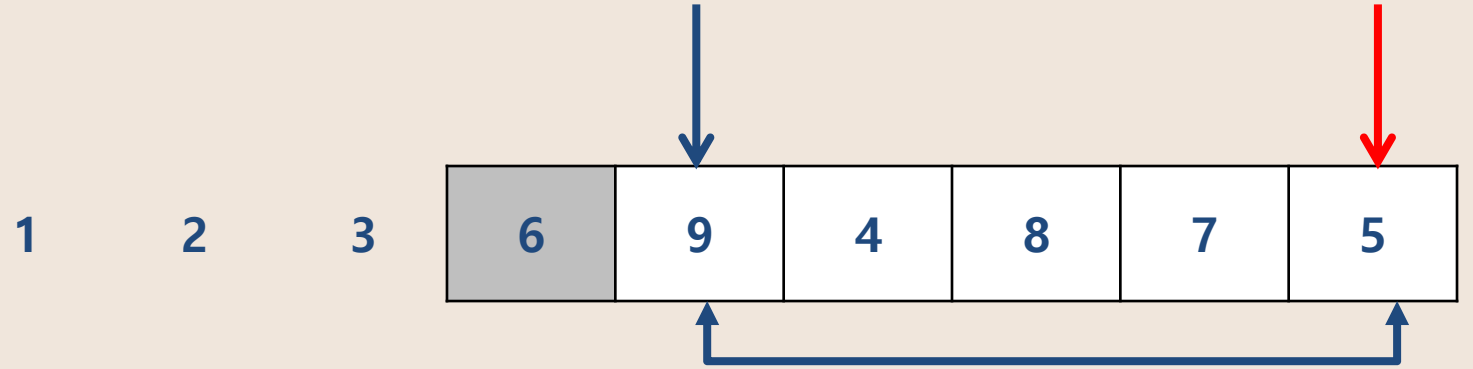


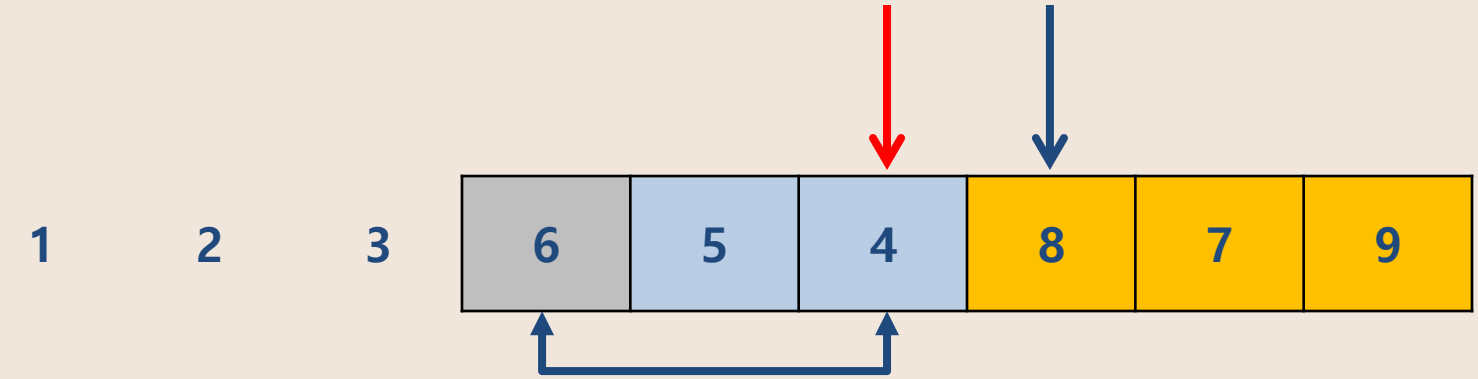
왜? i와 j가 교차하면 i, j는 피봇을 기준으로 작은 값과 큰 값들의 경계에 위치한다.


1
2
3
6
9
4
8
7
5


1
2
3
6
9
4
8
7
5


1
2
3
6
9
4
8
7
5


1
2
3
6
9
4
8
7
5


1
2
3
6
5
4
8
7
9


1
2
3
4
5
6
8
7
9

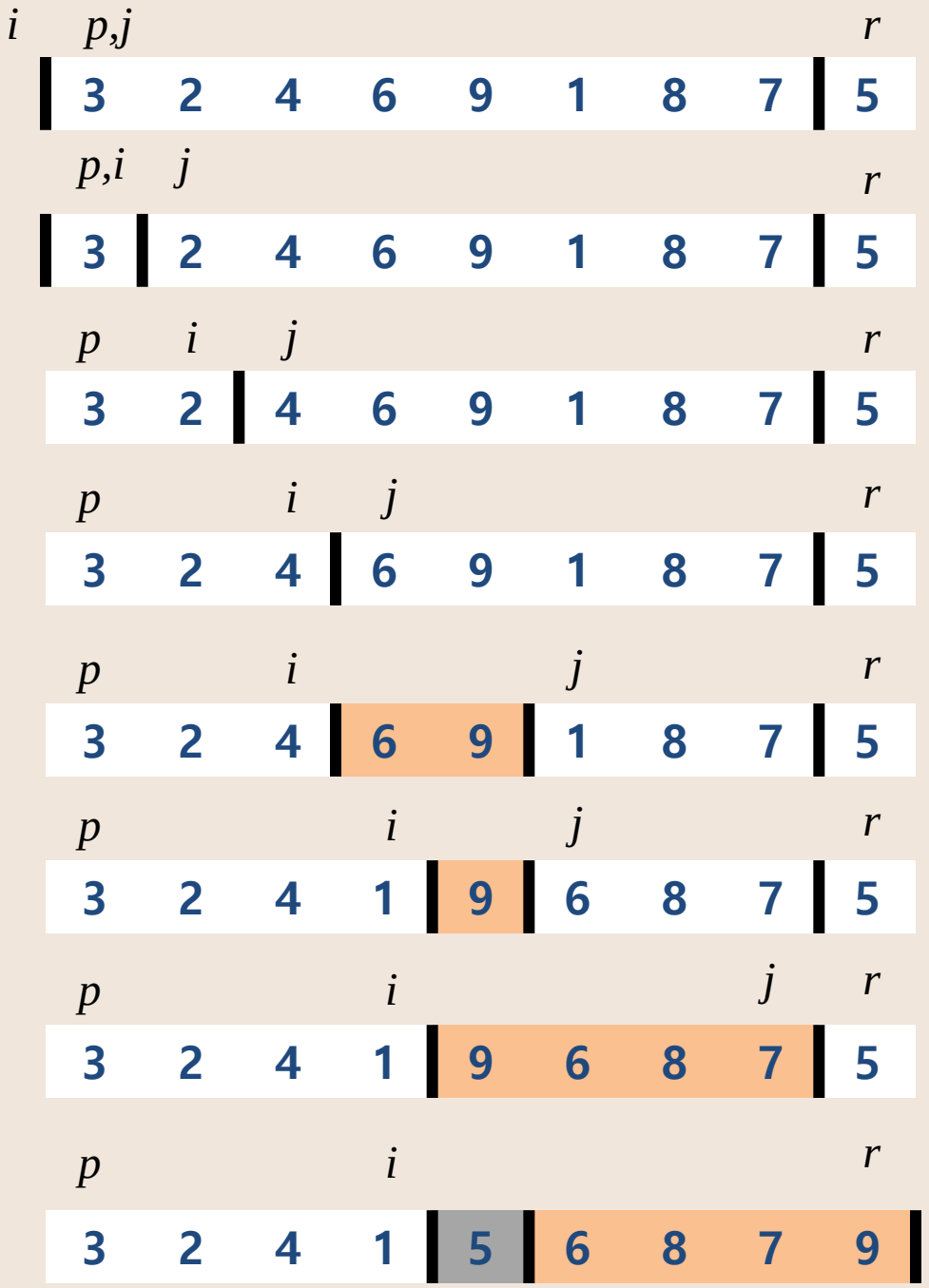
Lomuto partition 알고리즘

```
partition(A[], p, r)
    x ← A[r]
    i ← p – 1

    FOR j in p → r – 1
        IF A[j] ≤ x
                i++,  swap( A[i], A[j])

    swap(A[i+1], A[r])
    RETURN i + 1
```

```
partition(A[], p, r)
    x ← A[r]
    i ← p – 1
    FOR j in p → r – 1
        IF A[j] ≤ x
            i++, swap( A[i], A[j])
    swap(A[i+1], A[r])
    RETURN i + 1
```



문제 제시 : 병뚜껑 속의 숫자 게임

- 술래가 병뚜껑 속 숫자를 확인한 후, 다음 사람부터 숫자를 맞히기 시작한다. 술래는 Up 또는 Down을 통해 게임에 참여한 사람들이 병뚜껑 속 숫자에 점점 가까워질 수 있도록 힌트를 제시한다.
- 예시) 병뚜껑 속 숫자가 3일 경우

  첫 번째 사람이 14를 외쳤다면! ▶ 술래는 'DOWN' ! ▶ 두 번째 사람이 2를 외쳤다면! ▶ 술래는 'UP' ! ▶ 세 번째 사람이 4를 외쳤다면! ▶ 술래는 'DOWN' ! ▶ 결국 네 번째 사람이 병뚜껑 속 숫자인 3을 부를 수 밖에 없으므로 벌주 당첨!
- 이 게임은 숫자를 맞히는 게 아니라 피하는 게 핵심!
- 최대로 빨리 당첨 되려면 어떻게 하면 될까?

이진 검색(Binary Search)

- **자료의 가운데에 있는 항목의 키 값과 비교하여 다음 검색의 위치를 결정하고 검색을 계속 진행하는 방법**
  - 목적 키를 찾을 때까지 이진 검색을 순환적으로 반복 수행함으로써 검색 범위를 반으로 줄여가면서 보다 빠르게 검색을 수행함

- **이진 검색을 하기 위해서는 자료가 정렬된 상태여야 한다.**

검색 과정

1) 자료의 중앙에 있는 원소를 고른다.

2) 중앙 원소의 값과 찾고자 하는 목표 값을 비교한다.

3) 목표 값이 중앙 원소의 값보다 작으면 자료의 왼쪽 반에 대해서 새로 검색을 수행하고, 크다면 자료의 오른쪽 반에 대해서 새로 검색을 수행한다.

4) 찾고자 하는 값을 찾을 때까지 1)~3)의 과정을 반복한다.

예) 이진 검색으로 7을 찾는 경우

예) 이진 검색으로 20을 찾는 경우

알고리즘 : 반복구조

```
binarySearch( n, S[], k)
low ← 0
high ← n - 1

WHILE low <= high AND location = 0
        mid ← low + (high - low) / 2

        IF S[mid] == key
                RETURN mid
        ELIF S[mid] > key
                high ← mid – 1
        ELSE
                low ← mid + 1
RETURN -1
```

알고리즘 : 재귀구조

```
binarySearch(S[], low, high, key)
    IF low > high
        RETURN -1
    ELSE
        mid ← (low + high) / 2
        IF key == S[mid]
                RETURN mid
        ELIF key < a[mid]
                RETURN binarySearch(a[], low, mid - 1, key)
        ELSE
                RETURN binarySearch(a[], mid + 1, high, key)
```

분할 정복의 활용

- **병합 정렬은 외부 정렬의 기본이 되는 정렬 알고리즘이다. 또한, 멀티코어(Multi-Core) CPU 나 다수의 프로세서에서 정렬 알고리즘을 병렬화하기 위해 병합 정렬 알고리즘이 활용된다.**
- **퀵 정렬은 매우 큰 입력 데이터에 대해서 좋은 성능을 보이는 알고리즘이다.**
  - 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는데 접미어 배열(suffix array)와 함께 사용된다. 접미어 배열은 문자열에서 학습합니다.
- **최근접 점의 쌍(Closest Pair) 문제는 2차원 평면상의 n개의 점이 입력으로 주어질 때, 거리가 가장 가까운 한 쌍의 점을 찾는 문제이다.**
  - 컴퓨터 그래픽스, 컴퓨터 비전, 지리 정보 시스템, 항공 트래픽 제어, 마케팅(신규 가맹점 위치 선정등) 등의 분야

연습문제1

- **배열의 데이터를 퀵정렬하는 함수를 작성하고 테스트 해보시오.**

- **입력 예**
  - 11, 45, 23, 81, 28, 34
  - 11, 45, 22, 81, 23, 34, 99, 22, 17, 8
  - 1, 1, 1, 1, 1, 0, 0, 0, 0, 0

참고 문헌

[1] R. E. Neapolitan and K. Naimipour, Foundations of algorithms using C++ pseudocode. Jones and Bartlett, Sudbury, Mass., 2004.

→ Knapsack problem

[2] A. Levitin, Introduction to the design & analysis of algorithms. Pearson, Boston, 2012.

→ Greedy, Brute-force

[3] T. H. Cormen, Introduction to algorithms. The MIT Press, Cambridge, Masachusetts; London, 2009.

→ Powering a number, Fibonacci, Huffman coding, Activity-selection