분할 정복 알고리즘과 정렬

-합병 정렬&퀵 정렬-

HaRim Jung, Ph.D.
Visiting Professor / Senior Researcher
SKKU Institute for Convergence / Convergence Research Institute
Sungkyunkwan University, Korea

분할 정복 전략

❑ 분할 정복(divide and conquer) 전략이란?

- 주어진 문제(problem)를 (1) 부분 문제(sub-problem)들로 분할하고, (2) 각 부분 문제에 대한 해결책(부분 해: local solution)을 찾은 후, (3) 부분 해들을 통합하여 원래 문제의 해결책(final solution)을 찾는 전략 → 분할 정복 전략에 의해 설계된 알고리즘을 분할 정복 알고리즘이라고 함
  - **분할(divide)**: 주어진 문제를 (**동일한**) 부분 문제들로 (**재귀적으로**) 분할하고;
  - **정복(conquer)**: 부분 문제들의 부분 해를 찾은 후;
  - **통합(combine)**: (**필요하다면**) 부분 해들을 통합하는 하향식(top-down) 문제 해결 전략

❑ 분할 정복 알고리즘 분류

- 문제가 각 재귀마다 (1) 두 개의 부분 문제로 분할되고 (2) 부분 문제의 크기가 ½로 감소하는 알고리즘(**합병 정렬 알고리즘**)
- 문제가 각 재귀마다 (1) 두 개의 부분 문제로 분할되고 (2) 부분 문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘(**퀵 정렬 알고리즘**)
- 문제가 각 재귀마다 (1) 두 개의 부분 문제로 분할되고, (2) 그 중 한 개의 부분 문제는 고려할 필요가 없으며, (3) 부분 문제의 크기가 ½로 감소하는 알고리즘(이진 검색 알고리즘)
- 문제가 각 재귀마다 (1) 두 개의 부분 문제로 분할되고, (2) 그 중 한 개의 부분 문제는 고려할 필요가 없으며, (3) 부분 문제의 크기가 일정하지 않은 크기로 감소하는 알고리즘(선택 문제 알고리즘)

정렬

❑ 정렬(Sorting)의 개요

- 정렬은 탐색과 함께 컴퓨터가 **가장 많이 수행하는 연산** 중 하나로 순서 없이 나열된 항목들을 키(Key) 값을 기준으로 **오름차순** 혹은 내림차순으로 재배열시키는 연산(**본 강의에서는 오름차순 기준으로 정렬한다고 가정**)
- 정렬은 **탐색 성능을 향상**시키기 위해서도 필수적임(예: 이진 탐색)

❑ 정렬 알고리즘 분류

- **내부 정렬(Internal Sort)**: 정렬할 항목들의 집합 D가 주기억장치(메인 메모리)에 상주할 수 있을 경우, D를 주기억장치에서 정렬
  - **선택정렬**: 평균 수행 시간 $O(N^2)$
  - **삽입정렬**: 평균 수행 시간 $O(N^2)$
  - **셸정렬**: 정확한 수행 시간을 분석하기 어려움
  - **힙정렬**: 평균 수행 시간 O(NlogN)
  - **합병정렬**: 평균 수행 시간 O(NlogN)
  - **퀵정렬**: 평균 수행 시간 O(NlogN)
- **외부 정렬(External Sort):** 정렬하고자 하는 항목들의 집합 D의 사이즈가 커서 보조기억장치(디스크)에 저장하고 이를 작은 크기의 부분집합 D1, D2, ..., DN으로 나누어 주기억장치에서 정렬하고 합병

합병 정렬(1/9)

❑ 합병 정렬(merge sort)

- (i) 사이즈가 n인 하나의 리스트를 두 개의 **균등한** 크기의 서브리스트로 분할하고, (ii) 각 서브리스트를 재귀적인 방식으로 합병 정렬을 수행하여 정렬한 다음, (iii) 두 개의 정렬된 서브리스트를 합하여 전체가 정렬된 리스트가 되게 하는 분할 정복 정렬 알고리즘
- **문제**: n 개의 정수를 (오름차순으로) 정렬
- **입력**: 사이즈가 n인 리스트 L
- **출력**: 오름차순으로 **정렬된** n 사이즈 리스트
- **설계 전략**

1. 리스트 내 n 개의 원소 정렬 시 low, mid, high를 찾은 후, 두 서브리스트 L[low, mid]와 L[mid+1, high]로 분할
2. 모든 서브리스트의 사이즈가 1이 될 때까지 재귀적으로 분할
3. 분할 과정이 끝나면, n 개의 원소를 포함하는 하나의 리스트(정렬된 L)가 생성될 때까지 각각의 서브리스트를 정렬하면서 결합

  - 리스트의 사이즈가 1인 경우 해당 리스트는 정렬된 것으로 간주
  - **분할**: 리스트 L 내 n 개의 원소 정렬 시 low, mid, high를 찾은 후(여기서 low는 가장 왼쪽 인덱스, mid는 중간 인덱스, high는 가장 오른쪽 인덱스), 두 서브리스트 L[low, mid]와 L[mid+1, high]로 분할(**두 서브리스트가 부분 문제**)
  - **정복**: 각 서브리스트를 (**재귀적으로 합병 정렬을 이용하여**) 정렬(**두 서브리스트를 각각 정렬하는 것이 부분 해**)
  - **통합**: 두 서브리스트를 다시 하나의 정렬된 리스트로 합병(**두 개의 부분 해들을 통합**)

1	2	4	7	9	11	12

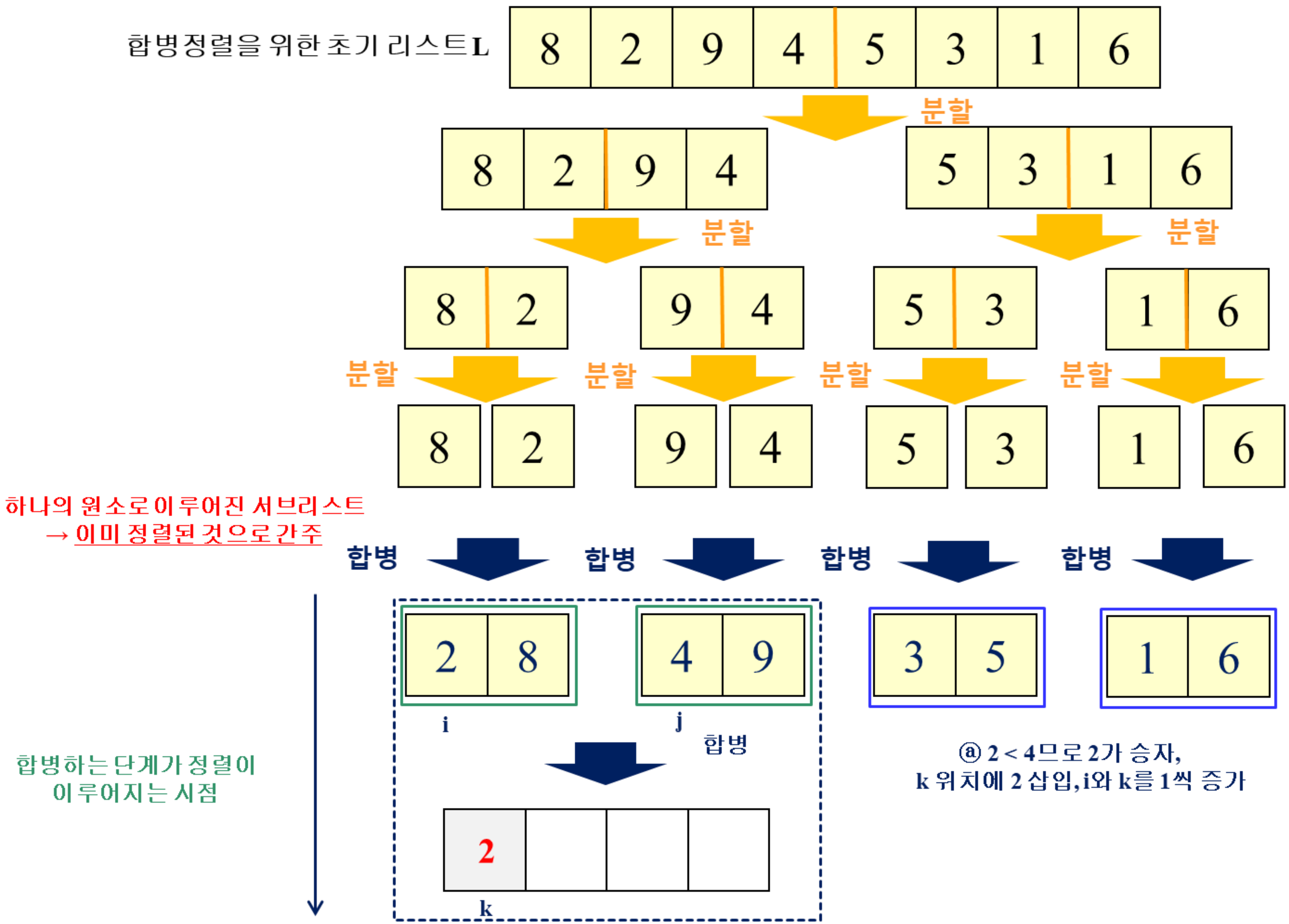
3	5	6	8	10	13

통합 **합병 정렬은 분할 과정보다 통합 과정이 주요 연산**

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

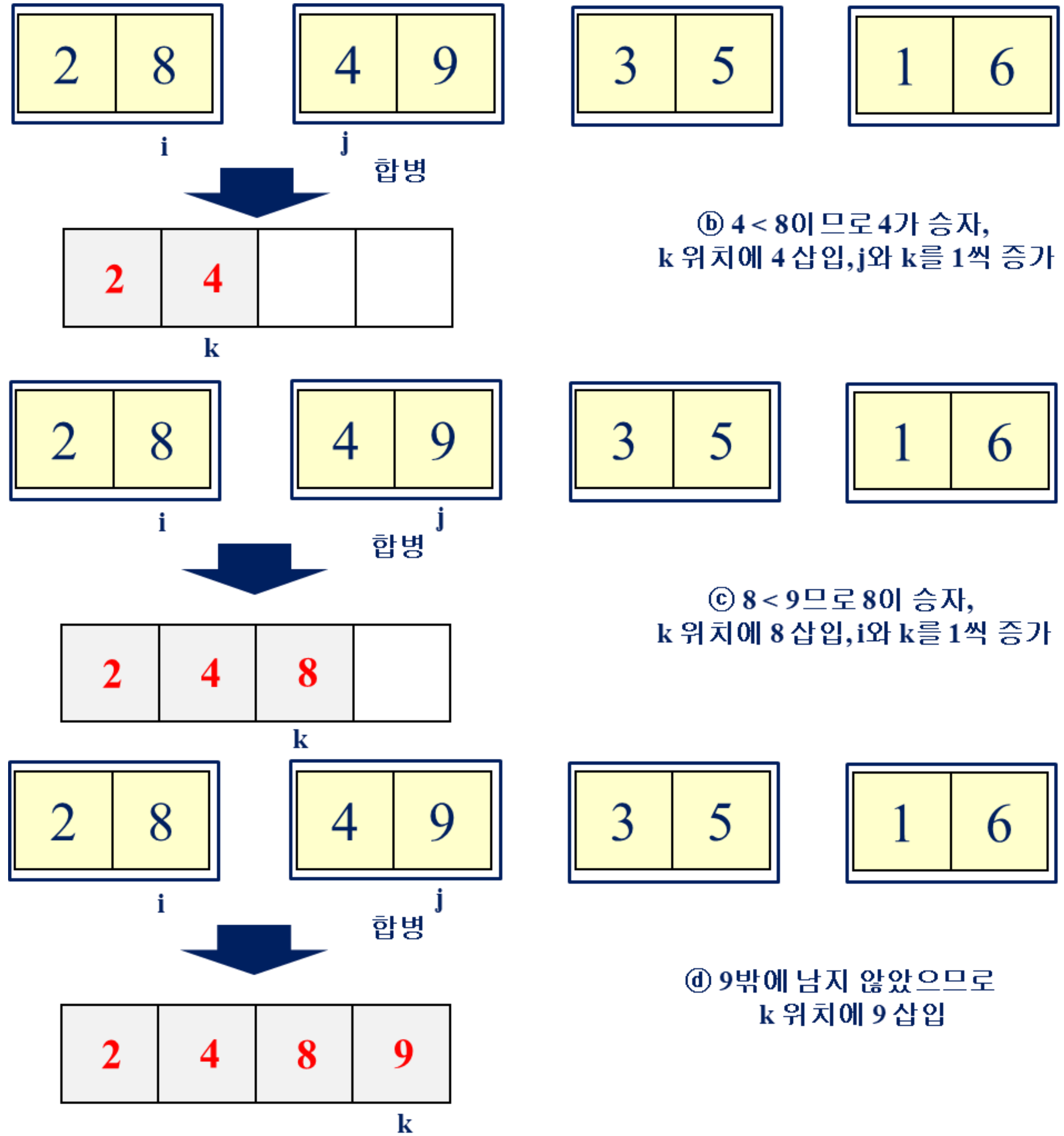
합병 정렬(2/9)

❑ 합병 정렬의 예



합병 정렬(3/9)

- 합병 정렬의 예 contd.



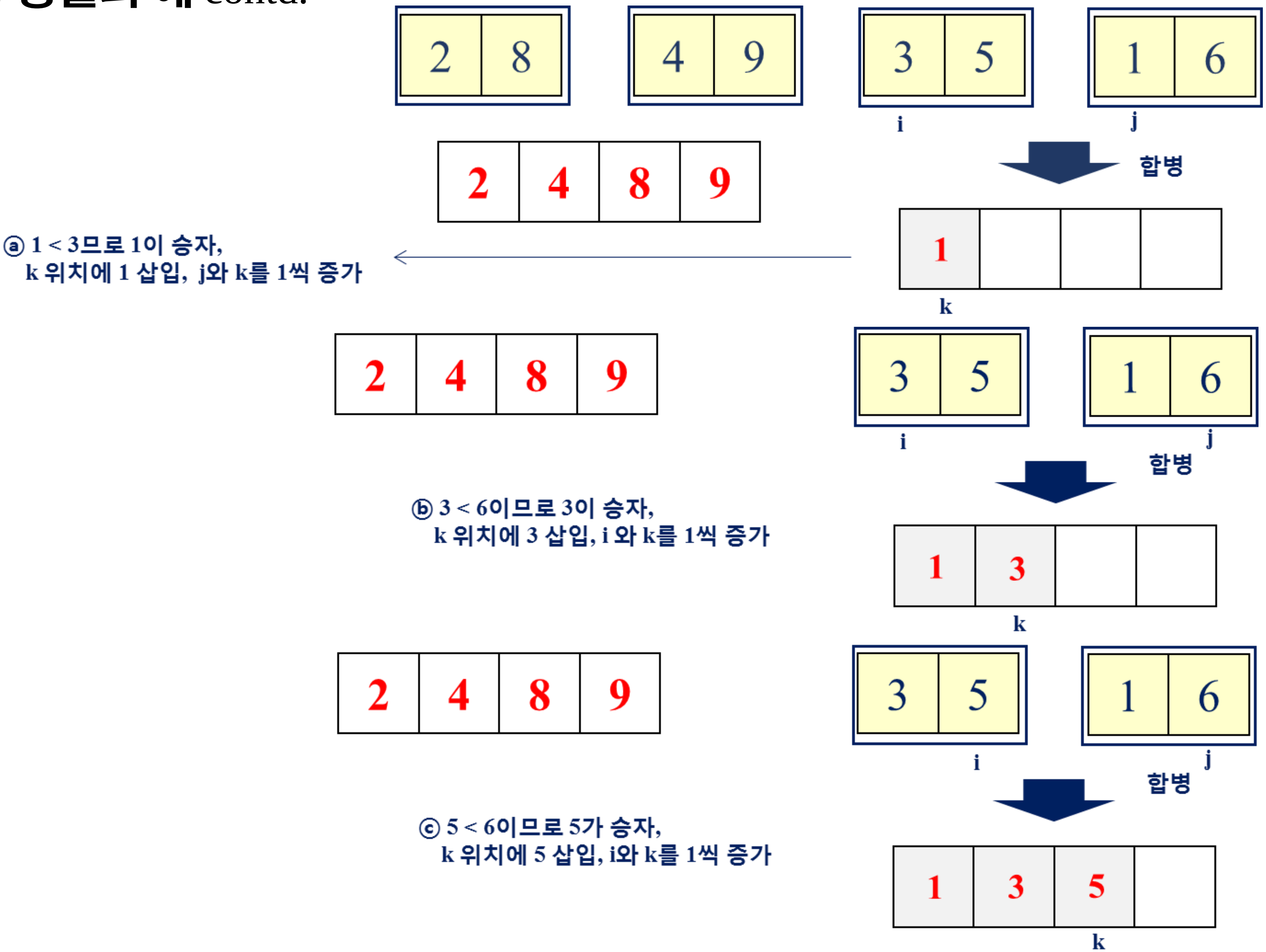
ⓑ 4 < 8이므로 4가 승자,
k 위치에 4 삽입, j와 k를 1씩 증가

ⓒ 8 < 9므로 8이 승자,
k 위치에 8 삽입, i와 k를 1씩 증가

ⓓ 9밖에 남지 않았으므로
k 위치에 9 삽입

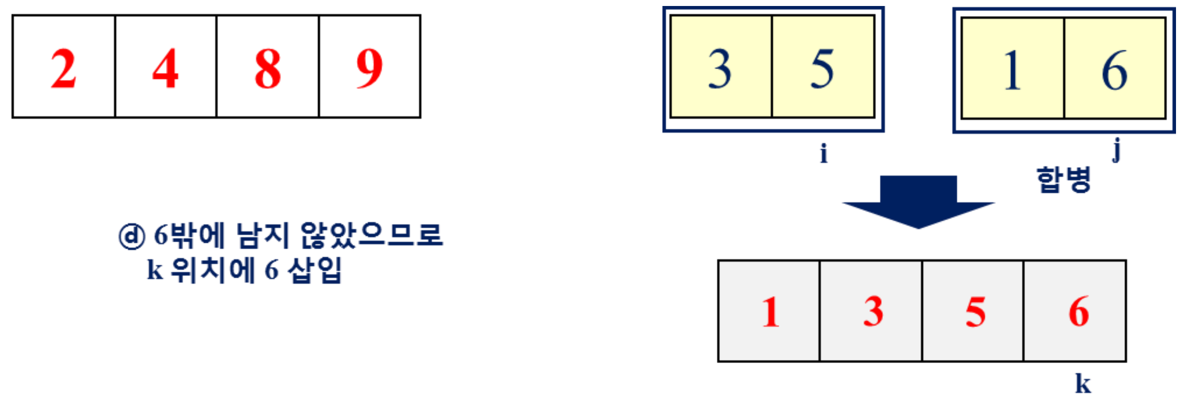
합병 정렬(4/9)

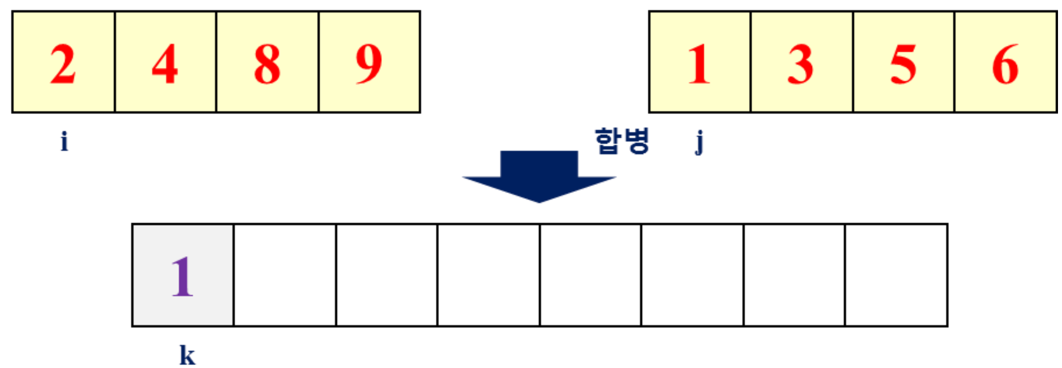
❑ **합병 정렬의 예** contd.



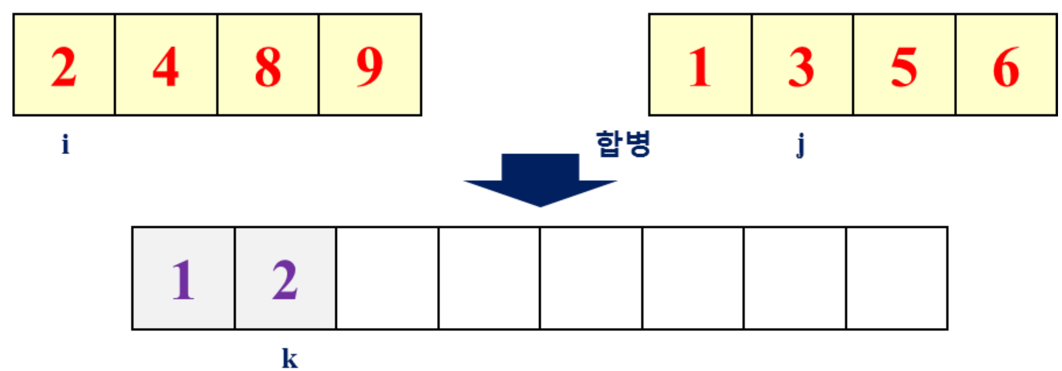
합병 정렬(5/9)

- 합병 정렬의 예 contd.





ⓐ 1 < 2므로 1이 승자,
k 위치에 1 삽입, j와 k를 1씩 증가

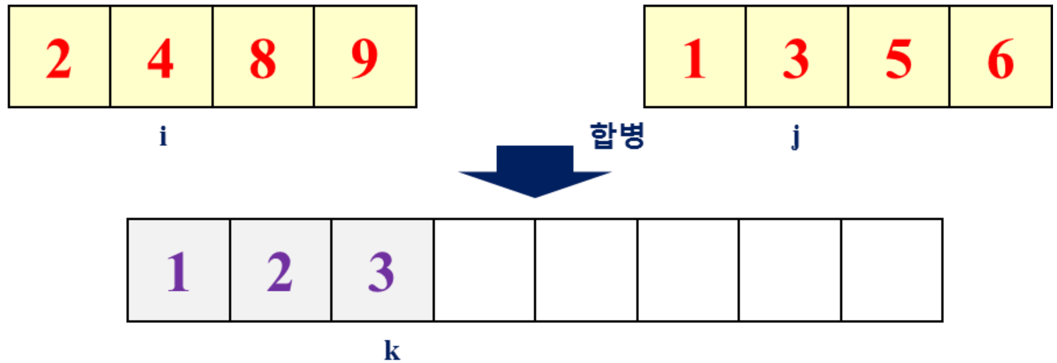


ⓑ 2 < 3이므로 2가 승자,
k 위치에 2 삽입, i와 k를 1씩 증가

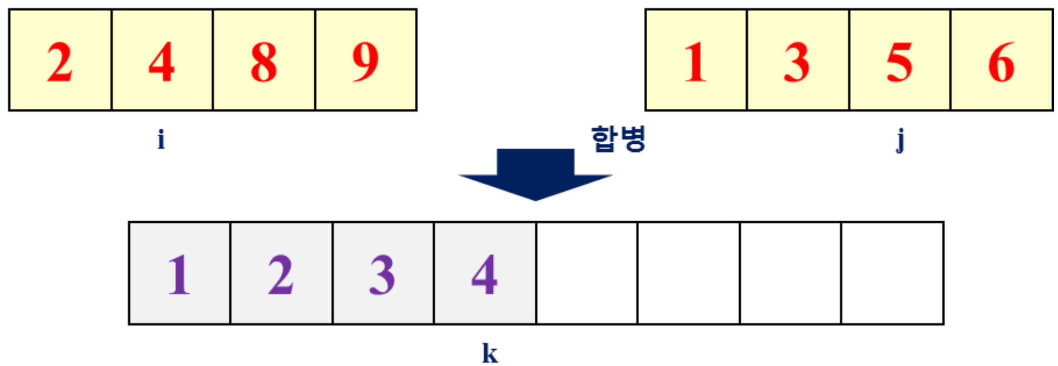
합병 정렬(6/9)

❑ 합병 정렬의 예 contd.

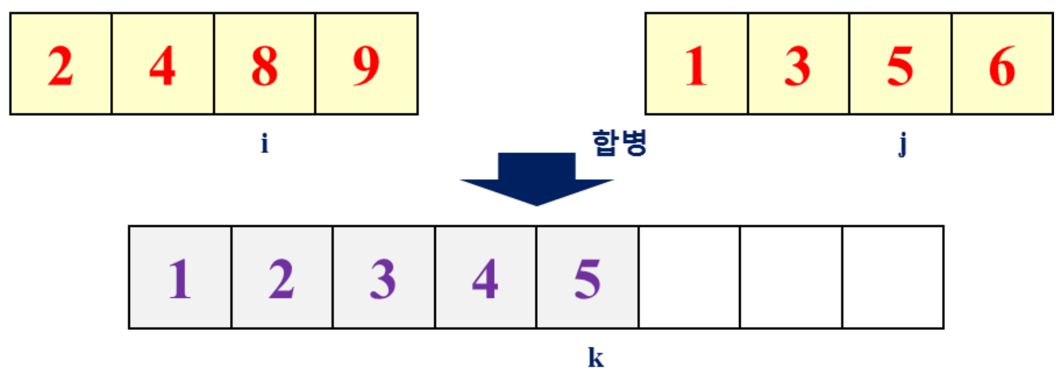
ⓒ 3 < 4이므로 3이 승자,
k 위치에 3 삽입, j와 k를 1씩 증가



ⓓ 4 < 5이므로 4가 승자,
k 위치에 4 삽입, i와 k를 1씩 증가



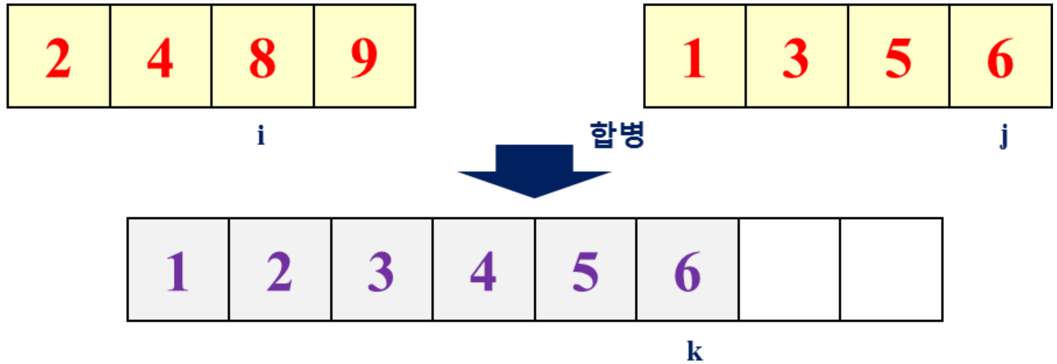
ⓔ 5 < 8이므로 5가 승자,
k 위치에 5 삽입, j와 k를 1씩 증가



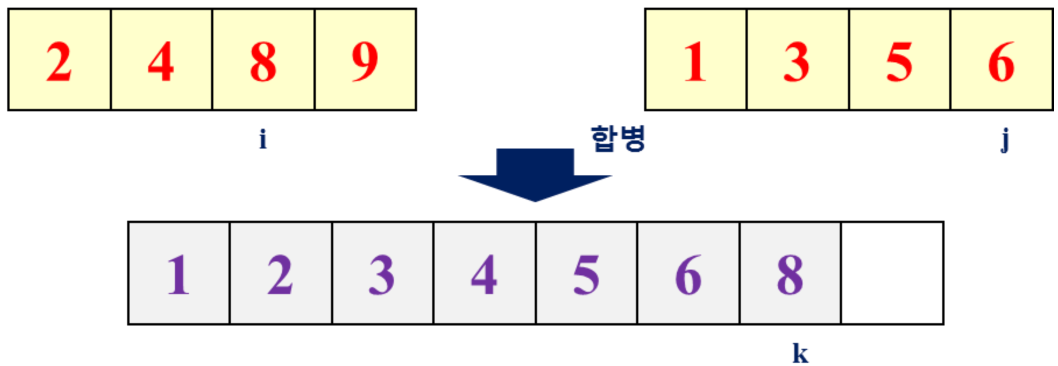
합병 정렬(7/9)

- 합병 정렬의 예 contd.

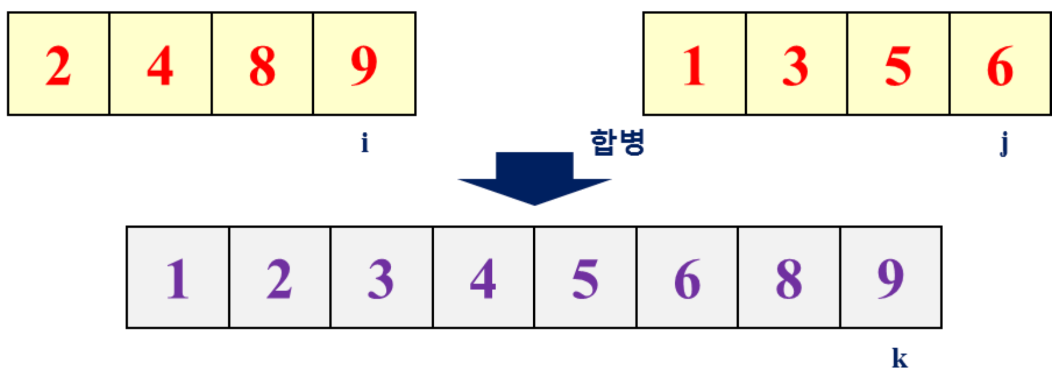
ⓕ 6 < 8이므로 6이 승자,
k 위치에 6 삽입, j와 k를 1씩 증가



ⓖ 왼쪽 서브리스트만 존재하므로
k 위치에 8 삽입, i와 k를 1씩 증가



ⓗ 왼쪽 서브리스트만 존재하므로
k 위치에 9 삽입



합병 정렬(8/9)

❑ 합병 정렬을 위한 함수 merge, merge_sort 정의

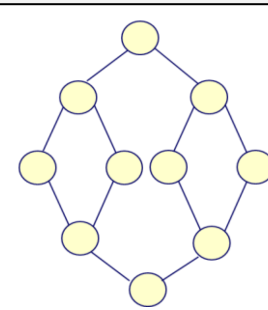
```python
1  def merge(lst, temp, low, mid, high):
2          i = low
3          j = mid+1
4          for k in range(low, high+1):
5              if i > mid:
6                  temp[k] = lst[j]
7                  j += 1
8              elif j > high:
9                  temp[k] = lst[i]
10                 i += 1
11             elif lst[j] < lst[i]:
12                 temp[k] = lst[j]
13                 j += 1
14             else:
15                 temp[k] = lst[i]
16                 i += 1
17         for k in range(low, high+1):
18             lst[k] = temp[k]
```

계속

```python
20 def merge_sort(lst, temp, low, high):
21     if high <= low:
22         return None
23     mid = low + (high - low) // 2
24     merge_sort(lst, temp, low, mid)
25     merge_sort(lst, temp, mid + 1, high)
26     merge(lst, temp, low, mid, high)
27 
28 lst = [54,88,77,26,93,17,49,10,17,77,11,31,22,44,17,20]
29 temp = [None] * len(lst)
30 print('정렬 전:\t', end='')
31 print(lst)
32 merge_sort(lst, temp, 0, len(lst)-1)
33 print('정렬 후:\t', end='')
34 print(lst)
```

- **라인 2-3:** 지역 변수 i와 j 선언 후 입력 값 low와 mid + 1을 참조시킴
- **라인 4:** 임시로 합병된 결과를 저장하기 위한 리스트 temp에 모든 원소가 저장될 때까지 반복
- **라인 5-7: if** lst의 앞부분 원소들을 전부 temp에 저장하였다면, 뒷부분 남은 원소들을 차례로 temp에 저장
- **라인 8-10: elif** lst의 뒷부분 원소들을 전부 temp에 저장하였다면, 앞부분 남은 원소들을 차례로 temp에 저장
- **라인 11-13: elif** lst[j] 값이 lst[i] 값보다 작으면, temp[k]에 lst[j]를 저장하고 j를 1 증가시킴
- **라인 14-16: else**(lst[i] 값이 lst[j] 값보다 작으면), temp[k]에 lst[i]를 저장하고 i를 1 증가시킴
- **라인 17-18:** temp에 모든 원소가 저장 되어 라인 4 for-루프를 탈출했다면, temp의 내용을 lst에 복사

- **라인 21-22: if** lst가 하나의 원소로 이루어졌다면, None값 반환 후 종료
- **라인 23:** lst의 중간 원소의 인덱스 계산
- **라인 24:** lst의 앞부분에 대해 merge_sort 호출 (재귀 호출)
- **라인 25:** lst의 뒷부분에 대해 merge_sort 호출 (재귀 호출)
- **라인 26:** merge 함수를 이용하여 합병 및 정렬

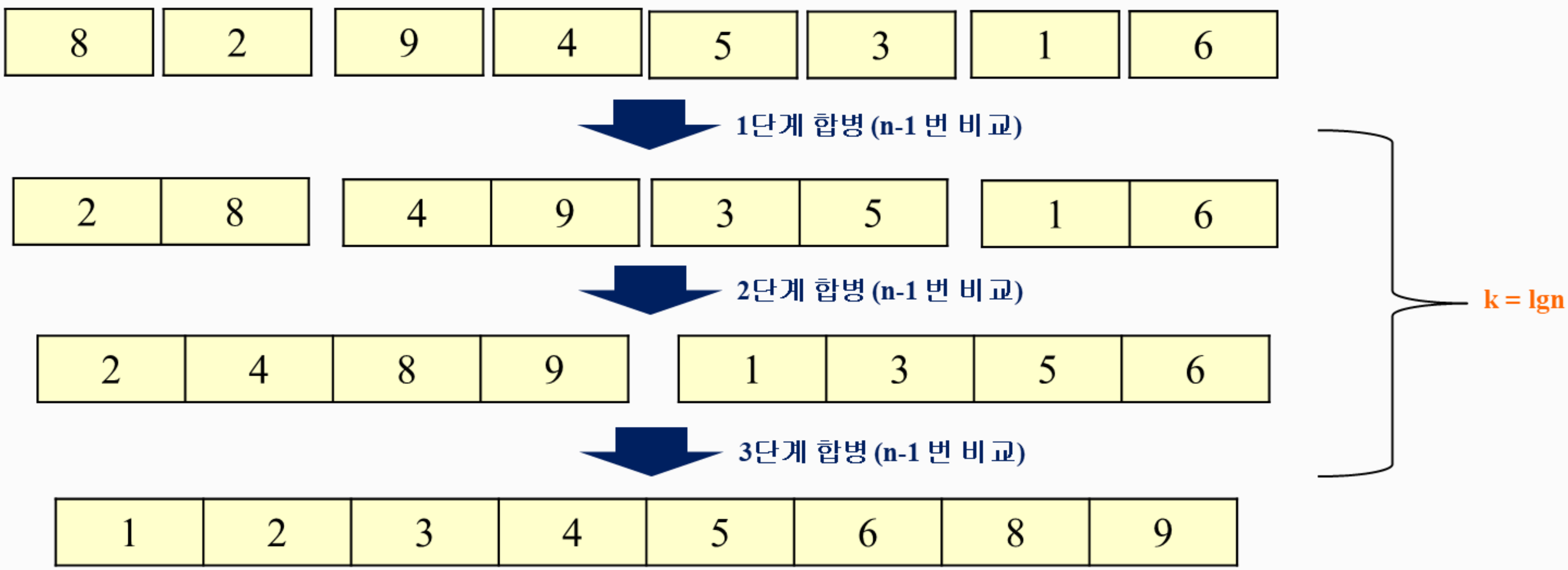


합병 정렬(9/9)

❑ 합병 정렬 수행 시간 분석 및 특징

-합병 정렬 수행 시간 분석-

- 리스트 사이즈(입력 사이즈) $n = 2^k$ 라고 가정
- 합병 정렬의 분할은 각 분할 단계마다 리스트의 중간 인덱스 계산과 2 번의 재귀 호출이므로 상수 시간 c 소요하며, lgn 단계가 존재하므로 clogn
- 단위 연산은 통합(합병)에서의 비교 연산(혹은 복사 연산)으로 각 통합 단계마다 (n – 1) 번 비교하며(혹은 n 번 복사하며), lgn 단계가 존재하므로 시간 복잡도는 O(nlogn)



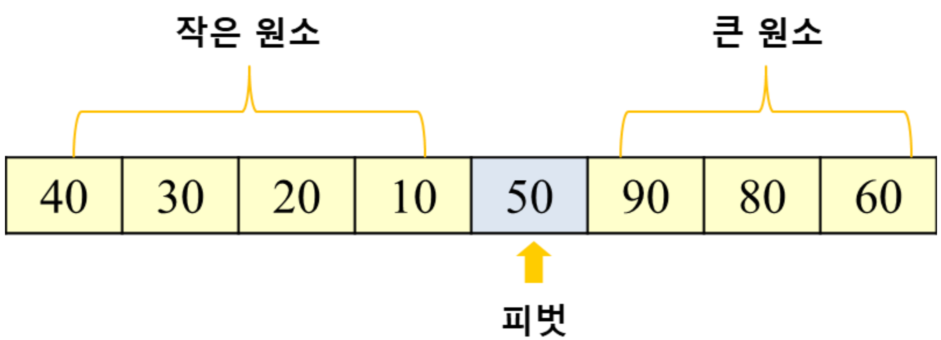
-합병 정렬의 특징-

- 입력에 민감하지 않음(input insensitive): 어떠한 입력에 대해서도 항상 O(NlogN) 수행 시간이 소요됨, i.e., 모든 경우의 분석이 가능
- 정렬 시 추가 메모리 공간이 필요함

퀵 정렬(1/15)

❑ 퀵 정렬(quick sort)

- 사이즈가 n인 하나의 리스트 내의 특정 원소인 피벗(pivot)을 기준으로 (i) 피벗보다 작은 원소들과 큰 원소들을 좌우로 분리(partition)하여 두 개의 **비균등한** 서브리스트를 구성하고, (ii) 각 서브리스트를 재귀적인 방식으로 퀵 정렬을 수행하여 정렬한 다음, (iii) 두 개의 정렬된 서브리스트를 합하여 전체가 정렬된 리스트가 되게 하는 분할 정복 정렬 알고리즘
- **문제**: n 개의 정수를 (오름차순으로) 정렬
- **입력**: 사이즈가 n인 리스트
- **출력**: 오름차순으로 **정렬된** n 사이즈 리스트
- **설계 전략**
  - 리스트의 사이즈가 1인 경우 해당 리스트는 정렬된 것으로 간주
  - **분할**: 리스트 내 n 개의 원소 정렬 시 피벗을 기준으로 피벗보다 작은 원소들을 피벗의 왼쪽으로 옮기고, 피벗보다 큰 요소들은 모두 피벗의 오른쪽으로 옮겨서 두 개의 서브리스트를 생성(추가 공간 X) → 이러한 분할 과정을 partition이라고 함(두 서브리스트가 부분 문제) **Note: 본 강의에서는 리스트의 마지막 원소를 피벗으로 선택**
  - **정복**: 각 서브리스트를 재귀적으로 퀵 정렬을 이용하여 정렬(**두 서브리스트를 각각 정렬하는 것이 부분 해**)
  - **통합**: 필요 없음



퀵 정렬은 분할 과정이 주요 연산

퀵 정렬(2/15)

❑ 퀵 정렬의 예

1. 퀵 정렬을 위한 초기 리스트 L

0	1	2	3	4	5	6	7
1	8	3	9	4	5	7	6

i j

피벗

2. L의 첫 원소(e.g., L[0] = 1)부터 피벗과 같거나 작은 값을 가지는 원소를 찾음(Note: 지역 변수 j를 이용하여 L 순회)

0	1	2	3	4	5	6	7
$\boxed{1}$	8	3	9	4	5	7	6

발견

i j

피벗

3. j가 가리키는 찾은 원소(e.g., 1)의 위치와 i가 가리키는 L의 첫 원소(e.g., 1)의 위치를 교환하고 i를 1 증가시킴

0	1	2	3	4	5	6	7
1	8	3	9	4	5	7	6

i j 교환

피벗

1	8	3	9	4	5	7	6

j

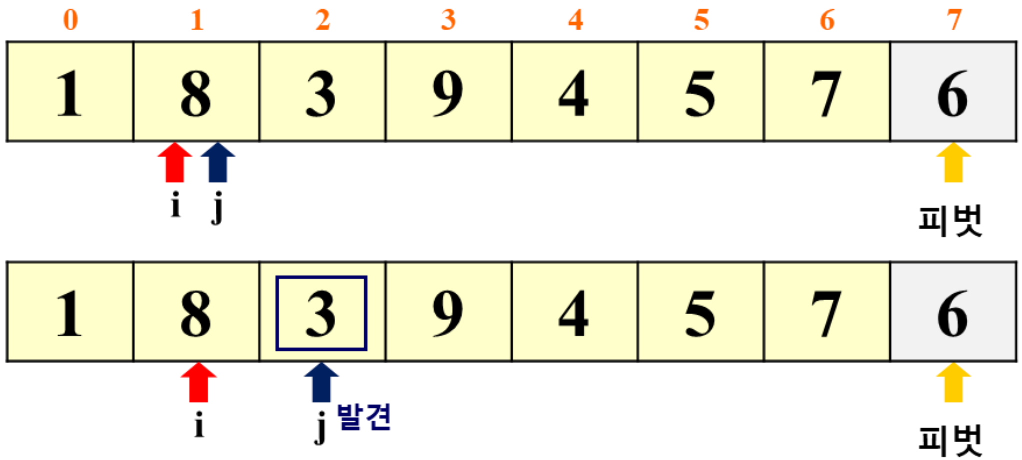
i i 증가

피벗

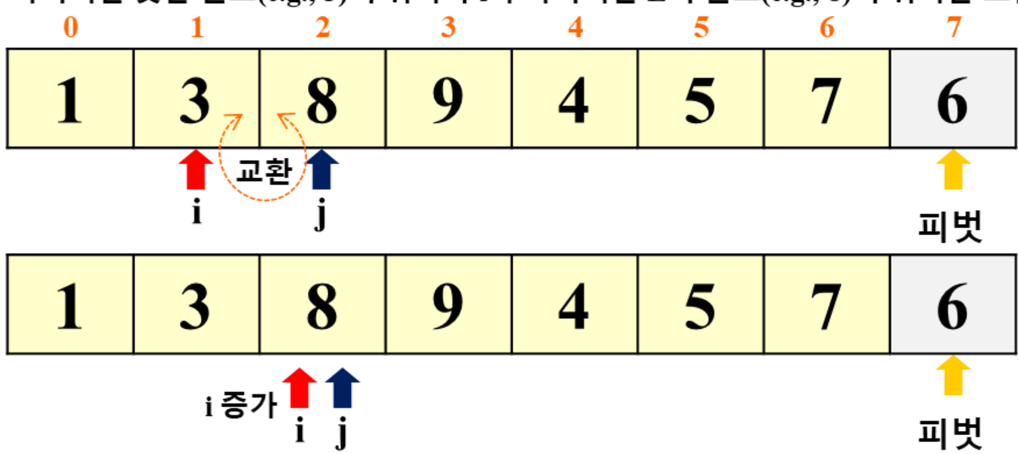
퀵 정렬(3/15)

❑ **퀵 정렬의 예** contd.

5. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L 순회



6. j가 가리키는 찾은 원소(e.g., 3)의 위치와 i가 가리키는 L의 원소(e.g., 8)의 위치를 교환하고 i를 1 증가시킴



퀵 정렬(4/15)

❑ 퀵 정렬의 예 contd.

7. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L 순회

0	1	2	3	4	5	6	7
1	3	8	9	4	5	7	6
		i	j				피벗

1	3	8	9	$\boxed{4}$	5	7	6
		i		j 발견			피벗

8. j가 가리키는 찾은 원소(e.g., 4)의 위치와 i가 가리키는 L의 원소(e.g., 8)의 위치를 교환하고 i를 1 증가시킴

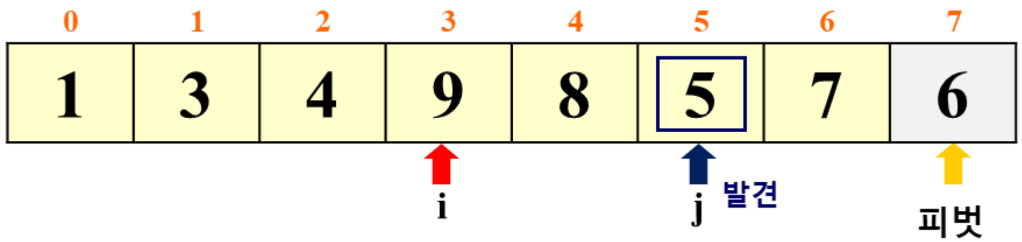
0	1	2	3	4	5	6	7
1	3	4	9	8	5	7	6
		i	교환	j			피벗

0	1	2	3	4	5	6	7
1	3	4	9	8	5	7	6
		i 증가	i	j			피벗

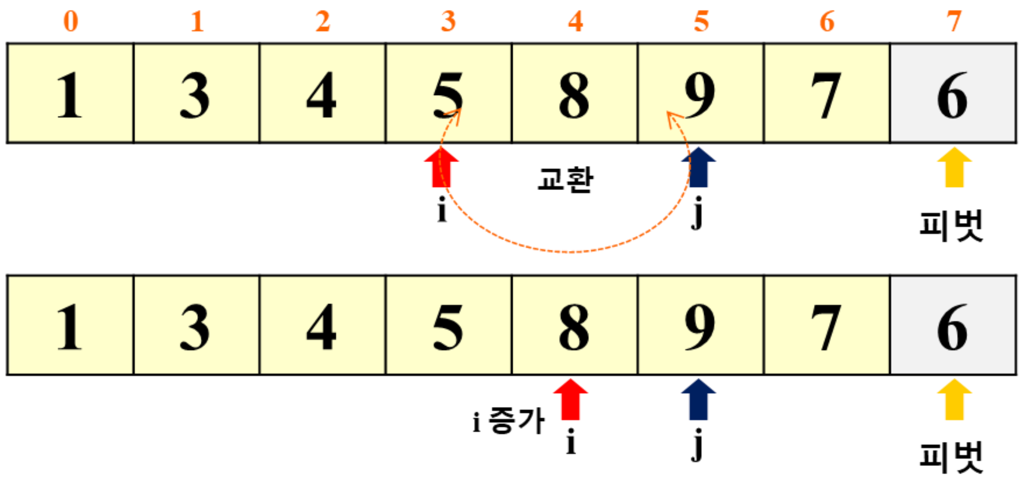
퀵 정렬(5/15)

- 퀵 정렬의 예 contd.

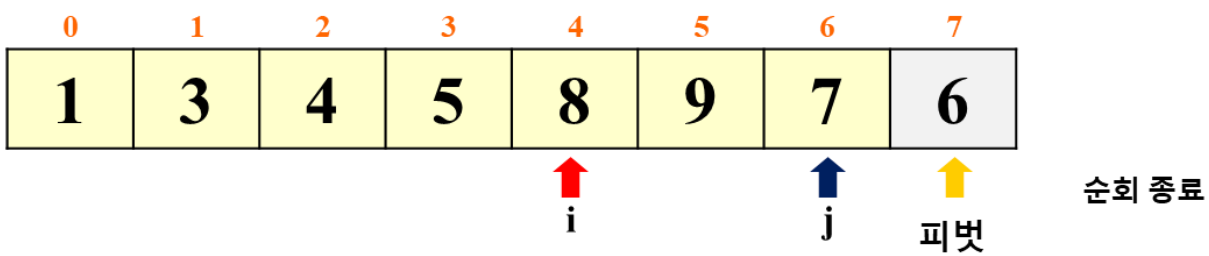
9. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L 순회



10. j가 가리키는 찾은 원소(e.g., 5)의 위치와 i가 가리키는 L의 원소(e.g., 9)의 위치를 교환하고 i를 1 증가시킴



11. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L 순회하다가 피벗 이전 위치에서 순회 종료



퀵 정렬(6/15)

❑ 퀵 정렬의 예 contd.

12. i가 가리키는 원소의 위치와 피벗의 위치 교환

0	1	2	3	4	5	6	7
1	3	4	5	6	9	7	8

교환

13. 피벗 6을 기준으로 두 개의 서브리스트(L1, L2)가 생성

0	1	2	3	4	5	6	7
1	3	4	5	6	9	7	8

14. L의 왼쪽 서브리스트 L1에 대해서 퀵 정렬 수행

0	1	2	3
1	3	4	5

i j

피벗

15. L1의 첫 원소(e.g., L1[0] = 1)부터 피벗과 같거나 작은 값을 가지는 원소를 찾음

0	1	2	3
1	3	4	5

발견

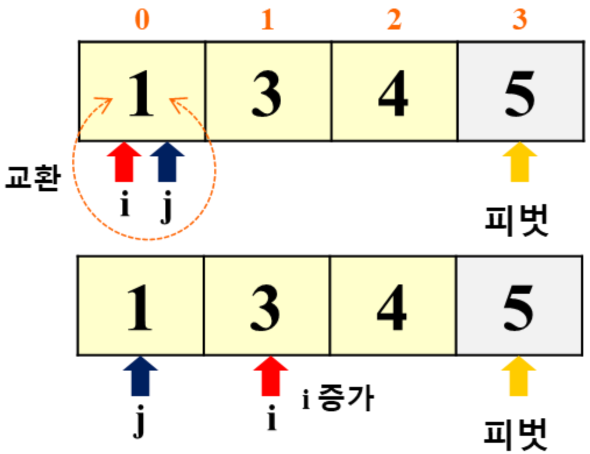
i j

피벗

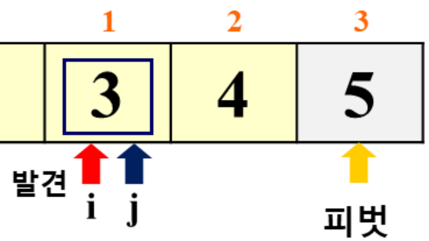
퀵 정렬(7/15)

❑ 퀵 정렬의 예 contd.

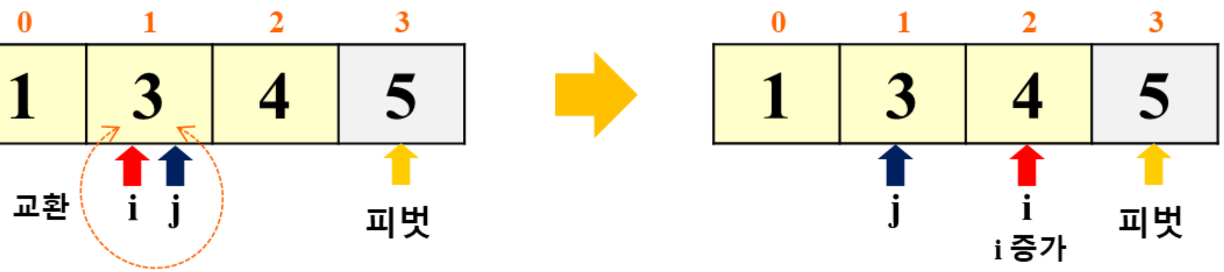
16. j가 가리키는 찾은 원소(e.g., 1)의 위치와 i가 가리키는 L1의 첫 원소(e.g., 1)의 위치를 교환하고 i를 1 증가시킴



17. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L1 순회

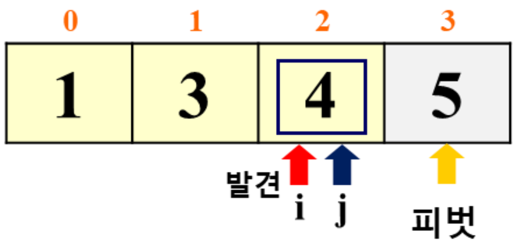


18. j가 가리키는 찾은 원소(e.g., 3)의 위치와 i가 가리키는 L1의 원소(e.g., 3)의 위치를 교환하고 i를 1 증가시킴

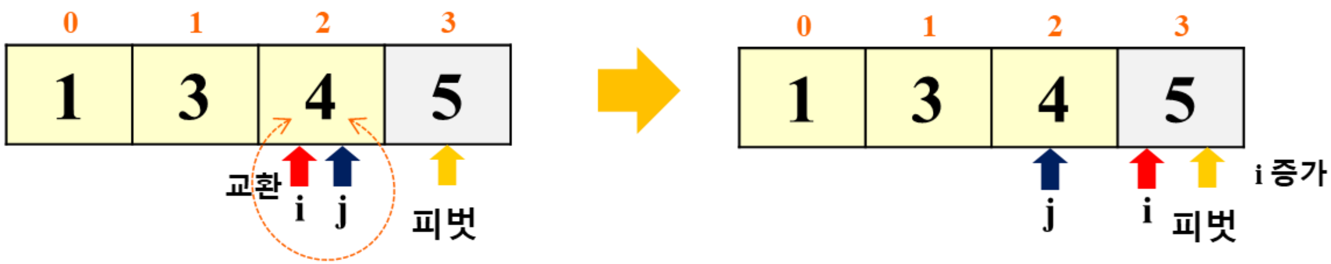


❑ 퀵 정렬의 예 contd.

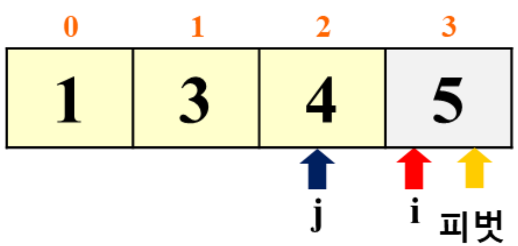
19. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L 1순회



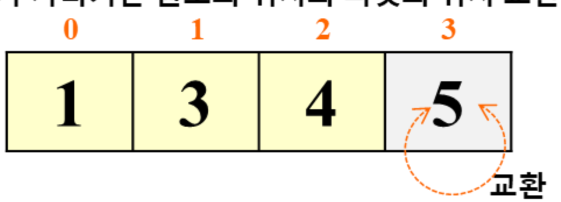
20. j가 가리키는 찾은 원소(e.g., 4)의 위치와 i가 가리키는 L1의 원소(e.g., 4)의 위치를 교환하고 i를 1 증가시킴



21. L1 순회하다가 피벗 이전 위치이므로 순회 종료



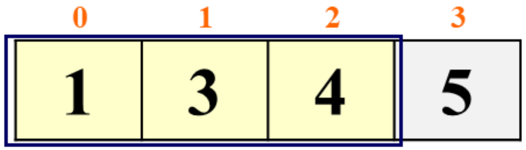
22. i가 가리키는 원소의 위치와 피벗의 위치 교환



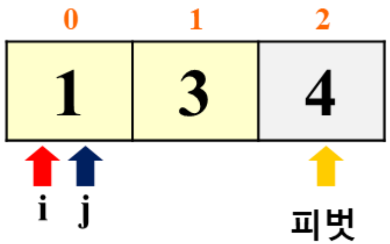
퀵 정렬(9/15)

- 퀵 정렬의 예 contd.

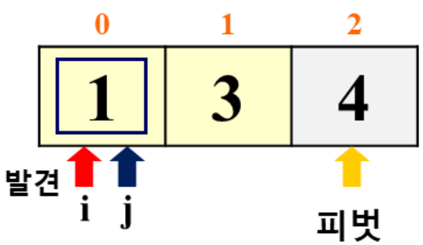
23. 피벗 5를 기준으로 L1의 서브리스트 L11이 생성



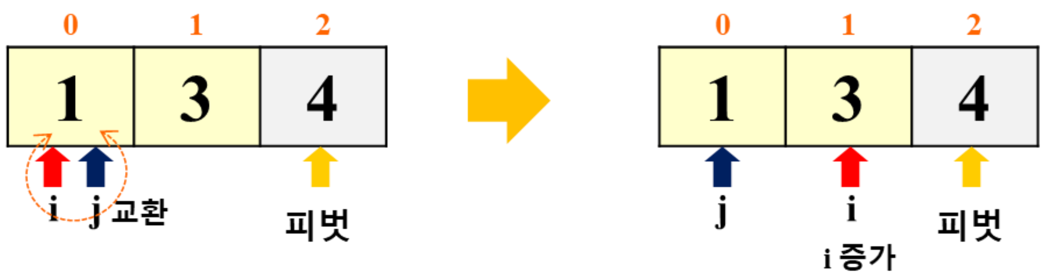
24. L11에 대해 퀵 정렬 수행



25. L11의 첫 원소(e.g., L11[0] = 1)부터 피벗과 같거나 작은 값을 가지는 원소를 찾음



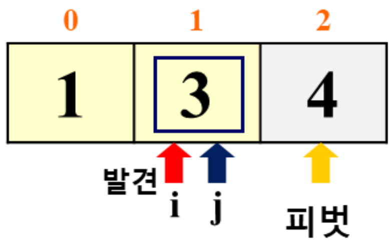
26. j가 가리키는 찾은 원소(e.g., 1)의 위치와 i가 가리키는 L11의 첫 원소(e.g., 1)의 위치를 교환하고 i를 1 증가시킴



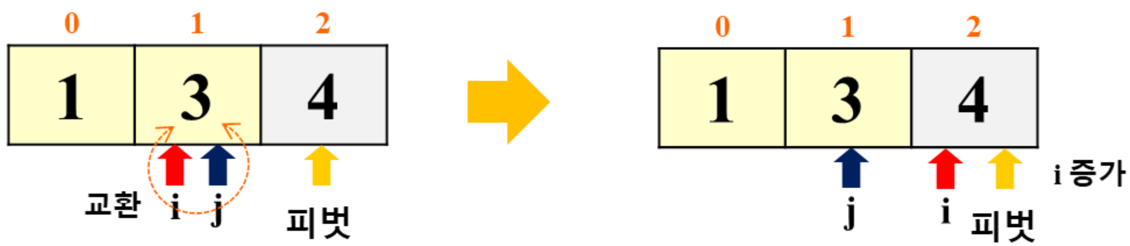
퀵 정렬(10/15)

❑ **퀵 정렬의 예** contd.

27. 피벗보다 작은 값을 가지는 원소를 찾을 때까지 지역 변수 j를 증가시키면서 L11순회

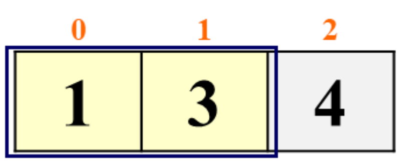


28. j가 가리키는 찾은 원소(e.g., 3)의 위치와 i가 가리키는 L11의 원소(e.g., 3)의 위치를 교환하고 i를 1 증가시킴



29. L11 순회하다가 피벗 이전 위치이므로 순회 종료 및 i가 가리키는 원소의 위치와 피벗의 위치 교환

30. 피벗 4를 기준으로 L11의 서브리스트 L111 생성

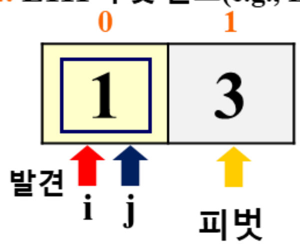


퀵 정렬(11/15)

- 퀵 정렬의 예 contd.

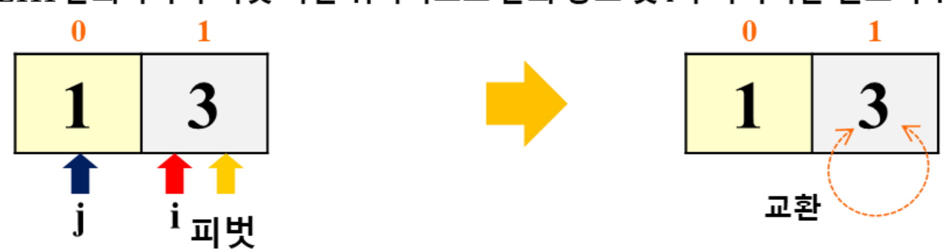
31. L111에 대해 퀵 정렬 수행

32. L111의 첫 원소(e.g., L111[0] = 1)부터 피벗과 같거나 작은 값을 가지는 원소를 찾음



33. j가 가리키는 찾은 원소(e.g., 1)의 위치와 i가 가리키는 L111의 첫 원소(e.g., 1)의 위치를 교환하고 i를 1 증가시킴

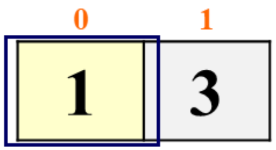
34. L111 순회하다가 피벗 이전 위치이므로 순회 종료 및 i가 가리키는 원소의 위치와 피벗의 위치 교환



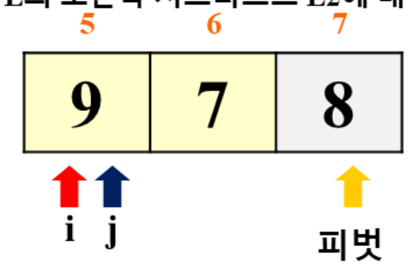
퀵 정렬(12/15)

❑ 퀵 정렬의 예 contd.

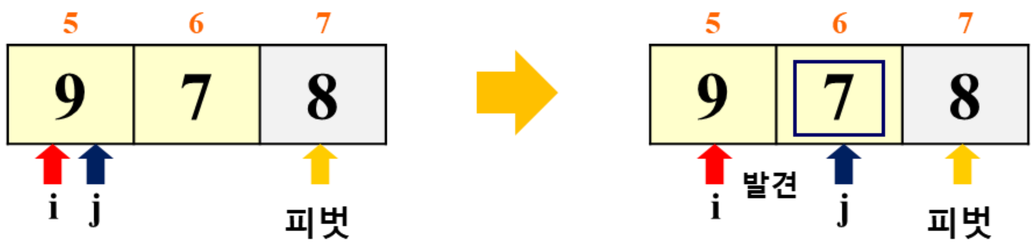
35. 피벗 3를 기준으로 L111의 서브리스트 L1111을 생성하지만 L1111의 시작과 끝이 같으므로 종료(return None) → L111 종료 → L11 종료 → L1 종료. 따라서 L의 왼쪽 서브리스트 L1은 이미 정렬된 상태



36. L의 오른쪽 서브리스트 L2에 대해서 퀵 정렬 수행



37. L2의 첫 원소(e.g., L2[5] = 9)부터 피벗과 같거나 작은 값을 가지는 원소를 찾음



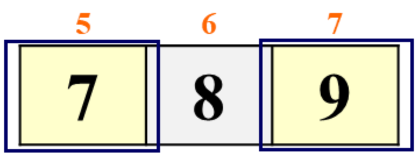
38. j가 가리키는 찾은 원소(e.g., 7)의 위치와 i가 가리키는 L2의 첫 원소(e.g., 9)의 위치를 교환하고 i를 1 증가시킴

퀵 정렬(13/15)

- 퀵 정렬의 예 contd.

39. L2 순회하다가 피벗 이전 위치이므로 순회 종료 및 i가 가리키는 원소의 위치와 피벗의 위치 교환

40. 피벗 8을 기준으로 L2의 서브리스트 L21과 L22가 생성되지만 L21과 L22는 시작과 끝이 같으므로 L21 종료 → L12 종료 → L2 종료 → 최종적으로 L 종료 및 정렬 완료



퀵 정렬(14/15)

❑ **퀵 정렬을 위한 함수** partition, qsort **정의**

```python
1   def partition(lst, low, high):
2       x = lst[high]
3       i = low
4       for j in range(low, high):
5           if lst[j] <= x:
6               lst[i], lst[j] = lst[j], lst[i]
7               i += 1
8       lst[i], lst[high] = lst[high], lst[i]
9       return i
10
11  def qsort(lst,low,high):
12      if low < high:
13          pi = partition(lst,low,high)
14          qsort(lst, low, pi-1)
15          qsort(lst, pi+1, high)
16
17  lst = [10, 80, 30, 90, 40, 50, 70, 60]
18  print('정렬 전:\t', end='')
19  print(lst)
20  qsort(lst, 0, len(lst)-1)
21  print('정렬 후:\t', end='')
22  print(lst)
```

i의 역할: for-루프 탈출 후 lst 내 피벗 위치 기억

i는 lst의 원소 중 피벗보다 값이 작은 원소의 개수만큼 증가함

- **라인 2-3:** 지역 변수 x와 i 선언 후 피벗 값 lst[high] (리스트의 마지막 원소)와 처음 위치 low를 참조시킴
- **라인 4:** 입력 받은 리스트 lst의 마지막 전까지(피벗 이전 위치까지) 순회(반복)
- **라인 5-7: if** lst[j]의 값이 피벗 값보다 작다면 lst[j]와 lst[i]의 위치 교환 후 i의 값 증가
- **라인 8-9:** 순회를 끝마쳤다면(j의 값이 high-1까지 증가시킨 후 탈출했다면), 피벗과 lst[i]의 위치 교환 후 피벗 위치(위치를 교환 했으므로 피벗은 인덱스 i 위치에 존재) 반환
- **라인 12: if** lst 사이즈가 1이 아니라면
  - **라인 13**: partition 함수를 호출하여 두 개의 서브리스트로 분할(NOTE: partition 함수는 피벗의 인덱스를 반환)
  - **라인 14**: 왼쪽 서브리스트에 대해 퀵 정렬 수행(qsort 재귀 호출)
  - **라인 15**: 오른쪽 서브리스트에 대해 퀵 정렬 수행(qsort 재귀 호출)

a = [50, 20, 70, 10, 70, 20, 50, 70, 70, 10]에 대해 partition(a, 0, 9)이 수행된 결과를 보이시오.

b = [50, 60, 80, 90, 30, 40, 70, 10, 20]에 대해 partition(b, 0, 8)이 수행된 결과를 보이시오.

퀵 정렬(15/15)

❑ 퀵 정렬의 수행 시간 분석 및 특징

-퀵 정렬 수행 시간 분석-

단위 연산은 분할 시 리스트의 원소와 피벗의 비교

- 입력 크기 $n = 2^k$ 라고 가정
- 최선의 경우와 평균의 경우: O(nlogn)
  - 최선의 경우 합병 정렬과 마찬가지로 k(= lg n)만큼 분할하며,각 분할마다 약 n 번의 원소를 비교
  - 평균의 경우 피벗의 인덱스가 리스트 전체에 걸쳐서 균등하게 임의로 결정된다는 가정 하에 O(nlogn)
- 최악의 경우: $O(n^2)$
  - 입력이 이미 정렬되어 있거나 역순으로 정렬되어 있다면 n번 분할하며, 각 분할 단계마다 $n-1, n-2, \ldots 1$ 번의 원소를 비교하므로 $\frac{n(n-1)}{2}$

-퀵 정렬의 특징-

- 평균 시간 복잡도가 O(nlogn)인 다른 정렬 알고리즘들보다 빠름(상수적으로 빠르다는 의미)
- 합병 정렬과 다르게 정렬 시 추가 메모리 공간이 필요하지 않음

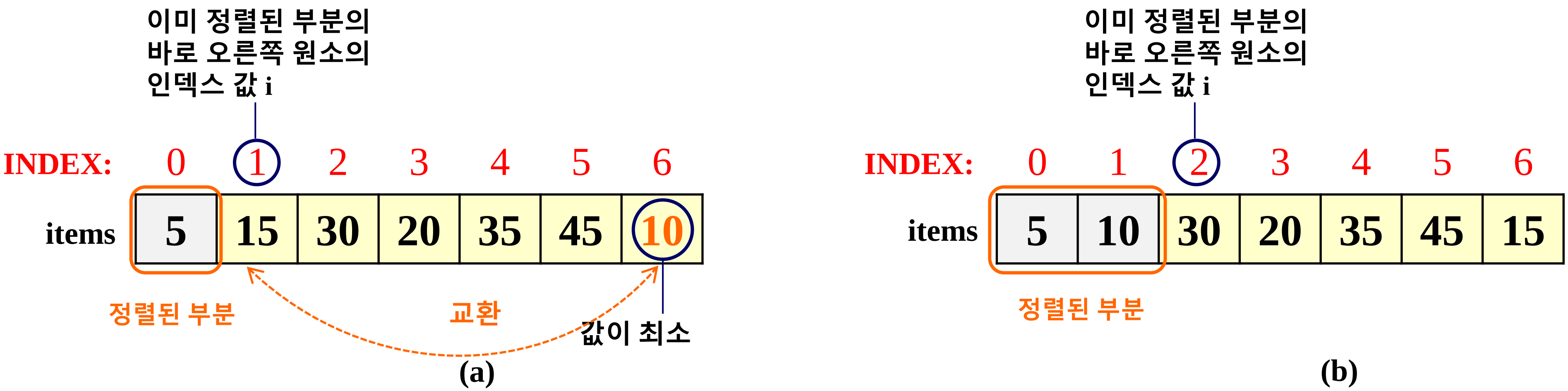
❑ 퀵 정렬과 합병 정렬과의 차이점

- 합병 정렬은 아무 연산 없이 균등한 두 부분으로 분할하는 반면에, 퀵 정렬은 분할할 때부터 기준 원소(피벗)를 중심으로 이보다 작은 것은 왼편, 큰 것은 오른편에 위치시킴
  - 즉, **정복 과정(정렬 과정)이 분할 과정에서 함께 진행**
- 각 부분 정렬이 끝난 후, 합병 정렬은 통합 과정이라는 후처리 작업이 필요하나, 퀵 정렬은 필요로 하지 않음

참고: 선택 정렬 (1/4)

❑ 선택 정렬(Selection Sort)

- (Python) 리스트에서 **아직 정렬되지 않은 부분**의 원소(항목)들 중 최소인 원소를 '**선택**'하여 **이미 정렬된 부분**의 바로 오른쪽 원소와 교환(자리 이동)하는 정렬 알고리즘



- 위의 그림(a)에서 Python 리스트 items의 왼쪽 부분은 이미 정렬되어 있고 나머지 부분은 아직 정렬되지 않은 상태임
- 그림(a)에서 이미 정렬된 왼쪽 부분의 값들은 아직 정렬되지 않은 오른쪽 부분의 어떤 값보다 크지 않음
  - 선택 정렬은 항상 오른쪽의 정렬되지 않은 부분에서 값이 최소인 원소를 찾아 왼쪽의 정렬된 부분의 바로 오른쪽 원소(e.g., 그림(a)에서 인덱스 i의 위치에 존재하는 원소)와 위치를 교환하기 때문
- 그림(a)에서 값이 최소인 10과 items[1](i = 1)의 위치를 교환 후, 그림(b)와 같이 i를 1 증가시키는 방식으로 반복적 비교·위치 교환 수행

참고: 선택 정렬 (2/4)

❑ **선택 정렬**(Selection Sort) contd.

-선택 정렬 과정-

1. N개의 항목으로 이루어진 Python 리스트 items에서 가장 작은 원소를 찾음
2. 가장 작은 원소가 items[j]라면 items[j]와 items[0](i = 0)의 위치를 서로 교환
3. i를 1 증가시킨 후, items[0]을 제외한 나머지 원소들(items[1], items[2], …, items[N-2]) 중 가장 작은 원소 items[j]를 찾은 후, items[j]와 items[1](i = 1)의 위치를 서로 교환

→ 위와 같은 과정을 반복 실행

- **선택 정렬을 위한 함수 selection_sort 정의**

Python 리스트 items의 마지막 인덱스 전까지만 i를 증가시키면 됨

Python 리스트 items의 마지막 인덱스까지 j를 증가시켜야 함

```python
def selection_sort(items):
    for i in range(0, len(items)-1):
        minimum = i
        for j in range(i, len(items)):
            if items[minimum] > items[j]:
                minimum = j
        items[i], items[minimum] = items[minimum], items[i]

items = [40,70,60,30,10,50]
print('정렬 전: ', end='')
print(items)
selection_sort(items)
print('정렬 후: ', end='')
print(items)
```

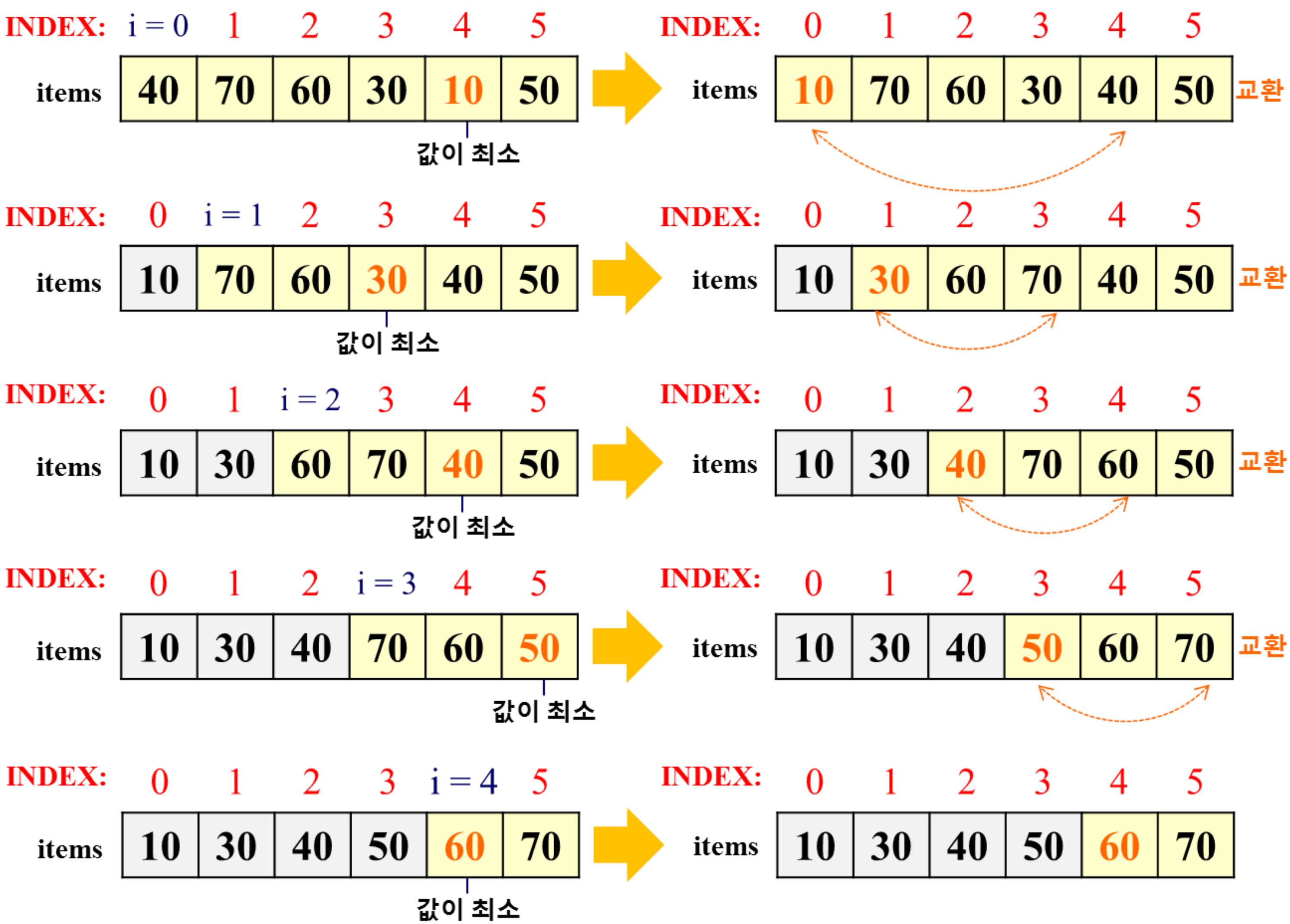
```
정렬 전: [40, 70, 60, 30, 10, 50]
정렬 후: [10, 30, 40, 50, 60, 70]
```

결과

참고: 선택 정렬 (3/4)

❑ **선택 정렬**(Selection Sort) contd.

예: items = [40, 70, 60, 30, 10, 50]에 대해 선택 정렬 수행 과정



참고: 선택 정렬 (4/4)

❑ 선택 정렬(Selection Sort) contd.

-선택 정렬 수행 시간 분석-

- 선택 정렬의 기본 연산 단위는 **Phase**로서 하나의 Phase마다 정렬되지 않은 부분에서 가장 작은 원소을 선택하여 교환
- 첫 번째 Phase에서는 N개의 원소들 중에서 가장 작은 값을 찾기 위해 N-1번 비교
- 두 번째 Phase에서는 N-1개의 원소들 중에서 가장 작은 값을 찾기 위해서 N-2번 비교
- 같은 방법으로 마지막 Phase에서 두 개의 원소를 1번 비교하여 가장 작은 값을 찾음
- 따라서 원소들의 총 비교 횟수는:

$$(N-1) + (N-2) + (N-3) + \ldots + 2 + 1 = \frac{N(N-1)}{2} = O(N^2)$$

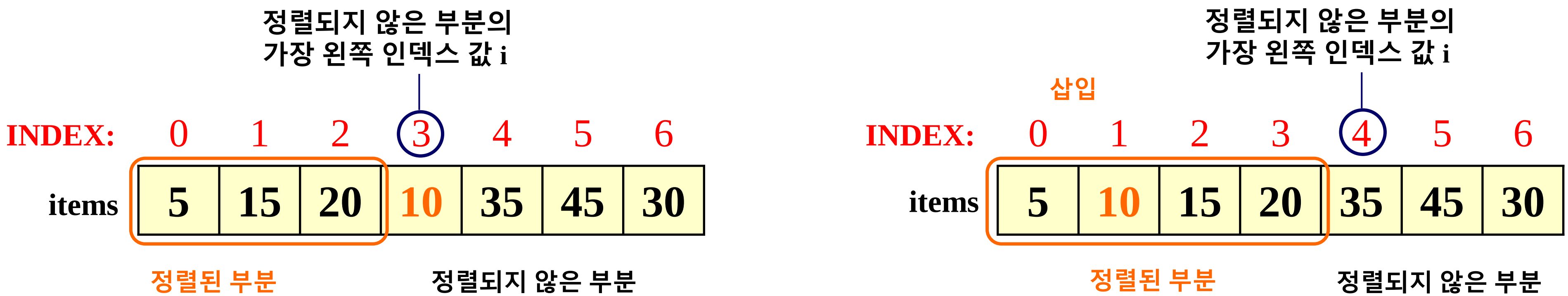
-선택 정렬의 특징-

- 입력에 민감하지 않음(Input Insensitive): 어떠한 입력에 대해서도 항상 $O(N^2)$ 수행 시간이 소요됨
- 값이 최소인 원소를 찾은 후 **원소의 위치를 교환하는 최대 횟수가 N-1번**으로 정렬 알고리즘들 중에서 (최악의 경우) 가장 적은 위치 교환 횟수임

참고: 삽입 정렬 (1/5)

❑ 삽입 정렬(Insertion Sort)

- (Python) 리스트가 정렬된 부분과 정렬되지 않은 부분으로 나뉘며, **정렬되지 않은 부분의 가장 왼쪽 원소**를 **정렬된 부분 중 적절한 위치에 '삽입'**하는 방식의 정렬 알고리즘



(a) 삽입 수행 전 **(b) 삽입 수행 후**

– 위의 그림(a)에서 Python 리스트 items의 왼쪽 부분은 이미 정렬되어 있고 나머지 부분은 아직 정렬되지 않은 상태임

– 정렬 안된 부분의 가장 왼쪽 원소(현재 원소)를 정렬된 부분의 원소들을 비교하며 그림(b)와 같이 현재 원소 삽입

– 현재 원소 삽입 후 정렬된 부분의 원소 수를 1 증가시키고, 정렬되지 않은 부분의 원소 수를 1 감소시킴

참고: 삽입 정렬 (2/5)

❑ 삽입 정렬(Insertion Sort) contd.

-삽입 정렬 과정-

1. (Python) 리스트 items 내 N개의 원소 정렬 시 **첫 원소 items[0]은 이미 정렬되었다고 취급**
2. 다음 원소 items[1]을 정렬된 부분 items[0]과 비교하여 적절한 위치에 삽입
3. 다음 원소 items[2]를 정렬된 부분 items[0], items[1]과 비교하여 적절한 위치에 삽입

→ 위와 같은 과정을 반복 실행

- **삽입 정렬을 위한 함수 insertion_sort 정의**

```python
def insertion_sort(items):
    for i in range(1, len(items)):
        for j in range(i, 0, -1):
            if items[j-1] > items[j]:
                items[j], items[j-1] = items[j-1], items[j]

items = [40, 70, 60, 30, 10, 50]
print('정렬 전: ', end='')
print(items)
insertion_sort(items)
print('정렬 후: ', end='')
print(items)
```

Python 리스트 items의 첫 번째 원소 items[0]은 이미 정렬된 부분이므로 i = 1부터 시작

뒤에서 앞으로 비교해 나간다는 것에 유의

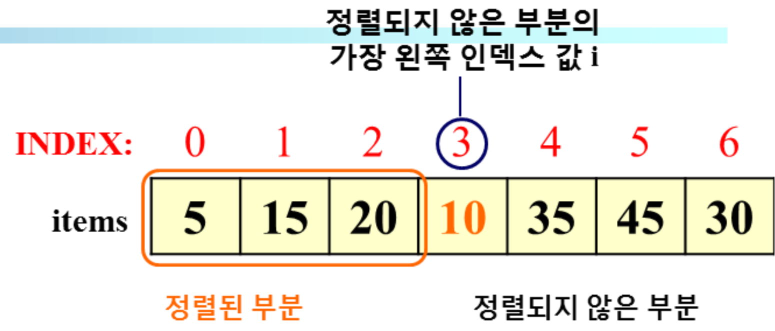
```
정렬 전: [40, 70, 60, 30, 10, 50]
정렬 후: [10, 30, 40, 50, 60, 70]
```

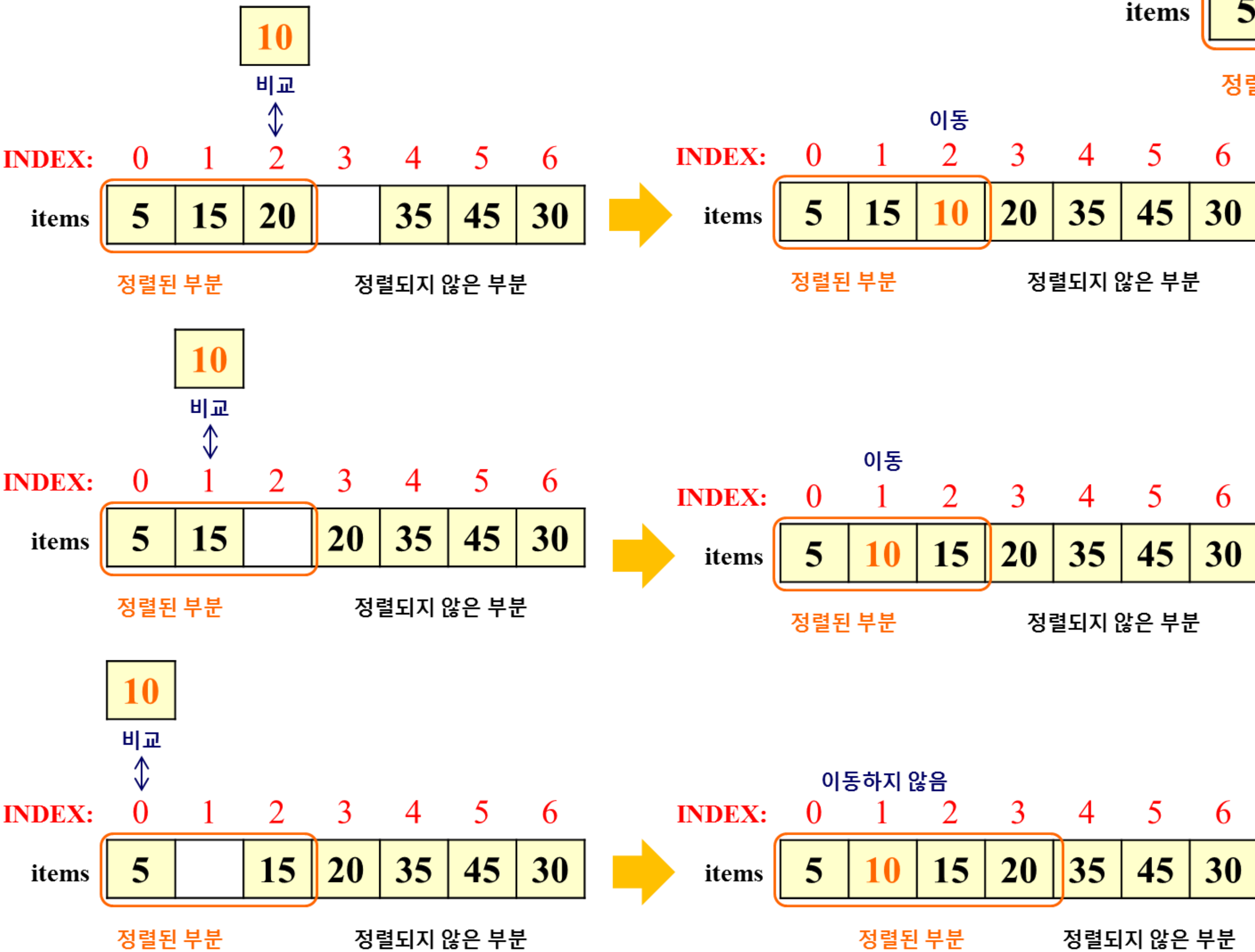
결과

참고: 삽입 정렬 (3/5)

- 삽입 정렬(Insertion Sort) contd.

  예: 정렬 안된 부분의 가장 왼쪽 원소(현재 원소) 10을 정렬된 부분에 삽입하는 과정





참고: 삽입 정렬 (4/5)

❑ **삽입 정렬**(Insertion Sort) contd.

예: items = [40, 70, 60, 30, 10, 50]에 대해 선택 정렬 수행 과정

INDEX:	0	i = 1	2	3	4	5
items	40	70	60	30	10	50

INDEX:	0	1	2	3	4	5
items	40	70	60	30	10	50

INDEX:	0	1	i = 2	3	4	5
items	40	70	60	30	10	50

INDEX:	0	1	2	3	4	5
items	40	60	70	30	10	50

INDEX:	0	1	2	i = 3	4	5
items	40	60	70	30	10	50

INDEX:	0	1	2	3	4	5
items	30	40	60	70	10	50

INDEX:	0	1	2	3	i = 4	5
items	30	40	60	70	10	50

INDEX:	0	1	2	3	4	5
items	10	30	40	60	70	50

INDEX:	0	1	2	3	4	i = 5
items	10	30	40	60	70	50

INDEX:	0	1	2	3	4	5
items	10	30	40	50	60	70

참고: 삽입 정렬 (5/5)

❑ **삽입 정렬**(Insertion Sort) contd.

-삽입 정렬 수행 시간 분석-

- **입력이 역으로 정렬되어 있는 경우 (최악의 경우)**
  - 첫 번째 단계에서는 한 번의 비교가 수행
  - 두 번째 단계에서는 두 번의 비교가 수행
  - 마지막 단계에서는 N-1 번의 비교가 수행
  - 따라서 원소들의 총 **비교 횟수**는:

$$1 + 2 + \ldots + (N-2) + (N-1) = \frac{N(N-1)}{2} = O(N^2)$$

  - 원소를 **교환하는 횟수**도 $\frac{N(N-1)}{2}$
- **입력이 임의의 순서로 나열되어 있는 경우 (평균의 경우)**
  - 정렬되지 않은 부분의 가장 왼쪽 원소(현재 원소)가 정렬된 부분에 최종적으로 삽입되는 곳은 평균적으로 정렬된 부분의 중간이므로 $\frac{1}{2} \times \frac{N(N-1)}{2} = O(N^2)$

-삽입 정렬의 특징-

- 입력에 민감함(Input Sensitive): 입력에 따라 소요되는 수행 시간이 변함
  - 입력이 이미 정렬되어 있는 경우(**최선의 경우**), N-1번 비교하면(교환 X) 정렬을 마치므로 **O(N)**
  - 입력이 역으로 정렬되어 있는 경우(**최악의 경우**) 혹은 임의의 순서로 나열되어 있는 경우(**평균의 경우**) **$O(N^2)$**
- 이미 정렬된 데이터의 뒷부분에 소량의 신규 데이터를 추가하여 정렬하는 경우(입력이 거의 정렬된 경우) 우수한 성능을 보임
- 입력의 사이즈가 작은 경우 우수한 성능을 보임

참고: 셸 정렬 (1/8)

❑ 셸 정렬(Shell Sort)

- 셸 정렬은 (Python) 리스트 내에서 **일정한 간격 h**로 떨어져 있는 원소들끼리 서브리스트(Sub-list)를 구성하고 각 서브리스트에 있는 원소들에 대해서 **삽입 정렬을 수행**하는 연산을 반복하면서 전체 원소들을 정렬하는 알고리즘
- **h-정렬**
  - 간격이 h인 원소들로 구성된 h 개의 서브리스트에 대해 삽입 정렬을 수행하는 알고리즘

INDEX:	0	1	2	3	4	5	6	7	8	9	10	11
입력	39	23	15	47	11	56	61	16	12	19	21	41

⬇

INDEX:	0	1	2	3	4	5	6	7	8	9	10	11
h-정렬 수행 후(h = 4)	11	19	15	16	12	23	21	41	39	56	61	47

- 셸 정렬은 h 값(간격)을 줄여가며 여러 단계를 거쳐 h-정렬을 수행하고, 마지막엔 간격을 1로 하여 정렬
  - h-정렬은 삽입 정렬을 수행하기 전에 작은 값을 가진 원소들을 리스트의 앞부분으로 옮기고, 큰 값을 가진 원소들이 리스트의 뒷부분으로 옮기는 **전처리 과정**임
  - h = 1인 경우는 삽입 정렬과 동일 – **이미 부분적으로 정렬된 원소들의 수가 많아져서 매우 빠른 속도로 정렬**이 됨
    - 즉 셸 정렬은 **삽입 정렬은 입력이 거의 정렬된 경우 및 입력의 사이즈가 적은 경우 좋은 성능을 보인다**는 사실에 의해 고안된 정렬 알고리즘

참고: 셸 정렬 (2/8)

- 셸 정렬(Shell Sort) contd.

대표적인 간격 h의 순서(h-sequence), 출처: WIKIPEDIA

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A168604	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_q = \min\left\{n \in \mathbb{N} \colon n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p \colon 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\right\}$ $I = \left\{0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k\right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8\ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{\left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1\right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11}\left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right) \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]

참고: 셸 정렬 (3/8)

❑ 셸 정렬(Shell Sort) contd.

예: items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]에 대해 셸 정렬 수행 과정 (1단계)

INDEX:	0	1	2	3	4	5	6	7	8	9	10	11
	39	23	15	47	11	56	61	16	12	19	21	41

$$h_1 = \left\lfloor \frac{12}{2} \right\rfloor = 6$$ h_1 만큼 떨어진 원소들로 구성된 h_1 개의 서브리스트에 대해 삽입정렬 수행

39	23	15	47	11	56	**61**	16	12	19	21	41	sub1 = [39, 61]
39	**23**	15	47	11	56	61	**16**	12	19	21	41	sub2 = [23, 16]
39	23	**15**	47	11	56	61	16	**12**	19	21	41	sub3 = [15, 12]
39	23	15	**47**	11	56	61	16	12	**19**	21	41	sub4 = [47, 19]
39	23	15	47	**11**	56	61	16	12	19	**21**	41	sub5 = [11, 21]
39	23	15	47	11	**56**	61	16	12	19	21	**41**	sub6 = [56, 41]

참고: 셸 정렬 (4/8)

❑ **셸 정렬**(Shell Sort) contd.

예: items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]에 대해 셸 정렬 수행 과정 (1단계) contd.

39	23	15	47	11	56	**61**	16	12	19	21	41
39	23	15	47	11	56	**61**	16	12	19	21	41

sub1을 삽입 정렬

39	**23**	15	47	11	56	61	**16**	12	19	21	41
39	**16**	15	47	11	56	61	**23**	12	19	21	41

sub2를 삽입 정렬

39	23	**15**	47	11	56	61	16	**12**	19	21	41
39	23	**12**	47	11	56	61	16	**15**	19	21	41

sub3을 삽입 정렬

39	23	15	**47**	11	56	61	16	12	**19**	21	41
39	23	15	**19**	11	56	61	16	12	**47**	21	41

sub4를 삽입 정렬

39	23	15	47	**11**	56	61	16	12	19	**21**	41
39	23	15	47	**11**	56	61	16	12	19	**21**	41

sub5를 삽입 정렬

39	23	15	47	11	**56**	61	16	12	19	21	**41**
39	23	15	47	11	**41**	61	16	12	19	21	**56**

sub6을 삽입 정렬

1단계 정렬 결과

39	16	12	19	11	41	61	23	15	47	21	56

참고: 셸 정렬 (5/8)

- **셸 정렬**(Shell Sort) contd.

예: items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]에 대해 셸 정렬 수행 과정 (2단계)

INDEX:	0	1	2	3	4	5	6	7	8	9	10	11
	39	16	12	19	11	41	61	23	15	47	21	56

$$h_2 = \left\lfloor \frac{6}{2} \right\rfloor = 3$$

h_2 만큼 떨어진 원소들로 구성된 h_2 개의 서브리스트에 대해 삽입정렬 수행

39	16	12	**19**	11	41	**61**	23	15	**47**	21	56

sub1 = [39, 19, 61, 47]

39	**16**	12	19	**11**	41	61	**23**	15	47	**21**	56

sub2 = [16, 11, 23, 21]

39	16	**12**	19	11	**41**	61	23	**15**	47	21	**56**

sub3 = [12, 41, 15, 56]

참고: 셸 정렬 (6/8)

❑ **셸 정렬**(Shell Sort) contd.

예: items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]에 대해 셸 정렬 수행 과정 (2단계) contd.

39	16	12	**19**	11	41	**61**	23	15	**47**	21	56
19	16	12	**39**	11	41	**47**	23	15	**61**	21	56

sub1을 삽입 정렬

39	**16**	12	19	**11**	41	61	**23**	15	47	**21**	56
39	**11**	12	19	**16**	41	61	**21**	15	47	**23**	56

sub2를 삽입 정렬

39	16	**12**	19	11	**41**	61	23	**15**	47	21	**56**
39	16	**12**	19	11	**15**	61	23	**41**	47	21	**56**

sub3을 삽입 정렬

2단계 정렬 결과

19	11	12	39	16	15	47	21	41	61	23	56

참고: 셸 정렬 (7/8)

❑ **셸 정렬**(Shell Sort) contd.

예: items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]에 대해 셸 정렬 수행 과정 (3단계)

INDEX:	0	1	2	3	4	5	6	7	8	9	10	11
	19	11	12	39	16	15	47	21	41	61	23	56

$$h_3 = \left\lfloor \frac{3}{2} \right\rfloor = 1$$

h_3 만큼 떨어진 원소들로 구성된 h_3 개의 서브리스트에 대해 삽입정렬 수행

19	**11**	**12**	**39**	**16**	**15**	**47**	**21**	**41**	**61**	**23**	**56**

sub1 = [19, 11, 12, 39, 16, 15, 47, 21, 41, 61, 23, 56] (전체 리스트)에 대해서 삽입 정렬

3단계(최종 단계) 정렬 결과

11	12	15	16	19	21	23	39	41	47	56	61

연습: items = [29, 5, 7, 19, 13, 24, 31, 8, 82, 18, 63, 44]에 대해 셸정렬 수행 과정을 보이시오.

참고: 셸 정렬 (8/8)

❑ 셸 정렬(Shell Sort) contd.

- 셸 정렬을 위한 함수 shell_sort 정의

```python
def shell_sort(items):
    h = len(items) // 2
    while h >= 1:
        for i in range(h, len(items)):
            j = i
            while j >= h and items[j] < items[j - h]:
                items[j], items[j - h] = items[j - h], items[j]
                j -= h
        print("{}-정렬 결과: ".format(h), items)
        h //= 2
items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]
print('정렬 전: ', end='')
print(items)
shell_sort(items)
print('정렬 후: ', end='')
print(items)
```

```
정렬 전: [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]
6-정렬 결과:  [39, 16, 12, 19, 11, 41, 61, 23, 15, 47, 21, 56]
3-정렬 결과:  [19, 11, 12, 39, 16, 15, 47, 21, 41, 61, 23, 56]
1-정렬 결과:  [11, 12, 15, 16, 19, 21, 23, 39, 41, 47, 56, 61]
정렬 후: [11, 12, 15, 16, 19, 21, 23, 39, 41, 47, 56, 61]
```

결과

-셸 정렬 수행 시간 분석-

- 쉘 정렬의 수행 시간은 h의 값에 따라 달라지므로 정확한 분석이 어려움
- 일반적으로 입력의 사이즈가 작은 경우 좋은 성능을 보임

참고: 힙 정렬 (1/4)

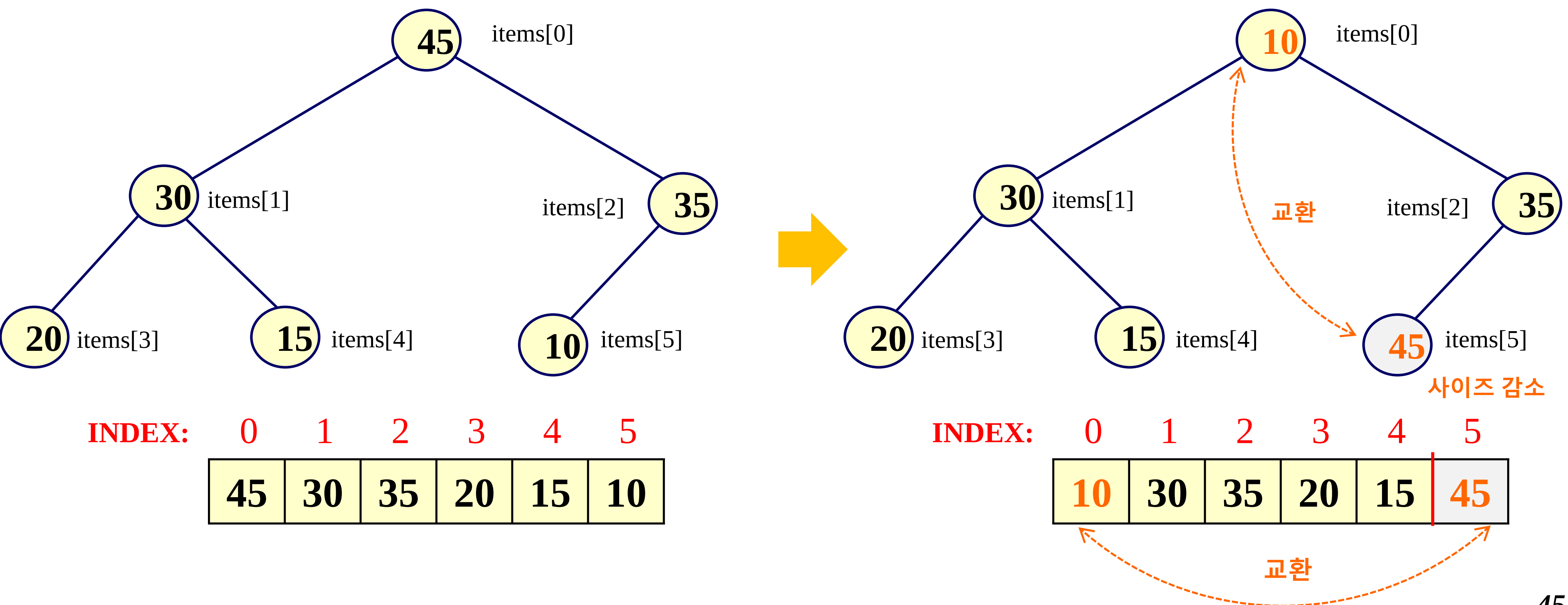
❑ 힙 정렬(Heap Sort)

- **힙(Heap)을 이용**하는 정렬 알고리즘

-힙 정렬 과정-

1. Python 리스트를 Max 힙으로 구성
2. 루트 노드를 힙의 가장 마지막 노드와 교환한 후, 힙 사이즈(Python 리스트의 사이즈)를 1 감소시키고 노드 교환으로 인해 위배된 힙 속성을 downheap 연산으로 복원하는 과정을 **힙 사이즈가 1이 될 때까지** 반복

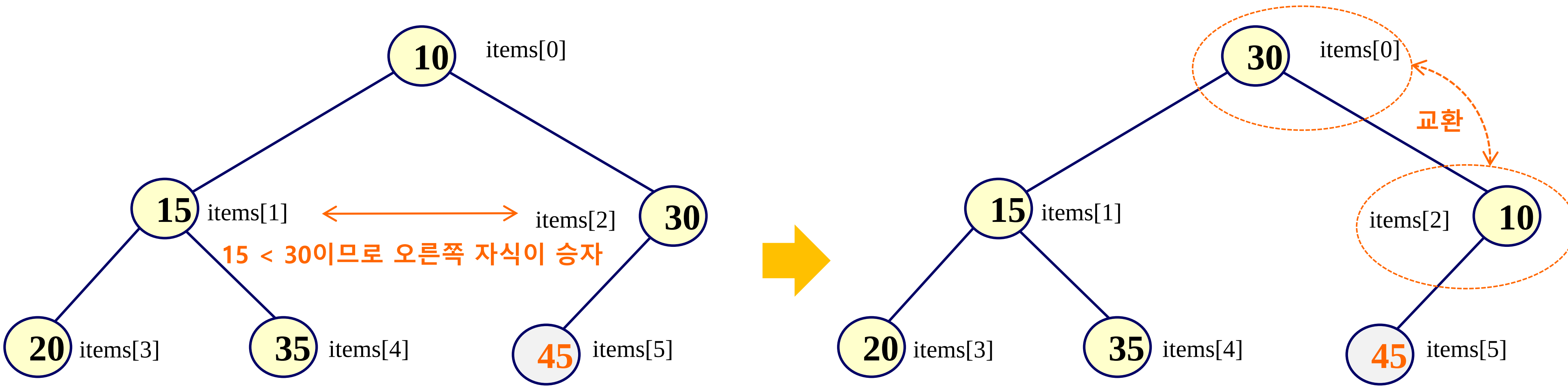
예: 루트 노드와 힙의 마지막 노드 교환 후 downheap 연산 수행 과정



참고: 힙 정렬 (2/4)

❑ **힙 정렬**(Heap Sort) contd.

예: 루트 노드와 힙의 마지막 노드 교환 후 downheap 연산 수행 과정 contd.



INDEX:	0	1	2	3	4	5
	10	35	30	20	15	45

교환

INDEX:	0	1	2	3	4	5
	30	35	10	20	15	45

교환

참고: 힙 정렬 (3/4)

❑ 힙 정렬(Heap Sort) contd.

- **힙 정렬을 위한 함수 downheap, heapify, heap_sort 정의**

```
def downheap(i, size):
    while 2*i + 1 <= size:
        k = 2*i + 1
        if k < size - 1 and items[k] < items[k+1]:
            k += 1
        if items[i] >= items[k]:
            break
        items[i], items[k] = items[k], items[i]
        i = k

def heapify(items):
    hsize = len(items)
    for i in range(hsize//2-1, -1, -1):
        downheap(i, hsize)

def heap_sort(items):
    N = len(items)
    for i in range(N):
        items[0], items[N-1] = items[N-1], items[0]
        downheap(0, N-2)
        N -= 1
items = [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]
print('정렬 전: ', end='')
print(items)
heapify(items)
print('최대힙 : ', end='')
print(items)
heap_sort(items)
print('정렬 후: ', end='')
print(items)
```

```
정렬 전: [39, 23, 15, 47, 11, 56, 61, 16, 12, 19, 21, 41]
최대힙 : [61, 47, 56, 23, 21, 41, 15, 16, 12, 19, 11, 39]
정렬 후: [11, 12, 15, 16, 19, 21, 23, 39, 41, 47, 56, 61]
```

결과

참고: 힙 정렬 (4/4)

❑ 힙 정렬(Heap Sort) contd.

-힙 정렬 수행 시간 분석-

- 상향식(Bottom-up)으로 힙을 구성하는 시간: **O(N)**
- 루트 노드와 힙의 마지막 노드를 교환한 후 downheap 수행 시간: **O(logN)**
- 루트 노드와 힙의 마지막 노드를 교환하는 횟수: N - 1
- 따라서 총 수행 시간은:

  O(N) + (N - 1) × O(logN) = **O(NlogN)**

-힙 정렬의 특징-

- 입력에 민감하지 않음(Input Insensitive): 어떠한 입력에 대해서도 항상 O(NlogN) 수행 시간이 소요됨
- 선택, 삽입, 쉘 정렬과 마찬가지로 정렬 시 추가 메모리 공간이 필요하지 않음

  참고: 추가 메모리 공간 사용을 가정하여 힙 정렬을 구현하는 방법:

  오름차순 정렬의 경우 Min 힙을 구성 후, 힙이 empty가 될 때까지 extract_min 메소드를 이용하여 원소들을 하나씩 삭제 및 반환 후 빈 Python 리스트에 append
- 일반적으로 입력의 사이즈가 큰 경우 성능이 좋지 않음