

제 7 장 정렬

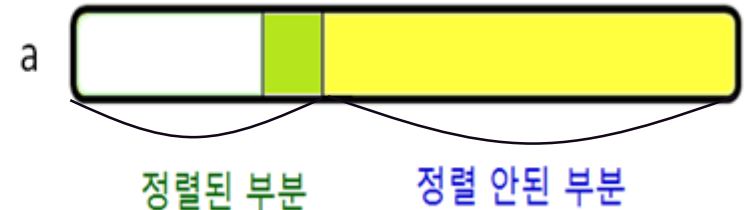
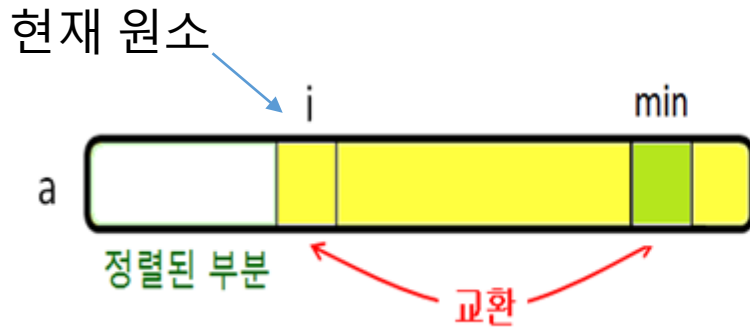
정렬

- 선택정렬
- 삽입정렬
- 쉘정렬
- 힙정렬
- 합병정렬
- 퀵정렬
- 기수정렬
- 외부정렬

- 이중피벗퀵정렬 (부록 II)
- Tim Sort (부록 III)
- 정렬의 하한 (부록 IV)

8.1 선택 정렬

- 선택 정렬(Selection Sort)은 배열에서 아직 정렬되지 않은 부분의 원소들 중에서 최솟값을 '선택'하여 정렬된 부분의 바로 오른쪽 원소와 교환하는 정렬알고리즘



- (a) 리스트 a의 왼쪽 부분은 이미 정렬되어 있고 나머지 부분은 아직 정렬 안된 상태
- 정렬된 부분의 키들은 오른쪽의 정렬 안된 부분의 어떤 키보다 크지 않다.
- 선택 정렬은 항상 정렬 안된 부분에서 최솟값(min)을 찾아 왼쪽의 정렬된 부분의 바로 오른쪽 원소(현재 원소)로 옮기기 때문
- 이 과정은 그림(a)에서 min을 a[i]와 교환 후에 (b)와 같이 i를 1 증가시키며, 이를 반복적으로 수행

```

01 def selection_sort(a):
02     for i in range(0, len(a)-1):
03         minimum = i
04         for j in range(i, len(a)):
05             if a[minimum] > a[j]:
06                 minimum = j
07         a[i], a[minimum] = a[minimum], a[i]
08
09 a = [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]
10 print('정렬 전:\t', end='')
11 print(a)
12 selection_sort(a)
13 print('정렬 후:\t', end='')
14 print(a)

```

정렬 안된 부분에서
최솟값 찾기

현재 원소와 최솟값
가진 원소 교환

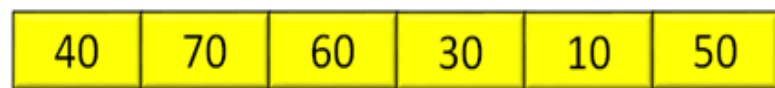
Console PyUnit

<terminated> selection.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

정렬 전: [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]

정렬 후: [10, 11, 17, 17, 17, 20, 22, 26, 31, 44, 49, 54, 77, 77, 88, 93]

[예제] 40, 70, 60, 30, 10, 50에 대해 selection_sort 수행 과정



$i = 0$

min



교환

$i = 1$

min



교환

$i = 2$

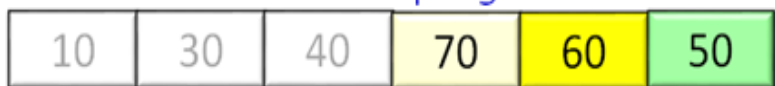
min



교환

$i = 3$

min



교환

min



교환

$i = 4$

수행 시간

- 선택 정렬은 루프가 1 번 수행될 때마다 정렬 안된 부분에서 가장 작은 원소를 선택
- 처음 루프가 수행될 때 N개의 원소들 중에서 min을 찾기 위해 N-1번 원소 비교
- 루프가 2 번째 수행될 때 N-1개의 원소들 중에서 min을 찾는 데 N-2번 비교
- 같은 방식으로 루프가 마지막으로 수행될 때: 2 개의 원소 1번 비교하여 min을 찾음
- 따라서 원소들의 총 비교 횟수

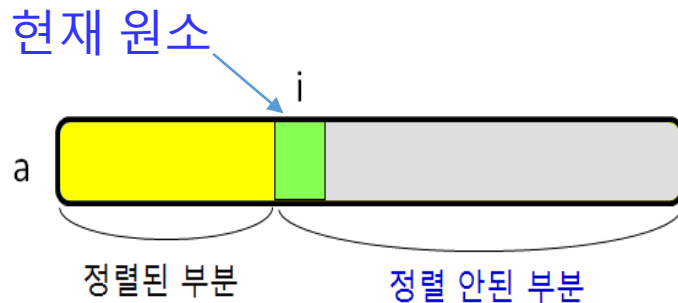
$$(N-1) + (N-2) + (N-3) + \dots + 2 + 1 = \frac{N(N-1)}{2} = O(N^2)$$

선택 정렬의 특징

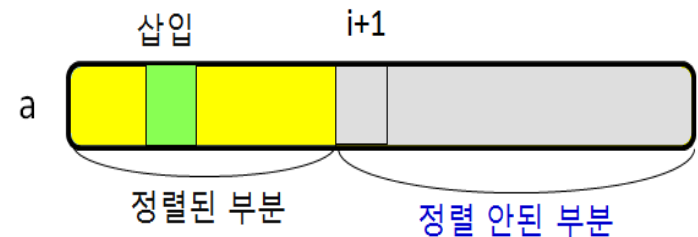
- 입력에 민감하지 않음(Input Insensitive)
 - 항상 $O(N^2)$ 수행시간이 소요
- 최솟값을 찾은 후 원소를 교환하는 횟수가 $N-1$
 - 이는 정렬알고리즘들 중에서 가장 작은 최악 경우 교환 횟수
- 하지만 선택 정렬은 효율성 측면에서 뒤떨어지므로 거의 활용되지 않음

8.2 삽입 정렬

- 삽입 정렬(Insertion Sort)은 리스트가 정렬된 부분과 정렬 안된 부분으로 나뉘며, 정렬 안된 부분의 가장 왼쪽 원소를 정렬된 부분에 '삽입'하는 방식의 정렬알고리즘



(a) 삽입 수행 전



(b) 삽입 수행 후

(a) 정렬 안된 부분의 가장 왼쪽 원소 i (현재 원소)를 정렬된 부분의 원소들을 비교하며 (b)와 같이 현재 원소 삽입.

- 현재 원소 삽입 후
 - 정렬된 부분의 원소 수가 1 증가
 - 정렬 안된 부분의 원소 수는 1 감소

현재 원소인 50을 정렬된 부분에 삽입하는 과정



```

01 def insertion_sort(a):
02     for i in range(1, len(a)):
03         for j in range(i, 0, -1):
04             if a[j-1] > a[j]:
05                 a[j], a[j-1] = a[j-1], a[j]
06
07 a = [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]
08 print('정렬 전:\t', end='')
09 print(a)
10 insertion_sort(a)
11 print('정렬 후:\t', end='')
12 print(a)

```

현재 원소의 인덱스

현재 원소가 정렬된 부분에
삽입될 곳을 찾아서

현재 원소와 직전
원소의 교환

Console PyUnit

<terminated> selection.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

정렬 전: [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]

정렬 후: [10, 11, 17, 17, 17, 20, 22, 26, 31, 44, 49, 54, 77, 77, 88, 93]

[예제] 40, 60, 70, 50, 10, 30, 20에 대해 insertion_sort 수행 과정

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

i=1 60

70		40	50	10	30	20
----	--	----	----	----	----	----



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

i=2 40

60	70		50	10	30	20
----	----	--	----	----	----	----



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

i=3 50

40	60	70		10	30	20
----	----	----	--	----	----	----



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

i=4 10

40	50	60	70		30	20
----	----	----	----	--	----	----



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

i=5 30

10	40	50	60	70		20
----	----	----	----	----	--	----



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

i=6 20

10	30	40	50	60	70	
----	----	----	----	----	----	--



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

수행 시간

- 삽입 정렬은 입력에 민감 (Input Sensitive)

- 입력이 이미 정렬된 경우(최선 경우)

N-1번 비교하면 정렬을 마침 = $O(N)$

- 입력이 역으로 정렬된 경우 (최악 경우)

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

- 최악 경우 데이터 교환 수: $O(N^2)$

- 입력 데이터의 순서가 랜덤인 경우(평균 경우)

현재 원소가 정렬된 앞 부분에 최종적으로 삽입되는 곳이
평균적으로 정렬된 부분의 중간이므로 $\frac{1}{2} \times \frac{N(N-1)}{2} \approx \frac{1}{4}N^2 =$
 $O(N^2)$

Applications

- 이미 정렬된 파일의 뒷부분에 소량의 신규 데이터를 추가하여 정렬하는 경우(입력이 거의 정렬된 경우) 우수한 성능을 보임
- 입력 크기가 작은 경우에도 매우 좋은 성능을 보임
삽입 정렬은 재귀 호출을 하지 않으며, 프로그램도 매우 간단하기 때문
- 삽입 정렬은 합병정렬이나 퀵정렬과 함께 사용되어 실질적으로 보다 빠른 성능에 도움을 줌
 - 단, 이론적인 수행 시간은 향상되지 않음

8.3 셸정렬

- 셸(Shell Sort)정렬은 삽입 정렬에 전처리과정을 추가한 것
- 전처리과정이란 작은 값을 가진 원소들을 배열의 앞부분으로 옮기며 큰 값을 가진 원소들이 배열의 뒷부분에 자리잡도록 만드는 과정
- 삽입정렬이 현재 원소를 앞부분에 삽입하기 위해 이웃하는 원소의 숫자끼리 비교하며 한 자리씩 이동하는 단점 보완
- 전처리과정은 여러 단계로 진행되며, 각 단계에서는 일정 간격으로 떨어진 원소들에 대해 삽입 정렬 수행

전처리과정 전과 후

입력

65	95	90	80	55	70	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

4-정렬 후

10	25	35	30	55	70	40	50	65	95	90	80
----	----	----	----	----	----	----	----	----	----	----	----

- **h-정렬(h-sort)**: 간격이 h인 원소끼리 정렬하는 것
- 4-정렬 후 결과: 작은 숫자들(10, 25, 35)이 배열의 앞부분으로, 큰 숫자들 (95, 90, 80)이 뒷부분으로 이동
- **셸정렬**은 h-정렬의 h 값(간격)을 줄여가며 정렬을 수행하고, 마지막엔 간격을 1로 하여 정렬
- h = 1인 경우는 삽입 정렬과 동일

대표적인 간격의 순서(h-Sequence)

Shell	$N/2, N/4, \dots, 1$ (나누기 2를 계속하여 1이 될 때까지의 순서)
Hibbard	$2^k-1, 2^{k-1}-1, \dots, 7, 3, 1$
Knuth	$(3^k - 1)/2, \dots, 13, 4, 1$
Sedgewick	$\dots, 109, 41, 19, 5, 1$
Marcin Ciura	1750, 701, 301, 132, 57, 23, 10, 4, 1

```

01 def shell_sort(a):
02     h = 4          # 3x+1 간격: 1, 4, 13, 40, 121, ... 중에서 4 와 1만 사용
03     while h >= 1:
04         for i in range(h, len(a)): # h-정렬 수행
05             j = i
06             while j >= h and a[j] < a[j-h]:
07                 a[j], a[j-h] = a[j-h], a[j]
08                 j -= h
09             h //= 3
10 a = [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]
11 print('정렬 전:\t', end='')
12 print(a)
13 shell_sort(a)
14 print('정렬 후:\t', end='')
15 print(a)

```

각 그룹에 대해 삽입 정렬 수행

다음 h 값 계산

Console PyUnit

<terminated> selection.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

정렬 전: [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]

정렬 후: [10, 11, 17, 17, 17, 20, 22, 26, 31, 44, 49, 54, 77, 77, 88, 93]

[예제] 4-정렬하는 과정

입력

65	95	90	80	55	70	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

i = 4 j = 4

65	95	90	80	55	70	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

55	95	90	80	65	70	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

i = 5 j = 5

55	95	90	80	65	70	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

55	70	90	80	65	95	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

i = 6 j = 6

55	70	90	80	65	95	35	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

55	70	35	80	65	95	90	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

i = 7 j = 7

55	70	35	80	65	95	90	50	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

55	70	35	50	65	95	90	80	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

i = 8 j = 8

55	70	35	50	65	95	90	80	10	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

55	70	35	50	10	95	90	80	65	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

j = 4

55	70	35	50	10	95	90	80	65	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

10	70	35	50	55	95	90	80	65	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

i = 9 j = 9

10	70	35	50	55	95	90	80	65	25	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

10	70	35	50	55	25	90	80	65	95	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

j = 5

10	70	35	50	55	25	90	80	65	95	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

10	25	35	50	55	70	90	80	65	95	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

$i=10$ $j=10$

10	25	35	50	55	70	90	80	65	95	40	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

10	25	35	50	55	70	40	80	65	95	90	30
----	----	----	----	----	----	----	----	----	----	----	----

 교환

$j=6$

10	25	35	50	55	70	40	80	65	95	90	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

$i=11$ $j=11$

10	25	35	50	55	70	40	80	65	95	90	30
----	----	----	----	----	----	----	----	----	----	----	----

 비교

10	25	35	50	55	70	40	30	65	95	90	80
----	----	----	----	----	----	----	----	----	----	----	----

 교환

$j=7$

10	25	35	50	55	70	40	30	65	95	90	80
----	----	----	----	----	----	----	----	----	----	----	----

 비교

10	25	35	30	55	70	40	50	65	95	90	80
----	----	----	----	----	----	----	----	----	----	----	----

 교환

수행 시간

- 수행 시간은 간격을 어떻게 설정하느냐에 따라 달라짐
- Hibbard의 간격: 2^k-1 (즉, $2^k-1, \dots, 5, 3, 1$) $O(N^{1.5})$ 시간
- Marcin Ciura의 간격: 1, 4, 10, 23, 57, 132, 301, 701, 1750
- 정확한 수행 시간은 아직 풀리지 않은 문제
- 일반적으로 쉘정렬은 입력이 그리 크지 않은 경우에 매우 좋은 성능을 보임

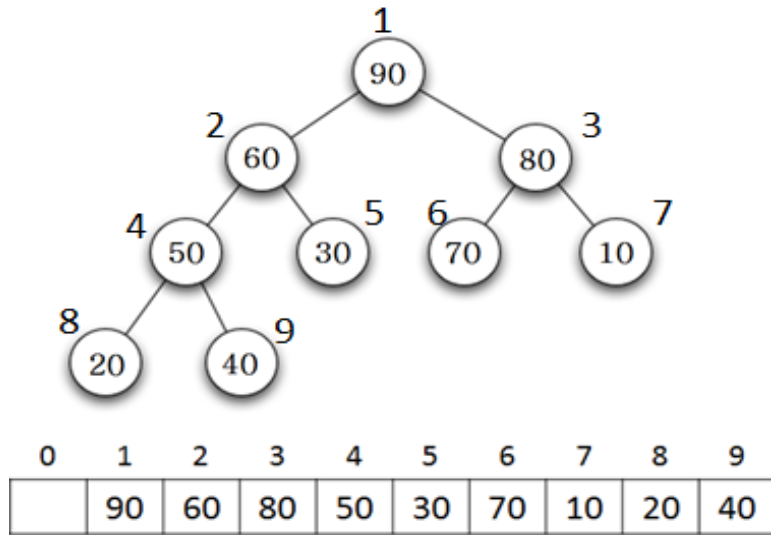
Applications

- 셀정렬은 임베디드(Embedded) 시스템에서 주로 사용되는데, 이는 간격에 따른 그룹별 정렬알고리즘을 하드웨어 설계를 통해 구현하는 것이 매우 쉽기(효율적이기) 때문

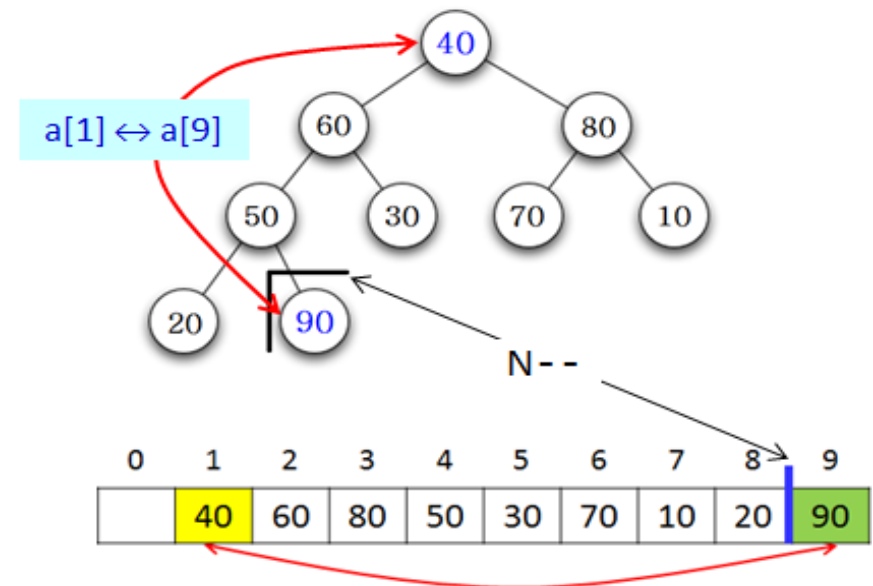
8.4 힙정렬

- 힙정렬(Heap Sort)은 이진힙을 이용하는 정렬
- 먼저 배열에 저장된 데이터의 키를 우선순위로 하는 최대힙(Max Heap)을 구성
- 루트를 힙의 가장 마지막 노드와 교환한 후
- 힙 크기를 1 감소시키고
- 루트로 의 이동으로 인해 위배된 힙속성을 downheap연산으로 복원
- 힙정렬은 이 과정을 반복하여 나머지 원소들을 정렬

루트와 힙의 마지막 노드 교환 후 downheap 연산 수행 과정

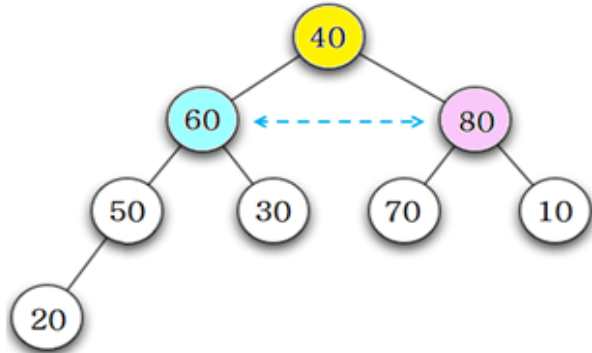


(a) 입력과 최대힙



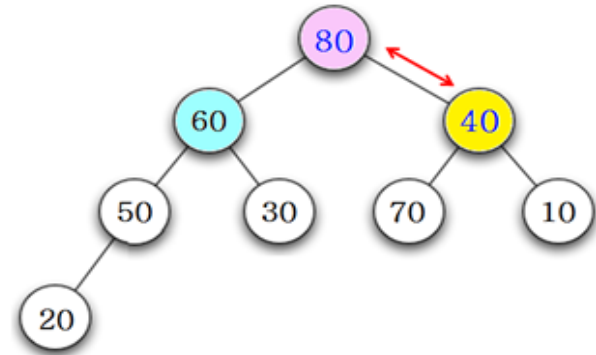
(b) 루트와 마지막 노드 교환

downheap()



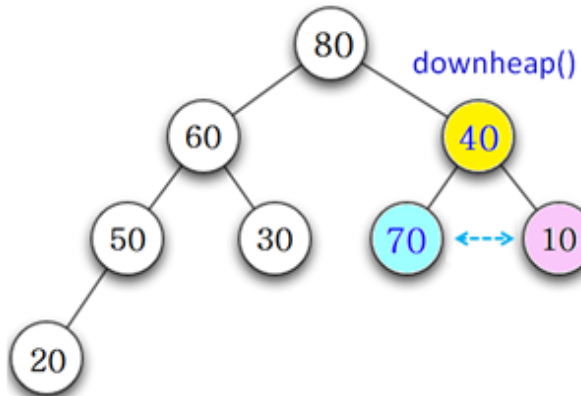
0	1	2	3	4	5	6	7	8	9
	40	60	80	50	30	70	10	20	90

(c) 루트의 두 자식 비교



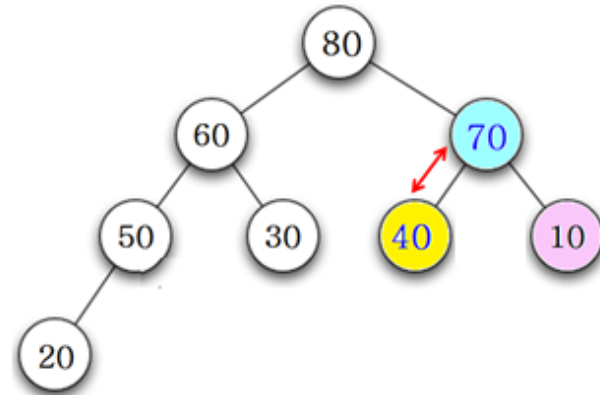
0	1	2	3	4	5	6	7	8	9
	80	60	40	50	30	70	10	20	90

(d) 루트와 자식 승자와 교환



0	1	2	3	4	5	6	7	8	9
	80	60	40	50	30	70	10	20	90

(e) 40의 두 자식 비교



0	1	2	3	4	5	6	7	8	9
	80	60	70	50	30	40	10	20	90

(f) 40과 자식 승자와 교환

- 힙정렬은 입력을 (a)와 같은 최대힙으로 만든다. 노드 옆의 숫자는 노드에 대응되는 리스트 원소의 인덱스
- (b) 루트와 마지막 노드를 교환한 후에 힙 크기를 1 줄이고,
- (c)~(f) `downheap()`을 2번 수행하여 위배된 힙속성을 충족시킴
- 이후의 과정은 $a[1] \sim a[8]$ 에 대해 동일한 과정을 반복 수행하여 힙 크기가 1이 되었을 때 종료

[예제] 앞선 예제에 이은 힙정렬 수행 과정

0	1	2	3	4	5	6	7	8	9
	80	60	70	50	30	40	10	20	90

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

교환

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

downheap()

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

교환

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

downheap()

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

교환

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

downheap()

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

교환

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

downheap()

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

교환

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

downheap()

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

교환

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

downheap()

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

교환

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

```
01 def downheap(i, size):
02     while 2*i <= size:
03         k = 2*i
04         if k < size and a[k] < a[k+1]:
05             k += 1
06         if a[i] >= a[k]:
07             break
08         a[i], a[k] = a[k], a[i]
09         i = k
```

루트로 올라온 키에 대해
힙속성을 회복시킴

```
10
11 def create_heap(a):
12     hsize = len(a) - 1
13     for i in reversed(range(1, hsize//2+1)):
14         downheap(i, hsize)
15
```

정렬하기 전에
최대힙 만들기

```

16 def heap_sort(a):
17     N = len(a) - 1
18     for i in range(N):
19         a[1], a[N] = a[N], a[1]
20         downheap(1, N-1)
21         N -= 1
22
23 a = [-1,54,88,77,26,93,17,49,10,17,77,11,31,22,44,17,20]
24 print('정렬 전:\t', end='')
25 print(a)
26 create_heap(a) # 힙 만들기
27 print('최대힙 : \t', end='')
28 print(a)
29 heap_sort(a)
30 print('정렬 후 :\t', end='')
31 print(a)


```

루트와 힙의
마지막 항목 교환

힙 크기 1 감소

[프로그램 7-4] heap_sort.py

heap_sort 수행 결과

Console  PyUnit

<terminated> heap_sort.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

정렬 전 : [-1, 54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]

최대힙 : [-1, 93, 88, 77, 26, 77, 31, 49, 20, 17, 54, 11, 17, 22, 44, 17, 10]

정렬 후 : [-1, 10, 11, 17, 17, 17, 20, 22, 26, 31, 44, 49, 54, 77, 77, 88, 93]

```
01 import heapq
02 a = [54,88,77,26,93,17,49,10,17,77,11,31,22,44,17,20]
03 print('정렬전:\t', a)
04
05 heapq.heapify(a) ●
06 print('힙:\t', a)
07
08 s = [] ●
09 while a:
10     s.append(heapq.heappop(a)) ●
11 print('정렬후:\t', s)
```

최소힙 만들기

정렬 결과를 저장 할 리스트

리스트 a의 가장 작은 항목을 삭제하여 리스트 s의 맨 뒤에 추가

[프로그램 7-5] 파이썬의 `heapq`를 이용한 힙정렬

수행 시간

- 먼저 상향식(Bottom-up)으로 힙을 구성: $O(N)$ 시간
- 루트와 힙의 마지막 노드를 교환한 후 `downheap()` 수행: $O(\log N)$ 시간
- 루트와 힙의 마지막 노드를 교환하는 횟수: $N-1$ 번
- 총 수행시간: $O(N) + (N-1) \times O(\log N) = O(N \log N)$
- 힙정렬은 어떠한 입력에도 항상 $O(N \log N)$ 시간이 소요
- 루프 내의 코드가 길고, 비효율적인 캐시 메모리 사용에 따라 특히 대용량의 입력을 정렬하기에 부적절
- C/C++ 표준 라이브러리(STL)의 `partial_sort`(부분 정렬)는 힙정렬로 구현됨
 - 부분 정렬: 가장 작은 k 개의 원소만 출력

8.5 합병 정렬

- 합병 정렬(Merge Sort)은 크기가 N 인 입력을 $1/2N$ 크기를 갖는 입력 2 개로 분할하고, 각각에 대해 재귀적으로 합병 정렬을 수행한 후, 2 개의 각각 정렬된 부분을 합병하는 정렬알고리즘
- 합병(Merge)이란 두 개의 각각 정렬된 입력을 합치는 것과 동시에 정렬하는 것
- 분할 정복(Divide-and-Conquer) 알고리즘: 입력을 분할하여 분할된 입력 각각에 대한 문제를 재귀적으로 해결한 후 취합하여 문제를 해결하는 알고리즘들

합병 과정

1	2	4	7	9	11	12
---	---	---	---	---	----	----

3	5	6	8	10	13
---	---	---	---	----	----



합병

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

```
01 def merge(a, b, low, mid, high):
```

```
02     i = low
```

```
03     j = mid+1
```

```
04     for k in range(low, high+1):
```

a의 앞/뒷부분 합병하여
b에 저장

```
05         if i > mid:
```

```
06             b[k] = a[j]
```

a의 앞부분이 먼저 소진되어
뒷부분 b로 복사

```
07             j += 1
```

```
08         elif j > high:
```

```
09             b[k] = a[i]
```

a의 뒷부분이 먼저 소진되어
앞부분 b로 복사

```
10             i += 1
```

```
11         elif a[j] < a[i]:
```

```
12             b[k] = a[j]
```

a[j]가 승자

```
13             j += 1
```

```
14         else:
```

```
15             b[k] = a[i]
```

a[i]가 승자

```
16             i += 1
```

```
17     for k in range(low, high+1):
```

b를 a로 복사

```
18         a[k] = b[k]
```

```
19
```

```

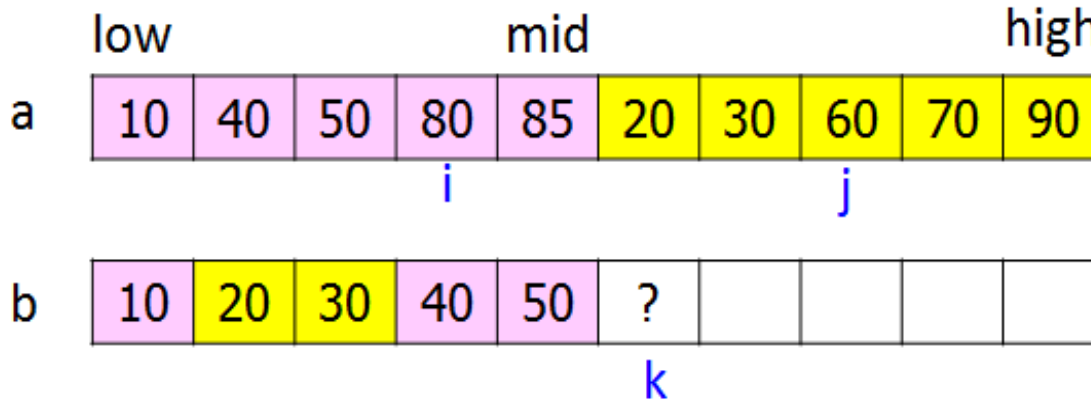
20 def merge_sort(a, b, low, high):
21     if high <= low:
22         return
23     mid = low + (high - low) // 2
24     merge_sort(a, b, low, mid)
25     merge_sort(a, b, mid + 1, high)
26     merge(a, b, low, mid, high)
27
28 a = [54,88,77,26,93,17,49,10,17,77,11,31,22,44,17,20]
29 b = [None] * len(a)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

```

● 중간 인덱스
 ● 앞/뒷부분 재귀호출
 ● 정렬된 앞/뒷부분 합병
 ● 보조 리스트
 ● 합병정렬 호출

[프로그램 7-6] merge_sort.py

- merge 함수에서 $a[\text{low}] \sim a[\text{mid}]$ 와 $a[\text{mid}+1] \sim a[\text{high}]$ 를 다음과 같이 합병



80과 60의 승자를 $b[k]$ 에 저장

- 60이 80보다 작으므로 60이 '승자'가 되어 $b[k]$ 에 저장
- 그 후 i 는 변하지 않고, j 와 k 만 각각 1씩 증가하고, 다시 $a[i]$ 와 $a[j]$ 의 승자를 선택
- 합병의 마지막 부분인 line 17에서 합병된 결과가 저장되어있는 $b[\text{low}] \sim b[\text{high}]$ 를 $a[\text{low}] \sim a[\text{high}]$ 로 복사

[예제] [80, 40, 50, 10, 70, 20, 30, 60]에 대한 합병정렬 수행 과정

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

low

mid

high

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

low mid

high

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

low high

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

mid

low

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

return

high

low

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

return

high

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

merge() 호출

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

합병

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

merge() 호출

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

합병

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

merge() 호출

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

합병

⋮

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


merge() 호출

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

합병

수행 시간

- 어떤 입력에 대해서도 $O(N\log N)$ 시간 보장
- 입력 크기 $N = 2^k$ 가정
- $T(N)$ = 크기가 N 인 입력에 대해 합병 정렬이 수행하는 원소 비교 횟수(시간)


$$\begin{cases} T(N) = 2T(N/2) + cN, & N > 1, c \text{는 상수} \\ T(1) = O(1) \end{cases}$$

$$\begin{aligned} T(N) &= 2T(N/2) + cN \\ &= 2[2T((N/2))/2 + c(N/2)] + cN \\ &= 4T(N/4) + 2cN \\ &= 4[2T(N/4)/2 + c(N/4)] + 2cN \\ &= 8T(N/8) + 3cN \\ &\vdots \\ &= 2^k T(N/2^k) + kcN, \quad N = 2^k, k = \log N \\ &= NT(1) + cN \log N = N \cdot O(1) + cN \log N \\ &= O(N) + O(N \log N) \\ &= O(N \log N) \end{aligned}$$

성능 향상 방법(1)

- 합병 정렬은 재귀 호출을 사용하므로 입력 크기가 1이 되어야 합병을 시작
- 이 문제점을 보완하기 위해 입력이 정해진 크기, 예를 들어, 7~10이 되면 삽입 정렬을 통해 정렬한 후 합병을 수행
- Line 21~22를 다음과 같이 수정. CALLSIZE = 7~10 정도

```
if high <= low:  
    return
```



```
if high < low + CALL_SIZE:  
    insertion_sort(a, low, high)  
    return
```

성능 향상 방법(2)

- 합병정렬에서는 입력 크기가 작아지면 합병하기 위한 두 개의 리스트가 이미 합병되어 있을 가능성이 높아짐
- 따라서 [프로그램 7-6]에서 line 26의 merge()를 호출하기 전에 다음의 if-문을 추가하면 불필요한 merge() 호출을 방지할 수 있음

```
if a[mid] <= a[mid+1]:  
    return
```

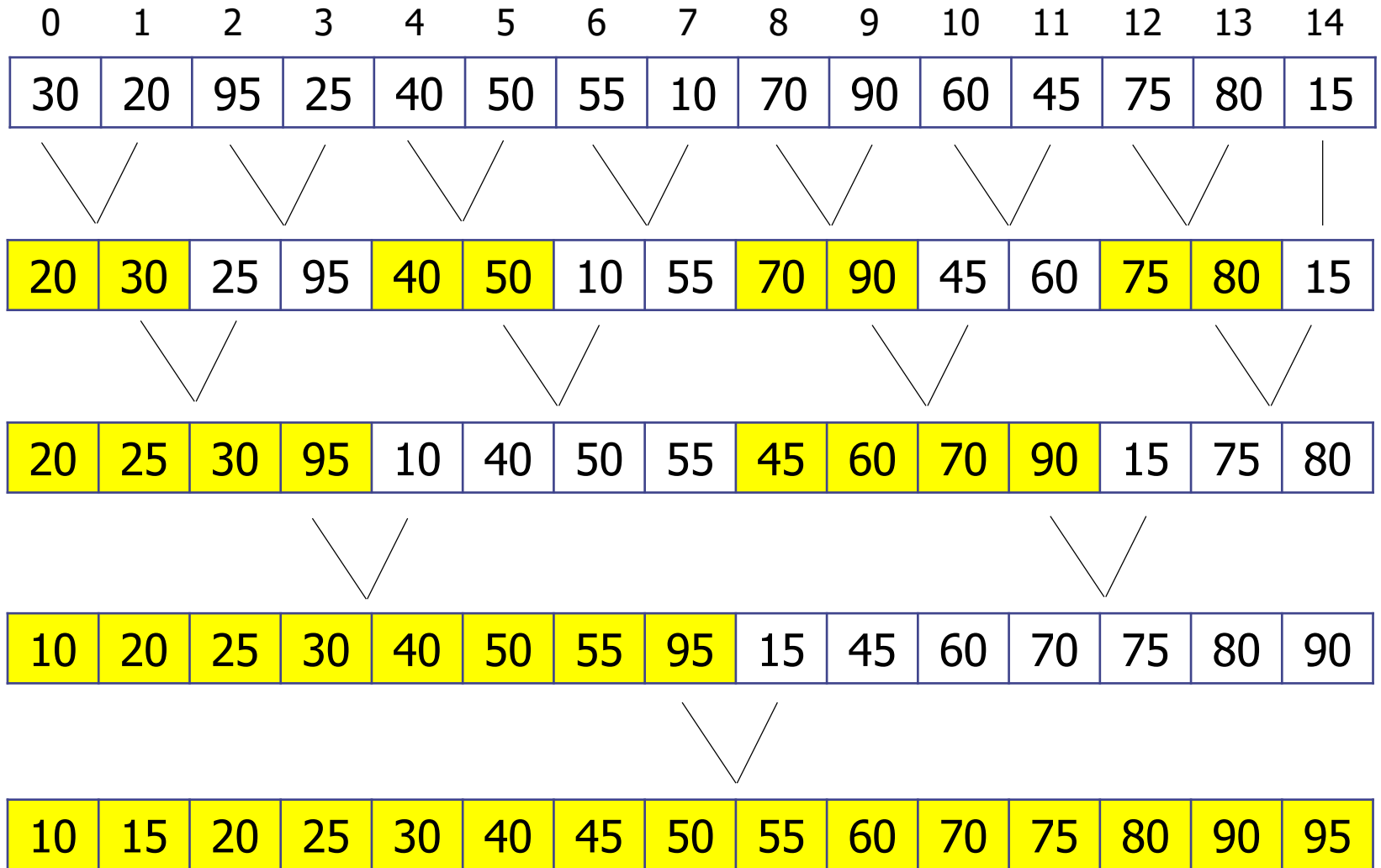
성능 향상 방법(3)

- merge()의 line 17에서 매번 보조 리스트 b를 입력 리스트 a로 복사하는데, 이를 a와 b를 번갈아 사용하도록 하여 합병 정렬의 성능을 향상시킬 수도 있음

반복 합병 정렬

- 입력 리스트에서 바로 2 개씩 짝지어 합병한 뒤, 다시 4 개씩 짝지어 합병하는 상향식 (Bottom-up)으로도 수행 가능
- 이러한 합병 정렬을 Bottom-up 합병 또는 반복(Iterative) 합병 정렬이라함

[예제] 반복합병정렬

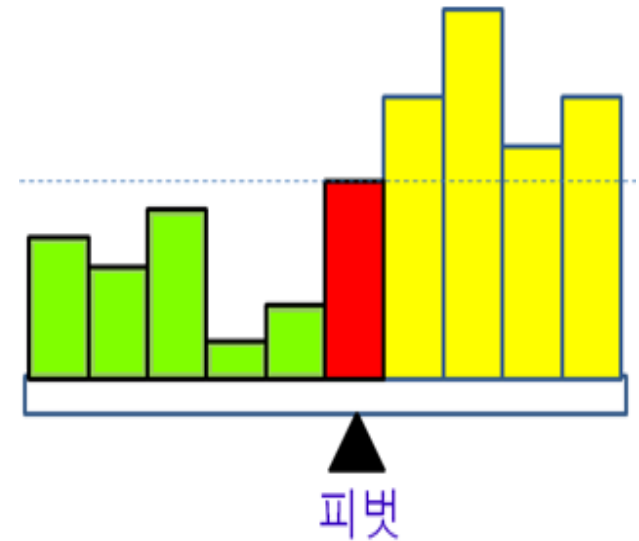
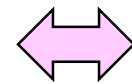
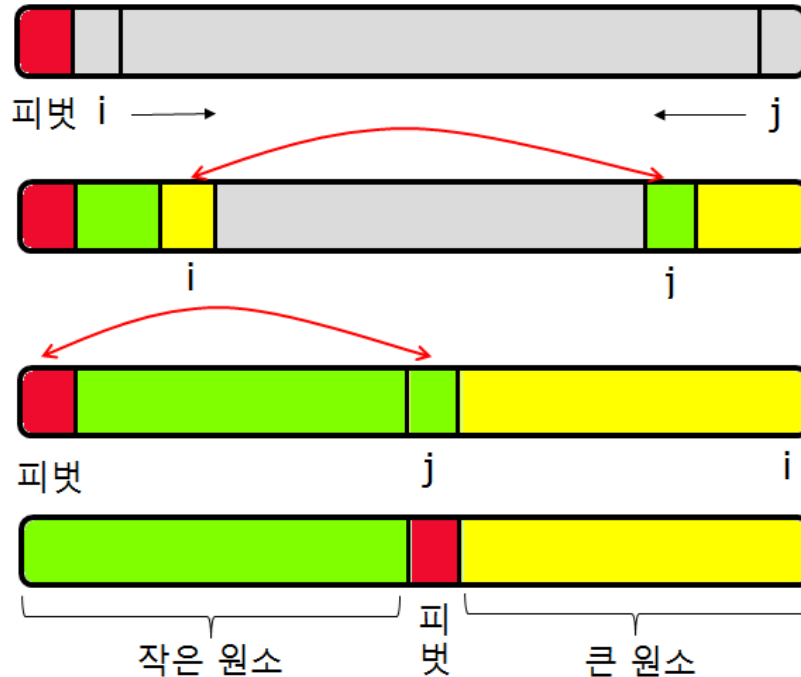


수행 시간

- 반복 합병 정렬의 수행 시간: 합병 정렬과 동일한 $O(N\log N)$
- [단점] 입력 크기의 보조 리스트 사용
- 대부분의 정렬알고리즘들은 보조 메모리없이 입력에서 정렬을 수행한다. 이러한 알고리즘을 **In-place 알고리즘**이라고 한다.
- 보조 메모리를 사용하지 않는 합병 정렬 알고리즘도 연구된 바 있으나 알고리즘이 너무 복잡하여 실질적인 효용 가치는 없음
- 합병 정렬은 자바 (Standard Edition 6) 객체 정렬에서 **시스템 sort**로 활용

8.6 퀵정렬

- **퀵정렬(Quick Sort)**은 입력의 맨 왼쪽 원소(피벗, Pivot)를 기준으로 피벗보다 작은 원소들과 큰 원소들을 각각 피벗의 좌우로 분할한 후, 피벗보다 작은 원소들과 피벗보다 큰 원소들을 각각 재귀적으로 정렬하는 알고리즘



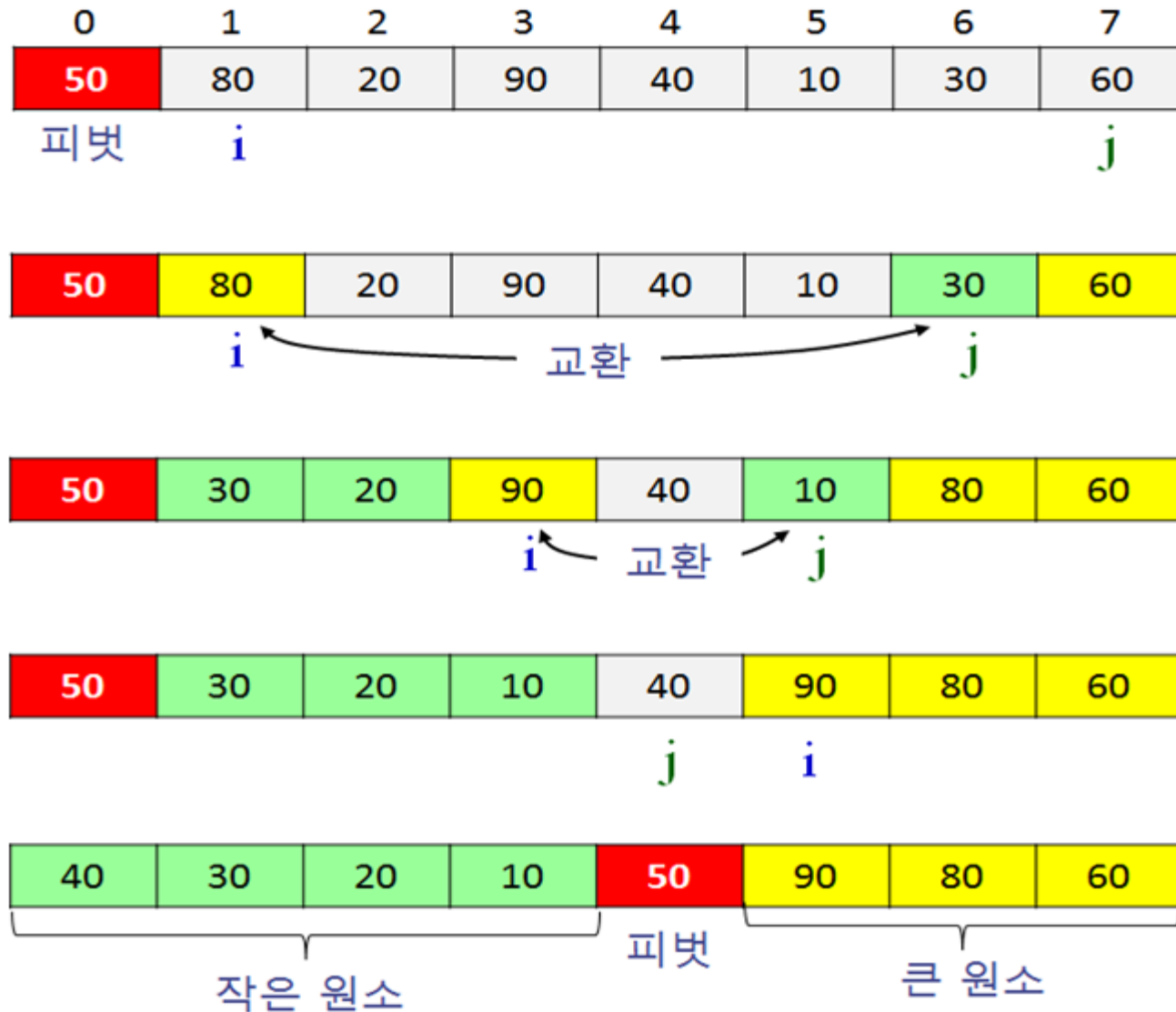
```

01 def qsort(a, low, high):
02     if low < high:
03         pivot = partition(a, low, high)
04         qsort(a, low, pivot-1)
05         qsort(a, pivot+1, high)
06
07 def partition(a, pivot, high):
08     i = pivot + 1
09     j = high
10     while True:
11         while i < high and a[i] < a[pivot]:
12             i += 1
13         while j > pivot and a[j] > a[pivot]:
14             j -= 1
15         if j <= i:
16             break
17         a[i], a[j] = a[j], a[i]
18         i += 1
19         j -= 1
20
21     a[pivot], a[j] = a[j], a[pivot]
22     return j
23
24 a = [54, 88, 77, 26, 93, 17, 49, 10, 17, 77, 11, 31, 22, 44, 17, 20]
25 print('정렬 전:\t', a)
26 qsort(a, 0, len(a)-1)
27 print('정렬 후:\t', a)

```


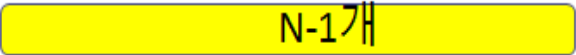


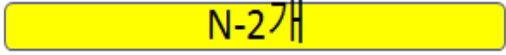
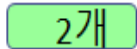

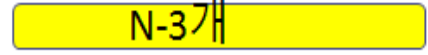
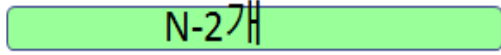


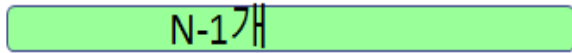

피벗을 기준으로 분할
 앞/뒷부분 재귀호출
 a[i]가 피벗보다 작으면 i를 1 증가
 a[j]가 피벗보다 크면 j를 1 감소
 루프 중단
 a[i]와 a[j] 교환
 a[j]와 피벗 교환
 피벗 인덱스
 퀵정렬 호출

[예제] 피벗인 50으로 partition()을 호출했을 때 수행 과정



수행 시간

- **최선 경우**: 피벗이 매번 입력을 1/2씩 분할을 하는 경우 $T(N) = 2T(N/2) + cN$, $T(1) = O(1)$ 로 합병 정렬의 수행 시간과 동일. 여기서 c 는 각각 상수
- **평균 경우**: 피벗이 입력을 다음과 같이 분할할 확률이 모두 같을 때, $T(N) = O(N \log N)$ 으로 계산됨

피벗	확률
 	1/N
  	1/N
  	1/N
⋮	
  	1/N
 	1/N

- 최악 경우: 피벗이 매번 가장 작은 경우 또는 가장 클 경우로 피벗보다 작은 부분이나 큰 부분이 없을 때
- 따라서 $T(N) = T(N-1) + N-1$, $T(1) = 0$

$$\begin{aligned}
 T(N) &= T(N-1) + N-1 = [T((N-1)-1) + (N-1)-1] + N-1 \\
 &= T(N-2) + N-2 + N-1 \\
 &= T(N-3) + N-3 + N-2 + N-1 \\
 &\quad \dots \\
 &= T(1) + 1 + 2 + \dots + N-3 + N-2 + N-1, \quad T(1) = 0 \\
 &= N(N-1)/2 = O(N^2)
 \end{aligned}$$

성능 향상 방법[1]

- 퀵정렬은 재귀 호출을 사용하므로 입력이 작은 크기가 되었을 때 삽입 정렬을 호출하여 성능 향상
- 크기 제한: CALLSIZE를 7~10 정도
- 다음과 같이 수정

```
02 if low < high:  
03     pivot = partition(a, low, high)  
04     qsort(a, low, pivot-1)  
05     qsort(a, pivot+1, high)
```



```
if high < low + CALL_SIZE:  
    insertion_sort(a, low, high)  
    return  
else:  
    pivot = partition(a, low, high)  
    qsort(a, low, pivot-1)  
    qsort(a, pivot+1, high)
```


성능 향상 방법[2]

- 퀵정렬은 피벗의 값에 따라 분할되는 두 영역의 크기가 결정되므로 한쪽이 너무 커지는 것을 방지하기 위해 랜덤하게 선택한 3 개의 원소들 중에서 중간값(Median)을 피벗으로 사용하여 성능 개선
- 이를 Median-of-Three 방법이라 함
- 가장 왼쪽(low), 중간(mid), 그리고 가장 오른쪽(high) 원소들 중에서 중간값을 찾는 것으로도 알려져 있음

성능 향상 방법[3]

- Tukey는 9 개의 원소들을 임의로 선택하여 이들을 3 개씩 하나의 그룹으로 만든 뒤, 각 그룹에서 중간값을 선택하고, 선택된 3 개의 중간값들에 대한 중간값을 피벗으로 사용하는 것을 제안
- Median-of-Three 방법보다 좋은 성능을 보임
- 다음 예제에서 60이 피벗이 됨

50 70 20 10 85 25 30 92 63 40 80 17 60 31 23 62 15 99

50 20 85 30 63 80 60 23 99

50

63

60

60

성능 향상 방법[4]

- 퀵정렬을 시작하기 전에 입력 배열에 대해 랜덤 섞기(Random Shuffling)를 수행
- 치우친 분할이 일어나는 것을 확률적으로 방지
- Knuth의 $O(N)$ 시간 random_shuffle 사용

```
def rand_shuffle(a):  
    N = len(a)  
    for i in range(N):  
        r = random.randint(0, i)  
        a[i], a[r] = a[r], a[i]
```

참고로 파이썬의 random.shuffle()은 균등분포를 보장하지 않는다.

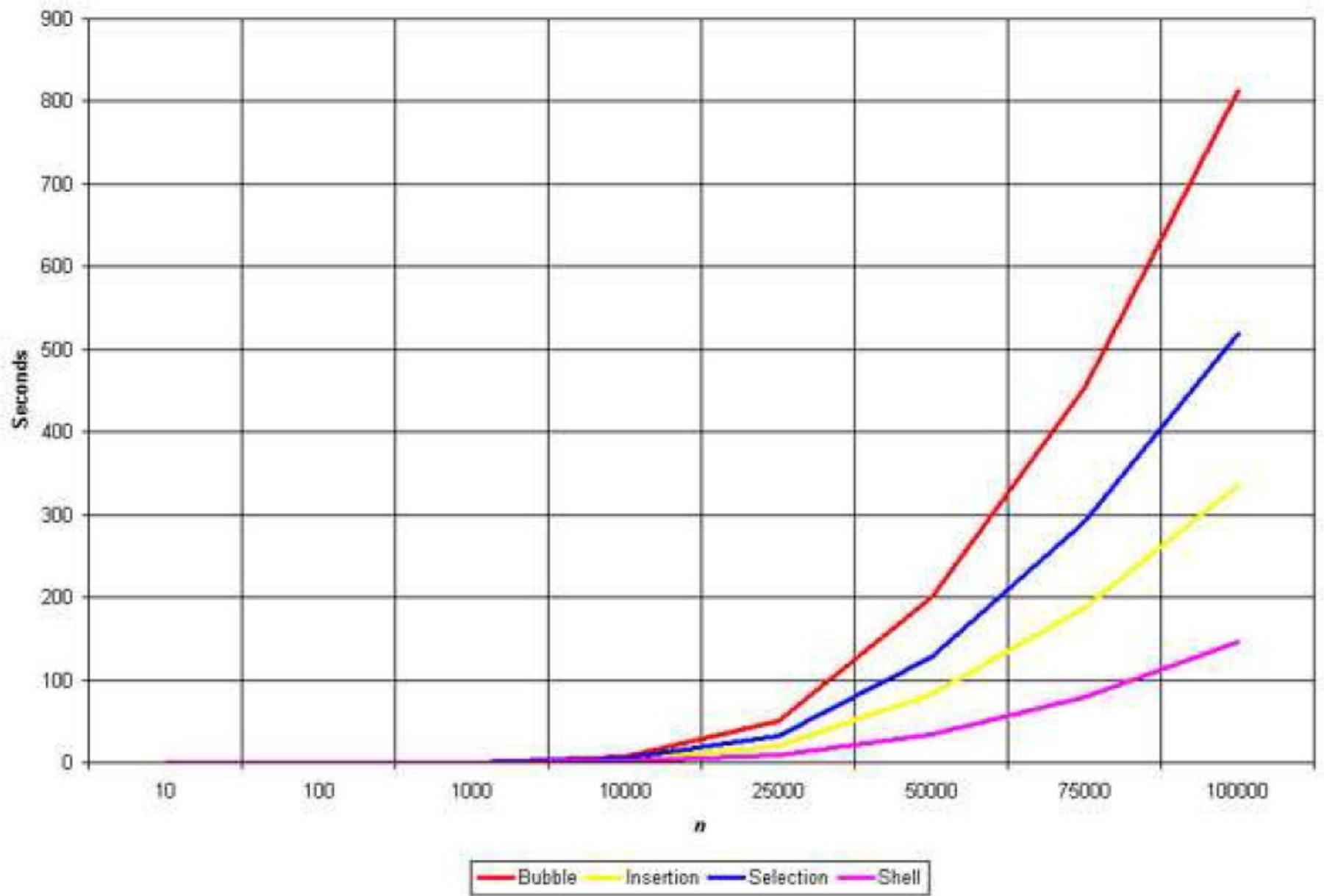


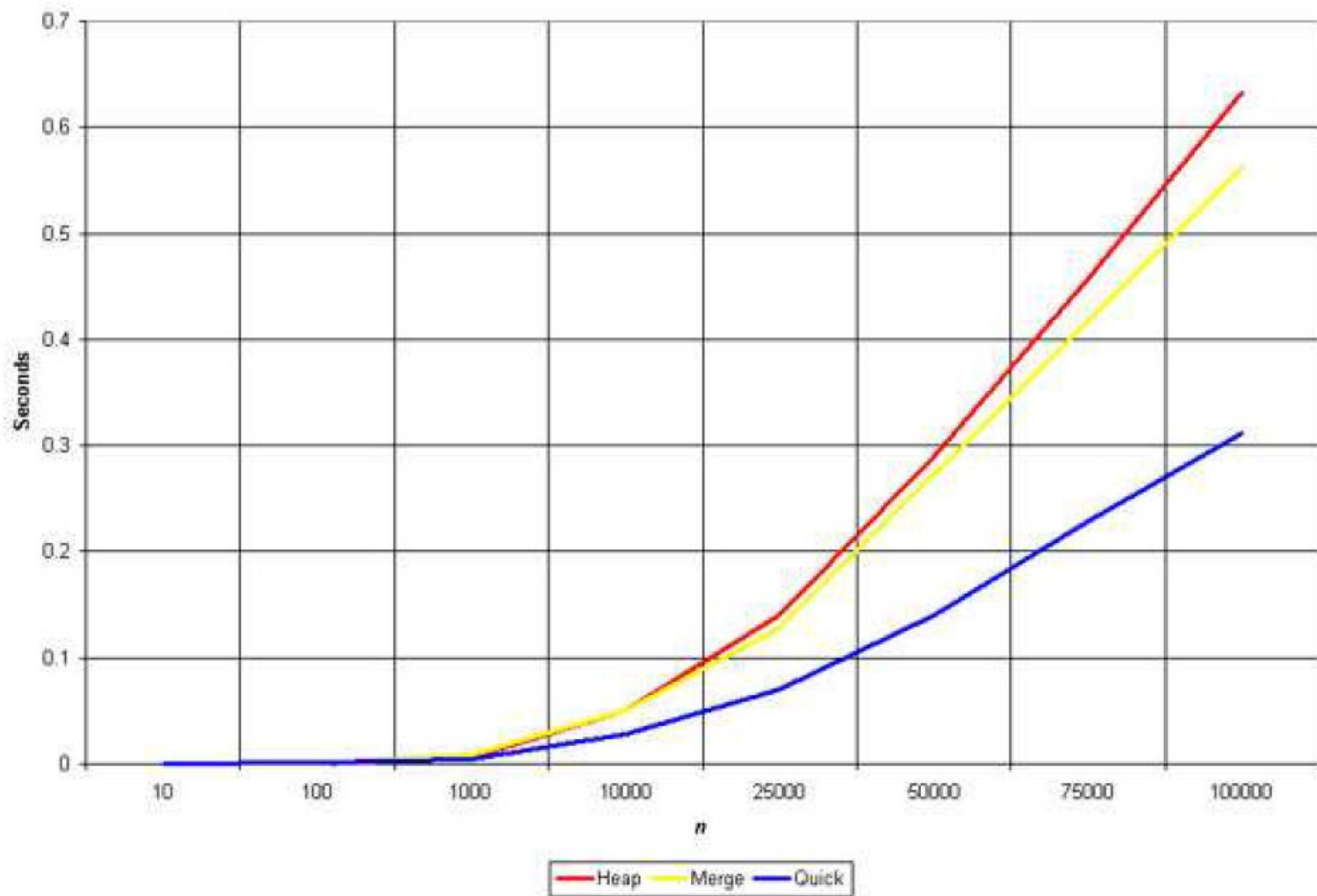
- 퀵정렬은 평균적으로 빠른 수행 시간을 가지며, 보조 메모리를 사용하지 않음
- 최악 경우 수행시간이 $O(N^2)$ 이므로, 성능 향상 방법들을 적용하여 사용하는 것이 바람직함
- 퀵정렬은 원시 타입(Primitive Type) 데이터를 정렬하는 자바 Standard Edition 6의 시스템 sort에 사용
- C-언어 라이브러리의 qsort, 그리고 Unix, g++, Visual C++ 등에서도 퀵정렬을 시스템 정렬로 사용
- 자바 SE 7에서는 2009년에 Yaroslavskiy가 고안한 이중피벗퀵(Dual Pivot Quick)정렬이 사용[부록 III]

정렬 알고리즘 성능 비교

	최선경우	평균경우	최악경우	추가공간	안정성
선택정렬	N^2	N^2	N^2	$O(1)$	X
삽입정렬	N	N^2	N^2	$O(1)$	O
셸정렬	$N \log N$?	$N^{1.5}$	$O(1)$	X
힙정렬	$N \log N$	$N \log N$	$N \log N$	$O(1)$	X
합병정렬	$N \log N$	$N \log N$	$N \log N$	N	O
퀵정렬	$N \log N$	$N \log N$	N^2	$O(1)$	X
Tim Sort	N	$N \log N$	$N \log N$	N	O

Tim Sort에 대해 보다 상세한 설명 [부록 III]





정렬의 하한

- 정렬 문제 자체를 해결하기 위해 원소들의 최소 비교 횟수는 얼마일까? 단, 원소의 비교는 반드시 원소 대 원소의 크기를 비교하는 것으로 가정
- 이 때의 정렬 문제를 **비교 정렬(Comparison Sort)**이라고 한다.
- 부록 IV에서는 비교 정렬을 위한 최소 비교 횟수가 $\Omega(N \log N)$ 임을 보여준다. 즉, 어떠한 정렬 알고리즘이라도 최소 $\Omega(N \log N)$ 만큼의 원소 비교를 수행하지 않으면, 알고리즘의 결과가 항상 정렬되어 있다는 보장을 할 수 없다는 의미
- 참고로 7.7절의 기수정렬은 키의 비교를 부분적으로 수행하는 정렬

- **안정한 정렬(Stable Sort)** 알고리즘은 중복된 키에 대해 입력에서 앞서 있던 키가 정렬 후에도 앞서 있음
- [예제] 안정한 정렬 결과에서는 [20 B]와 [20 E]가 각각 입력 전과 후에 항상 상대적인 순서가 유지되지만, 불안정한 정렬 결과에서는 입력 전과 후에 그 순서가 뒤바뀜

정렬 전 [90 A] [20 B] [60 C] [40 D] [20 E] [60 F] [50 G] [10 H]

stable 정렬 [10 H] [20 B] [20 E] [40 D] [50 G] [60 C] [60 F] [90 A]

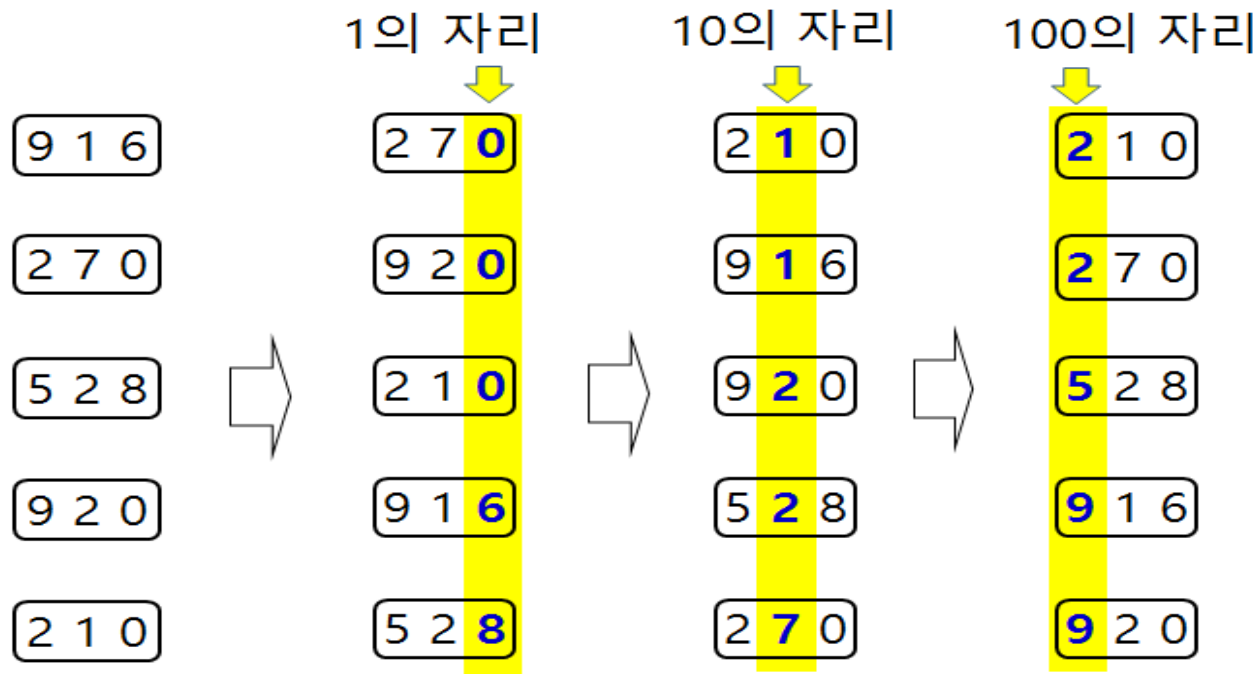
unstable 정렬 [10 H] [20 E] [20 B] [40 D] [50 G] [60 C] [60 F] [90 A]

7.7 기수 정렬

- 기수 정렬(Radix Sort)은 **키를 부분적으로 비교**하는 정렬
 - 키가 숫자로 되어있으면, 각 자릿수에 대해 키를 비교
 - **기(radix)**는 특정 진수를 나타내는 숫자들
10진수의 기 = 0, 1, 2, ..., 9, 2진수의 기 = 0, 1
- **LSD(Least Significant Digit) 기수 정렬**: 자릿수 비교를 최하위 숫자로부터 최상위 숫자 방향으로 정렬
- **MSD(Most Significant Digit) 기수 정렬**: 반대 방향으로 정렬

주어진 3 자리 십진수 키에 대한 LSD 기수 정렬

- 가장 먼저 각 키의 1의 자리만 비교하여 작은 수부터 큰 수로 정렬
- 그 다음에는 10의 자리만을 각각 비교하여 키들을 정렬
- 마지막으로 100의 자리 숫자만을 비교하여 정렬 종료



- LSD 기수 정렬을 위해서는 반드시 지켜야 할 순서가 있음
 - 앞 그림에서 10의 자리가 1인 210과 916이 있는데, 10의 자리에 대해 정렬할 때 210이 반드시 916 위에 위치하여야
 - 10의 자리가 같기 때문에 916이 210보다 위에 있어도 문제가 없어 보이지만, 그렇게 되면 1의 자리에 대해 정렬해 놓은 것이 아무 소용이 없게 됨
 - 따라서 LSD 기수 정렬은 안정성(Stability)이 반드시 유지되어야

3자리 십진수 키에 대한 LSD 기수 정렬 수행 과정

- a는 입력이고, t는 같은 크기의 보조 리스트이다.

a			t			a			t			a			t		
2	5	1	4	3	0	4	3	0	3	0	1	3	0	1	0	0	2
4	3	0	5	4	0	5	4	0	4	0	1	4	0	1	0	1	0
3	0	1	0	1	0	0	1	0	0	0	2	0	0	2	0	2	2
5	4	0	2	5	1	2	5	1	2	0	4	2	0	4	1	1	5
5	5	1	3	0	1	3	0	1	0	1	0	0	1	0	1	2	4
4	0	1	5	5	1	5	5	1	1	1	5	1	1	5	2	0	4
0	0	2	4	0	1	4	0	1	0	2	2	0	2	2	2	5	1
0	1	0	0	0	2	0	0	2	1	2	4	1	2	4	3	0	1
1	2	4	0	2	2	0	2	2	4	3	0	4	3	0	4	0	1
0	2	2	1	2	4	1	2	4	5	4	0	5	4	0	4	3	0
2	0	4	2	0	4	2	0	4	2	5	1	2	5	1	5	4	0
1	1	5	1	1	5	1	1	5	5	5	1	5	5	1	5	5	1

```

01 def lsd_sort(a):
02     WIDTH = 3
03     N = len(a)
04     R = 128
05     temp = [None] * N
06     for d in reversed(range(WIDTH)):
07         count = [0] * (R+1)
08         for i in range(N):
09             count[ord(a[i][d])+1] += 1
10         for j in range(1, R):
11             count[j] += count[j-1]
12         for i in range(N):
13             p = ord(a[i][d])
14             temp[count[p]] = a[i]
15             count[p] += 1
16         for i in range(N):
17             a[i] = temp[i]
18         print('%d번째 문자:\t ' % d, end='')
19         for x in a: print(x, ' ', end='')
20         print()
21 a = ['ICN', 'SFO', 'LAX', 'FRA', 'SIN', 'ROM', 'HKG', 'TLV',
22      'SYD', 'MEX', 'LHR', 'NRT', 'JFK', 'PEK', 'BER', 'MOW']
23 print('정렬 전:\t ', end='')
24 for x in a: print(x, ' ', end='')
25 print()
26 lsd_sort(a)

```

입력: 3개의 문자로 된
스트링

문자 종류 수(UTF-8, ASCII)

2, 1, 0 순으로

빈도수 계산

temp에 저장할 시작
인덱스 계산

d번째 문자를 기준으로
각 a[i]를 적절한 temp
원소로 복사

temp를 a로 복사

[프로그램 7-8의 수행 결과

Console PyUnit

<terminated> lsd_sort.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

정렬 전:	ICN	SFO	LAX	FRA	SIN	ROM	HKG	TLV	SYD	MEX	LHR	NRT	JFK	PEK	BER	MOW
2번째 문자:	FRA	SYD	HKG	JFK	PEK	ROM	ICN	SIN	SFO	LHR	BER	NRT	TLV	MOW	LAX	MEX
1번째 문자:	LAX	ICN	PEK	BER	MEX	JFK	SFO	LHR	SIN	HKG	TLV	ROM	MOW	FRA	NRT	SYD
0번째 문자:	BER	FRA	HKG	ICN	JFK	LAX	LHR	MEX	MOW	NRT	PEK	ROM	SFO	SIN	SYD	TLV

다음 슬라이드의 [그림 7-13]에 대해

- lsd_sort.py의 line 08과 10의 for-루프 수행에 대해, 입력 크기가 9이고 각 스트링은 1개의 문자로만 되어 있는 간단한 예제를 보여줌
- 이 예제는 [프로그램 7-8]을 쉽게 이해할 수 있도록 문자의 종류는 3개로서, A, B, C 만을 사용하였고, 입력 스트링의 길이가 1로서 d는 무시함

수행 시간

- LSD 기수 정렬의 수행시간은 $O(d(N+R))$

여기서 d 는 키의 자리 수이고, R 은 기(Radix)이며, N 은 입력의 크기

- $O(d(N+R))$ 인 이유: line 06의 바깥쪽의 for-루프는 d 회 수행되고, 각 자릿수에 대해 line 08, 12, 14의 for-루프들이 각각 N 번씩 수행되며, line 10의 for-루프는 R 회 수행되기 때문

장단점 및 응용

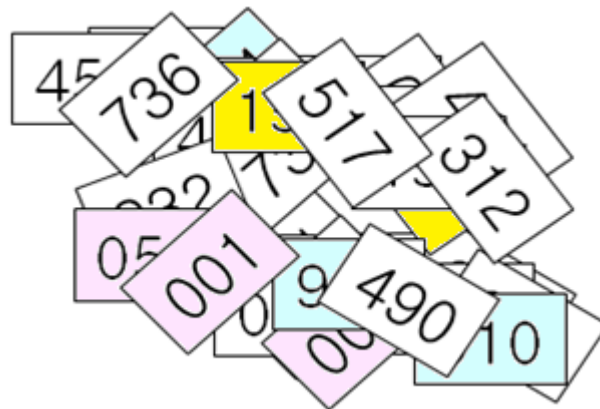
- LSD 기수 정렬은 제한적인 범위 내에 있는 숫자(문자)에 대해서 좋은 성능을 보임
 - 인터넷 주소, 계좌번호, 날짜, 주민등록번호 등을 정렬할 때 매우 효율적
- 기수 정렬은 범용 정렬알고리즘이 아님
 - 입력의 형태 따라 알고리즘을 수정해야 할 여지가 있으므로 일반적인 시스템 라이브러리에서 활용되지 않음
- 선형 크기의 추가 메모리를 필요
- 입력 크기가 커질수록 캐시메모리를 비효율적 사용
- 루프 내에 명령어(코드)가 많음

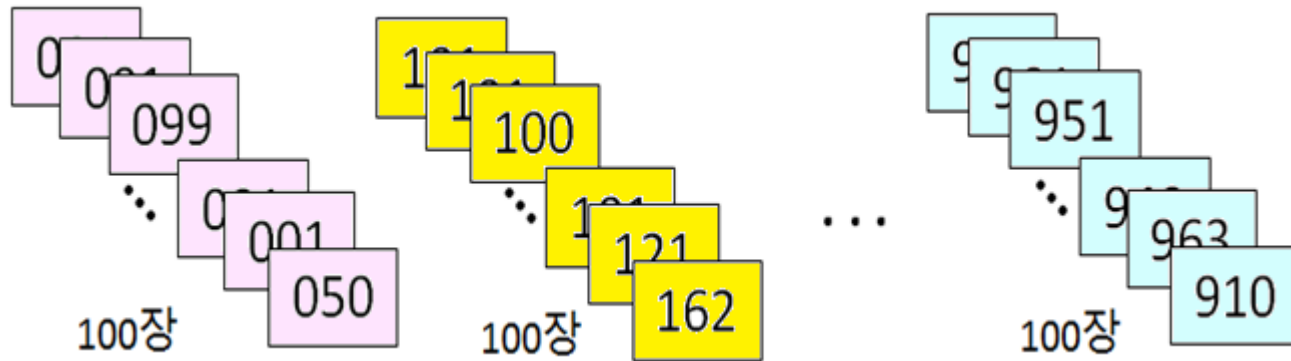
Applications

- GPU(Graphics Processing Unit) 기반 병렬(Parallel) 정렬의 경우 LSD 기수 정렬을 병렬처리 할 수 있도록 구현하여 시스템 sort로 사용
- Thrust Library of Parallel Primitives, v.1.3.0의 시스템 sort로 사용

MSD(Most Significant Digit) 기수 정렬

- 1,000장의 카드에 000부터 999까지 각각 다른 숫자가 적혀 있고, 이 카드들이 섞여있다. 이를 어떻게 정렬해야 할까?





- 먼저 카드를 한 장씩 보고 100자리의 숫자에 따라 읽은 카드를 분류하여 10 개의 더미를 만든다.
- 각각의 더미에 대해 10의 자리 숫자만을 보고 마찬가지로 10 개의 작은 더미를 만들고,
- 마지막으로 각각의 작은 더미에서는 카드의 1의 자리를 보고 정렬하여 각각의 더미를 차례로 모아 정렬

MSD 기수 정렬

최상위 자릿수부터 최하위 자릿수 순으로 정렬하는 과정

a			t			a			t			a			t		
1	5	1	0	2	2	0	2	2	0	0	7	0	0	7	0	0	7
4	3	9	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
4	9	1	0	0	7	0	0	7	0	2	2	0	2	2	0	2	2
5	2	0	1	5	1	1	5	1	1	0	8	1	0	8	1	0	5
4	3	7	1	2	4	1	2	4	1	0	5	1	0	5	1	0	8
0	2	2	1	0	8	1	0	8	1	2	4	1	2	4	1	2	4
4	3	0	1	0	5	1	0	5	1	5	1	1	5	1	1	5	1
0	1	0	4	3	9	4	3	9	4	3	9	4	3	9	4	3	0
1	2	4	4	9	1	4	9	1	4	3	7	4	3	7	4	3	7
0	0	7	4	3	7	4	3	7	4	3	0	4	3	0	4	3	9
1	0	8	4	3	0	4	3	0	4	9	1	4	9	1	4	9	1
1	0	5	5	2	0	5	2	0	5	2	0	5	2	0	5	2	0

- MSD 기수 정렬은 입력의 최상위 자릿수에 대해 정렬한 후에 배열을 0으로 시작되는 키들, 1로 시작되는 키들, ..., 9로 시작되는 키들에 대해 각각 차례로 재귀 호출
- 그 다음 자릿수에 대해서도 동일한 방식으로 정렬이 진행



수행 시간

- MSD 기수 정렬의 수행 시간은 $O(d(N+R))$
 - LSD 기수 정렬의 수행 시간과 동일한데 LSD 기수 정렬이 수행 방향만 반대이기 때문
- 최하위 자릿수로 갈수록 너무 많은 수의 재귀 호출 발생
 - 재귀호출 시 입력 크기가 작아지면 삽입 정렬 사용

Applications

- 키의 앞부분(Prefix)만으로 정렬하는 경우 매우 좋은 성능을 보임
 - 전화번호를 지역 번호 기준으로 정렬하기, 생년월일을 년도 별로 정렬하기, IP 주소를 첫 8-비트를 기준으로 정렬하기, 항공기 도착시간 또는 출발시간을 기준으로 정렬하기 등

[그림 7-13] 빈도수 계산과 빈도수 누적 계산

```
08 for i in range(N):  
09     count[ord(a[i][d])+1] += 1
```

ord('A')=0
ord('B')=1
ord('C')=2

0번째 문자 = A
1번째 문자 = B
2번째 문자 = C

	a
0	B
1	A
2	C
3	A
4	C
5	B
6	B
7	A
8	B

빈도수 계산

count

0	0
1	3
2	4
3	2

count

0	0
1	3
2	7
3	9

i번째 문자를 저장할
시작 인덱스, i=0,1,2

3번째 문자는 없으므로
count[3]은 사용 안함

temp에 저장할 시작 인덱스 계산

```
10 for j in range(1, R):  
11     count[j] += count[j-1]
```

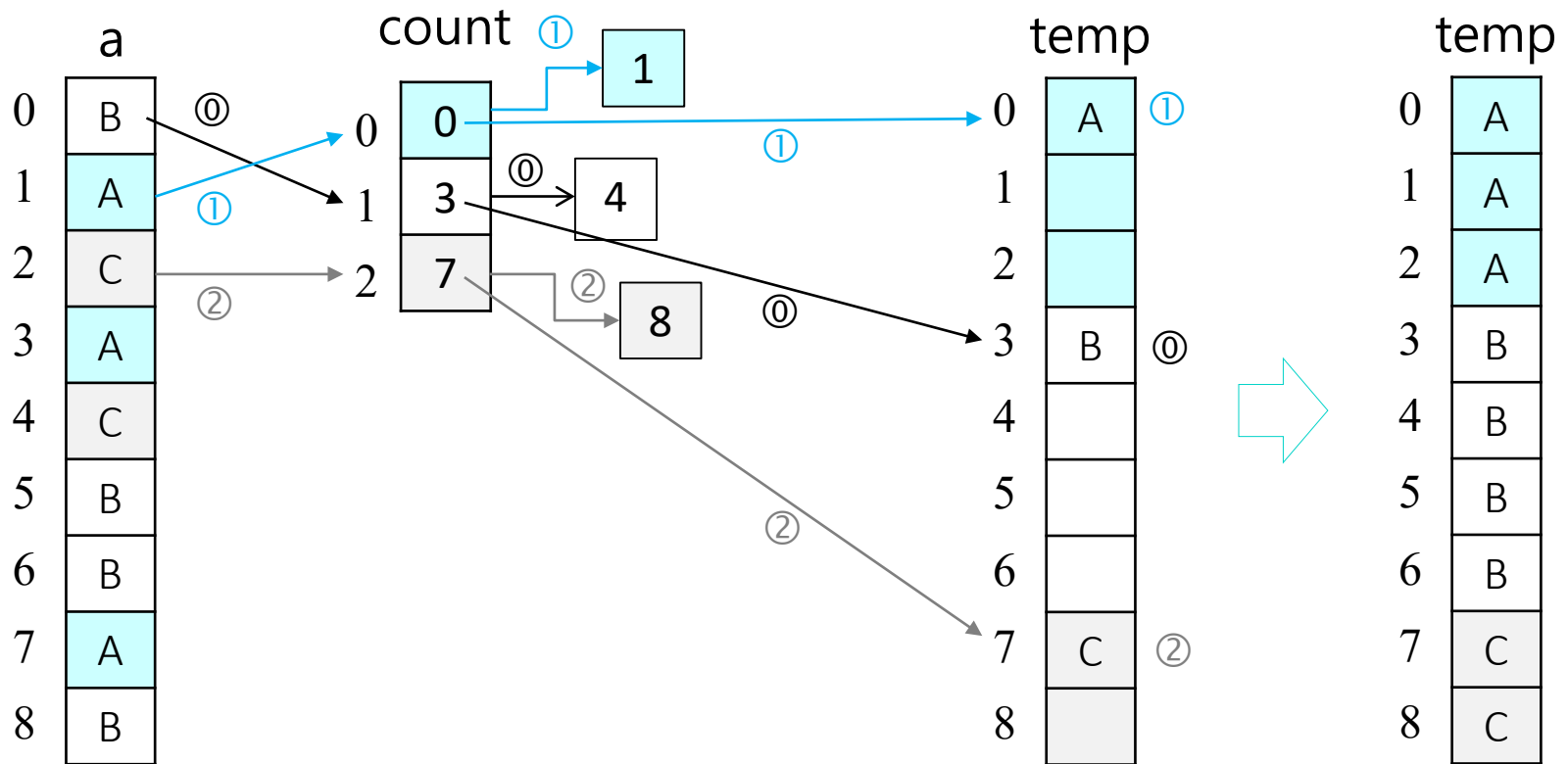
- 입력 리스트 a에 A가 3개, B가 4개, C가 2개 있는데, 각 문자의 빈도수는 count 리스트에 **한 칸씩 밀려서 저장**된 것에 유의
- 그 이유는 line 10의 for-루프에서 빈도수를 누적하여 계산된 값은 보조 리스트 temp의 인덱스로 사용하기 위함이다. 즉, 이 인덱스는 입력 리스트 a를 차례로 읽어가며, 즉, a[i]를 읽었을 때, a[i]를 저장할 temp의 인덱스이다.

```

12 for i in range(N):
13     p = ord(a[i][d])
14     temp[count[p]] = a[i]
15     count[p] += 1

```

ord('A')=0
ord('B')=1
ord('C')=2



[그림 7-14]

- [그림 7-14]는 line 12의 for-루프로서 입력 리스트 $a[i]$ 를 차례로 읽어가며, count 리스트를 사용하여 $a[i]$ 를 적절한 temp 원소에 저장
- $a[0]$ 인 B를 읽고, B의 ord 값에 대응되는 count 원소에 저장된 값이 바로 B가 저장되어야 할 temp 리스트의 인덱스이다.
- 편의상 $\text{ord}('A') = 0$, $\text{ord}('B') = 1$, $\text{ord}('C') = 2$ 로 가정
- 참고로 $\text{ord}()$ 는 인자로 주어진 문자의 Unicode 값을 리턴한다. $\text{ord}('A') = 65$, $\text{ord}('B') = 66$, $\text{ord}('C') = 67$

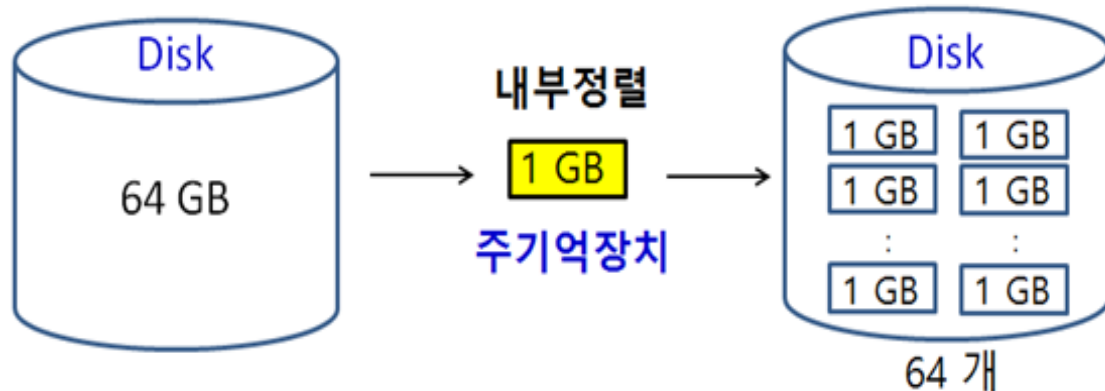
- 먼저 $a[0]$ 을 읽고, ④으로 표시된 화살표에 따라 $temp[3]$ 에 $a[0]$ 인 B를 저장하고 다음에 리스트 a에서 읽게 되는 B를 위해 $count[1]$ 의 원소를 1 증가시킨다.
- 그 다음엔 $a[1]$, 즉, A를 읽고, ①로 표시된 대로 $a[1]$ 을 $temp[0]$ 에 저장하고, $count[0]$ 을 1 증가시킨다.
- 이와 같이 $a[8]$ 까지 처리하면 맨 오른쪽의 temp 리스트를 얻는다.

7.8 외부 정렬

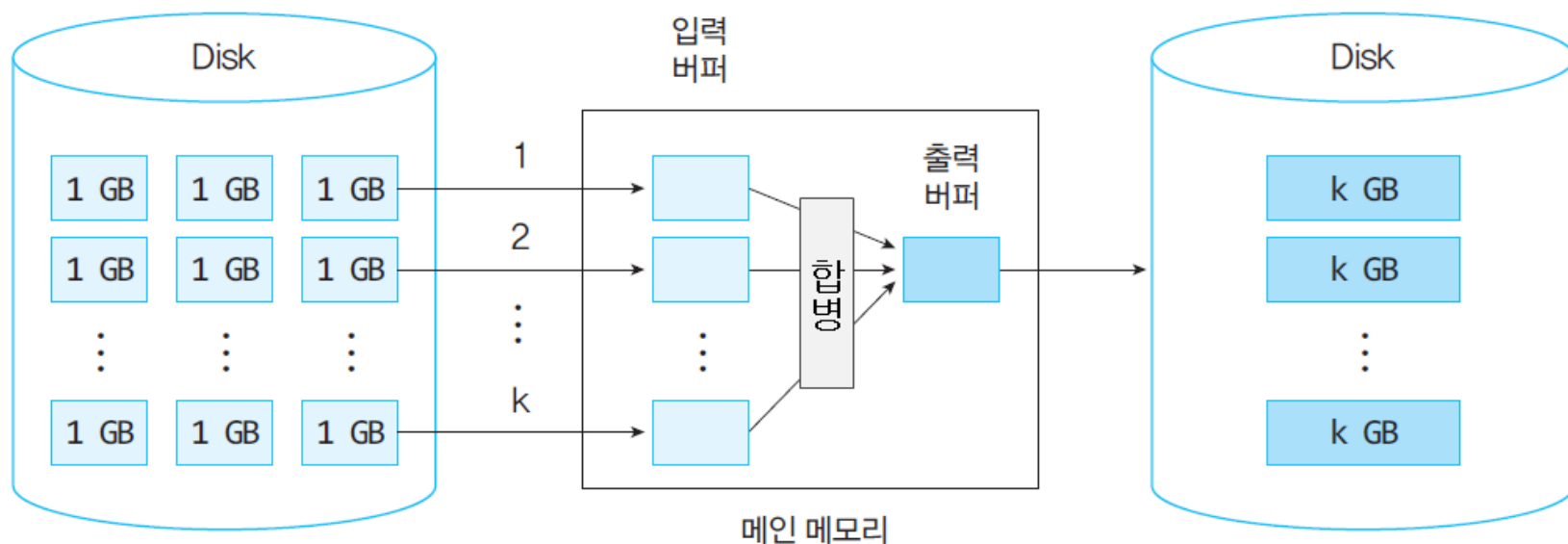
- 실세계에서는 대용량의 데이터를 하드디스크나 테이프와 같은 보조기억장치(또는 외부 메모리)에 저장한다.
- 내부정렬만으로는 보조기억장치에 저장된 대용량의 데이터를 정렬하기 어려움
- 외부 정렬(External Sort)이란 보조기억장치에 있는 대용량의 데이터를 정렬하는 알고리즘
- 기본적으로 합병(Merge)을 사용하여 정렬 수행
- 외부 정렬의 수행 시간: 원소의 비교 횟수가 아니라 입력 전체를 처리하는 횟수로 계산
 - 왜냐하면 보조기억장치의 접근 시간이 주기억장치의 접근시간보다 매우 느리기 때문
 - 패스(Pass)는 입력 전체를 처리하는 단위

- 보조 기억 장치 종류: 자기(Magnetic) 하드디스크와 테이프 외에도 SSD(Solid State Drive), 광학(Optical) 디스크, 플래시(Flash) 메모리 등

- 컴퓨터의 주기억장치에 데이터를 저장할 수 있는 용량이 1 GB (Gigabyte)이고, 입력 크기가 64 GB:
- 먼저 디스크로부터 주기억장치에 수용할 만큼의 입력 (1 GB)을 읽어 들여 내부 정렬 알고리즘을 사용하여 정렬하고, 그 결과를 디스크에 일단 다시 저장
- 이 과정을 반복하면, 원래의 입력이 64개의 정렬된 블록으로 분할되어 디스크에 저장됨
 - 정렬된 블록(데이터)을 **런(Run)**이라고 함



- 그 다음 과정은 블록들을 부분적으로 주기억장치의 입력 버퍼(Buffer)에 읽어 들여서, 합병을 수행하여 부분적으로 디스크에 쓰는 과정을 반복
- 그림은 1 GB블록들을 부분적으로 k개의 입력 버퍼로 읽어 들여 k GB 크기의 블록을 만드는 과정



- 입력 버퍼가 k 개 만큼 있으므로 k GB 블록이 총 $64/k$ 개 만들어짐
- 다음으로 k GB 블록을 k 개씩 짝지어 합병시키면, k^2 GB 블록 $64/k^2$ 개가 만들어짐
- 이 과정을 반복하여 계속 합병을 진행하면, 블록 크기는 k 배로 커지고 블록의 수는 $1/k$ 로 줄어들게 되어 결국에는 64 GB 블록 하나만 남음

[예제] $k = 2$ 이면

- 첫 번째 pass 후에 $64/2 = 32$ 개의 2 GB 블록이 만들어지고,
- 두 번째 pass 후에 $32/2 = 64/2^2 = 16$ 개의 4($=2^2$) GB 블록이 만들어지고,
- 세 번째 pass 후에 $16/2 = 8$ 개의 8 GB 블록이 만들어지고,
...
- 여섯 번째 pass 후에 64 GB 블록 하나만 남는다.

수행 시간

- 입력의 크기가 N 이고, 첫 pass에 N/M 개의 블록을 만들고, k 개의 블록을 하나의 블록으로 합병하는 방식으로 정렬을 수행하면, 정렬을 마칠 때까지 $\log_k(N/M)$ pass 가 필요
 - 계산 편의상 N 이 M 의 배수라고 가정. 즉, N/M 은 정수
- 정렬을 위해선 총 $\log_k(N/M) + 1$ 번의 pass가 필요

Applications

- 인터넷의 IP 주소, 통신/전화 회사의 전화번호, 은행에서의 고객/계좌, 기업의 물품/재고 데이터베이스, 인사 데이터베이스 등의 관리를 위해 사용되며, 일반적인 데이터베이스의 중복된 데이터를 제거하는 데에도 사용



요약

- **선택정렬**은 아직 정렬되지 않은 부분의 배열 원소들 중에서 최솟값을 선택하여 정렬된 부분의 바로 오른쪽 원소와 교환하는 정렬알고리즘
- **삽입정렬**은 수행과정 중에 배열이 정렬된 부분과 정렬되지 않은 부분으로 나뉘어지며, 정렬되지 않은 부분의 가장 왼쪽의 원소를 정렬된 부분에 삽입하는 방식의 정렬알고리즘
- **셸정렬**은 전처리과정을 추가한 삽입정렬이다.
전처리과정이란 작은 값을 가진 원소들을 배열의 앞부분으로 옮겨 큰 값을 가진 원소들이 배열의 뒷부분으로 이동

- **힙정렬**: 입력에 대해 최대힙을 만들어 루트노드와 힙의 가장 마지막 노드를 교환하고, 힙 크기를 1 감소시킨 후에 루트노드로부터 downheap을 수행하는 과정을 반복하여 정렬하는 알고리즘
- **합병정렬**: 입력을 반씩 두 개로 분할하고, 각각을 재귀적으로 합병정렬을 수행한 후, 두 개의 각각 정렬된 부분을 합병하는 정렬알고리즘
- **퀵정렬**: 피벗보다 작은 원소들과 큰 원소들을 각각 피벗의 좌우로 분할한 후, 피벗보다 작은 원소들과 피벗보다 큰 원소들을 각각 재귀적으로 정렬하는 알고리즘
- 원소 대 원소의 크기를 비교하는 **비교정렬의 하한은 $\Omega(N\log N)$**

- **기수정렬**: 키를 부분적으로 비교하는 정렬
LSD/MSD기수정렬의 수행시간은 $O(d(N+R))$
- **외부정렬**: 보조기억장치에 있는 대용량의 데이터를 정렬하는 알고리즘으로 합병을 사용하여 정렬 수행