제8장 정렬

정렬의 중요성

- **국세청에서 고용주와 고용인에게서 받은 신고서를 대조하는 경우**
  - 임의 정렬된 경우와 고용인의 주민번호로 정렬된 경우의 성능 차이

- **보간법(interpolation)에 의한 탐색**
  - 리스트가 정렬되었을 때, 짐작되는 위치에서부터 탐색
  - k를 a[i]와 비교하는 것으로 검색을 시작
  - $i = \frac{k - a[1].key}{a[n].key - a[1].key} * n$
  - 탐색의 성질은 리스트에 있는 키의 분포에 달려있음

- **리스트를 반복해서 탐색할 때**
  리스트를 정렬된 상태로 유지하는 것이 이롭다.

01	1
02	28
03	124
04	327
05	457
06	658
07	687
08	732
09	766
10	835
11	865
12	901
13	1078
14	1087
15	1187
16	1245
17	1298
18	1310
19	1347
20	1398

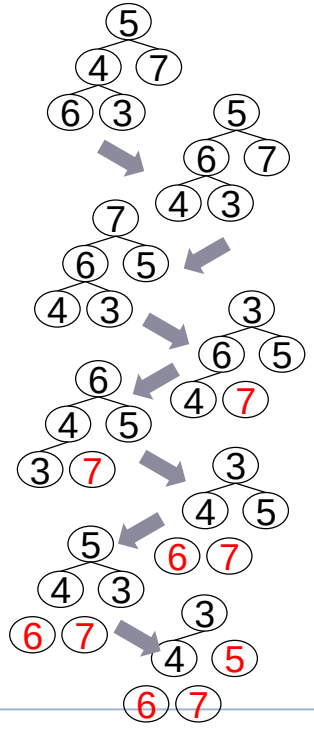
찾는 값 1187
i = (1187-1 / 1398-1) * 20 = 17
→ (1187-1 / 1245-1) * 16 = 15

정렬

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

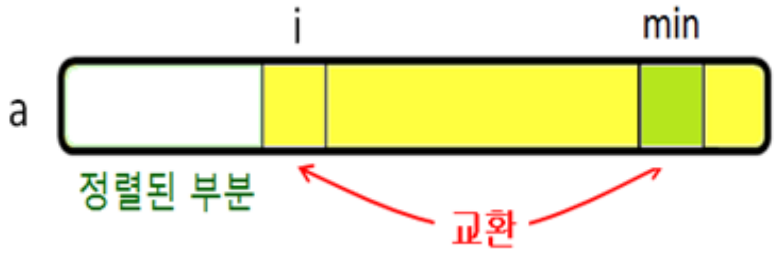
- 이중피벗퀵정렬(부록VI)
- Tim Sort (부록VI)

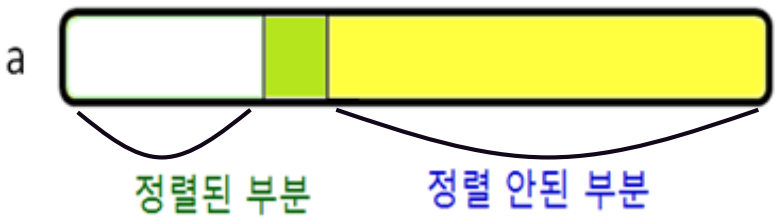
Sorting Algorithms

	Bubble	Selection	Insertion	Quick	Merge	Heap	Radix	Count
평균	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(nlgn)	O(nlgn)	O(nlgn)	O(d(n+r))	O(n)
최선	$O(n^2)$	$O(n^2)$	O(n)	O(nlgn)	O(n)	O(nlgn)	O(d(n+r))	O(n)
최악	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(nlgn)	O(nlgn)	O(d(n+r))	O(n)
방식	두 쌍의 값의 순서를 정렬하는 방식을 반복 적용	그 자리에 있을 값을 찾아 두 값을 교환	정렬된 리스트에 새 값을 자리 찾아 추가	pivot값 기준 큰값과 작은값으로 분할하는 과정을 반복	binary tree를 bottom-up 형태로 만들면서, 정렬된 리스트를 합병	max-heap의 delete-max를 반복 수행하여 큰 값부터 뒤로 정렬		
예제	5 4 7 6 3 ------ [1] ------ 5 4 7 3 6 5 4 3 7 6 5 3 4 7 6 3 5 4 7 6 ------ [2] ------ 3 5 4 6 7 3 5 4 6 7 3 4 5 6 7 ------ [3] ------ 3 4 5 6 7 3 4 5 6 7 ------ [4] ------ 3 4 5 6 7	5 4 7 6 3 ------ [1] ------ 5 4 7 3 6 3 4 7 5 6 ------ [2] ------ 3 4 7 5 6 3 4 7 5 6 ------ [3] ------ 3 4 7 5 6 3 4 5 7 6 ------ [4] ------ 3 4 5 7 6 3 4 5 6 7	5 4 7 6 3 ------ [1] ------ 5 4 7 6 3 5 7 6 3 4 5 7 6 3 ------ [2] ------ 4 5 7 6 3 ------ [3] ------ 4 5 7 6 3 4 5 7 3 4 5 6 7 3 ------ [4] ------ 4 5 6 7 3 4 5 6 7 4 5 6 7 4 5 6 7 3 4 5 6 7	5 4 7 6 3 ------ [1] ------ 5 4 7 6 3 3 4 7 6 5 ------ [2] ------ [3][4 7 6 5] [3][4 7 6 5] [3][4 5 6 7] ------ [3] ------ [3][4][5][6 7] [3][4][5][6 7] [3][4][5][6][7]	5 4 7 6 3 ------ [1] ------ [5][4][7][6][3] [4 5][6 7][3] ------ [2] ------ [4 5 6 7][3] ------ [3] ------ [3 4 5 6 7]			

8.1 선택정렬(Selection Sort)

- **선택정렬은 배열에서 아직 정렬되지 않은 부분의 원소들 중에서 최솟값을 '선택'하여 정렬된 부분의 바로 오른쪽 원소와 교환하는 정렬알고리즘**





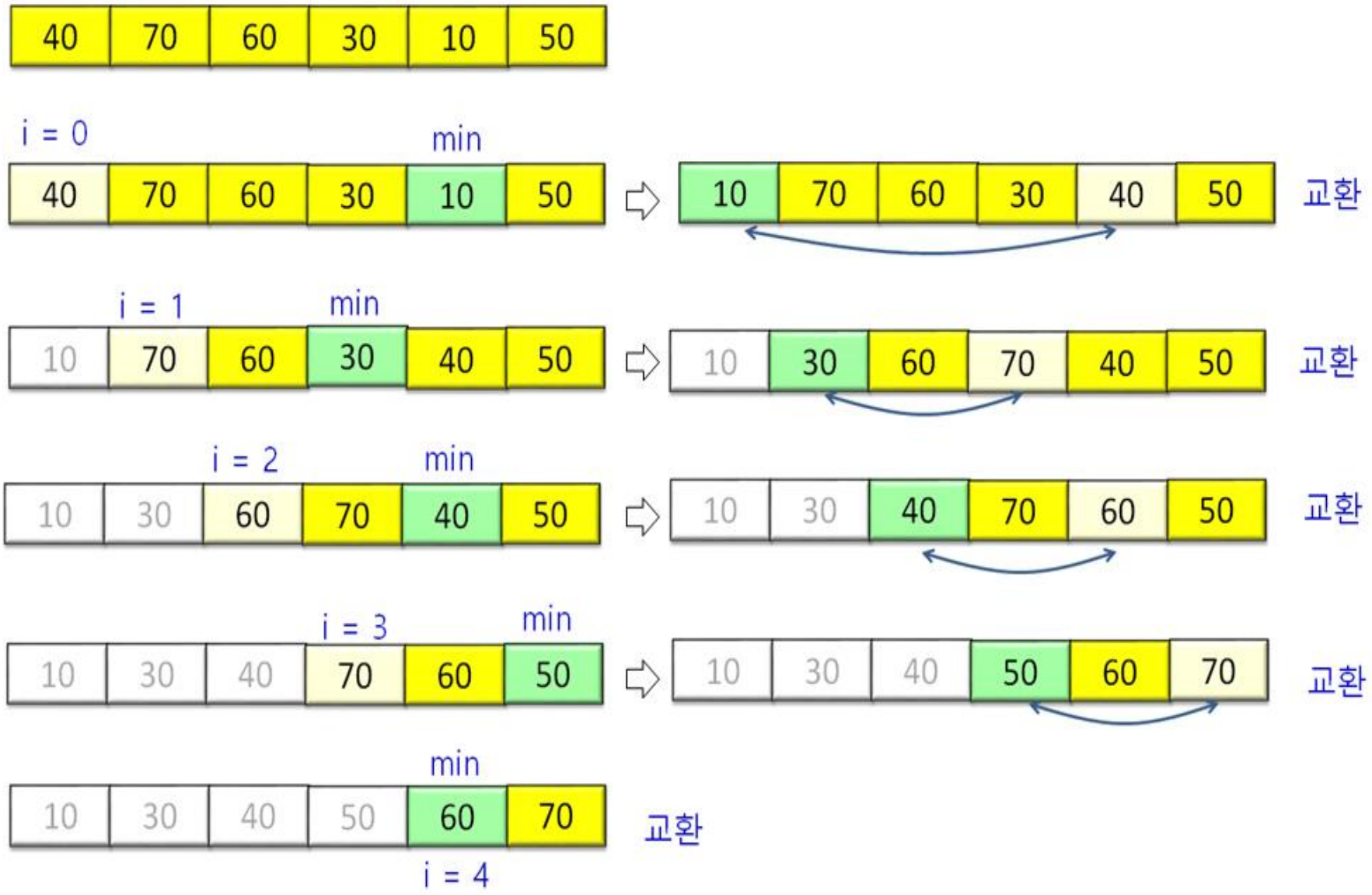
  - (a) 배열 a의 왼쪽 부분은 이미 정렬되어 있고 나머지 부분은 정렬 안된 부분
    - 정렬된 부분의 키들은 오른쪽의 정렬되지 않은 부분의 어떤 키보다 크지 않다.
    - 선택정렬은 항상 정렬 안된 부분에서 최솟값(min)을 찾아 왼쪽의 정렬된 부분의 바로 오른쪽 원소(현재 원소)로 옮기기 때문
  - 이 과정은 그림(a)에서 min을 a[i]와 교환 후에 (b)와 같이 i를 1 증가시키며, 이를 반복적으로 수행

Selection 클래스

```
01 import java.lang.Comparable;
02 public class Selection {
03     public static void sort(Comparable[] a) {
04         int N = a.length;
05         for (int i = 0; i < N; i++) {
06             int min = i;
07             for (int j = i+1; j < N; j++) {  // min 찾기
08                 if (isless(a[j], a[min])) min = j;
09             }
10             swap(a, i, min); // min과 a[i]의 교환
11         }
12     }
13      private static boolean isless(Comparable i, Comparable j) { // 키 비교
14         return (i.compareTo(j) < 0);
15     }
16     private static void swap(Comparable[] a, int i, int j) {    // 원소 교환
17         Comparable temp = a[i];
18         a[i] = a[j];
19         a[j] = temp;
20     }
21 }
```

- Line 01: compareTo() 메소드를 사용하여 키들을 비교하기 위해 Comparable 인터페이스를 import한다.
- Line 05: for-루프는 i가 0부터 N-1까지 변하면서, line 06~09에서 찾은 min에 대해 line 10에서 a[i]와a[min]을 교환

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



수행시간

- **선택정렬은 루프가 1 번 수행될 때마다 정렬되지 않은 부분에서 가장 작은 원소를 선택**
  - 처음 루프 수행에서 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**

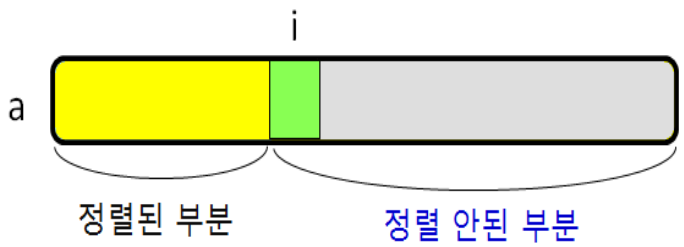
$$(N-1) + (N-2) + (N-3) + \cdots + 2 + 1 = \frac{N(N-1)}{2} = O(N^2)$$

선택정렬의 특징

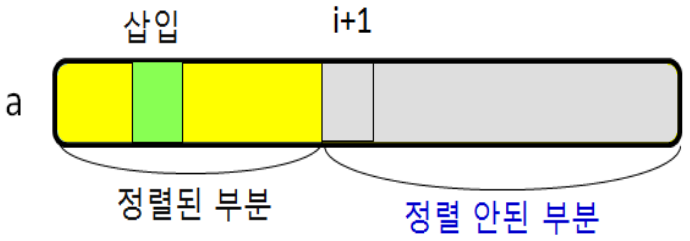
- **입력에 민감하지 않음(Input Insensitive)**
  - 항상 $O(N^2)$ 수행시간이 소요
- **최솟값을 찾은 후 원소를 교환하는 횟수가 N-1**
  - 이는 정렬알고리즘들 중에서 가장 작은 (최악 경우) 교환 횟수
- **하지만 선택정렬은 효율성 측면에서 뒤떨어지므로 거의 활용되지 않음**

8.2 삽입정렬 (Insertion Sort)

- **배열이 정렬된 부분과 정렬되지 않은 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

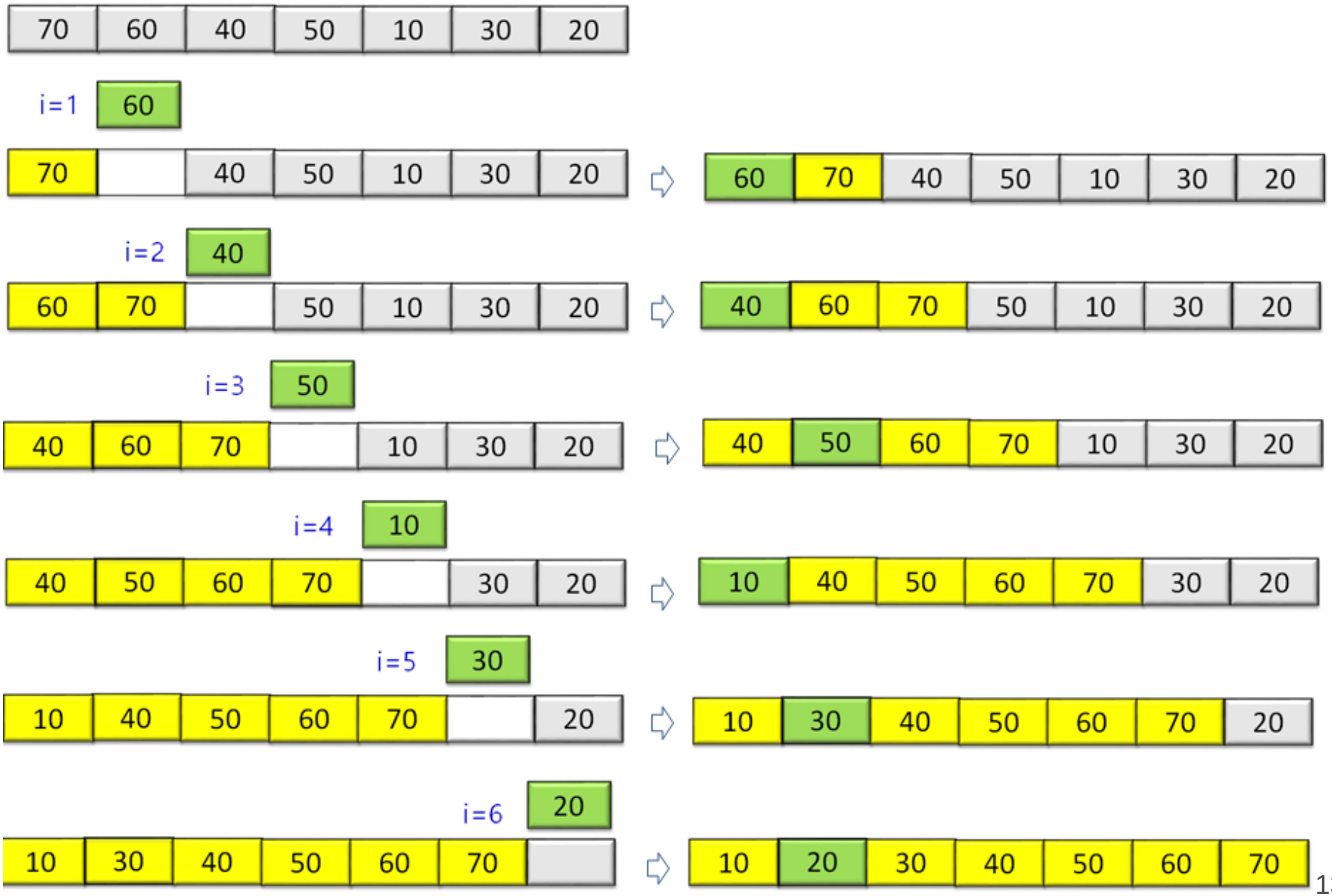
현재 원소인 50을 정렬된 부분에 삽입하는 과정

Insertion 클래스

```
01 import java.lang.Comparable;
02 public class Insertion {
03     public static void sort(Comparable[] a) {
04         int N = a.length;
05         for (int i = 1; i < N; i++) {   // i는 현재 원소의 인덱스
06             for (int j = i; j > 0; j--) {  // 현재 원소를 정렬된 앞부분에 삽입
07                 if ( isLess(a[j], a[j-1]) )
08                   swap(a, j, j-1);
09                 else break;
10             }
11         }
12     }
     // isLess(), swap() 메소드 선언, Selection 클래스와 동일함
   }
```

- Line 05: for-루프는 i를 1부터 N-1까지 변화시키며,
- Line 06~10: 현재 원소인 a[i]를 정렬된 앞 부분(a[0]~a[i-1])에 삽입

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



수행시간

- **삽입정렬은 입력에 민감 (Input Sensitive)**
- **입력이 이미 정렬된 경우(최선 경우)**
  - N-1번 비교하면 정렬을 마침 = O(N)
- **입력이 역으로 정렬된 경우 (최악 경우)**
  - $1 + 2 + \cdots + (N-2) + (N-1) = \frac{N(N-1)}{2} \approx \frac{1}{2}N^2 = O(N^2)$
- **최악 경우 데이터 교환 수: $O(N^2)$**
- **입력 데이터의 순서가 랜덤인 경우(평균경우)**
  - 현재 원소가 정렬된 앞 부분에 최종적으로 삽입되는 곳이 평균적으로 정렬된 부분의 중간이므로 $\frac{1}{2} \times \frac{N(N-1)}{2} \approx \frac{1}{4}N^2 = O(N^2)$

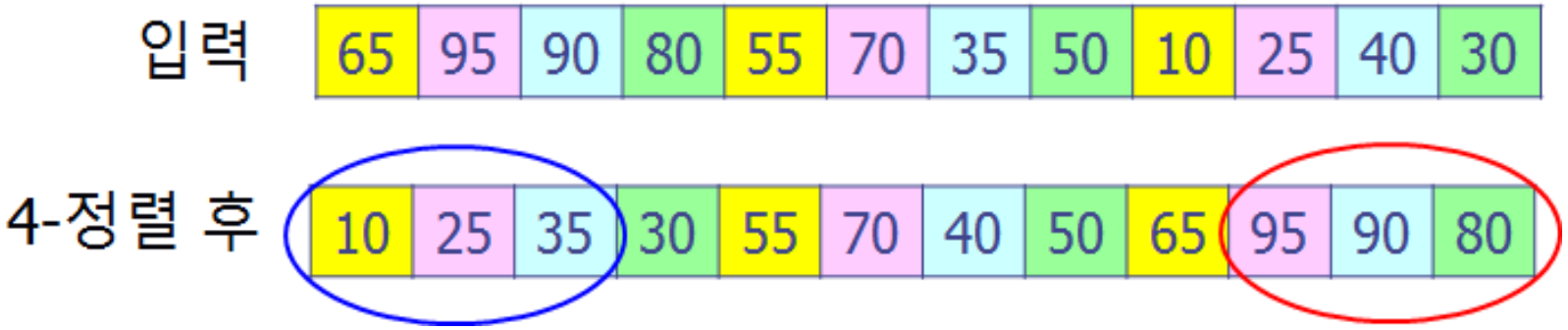
응용

- **이미 정렬된 파일의 뒷부분에 소량의 신규 데이터를 추가하여 정렬하는 경우(입력이 거의 정렬된 경우) 우수한 성능을 보임**
- **입력크기가 작은 경우에도 매우 좋은 성능을 보임**
  - 삽입정렬은 재귀호출을 하지 않으며, 프로그램도 매우 간단하기 때문
- **삽입정렬은 합병정렬이나 퀵정렬과 함께 사용되어 실질적으로 보다 빠른 성능에 도움을 줌**
- **단, 이론적인 수행시간은 향상되지 않음**

8.3 쉘정렬 (Shell Sort)

- **쉘정렬은 삽입정렬에 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 숫자들끼리 비교하며 한 자리씩 이동하는 단점 보완
  - 전처리과정은 여러 단계로 진행되며, 각 단계에서는 일정 간격으로 떨어진 원소들에 대해 삽입정렬 수행

전처리과정 전과 후



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

대표적인 간격의 순서(h-Sequence)

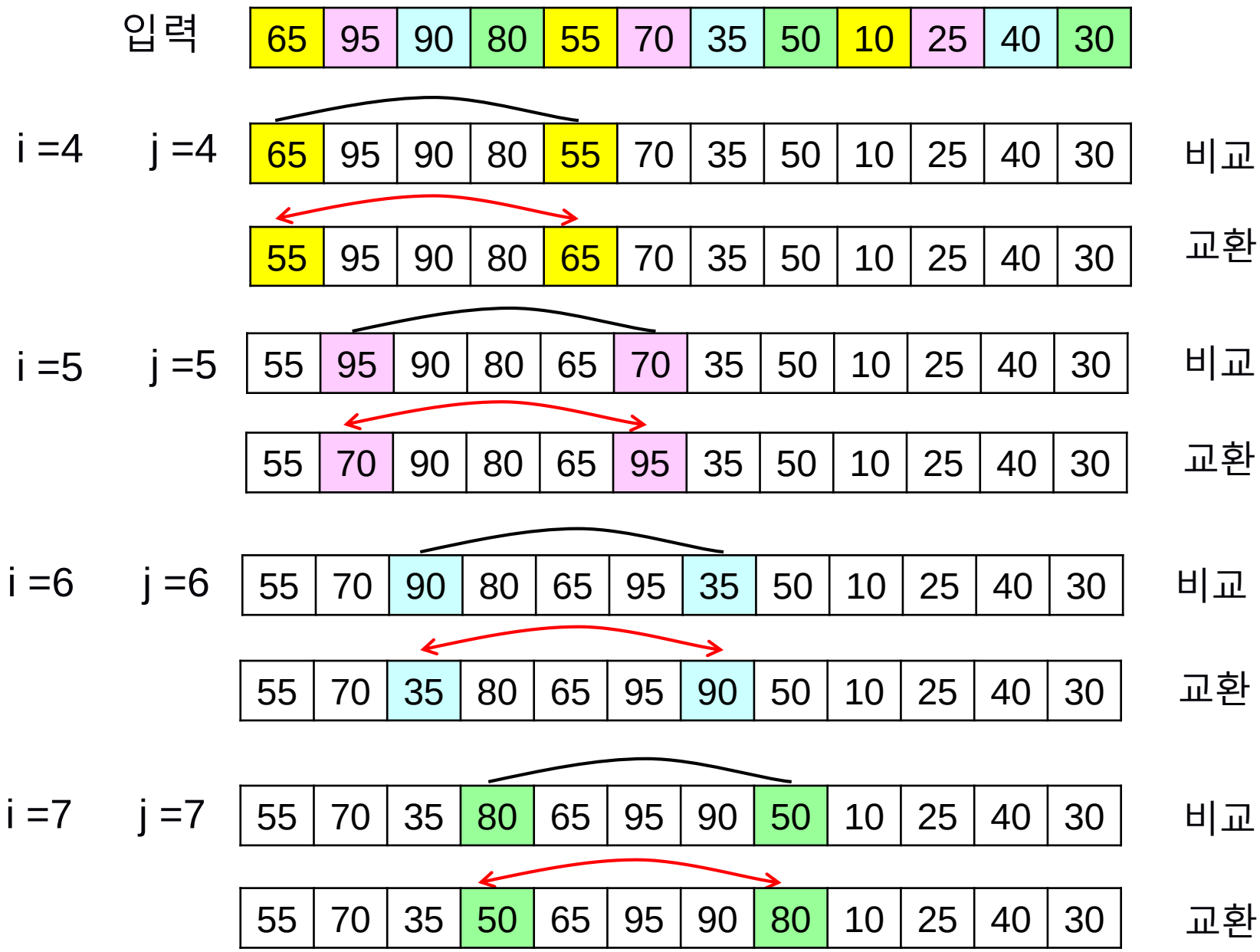
Shell	N/2, N/4, …, 1 (나누기 2를 계속하여 1이 될 때까지의 순서)
Hibbard	2^{k-1}, 2^{k-1}-1, …, 7, 3, 1
Knuth	$(3^k - 1)/2$, …, 13, 4, 1
Sedgewick	…, 109, 41, 19, 5, 1
Marcin Ciura	1750, 701, 301, 132, 57, 23, 10, 4, 1

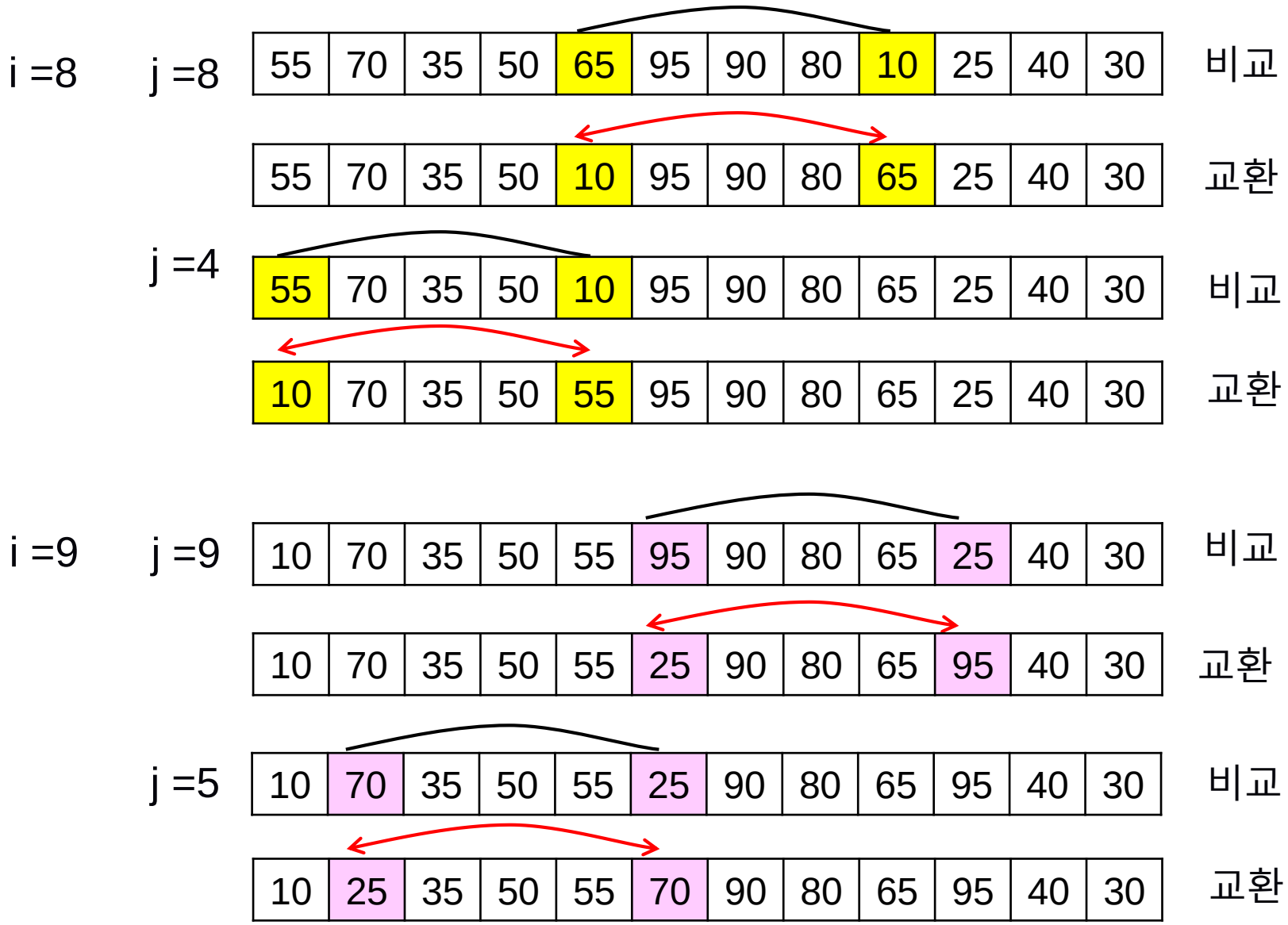
Shell 클래스

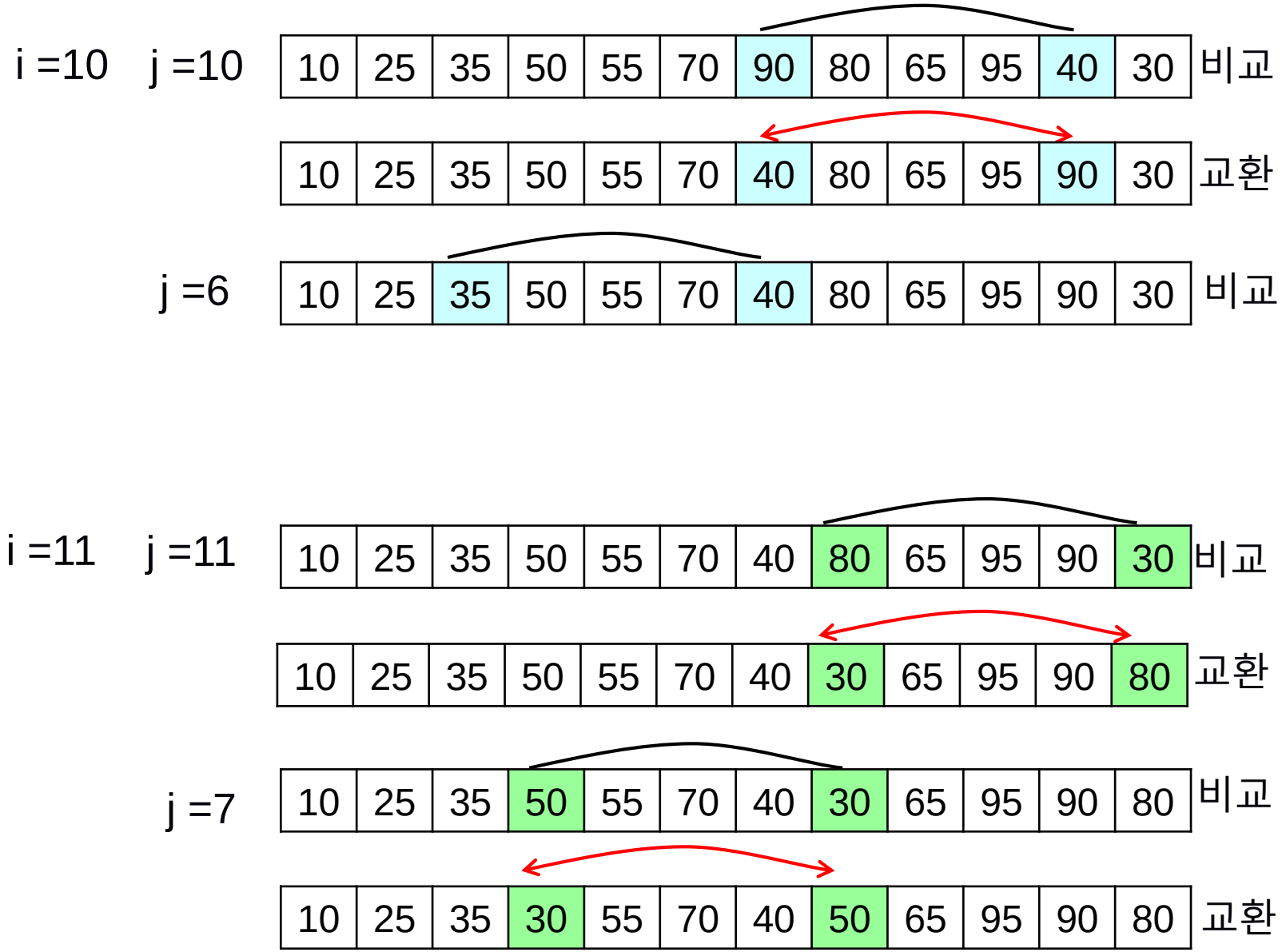
```
01 import java.lang.Comparable;
02 public class Shell {
03     public static void sort(Comparable[] a) {
04         int N = a.length;
05         int h=4;  // 3x+1 간격: 1, 4, 13, 40, 121,... 중에서 4 와 1만 사용
06         while (h >= 1) {
07             for (int i = h; i < N; i++) {  // h-정렬 수행
08                 for (int j = i; j >= h && isLess(a[j], a[j-h]); j -= h) {
09                     swap(a, j, j-h);
10                 }
11             }
12             h /= 3;
13         }
14     }
       // isLess(), swap() 메소드 선언, Selection 클래스와 동일함
   }
```

- Line 07: for-루프는 h-정렬 수행
- Line 12: h /= 3은 h 값을 1/3로 감소시킴

[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

수행시간

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

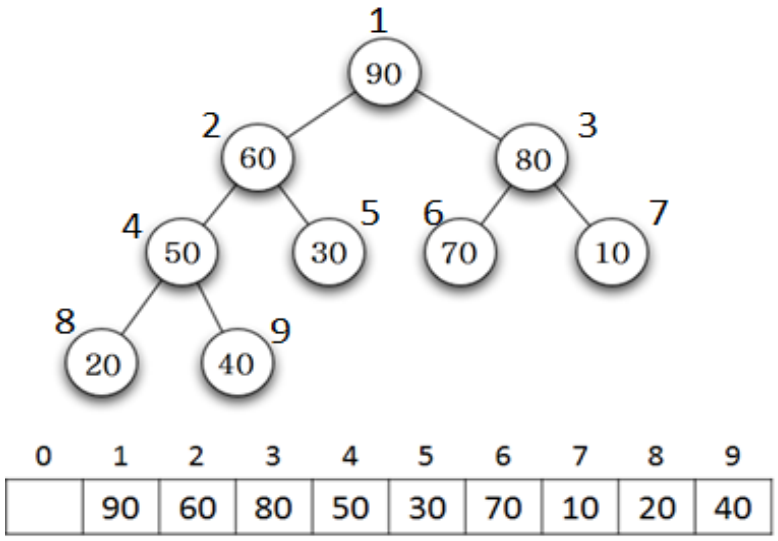
- **응용**
  - 쉘정렬은 임베디드(Embedded) 시스템에서 주로 사용
    - 간격에 따른 그룹별 정렬알고리즘을 하드웨어 설계를 통해 구현하는 것이 매우 쉽기(효율적이기) 때문

8.4 힙정렬 (Heap Sort)

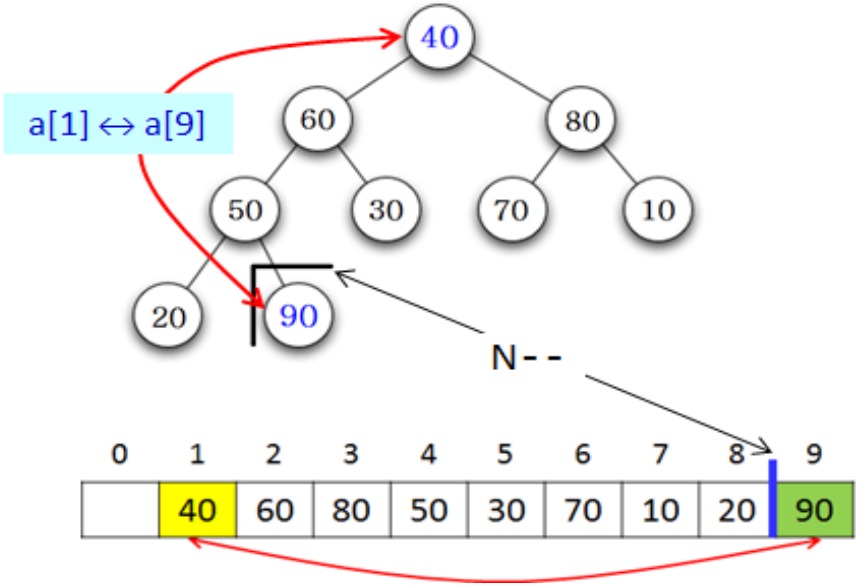
- **힙정렬은 힙 자료구조를 이용하는 정렬**
  - 먼저 배열에 저장된 데이터의 키를 우선순위로 하는 최대힙(Max Heap) 구성
  - 루트노드의 숫자를 힙의 가장 마지막 노드에 있는 숫자와 교환한 후
  - 힙 크기를 1 감소시키고
  - 루트노드로 이동한 숫자로 인해 위배된 힙속성을 downheap연산으로 복원
  - 힙정렬은 이 과정을 반복하여 나머지 원소들을 정렬

8.4 힙정렬 (Heap Sort)

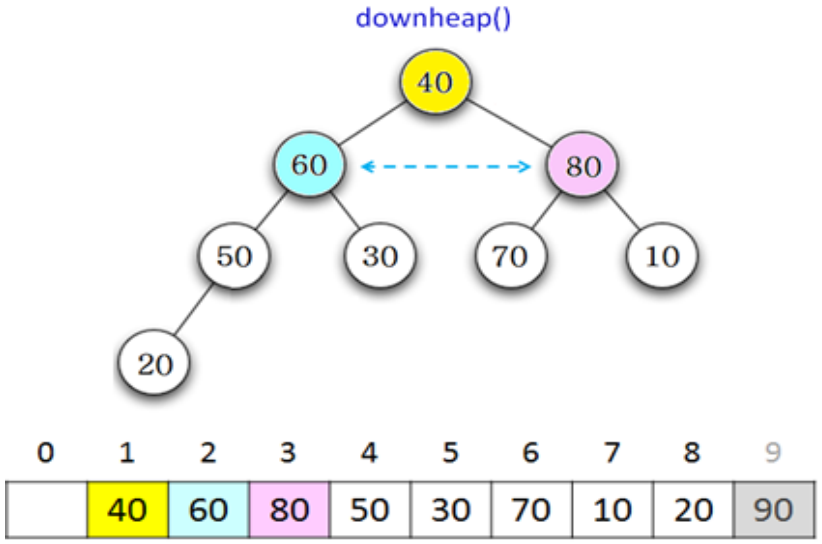
- 루트노드와 힙의 마지막 노드 교환 후 downheap 연산 수행 과정



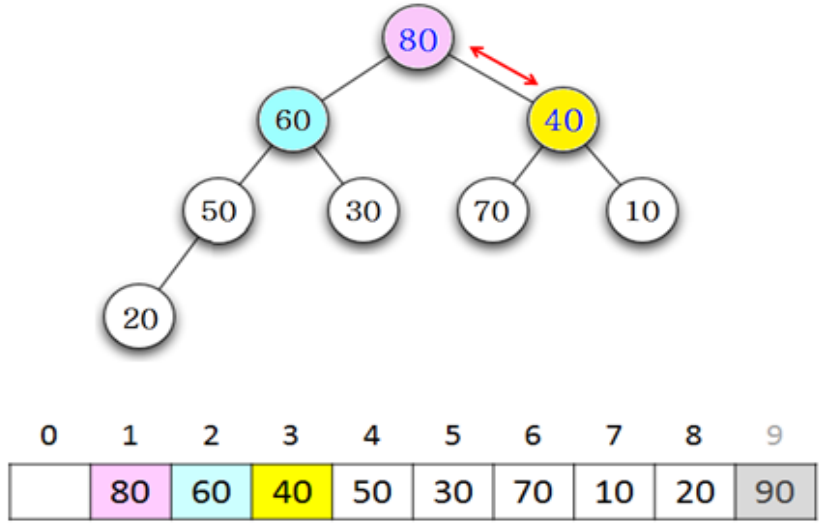
(a) 입력배열과 최대힙



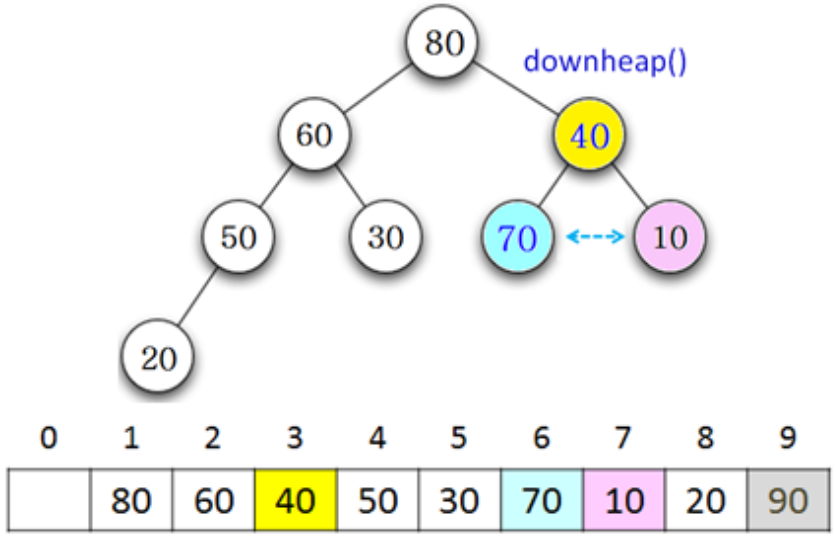
(b) 루트노드와 마지막 노드 교환



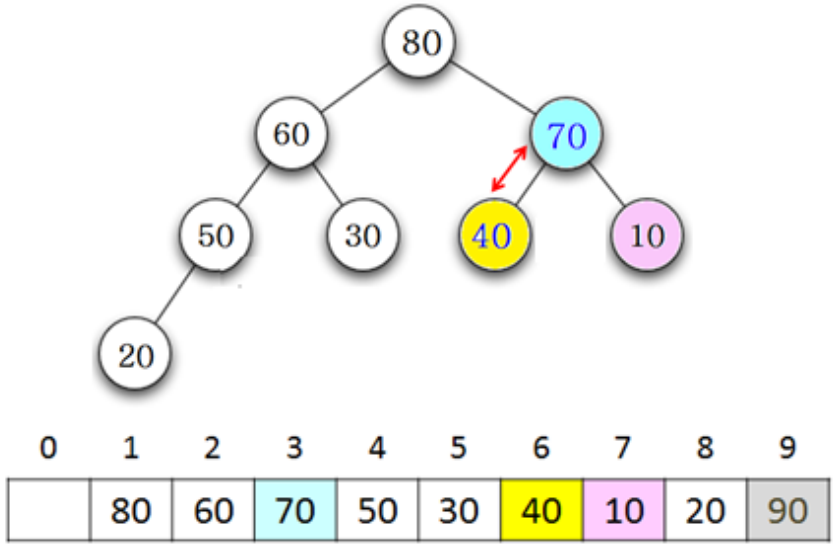
(c) 루트노드의 두 자식 비교



(d) 루트노드와 자식 승자와 교환



(e) 40의 두 자식 비교

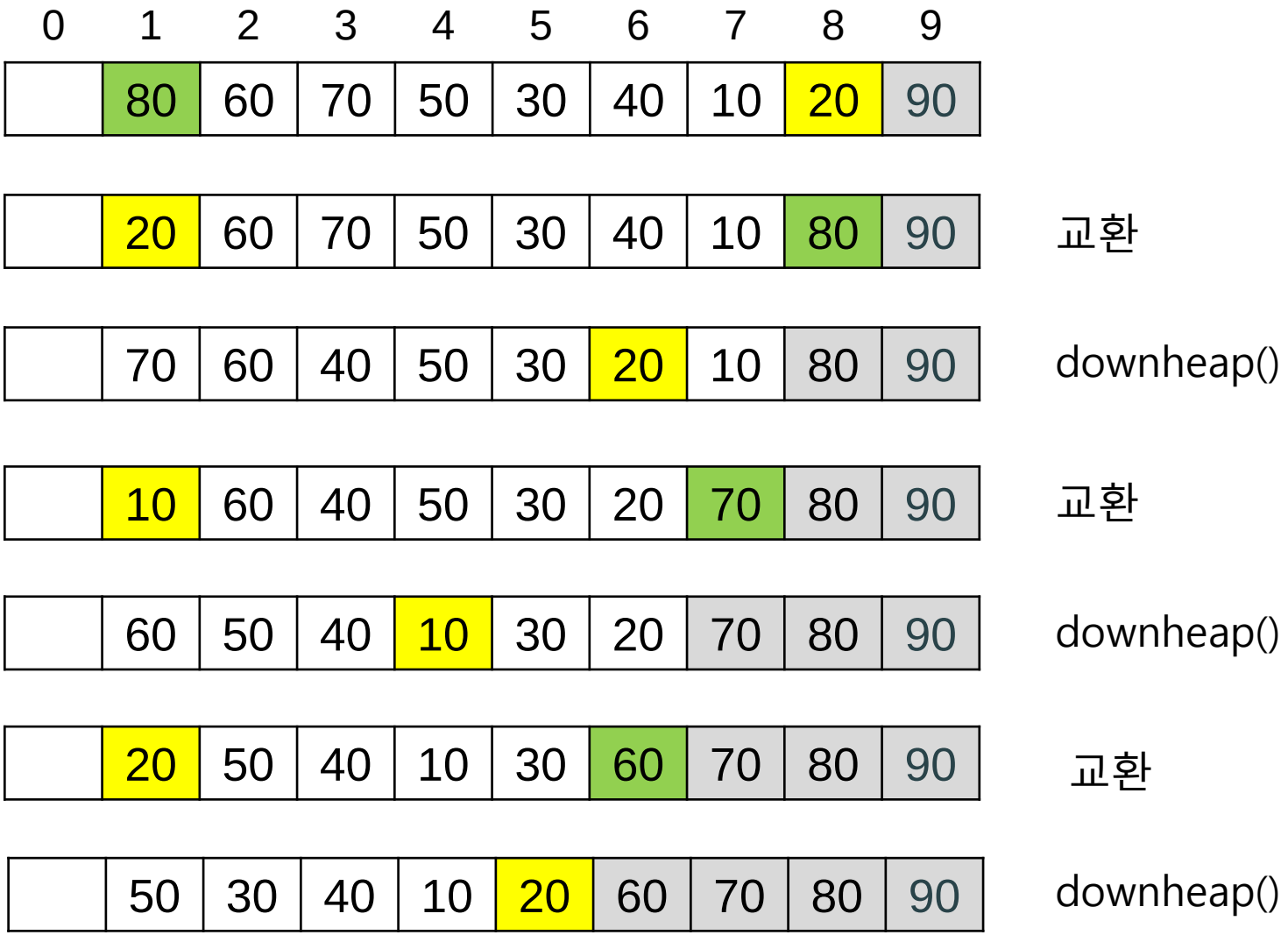


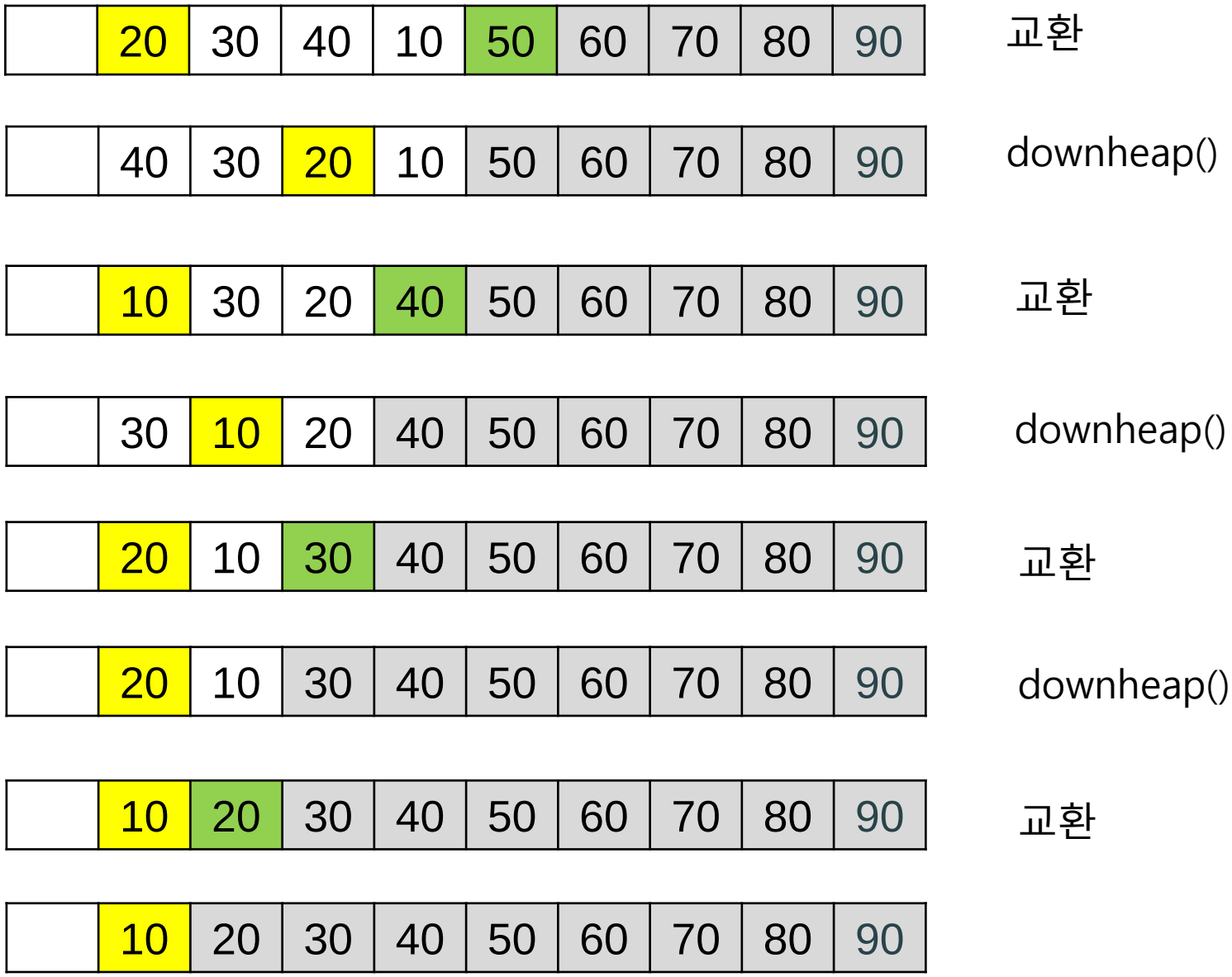
(f) 40과 자식 승자와 교환

8.4 힙정렬 (Heap Sort)

- 힙정렬은 입력배열을 (a)와 같은 최대힙으로 만든다. 노드 옆의 숫자는 노드에 대응되는 배열 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]~a[8]에 대해 동일한 과정을 반복 수행하여 힙 크기가 1이 되었을 때 종료

[예제] 앞선 예제에 이어서 힙정렬 수행 과정




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 import java.lang.Comparable;
02 public class Heap {
03     public static void sort(Comparable[] a) {
04         int heapSize = a.length-1;  // a[0]은 사용 안함
05         for (int i = heapSize/2; i > 0; i--) // 힙 만들기
06             downheap(a, i, heapSize);
07         while (heapSize > 1) {          // 힙정렬
08             swap(a, 1, heapSize--);    // a[1]과 힙의 마지막 항목과 교환
09             downheap(a, 1, heapSize); // 위배된 힙 속성 고치기
10         }
11     }
12     private static void downheap(Comparable[] a, int p, int heapSize) {
13         while (2*p <= heapSize) {
14             int s = 2*p;  // s = 왼쪽 자식의 인덱스
15             if (s < heapSize && isless(a[s], a[s+1])) s++; // 오른쪽 자식이 큰 경우
16             if (!isless(a[p], a[s])) break; // 부모가 자식 승자보다 크면 힙 속성 만족
17             swap(a, p, s);  // 힙 속성 만족 안하면 부모와 자식 승자 교환
18             p = s;    // 이제 자식 승자의 자리에 부모가 있게됨
19         }
20     }
    // isLess(), swap() 메소드 선언, Selection 클래스와 동일함
}
```

- Line 05~06: 상향식 힙만들기로 입력에 대한 최대힙을 구성
- Line 07~10: 정렬을 수행
- Line 07: while-루프가 수행될 때마다 line 12의 downheap() 메소드를 호출하여 힙속성을 충족시킴

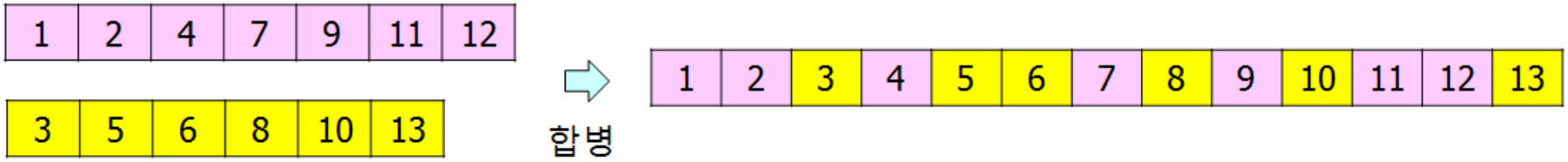
수행시간

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

8.5 합병정렬 (Merge Sort)

- 합병정렬은 크기가 N인 입력을 ½N크기를 갖는 입력 2개로 분할하고, 각각에 대해 재귀적으로 합병정렬을 수행한 후, 2개의 각각 정렬된 부분을 합병하는 정렬알고리즘
- 합병(Merge)이란 두 개의 각각 정렬된 입력을 합치는 것과 동시에 정렬하는 것
- 분할정복(Divide-and-Conquer) 알고리즘: 입력을 분할하여 분할된 입력 각각에 대한 문제를 재귀적으로 해결한 후 취합하여 문제를 해결하는 알고리즘들

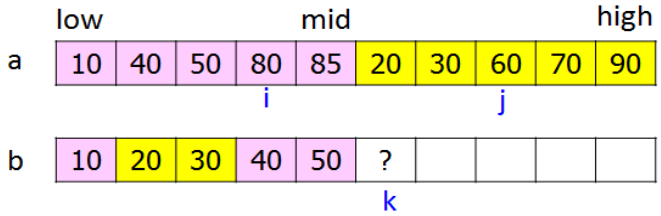
합병 과정



Merge 클래스

```
01 import java.lang.Comparable;
02 public class Merge {
03     private static void merge(Comparable[] a, Comparable[] b, int low, int mid, int high) {
04         int i = low, j = mid + 1;
05         for (int k = low; k <= high; k++) { // 배열 a의 앞부분과 뒷부분을 합병하여 보조배열 b에 저장
06             if      (i > mid)              b[k] = a[j++];    // 앞부분이 먼저 소진된 경우
07             else if (j > high)             b[k] = a[i++];    // 뒷부분이 먼저 소진된 경우
08             else if (isless(a[j], a[i]))   b[k] = a[j++];    // a[j]가 승자
09             else                           b[k] = a[i++];    // a[i]가 승자
10         }
11         for (int k = low;k <= high; k++)   a[k] = b[k];      // 보조배열 b를 배열 a에 복사
12     }
13     private static void sort(Comparable[] a, Comparable[] b, int low, int high) {
14         if (high <= low) return;
15         int mid = low + (high - low) / 2;
16         sort(a, b, low, mid);         // 앞부분 재귀호출
17         sort(a, b, mid + 1, high);    // 뒷부분 재귀호출
18         merge(a, b, low, mid, high); // 합병 수행
19     }
20     public static void sort(Comparable[] a) {
21         Comparable[] b = new Comparable[a.length];
22         sort(a, b, 0, a.length-1);
23     }
24     private static boolean isless(Comparable v, Comparable w) {
25         return (v.compareTo(w) < 0);
26     }
27 }
```

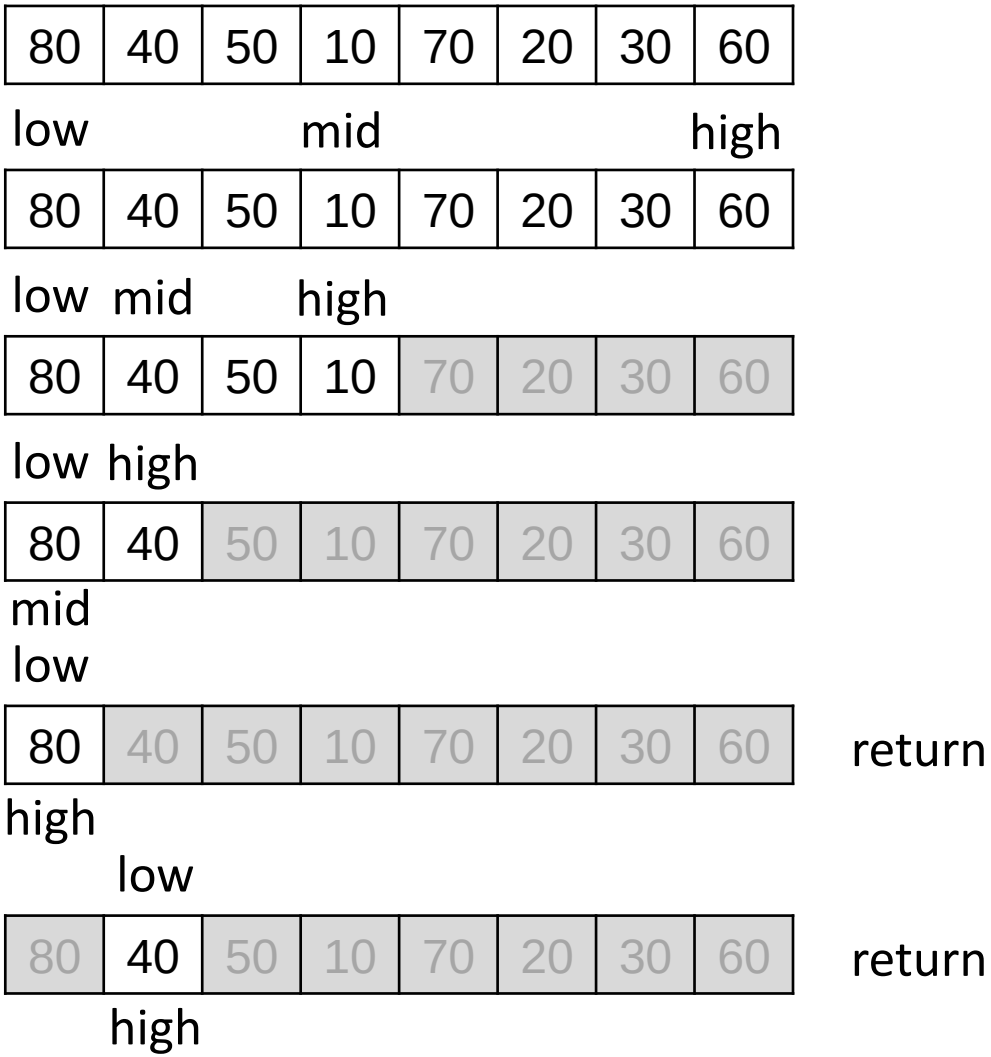
- Line 21: 입력배열 a와 같은 크기의 보조배열 b 선언
- Line 22: line 13의 sort()를 호출하는 것으로 정렬 시작
- Line 15: 정렬할 배열 부분 a[low]~a[high]를 1/2로 나누기 위해 중간 인덱스 mid를 계산
- Line 16: 1/2로 나눈 앞부분인 a[low]~a[mid]를 sort()의 인자로 넘겨 재귀호출
- Line 17: 뒷부분인a[mid+1]~a[high]를 sort()의 인자로 넘겨 재귀호출
- 앞부분과 뒷부분에 대한 호출이 끝나면, 각 부분이 정렬되어 있으므로 합병을 위해 line 18에서 merge()를 호출
- Line 03~12: a[low]~a[mid]와a[mid+1]~a[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 i는 변하지 않고, j와 k만 각각 1씩 증가하고, 다시 a[i]와 a[j]의 승자를 선택
- 합병의 마지막 부분인 line 11에서 합병된 결과가 저장되어있는 b[low]~b[high]를 a[low]~a[high]로 복사

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




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

수행시간

- 어떤 입력에 대해서도 O(NlogN) 시간 보장
- 입력 크기 N = 2k 가정
- T(N) = 크기가 N인 입력에 대해 합병정렬에서의 원소 비교 횟수(시간)

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

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

성능향상방법(1)

- 합병정렬은 재귀호출을 사용하므로 입력 크기가 1이 되어야 합병 시작
- 이 문제점을 보완하기 위해 입력이 정해진 크기, 예를 들어, 7~10이 되면 삽입정렬을 통해 정렬한 후 합병을 수행
  - Line 14를 다음과 같이 수정. CALLSIZE = 7~10 정도

```
if (high <= low) return;
```

```
if (high < low + CALLSIZE) {
    Insertion.sort(a, low, high);
    return;
}
```

- merge()의 line 11에서 매번 보조배열 b를 입력배열 a로 복사하는데, 이를 a와 b를 번갈아 사용하도록 하여 합병정렬의 성능을 향상시킬 수 있음

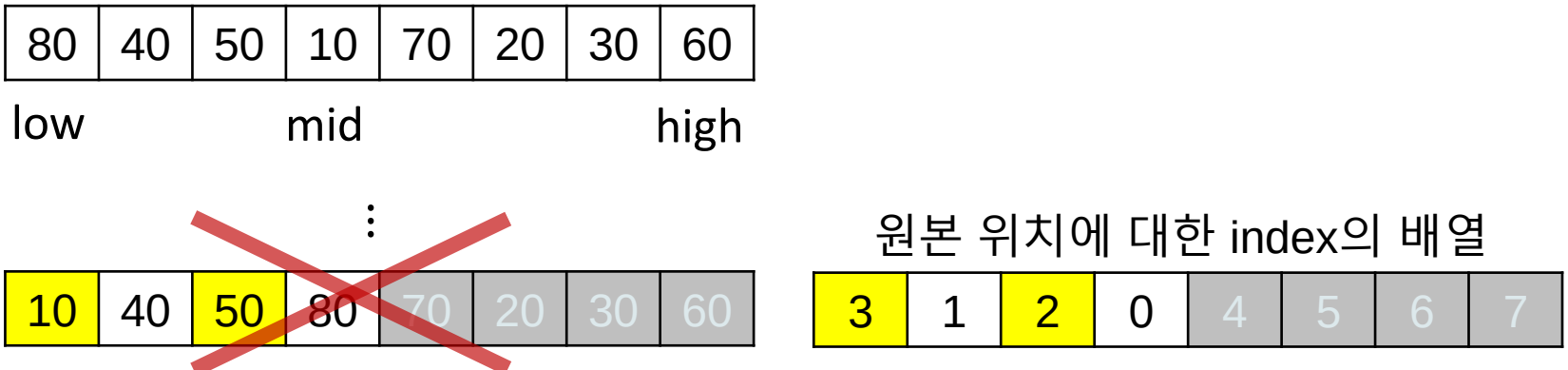
성능향상방법(2)

- **합병하기 위한 두 개의 리스트가 이미 정렬되어 있다면 합병 생략**
  - 따라서 Merge 클래스의 line 18에 있는 merge()를 호출하기 직전에, 즉, line 17과 line 18 사이에 다음의 if-문을 추가하면 불필요한 merge() 호출을 방지할 수 있음

```
if (!isless(a[mid+1], a[mid])) return;
```

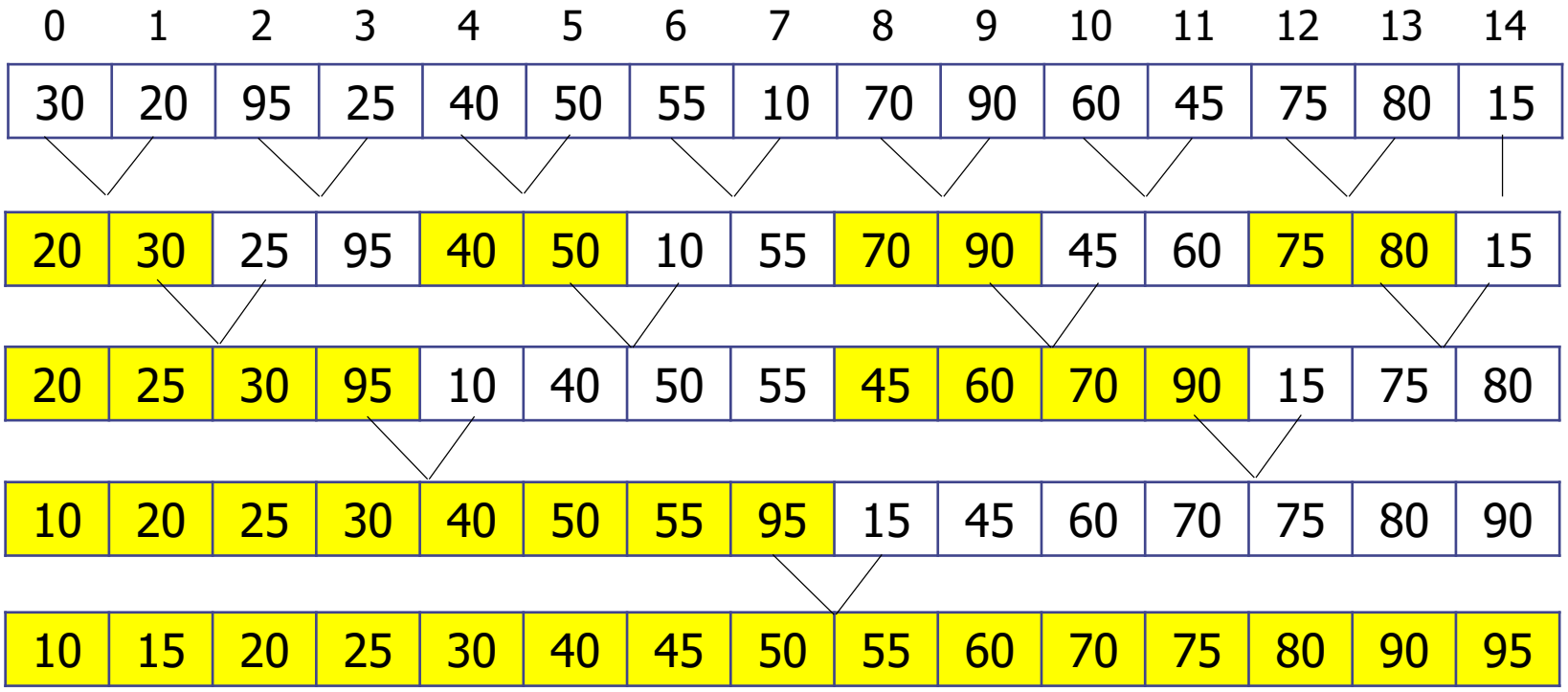
- **모든 정렬 방식에서 실제 data를 이동하지 않고, index 또는 reference를 저장하는 배열에 저장된 위치만을 저장하여 데이터 이동에 대한 부하를 감소시킬 수 있음**

합병



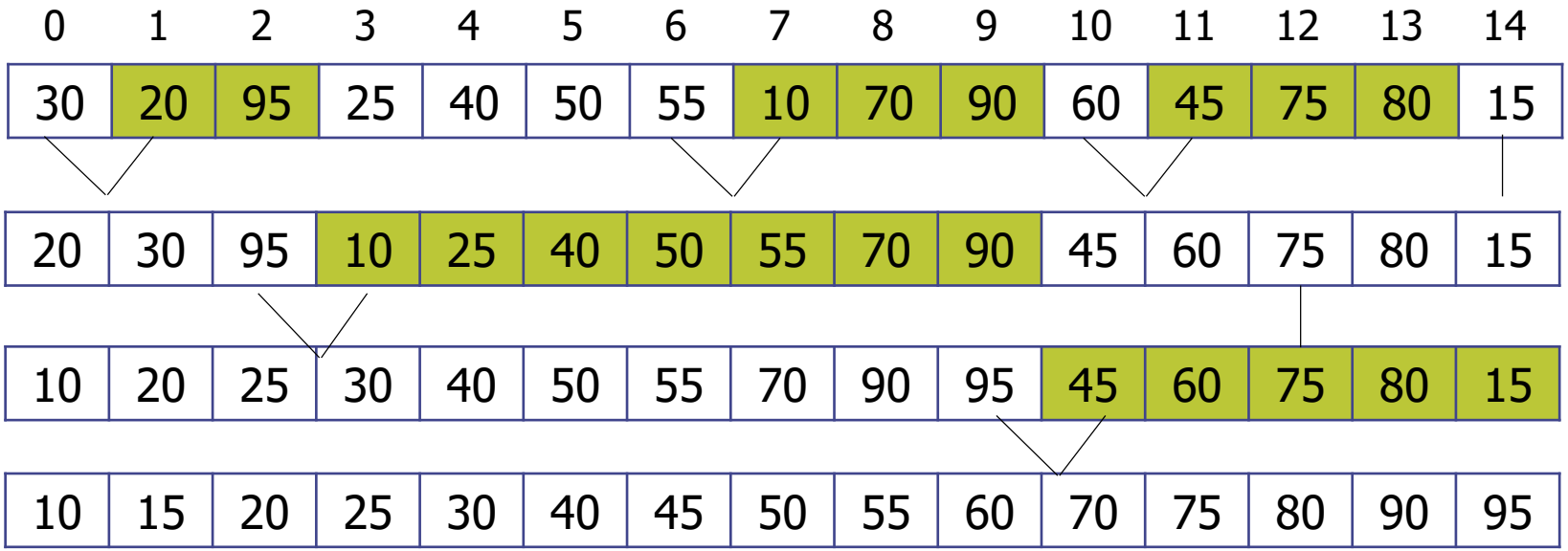
반복합병정렬

- 입력배열에서 바로 2 개씩 짝지어 합병한 뒤, 다시 4 개씩 짝지어 합병하는 상향식 (Bottom-up)으로도 수행 가능
- 이 합병정렬을 Bottom-up 합병 또는 반복(Iterative)합병정렬이라함



성능 향상 방안

- **자연 합병 정렬 (Natural Merge Sort)**
  - 입력 리스트 내에 이미 존재하고 있는 순서를 고려
  - 이미 정렬되어 있는 레코드의 서브리스트를 식별할 수 있도록 초기 패스를 만들어야 함



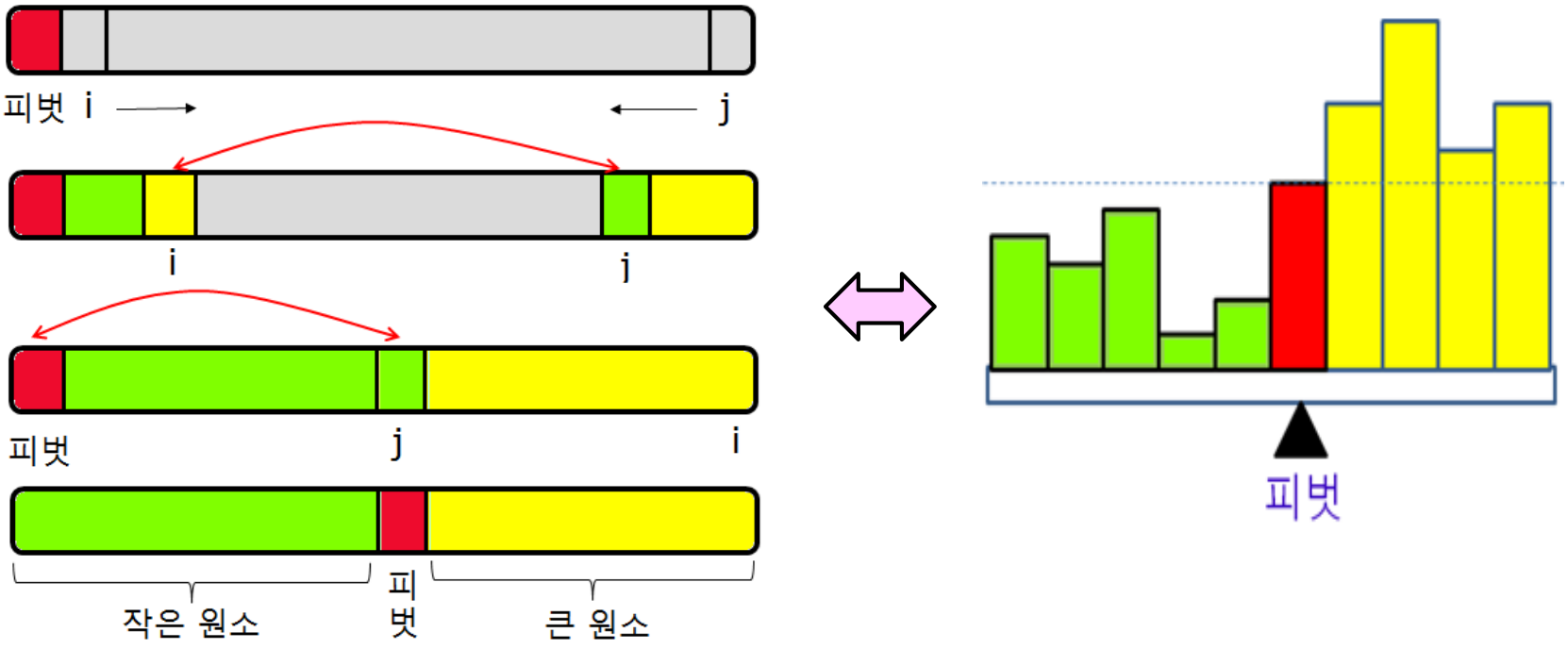
수행시간

- 반복합병정렬의 수행시간: 합병정렬과 동일한 O(NlogN)

- [단점] 입력배열 크기의 보조배열 사용
- 대부분의 정렬알고리즘들은 보조배열 없이 입력배열에서 정렬을 수행
  - 이러한 알고리즘을 In-place 알고리즘이라고 한다.
- 보조배열을 사용하지 않는 합병정렬 알고리즘도 연구된 바 있으나 알고리즘이 너무 복잡하여 실질적인 효용 가치는 없음
- 합병정렬은 자바 (Standard Edition 6) 객체 정렬, Perl, Python 등에서 시스템 sort로 활용

8.6 퀵정렬 (Quick Sort)

- 퀵정렬은 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



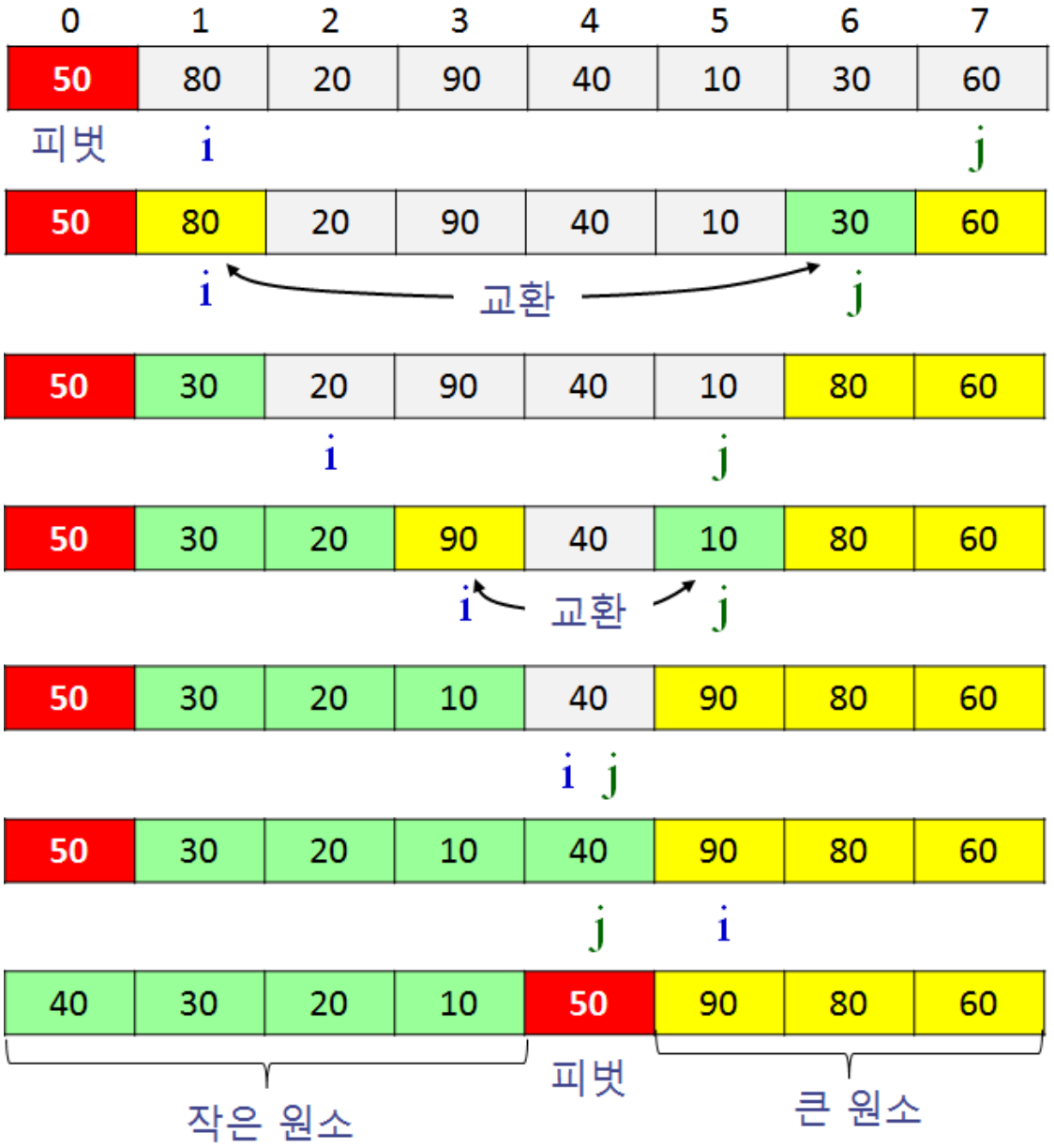
Quick 클래스

```
1  import java.lang.Comparable;
2  public class Quick {
3      public static void sort(Comparable[] a) {
4          sort(a, 0, a.length - 1);
5      }
6      private static void sort(Comparable[] a, int low, int high) {
7          if (high <= low) return;
8          int j = partition(a, low, high);
9          sort(a, low, j-1);  // 피벗보다 작은 부분을 재귀호출
10         sort(a, j+1, high); // 피벗보다  큰  부분을 재귀호출
11     }
12     private static int partition(Comparable[] a, int pivot, int high) {
13         int i = pivot+1;
14         int j = high;
15         Comparable p = a[pivot];
16         while (true) {
17             while (i < high  && isless(a[i], p)) i++; //피벗보다 작으면 i++
18             while (j > pivot && isless(p, a[j])) j--; //피벗보다 크면    j--
19             if (i >= j) break;  // i와 j가 교차되면 루프 나가기
20             swap(a, i, j);
21         }
22         swap(a, pivot, j); // 피벗과 a[j] 교환
23         return j;          //a[j]의 피벗이 "영원히" 자리 잡은 곳
24     }
```

```
25      private static boolean isless(Comparable u, Comparable v) {
26          return (u.compareTo(v) < 0);
27      }
28      private static void swap(Comparable [] a, int i, int j) {
29          Comparable temp = a[i];
30          a[i] = a[j];
31          a[j] = temp;
32      }
33  }
```

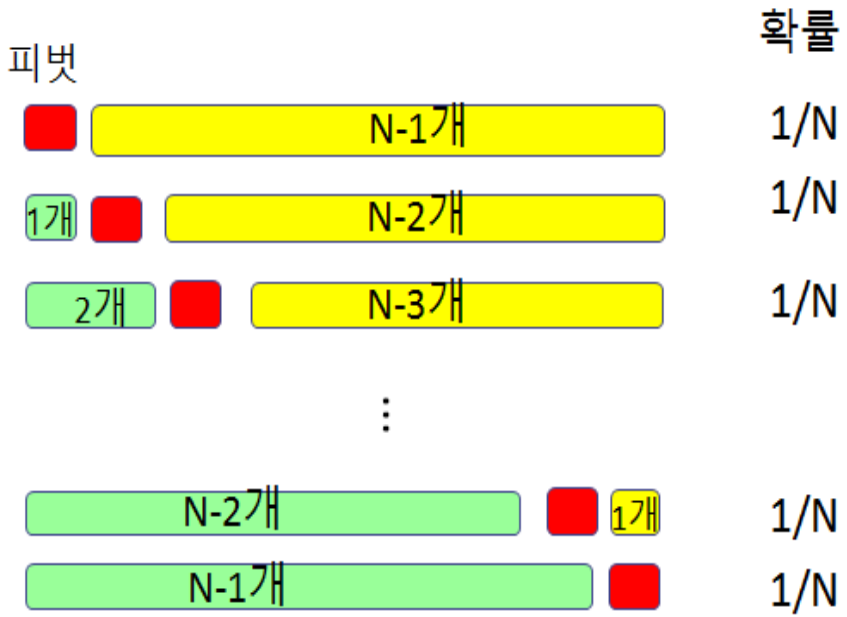
- Line 04: sort(a, 0, a.length-1)로 호출 시작
- Line 08: 피벗인 a[low]를 기준으로 a[low]~a[j-1]과 a[j+1]~a[high]로 분할하며, a[j]에 피벗이 고정
- Line 09: a[low]~a[j-1]을 재귀호출하여 정렬
- Line 10: a[j+1]~a[high]를 재귀호출하여 정렬

[예제] 피벗인 50으로 line 12의 partition()을 호출했을 때 수행 과정



수행시간

- 최선경우: 피벗이 매번 입력을 1/2씩 분할을 하는 경우
  T(N) = 2T(N) + cN, T(1) = c'
  으로 합병정렬의 수행시간과 동일. 여기서 c와 c'는 각각 상수
- 평균경우: 피벗이 입력을 다음과 같이 분할할 확률이 모두 같을 때,
  T(N) = O(NlogN)으로 계산



수행시간

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

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

성능향상방법[1]

- **퀵정렬은 재귀호출을 사용하므로 입력이 작은 크기가 되었을 때 삽입정렬을 호출하여 성능 향상**
  - 크기 제한: CALLSIZE를 7~10 정도
  - Line 07을 다음과 같이 수정

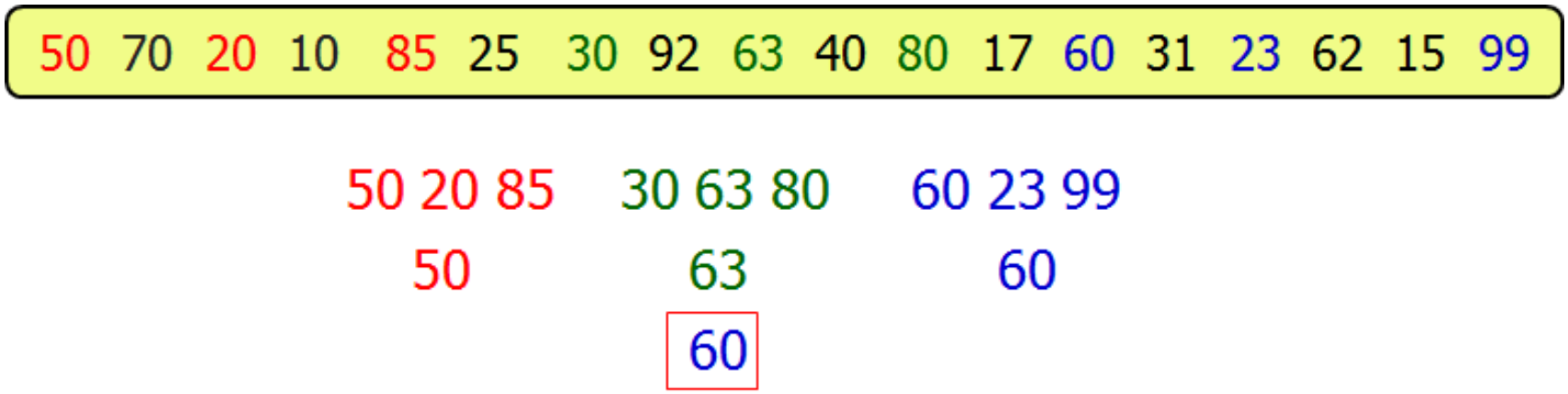
```
if (high <= low) return;
```

```
if (high < low + CALLSIZE) {
    Insertion.sort(a, low, high);
    return;
}
```

- **Median-of-Three**
  - 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
  - 가장 왼쪽(low), 중간(mid), 그리고 가장 오른쪽(high) 원소들 중에서 중간값을 찾는 것으로도 알려져 있음

성능향상방법[2]

- 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



성능향상방법[3]

- 퀵정렬 시작 전에 입력 배열에 대해 랜덤섞기(Random Shuffling) 수행
- 치우친 분할이 일어나는 것을 확률적으로 방지
- Knuth의 O(N) 시간 Random Shuffle 메소드를 사용

```
private static void shuffle(Object[] a){
    int N = a.length;
    Random rand = new Random();
    for (int i = 0; i < N; i++){
        int r = rand.nextInt(i+1);
        swap(a, i , r);
    }
}
```

퀵정렬의 장단점 및 응용

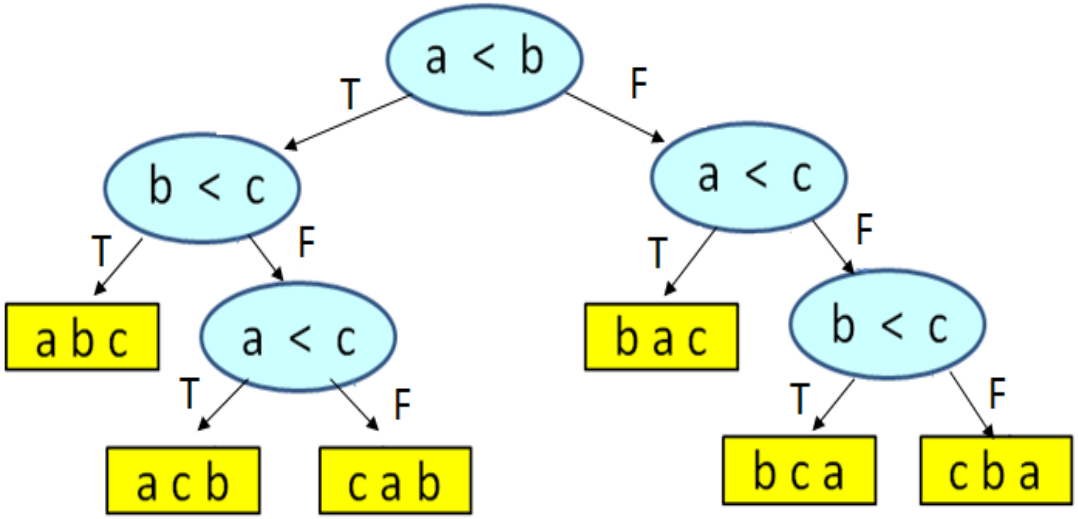
- 평균적으로 빠른 수행시간을 가지며, 보조배열을 사용하지 않음
- 최악경우 수행시간이 $O(N^2)$이므로, 성능 향상 방법들을 적용하여 사용하는 것이 바람직함
- 원시 타입(Primitive Type) 데이터를 정렬하는 자바 Standard Edition 6의 시스템 sort에 사용
- C-언어 라이브러리의 qsort, 그리고 Unix, g++, Visual C++, Python 등에서도 퀵정렬을 시스템 정렬로 사용
- 자바 SE 7에서는 2009년에 Yaroslavskiy가 고안한 이중피벗퀵(Dual Pivot Quick)정렬이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

8.7 정렬의 하한 및 정렬알고리즘의 비교

- 주어진 문제의 하한(Lower Bound)이란 문제를 해결하기 위해 수행되어야 할 최소한의 기본 연산의 횟수를 의미한다.
- 간단한 예로, N개의 정수를 가진 배열에서 최솟값을 찾는 문제의 하한은 N-1이다.
  - 만약 N-1보다 적은 비교 횟수로 최솟값을 찾았다면, 적어도 하나의 원소가 비교되지 않은 것이므로 찾은 숫자가 최솟값이 아닐 수도 있다.

0	1	2	3	4	5	6	7	8			N-1
30	20	95	25	40	50	55	10	70			5

- 정렬 문제의 하한을 위해 반드시 원소 대 원소의 크기를 비교(원소들을 '통째로' 비교)하는 것으로 가정
- 이를 비교정렬(Comparison Sort)이라함
- 3 개의 서로 크기가 다른 원소, a, b, c가 있을 때, a, b, c의 값에 따라서 총 6 개의 정렬 결과



결정트리(Decision Tree)

- **결정트리의 내부노드에서는 2 의 다른 원소를 비교하며, 비교 결과에 따라 참이면 왼쪽으로, 거짓이면 오른쪽으로 내려가며 이파리에 이르렀을 때 주어진 원소의 값에 따른 정렬 결과를 얻음**
- **결정트리는 정렬알고리즘이 아님**
  - 가능한 모든 정렬 결과에 대해 최소의 비교 횟수를 보여줄 따름

결정트리의 특성

- 결정트리는 이진트리이다.
- 총 이파리 수는 N! 개이다.
- 결정트리에는 불필요한 비교를 하는 내부노드가 없다.

- 비교정렬의 하한 = 결정트리의 높이
- 트리에서 루트노드부터 이파리노드까지 가려면 적어도 3번의 비교를 해야 어느 경우에라도 올바른 정렬 결과를 얻음
- 이때 비교 횟수 3은 이파리노드를 제외한 결정트리의 높이인 3과 동일

결정트리의 높이

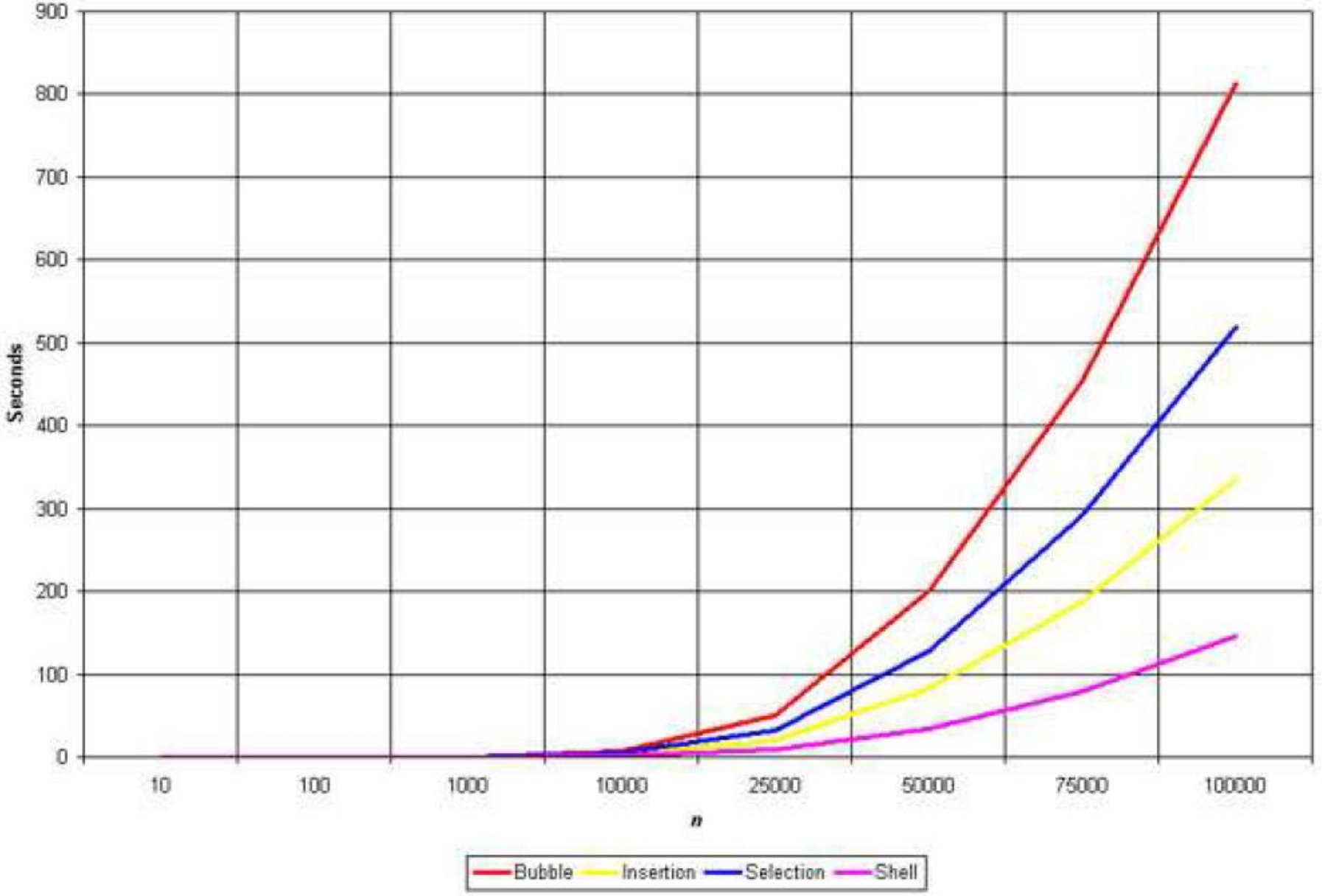
- k개의 이파리가 있는 이진트리의 높이는 logk보다 크다.
- 따라서 N개의 서로 다른 원소에 대한 결정트리의 높이는 적어도 logN!
- N! ≥ (N/2)N/2이므로
  - logN! ≥ log(N/2)N/2 = (N/2)log(N/2) = Ω(NlogN)
- 따라서 비교정렬의 하한은 Ω(NlogN)이다.

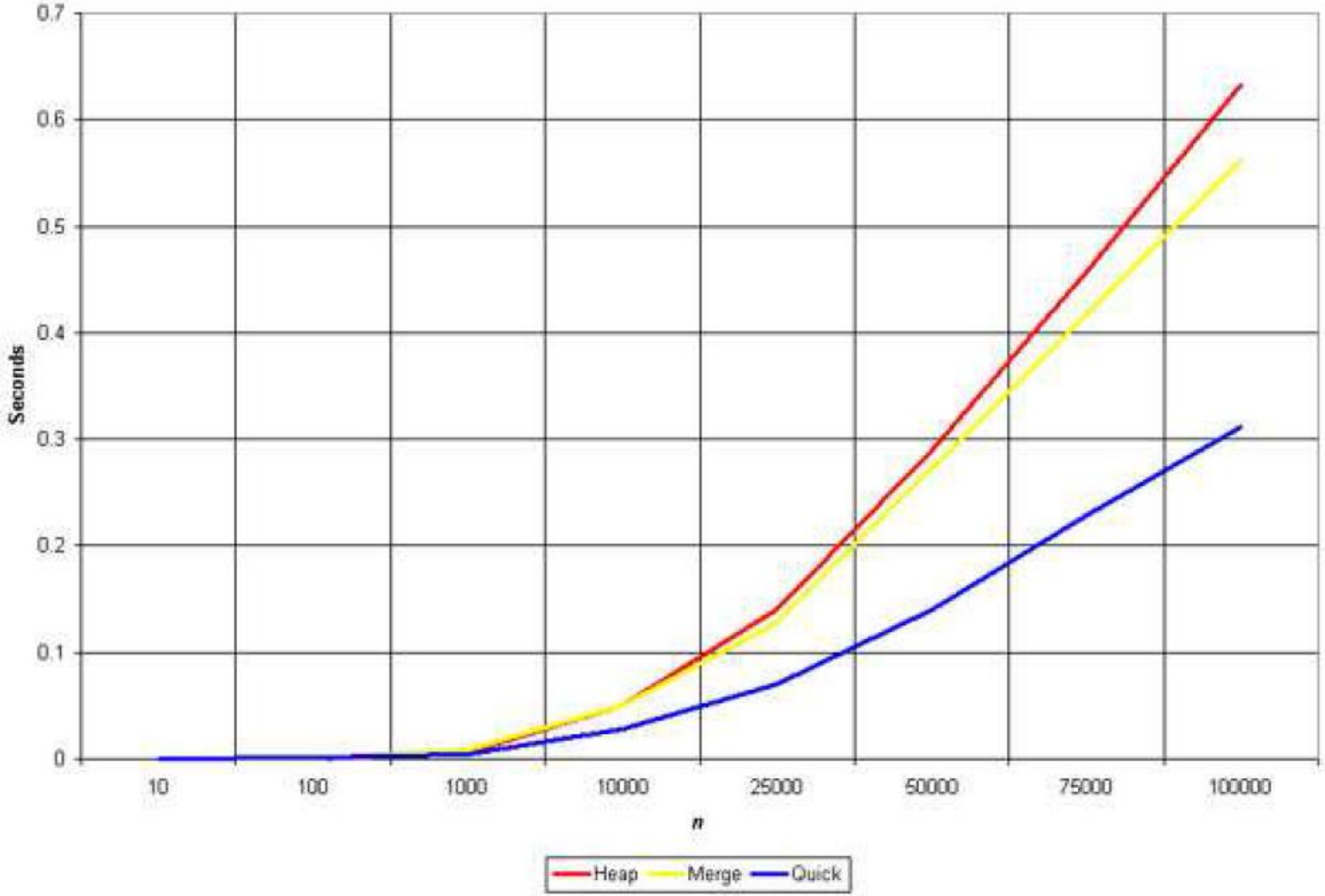
- 주어진 문제의 하한과 같은 수행시간을 가진 알고리즘을 최적 알고리즘이라고 하고, 힙정렬과 합병정렬은 비교 정렬에서의 최적 알고리즘이 됨

정렬알고리즘 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
쉘정렬	NlogN	?	$N^{1.5}$	O(1)	X
힙정렬	NlogN	NlogN	NlogN	O(1)	X
합병정렬	NlogN	NlogN	NlogN	N	O
퀵정렬	NlogN	NlogN	N^2	O(1)	X
Tim Sort	N	NlogN	NlogN	N	O

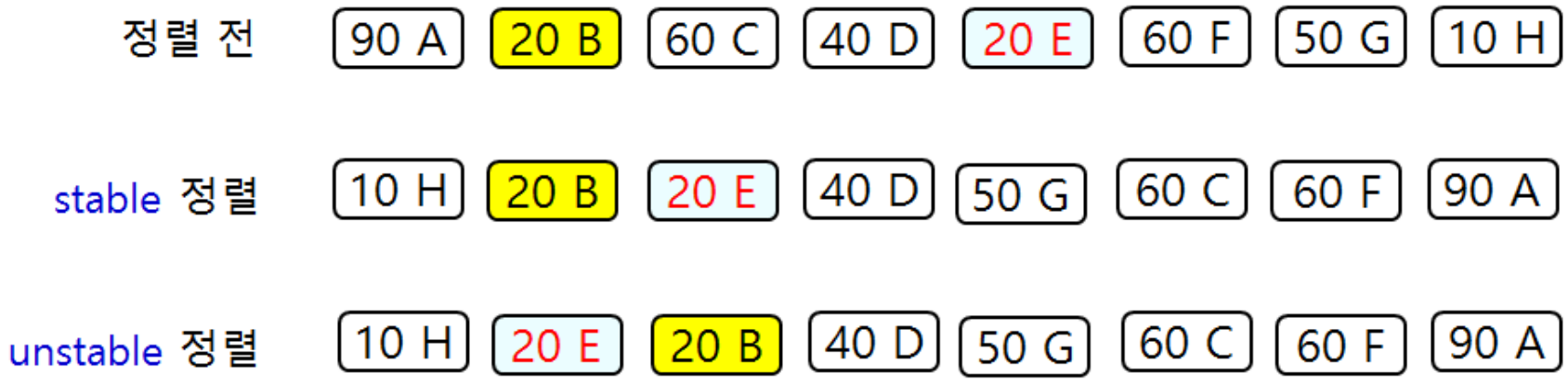
Tim Sort에 대해 보다 상세한 설명은 부록 VI


900
800
700
600
500
400
300
200
100
0
Seconds
10
100
1000
10000
25000
50000
75000
100000
n
Bubble
Insertion
Selection
Shell


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Seconds
10
100
1000
10000
25000
50000
75000
100000
n
Heap
Merge
Quick

Stable Sort

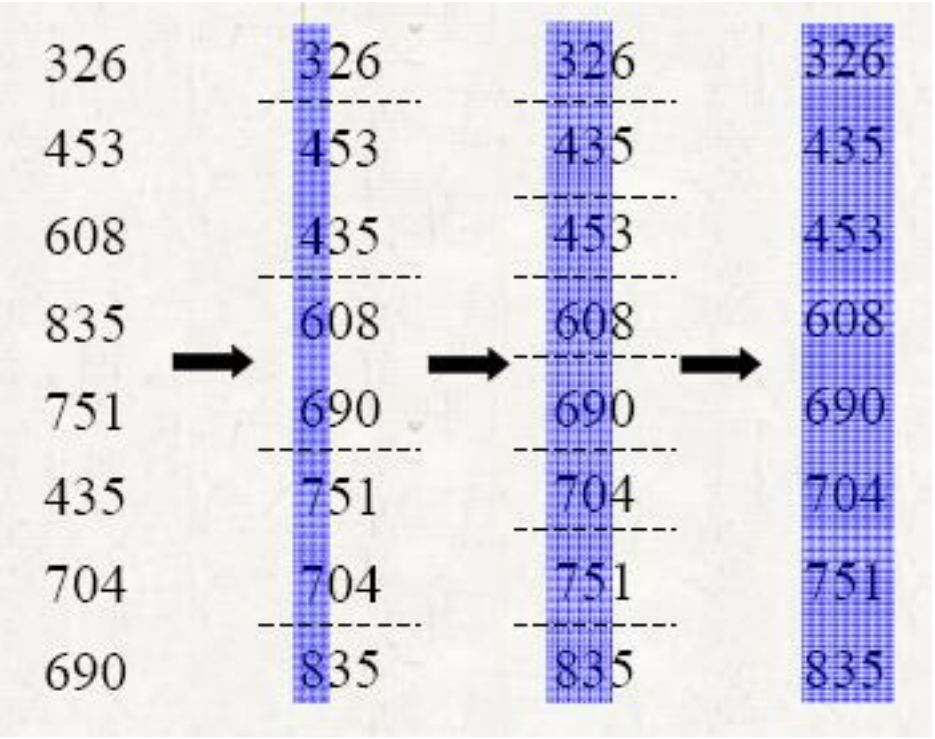
- 안정한 정렬(Stable Sort) 알고리즘은 중복된 키에 대해 입력에서 앞서 있던 키가 정렬 후에도 앞서 있음
- [예제] 안정한 정렬 결과에서는 [20 B]와 [20 E]가 각각 입력 전과 후에 항상 상대적인 순서가 유지되지만, 불안정한 정렬 결과에서는 입력 전과 후에 그 순서가 뒤바뀜

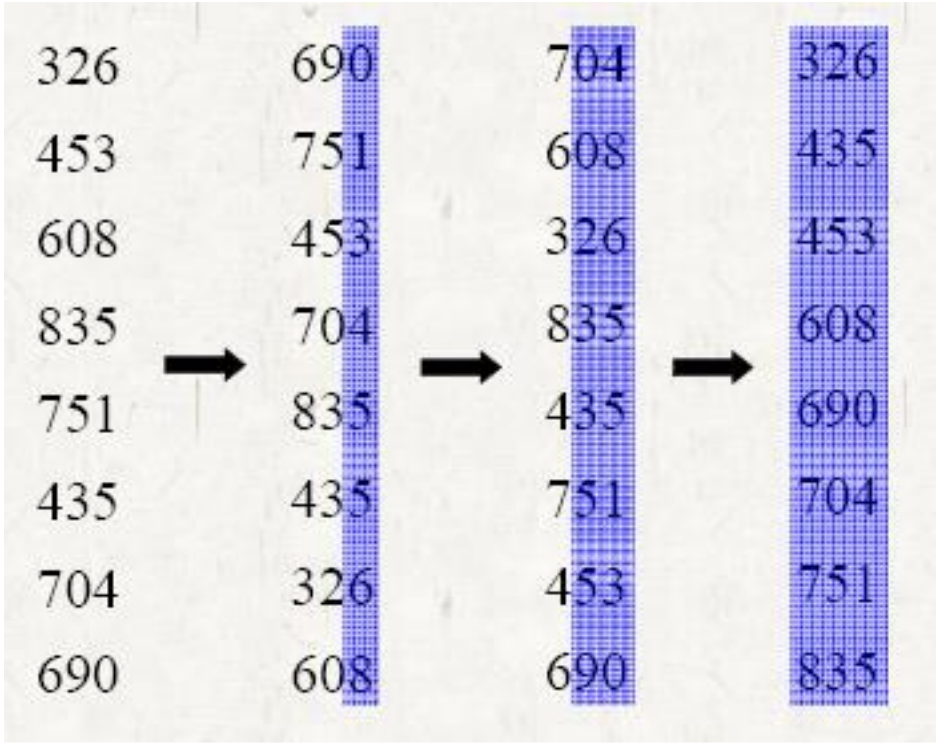


8.8 기수정렬

- **기수정렬(Radix Sort)은 키를 부분적으로 비교하는 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) 기수정렬**
  - 자릿수 비교를 최하위 숫자로부터 최상위 숫자 방향으로 정렬
- **MSD(Most Significant Digit) 기수정렬**
  - 반대 방향으로 정렬

여러 키에 의한 정렬






```
RADIX-SORT(A, d)
1  for i ← 1 to d
2        do use a stable sort to sort array A on digit i
```

주어진 3 자리 십진수 키에 대한 LSD 기수정렬

- 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 기수정렬을 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 기수정렬은 안정성(Stability)이 반드시 유지되어야

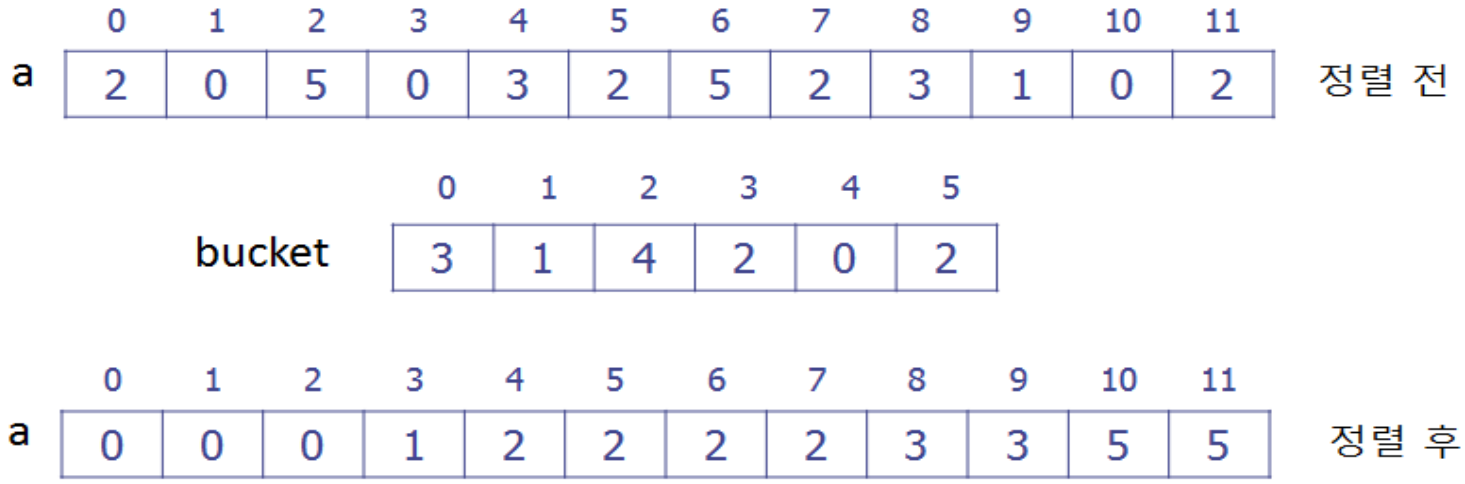
- **LSD 기수정렬은 키의 각 자릿수에 대해 버킷(Bucket)정렬 사용**
  - 버킷정렬은 키를 배열의 인덱스로 사용하는 정렬로서 2단계로 수행

> [1] 입력배열에서 각 숫자의 빈도수를 계산
>
> [2] 버킷 인덱스 0부터 차례로 빈도수만큼 배열에 저장

- **버킷정렬은 일반적인 입력에는 매우 부적절**
  - 왜냐하면 버킷 수가 입력크기보다 훨씬 더 클 수 있기 때문

버킷 정렬

- 빈도수 계산을 위해 1차원 배열 bucket을 사용
- 그림의 정렬 전 배열 a에 대해 버킷정렬은 [1]에서 0, 1, 2, 3, 4, 5의 빈도수를 각각 계산하여 배열 bucket에 저장
- [2]에서는 bucket[0] = 3이므로, 0은 3번 연속하여 a[0], a[1], a[2]에 각각 저장
- bucket[1] = 1이므로, 다음 빈 곳인 a[3]에 1을 1번 저장
- bucket[2] = 4이므로 4 번 연속하여 2를 저장
- 동일한 방법으로 3을 2 번 저장하고 5를 2 번 저장한 후 정렬을 종료



BucketSort 클래스

```
01 public class BucketSort {
02     public static void sort(int[] a, int R) {
03         int [] bucket = new int[R];
04         for (int i = 0; i < R; i++) bucket[i] = 0;  // 초기화
05         for (int i = 0; i < a.length; i++)  bucket[a[i]]++; // 1단계:빈도수 계산
06         // 2단계: 순차적으로 버킷의 인덱스를 배열 a에 저장
07         int j = 0;  // j는 다음 저장될 배열 a 원소의 인덱스
08         for (int i = 0; i<R; i++)
09             while((bucket[i]--) != 0)  // 버킷 i에 저장된 빈도수가 0이 될 때까지
10                 a[j++] = i;  // 버킷 인덱스 i를 저장
11     }
12     public static void main(String[] args) {
13         int [] a = {2, 0, 5, 0, 3, 2, 5, 2, 3, 1, 0, 2};
14         sort(a, 10);
15         System.out.print("정렬 결과: ");
16         for (int i = 0; i < a.length; i++) {
17             System.out.printf(a[i]+"  ");
18         }
19     }
20 }
```

이 코드는 stability가 보장되지 않음

- Line 05: 각 숫자의 빈도 수를 계산하여 bucket 배열에 저장
- Line 07 ~ 11: 차례로 bucket 배열의 원소에 저장된 빈도 수만큼 같은 숫자를 배열 a에 복사

3자리 십진수 키에 대한 LSD 기수정렬 수행 과정

- 배열 a는 입력 배열이고, t는 같은 크기의 보조배열이다.

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

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

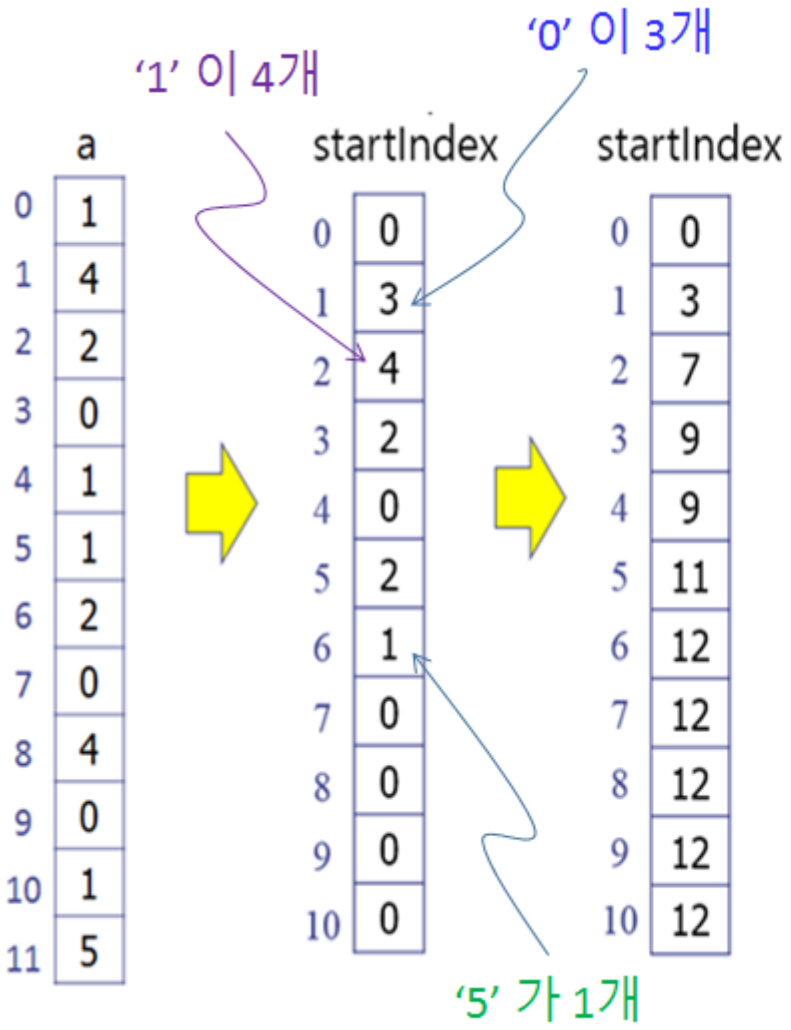
a			t		
3	0	1	0	0	2
4	0	1	0	1	0
0	0	2	0	2	2
2	0	4	1	1	5
0	1	0	1	2	4
1	1	5	2	0	4
0	2	2	2	5	1
1	2	4	3	0	1
4	3	0	4	0	1
5	4	0	4	3	0
2	5	1	5	4	0
5	5	1	5	5	1

LSDSort 클래스

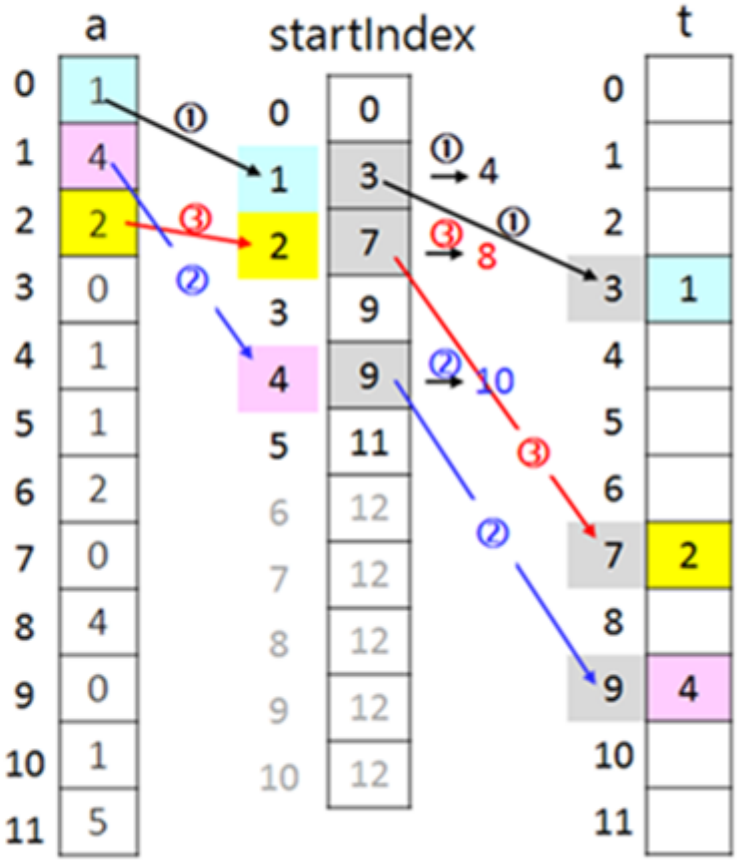
```
01 public class LSDsort {
02     public static void sort(int[] a) {
03         int R = 10;
04         int N = a.length;
05         int[] t = new int[N];
06         for (int k = 10; k <= 1000; k*=10) {
07             int[] startIndex = new int[R+1];
08             for (int i = 0; i < N; i++)     // 빈도수 계산
09                 startIndex[(a[i]%k)/(k/10) + 1]++;  //a[i]의 각 자릿수를 추출
10             for (int r = 0; r < R; r++)    // 각 버킷 인덱스 값이 저장될 배열 t의 시작 인덱스 계산
11                 startIndex[r+1] += startIndex[r];
12             for (int i = 0; i < N; i++)    // 해당 버킷의 인덱스 값에 대응되는 a[i]를 배열 t에 차례로 저
13                 t[startIndex[(a[i]%k)/(k/10)]++] = a[i];
14             for (int i = 0; i < N; i++)    // 배열 t를 배열 a로 복사
15                 a[i] = t[i];
16             System.out.println();
17             System.out.println(+k/10+"의 자리 정렬 결과:");
18             for (int i = 0; i < N; i++)
19                 System.out.print(String.format("%03d", a[i]) + " ");
20             System.out.println();
21         }
22     }
23     public static void main(String[] args) {
24         int [] a = {251,430,301,540,551,401,2,10,124,22,204,115};
25         sort(a);
26     }
27 }
```

- Line 06의 for-루프는 3자리 십진수 키를 k = 10일 때 1의 자리, k = 100일 때 10의 자리, k = 1000일 때 100자리의 숫자를 차례로 처리하기 위해 3 번 반복
- Line 08~09: 빈도수를 계산하는데, (a[i]%k)/(k/10)가 k = 10, 100, 1000일 때 각각 a[i]의 3자리 십진수 키로부터 1, 10, 100의 자리를 추출
- 예를 들어, a[i] = 259라면,
  - k = 10일 때, 259%10 = 9이고, 9를 (k/10) = (10/10) =1로 나누면 그대로 9가 되어, 259의 1의 자리인 '9'를 추출
  - k = 100일 때, 259%100 = 59이고, 59를 (k/10) = (100/10) =10으로 나눈 몫은 5이다. 따라서 259의 10의 자리인 '5'를 추출
  - k = 1,000일 때 259%1000 = 259이고, 259를 (1000/10) = 100으로 나누면 몫이 2이다. 따라서 259의 100자리인 '2'를 추출

- Line 09의 startIndex[(a[i]%k)/(k/10)+1]++에서 추출한 숫자에 1을 더한 startIndex 원소의 빈도수를 1 증가시킴
- [그림 8-15](a)를 보면 배열 a에 '0'이 3 개 있지만 startIndex[0]에 3을 저장하지 않고 "한 칸 앞의" startIndex[1]에 3을 저장
- 다른 숫자에 대해서도 각각 한 칸씩 앞의 startIndex 원소에 빈도수를 저장
- startindex[i]는 'i'를 다음에 배열 t에 저장할 곳(인덱스) (line 12~13)



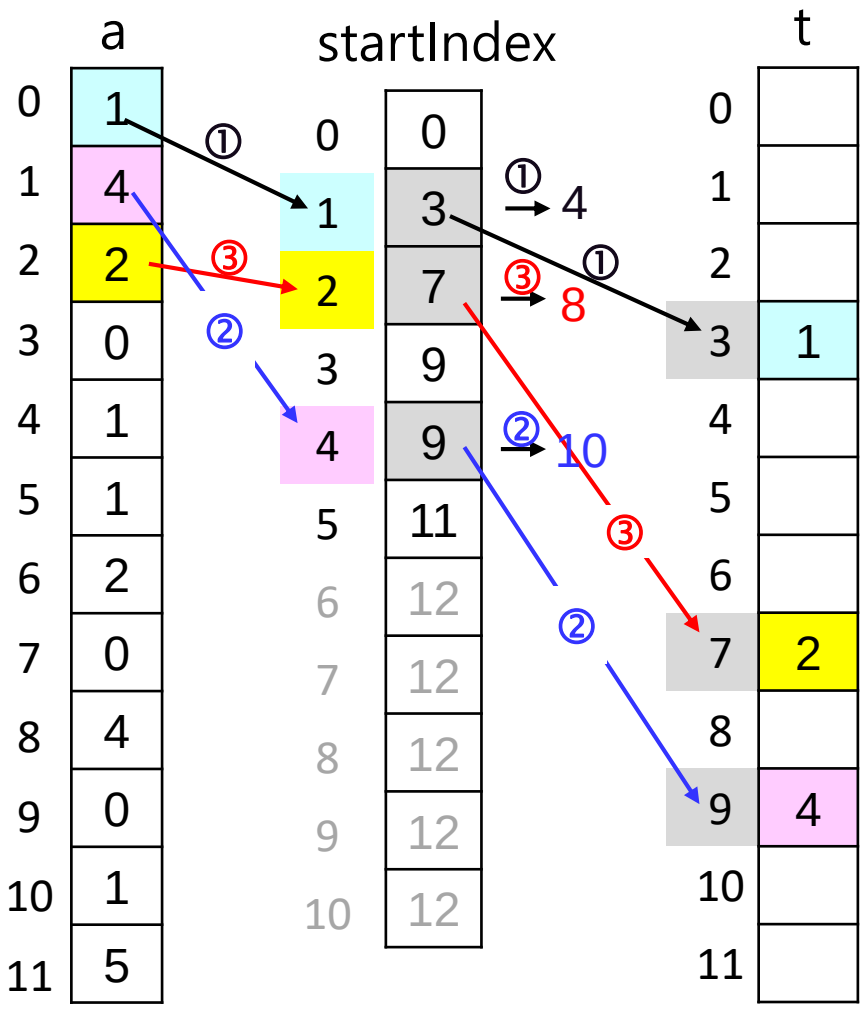
(a) 빈도수 및 누적 계산



(b) 버킷정렬

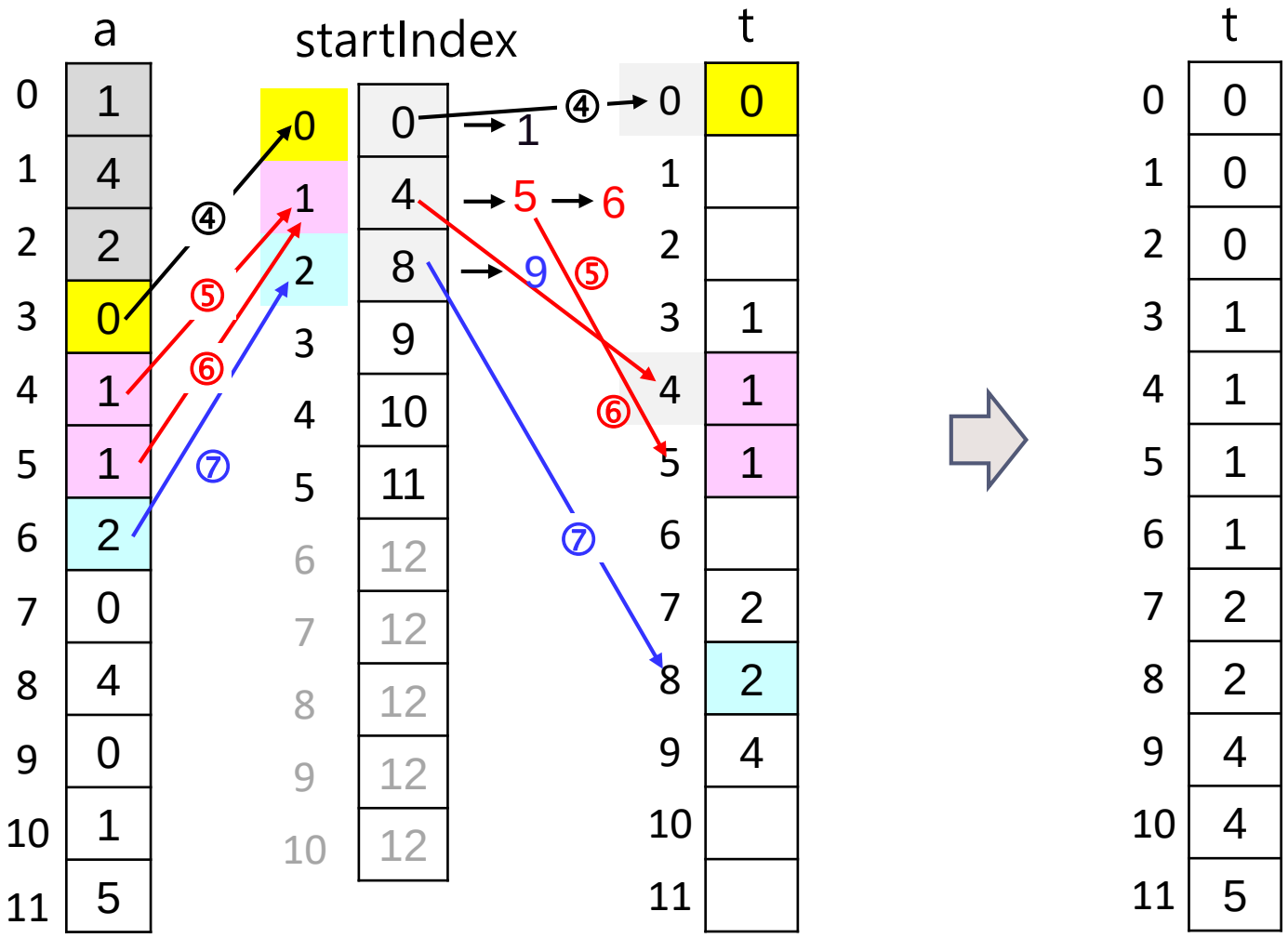
[그림 8-15] startIndex 배열 원소의 활용

```
// 숫자를 적절한 곳에 복사
for (int i = 0; i < N; i++)                    // k = 10, 100, 1000
    t[startIndex[(a[i]%k)/(k/10)]++] = a[i];
```



- [그림 8-15](b)에서는 a[0] = 1이므로 t[startIndex [1]] = t[3]에 ‘1’을 저장하고, startIndex[1] = 4로 갱신하여 다음에 ‘1’을 저장할 시작 인덱스를 만듦
- 다음으로 a[1] = 4이므로 t[startIndex[4]] = t[9]에 ‘4’를 저장하고, startIndex[4] = 10으로 갱신
- a[2] = 2이므로 t[startIndex[2]] = t[7]에 ‘2’를 저장하고, startIndex[2] = 8로 갱신
- 이와 같이 a[11] = 5를 t[startIndex[5]] = t[11]에 저장하며 입력의 한 자릿수에 대한 정렬 종료

```
for (int i = 0; i < N; i++)
   t[startIndex[(a[i]%k)/(k/10)]++] = a[i];
```

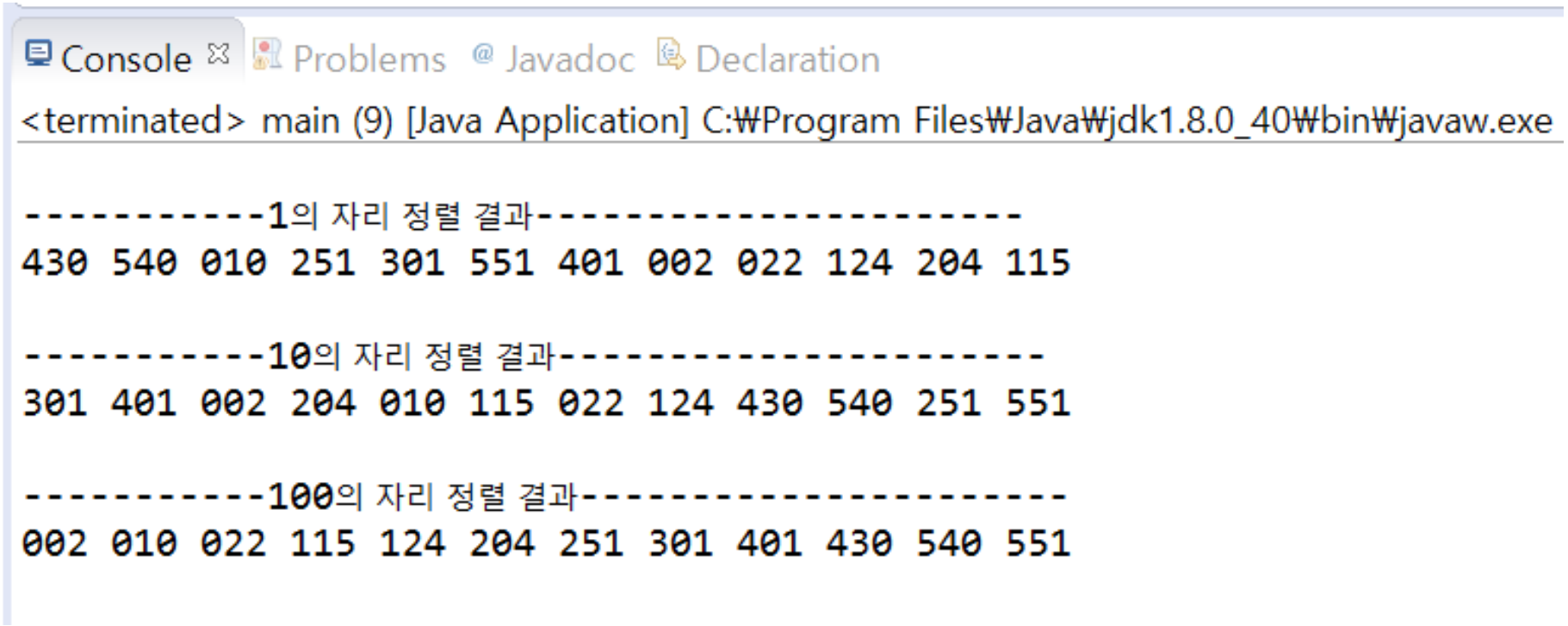


main 클래스

*main.java

```
1  public class main {
2      public static void main(String[] args) {
3          int [] a = {251,430,301,540,551,401,2,10,124,22,204,115};
4          int N = a.length;
5          int R = 10;
6          int[] t = new int[N];
7
8          for (int k = 10; k <= 1000; k*=10) {
9              int[] startIndex = new int[R+1];
10             for (int i = 0; i < N; i++)
11                 startIndex[(a[i]%k)/(k/10) + 1]++;
12             for (int r = 0; r < R; r++)
13                 startIndex[r+1] += startIndex[r];
14             for (int i = 0; i < N; i++)
15                 t[startIndex[(a[i]%k)/(k/10)]++] = a[i];
16             for (int i = 0; i < N; i++)
17                 a[i] = t[i];
18             System.out.println();
19             System.out.println("-----------"+k/10+"의 자리 정렬 결과----------------------");
20             for (int i = 0; i < N; i++)
21                 System.out.print(String.format("%03d", a[i]) + " ");
22             System.out.println();
23         }
24     }
25 }
```

LSDSort 클래스를 실행 결과



수행시간

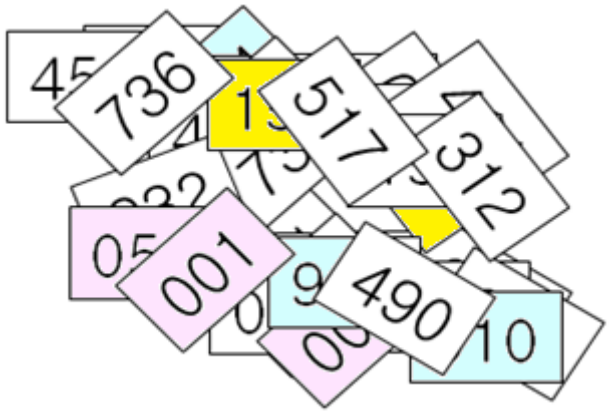
- **LSD 기수정렬의 수행시간은 O(d(N+R))**
  - 여기서 d는 키의 자리 수이고, R은 기(Radix)이며, N은 입력의 크기
- **Line 06의 for-루프는 d번 수행되고, 각 자릿수에 대해 line 08, 12, 14의 for-loop들이 각각 N번씩 수행되며, line 10의 for-loop는 R회 수행되기 때문**

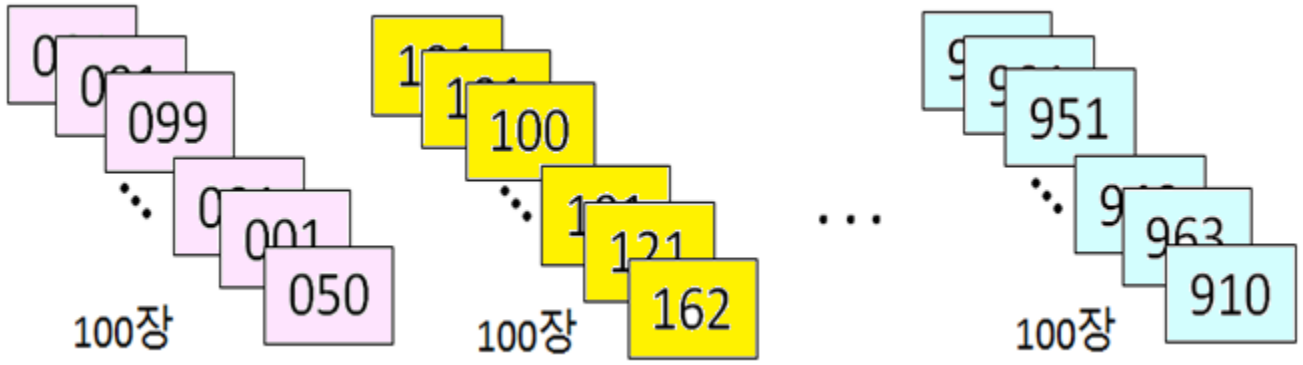
장단점 및 응용

- **장점 :LSD 기수정렬은 제한적인 범위 내에 있는 숫자(문자)에 대해서 좋은 성능을 보임**
  - 인터넷 주소, 계좌번호, 날짜, 주민등록번호 등을 정렬할 때 매우 효율적
  - 응용에서는 GPU(Graphics Processing Unit) 기반 병렬(Parallel) 정렬의 경우 LSD 기수정렬을 병렬처리 할 수 있도록 구현하여 시스템 sort로 사용 하거나 Thrust Library of Parallel Primitives, v.1.3.0의 시스템 sort로 사용
- **단점**
  - 기수정렬은 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

MSD(Most Significant Digit) 기수정렬

- MSD(Most Significant Digit) 기수정렬은 최상위 자릿수부터 최하위 자릿수로 버킷정렬을 수행
- 1,000장의 카드에 000부터 999까지 각각 다른 숫자가 적혀 있고, 이 카드들이 섞여있다. 이를 어떻게 정렬해야 할까?

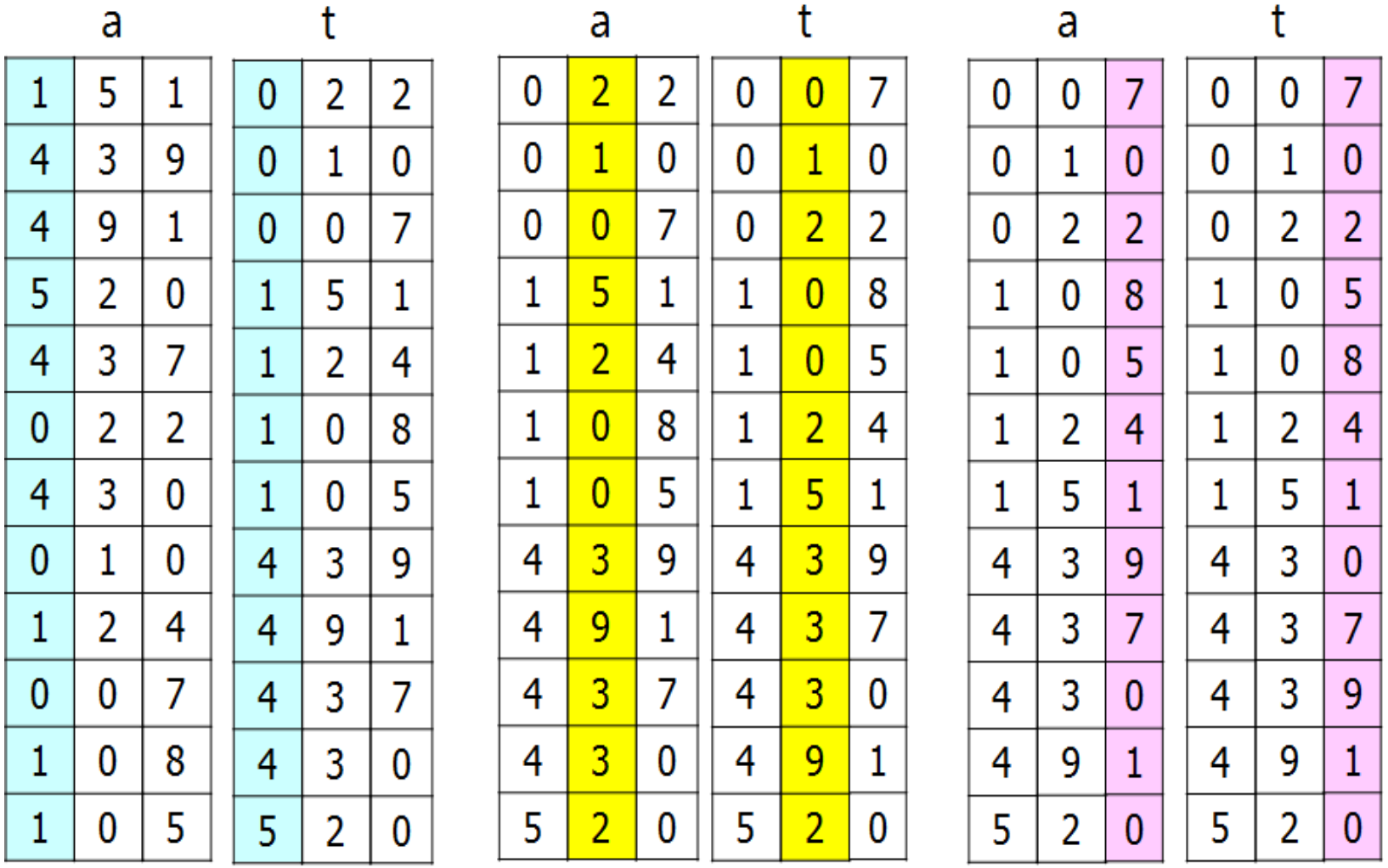




- 먼저 카드를 한 장씩 보고 100자리의 숫자에 따라 읽은 카드를 분류하여 10 개의 더미를 만든다.
- 각각의 더미에 대해 10의 자리 숫자만을 보고 마찬가지로 10 개의 작은 더미를 만들고,
- 마지막으로 각각의 작은 더미에서는 카드의 1의 자리를 보고 정렬하여 각각의 더미를 차례로 모아 정렬

MSD 기수정렬

- 최상위 자릿수부터 최하위 자릿수 순으로 정렬하는 과정



- MSD 기수정렬은 입력의 최상위 자릿수에 대해 정렬한 후에 배열을 0으로 시작되는 키들, 1로 시작되는 키들, …, 9로 시작되는 키들에 대해 각각 차례로 재귀호출
- 그 다음 자릿수에 대해서도 동일한 방식으로 정렬이 진행

수행시간

- **MSD 기수정렬의 수행시간은 O(d(N+R))**
  - LSD 기수정렬의 수행시간과 동일한데 LSD기수정렬이 수행 방향만 반대이기 때문
- **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
- **최하위 자릿수로 갈수록 너무 많은 수의 재귀호출 발생**
  - 재귀호출 시 입력크기가 작아지면 삽입정렬 사용

8.9 외부정렬

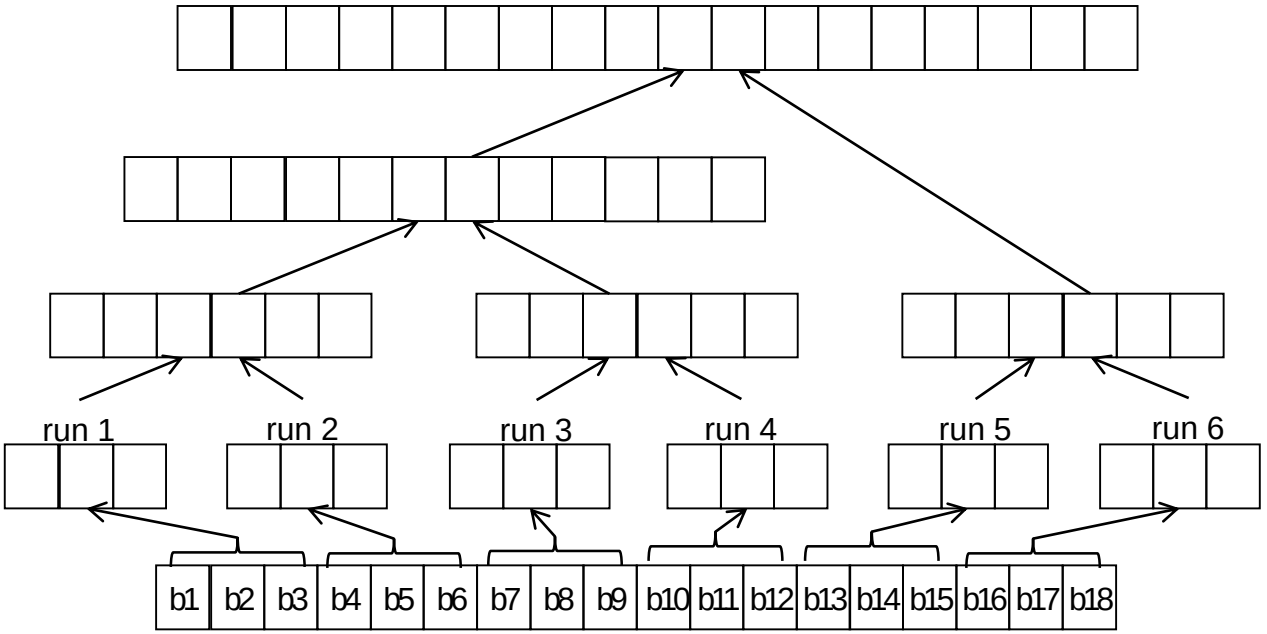
- **실세계에서는 대용량의 데이터를 하드디스크나 테이프와 같은 보조기억장치(또는 외부 메모리)에 저장**
  - 내부정렬만으로는 보조기억장치의 대용량의 데이터를 정렬하기 어려움
  - 보조기억 장치 종류: 자기(Magnetic) 하드디스크와 테이프 외에 SSD (Solid State Drive), 광학(Optical) 디스크, 플래시(Flash) 메모리 등
- **외부정렬(External Sort)이란 보조기억장치에 있는 대용량의 데이터를 정렬하는 알고리즘**
  - 기본적으로 합병(Merge)을 사용하여 정렬 수행
  - 외부정렬의 수행시간
    - 원소의 비교 횟수가 아니라 입력 전체를 처리하는 횟수로 계산
    - 보조기억장치의 접근시간이 주기억장치의 접근시간보다 매우 느리기 때문
    - 패스(Pass)는 입력 전체를 처리하는 단위

외부 정렬 (1)

- **정렬하려는 리스트 전체가 컴퓨터의 메인 메모리에 모두 올라갈 수 없다면 내부 정렬은 사용할 수 없음**
- **블록(block) - 한 번에 디스크로부터 읽거나 디스크에 쓸 수 있는 데이타의 단위, 여러 개의 레코드들로 구성됨**
- **판독/기록 시간에 영향을 미치는 세 가지 요소**
  - (1) 탐구 시간(Seek time)
    - 판독/기록 헤드가 디스크 상의 원하는 실린더로 이동하는데 걸리는 시간.
    - 헤드가 가로질러야 하는 실린더 수에 따라 결정됨
  - (2) 회전지연 시간(Latency time)
    - 트랙의 해당 섹트가 판독/기록 헤드 밑으로 회전해 올 때까지 걸리는 시간
  - (3) 전송 시간(Transmission time)
    - 디스크로 또는 디스크로부터 데이타 블록을 전송하는데 걸리는 시간

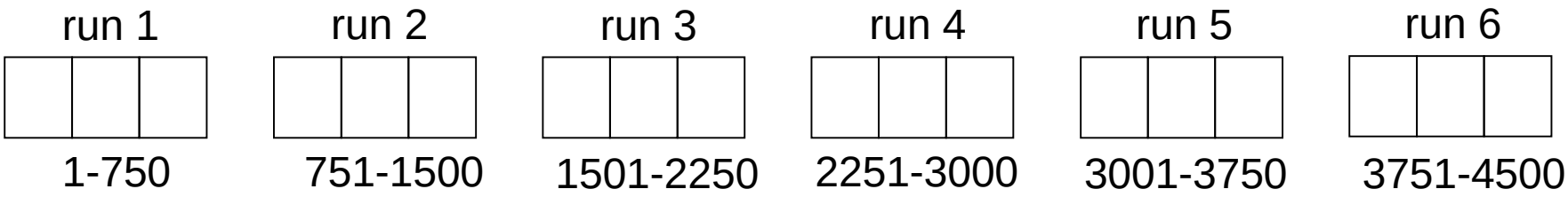
외부 정렬 (2)

- **합병 정렬**
  - 외부 저장장치 정렬에서 가장 널리 알려진 방법: 합병 정렬(merge sort)
    - 1단계
      - 입력 리스트의 여러 세그먼트들을 좋은 내부 정렬 방법으로 정렬.
      - 이 정렬된 세그먼트들을 런(run)이라 함. 런이 생성되면 외부 저장 장치에 기록됨
    - 2단계
      - 1단계에서 만들어진 런들을 하나의 런이 될 때까지 합병 트리 형식을 따라 합병함



외부 정렬 (3)

- **최대 750개의 레코드만 정렬할 수 있는 내부 메모리를 가지고 있는 컴퓨터를 이용하여 4500개의 레코드로 된 리스트를 정렬**
  - 입력 리스트는 디스크에 저장되어 있고 250 레코드의 블록 길이
  - 작업 공간으로 사용할 또 다른 디스크를 갖고 있음
  - (1) 내부적으로 한 번에 3개의 블록(750개의 레코드)을 정렬해서 여섯 개의 런 R1-R6을 만들고, 이 런들을 작업 디스크에 수록

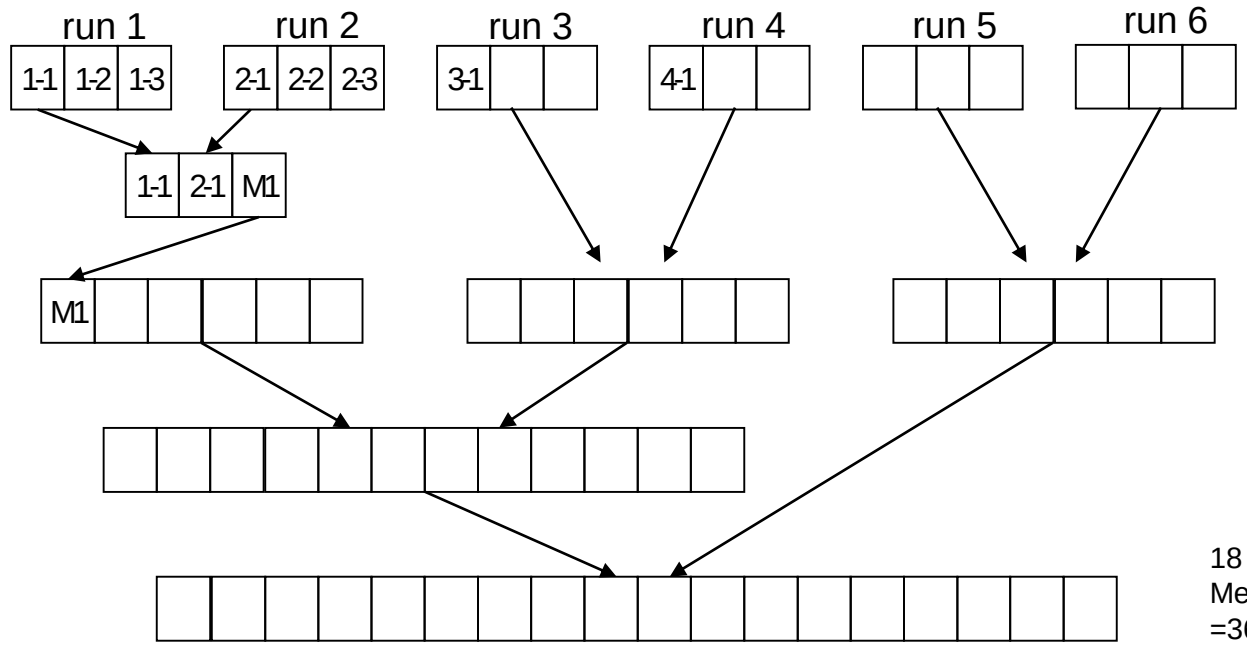


내부 정렬 후 얻어진 블록화 된 런 (한 런당 세 블록)

외부 정렬 (4)

- (2) 런들을 합병
  - 내부 메모리에 250개의 레코드를 수용할 수 있는 3개의 블록을 마련
  - 2개의 블록은 입력 버퍼로 쓰고 하나는 출력 버퍼로 씀
  - 입력 버퍼로 각 런으로부터 블록 하나씩 읽어들임
  - 런의 블록들은 입력 버퍼에서 출력 버퍼로 합병
  - 출력 버퍼가 다 차면 디스크에 기록됨
  - 입력 버퍼가 공백이 되면 같은 런에서 다음 블록을 읽어 들여 다시 채움

6개 런의 합병



18 block 읽고, 6 run 정렬하고
18 block 기록
$=18t_I+6t_{InSort}+18t_O$
$=36t_{IO}+6t_{InSort}$

18 block 읽어,
4500 레코드 Merge하고
18 block 기록
$=36t_{IO}+4500t_{merge}$

12 block 읽어, 3000 레코드
Merge하고 12 block 기록
$=24t_{IO}+3000t_{merge}$

18 block 읽어, 4500 레코드
Merge하고, 18 block 기록
$=36t_{IO}+4500t_{merge}$

외부 정렬 (5) - 복잡도 분석

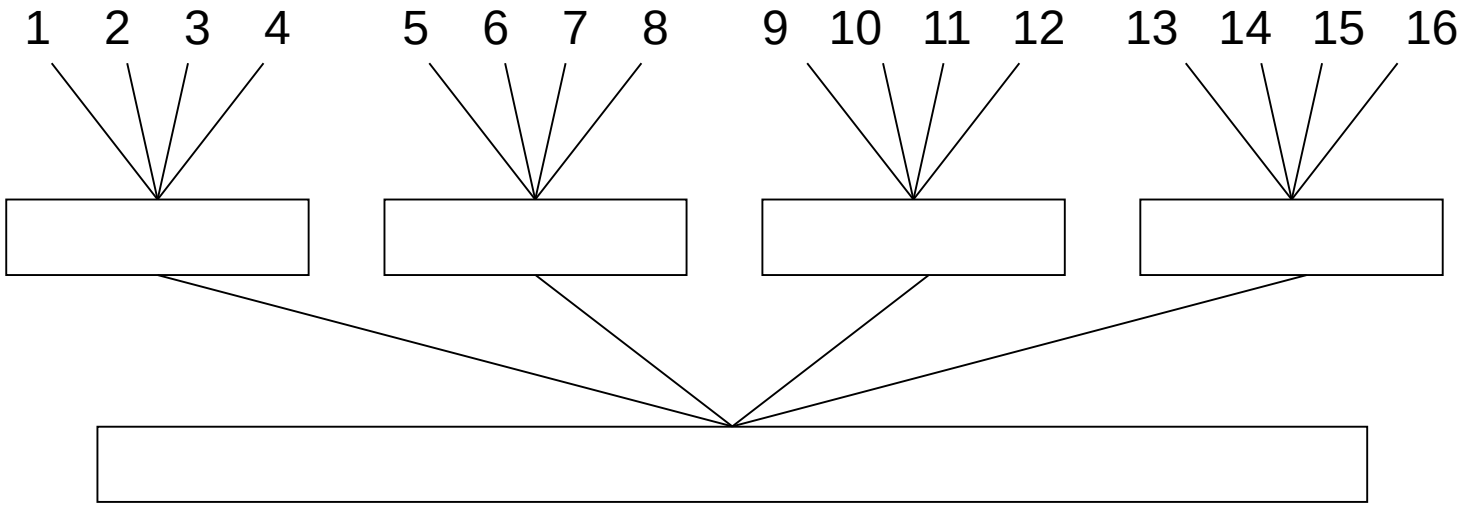
- 표기법
  - t_{IO} = 한 블록을 입력 또는 출력하는데 걸리는 시간 = $t_{seek} + t_{latency} + t_{read/write}$
    - t_{seek} = 최대 탐구 시간 , $t_{latency}$ = 최대 회전지연 시간 ,
      $t_{read/write}$ = 250개의 레코드로 구성된 한 블록을 읽거나 쓰는데 걸리는 시간
  - t_{InSort} = 750개의 레코드를 내부적으로 정렬하는 시간
  - $n*t_{merge}$ = 입력 버퍼에서 출력 버퍼로 n개의 레코드를 합병하는데 걸리는 시간

연산	시간
(1) 18블록의 입력 판독: $18t_{IO}$ + 내부 정렬: $6t_{InSort}$ + 18블록 기록: $18t_{IO}$	$36t_{IO}+6t_{InSort}$
(2) 1-6런을 쌍으로 합병	$36t_{IO}+4500t_{merge}$
(3) 두개의 1500 레코드로 된 런을 합병 (12 블록)	$24t_{IO}+3000t_{merge}$
(4) 3000 레코드의 런과 1500 레코드의 런을 합병	$36t_{IO}+4500t_{merge}$
디스크 정렬 예제에 대한 전체 시간	$132t_{IO} + 12000t_{merge} + 6t_{InSort}$

- **2원 합병보다 높은 차수의 합병을 이용하면 런에 대해서 일어나는 합병 패스 수를 줄일 수 있음**
- **입력, 출력, 합병을 병렬적으로 처리하기 위해서 적당한 버퍼-관리 방법이 필요**

외부 정렬 (6)

- **k-원 합병**
  - m개의 런이 있을 때의 합병 트리
    - $\lceil \log_2 m \rceil + 1$ 의 레벨을 가짐
    - 총 $\lceil \log_2 m \rceil$ 패스 필요 → 고차 합병을 사용하여 $\lceil \log_k m \rceil$ 으로 줄일 수 있음
  - k-원 합병은 비교 횟수가 많아지는 문제점이 발생
    - k가 클 경우(k>=6)에는 패자트리를 이용하여 비교횟수 감소 가능

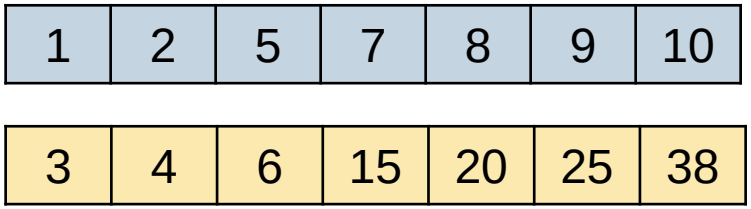


16개 런에 대한 4-원 합병
2-원인 경우에 데이타 패스가 4였지만 2로 줄어듬

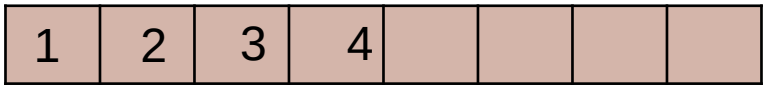
외부 정렬 (7)

- k개의 런이 k-원 합병에 의해 한꺼번에 합병되기 위해
  - k개의 입력 버퍼와 1개의 출력 버퍼 필요할 것
    → 입출력과 내부 합병을 병렬로 처리하기엔 불충분

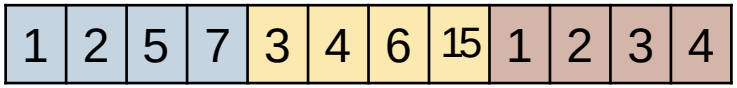
입력 디스크



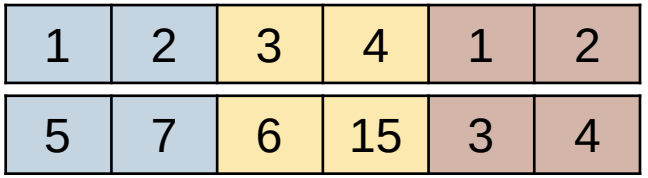
출력 디스크



메모리 (방식 1)



메모리 (방식 2)

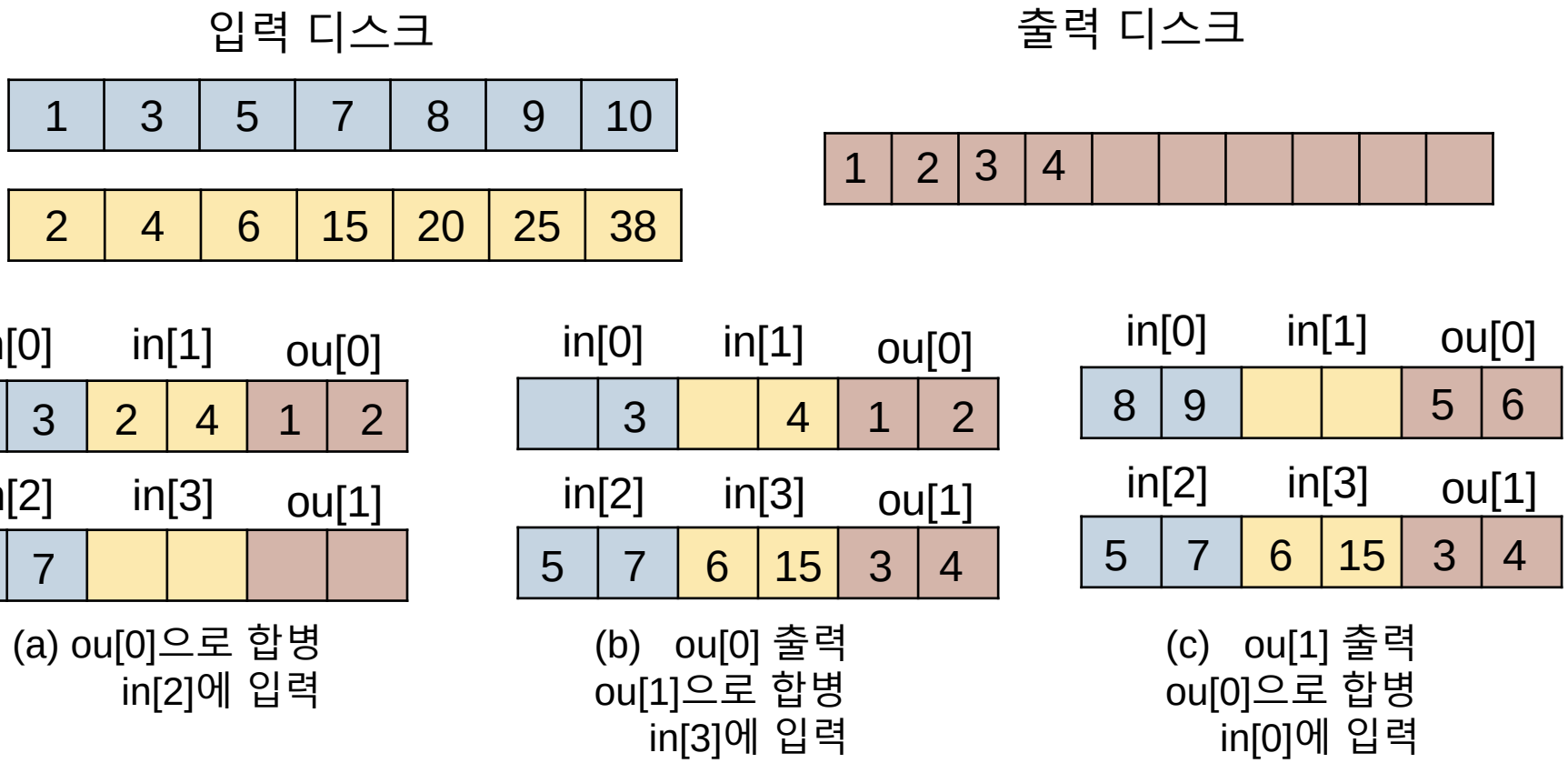


- 두 개의 출력 버퍼가 있으면 해결 됨
  - 하나가 출력되는 동안 다른 하나에 레코드들이 합병됨
- 입력 시간의 지연은 2k개의 입력 버퍼를 사용하면 해결 됨

∴ 단순히 한 개의 런에 두 개의 버퍼를 배당하는 것은 문제의 해결 방법이 아님

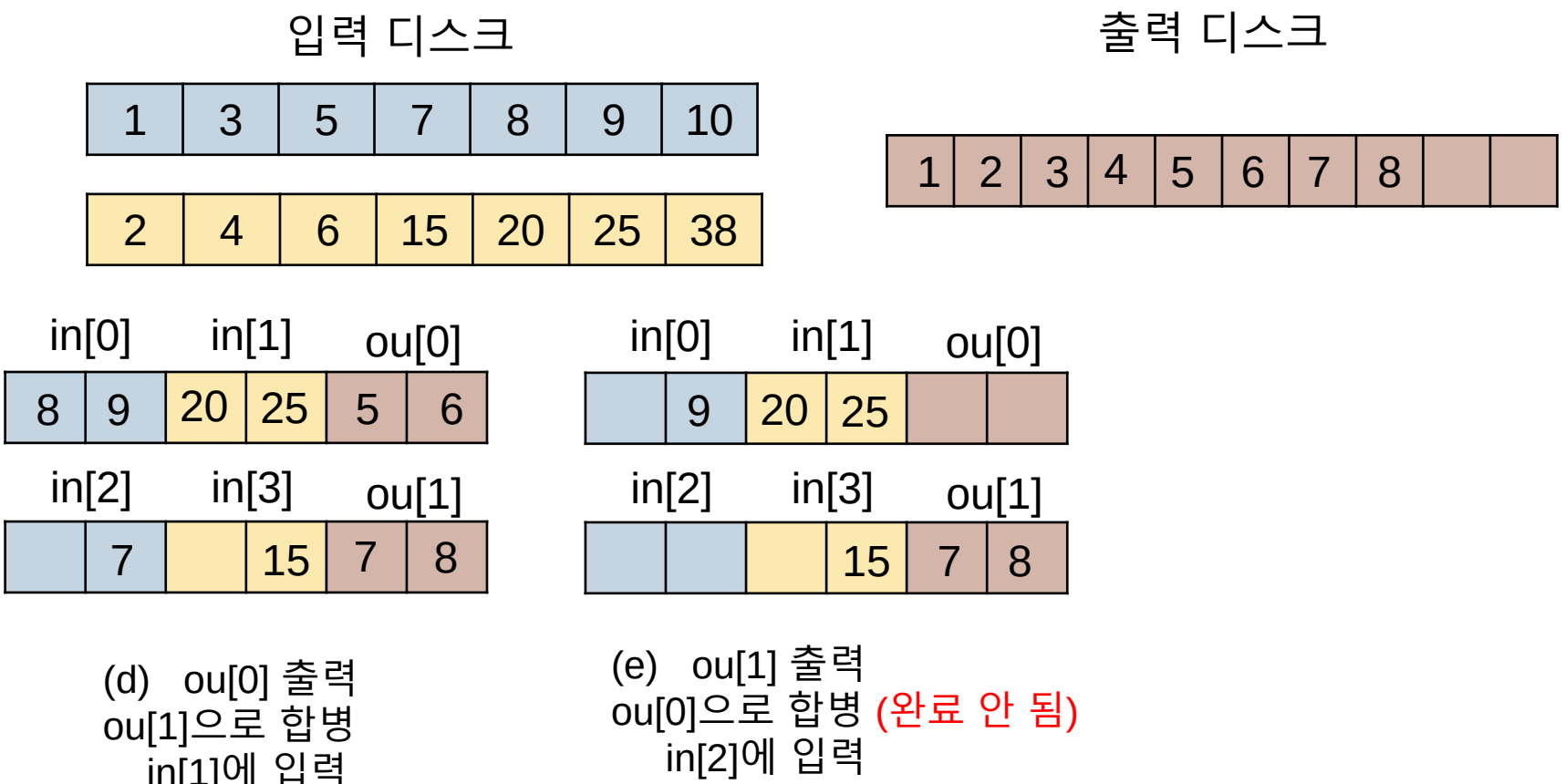
외부 정렬 (8)

- **고정 버퍼 할당의 불충분의 예 (1)**
  - 2-원 합병을 4개의 입력버퍼 in[i](0≤i<3),
    2개의 출력 버퍼 ou[0], ou[1]을 이용해서 수행
  - 각 버퍼에는 두 개의 레코드를 담을 수 있음



외부 정렬 (9)

- **고정 버퍼 할당의 불충분의 예 (2)**



- ∴ 버퍼들을 필요에 따라 어떤 런에도 할당할 수 있도록 해야 함

런 당 두 고정 버퍼 할당인 연속적 병렬 연산에 불충분함을 보여주는 예

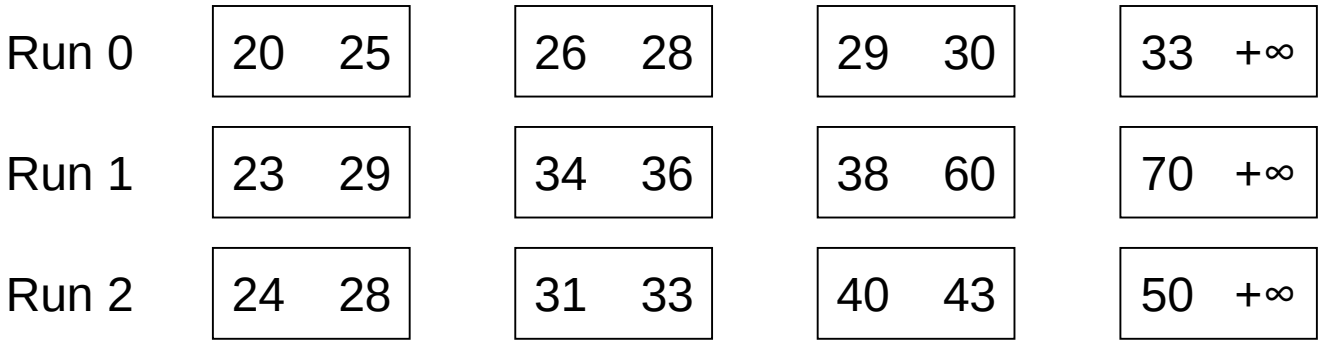
외부 정렬 (10)

- **버퍼 할당 방법**
  - 각 런의 레코드를 가지는 입력 버퍼가 최소 하나가 늘 존재
  - 남아있는 버퍼는 우선순위에 따라 할당하여 채워짐
    - k-원 합병 알고리즘에 의해 입력 버퍼에 있는 레코드들이 가장 먼저 소모되는 런이 다음 버퍼로 채워지는 런임
    - 런의 최대 값이 가장 작은 런(nextBlockFrom)의 버퍼가 먼저 소모 됨
    - 키가 똑같을 때는 더 작은 인덱스의 런을 우선함
  - 같은 런으로부터 입력되는 모든 버퍼 레코드는 큐로 처리

- **컴퓨터 병렬 처리 능력에 대한 가정**
  - (1) 디스크 드라이브는 두 개, 입출력 채널은 동시에 각 디스크에 대해 판독/기록 가능
  - (2) I/O 장치와 메모리 블록 사이에 데이타 전송 중이면 CPU는 동일한 구역의 메모리 블록을 참조 못함
  - (3) 입출력 버퍼는 모두 같은 크기

외부 정렬 (11)

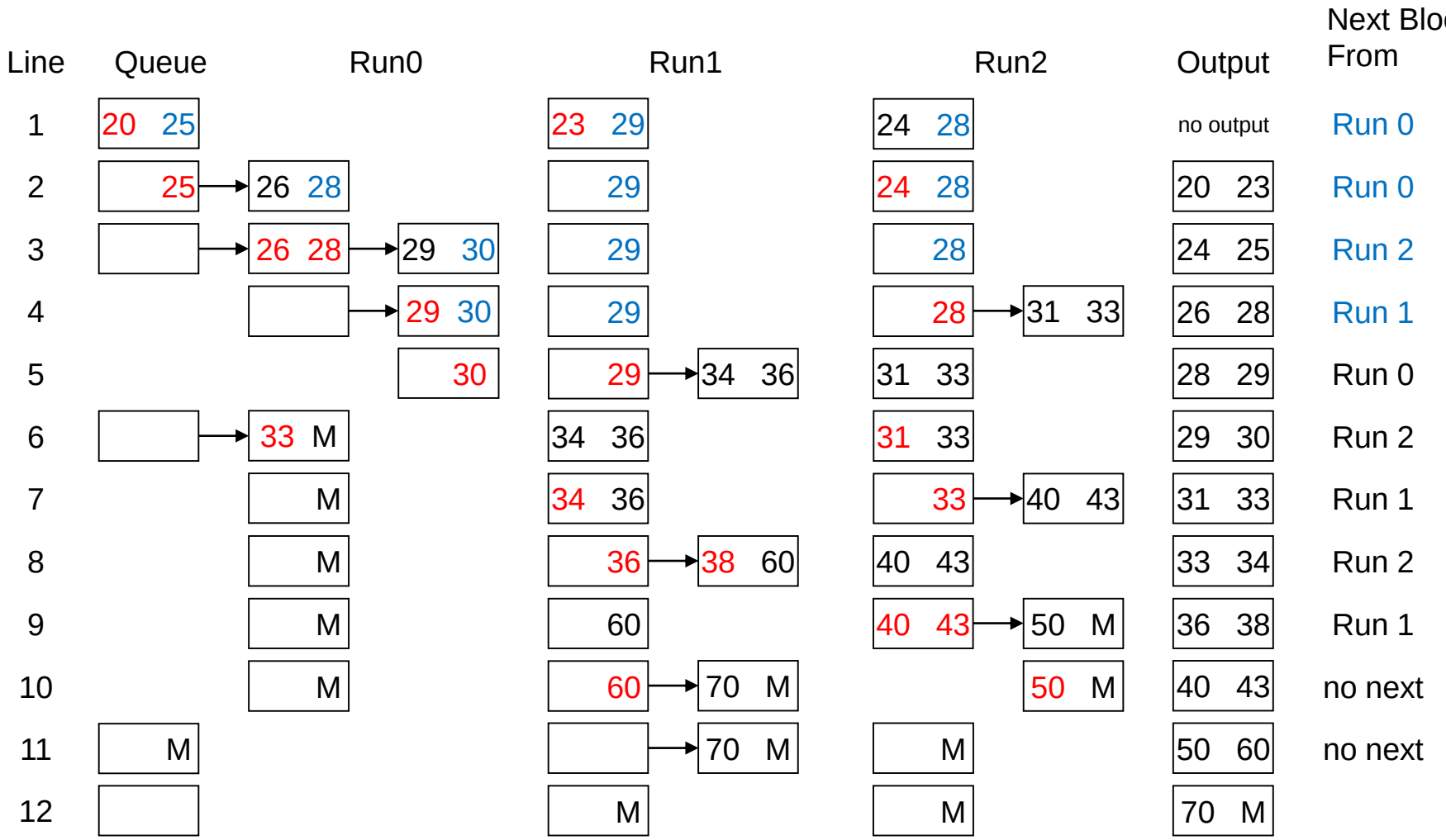
- **세 개의 런으로 3-원 합병을 수행하는 예**
  - 각 런은 두 개의 레코드로 된 네 개의 블록으로 구성
  - 모든 런에서 네 번째 블록의 마지막 레코드의 키 : +∞
  - 여섯 개의 입력 버퍼와 두 개의 출력 버퍼



세 개의 런

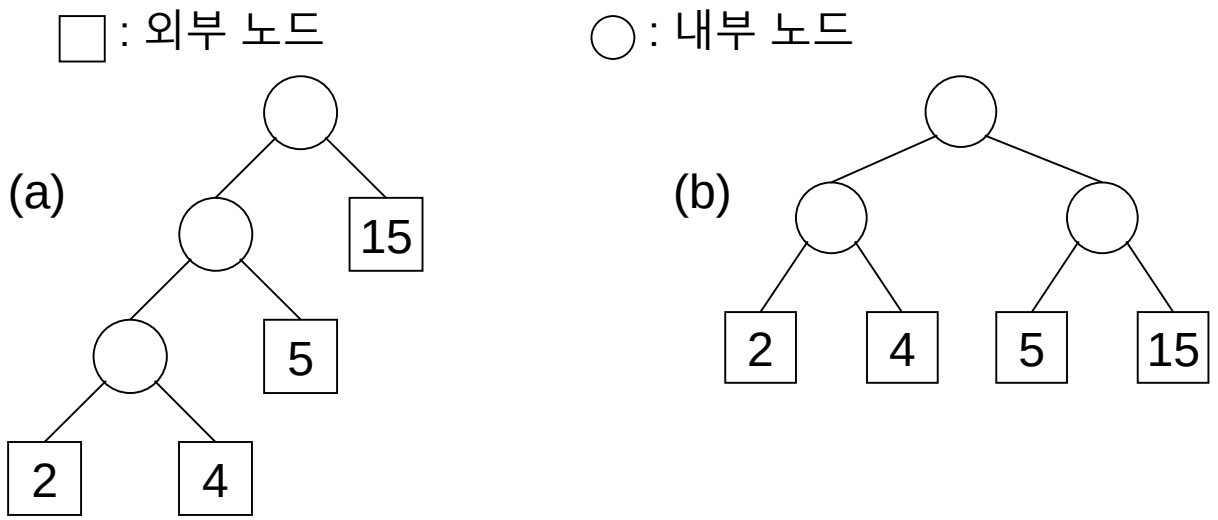
외부 정렬 (12) - 버퍼링(Buffering)의 예

- 앞의 예에서 입력 버퍼 큐의 상태, 다음 블록이 읽혀지고 있는 런, 버퍼링 알고리즘의 3단계~8단계까지 루프의 반복이 시작될 때마다 출력되는 출력 버퍼의 상태



외부 정렬 (13) - 런의 최적 합병

- **여러 런들이 상이한 크기를 갖는 경우**
  - (ex) 길이가 각각 2,4,5,15인 네 개의 런
    - 2원 합병 기법을 이용하여 합병하는 두 가지 방법



(a) 어떤 레코드는 한 번만 합병, 어떤 레코드는 최대 3번까지 합병
길이 2와 4인 런은 디스크에서 6번 읽고 쓰고, 5인 런은 4번, 15인 런은 2번 읽고 씀.
(2+4)*6 + 5*4 + 15*2 = 35 + 20 + 30 = 85번의 디스크 IO

(b) 각 레코드는 정확히 두 번씩 합병
모든 런이 디스크에서 4번 읽고 쓰여짐. (2+4+5+15)*4 = 104번의 디스크 IO

- **Huffman Algorithm을 적용하면 최적 합병 방식을 결정할 수 있음**

요약

- 선택정렬은 아직 정렬되지 않은 부분의 배열 원소들 중에서 최솟값을 선택하여 정렬된 부분의 바로 오른쪽 원소와 교환하는 정렬알고리즘
- 삽입정렬은 수행과정 중에 배열이 정렬된 부분과 정렬되지 않은 부분으로 나뉘어지며, 정렬되지 않은 부분의 가장 왼쪽의 원소를 정렬된 부분에 삽입하는 방식의 정렬알고리즘
- 쉘정렬은 전처리과정을 추가한 삽입정렬이다. 전처리과정이란 작은 값을 가진 원소들을 배열의 앞부분으로 옮겨 큰 값을 가진 원소들이 배열의 뒷부분으로 이동
- 힙정렬은 입력에 대해 최대힙을 만들어 루트노드와 힙의 가장 마지막 노드를 교환하고, 힙 크기를 1 감소시킨 후에 루트노드로부터 downheap을 수행하는 과정을 반복하여 정렬하는 알고리즘

- 합병정렬은 입력을 반씩 두 개로 분할하고, 각각을 재귀적으로 합병정렬을 수행한 후, 두 개의 각각 정렬된 부분을 합병하는 정렬알고리즘
- 퀵정렬은 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 Ω(NlogN)
- 기수정렬은 키를 부분적으로 비교하는 정렬 LSD/MSD기수정렬의 수행시간은 O(d(N+R))
- 외부정렬이란 보조기억장치에 있는 대용량의 데이터를 정렬하는 알고리즘으로 합병을 사용하여 정렬 수행