

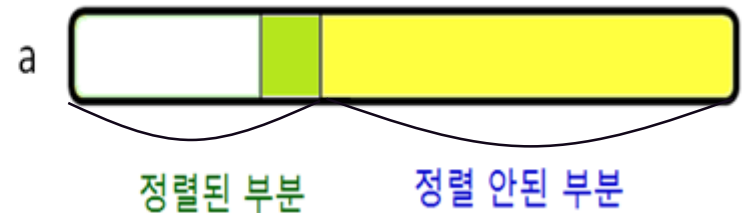
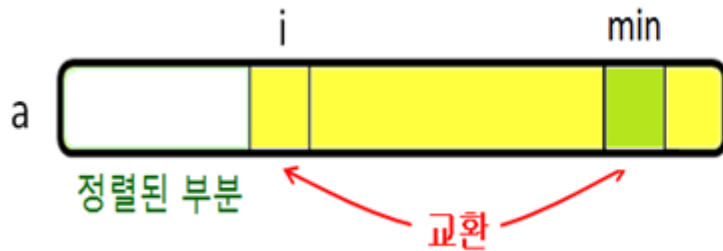
자바와 함께 하는 정렬알고리즘

# 정렬

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

## 8.1 선택정렬

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



- (a) 배열  $a$ 의 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  $i$ 를 1 증가시키며, 이를 반복적으로 수행

# Selection 클래스

```
package com.company;

public class Selection {

    public void prtArr(int[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            System.out.print(a[i] + " ");
        }
    }

    private void swap(int[] a, int i, int j) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }

    public void selectionSort(int[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int min = i;

            for (int j = i; j < N; j++) {
                if (a[min] > a[j])
                    min = j;
            }
            swap(a, i, min);
        }
    }
}
```

```
package com.company;

public class Main {

    public static void main(String[] args) {
        // write your code here
        int[] a = {7, 10, 2, 3, 9, 8, 22, 17};

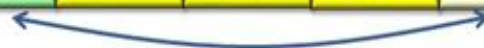
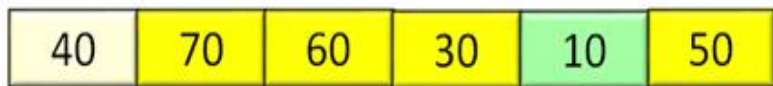
        Selection s = new Selection();
        s.selectionSort(a);
        s.prtArr(a);
    }
}
```

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



$i = 0$

min



교환

$i = 1$

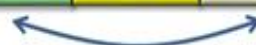
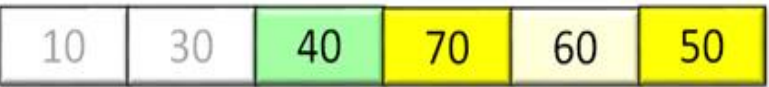
min



교환

$i = 2$

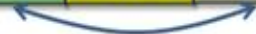
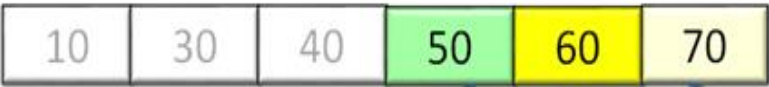
min



교환

$i = 3$

min



교환

min



교환

$i = 4$

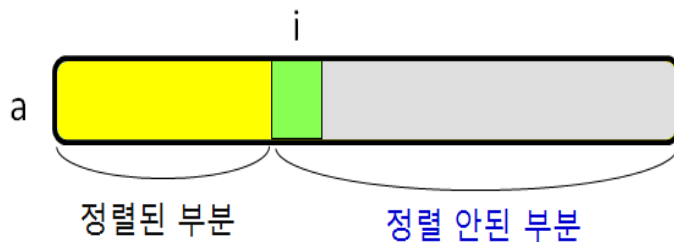
# 수행시간

- 선택정렬은 루프가 1 번 수행될 때마다 정렬되지 않은 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

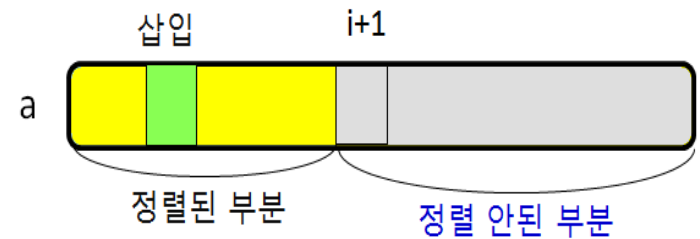
$$(N-1) + (N-2) + (N-3) + \dots + 2 + 1 = \frac{N(N-1)}{2} = O(N^2)$$

## 8.2 삽입정렬

- 삽입정렬(Insertion Sort)은 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 원소인 50을 정렬된 부분에 삽입하는 과정

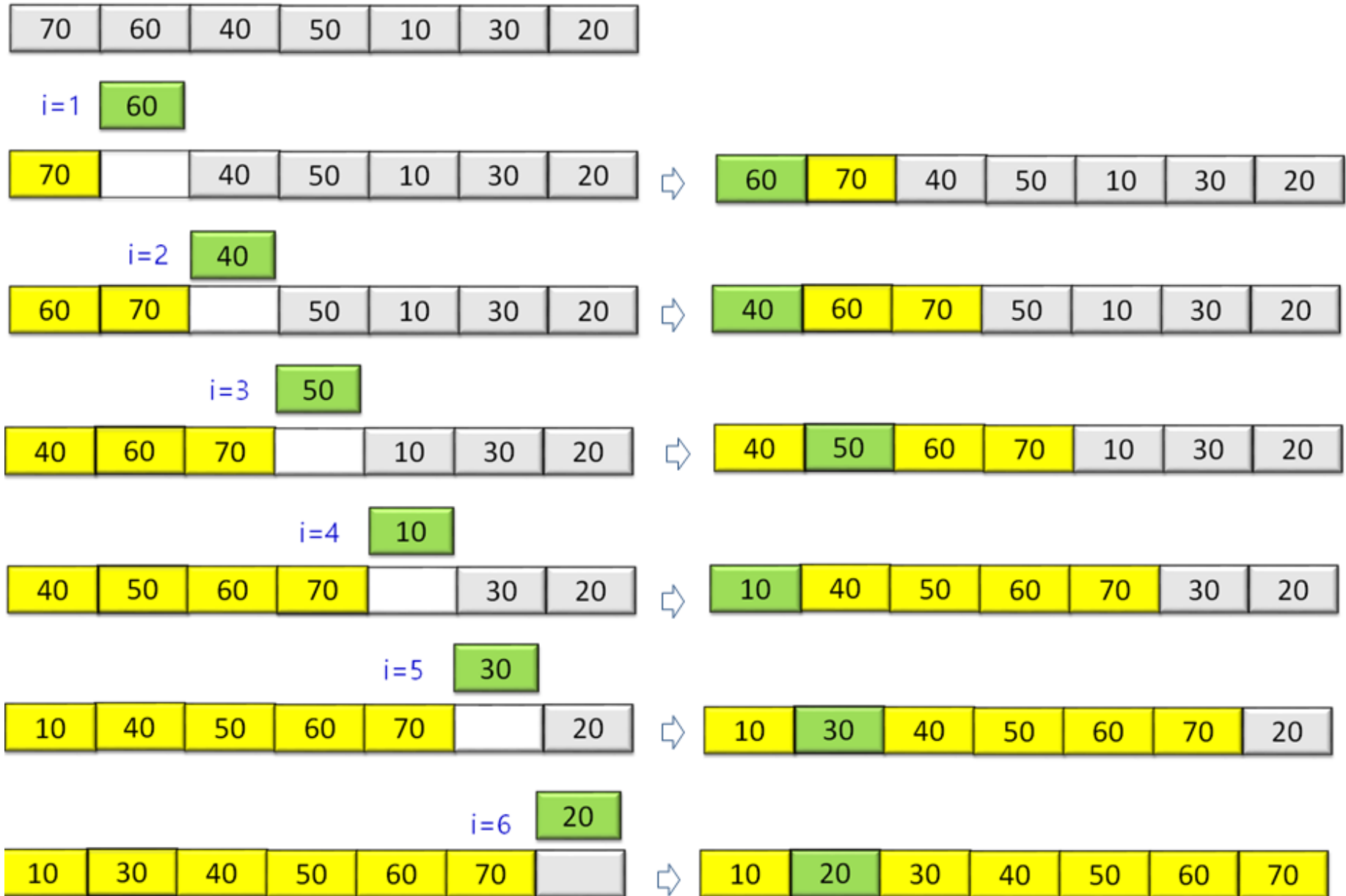


# Insertion 클래스

```
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(최선경우)

N-1번 비교하면 정렬을 마침 =  $O(N)$

- 입력이 역으로 정렬된 경우 (최악경우)

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

- 최악경우 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 셸정렬

- 셸(Shell Sort)정렬은 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

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 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 1로하여 정렬
- h = 1인 경우는 삽입정렬과 동일

# 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:  $h /= 3$ 은 h 값을  $1/3$ 로 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



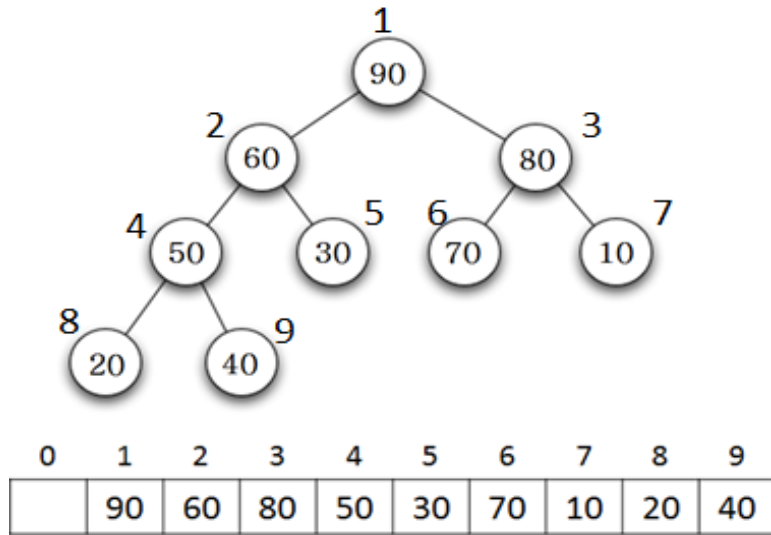
10	25	35	30	55	70	40	50	65	95	90	80
----	----	----	----	----	----	----	----	----	----	----	----

교환

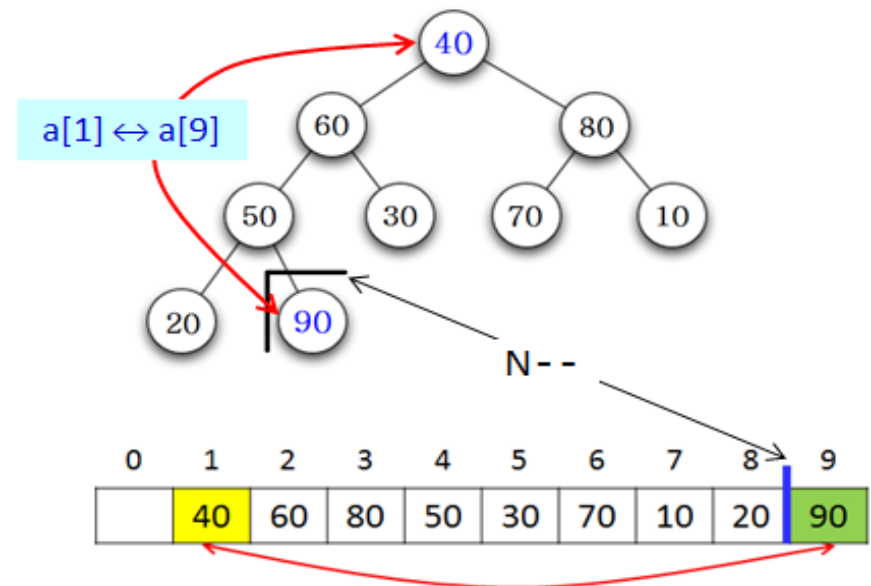
## 8.4 힙정렬

- 힙정렬(Heap Sort)은 힙 자료구조를 이용하는 정렬
- 먼저 배열에 저장된 데이터의 키를 우선순위로 하는 최대힙(Max Heap)을 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

# 루트노드와 힙의 마지막 노드 교환 후 downheap 연산 수행 과정

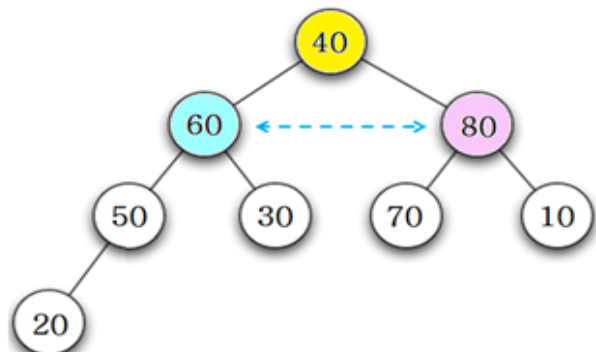


(a) 입력배열과 최대힙



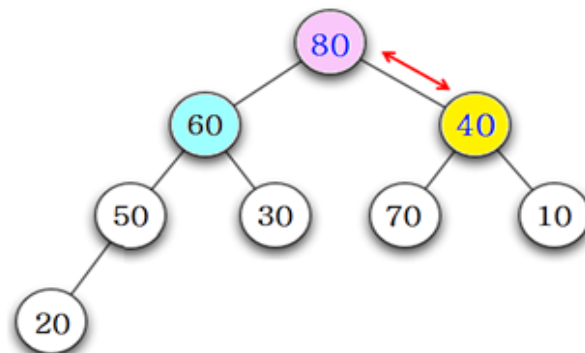
(b) 루트노드와 마지막 노드 교환

downheap()



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

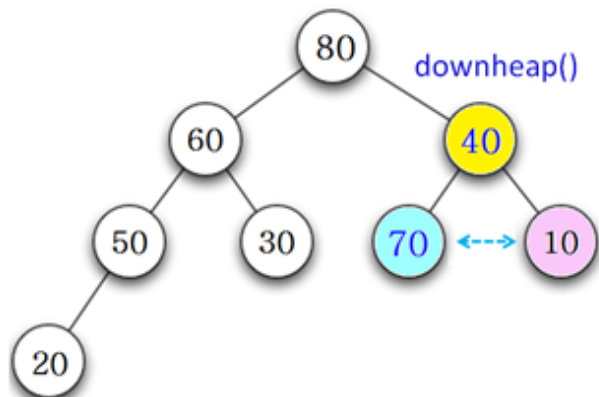
(c) 루트노드의 두 자식 비교



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

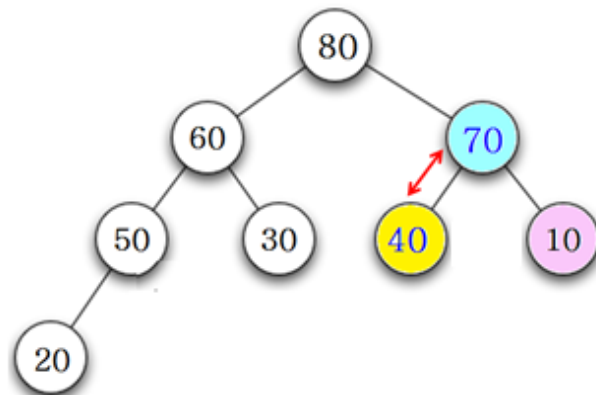
(d) 루트노드와 자식 승자와 교환

downheap()



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 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  $a[1] \sim a[8]$ 에 대해 동일한 과정을 반복 수행하여 힙 크기가 1이 되었을 때 종료

## [예제] 앞선 예제에 이어서 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
--	----	----	----	----	----	----	----	----	----

## Heap 클래스

```
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     }
21     // isLess(), swap() 메소드 선언, Selection 클래스와 동일함
22 }
```

- 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

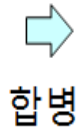
- 먼저 상향식(Bottom-up)으로 힙을 구성:  $O(N)$  시간
- 루트와 힙의 마지막 노드를 교환한 후 `downheap()` 수행:  $O(\log N)$  시간
- 루트와 힙의 마지막 노드를 교환하는 횟수:  $N-1$  번
- 총 수행시간:  $O(N) + (N-1) * O(\log N) = O(N \log N)$
- 힙정렬은 어떠한 입력에도 항상  $O(N \log N)$  시간이 소요
- 루프 내의 코드가 길고, 비효율적인 캐시메모리 사용에 따라 특히 대용량의 입력을 정렬하기에 부적절
- C/C++ 표준 라이브러리(STL)의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

## 8.5 합병정렬

- 합병정렬(Merge Sort)은 크기가  $N$ 인 입력을  $1/2N$ 크기를 갖는 입력 2 개로 분할하고, 각각에 대해 재귀적으로 합병정렬을 수행한 후, 2 개의 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

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



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

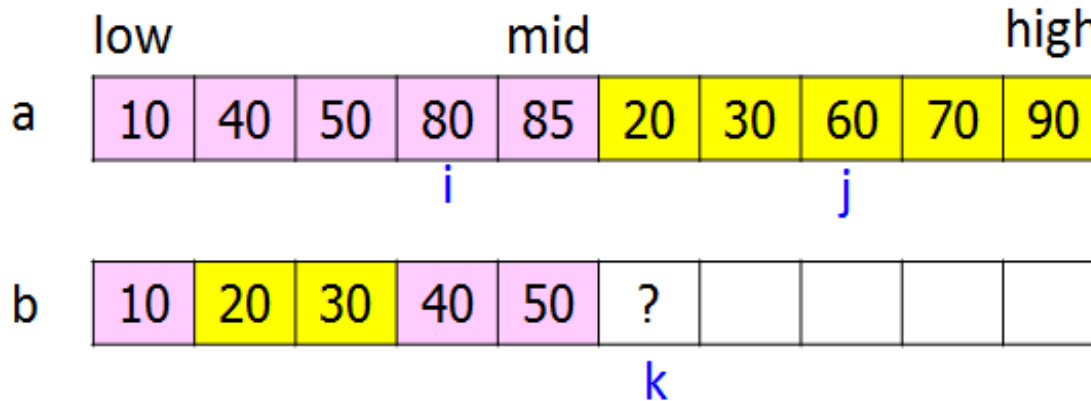
## Merge 클래스

```
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  $a$ 와 같은 크기의 보조배열  $b$  선언
- Line 22: line 13의 `sort()`를 호출하는 것으로 정렬 시작
- Line 15: 정렬할 배열 부분  $a[\text{low}] \sim a[\text{high}]$ 를 1/2로 나누기 위해 중간 인덱스  $\text{mid}$ 를 계산
- Line 16: 1/2로 나눈 앞부분인  $a[\text{low}] \sim a[\text{mid}]$ 를 `sort()`의 인자로 넘겨 재귀호출
- Line 17: 뒷부분인  $a[\text{mid}+1] \sim a[\text{high}]$ 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:  $a[\text{low}] \sim a[\text{mid}]$  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 11에서 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

# 수행시간

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

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

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

## 성능향상방법(1)

- 합병정렬은 재귀호출을 사용하므로 입력 크기가 1이 되어야 합병을 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;  
}
```

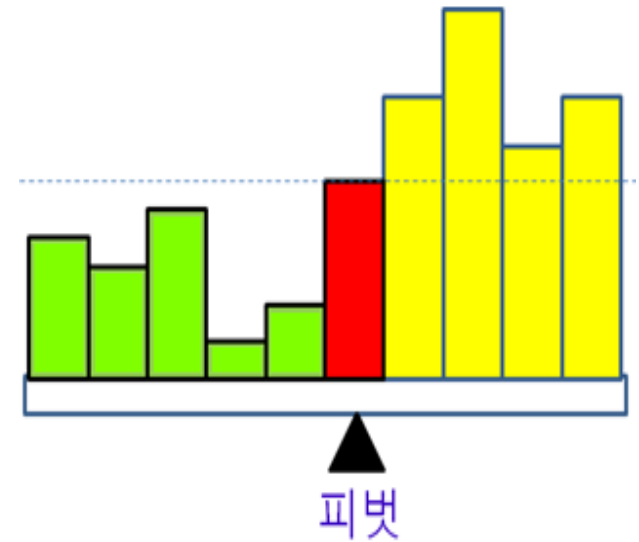
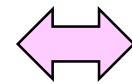
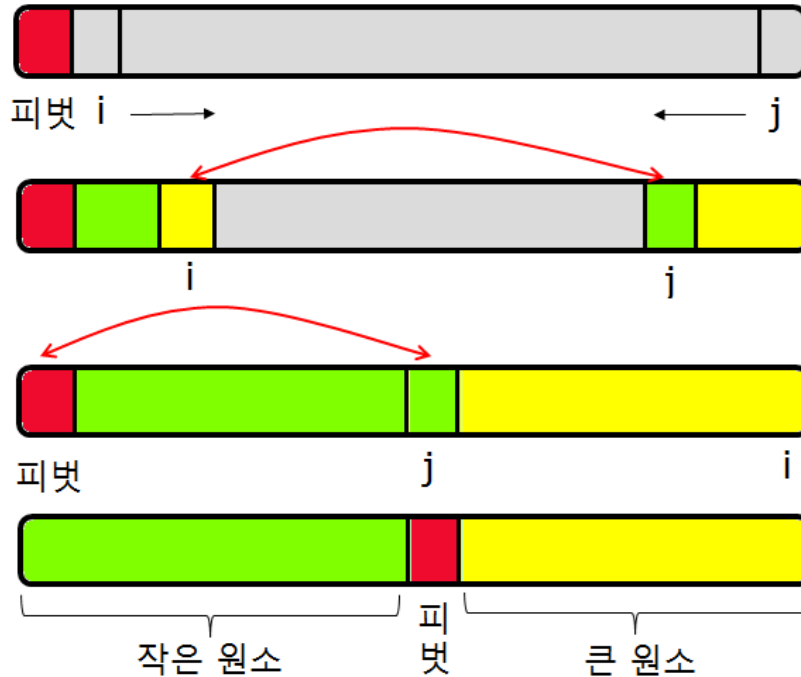
## 성능향상방법(2)

- 합병정렬에서는 입력 크기가 작아지면 합병하기 위한 두 개의 리스트가 이미 합병되어 있을 가능성이 높아짐
- 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;
```

## 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





## Quick 클래스

```
01 import java.lang.Comparable;
02 public class Quick {
03     public static void sort(Comparable[] a) {
04         sort(a, 0, a.length - 1);
05     }
06     private static void sort(Comparable[] a, int low, int high) {
07         if (high <= low) return;
08         int j = partition(a, low, high);
09         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(p, a[i])) i++; // 피벗과 같거나 작으면
18             while (j >= pivot && isless(p, a[j])) j--; // 피벗보다 크면
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     }
}
```

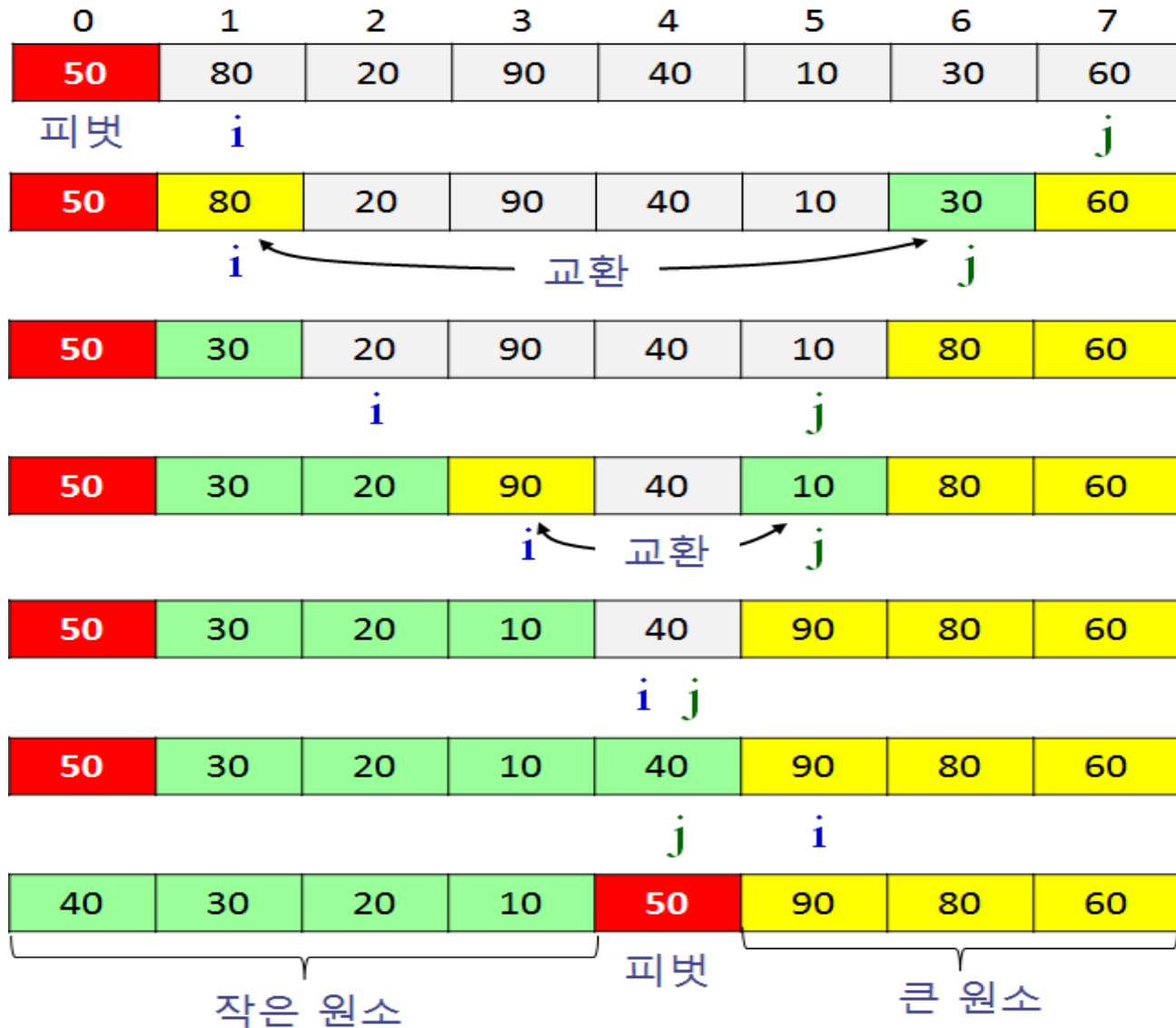
```

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 }

```

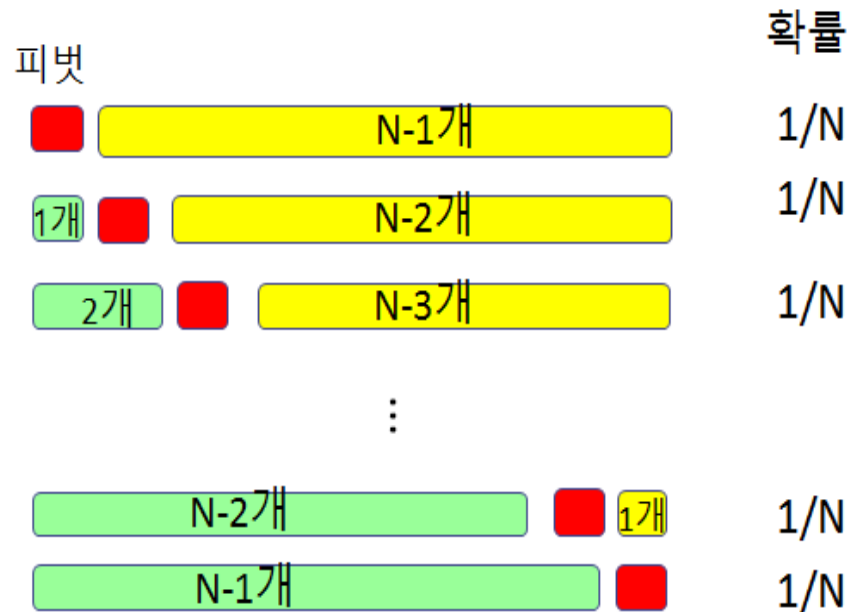
- Line 04: sort(a, 0, a.length-1)로 호출 시작
- line 08: 피벗인 a[low]를 기준으로 a[low] ~ a[j-1]과 a[j+1] ~ a[high]로 분할하며, a[j]에 피벗이 고정
- Line 09: a[low] ~ a[j-1]을 재귀호출하여 정렬
- Line 10: a[j+1] ~ a[high]를 재귀호출하여 정렬

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



# 수행시간

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



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

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

## 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;  
}
```

## 성능향상방법[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 방법이라함
- 가장 왼쪽(low), 중간(mid), 그리고 가장 오른쪽(high) 원소들 중에서 중간값을 찾는 것으로도 알려져 있음

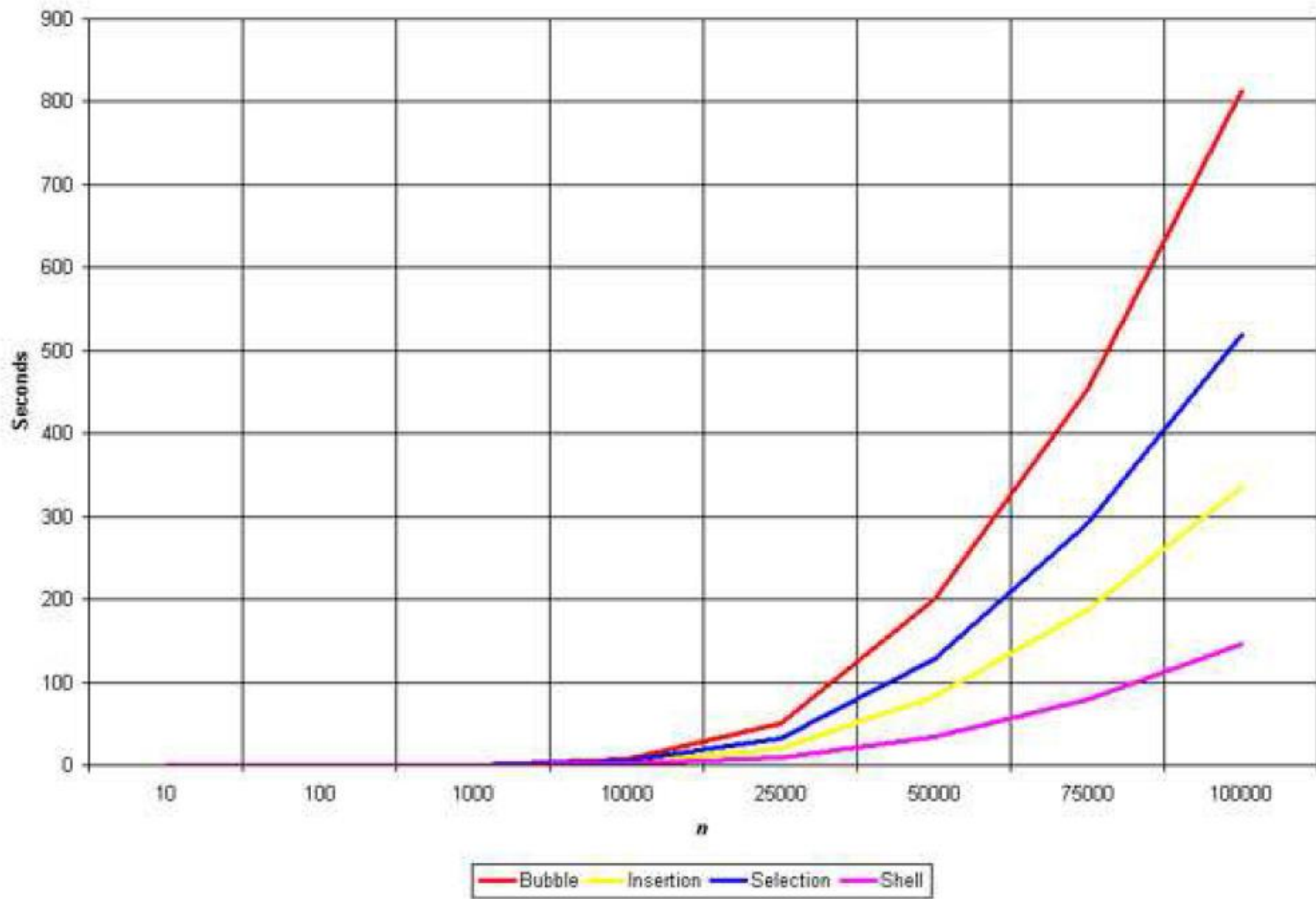
- 퀵정렬은 평균적으로 빠른 수행시간을 가지며, 보조배열을 사용하지 않음
- 최악경우 수행시간이  $O(N^2)$ 이므로, 성능 향상 방법들을 적용하여 사용하는 것이 바람직함
- 퀵정렬은 원시 타입(Primitive Type) 데이터를 정렬하는 자바 Standard Edition 6의 시스템 sort에 사용
- C-언어 라이브러리의 qsort, 그리고 Unix, g++, Visual C++, Python 등에서도 퀵정렬을 시스템 정렬로 사용
- 자바 SE 7에서는 2009년에 Yaroslavskiy가 고안한 이중피벗퀵(Dual Pivot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

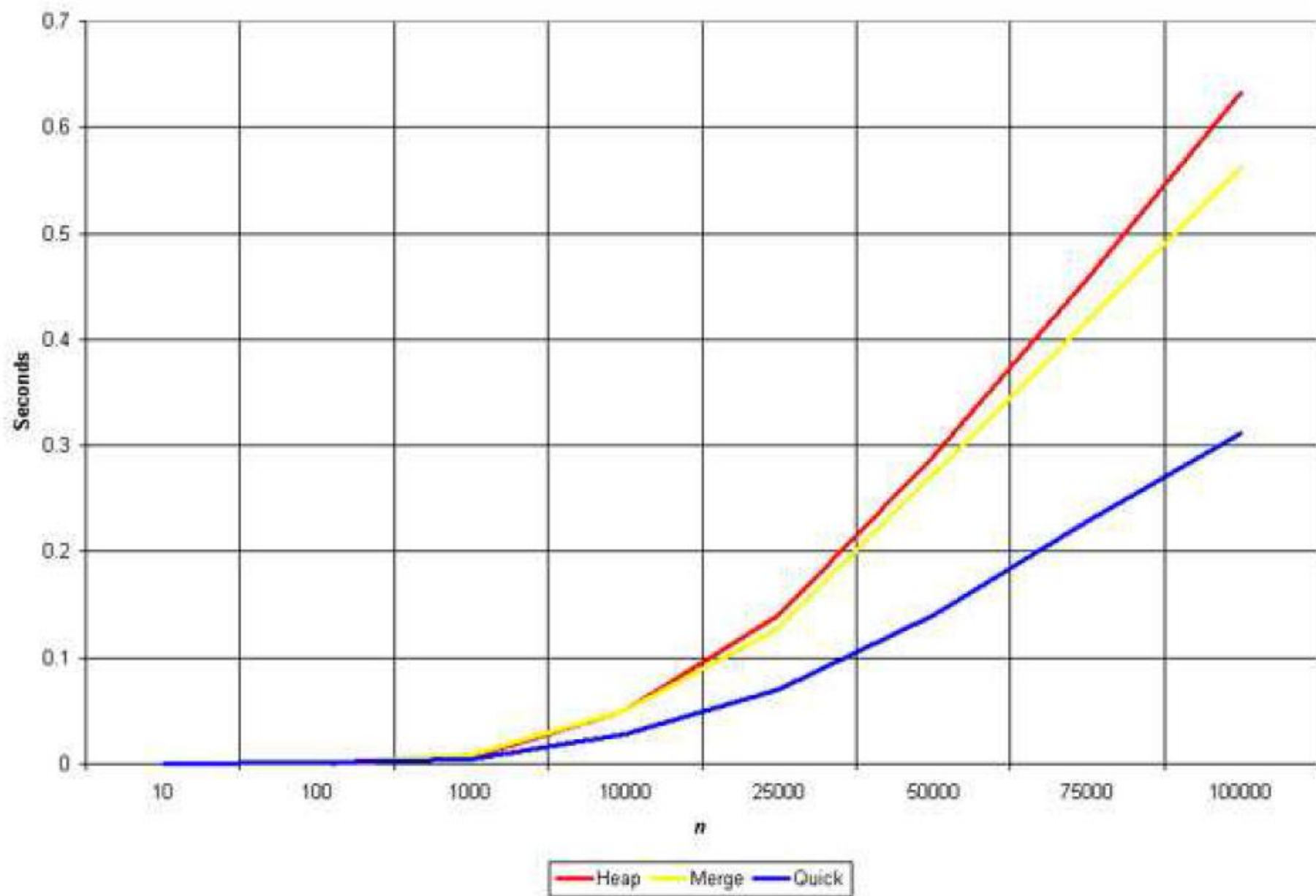


## 정렬알고리즘 성능 비교

	최선경우	평균경우	최악경우	추가공간	안정성
선택정렬	$N^2$	$N^2$	$N^2$	$O(1)$	X
삽입정렬	$N$	$N^2$	$N^2$	$O(1)$	O
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 설명은 부록 VI





- **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]

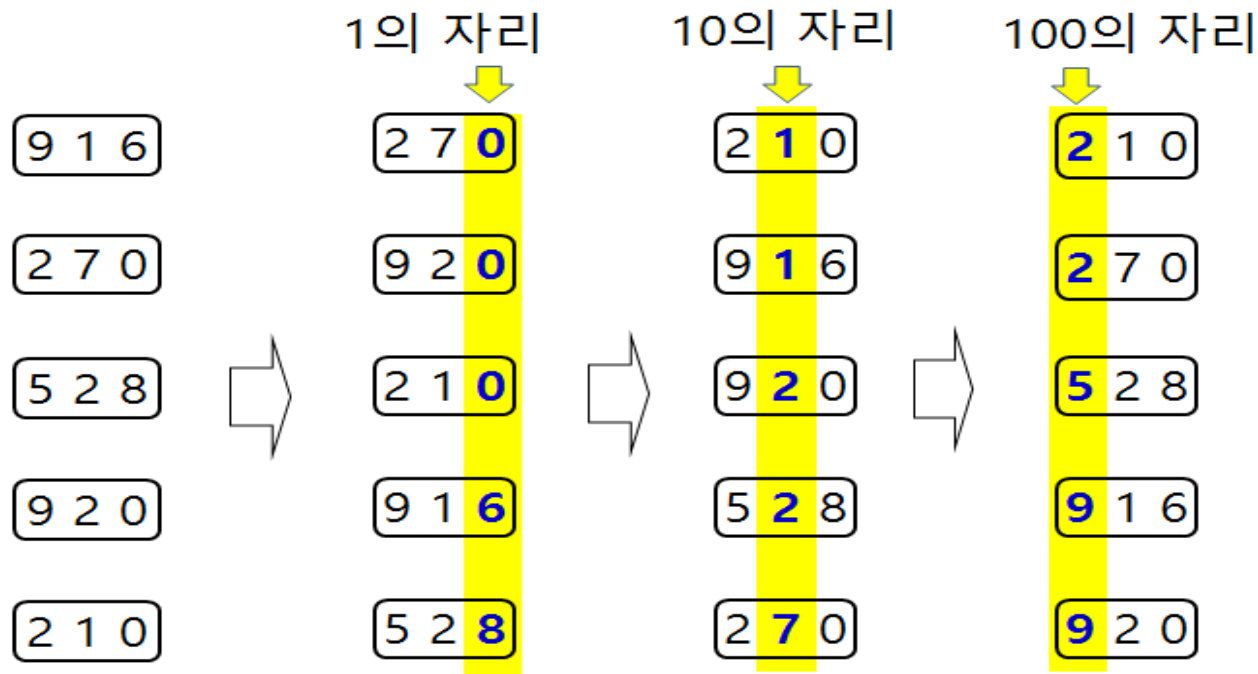
unstable 정렬      [10 H]   [20 E]   [20 B]   [40 D]   [50 G]   [60 C]   [60 F]   [90 A]

## 8.8 기수정렬

- 기수정렬(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) 기수정렬**: 자릿수 비교를 최하위 숫자로부터 최상위 숫자 방향으로 정렬
- **MSD(Most Significant Digit) 기수정렬**: 반대 방향으로 정렬

## 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



- 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

	0	1	2	3	4	5	6	7	8	9	10	11	
a	2	0	5	0	3	2	5	2	3	1	0	2	정렬 전

	0	1	2	3	4	5
bucket	3	1	4	2	0	2

	0	1	2	3	4	5	6	7	8	9	10	11	
a	0	0	0	1	2	2	2	2	3	3	5	5	정렬 후

# 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 }
```

- 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.

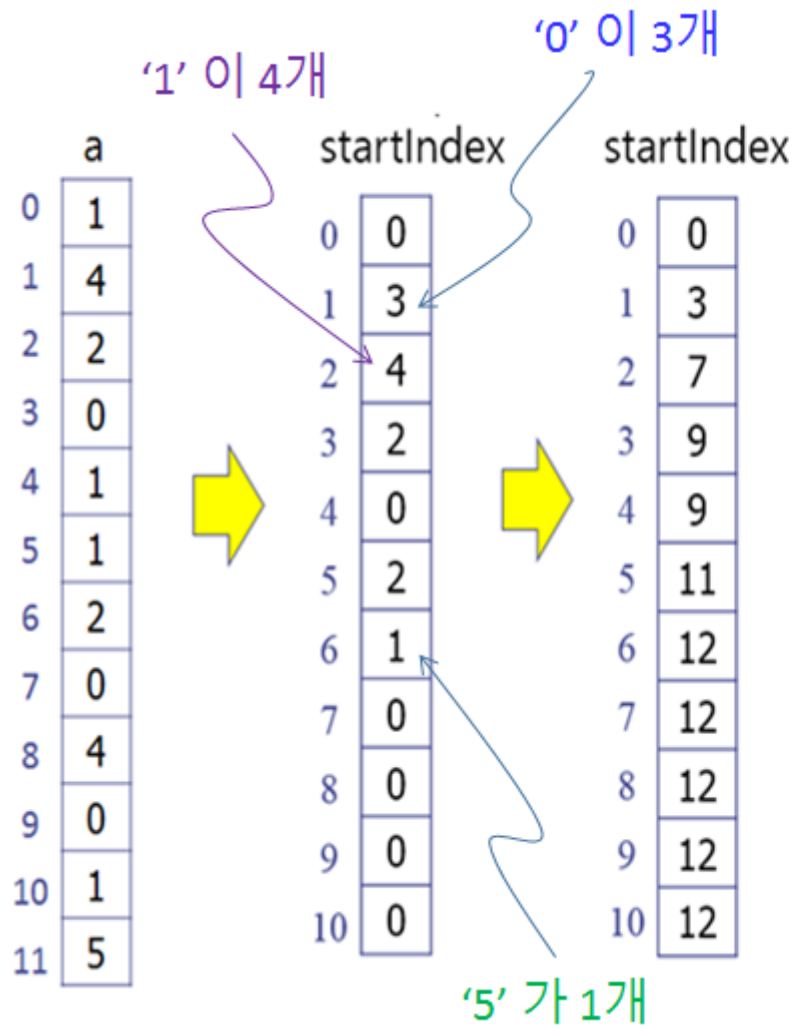
a			t			a			t			a			t		
2	5	1	4	3	0	4	3	0	3	0	1	3	0	1	0	0	2
4	3	0	5	4	0	5	4	0	4	0	1	4	0	1	0	1	0
3	0	1	0	1	0	0	1	0	0	0	2	0	0	2	0	2	2
5	4	0	2	5	1	2	5	1	2	0	4	2	0	4	1	1	5
5	5	1	3	0	1	3	0	1	0	1	0	0	1	0	1	2	4
4	0	1	5	5	1	5	5	1	1	1	5	1	1	5	2	0	4
0	0	2	4	0	1	4	0	1	0	2	2	0	2	2	2	5	1
0	1	0	0	0	2	0	0	2	1	2	4	1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

# LSDSort 클래스

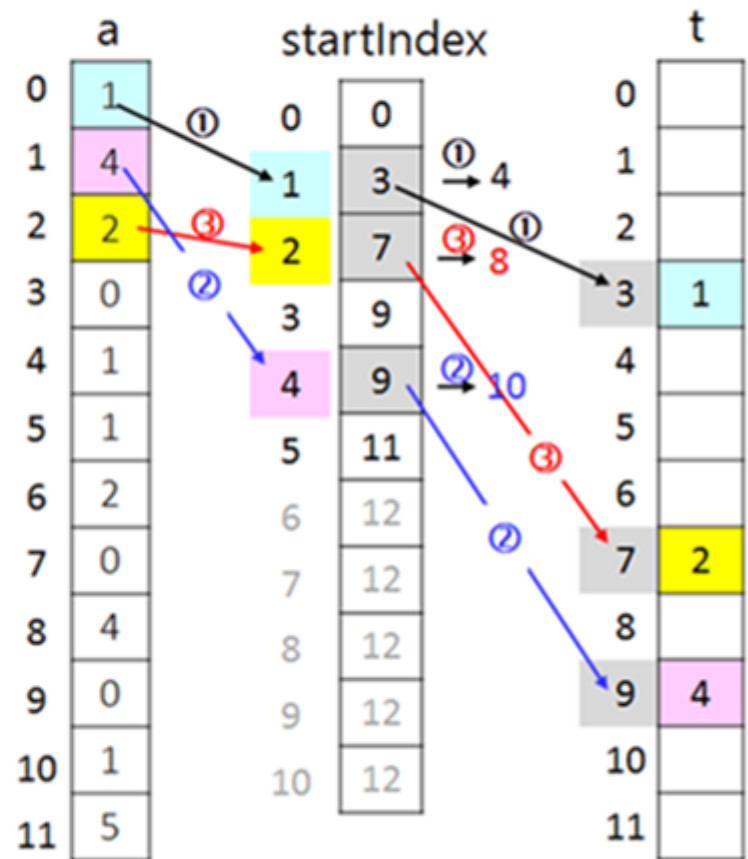
```
01 public class LSDsort {
02     public static void sort(int[] a) {
03         int R = 10;
04         int N = a.length;
05         int[] t = new int[N];
06         for (int k = 10; k <= 1000; k*=10) {
07             int[] startIndex = new int[R+1];
08             for (int i = 0; i < N; i++) // 빈도수 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 저장
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  $k = 10$ 일 때 1의 자리,  $k = 100$ 일 때 10의 자리,  $k = 1000$ 일 때 100자리의 숫자를 차례로 처리하기 위해 3 번 반복
- Line 08~09: 빈도수를 계산하는데,  $(a[i]\%k)/(k/10)$ 가  $k = 10, 100, 1000$ 일 때 각각  $a[i]$ 의 3자리 십진수 키로부터 1, 10, 100의 자리를 추출
- 예를 들어,  $a[i] = 259$ 라면,
  - $k = 10$ 일 때,  $259\%10 = 9$ 이고, 9를  $(k/10) = (10/10) = 1$ 로 나누면 그대로 9가 되어, 259의 1의 자리인 '9'를 추출
  - $k = 100$ 일 때,  $259\%100 = 59$ 이고, 59를  $(k/10) = (100/10) = 10$ 으로 나눈 몫은 5이다. 따라서 259의 10의 자리인 '5'를 추출
  - $k = 1,000$ 일 때  $259\%1000 = 259$ 이고, 259를  $(1000/10) = 100$ 으로 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 `a`에 '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 `t`에 저장할 곳(인덱스) (line 12~13)



(a) 빈도수 및 누적 계산



(b) 버킷정렬

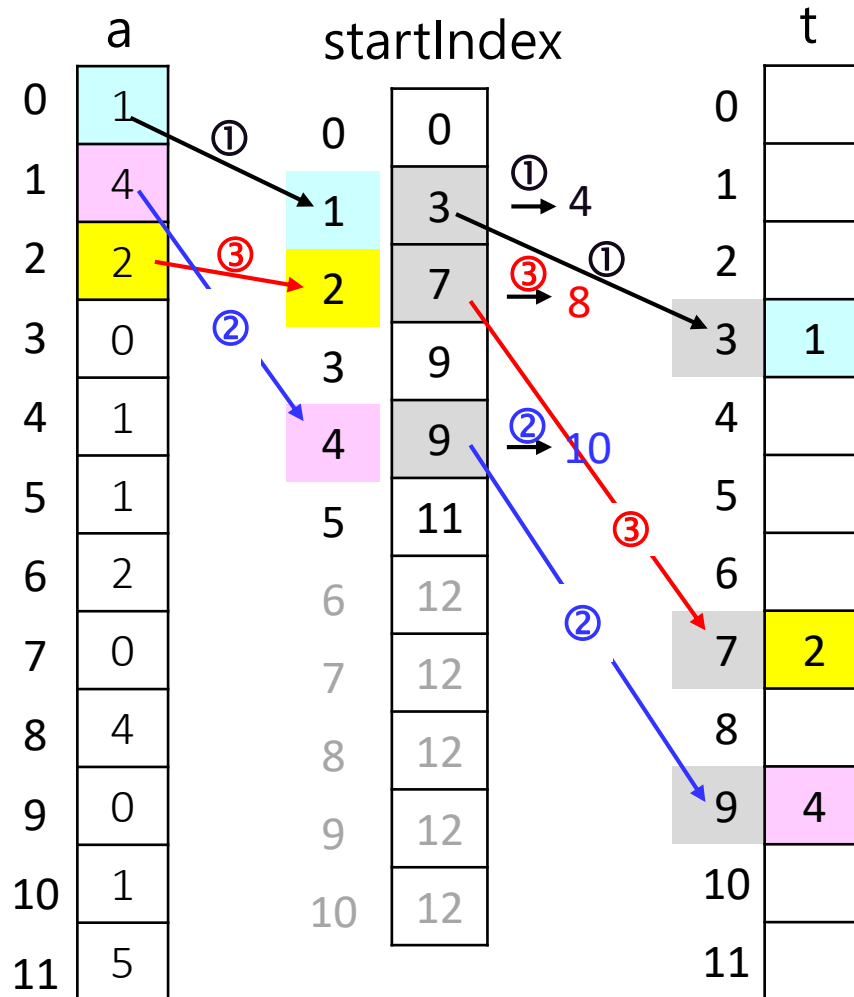
[그림 8-15] startIndex 배열 원소의 활용



// 숫자를 적절한 곳에 복사

for (int i = 0; i < N; i++) // k = 10, 100, 1000

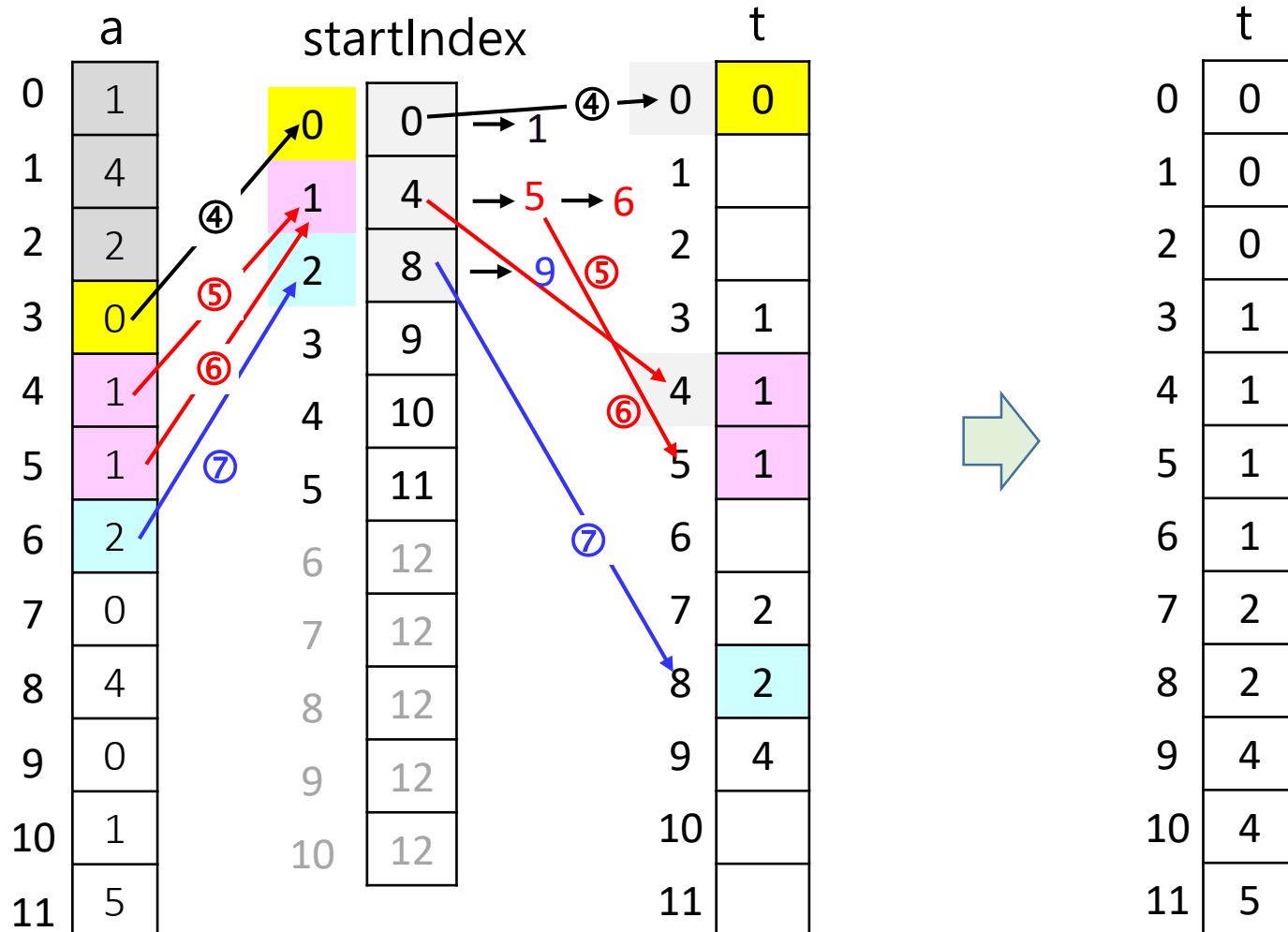
t[startIndex[(a[i]%k)/(k/10)]++] = a[i];



- [그림 8-15](b)에서는  $a[0] = 1$ 이므로  $t[\text{startIndex}[1]] = t[3]$ 에 '1'을 저장하고,  $\text{startIndex}[1] = 4$ 로 갱신하여 다음에 '1'을 저장할 시작 인덱스를 만든다.
- 다음으로  $a[1] = 4$ 이므로  $t[\text{startIndex}[4]] = t[9]$ 에 '4'를 저장하고,  $\text{startIndex}[4] = 10$ 으로 갱신
- $a[2] = 2$ 이므로  $t[\text{startIndex}[2]] = t[7]$ 에 '2'를 저장하고,  $\text{startIndex}[2] = 8$ 로 갱신
- 이와 같이  $a[11] = 5$ 를  $t[\text{startIndex}[5]] = t[11]$ 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

Console Problems Javadoc Declaration

<terminated> main (9) [Java Application] C:\Program Files\Java\jdk1.8.0\_40\bin\javaw.exe

-----1의 자리 정렬 결과-----

430 540 010 251 301 551 401 002 022 124 204 115

-----10의 자리 정렬 결과-----

301 401 002 204 010 115 022 124 430 540 251 551

-----100의 자리 정렬 결과-----

002 010 022 115 124 204 251 301 401 430 540 551

# 수행시간

- LSD 기수정렬의 수행시간은  $O(d(N+R))$

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

- Line 06의 for-루프는  $d$ 번 수행되고, 각 자릿수에 대해 line 08, 12, 14의 for-loop들이 각각  $N$ 번씩 수행되며, line 10의 for-loop는  $R$ 회 수행되기 때문

## 장단점 및 응용

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

## 응용

- GPU(Graphics Processing Unit) 기반 병렬(Parallel) 정렬의 경우 LSD 기수정렬을 병렬처리 할 수 있도록 구현하여 시스템 sort로 사용
- Thrust Library of Parallel Primitives, v.1.3.0의 시스템 sort로 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**은 입력에 대해 최대힙을 만들어 루트노드와 힙의 가장 마지막 노드를 교환하고, 힙 크기를 1 감소시킨 후에 루트노드로부터 downheap을 수행하는 과정을 반복하여 정렬하는 알고리즘
- **합병정렬**은 입력을 반씩 두 개로 분할하고, 각각을 재귀적으로 합병정렬을 수행한 후, 두 개의 각각 정렬된 부분을 합병하는 정렬알고리즘
- **퀵정렬**은 피벗보다 작은 원소들과 큰 원소들을 각각 피벗의 좌우로 분할한 후, 피벗보다 작은 원소들과 피벗보다 큰 원소들을 각각 재귀적으로 정렬하는 알고리즘
- **기수정렬**은 키를 부분적으로 비교하는 정렬  
LSD/MSD기수정렬의 수행시간은  $O(d(N+R))$