Sorting & Search

정렬

❖ 정렬(Sorting)

- ✓ 정렬은 데이터를 순서대로 나열하는 것
- ✓ 정렬 방식은 작은 것부터 큰 순서대로 나열하는 오름차순(Ascending) 정렬과 큰 것에서 작은 순서대로 나열하는 내림차순(Descending) 정렬이 있고 정렬을 하는 방법은 구현하는 알고리즘에 따라 여러 가지

기준	정렬 방식	설명
실행 방법	비교식 정렬 Comparative Sort	비교할 각 키값을 한 번에 두 개씩 비교하여 교환함으로써 정렬을 실행하는 방식
	분배식 정렬 Distribute Sort	키값을 기준으로 하여 자료를 여러 개의 부분집합으로 분해하고, 각 부분집합을 정렬함으로써 전체를 정렬하는 방식
정렬 장소	내부 정렬 Internal Sort	컴퓨터 메모리 내부에서 정렬
	외부 정렬 External Sort	메모리의 외부인 보조 기억 장치에서 정렬

정렬

❖ 정렬(Sorting)

✓ 내부 정렬(internal sort)

- 정렬할 자료를 메인 메모리에 올려서 정렬하는 방식
- 정렬 속도가 빠르지만 정렬할 수 있는 자료의 양이 메인 메모리의 용량에 따라 제한됨

구분	종류	설명
비교식	교환 방식	키를 비교하고 교환하여 정렬하는 방식(선택 정렬, 버블 정렬, 퀵 정렬)
	삽입 방식	키를 비교하고 삽입하여 정렬하는 방식(삽입 정렬, 셸 정렬)
	병합 방식	키를 비교하고 병합하여 정렬하는 방식(2-way 병합, n-way 병합)
	선택 방식	이진 트리를 사용하여 정렬하는 방식(히프 정렬, 트리 정렬)
분배식	분배 방식	키를 구성하는 값을 여러 개의 부분집합에 분배하여 정렬하는 방식(기수 정렬)

✓ 외부 정렬(external sort)

- 정렬할 자료를 보조 기억장치에서 정렬하는 방식
- 대용량의 보조 기억 장치를 사용하기 때문에 내부 정렬보다 속도는 떨어지지만 내부 정렬로 처리할 수 없는 대용량의 자료에 대한 정렬 가능
- 병합 방식 : 파일을 부분 파일로 분리하여 각각을 내부 정렬 방법으로 정렬하여 병합하는 정렬 방식 (2-way 병합, n-way 병합)

Selection Sort

❖ Selection Sort

✓ 선택 정렬은 첫 번째 자리부터 마지막에서 두 번째 자리까지의 데이터를 기준으로 기준 뒤에 있는 모든 데이터들과 비교해서 기준 자리의 데이터가 크면 2개 요소의 자리를 변경

초기 상태 50 40 10 20 30

1Pass 10 50 40 20 30
첫번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환, 40이 10보다 작아서 교환

2Pass 10 20 50 40 30
두번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환, 40이 20보다 작아서 교환

3Pass 10 20 30 50 40
세번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환, 40이 30보다 작아서 교환

4Pass 10 20 30 40 50
네번째 자리를 기준으로 해서 자신의 뒤의 모든 자리와 비교해서 교환
50이 40보다 작아서 교환

Selection Sort

- Selection Sort
  - ✓ 메모리 사용 공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - ✓ 비교 횟수
    - 1단계 : 첫 번째 원소를 기준으로 n개의 원소 비교
    - 2단계 : 두 번째 원소를 기준으로 마지막 원소까지 n-1개의 원소 비교
    - 3단계 : 세 번째 원소를 기준으로 마지막 원소까지 n-2개의 원소 비교
    - i 단계 : i 번째 원소를 기준으로 n-i개의 원소 비교
  - ✓ 어떤 경우에서나 비교 횟수가 같으므로 시간 복잡도는 $O(n^2)$이 됨

전체 비교횟수 : $(n-1)+(n-2)+\ \dots\ +2+1 = \sum_{i=1}^{n-1} n-i = \frac{n(n-1)}{2}$

Selection Sort

```
public class SelectionSort {
    public static void main(String[] args) {
            int test[] = { 20, 30, 40, 50, 10 };
            int i, j, temp;
            int cnt = test.length;
            System.out.print("정렬 전:");
            for (i = 0; i < cnt; i++) {
                    System.out.print(test[i] + " ");
            }
            System.out.println();
```

Selection Sort

```
for (i = 0; i < cnt-1; i++) {
        for (j = i + 1; j < cnt; j++) {
                if (test[i] > test[j]) {
                        temp = test[i];
                        test[i] = test[j];
                        test[j] = temp;
                }
        }

        System.out.print((i+1) + "pass:");
        for (int k = 0; k < cnt; k++) {
                System.out.print(test[k] + " ");
        }
        System.out.println();
}
```

Selection Sort

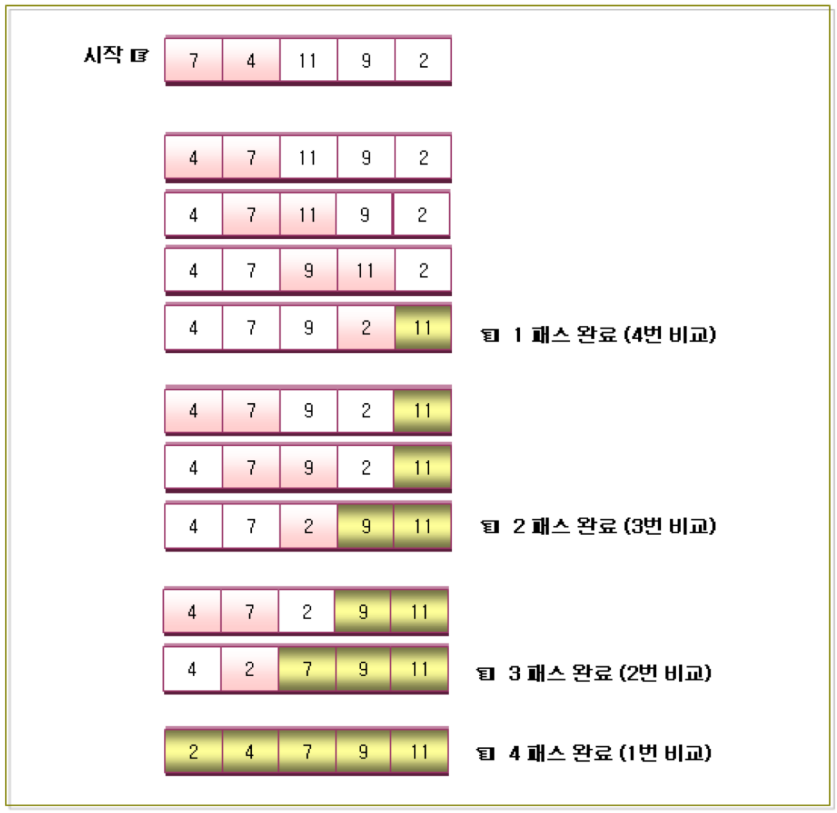
```
                System.out.println("======================");
                System.out.println("======================");
                System.out.println("정렬 후");
                for (i = 0; i < cnt; i++) {
                        System.out.print(test[i] + " ");
                }
        }
}
```

Selection Sort

정렬 전:20 30 40 50 10
1pass:10 30 40 50 20
2pass:10 20 40 50 30
3pass:10 20 30 50 40
4pass:10 20 30 40 50
==========
==========
정렬 후
10 20 30 40 50

Bubble Sort

- Bubble Sort
  - ✓ 버블 정렬은 n개의 데이터가 있을 때 1부터 n-1 번째 자료까지 n-1번 동안 다음 자료와 비교해가면서 정렬하는 방법
  - ✓ 버블 정렬의 효과를 높이기 위해서는 비교 시 횟수만큼 빼면서 정렬하면 성능을 높일 수 있고 flag처리 등을 이용해서 자리 바꿈이 일어나지 않을 때 멈추게 하면 성능을 더욱 향상 시킬 수 있음



Bubble Sort

- Bubble Sort
  - ✓ 메모리 사용 공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - ✓ 연산 시간
    - 최선의 경우: 자료가 이미 정렬되어있는 경우
      - 비교 횟수 : i번째 원소를 (n-i)번 비교하므로, $(n(n-1))/2$ 번
      - 자리 교환 횟수 : 자리 교환이 발생하지 않음
    - 최악의 경우 : 자료가 역순으로 정렬되어있는 경우
      - 비교 횟수 : i번째 원소를 (n-i)번 비교하므로, $(n(n-1))/2$ 번
      - 자리 교환 횟수 : i번째 원소를 (n-i)번 교환하므로, $(n(n-1))/2$ 번
    - 최선의 경우와 최악의 경우에 대한 평균 시간 복잡도를 빅-오 Big Oh 표기법으로 나타내면 $O(n^2)$이 됨

Bubble Sort

```
public class BubbleSort {
    public static void main(String[] args) {
            int test[] = { 20, 30, 40, 50, 10 };
            int i, j, temp, flag;
            int cnt = test.length;
            System.out.println("정렬 전");
            for (i = 0; i < cnt; i++) {
                    System.out.print(test[i] + " ");
            }
            System.out.println();
```

Bubble Sort

```
for (i = 0; i < cnt-1; i++) {
        flag = 0;
        for (j = 0; j < cnt - (i+1); j++) {
                if (test[j] > test[j + 1]) {
                        temp = test[j];
                        test[j] = test[j + 1];
                        test[j + 1] = temp;
                        flag = 1;
                }
        }
        System.out.print((i+1) + "pass:");
        for (int k = 0; k < cnt; k++) {
                System.out.print(test[k] + " ");
        }
        System.out.println();
        if (flag == 0) {
                break;
        }
}
```

Bubble Sort

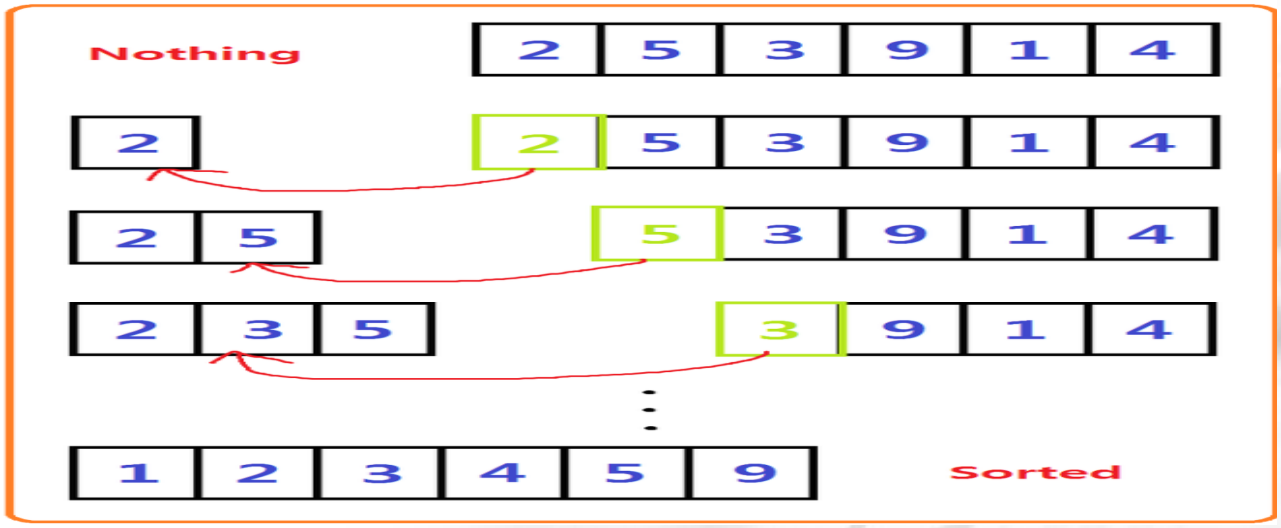
```
            System.out.println("======================");
            System.out.println("======================");
            System.out.println("정렬 후");
            for (int k = 0; k < cnt; k++) {
                        System.out.print(test[k] + " ");
            }
            System.out.println();
    }
}
```

Bubble Sort

정렬 전
20 30 40 50 10
1pass:20 30 40 10 50
2pass:20 30 10 40 50
3pass:20 10 30 40 50
4pass:10 20 30 40 50
==========
==========
정렬 후
10 20 30 40 50

Insertion Sort

- Insertion Sort
  - ✓ 정렬되어 있는 부분 집합에 정렬할 새로운 원소의 위치를 찾아 삽입하는 방법
    - 정렬할 자료를 두 개의 부분 집합 S(Sorted Subset)와 U(Unsorted Subset)로 가정
    - 부분 집합 S: 정렬된 앞부분의 원소들
    - 부분 집합 U: 아직 정렬되지 않은 나머지 원소들
    - 정렬되지 않은 부분 집합 U의 원소를 하나씩 꺼내서 이미 정렬되어 있는 부분 집합 S의 마지막 원소부터 비교하면서 위치를 찾아 삽입
    - 삽입 정렬을 반복하면서 부분 집합 S의 원소는 하나씩 늘리고 부분 집합 U의 원소는 하나씩 감소하게 함 - 부분 집합 U가 공집합이 되면 삽입 정렬이 완성



Insertion Sort

- Insertion Sort
  - ✓ 메모리 사용 공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - ✓ 연산 시간
    - 최선의 경우: 원소들이 이미 정렬되어 있어서 비교 횟수가 최소인 경우
      - 이미 정렬되어 있는 경우에는 바로 앞자리 원소와 한번만 비교
      - 전체 비교 횟수 = n-1
      - 시간 복잡도 : O(n)
    - 최악의 경우: 모든 원소가 역순으로 되어있어서 비교 횟수가 최대인 경우
      - 전체 비교 횟수 = 1+2+3+ … +(n-1) = n(n-1)/2
      - 시간 복잡도 : $O(n^2)$
    - 삽입 정렬의 평균 비교 횟수 = n(n-1)/4
    - 평균 시간 복잡도: $O(n^2)$

Insertion Sort

```
public class InsertionSort {
    public static void main(String[] args) {
            int[] list = { 5, 3, 8, 4, 9, 1, 6, 2, 7 };
            System.out.println("정렬 전");
            for(int e : list) {
                    System.out.print(e + "   ");
            }
            System.out.println();
```

Insertion Sort

```
System.out.println("정렬 수행");
for (int i = 1; i < list.length; i=i+1) {
        int standard = list[i];  // 기준
        int j = i-1;
        for( ;j>=0; j=j-1) {
                if(standard >= list[j]) {
                        break;
                }
                list[j+1] = list[j];
        }
        list[j + 1] = standard;  // 기준값 저장
        System.out.print((i)+"pass:");
        for(int k=0; k<i+1; k++) {
                System.out.print(list[k] + "   ");
        }
        System.out.println();
}
```

Insertion Sort

```
                System.out.println("정렬 후");
                for(int e : list) {
                        System.out.print(e + "   ");
                }
        }
}
```

Insertion Sort

정렬 전
5 3 8 4 9 1 6 2 7

정렬 수행
1pass:3 5
2pass:3 5 8
3pass:3 4 5 8
4pass:3 4 5 8 9
5pass:1 3 4 5 8 9
6pass:1 3 4 5 6 8 9
7pass:1 2 3 4 5 6 8 9
8pass:1 2 3 4 5 6 7 8 9

정렬 후
1 2 3 4 5 6 7 8 9

Quick Sort

❖ Quick Sort

❶ 왼쪽 끝에서 오른쪽으로 움직이면서 크기를 비교하여 피봇보다 크거나 같은 원소를 찾아 L로 표시한다. 단, L은 R과 만나면 더 이상 오른쪽으로 이동하지 못하고 멈춘다.

❷ 오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾아 R로 표시한다. 단, R은 L과 만나면 더 이상 왼쪽으로 이동하지 못하고 멈춘다.

❸-ⓐ ❶과 ❷에서 찾은 L 원소와 R 원소가 있는 경우, 서로 교환하고 L과 R의 현재 위치에서 ❶과 ❷ 작업을 다시 수행한다.

❸-ⓑ ❶~❷를 수행하면서 L과 R이 같은 원소에서 만나 멈춘 경우, 피봇과 R의 원소를 서로 교환한다. 교환된 자리를 피봇 위치로 확정하고 현재 단계의 퀵 정렬을 끝낸다.

❹ 피봇의 확정된 위치를 기준으로 만들어진 새로운 왼쪽 부분집합과 오른쪽 부분집합에 대해서 ❶~❸의 퀵 정렬을 순환적으로 반복 수행하는데, 모든 부분집합의 크기가 1 이하가 되면 전체 퀵 정렬을 종료한다.

Quick Sort

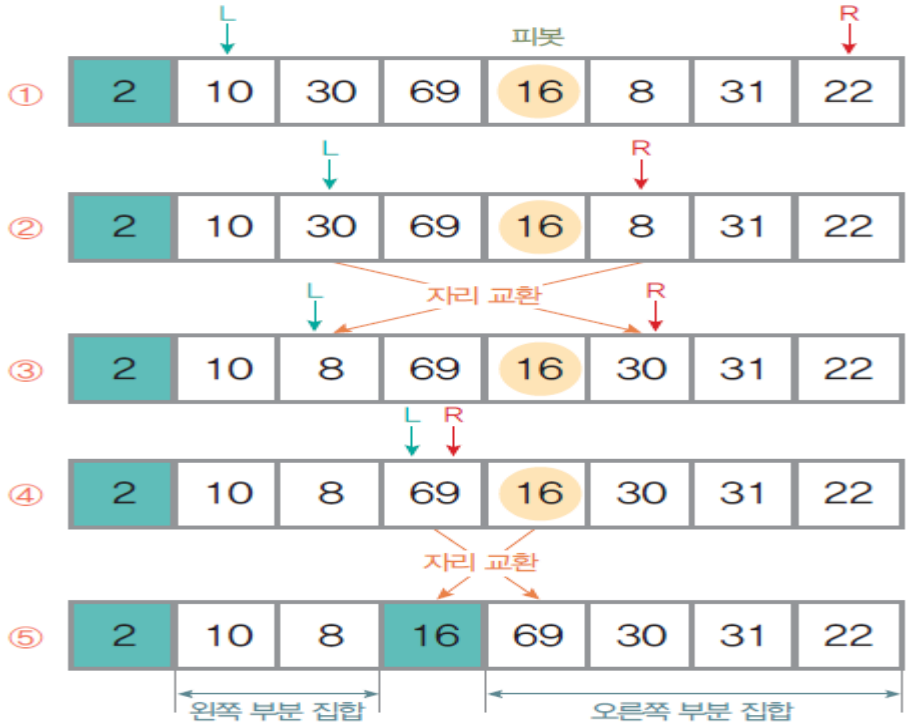
- ❖ Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - ● 1단계
      - ◆ 원소 2를 피봇으로 선택하고 퀵 정렬을 시작
      - ◆ L은 정렬 범위의 왼쪽 끝에서 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를 찾고, R은 정렬 범위의 오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음. L은 원소 69를 찾았지만, R은 피봇보다 작은 원소를 찾지 못한 상태로 원소 69에서 L과 만남. L과 R이 만나 더 이상 진행할 수 없는 상태가 되었으므로
      - ◆ 원소 69를 피봇과 자리를 교환하고 피봇 원소 2의 위치를 확정

Quick Sort

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 2단계: 위치가 확정된 피봇 2의 왼쪽 부분 집합은 공집합이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합에 대해서 퀵 정렬을 수행
      - 오른쪽 부분 집합의 원소가 일곱 개이므로 가운데 있는 원소 16을 피봇으로 선택하고 퀵 정렬을 시작.
      - L은 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소인 30을 찾고 R은 왼쪽으로 움직이면서 피봇보다 작은 원소인 8을 찾음.
      - L이 찾은 30과 R이 찾은 8을 서로 자리를 교환한 후 현재 위치에서 L은 다시 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소69를 찾고 R은 피봇보다 작은 원소를 찾음
      - R이 원소 69에서 L과 만나 더 이상 진행할 수 없는 상태가 되었으므로,
      - 원소 69를 피봇과 교환하고 피봇 원소 16의 위치를 확정

Quick Sort

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 2단계: 위치가 확정된 피봇 2의 왼쪽 부분 집합은 공 집합이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합에 대해서 퀵 정렬을 수행

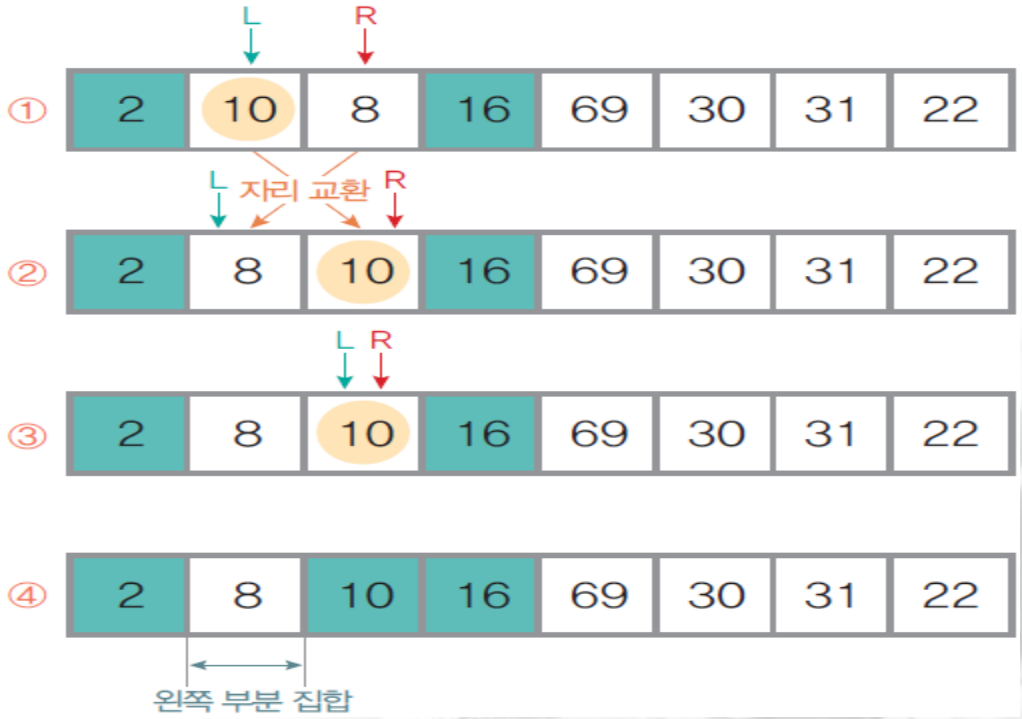


Quick Sort

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 3단계: 위치가 확정된 피봇 16을 기준으로 새로 생긴 왼쪽 부분 집합에서 퀵 정렬을 수행
      - 원소 10을 피봇으로 선택하고 L은 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를 R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
      - L이 찾은 10과 R이 찾은 8을 서로 교환
      - 현재 위치에서 L은 다시 오른쪽으로 움직이다가 원소 10에서 R과 만나서 멈춤 - 더 이상 진행할 수 없는 상태가 되었으므로
      - R의 원소와 피봇을 교환하고 피봇 원소 10의 위치를 확정한다(이 경우에는 R과 피봇 위치가 같았으므로 자리를 교환하기 전과 후의 상태가 같음)

Quick Sort

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 3단계: 위치가 확정된 피봇 16을 기준으로 새로 생긴 왼쪽 부분 집합에서 퀵 정렬을 수행



Quick Sort

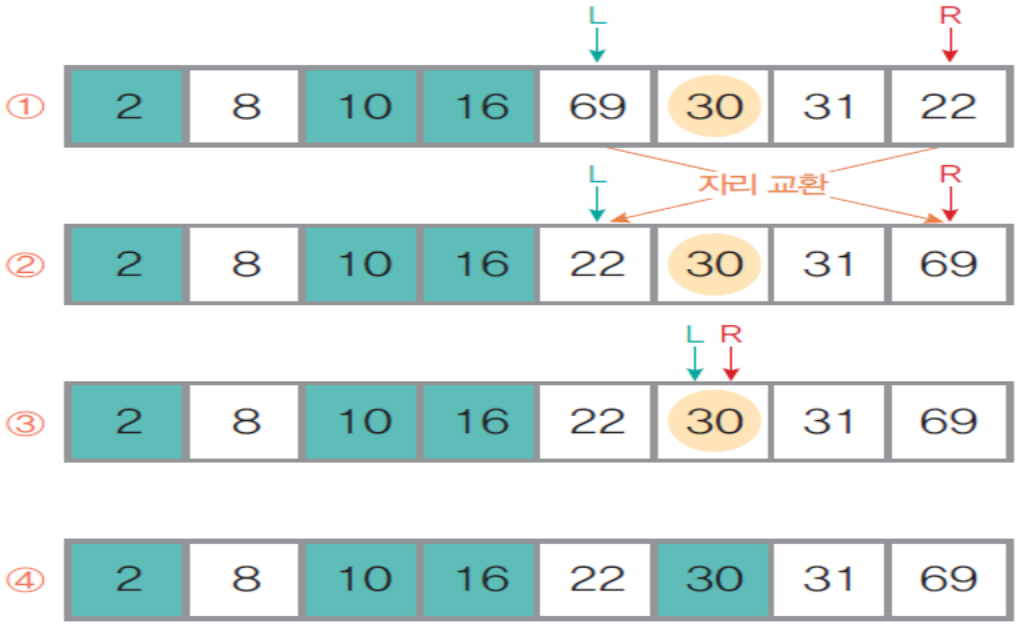
- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 4단계: 피봇 10의 왼쪽 부분 집합은 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합은 공집합이므로 역시 퀵 정렬을 수행하지 않고 [2]에서 피봇이었던 원소 16의 오른쪽 부분 집합에 대해서 퀵 정렬을 수행
      - 오른쪽 부분 집합의 원소가 네 개이므로 가운데 있는 원소 30을 피봇으로 선택. L은 오른쪽으로 이동하면서 피봇보다 크거나 같은 원소를 찾고 R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
      - L이 찾은 69와 R이 찾은 22를 서로 교환. 현재 위치에서 다시 L은 피봇보다 크거나 같은 원소를 찾아 오른쪽으로 움직이고 R은 피봇보다 작은 원소를 찾아 왼쪽으로 움직이다가
      - 원소 30에서 L과 R이 만나 멈춤. 더 이상 진행할 수 없음
      - R의 원소와 피봇을 교환하고 피봇 원소30의 위치가 확정 (이 경우에 R과 피봇 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음)

Quick Sort

❖ Quick Sort

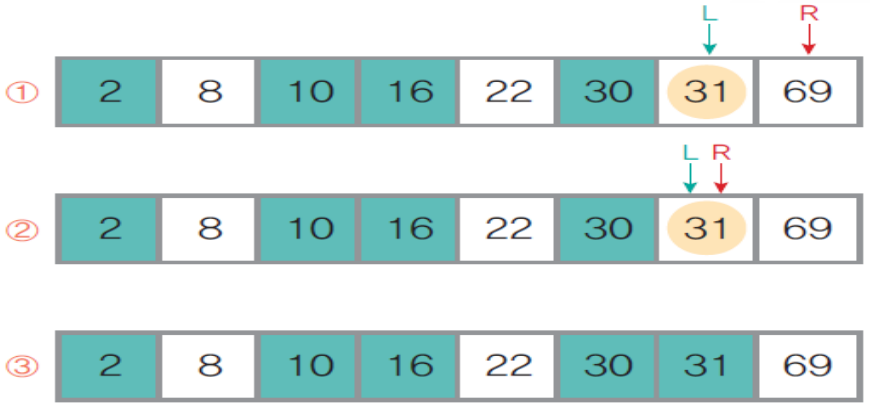
✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정

- 4단계: 피봇 10의 왼쪽 부분 집합은 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합은 공집합이므로 역시 퀵 정렬을 수행하지 않고 [2]에서 피봇이었던 원소 16의 오른쪽 부분 집합에 대해서 퀵 정렬을 수행



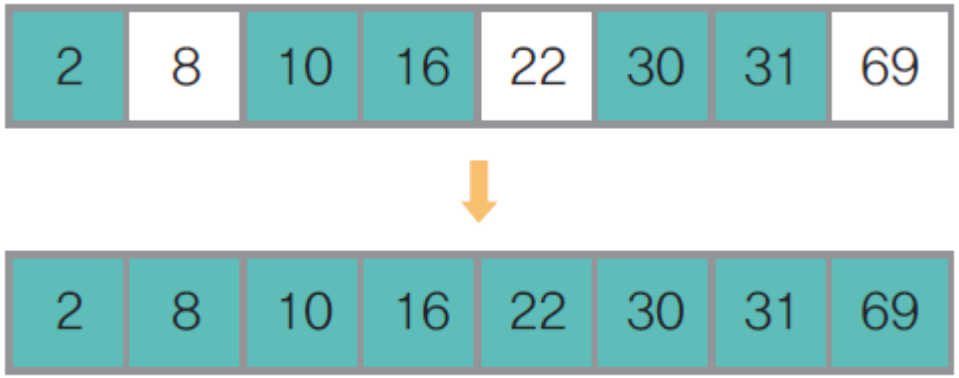
Quick Sort

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 5단계: 피봇 30의 왼쪽 부분 집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합에 대해서 퀵 정렬을 수행
      - 피봇은 31을 선택하고 L은 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를 찾고 R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
      - L과 R이 원소 31에서 만나 더 이상 진행하지 못하는 상태가 되어
      - 원소 31을 피봇과 교환하여 위치를 확정(이 경우에는 R과 피봇 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음)



Quick Sort

- Quick Sort
  - ✓ 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정
    - 6단계: 피봇 31의 오른쪽 부분 집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않지 않고 모든 부분 집합의 크기가 1 이하이므로 전체 퀵 정렬을 종료



Quick Sort

- Quick Sort
  - ✓ 메모리 사용 공간
    - n개의 원소에 대하여 n개의 메모리 사용
  - ✓ 연산 시간
    - 최선의 경우
      - 피봇에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 n/2개씩 2등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우
    - 최악의 경우
      - 피봇에 의해 원소들을 분할하였을 때 한개와 n-1개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우
    - 평균 시간 복잡도: $O(n \log_2 n)$
    - 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법

Quick Sort

```
public class QuickSort {

    public static void quickSort(int left, int right, int[] data) {
            // 정렬 결과 출력
            for (int temp : data) {
                    System.out.print(temp + "₩t");
            }
            System.out.println();

            // 가장 왼쪽의 데이터를 pivot으로 설정
            int pivot = left;
            // 피봇의 위치를 j에 대입
            int j = pivot;
            // 피봇과 비교할 데이터의 위치의 초기값을 설정
            int i = left + 1;
```

Quick Sort

```
        if (left < right) {
                for (; i <= right; i=i+1) {
                        if (data[i] < data[pivot]) {
                                j = j + 1;
                                int temp = data[j];
                                data[j] = data[i];
                                data[i] = temp;
                        }
                }

                int temp = data[left];
                data[left] = data[j];
                data[j] = temp;

                pivot = j;

                quickSort(left, pivot - 1, data);
                quickSort(pivot + 1, right, data);
        }

}
```

Quick Sort

```
    public static void main(String[] args) {

            int[] list = { 69, 10, 30, 2, 14, 8, 31, 22, 16 };
            int n = list.length;
            // 퀵 정렬 수행(left: 배열의 시작 = 0, right: 배열의 끝 = 8)
            quickSort(0, n - 1, list);

            // 정렬 결과 출력
            System.out.println("정렬 후");
            for (int temp : list) {
                    System.out.print(temp + "₩t");
            }
            System.out.println();
    }

}
```

Quick Sort

69	10	30	2	14	8	31	22	16
16	10	30	2	14	8	31	22	69
8	10	2	14	16	30	31	22	69
2	8	10	14	16	30	31	22	69
2	8	10	14	16	30	31	22	69
2	8	10	14	16	30	31	22	69
2	8	10	14	16	30	31	22	69
2	8	10	14	16	30	31	22	69
2	8	10	14	16	22	30	31	69
2	8	10	14	16	22	30	31	69
2	8	10	14	16	22	30	31	69

정렬 후

2	8	10	14	16	22	30	31	69

Quick Sort

```
// 퀵 정렬(비재귀 버전)
import java.util.Scanner;
class QuickSort2 {

    //--- 배열 요소 a[idx1]와 a[idx2]의 값을 교환 ---//
    static void swap(int[] a, int idx1, int idx2) {
        int t = a[idx1];  a[idx1] = a[idx2];  a[idx2] = t;
    }
```

Quick Sort

```
//--- 퀵 정렬(비재귀 버전)---//
static void quickSort(int[] a, int left, int right) {
    IntStack lstack = new IntStack(right - left + 1);
    IntStack rstack = new IntStack(right - left + 1);

    lstack.push(left);
    rstack.push(right);

    while (lstack.isEmpty() != true) {
        int pl = left  = lstack.pop();        // 왼쪽 커서
        int pr = right = rstack.pop();        // 오른쪽 커서
        int x = a[(left + right) / 2];        // 피벗은 가운데 요소
        do {
            while (a[pl] < x) pl++;
            while (a[pr] > x) pr--;
            if (pl <= pr)
                swap(a, pl++, pr--);
        } while (pl <= pr);
```

Quick Sort

```
            if (left < pr) {
                lstack.push(left);          // 왼쪽 그룹 범위의
                rstack.push(pr);            // 인덱스를 푸시
            }

            if (pl < right) {
                lstack.push(pl);            // 오른쪽 그룹 범위의
                rstack.push(right);         // 인덱스를 푸시
            }
        }
    }
```

Quick Sort

```
    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);
        System.out.println("퀵 정렬");
        System.out.print("요솟수: ");
        int nx = stdIn.nextInt();
        int[] x = new int[nx];

        for (int i = 0; i < nx; i++) {
            System.out.print("x[" + i + "]: ");
            x[i] = stdIn.nextInt();
        }

        quickSort(x, 0, nx - 1);            // 배열 x를 퀵정렬
        System.out.println("오름차순으로 정렬했습니다.");
        for (int i = 0; i < nx; i++)
            System.out.println("x[" + i + "]=" + x[i]);
    }
}
```

Shell Sort

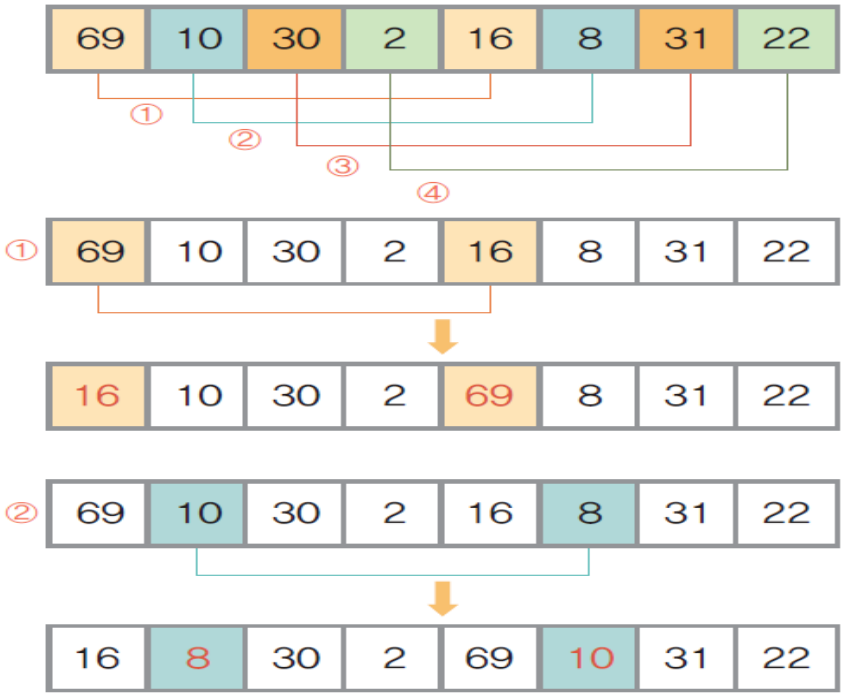
- Shell Sort
  - ✓ 일정한 간격(interval)으로 떨어져있는 자료들끼리 부분 집합을 구성하고 각 부분 집합에 있는 원소들에 대해서 삽입 정렬을 수행하는 작업을 반복하면서 전체 원소들을 정렬하는 방법
  - ✓ 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분 집합으로 나누어 정렬하게 되면 비교 연산과 교환 연산 감소
  - ✓ 셸 정렬의 부분 집합
    - 부분 집합의 기준이 되는 간격을 매개변수 h에 저장
    - 한 단계가 수행될 때마다 h의 값을 감소시키고 셸 정렬을 순환 호출
    - h가 1이 될 때까지 반복
  - ✓ 셸 정렬의 성능은 매개변수 h의 값에 따라 달라짐
    - 정렬할 자료의 특성에 따라 매개변수 생성 함수를 사용
    - 일반적으로 사용하는 h의 값은 원소 개수의 1/2을 사용하고 한 단계 수행될 때마다 h의 값을 반으로 감소시키면서 반복 수행

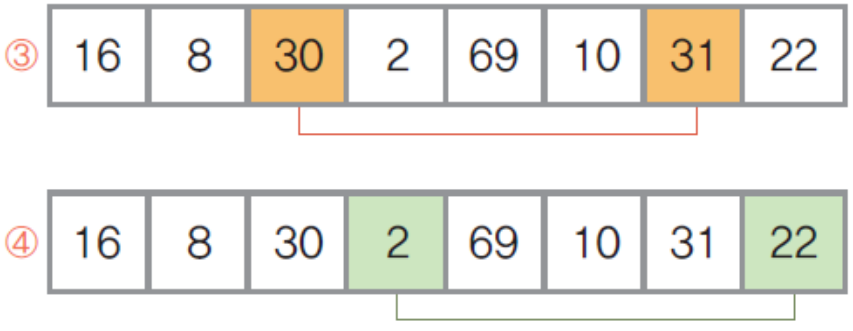
Shell Sort

- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - 원소 개수가 여덟 개이므로 매개변수 h는 4에서 시작한다. h=4이므로 간격이 4만큼 떨어져 있는 원소들을 같은 부분 집합으로 만들면 네 개의 부분 집합이 만들어짐(같은 부분 집합은 동일한 색으로 표시)
      - 첫 번째 부분 집합 {69, 16}에 대해서 삽입 정렬을 수행하여 정렬
      - 두 번째 부분 집합{10, 8}에 대해서 삽입 정렬을 수행
      - 세 번째 부분 집합 {30, 31}에 대해서 삽입 정렬을 수행. 30<31이므로 자리 이동은 이루어지지 않음.
      - 네 번째 부분 집합 {2, 22}에 대해서 삽입 정렬을 수행. 2<22이므로 자리 이동은 이루어지지 않음

Shell Sort

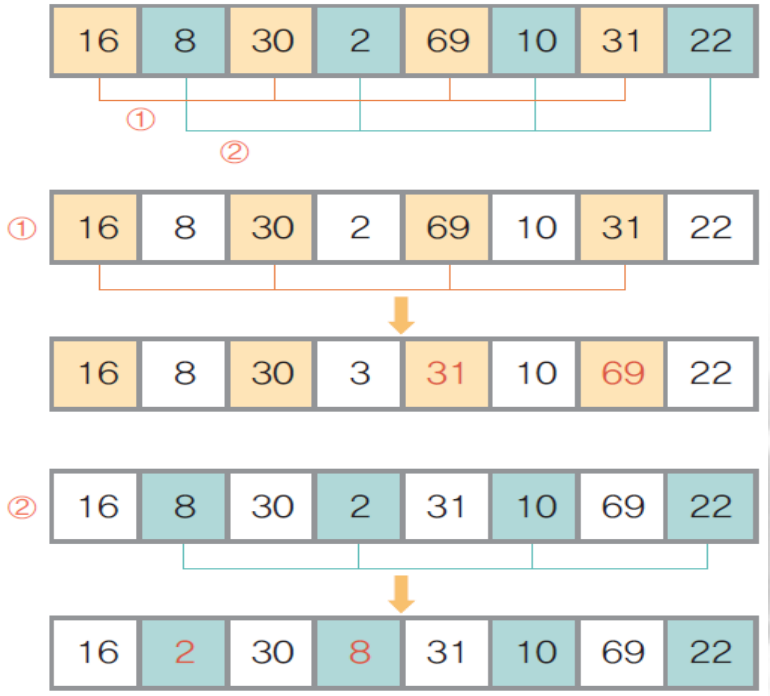
- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - 원소 개수가 여덟 개이므로 매개변수 h는 4에서 시작한다. h=4이므로 간격이 4만큼 떨어져 있는 원소들을 같은 부분 집합으로 만들면 네 개의 부분 집합이 만들어짐(같은 부분 집합은 동일한 색으로 표시)





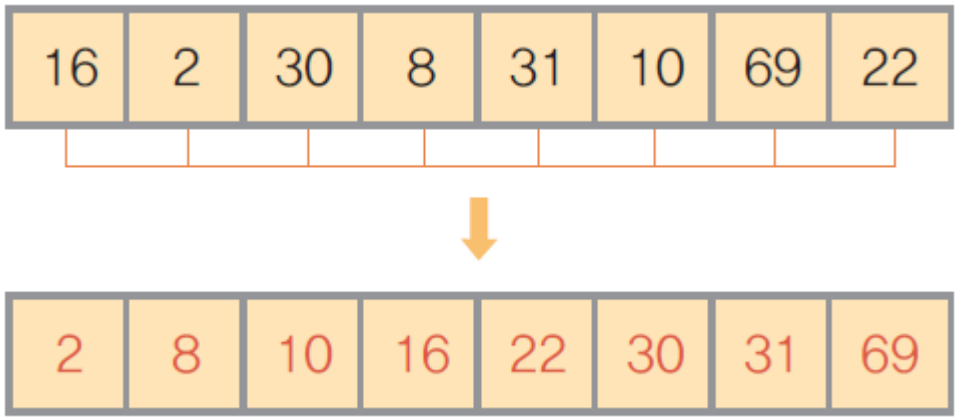
Shell Sort

- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - 이제 h를 2로 변경하고 다시 셸 정렬을 시작. h=2이므로 간격이 2만큼 떨어진 원소들을 같은 부분 집합으로 만들면 두 개의 부분 집합이 만들어짐
      - ◆ 첫 번째 부분 집합 {16, 30, 69, 31}에 대해 삽입 정렬을 수행하여 정렬
      - ◆ 두 번째 부분 집합 {8, 2, 10, 22}에 대해 삽입 정렬을 수행하여 정렬



Shell Sort

- Shell Sort
  - ✓ 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정
    - 이제 h를 1로 변경하고 다시 셸 정렬을 시작. h=1이므로 간격이 1만큼 떨어져 있는 원소들을 같은 부분 집합으로 만들면 한 개의 부분 집합이 만들어짐 - 전체 원소에 대해서 삽입 정렬을 수행



Shell Sort

- Shell Sort
  - ✓ 메모리 사용 공간
    - n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장 공간 사용
  - ✓ 연산 시간
    - 비교 횟수
      - 처음 원소의 상태에 상관없이 매개변수 h에 의해 결정
    - 일반적인 시간 복잡도: $O(n^{1.25})$
    - 셸 정렬은 삽입 정렬의 시간 복잡도 $O(n^2)$ 보다 개선된 정렬 방법

Shell Sort

```
public class ShellSort {
    public static void intervalSort(int a[], int begin, int end, int interval){
            int i, j, item;
            for(i=begin+interval; i<=end; i=i+interval){
                    item = a[i];
                    for(j=i-interval; j>=begin && item<a[j]; j-=interval)
                            a[j+interval] = a[j];
                    a[j+interval] = item;
            }
    }
```

Shell Sort

```
public static void shellSort(int a[], int size){
  int i, j, interval, t=0;
  interval = size/2;
  while(interval >= 1){
        for(i=0; i<interval; i++)
              intervalSort(a, i, size-1, interval);
        System.out.printf("₩nShell Sort %d 단계 : %d >> ", ++t, interval);
        for(j=0; j<size; j++)
              System.out.printf("%d    ", a[j]);
        System.out.println();
        interval /= 2;
  }
}
```

Shell Sort

```
    public static void main(String args[]){
            int a[] = {69, 10, 30, 2, 16, 8, 31, 22};
            int size = a.length;
            System.out.printf("₩n정렬할 원소 : ");
            for(int i=0; i<a.length; i++)
                    System.out.printf(" %d", a[i]);
            System.out.println();
            shellSort(a, size);
    }

}
```

Shell Sort

정렬할 원소 : 69 10 30 2 16 8 31 22

Shell Sort 1 단계 : 4 >> 16 8 30 2 69 10 31 22

Shell Sort 2 단계 : 2 >> 16 2 30 8 31 10 69 22

Shell Sort 3 단계 : 1 >> 2 8 10 16 22 30 31 69

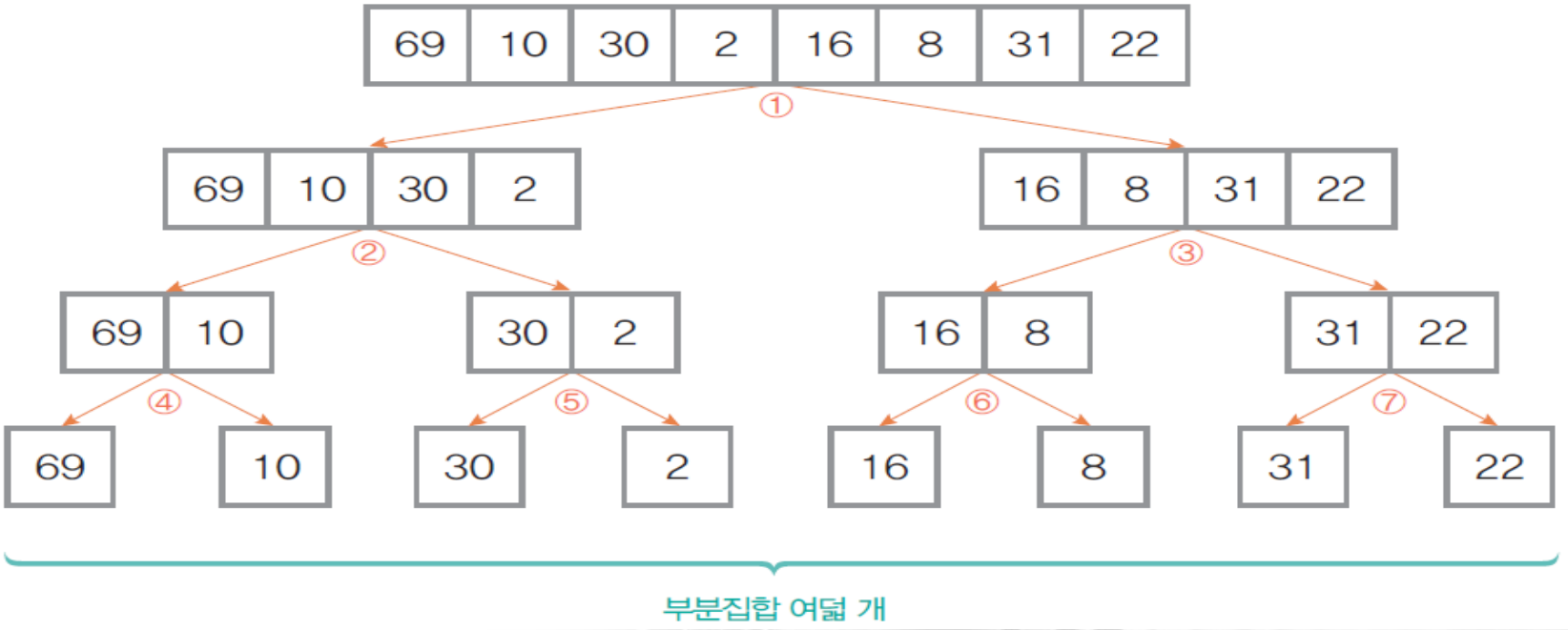
Merge Sort

- Merge Sort
  - ✓ 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법
  - ✓ 부분 집합으로 분할(divide)하고, 각 부분 집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분 집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법 사용
  - ✓ 병합 정렬 방법의 종류
    - 2-way 병합: 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
    - n-way 병합: n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
  - ✓ 2-way 병합 정렬

- 분할Divide : 자료들을 두 개의 부분집합으로 분할한다.
- 정복Conquer : 부분집합에 있는 원소를 정렬한다.
- 결합Combine : 정렬된 부분집합들을 하나의 집합으로 정렬하여 결합한다.

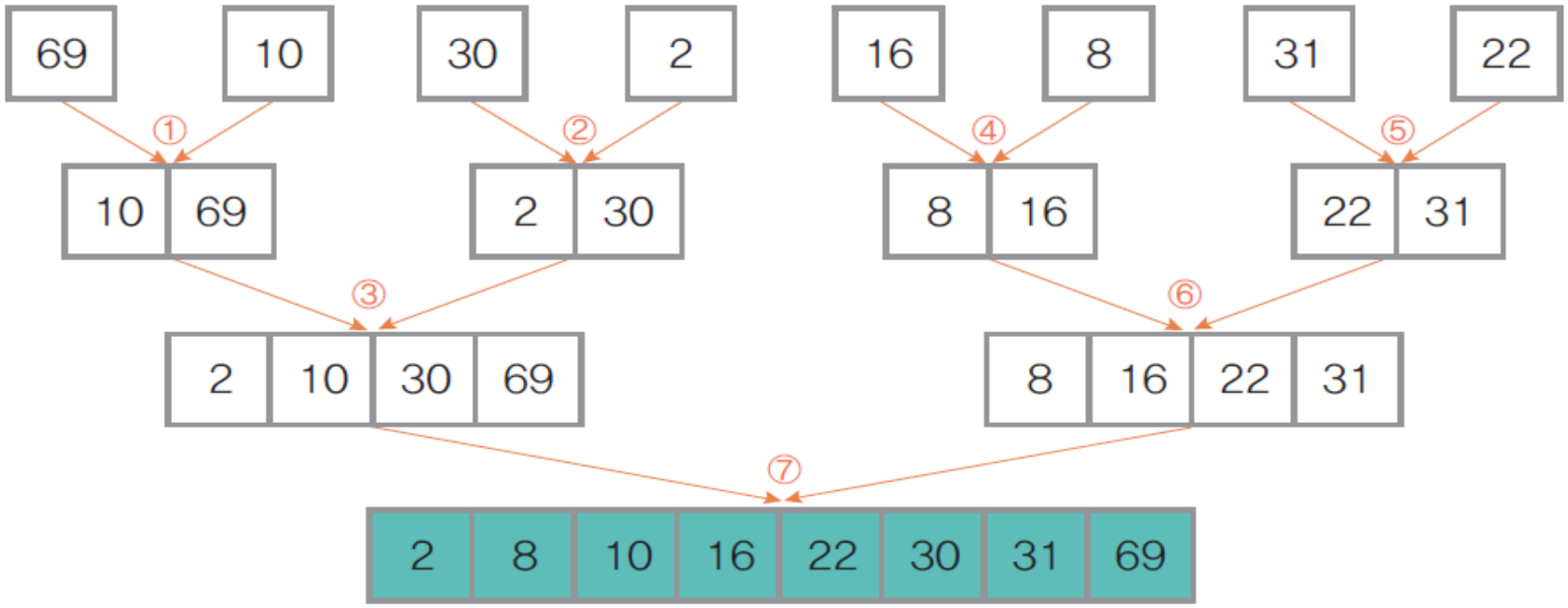
Merge Sort

- Merge Sort
  - ✓ 2-way 병합 정렬
    - 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 병합 정렬 방법으로 정렬하는 과정
      - ◆ 분할 단계 : 정렬할 전체 자료의 집합에 대해서 최소 원소의 부분 집합이 될 때까지 분할 작업을 반복하여 한 개의 원소를 가진 부분 집합 8개 만듦



Merge Sort

- Merge Sort
  - ✓ 2-way 병합 정렬
    - 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 병합 정렬 방법으로 정렬하는 과정
      - 정복과 결합 단계: 부분 집합 두 개를 정렬하여 하나로 결합. 전체 원소가 집합 하나로 묶일 때까지 반복



Merge Sort

- Merge Sort
  - ✓ 메모리 사용 공간
    - 각 단계에서 새로 병합하여 만든 부분 집합을 저장할 공간이 추가로 필요
    - 원소 n개에 대해서 (2 x n)개의 메모리 공간 사용
  - ✓ 연산 시간
    - 분할 단계: n개의 원소를 두개로 분할하기 위해서 $\log_2 n$번의 단계 수행
    - 병합 단계: 부분 집합의 원소를 비교하면서 병합하는 단계에서 최대 n번의 비교 연산 수행
    - 전체 병합 정렬의 시간 복잡도 : $O(n \log_2 n)$

Merge Sort

```
class Sort{
    private static int sorted[] = new int [30];

    public static void merge(int a[], int m, int middle, int n){
            int size = a.length;
            int  i, j, k, t;
            i = m;
            j = middle+1;
            k = m;
            while(i<=middle && j<=n){
                    if(a[i] <= a[j])  sorted[k] = a[i++];
                    else  sorted[k] = a[j++];
                    k++;
            }
            if(i>middle){
                    for(t=j; t<=n; t++, k++)
                            sorted[k] = a[t];
            }
            else{
                    for(t=i; t<=middle; t++, k++)
                            sorted[k] = a[t];
            }
```

Merge Sort

```
        for(t=m; t<=n; t++)
                a[t] = sorted[t];
        System.out.printf("₩n 병합 정렬 >> ");
        for(t=0; t<size; t++)
                System.out.printf("%3d ", a[t]);
}
```

Merge Sort

```
public static void mergeSort(int a[], int m, int n)      {
        int middle;
        if(m<n){
                middle = (m+n)/2;
                mergeSort(a, m, middle);
                mergeSort(a, middle+1, n);
                merge(a, m, middle, n);
        }
}
```

Merge Sort

```
    public static void main(String args[]){
            int a[] = {69, 10, 30, 2, 16, 8, 31, 22};
            int size = a.length;
            System.out.printf("₩n정렬할 원소 : ");
            for(int i=0; i<a.length; i++)
                    System.out.printf(" %d", a[i]);
            System.out.println();
            mergeSort(a, 0, size-1);
    }
}
```

Merge Sort

정렬할 원소 : 69 10 30 2 16 8 31 22

Shell Sort 1 단계 : 간 = 4 >> 16 8 30 2 69 10 31 22

Shell Sort 2 단계 : 간 = 2 >> 16 2 30 8 31 10 69 22

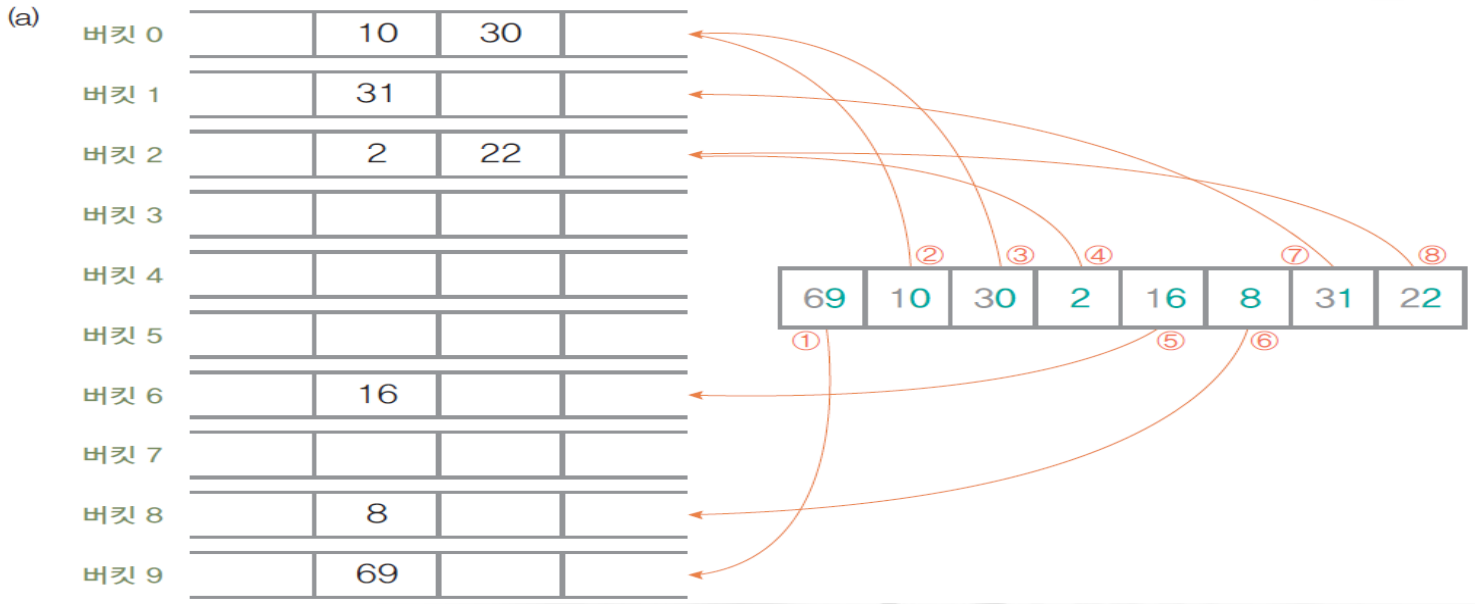
Shell Sort 3 단계 : 간 = 1 >> 2 8 10 16 22 30 31 69

Radix Sort

- Radix Sort
  - ✓ 원소의 키 값을 나타내는 기수를 이용한 정렬 방법
  - ✓ 정렬할 원소의 키 값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소를 꺼내는 방법을 반복하면서 정렬
    - 원소의 키를 표현하는 기수만큼의 버킷 사용
    - 10진수로 표현된 키 값을 가진 원소들을 정렬할 때에는 0 부터 9까지 10개의 버킷 사용
  - ✓ 키 값의 자리 수 만큼 기수 정렬을 반복
    - 키 값의 일의 자리에 대해서 기수 정렬을 수행
    - 다음 단계에서는 키 값의 십의 자리에 대해서
    - 다음 단계에서는 백의 자리에 대해서 기수 정렬 수행
  - ✓ 한 단계가 끝날 때마다 버킷에 분배된 원소들을 버킷의 순서대로 꺼내서 다음 단계의 기수 정렬을 수행해야 하므로 큐를 사용하여 버킷을 만듦

Radix Sort

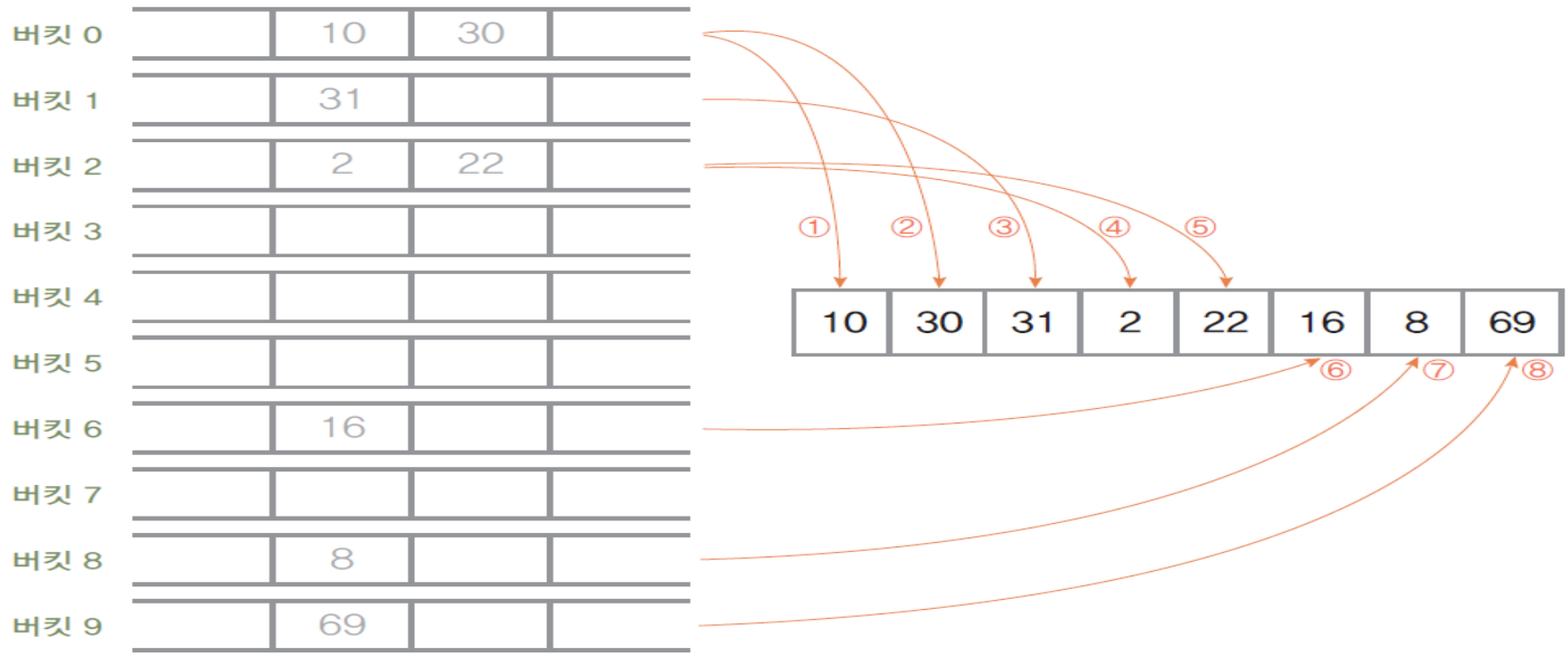
- Radix Sort
  - ✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
    - 키 값이 10진수이므로 0 부터 9까지 열 개의 버킷을 사용하고 키 값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
      - ◆ 키 값의 1의 자리에 대해서 기수 정렬을 수행
        - 정렬할 원소의 키 값의 1의 자리에 맞춰 버킷에 분배
        - 버킷에 분배되어 있는 원소들을 버킷 0 부터 버킷 9까지 순서대로 꺼내고, 꺼낸 순서대로 저장



Radix Sort

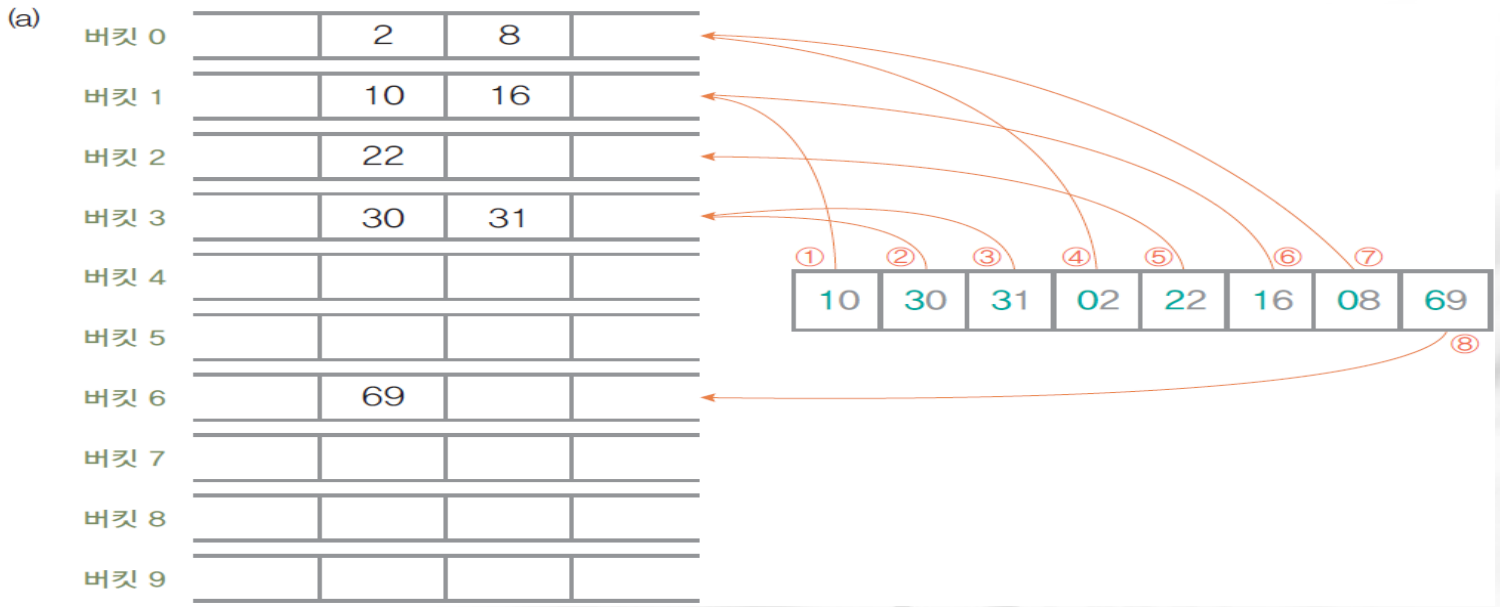
- Radix Sort
  - ✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
    - 키 값이 10진수이므로 0 부터 9까지 열 개의 버킷을 사용하고 키 값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
      - ◆ 키 값의 1의 자리에 대해서 기수 정렬을 수행

(b)



Radix Sort

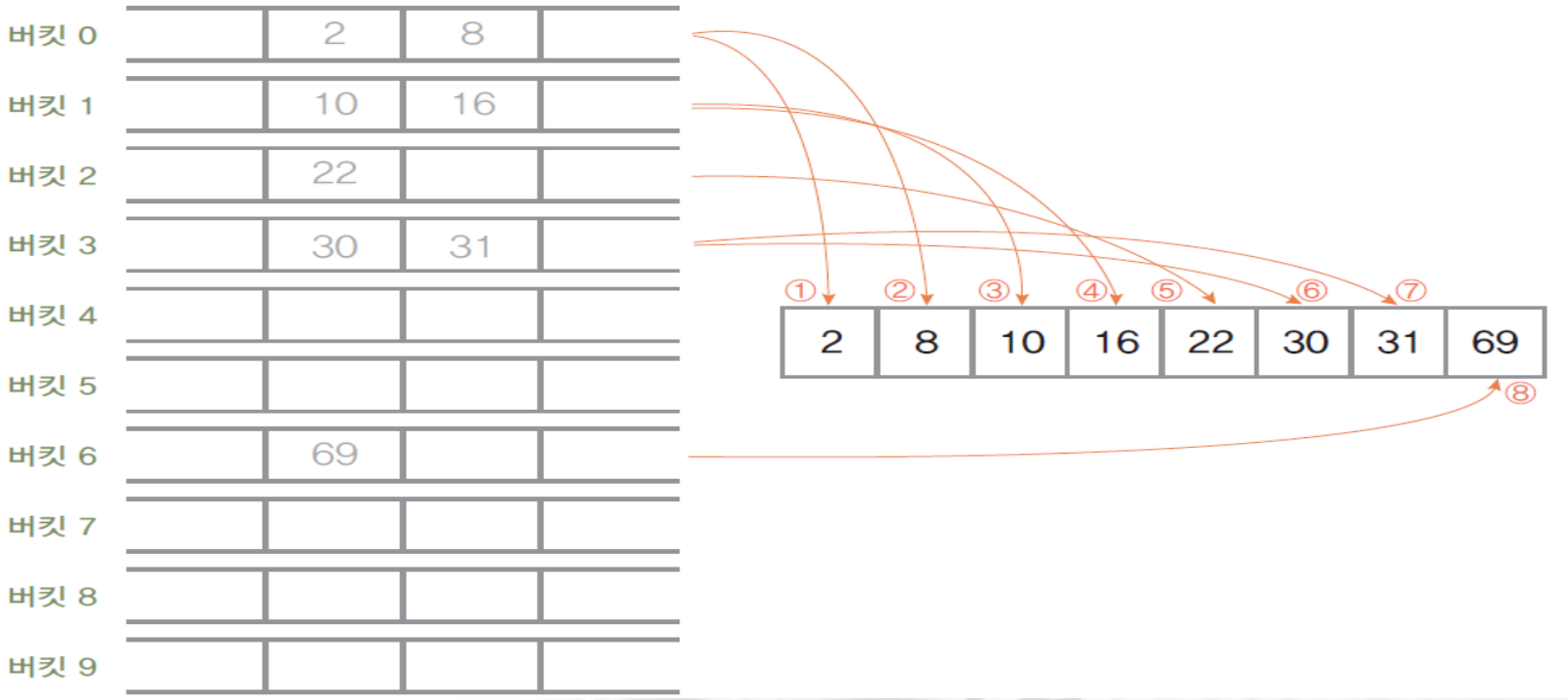
- ❖ Radix Sort
  - ✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
    - ● 키 값이 10진수이므로 0 부터 9까지 열 개의 버킷을 사용하고 키 값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
      - ◆ 키 값의 10의 자리에 대해서 기수 정렬을 수행
        - ▪ 정렬할 원소의 10의 자리에 대해서 버킷에 분배.
        - ▪ 버킷에 분배되어 있는 원소들을 버킷 0부터 버킷 9까지 순서대로 꺼내고, 꺼낸 순서대로 저장하면 전체 원소에 대한 정렬이 완성



Radix Sort

- Radix Sort
  - ✓ {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
    - 키 값이 10진수이므로 0 부터 9까지 열 개의 버킷을 사용하고 키 값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
      - ◆ 키 값의 10의 자리에 대해서 기수 정렬을 수행

(b)



Radix Sort

- Radix Sort
  - ✓ 메모리 사용 공간
    - 원소 n개에 대해서 n개의 메모리 공간 사용
    - 기수 r에 따라 버킷 공간이 추가로 필요
  - ✓ 연산 시간
    - 연산 시간은 정렬할 원소의 수 n과 키 값의 자릿수 d와 버킷의 수를 결정하는 기수 r에 따라서 달라짐
      - 정렬할 원소 n개를 r개의 버킷에 분배하는 작업: (n+r)
      - 이 작업을 자릿수 d 만큼 반복
    - 수행할 전체 작업 : d(n+r)
    - 시간 복잡도 : O(d(n+r))

Radix Sort

```
import java.util.ArrayList;
import java.util.List;

class Linear_Queue {

    int rear = -1;
    int front = 0;
    int maxsize = 0;
    int[] Linear_Queue;

    public Linear_Queue(int maxsize){
        this.maxsize = maxsize;
        Linear_Queue = new int[maxsize];
    }

    public void EnQueue(int num)
    {
        if(rear != maxsize-1){
            Linear_Queue[++rear] = num;
        }else{
            System.out.println("데이터 다참");
        }
    }
```

Radix Sort

```
public int DeQueue()
    {
        if(rear!=front || (rear==0 && front==0)){
            int tmp =Linear_Queue[front];
            for(int i=1;i<=rear;i++)
            {
                Linear_Queue[i-1] = Linear_Queue[i];
            }
            rear--;
            return tmp;
        } else{
            return -1;
        }
    }
}
```

Radix Sort

```
public class RadixSort {
    public static void radix_sort(int[] data)
    {
        int maxsize = getMaxlength(data);
        List<Linear_Queue> bucket = new ArrayList<Linear_Queue>();
        int powed=1;
        int index = 0;

        for(int i=0;i<10;i++){
            bucket.add(new Linear_Queue(10));
        }
```

Radix Sort

```
    for(int i=1;i<=maxsize;i++)        //자리수가 가장 큰 수만큼 전체 반복문을 반복합니다.
    {
        for(int j=0;j<data.length;j++)
        {
            bucket.get((data[j]/powed)%10).EnQueue(data[j]);    //각 자리수의 맞는 index의 bucket
에 넣습니다.
        }
        for(int k=0;k<10;k++)        //버킷에서 데이터를 가지고옵니다.
        {
            int bucket_num = bucket.get(k).rear;

            for(int n=0;n<=bucket_num;n++)
            {
                data[index] = bucket.get(k).DeQueue();
                index++;
            }
        }
        index =0;
        powed *=10;
    }
}
```

Radix Sort

```
public static int getMaxlength(int[] data)
{
    int maxsize = 0;
    for(int i=0;i<data.length;i++)
    {
        int length = (int)Math.log10((double)data[i])+1;
        if(maxsize <length)
        {
            maxsize = length;
        }
    }
    return maxsize;              //가장 큰 자리수를 반환합니다.
}
```

Radix Sort

```
    public static void main(String [] args) {
    int[] x = {69, 10, 30, 2, 16, 8, 31, 22};
    for (int i = 0; i <x.length ; i++) {
                        System.out.print(x[i] + " ");
        }
        System.out.println();

        radix_sort(x);
        System.out.println("정렬 후");
        for (int i = 0; i <x.length ; i++) {
                        System.out.print(x[i] + " ");
        }
        System.out.println();
  }
}
```

Radix Sort

69 10 30 2 16 8 31 22
정렬 후
2 8 10 16 22 30 31 69

Heap Sort

- Heap Sort
  - 히프 자료 구조를 이용한 정렬 방법
  - 히프에서는 항상 가장 큰 원소가 루트 노드가 되고 삭제 연산을 수행하면 항상 루트 노드의 원소를 삭제하여 반환
  - 최대 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 내림차순으로 정렬 수행
  - 최소 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 오름차순으로 정렬 수행
  - 오름차순 정렬
    - 최대 heap을 구성
    - 원소 개수 만큼의 배열을 생성
    - heap의 삭제 연산을 데이터의 개수만큼 반복 수행하는데 이 때 리턴되는 데이터를 배열의 마지막에서부터 저장
    - 삭제 연산이 종료되면 배열의 데이터도 오름차순으로 정렬됨

Heap Sort

- Heap Sort
  - ✓ 메모리 사용 공간
    - 원소 n개에 대해서 n개의 메모리 공간 사용
    - 크기 n의 히프 저장 공간
    - 연산 시간
    - 히프 재구성 연산 시간
      - n개의 노드에 대해서 완전 이진 트리는 $\log_2(n+1)$의 레벨을 가지므로 완전 이진 트리를 히프로 구성하는 평균시간은 $O(\log_2 n)$
      - n개의 노드에 대해서 n번의 히프 재구성 작업 수행
    - 평균 시간 복잡도 : $O(n \log_2 n)$

Heap Sort

```
class HeapSort {
    // 배열 요소 a[idx1]과 a[idx2]의 값을 바꿉니다.
    static void swap(int[] a, int idx1, int idx2) {
            int t = a[idx1];
            a[idx1] = a[idx2];
            a[idx2] = t;
    }
```

Heap Sort

```
// a[left] ~ a[right]를 힙으로 만듭니다.
static void downHeap(int[] a, int left, int right) {
        int temp = a[left];                                     // 루트
        int child;                                              // 큰 값을 가진 노드
        int parent;                                             // 부모

        for (parent = left; parent < (right + 1) / 2; parent = child) {
                int cl = parent * 2 + 1;
                        // 왼쪽 자식
                int cr = cl + 1;
                                        // 오른쪽 자식
                child = (cr <= right && a[cr] > a[cl]) ? cr : cl;       // 큰 값을 가진 노드
를 자식에 대입
                if (temp >= a[child])
                        break;
                a[parent] = a[child];
        }
        a[parent] = temp;
}
```

Heap Sort

```
// 힙 정렬
static void heapSort(int[] a, int n) {
        for (int i = (n - 1) / 2; i >= 0; i--) // a[i] ~ a[n-1]를 힙으로 만들기
                downHeap(a, i, n - 1);

        for (int i = n - 1; i > 0; i--) {
                swap(a, 0, i);                                  // 가장 큰 요소와 아
직 정렬되지 않은 부분의 마지막 요소를 교환
                downHeap(a, 0, i - 1);          // a[0] ~ a[i-1]을 힙으로 만듭니다.
        }
}
```

Heap Sort

```
    public static void main(String[] args) {
            int[] x = {69, 10, 30, 2, 16, 8, 31, 22};
            for (int i = 0; i <x.length ; i++) {
                    System.out.print(x[i] + " ");
            }
            System.out.println();
            heapSort(x, x.length);  // 배열 x를 힙 정렬합니다.
            System.out.println("정렬 후");
            for (int i = 0; i <x.length ; i++) {
                    System.out.print(x[i] + " ");
            }
            System.out.println();
    }
}
```

Heap Sort

69 10 30 2 16 8 31 22
정렬 후
2 8 10 16 22 30 31 69