

자료구조론

11장 정렬(sort)

□ 이 장에서 다룰 내용

- ❖ 정렬
- ❖ 선택 정렬
- ❖ 버블 정렬
- ❖ 삽입 정렬
- ❖ 퀵 정렬
- ❖ 병합 정렬
- ❖ 기수 정렬

□ 정렬

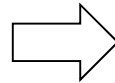
❖ 정렬(sort)

- 순서 없이 배열된 자료들을 어떤 기준에 따라 오름차순(ascending order)으로 또는 내림차순(descending order)으로 재배열하는 것
- 자료를 정렬하는 데 기준이 되는 특정 값을 키(key)라고 함

❖ 정렬의 예

- 컴퓨터 아이콘들을 이름 순으로 정렬
- 학생들의 정보를 성적을 기준으로 내림차순 정렬

점수 번호	학년	이름	성적
1	2	이	60
2	3	김	90
3	1	박	80
4	2	김	70
5	2	최	80



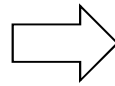
점수 번호	학년	이름	성적
2	3	김	90
3	1	박	80
5	2	최	80
4	2	김	70
1	2	이	60

□ 정렬

❖ 정렬의 예

- 학생들의 정보를 학년-이름을 기준으로 오름차순 정렬

접수 번호	학년	이름	성적
1	2	이	60
2	3	김	90
3	1	박	80
4	2	김	70
5	2	최	80



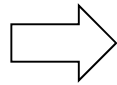
접수 번호	학년	이름	성적
3	1	박	80
4	2	김	70
1	2	이	60
5	2	최	80
2	3	김	90

□ 정렬

❖ 안정 정렬(stable sort)이 필요한 경우도 있음

- 학생들의 정보를 성적을 기준으로 내림차순 정렬

접수 번호	학년	이름	성적
1	2	이	60
2	3	김	90
3	1	박	80
4	2	김	70
5	2	최	80



접수 번호	학년	이름	성적
2	3	김	90
3	1	박	80
5	2	최	80
4	2	김	70
1	2	이	60

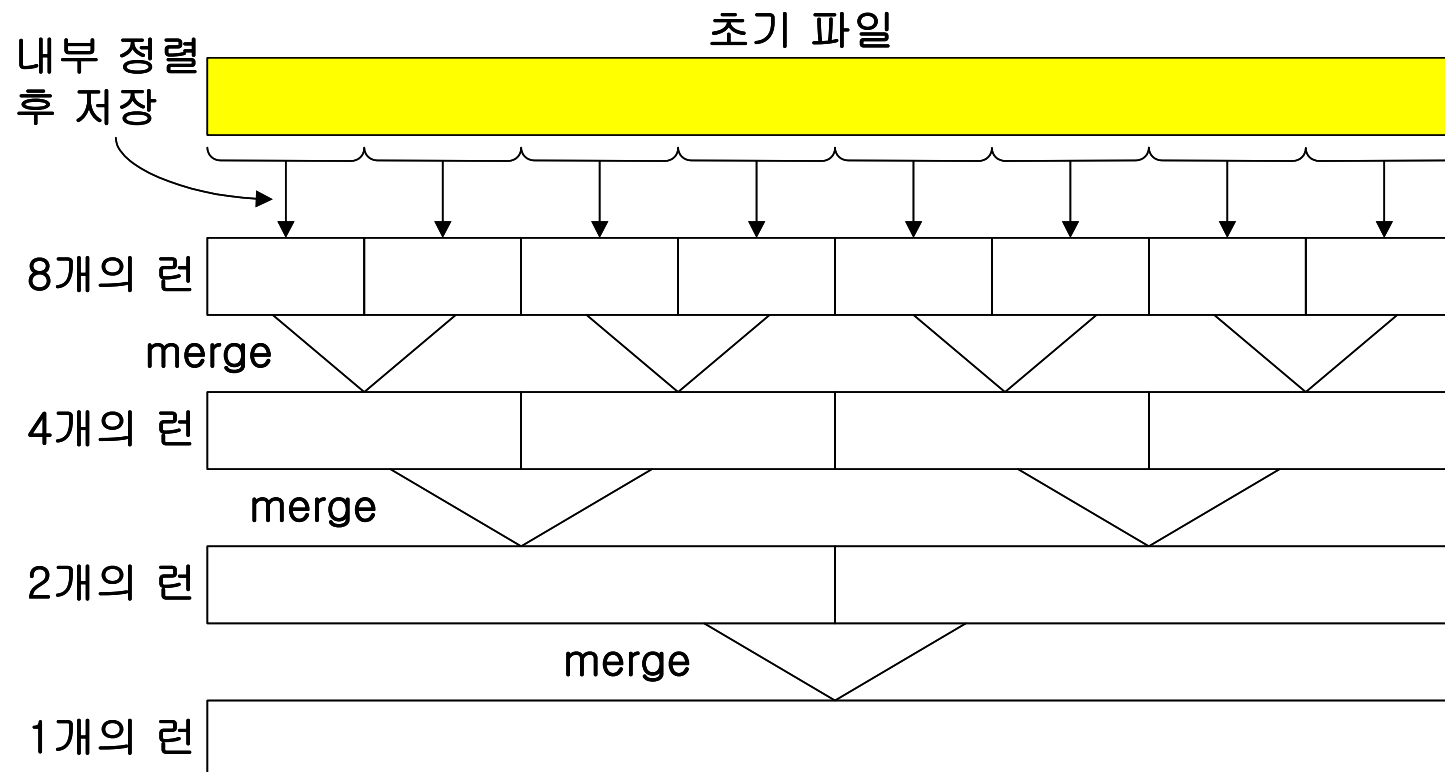
접수 번호	학년	이름	성적
2	3	김	90
5	2	최	80
3	1	박	80
4	2	김	70
1	2	이	60

❖ 정렬의 분류

- 내부 정렬(internal sort)
 - 정렬할 자료를 메인 메모리에 올려서 정렬하는 방식
 - 정렬 속도가 빠르지만 정렬할 수 있는 자료의 양이 메인 메모리의 용량에 따라 제한됨
 - 효율성 기준 : 비교 횟수, 자료 이동 횟수, 추가 기억장소 필요량
- 외부 정렬(external sort)
 - 정렬할 자료를 보조 기억장치에 두고 정렬하는 방식
 - 내부 정렬보다 속도는 떨어지지만 내부 정렬로 처리할 수 없는 대용량 자료에 대한 정렬 가능
 - 효율성 기준 : 보조 기억장치로의 입력과 출력 횟수

❖ 외부 정렬 방식

- 병합 방식 : 파일을 부분 파일로 분리하여 각각을 내부 정렬 방법으로 정렬하여 병합하는 정렬 방식
 - 2-way 병합, k-way 병합, ...
- 2-way 병합 예:



❖ 내부 정렬 방식

- 기본 정렬 : 알고리즘은 단순하지만, 평균 시간 복잡도 $O(n^2)$ 이므로 정렬할 원소 수가 작을 때 유용
 - 선택 정렬(selection sort)
 - 버블 정렬(bubble sort)
 - 삽입 정렬(insertion sort)
- 고급 정렬 : 평균 시간 복잡도 $O(n \log n)$
 - 퀵 정렬(quick sort)
 - 병합 정렬(merge sort)
 - 힙 정렬(heap sort)
- 원소끼리의 비교에 기반하지 않은 특수 정렬
 - 기수 정렬(radix sort) – 버킷 정렬(bucket sort)
 - 계수 정렬(counting sort)

□ 선택 정렬

❖ 선택 정렬(selection sort)

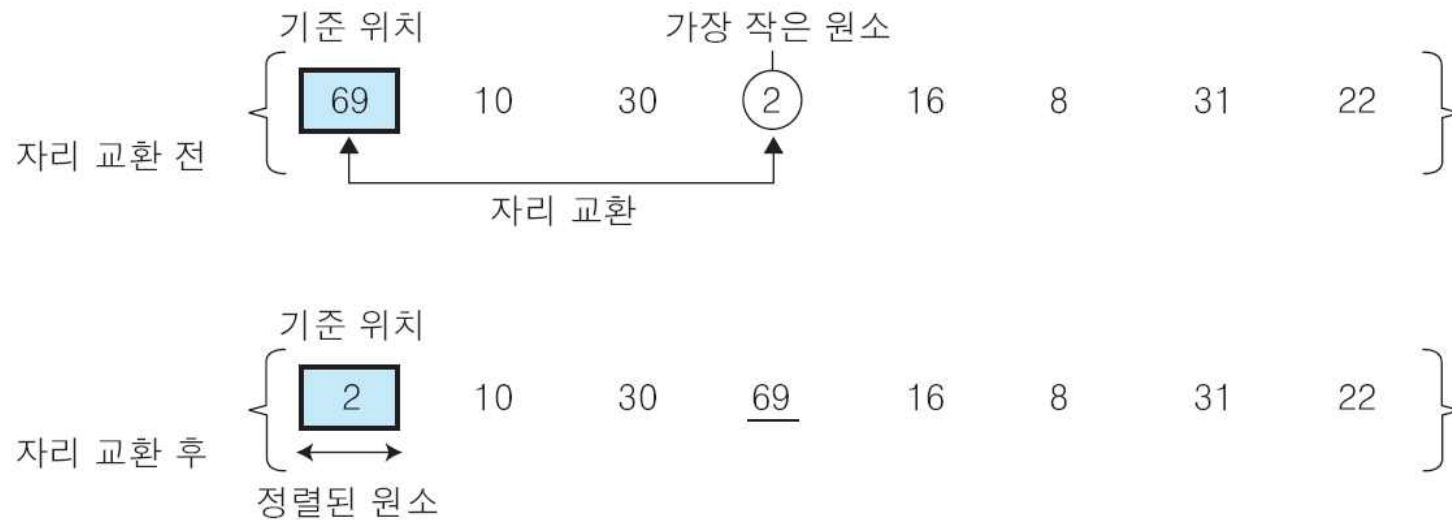
- 전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 그 기준 위치에 저장하는 방식으로 정렬한다.
- 수행 방법
 - 전체 원소 중에서 가장 작은 원소를 찾아서 선택하여 첫 번째 원소와 자리를 교환한다.
 - 두 번째로 작은 원소를 찾아 선택하여 두 번째 원소와 자리를 교환한다.
 - 세 번째로 작은 원소를 찾아서 세 번째 원소와 자리를 교환한다.
 - 이 과정을 반복하면서 정렬을 완성한다.

□ 선택 정렬

❖ 선택 정렬 수행 과정

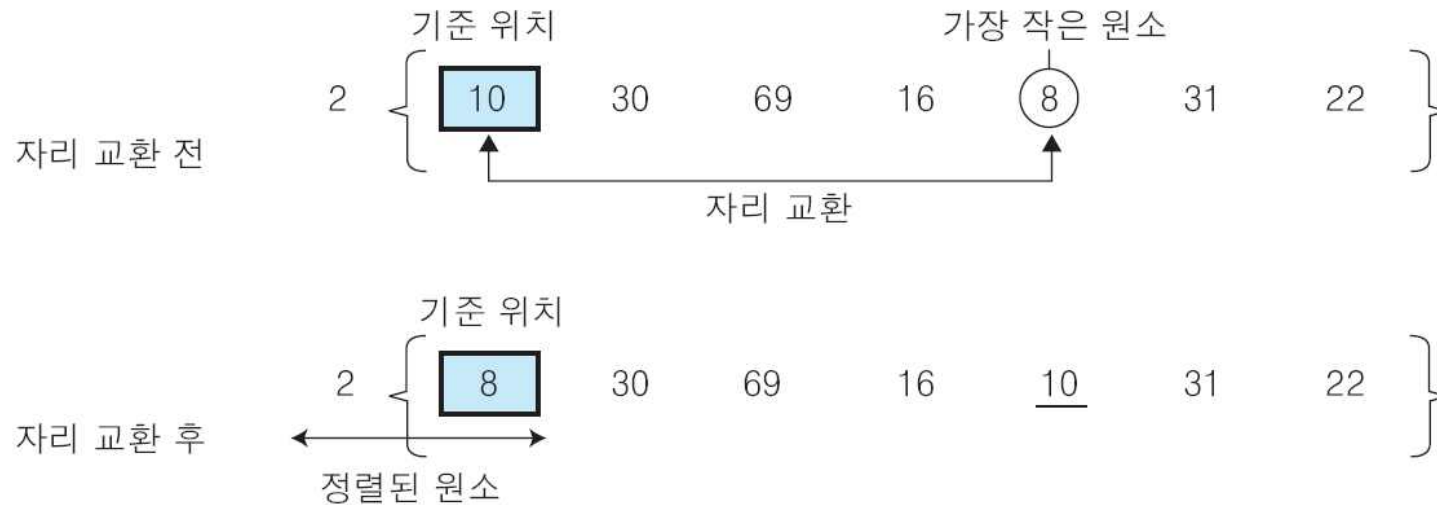
- 69, 10, 30, 2, 16, 8, 31, 22

① 첫 번째 자리를 기준 위치로 정하고, 전체 원소 중에서 가장 작은 원소 2를 선택하여 기준 위치에 있는 원소와 자리 교환



□ 선택 정렬

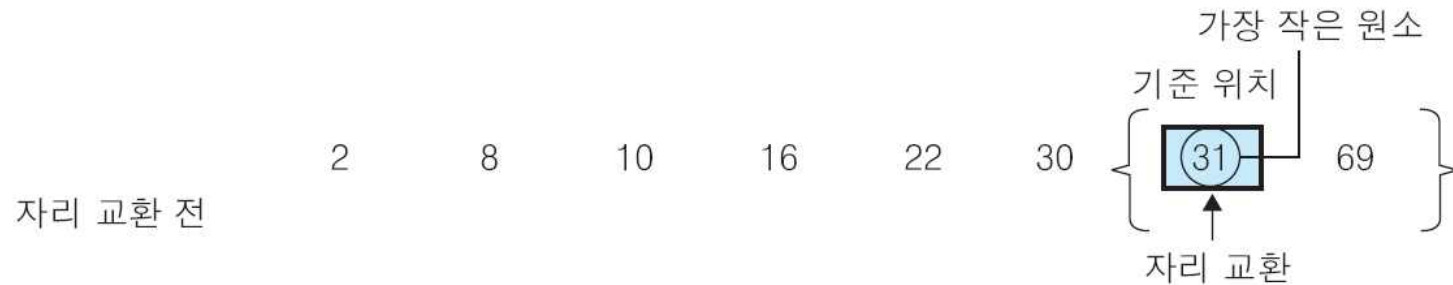
- ② 두 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 8을 선택하여 기준 위치에 있는 원소와 자리 교환



이 과정을 반복 ...

□ 선택 정렬

- ⑦ 일곱 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 31을 선택하여 기준 위치 원소와 자리 교환. (제자리)



- ⑧ 마지막에 남은 원소 69는 전체 원소 중에서 가장 큰 원소로서 이미 마지막 자리에 정렬된 상태이므로 실행을 종료하고 선택 정렬이 완성된다.

선택 정렬 완성 2 8 10 16 22 30 31 69

□ 선택 정렬

❖ 선택 정렬 알고리즘

```
selectionSort(a[], n) // a[0..n-1] 을 정렬
    for (i ← 0; i < n-1; i ← i+1) {
        a[i..n-1] 중에서 가장 작은 원소 a[k]를 선택하여, a[i]와 교환한다;
    }
end selectionSort()
```

□ 선택 정렬

❖ 선택 정렬 알고리즘 분석

- 추가 기억장소 필요량 : $O(1)$
- 연산 시간

1단계 : $n-1$ 개의 원소 비교

2단계 : $n-2$ 개의 원소 비교

3단계 : $n-3$ 개의 원소 비교

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

➔ 시간 복잡도 $O(n^2)$

□ 선택 정렬

❖ 선택 정렬 프로그램

```
public class Main
{
    public static void main(String [] args)
    {
        int [] a = {69, 10, 30, 2, 16, 8, 31, 22};

        System.out.print("정렬 전 : ");
        for(int i=0; i<a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();

        Sort.selectionSort(a);

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

□ 선택 정렬

```
public class Sort
{
    public static void selectionSort(int [] a) {
        int min;
        for(int i=0; i<a.length-1; i++) {
            // a[i], a[i+1], ... 중에서 최소값의 인덱스 min을 찾음
            min = i;
            for(int j=i+1; j<a.length; j++) {
                if(a[j] < a[min])
                    min = j;
            }
            // a[i]와 최소 원소 a[min]의 자리를 교환
            swap(a, min, i);
        }
    }
}
```

min ?

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

```
private static void swap(int [] a, int i, int j) {
    // a[i]와 a[j]를 교환
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
}
```


❖ 버블 정렬(bubble sort)

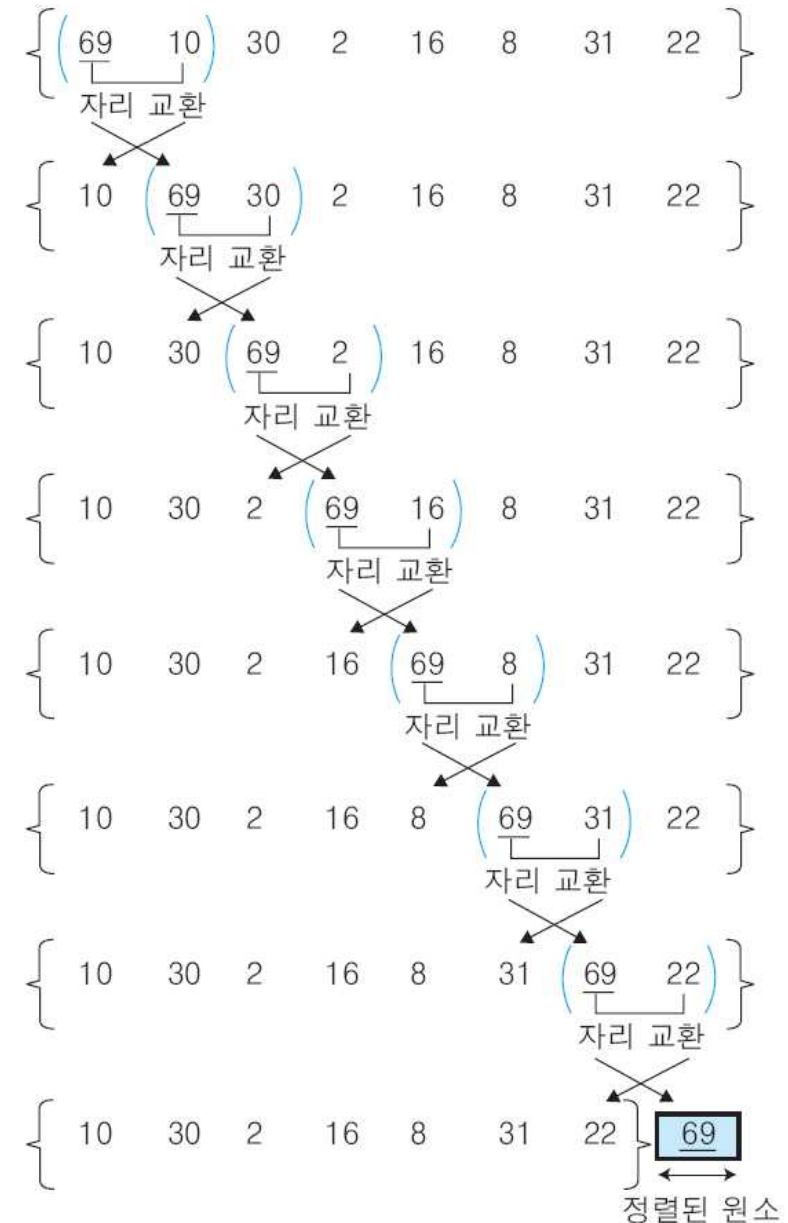
- 인접한 두 개의 원소를 비교하여 정렬 기준에 맞지 않으면 자리를 교환하는 방식으로 정렬한다.
- 수행 방법
 - 첫 번째 원소부터 마지막 원소까지 비교-교환 작업을 반복하면 가장 큰 원소가 마지막 위치에 놓인다.
 - 가장 큰 원소를 제외하고, 나머지 원소들에 대해 위의 작업을 반복하면 두번째 큰 원소가 뒤에서 두번째 위치에 놓인다.
 - 가장 큰 원소 두개를 제외하고 위의 작업을 반복하면 세번째 큰 원소가 뒤에서 세번째 위치에 놓인다.
 - 이 과정을 반복하면서 정렬을 완성한다.

□ 버블 정렬

❖ 버블 정렬 수행 과정

- 69, 10, 30, 2, 16, 8, 31, 22

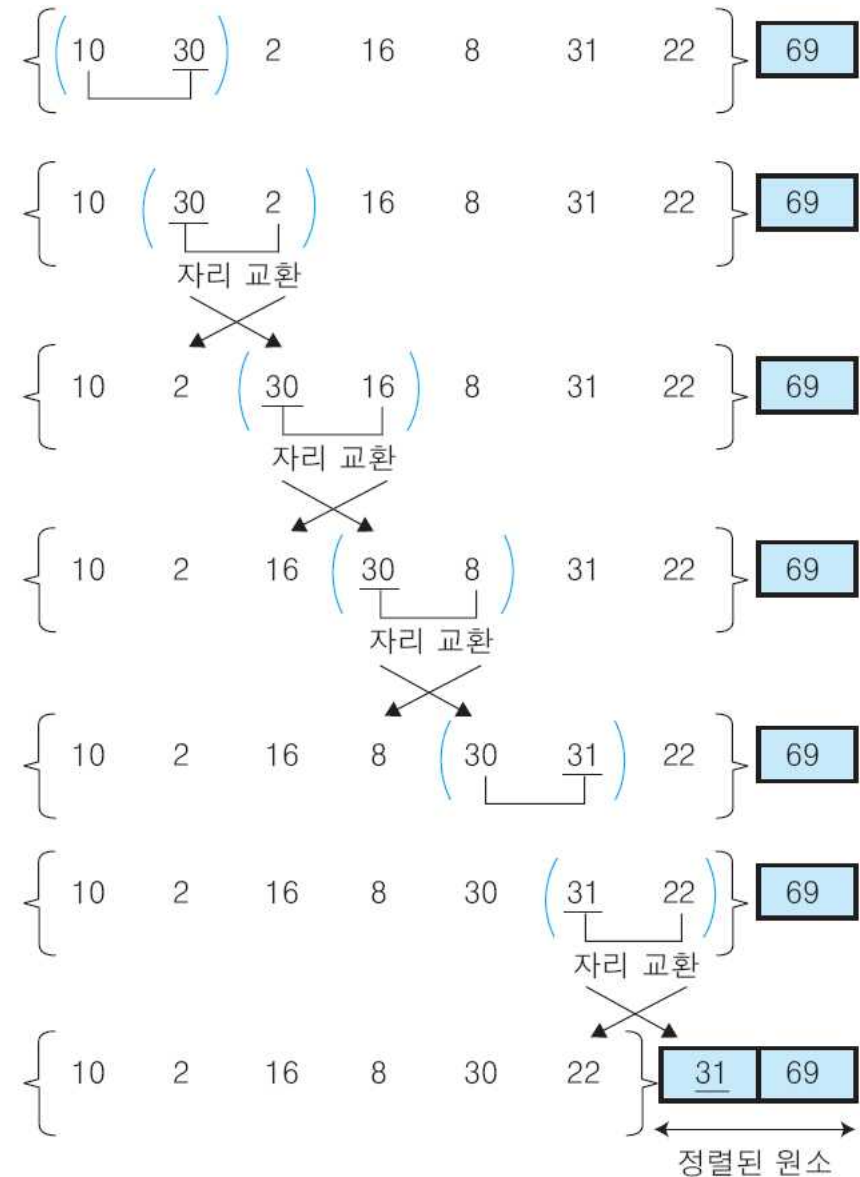
① 단계 1: 인접한 두 원소를 비교하여 자리를 교환하는 작업을 첫 번째 원소부터 마지막 원소까지 차례로 반복하여 69를 가장 뒤로 보냄



□ 버블 정렬

② 단계 2: 다음 단계를 수행하여 나머지 원소 중에서 가장 큰 원소 31을 끝에서 두 번째 자리로 보냄

이 과정을 반복...



□ 버블 정렬

⑦ 단계 7: 8을 끝에서 일곱 번째 자리로 보냄



마지막에 남은 첫 번째 원소는 전체 원소 중에서 가장 작은 원소로 이미 정렬된 상태이므로 정렬 완성

버블 정렬 완성

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

❑ 버블 정렬

❖ 버블 정렬 알고리즘

```
bubbleSort(a[], n) // a[0..n-1] 을 정렬
  for (i ← n-1; i > 0; i ← i-1) {
    for (j ← 0; j < i; j ← j+1) {
      if (a[j] > a[j+1]) then {
        a[j]와 a[j+1]을 교환;
      }
    }
  }
end bubbleSort()
```

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

□ 버블 정렬

❖ 버블 정렬 알고리즘 분석

- 추가 기억장소 필요량 : $O(1)$
 - 연산 시간
 - 최선의 경우 : 자료가 이미 정렬되어있는 경우
 - 비교횟수 : i 번째 원소를 $(n-i)$ 번 비교하므로, $n(n-1)/2$ 번
 - 자리교환횟수 : 자리교환이 발생하지 않는다.
 - 최악의 경우 : 자료가 역순으로 정렬되어있는 경우
 - 비교횟수 : i 번째 원소를 $(n-i)$ 번 비교하므로, $n(n-1)/2$ 번
 - 자리교환횟수 : i 번째 원소를 $(n-i)$ 번 교환하므로, $n(n-1)/2$ 번
- ➔ 평균 시간 복잡도 : $O(n^2)$

□ 삽입 정렬

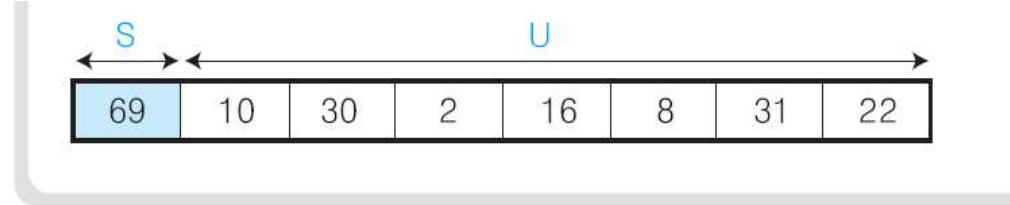
❖ 삽입 정렬(insertion sort)

- 이미 정렬되어있는 부분에 새로운 원소의 위치를 찾아 삽입하는 방식으로 정렬한다.
- 수행 방법
 - 원소들을 두 개의 부분 S와 U로 나누어 생각하자.
 - S : 이미 정렬된 앞부분의 원소들
 - U : 아직 정렬되지 않은 나머지 원소들
 - U의 원소를 하나씩 꺼내서 S의 마지막 원소부터 비교하면서 위치를 찾아 삽입한다.
 - U가 공집합이 되면 삽입 정렬이 완성된다.

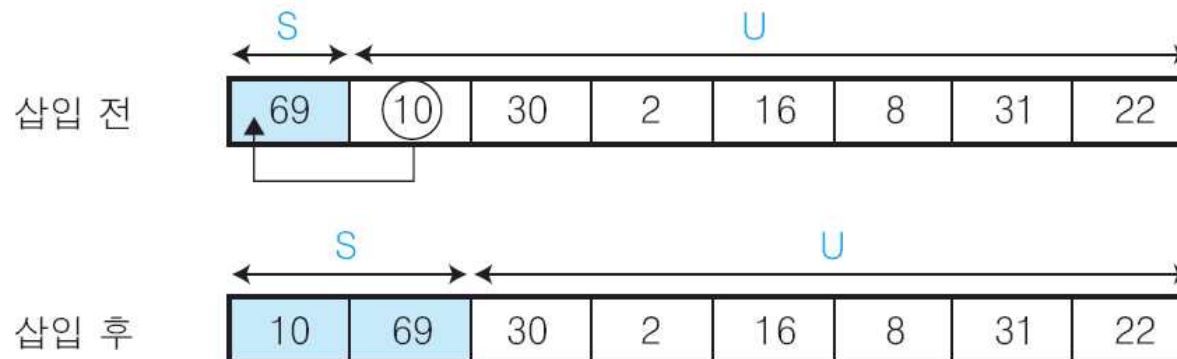
❑ 삽입 정렬

❖ 삽입 정렬 수행 과정

- 69, 10, 30, 2, 16, 8, 31, 22
- 초기 상태
 - 첫 번째 원소는 정렬되어있는 S
 - 나머지 원소들은 정렬되지 않은 U

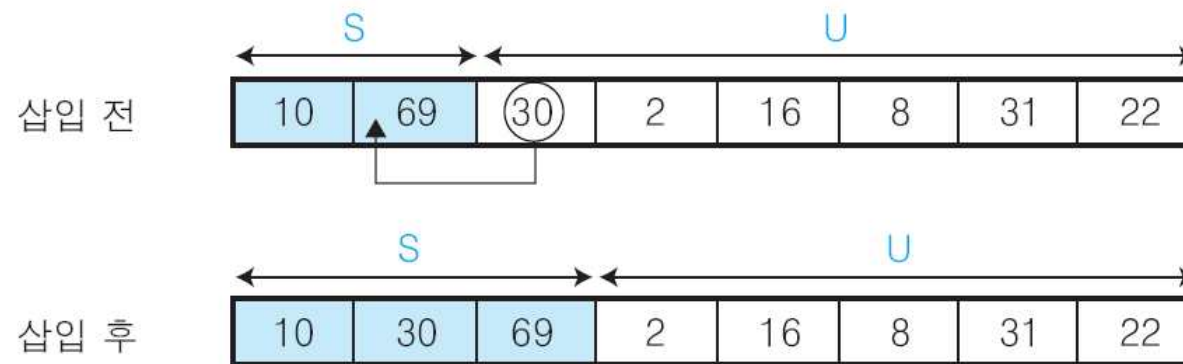


- ① U의 첫 번째 원소 10을 S의 마지막 원소 69와 비교하여 ($10 < 69$) 이므로 원소 10은 원소 69의 앞자리가 된다. 더 이상 비교할 S의 원소가 없으므로 찾은 위치에 원소 10을 삽입한다.



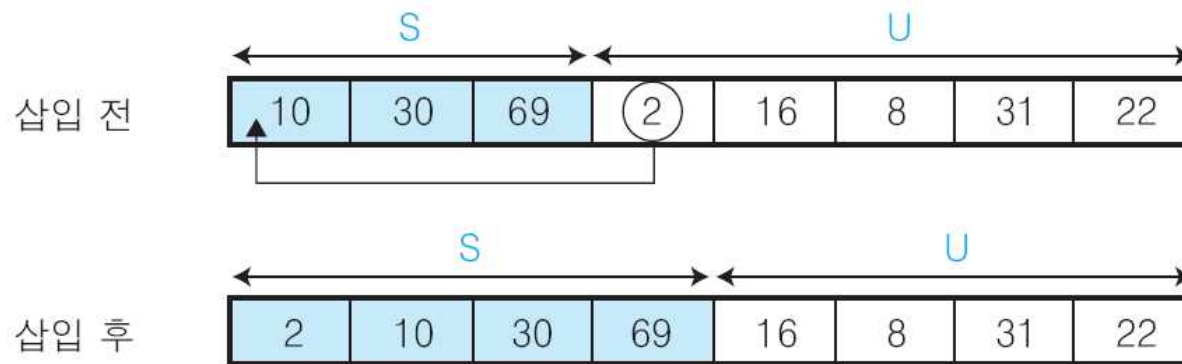
□ 삽입 정렬

- ② U의 첫 번째 원소 30을 S의 마지막 원소 69와 비교하여 ($30 < 69$) 이므로 원소 69의 앞자리 원소 10과 비교한다. ($30 > 10$) 이므로 원소 10과 69 사이에 삽입한다.



□ 삽입 정렬

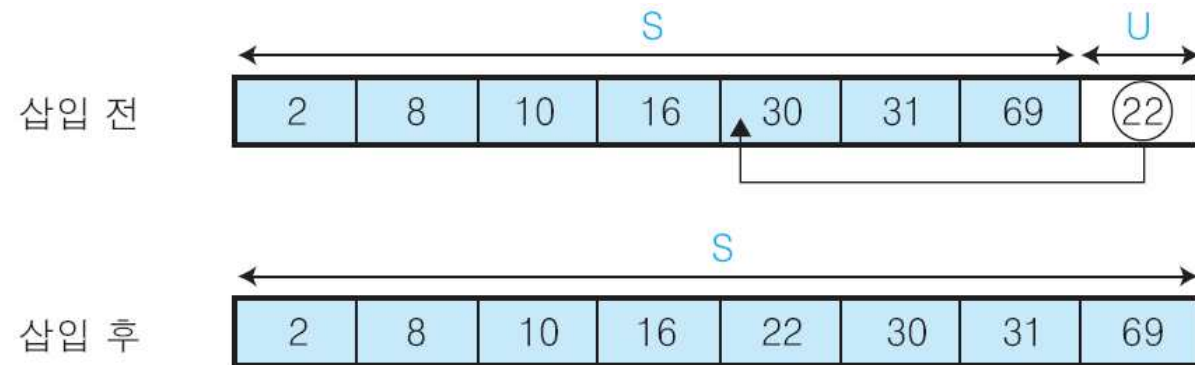
- ③ U의 첫 번째 원소 2를 S의 마지막 원소 69와 비교하여 ($2 < 69$)
이므로 원소 69의 앞자리 원소 30과 비교하고,
($2 < 30$) 이므로 다시 그 앞자리 원소 10과 비교하는데,
($2 < 10$) 이면서 더 이상 비교할 S의 원소가 없으므로 원소 10의
앞에 삽입한다.



이 과정을 반복...

□ 삽입 정렬

- ⑦ U의 첫 번째 원소 22를 S의 원소들과 비교하여 16과 30 사이에 삽입한다.

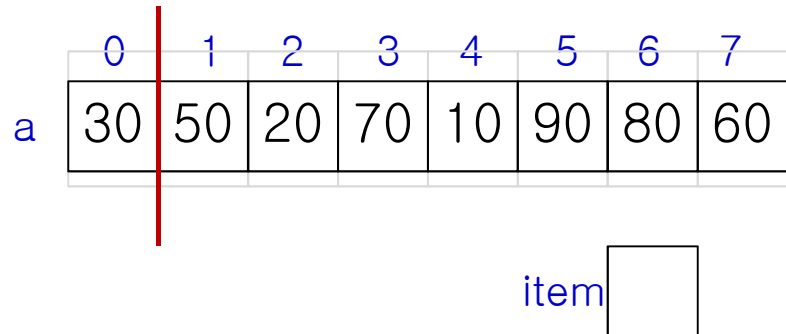


U가 공집합이 되었으므로 삽입 정렬이 완성된다.

❑ 삽입 정렬

❖ 삽입 정렬 프로그램

```
public class Sort
{
    public static void insertionSort(int a[]) { // a[0..a.length-1]을 정렬
        int i, j, item;
        for(i=1; i<a.length; i++) {
            item = a[i];
            // item이 삽입될 위치 j를 찾음
            for(j = i; (j>0) && (a[j-1]>item); j--) {
                a[j] = a[j-1];
            }
            a[j] = item;
        }
    }
}
```



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

a

item

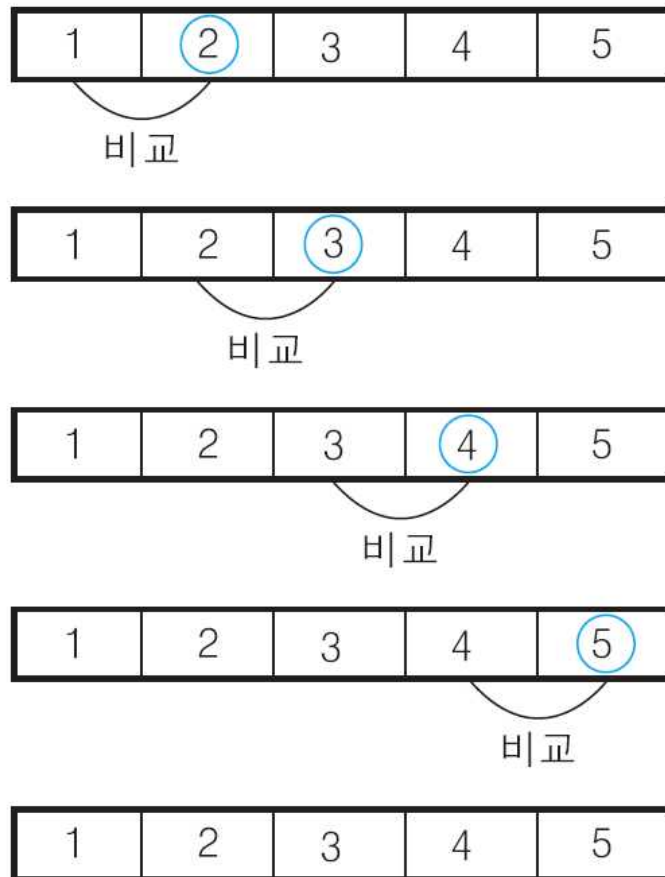
□ 삽입 정렬

❖ 삽입 정렬 알고리즘 분석

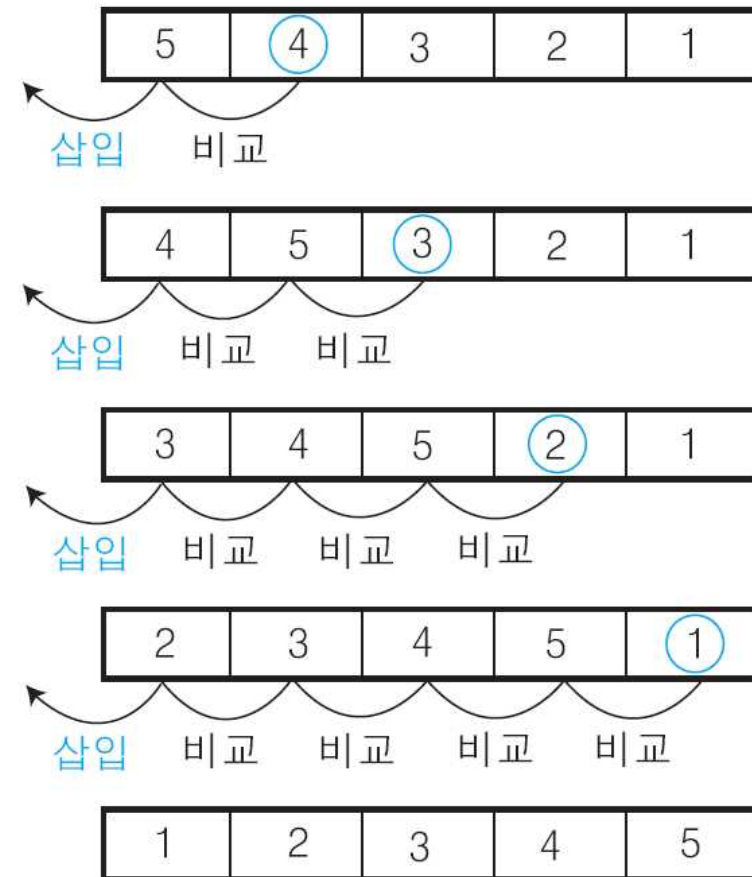
- 추가 기억장소 필요량 : $O(1)$
 - 연산 시간
 - 최선의 경우 : 원소들이 이미 정렬되어있어 비교횟수가 최소
 - 바로 앞자리 원소와 한번만 비교하므로 전체 비교횟수 = $n-1$
 - 시간 복잡도 : $O(n)$
 - 최악의 경우 : 모든 원소가 역순으로 되어있어서 비교횟수가 최대
 - 전체 비교횟수 = $1+2+3+\dots+(n-1) = n(n-1)/2$
 - 시간 복잡도 : $O(n^2)$
 - 삽입 정렬의 평균 비교횟수 = $n(n-1)/4$
- ➔ 평균 시간 복잡도 : $O(n^2)$

□ 삽입 정렬

- 최선의 경우: 입력 데이터가 이미 정렬되어 있음



- 최악의 경우: 입력 데이터가 역순으로 정렬되어 있음



□ 퀵 정렬

❖ 퀵 정렬(quick sort)

- 기준 값을 중심으로 왼쪽 부분과 오른쪽 부분으로 분할한 후, 이 두 부분을 따로 정렬하여 모으는 방식으로 정렬한다.
 - 분할 과정에서 왼쪽 부분에는 기준 값보다 작은 원소들을 이동시키고, 오른쪽 부분에는 기준 값보다 큰 원소들을 이동시킨다.
- 기준 값 : 피벗(pivot)
 - 기준 값을 정하는 방법은 다양하나, 전체 원소 중에서 가운데에 위치한 원소를 선택하기도 함
- 분할-정복(divide-and-conquer) 기법의 알고리즘임
 - divide
 - conquer
 - combine

□ 퀵 정렬

❖ 퀵 정렬 알고리즘

```
quickSort(a[], begin, end) // a[begin..end] 를 정렬
  if (begin < end) then {
    p ← partition(a, begin, end); // a[begin.. end]를
                                // a[begin..p-1]와 a[p+1..end]로
                                // 분할하고, 기준값은 a[p]에 저장
    quickSort(a, begin, p-1); // 왼쪽 분할을 정렬함
    quickSort(a, p+1, end);  // 오른쪽 분할을 정렬함
  }
end quickSort()
```


퀵 정렬

❖ 퀵 정렬 알고리즘의 partition 알고리즘

```
partition(a[], begin, end) // a[begin..end]를 분할한 후, 기준값 위치를 리턴
    pivot ← A[begin];
    i ← begin;
    j ← end + 1;
    do {
        do { i = i + 1; } while (i ≤ end && A[i] < pivot);
        do { j = j - 1; } while (A[j] > pivot);
        if (i < j) then swap(A[i], A[j]);
    } while (i < j);
    swap(A[begin], A[j]);
    return j;
end partition()
```

□ 퀵 정렬

❖ 퀵 정렬 알고리즘 분석

- 추가 기억장소 필요량 : $O(1)$

- 연산 시간

- 최선의 경우

- 피봇에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 $n/2$ 개씩 이등분이 되는 경우가 반복되어 수행 단계 수가 최소($\log_2 n$)가 되는 경우

- 시간 복잡도 : $O(n \log_2 n)$

- 최악의 경우

- 피봇에 의해 원소들을 분할했을 때 0개와 $n-1$ 개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대(n)가 되는 경우

- 시간 복잡도 : $O(n^2)$

- ➔ 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 방법임

□ 병합 정렬

❖ 병합 정렬(merge sort)

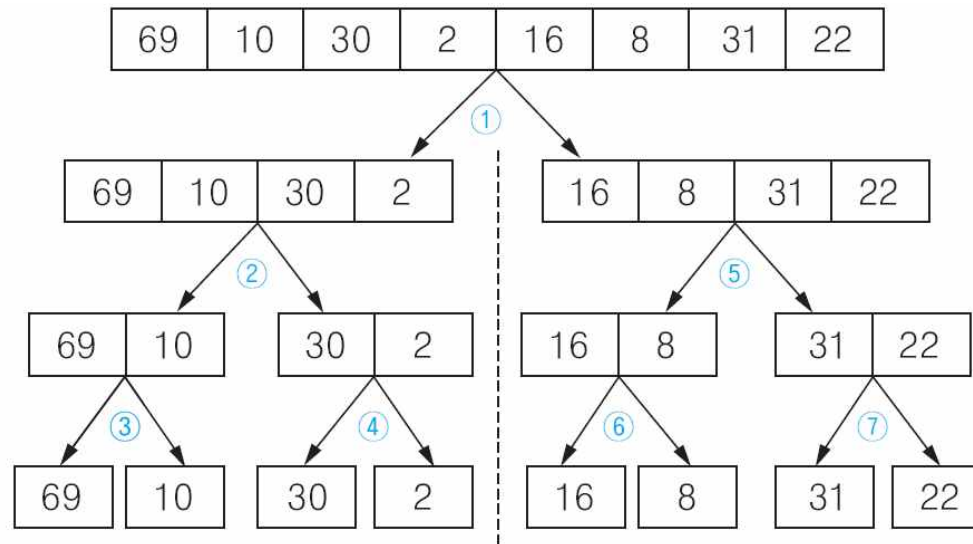
- 정렬된 자료 집합들을 하나로 병합하는 방식으로 정렬한다.
- 분할-정복(divide-and-conquer) 기법의 알고리즘임
 - divide : 주어진 입력 원소들을 같은 크기의 2개의 부분으로 분할
 - conquer : 각 부분을 따로 정렬
 - combine : 정렬된 2개의 부분들을 하나로 병합
- 병합 정렬 방법의 종류
 - 2-way 병합 : 위와 같이 2개의 정렬된 자료의 집합을 병합
 - k-way 병합 : k 개의 정렬된 자료의 집합을 병합

❑ 병합 정렬

❖ 병합 정렬 수행 과정

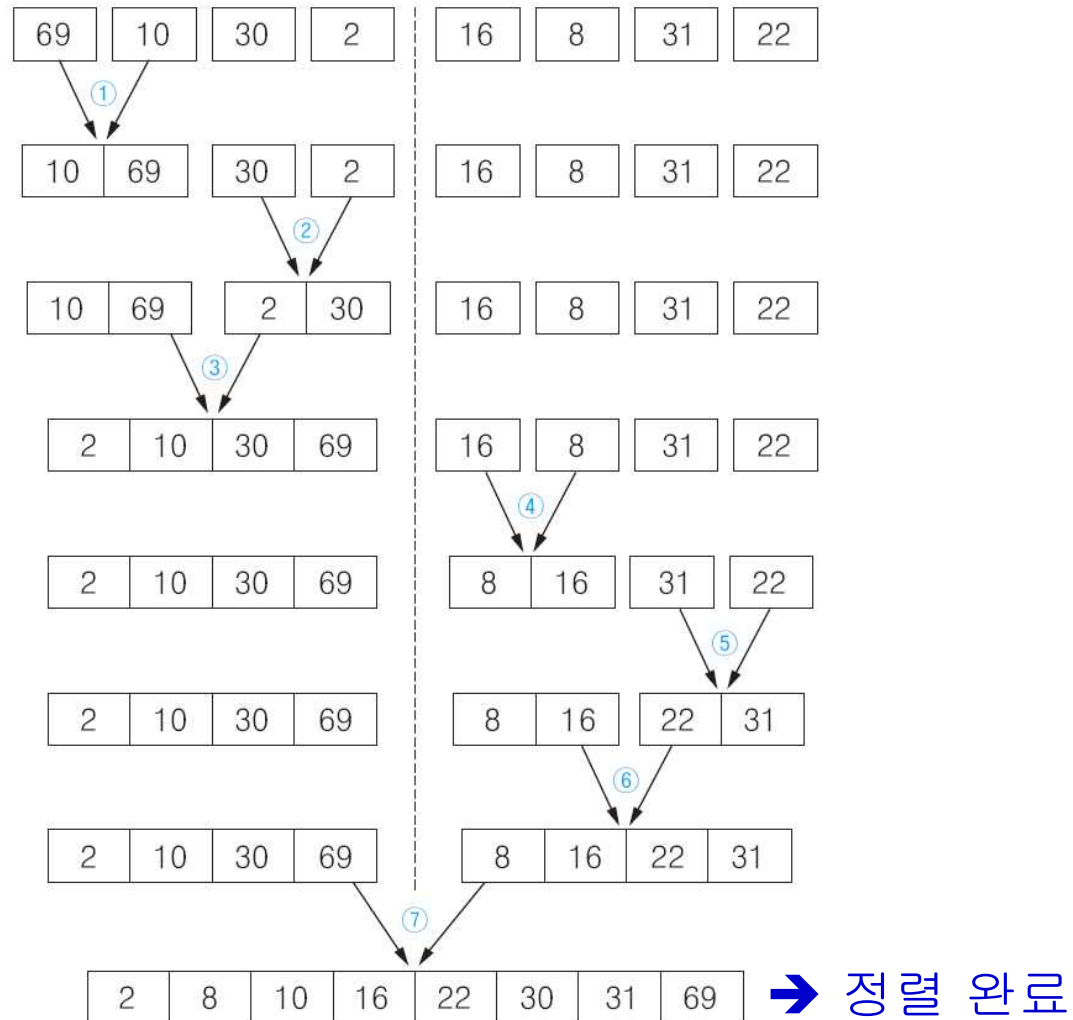
- 69, 10, 30, 2, 16, 8, 31, 22

① 분할 단계 : 정렬할 전체 자료의 집합에 대해서 분할작업을 반복하여 1개의 원소를 가진 부분집합 8개를 만든다.



□ 병합 정렬

② 병합단계 : 정렬된 2개의 부분을 하나로 병합한다. 모든 원소가 병합될 때까지 계속한다.



□ 병합 정렬

❖ 병합 정렬 알고리즘

```
mergeSort(a[], m, n) // a[m..n]을 정렬
  if (m < n) then {
    middle ← (m+n)/2;
    mergeSort(a, m, middle);
    mergeSort(a, middle+1, n);
    merge(a, m, middle, n); // a[m..middle]과 a[middle+1..n]을 병합
  }
end mergeSort()
```

□ 병합 정렬

❖ 병합 정렬 알고리즘 분석

- 추가 기억장소 필요량 : $O(n)$
 - 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
 - 연산 시간
 - 분할 : n 개의 원소를 분할하기 위해서 $\log_2 n$ 번의 단계 수행
 - 병합 : 각 단계마다 최대 n 번의 비교연산 수행
- ➔ 시간 복잡도 : $O(n \log_2 n)$

□ 기수 정렬

❖ 기수 정렬(radix sort)

- 원소의 키값을 나타내는 기수를 이용한 정렬 방법

radixSort(A[], d)

```
{  
    for  $j = d$  downto 1 {  
        Do a stable sort on A[ ] by  $j^{\text{th}}$  digit;  
    }  
}
```

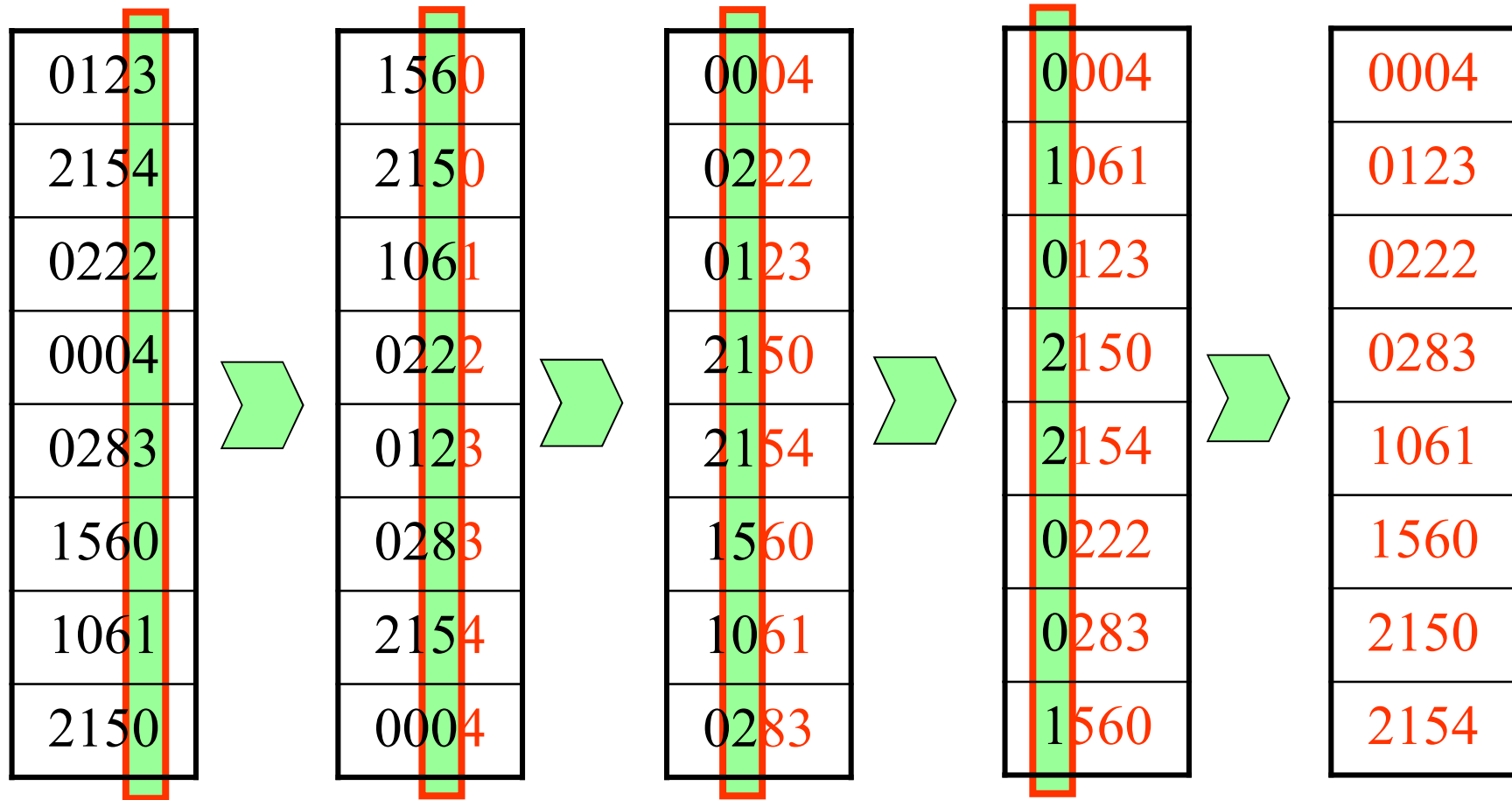
✓ 안정 정렬(stable sort)

- 같은 값을 가진 원소들 간에 정렬 후에도 원래의 순서가 유지되도록 하는 정렬

예) 정렬 전 : 8 3 5 7 5' 6

정렬 후 : 3 5 5' 6 7 8

➤ radixSort 작동 예



✓ 수행시간 : $O(n)$

- 각 digit에 대해 계수정렬을 이용하면 $O(n)$ 시간이 걸리고,
- digit 수는 상수이므로(예에서 $d = 4$) $d \times O(n) = O(n)$