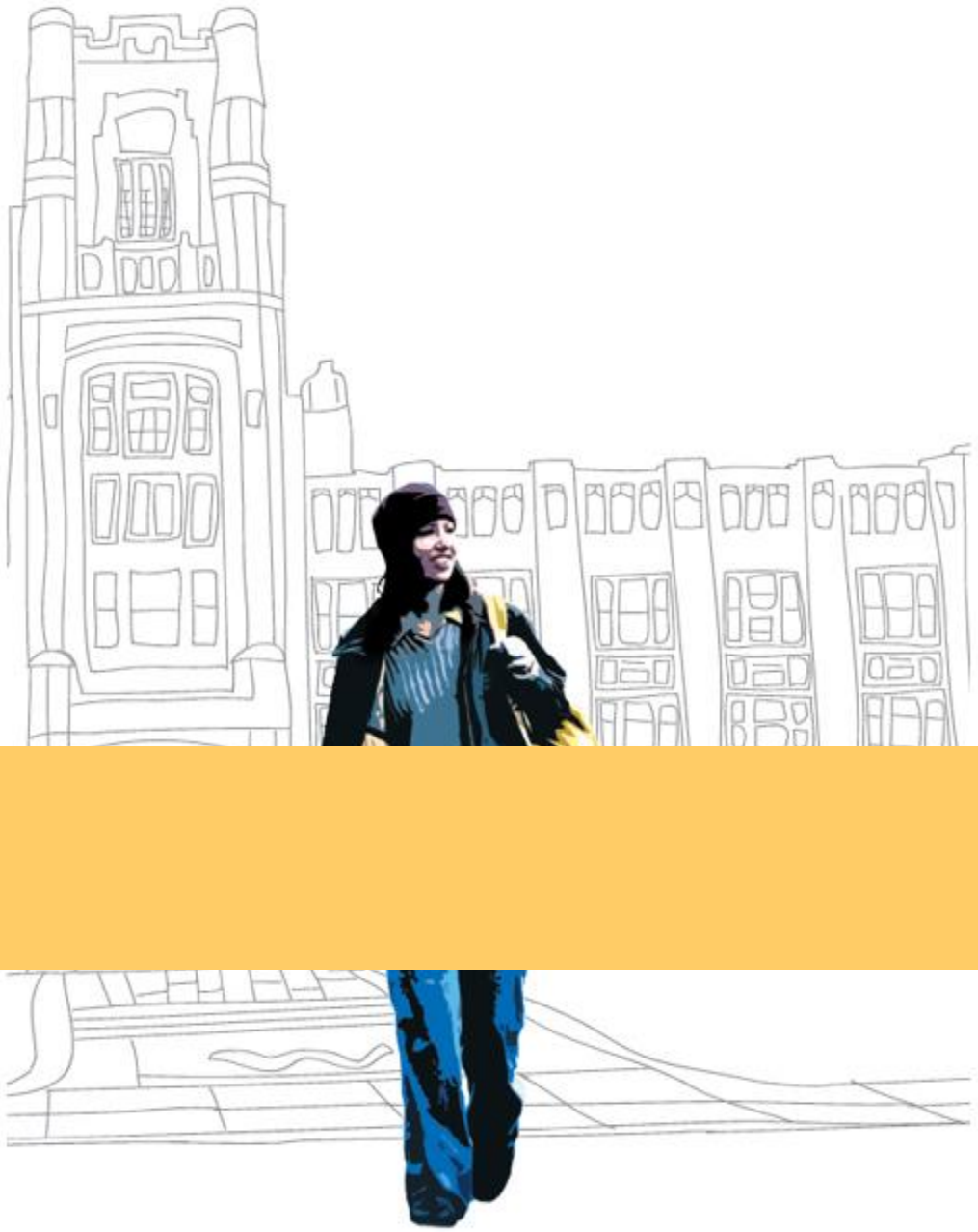


11 정렬

❑ 이 장에서 다룰 내용

1. 정렬
2. 선택 정렬
3. 버블 정렬
4. 퀵 정렬
5. 삽입 정렬
6. 셸 정렬
7. 병합 정렬

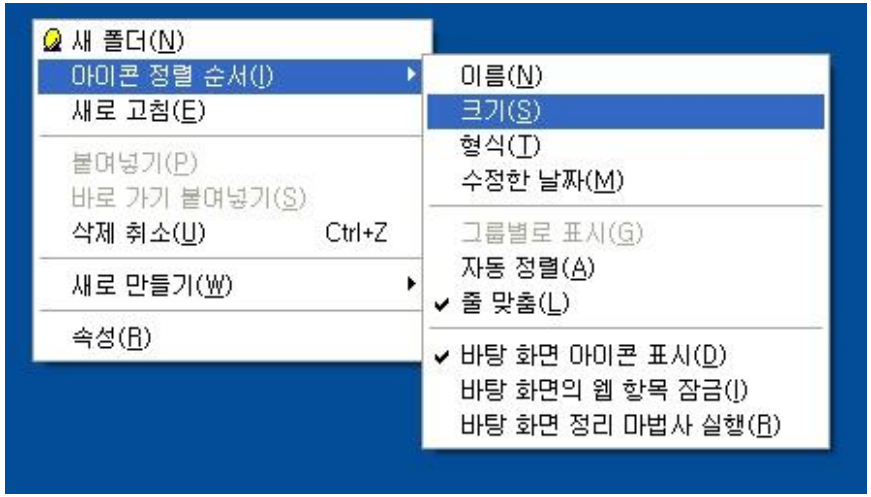
❑ 정렬

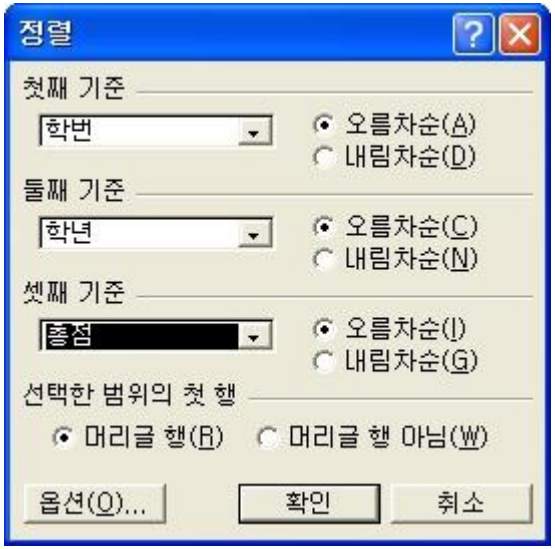
❖ 정렬(sort)

- 2개 이상의 자료를 작은 것부터 큰 순서(오름차순, ascending)로 정렬 또는 큰 것부터 작은 것 순서(내림차순, descending)로 재배열하는 것

- 키 : 자료를 정렬하는 데 사용하는 기준 값

- 정렬의 예





❖ 정렬방법의 분류

▪ 실행 방법에 따른 분류

- 비교식 정렬(comparative sort)
  - ➢ 비교하고자 하는 각 키 값들을 한번에 두 개씩 비교하여 교환하는 방식으로 정렬을 실행하는 방법
- 분산식 정렬(distribute sort)
  - ➢ 키 값을 기준으로 하여 자료를 여러 개의 부분 집합으로 분해하고, 각 부분집합을 정렬함으로써 전체를 정렬하는 방식으로 실행하는 방법

❑ 정렬

▪ 정렬 장소에 따른 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**

- 내부 정렬 방식
  - ➢ **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-way병합, n-way 병합
  - ➢ **분배 방식: 키를 구성하는 값을 여러 개의 부분집합에 분배하여 정렬하는 방식**
    - » 기수 정렬
  - ➢ **선택 방식 : 이진 트리를 사용하여 정렬하는 방식**
    - » 힙 정렬, 트리 정렬

❑ 정렬

- 외부 정렬(external sort)
  - ➢ **정렬할 자료를 보조 기억장치에서 정렬하는 방식**
  - ➢ **내부 정렬보다 속도는 떨어지지만 내부 정렬로 처리할 수 없는 대용량 자료에 대한 정렬 가능**

- 외부 정렬 방식
  - ➢ **병합 방식 : 파일을 부분 파일로 분리하여 각각을 내부 정렬 방법으로 정렬하여 병합하는 정렬 방식**
    - » 2-way 병합, n-way 병합

❑ 선택 정렬

❖ 선택 정렬(selection sort)

- **전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 자리를 교환하는 방식으로 정렬**

- **수행 방법**

① 전체 원소 중에서 가장 작은 원소를 찾아서 선택하여 첫 번째 원소와 자리를 교환한다.

② 그 다음 두 번째로 작은 원소를 찾아 선택하여 두 번째 원소와 자리를 교환한다.

③ 그 다음에는 세 번째로 작은 원소를 찾아서 세 번째 원소와 자리를 교환한다.

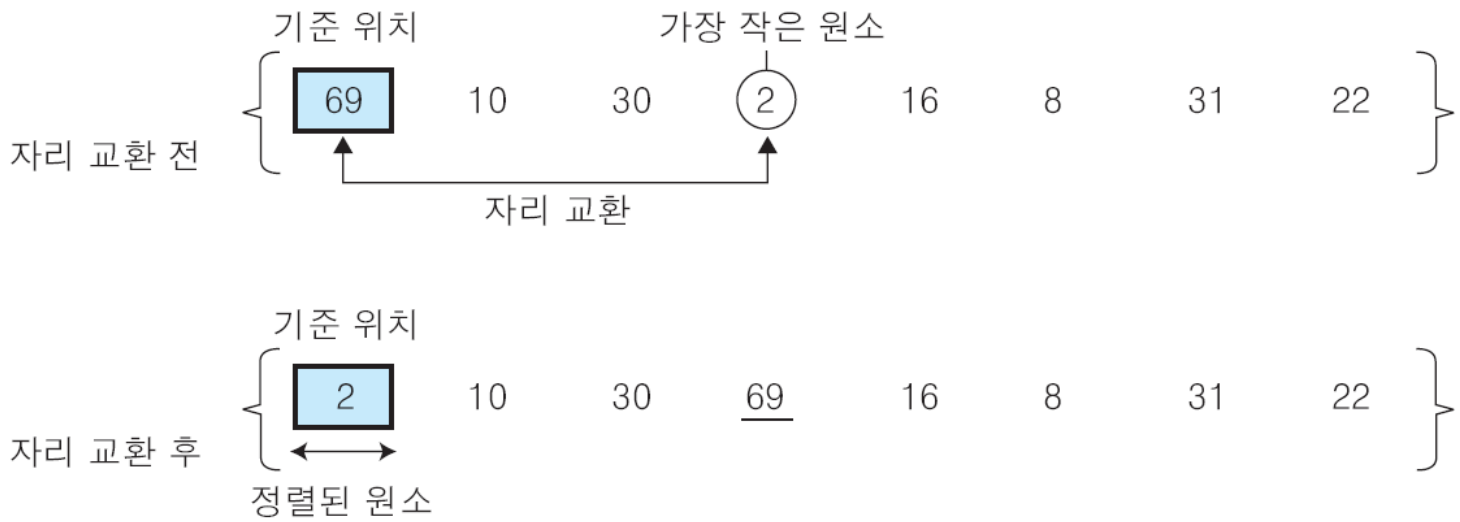
④ 이 과정을 반복하면서 정렬을 완성한다.

❑ 선택 정렬

❖ 선택 정렬 수행 과정

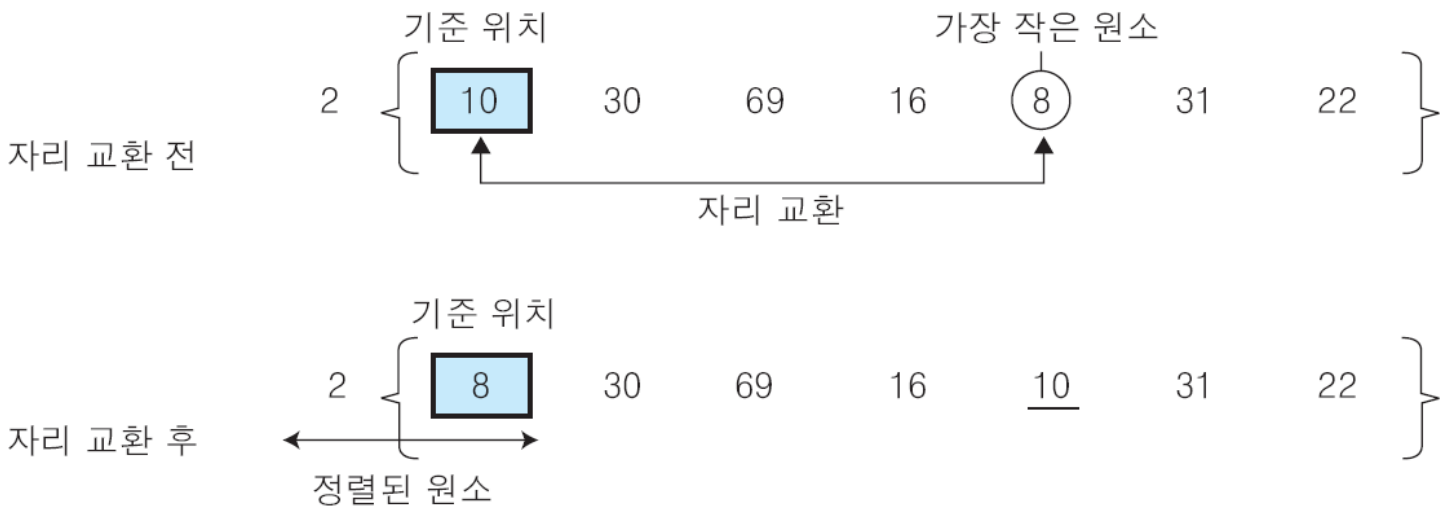
- **정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 선택 정렬 방법으로 정렬하는 과정을 살펴보자.**

① 첫 번째 자리를 기준 위치로 정하고, 전체 원소 중에서 가장 작은 원소 2를 선택하여 기준 위치에 있는 원소 69와 자리 교환



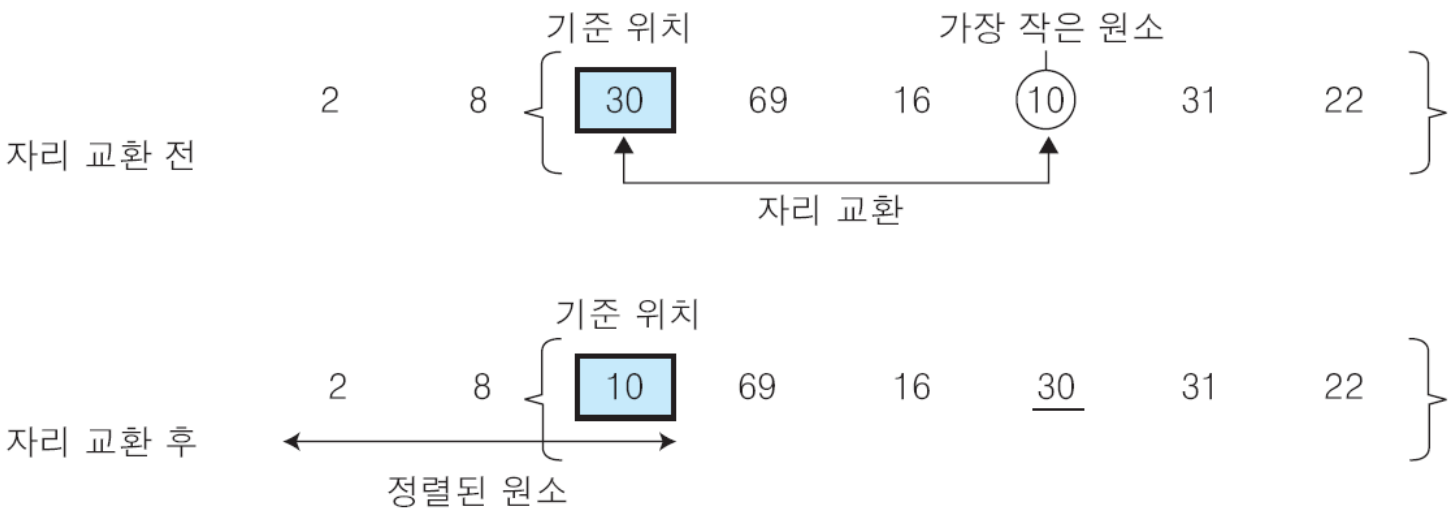
❑ 선택 정렬

② 두 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 8을 선택하여 기준 위치에 있는 원소 10과 자리 교환



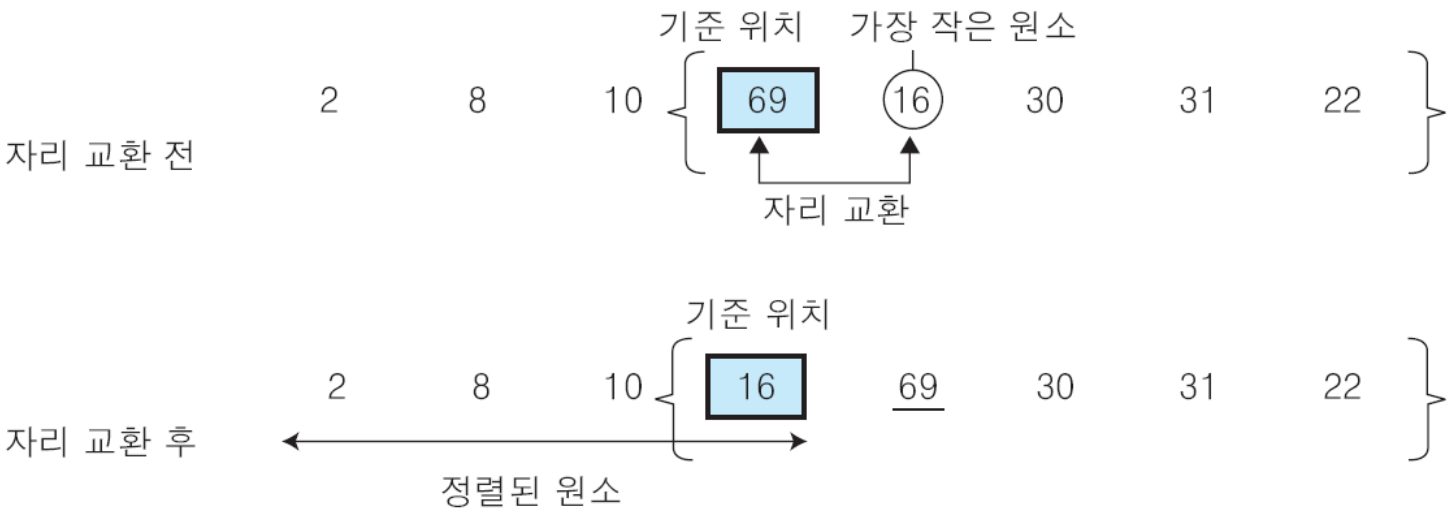
❑ 선택 정렬

③ 세 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 10을 선택하여 기준 위치에 있는 원소 30과 자리 교환



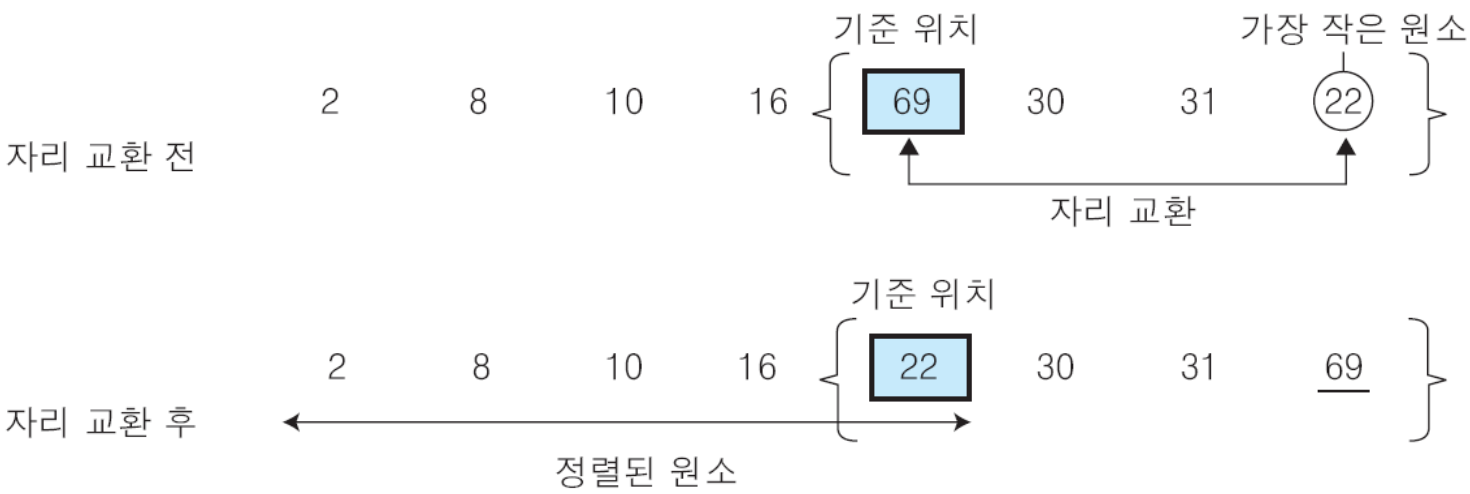
❑ 선택 정렬

④ 네 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 16을 선택하여 기준 위치에 있는 원소 69와 자리 교환



❑ 선택 정렬

⑤ 다섯 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 22를 선택하여 기준 위치에 있는 원소 69와 자리 교환



❑ 선택 정렬

⑥ 여섯 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 30을 선택하여 기준 위치에 있는 원소 30과 자리 교환 (제자리)

❑ 선택 정렬

⑦ 일곱 번째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 31을 선택하여 기준 위치에 있는 원소 31과 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

[알고리즘 11-1]

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

❖ 선택 정렬 알고리즘 분석

▪ 메모리 사용공간

- n개의 원소에 대하여 n개의 메모리 사용

▪ 비교횟수

- 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

❑ 선택 정렬

- i 단계 : i 번째 원소를 기준으로 n−i개의 원소 비교

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

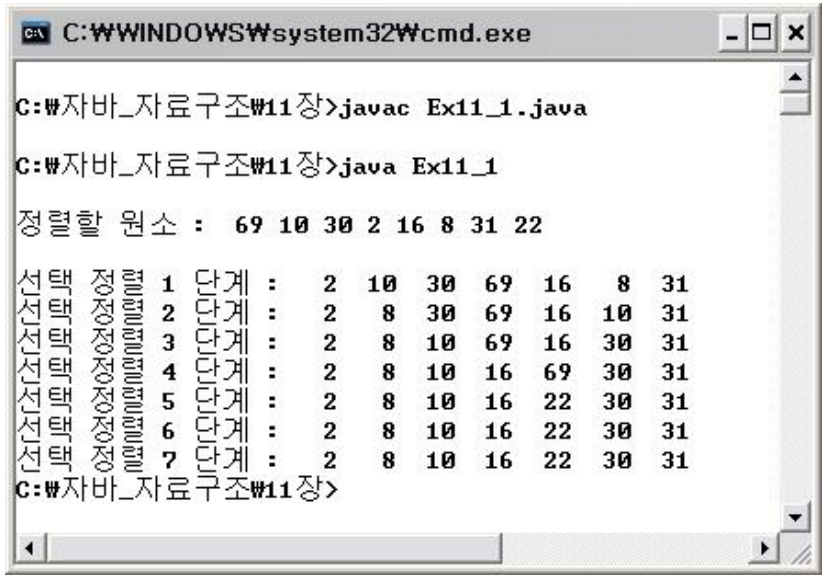
평균

- **어떤 경우에서나 비교횟수가 같으므로 시간 복잡도는 $O(n^2)$**

❖ 선택 정렬 프로그램

[예제 11-1]

❖ 실행 결과



>> 계속

❑ 버블 정렬

❖ 버블 정렬(bubble sort)

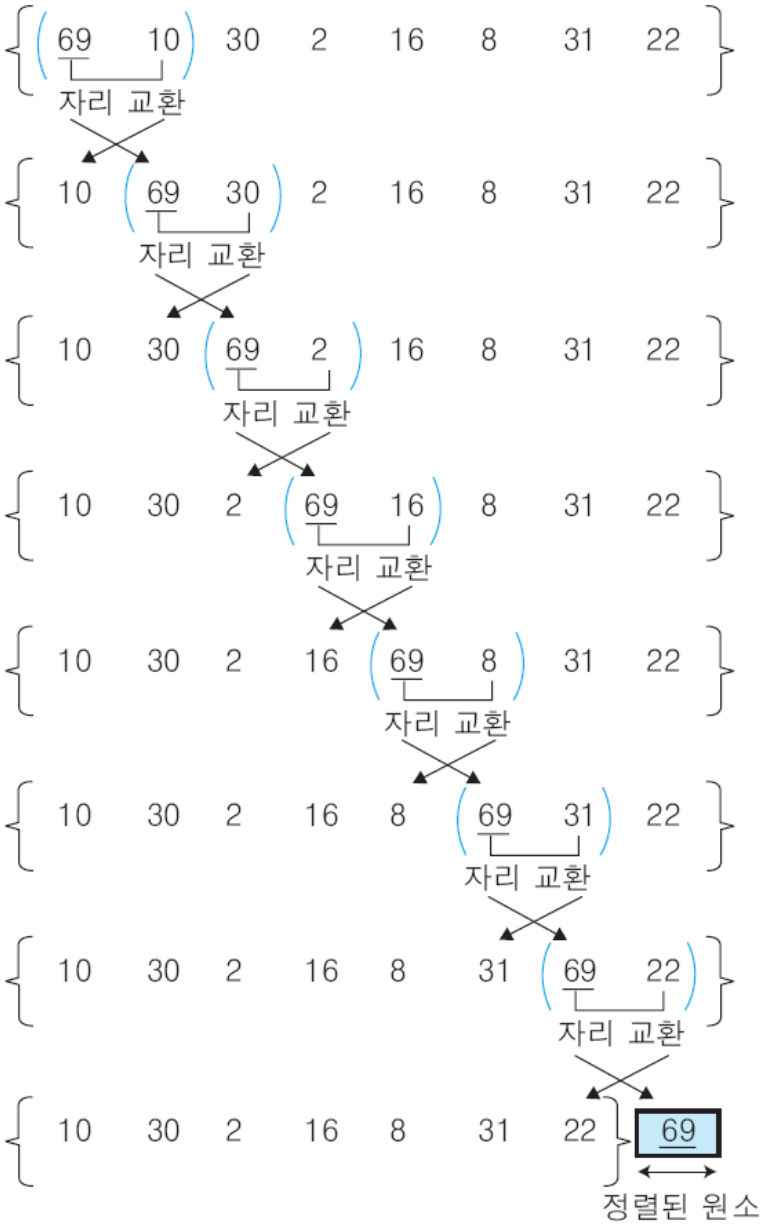
- **인접한 두 개의 원소를 비교하여 자리를 교환하는 방식**
  - 첫 번째 원소부터 마지막 원소까지 반복하여 한 단계가 끝나면 가장 큰 원소가 마지막 자리로 정렬
  - 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환하면서 맨 마지막 자리로 이동하는 모습이 물 속에서 물 위로 올라오는 물방울 모양과 같다고 하여 버블(bubble) 정렬이라 함.

❖ 버블 정렬 수행 과정

- **정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 버블 정렬 방법으로 정렬하는 과정을 살펴보자.**

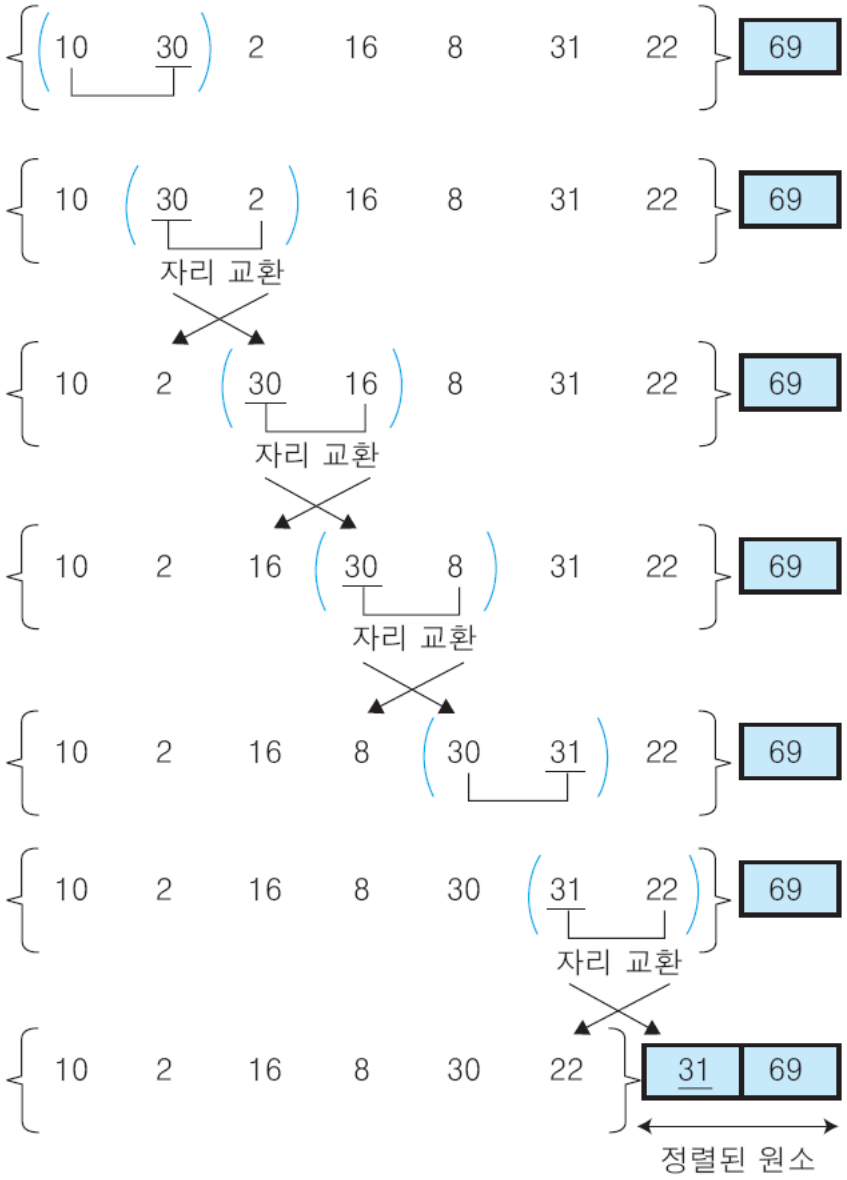
① 인접한 두 원소를 비교하여 자리를 교환하는 작업을 첫 번째 원소부터 마지막 원소까지 차례로 반복하여 가장 큰 원소 69를 마지막 자리로 정렬

❑ 버블 정렬


{ (69 10) 30 2 16 8 31 22 }
자리 교환
{ 10 (69 30) 2 16 8 31 22 }
자리 교환
{ 10 30 (69 2) 16 8 31 22 }
자리 교환
{ 10 30 2 (69 16) 8 31 22 }
자리 교환
{ 10 30 2 16 (69 8) 31 22 }
자리 교환
{ 10 30 2 16 8 (69 31) 22 }
자리 교환
{ 10 30 2 16 8 31 (69 22) }
자리 교환
{ 10 30 2 16 8 31 22 } 69
정렬된 원소

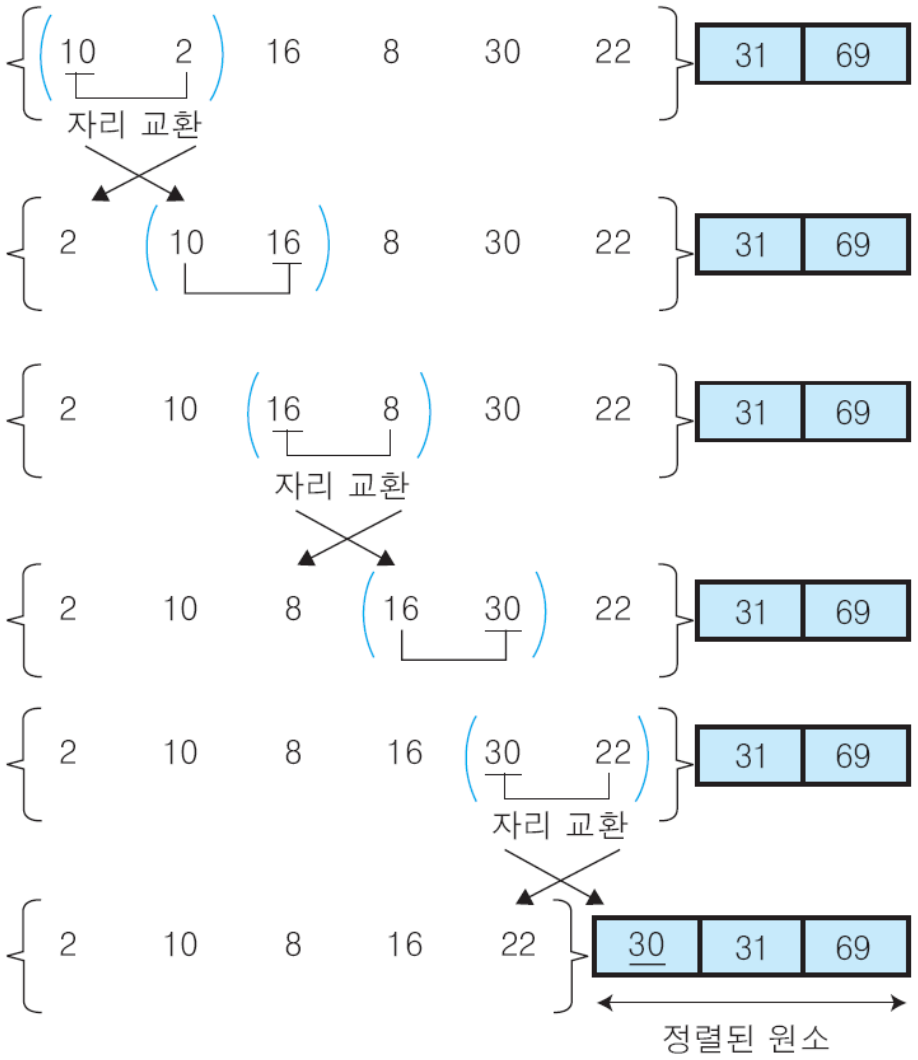
❑ 버블 정렬

② 버블 정렬을 수행하여 나머지 원소 중에서 가장 큰 원소 31을 끝에서 두 번째 자리로 정렬.



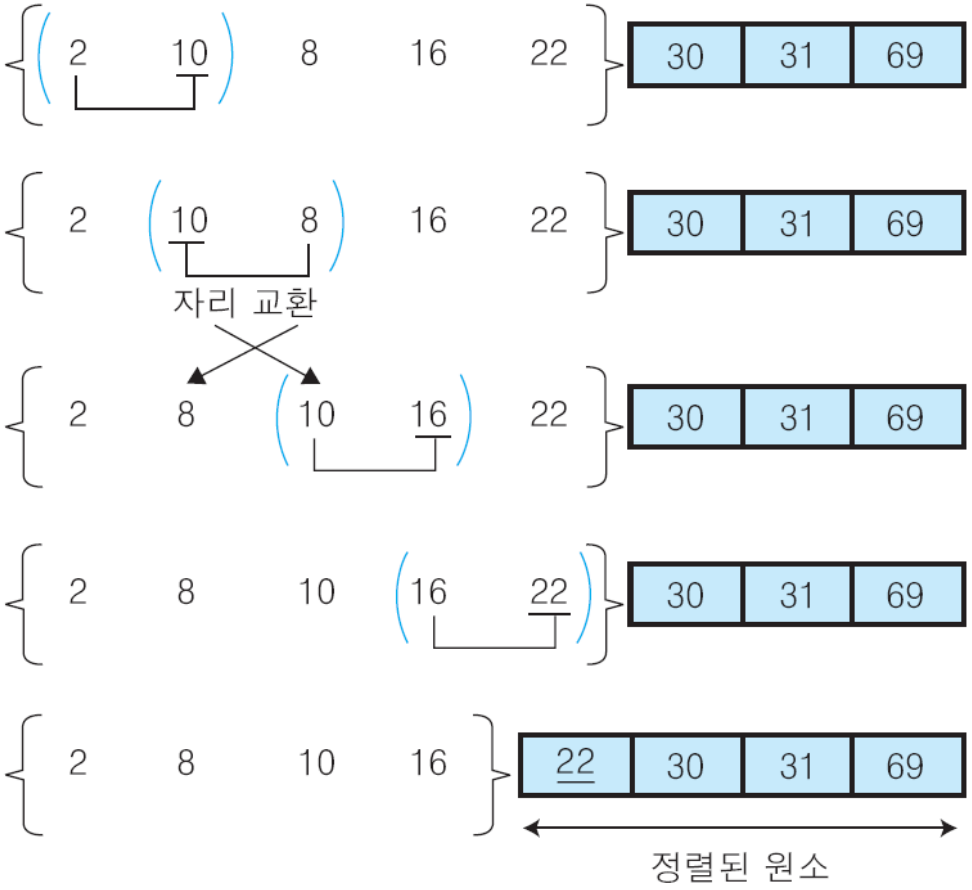
❑ 버블 정렬

③ 버블 정렬을 수행하여 나머지 원소 중에서 가장 큰 원소 30을 끝에서 세 번째 자리로 정렬.



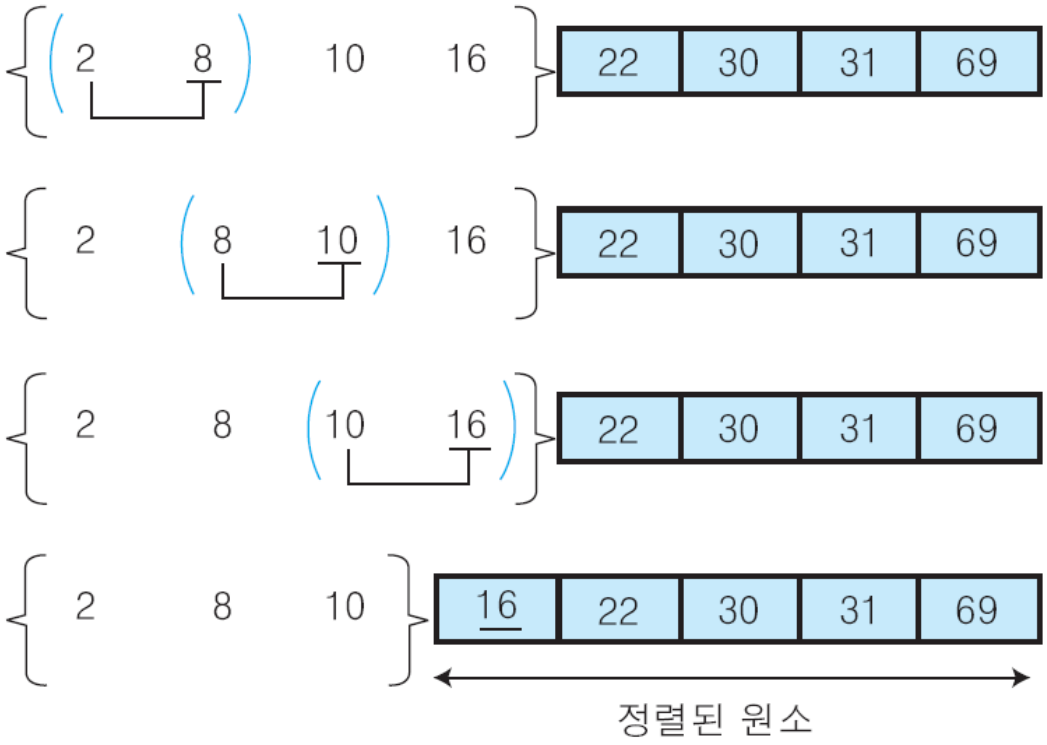
❑ 버블 정렬

④ 버블 정렬을 수행하여 나머지 원소 중에서 가장 큰 원소 22를 끝에서 네 번째 자리로 정렬.



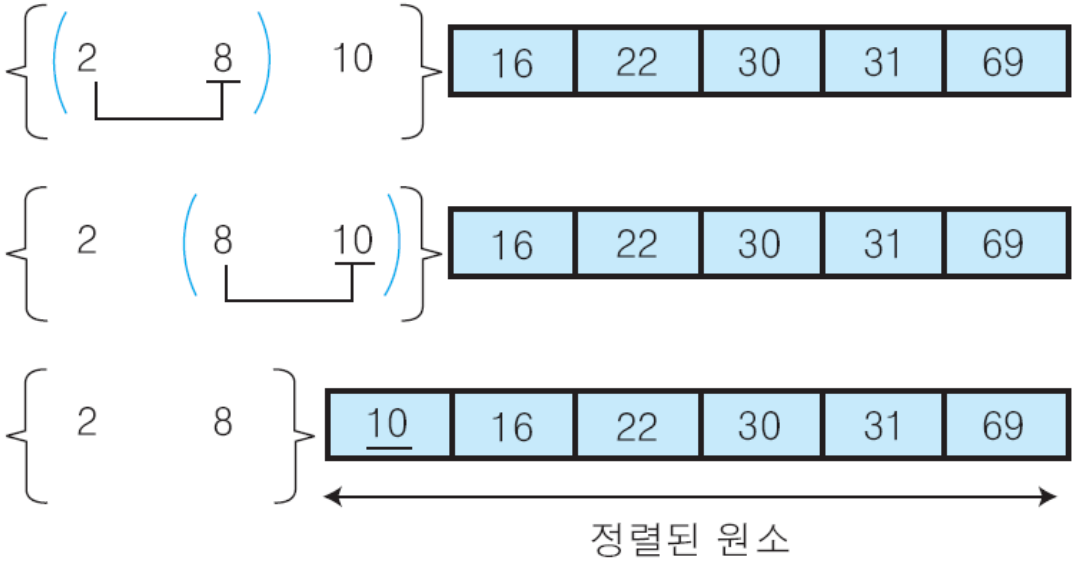
❑ 버블 정렬

⑤ 버블 정렬을 수행하여 나머지 원소 중에서 가장 큰 원소 16을 끝에서 다섯 번째 자리로 정렬.



❑ 버블 정렬

⑥ 버블 정렬을 수행하여 나머지 원소 중에서 가장 큰 원소 10을 끝에서 여섯 번째 자리로 정렬.



❑ 버블 정렬

⑦ 버블 정렬을 수행하여 나머지 원소 중에서 가장 큰 원소 8을 끝에서 일곱 번째 자리로 정렬.

마지막에 남은 첫 번째 원소는 전체 원소 중에서 가장 작은 원소로 이미 정렬된 상태이므로 실행을 종료하고 버블 정렬이 완성된다.

❑ 버블 정렬

❖ 버블 정렬 알고리즘

[알고리즘 11-2]

```
bubbleSort(a[],n)
    for (i←n-1; i≥0; i←i-1) {
        for (j←0; j<i; j←j+1) {
            if (a[j]>a[j+1]) then {
                temp ← a[j];
                a[j] ← a[j+1];
                a[j+1] ← temp;
            }
        }
    }
end bubbleSort()
```

❑ 버블 정렬

❖ 버블 정렬 알고리즘 분석

▪ 메모리 사용공간

- n개의 원소에 대하여 n개의 메모리 사용

▪ 연산 시간

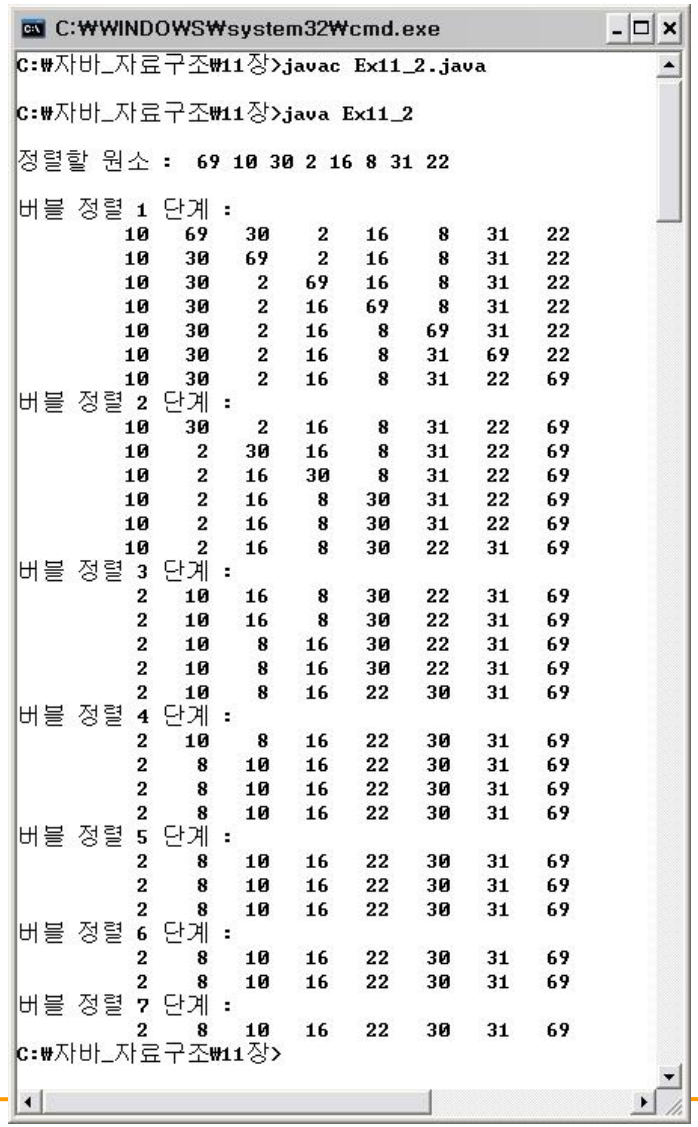
- 최선의 경우 : 자료가 이미 정렬되어있는 경우
  - ➢ **비교횟수 : i번째 원소를 (n-i)번 비교하므로, n(n-1)/2 번**
  - ➢ **자리교환횟수 : 자리교환이 발생하지 않는다.**
- 최악의 경우 : 자료가 역순으로 정렬되어있는 경우
  - ➢ **비교횟수 : i번째 원소를 (n-i)번 비교하므로, n(n-1)/2 번**
  - ➢ **자리교환횟수 : i번째 원소를 (n-i)번 교환하므로, n(n-1)/2 번**
- 평균 시간 복잡도 : $O(n^2)$

❑ 버블 정렬

❖ 버블 정렬 프로그램

[예제 11-2]

❖ 실행 결과



>> 계속

❑ 퀵 정렬

❖ 퀵 정렬(quick sort)

- **정렬할 전체 원소에 대해서 정렬을 수행하지 않고, 기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로 분할하여 정렬하는 방법**
  - 왼쪽 부분 집합에는 기준 값보다 작은 원소들을 이동시키고, 오른쪽 부분 집합에는 기준 값보다 큰 원소들을 이동시킨다.
  - 기준 값 : 피봇(pivot)
    - ➢ 일반적으로 전체 원소 중에서 가운데에 위치한 원소를 선택
- **퀵 정렬은 다음의 두 가지 기본 작업을 반복 수행하여 완성한다.**
  - 분할(divide)
    - ➢ 정렬할 자료들을 기준값을 중심으로 2개의 부분 집합으로 분할하기
  - 정복(conquer)
    - ➢ 부분 집합의 원소들 중에서 기준값보다 작은 원소들은 왼쪽 부분 집합으로, 기준값보다 큰 원소들은 오른쪽 부분집합으로 정렬하기.
    - ➢ 부분 집합의 크기가 1 이하로 충분히 작지 않으면 순환호출을 이용하여 다시 분할.

❑ 퀵 정렬

▪ 퀵 정렬 수행 방법

- 부분 집합으로 분할하기 위해서 L과 R을 사용

  ① 왼쪽 끝에서 오른쪽으로 움직이면서 크기를 비교하여 피봇 보다 크거나 같은 원소를 찾아 L로 표시

  ② 오른쪽 끝에서 왼쪽으로 움직이면서 피봇 보다 작은 원소를 찾아 R로 표시

  ③ L이 가리키는 원소와 R이 가리키는 원소를 서로 교환한다.

- L와 R이 만나게 되면 피봇과 R의 원소를 서로 교환하고, 교환한 위치를 피봇의 위치로 확정한다.

- 피봇의 확정된 위치를 기준으로 만들어진 왼쪽 부분 집합과 오른쪽 부분 집합에 대해서 퀵 정렬을 순환적으로 반복 수행하는데 부분 집합의 크기가 1 이하가 되면 퀵 정렬을 종료한다.

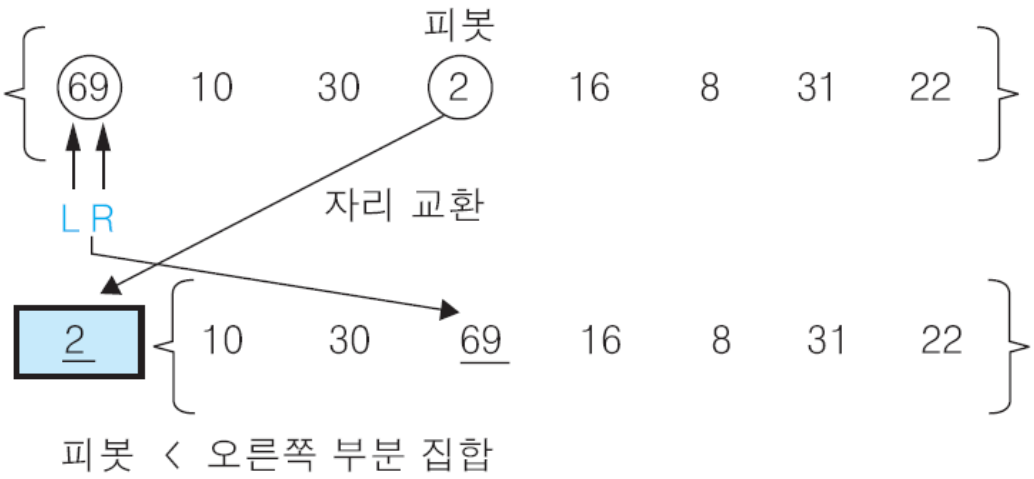
❖ 퀵 정렬 수행 과정

- **정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 퀵 정렬 방법으로 정렬하는 과정을 살펴보자.**

① 원소의 개수가 8개이므로 네 번째 자리에 있는 원소 2를 첫 번째 피봇으로 선택하고 퀵 정렬 시작.

❑ 퀵 정렬

- L이 오른쪽으로 이동하면서 피봇 보다 크거나 같은 원소를 찾고, R은 왼쪽으로 이동하면서 피봇 보다 작은 원소를 찾는다.
- L은 원소 69를 찾았지만, R은 피봇 보다 작은 원소를 찾지 못한 채로 원소 69에서 L과 만나게 된다.
- L과 R이 만났으므로, 원소 69를 피봇과 교환하여 피봇 원소 2의 위치를 확정한다.

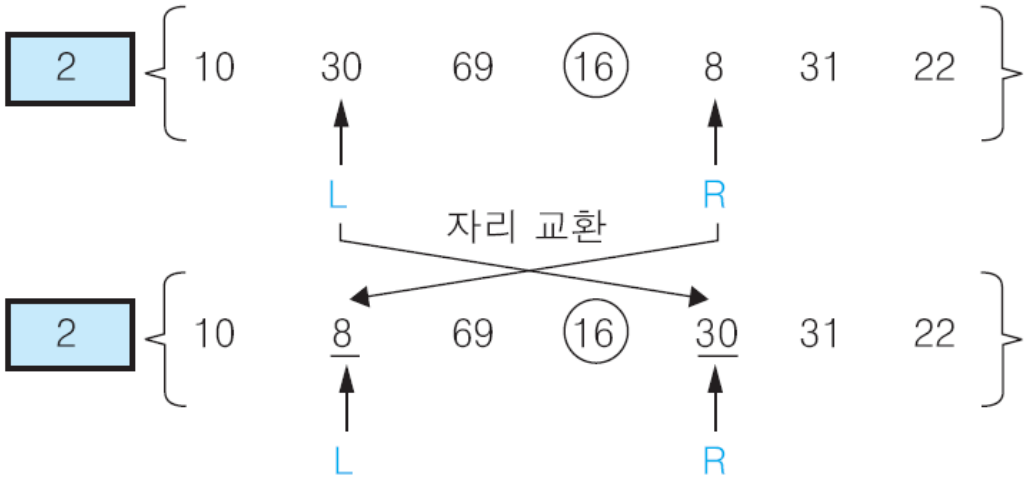


❑ 퀵 정렬

② 피봇 2의 왼쪽 부분 집합은 공집합이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합에 대해서 퀵 정렬 수행.

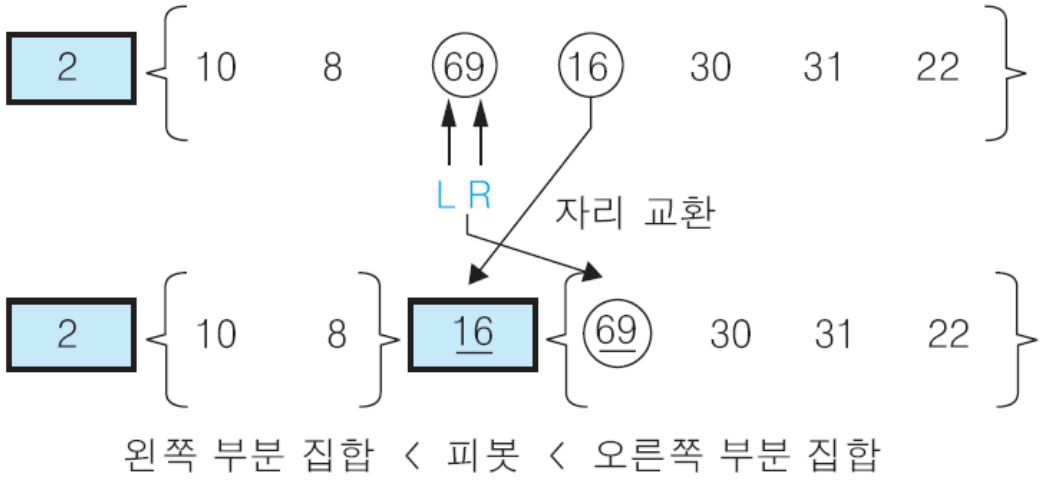
- 오른쪽 부분 집합의 원소가 7개 이므로 가운데 있는 원소 16을 피봇으로 선택

- L이 찾은 30과 R이 찾은 8을 서로 교환한다.

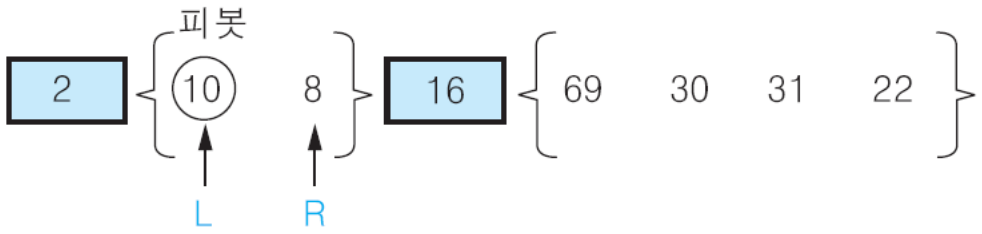


❑ 퀵 정렬

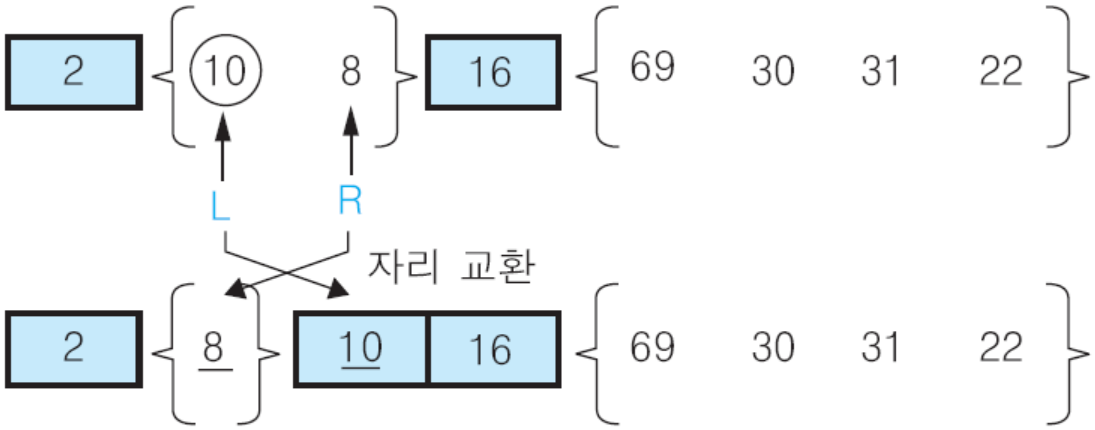
- 현재 위치에서 L과 R의 작업을 반복한다.
- L은 원소 69를 찾았지만, R은 피봇 보다 작은 원소를 찾지 못한 채로 원소 69에서 L과 만나게 된다. L과 R이 만났으므로, 원소 69를 피봇과 교환하여 피봇 원소 16의 위치를 확정한다.



③ 피봇 16의 왼쪽 부분 집합에서 원소 10을 피봇으로 선택하여 퀵 정렬 수행.



- L의 원소 10과 R의 원소 8을 교환하는데, L의 원소가 피봇이므로 피봇 원소 10의 위치가 확정된다.

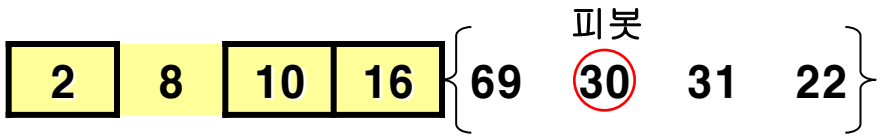


왼쪽 부분 집합 < 피봇

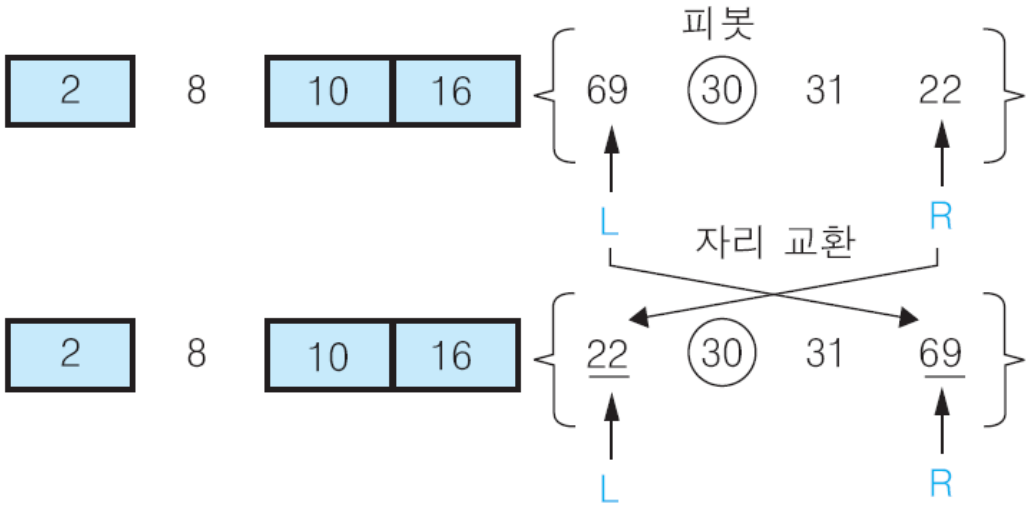
❑ 퀵 정렬

④ 피봇 10의 왼쪽 부분 집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합은 공집합이므로 역시 퀵 정렬을 수행하지 않는다.

- 이제 1단계의 피봇이었던 원소 16의 오른쪽 부분 집합에 대해 퀵 정렬 수행. 오른쪽 부분 집합의 원소가 4개이므로 원소 30을 피봇으로 선택.

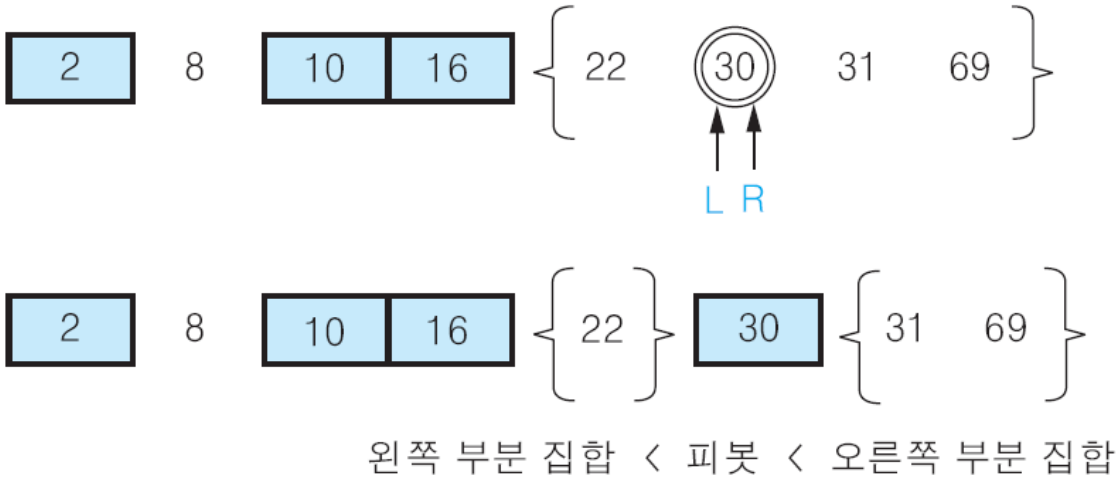


- L이 찾은 69와 R이 찾은 22를 서로 교환한다.



❑ 퀵 정렬

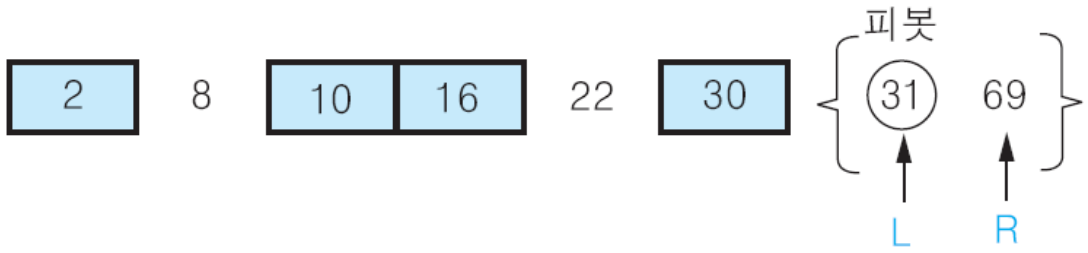
- 현재 위치에서 L과 R의 작업을 반복한다.
- L은 오른쪽으로 이동하면서 피봇 보다 크거나 같은 원소인 30을 찾고, R은 왼쪽으로 이동하면서 피봇 보다 작은 원소를 찾다가 못 찾고 원소 30에서 L과 만난다.
- L과 R이 만났으므로 피봇과 교환하는데 R의 원소가 피봇이므로 결국 제자리가 확정된다.



❑ 퀵 정렬

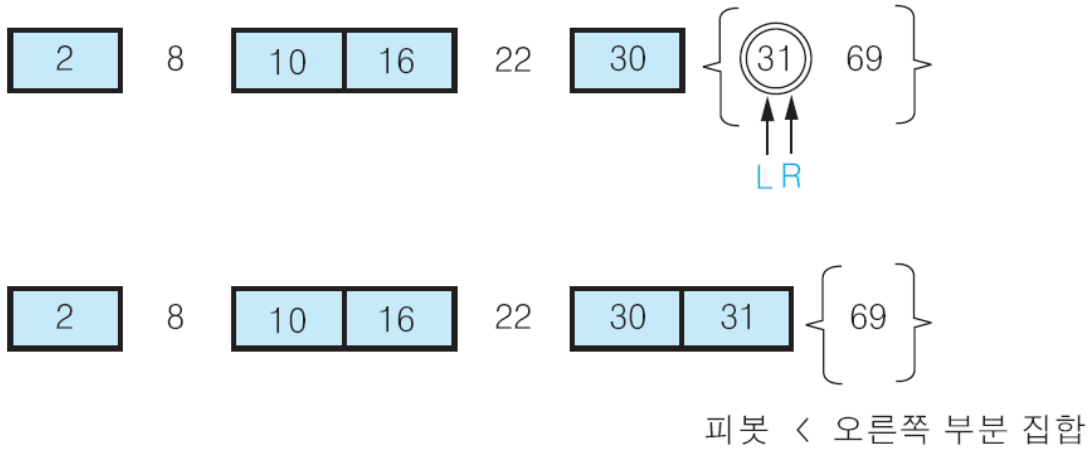
⑤ 피봇 30의 왼쪽 부분 집합의 원소가 한 개 이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분 집합에 대해서 퀵 정렬 수행.

- 오른쪽 부분 집합의 원소 2개 중에서 원소 31을 피봇으로 선택



- L은 오른쪽으로 이동하면서 원소 31을 찾고, R은 왼쪽으로 이동하면서 피봇 보다 작은 원소를 찾다가 못 찾은 채로 원소 31에서 L과 만난다.
- L과 R이 만났으므로 피봇과 교환하는데 R의 원소가 피봇이므로 결국 제자리가 확정된다.

❑ 퀵 정렬



- 피봇 31의 오른쪽 부분 집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않는다.
- 이로써 전체 퀵 정렬이 모두 완성되었다.

❖ 퀵 정렬 알고리즘

[알고리즘 11-3]

```
quickSort(a[], begin, end)
    if (m < n) then {
        p ← partition(a, begin, end);
        quickSort(a[], begin, p-1);
        quickSort(a[], p+1, end);
    }
end quickSort()
```

❖ 퀵 정렬 알고리즘의 분할 연산 알고리즘

[알고리즘 11-4]

```
partiton(a[], begin, end)
    pivot ← (begin + end)/2;
    L ← begin;
    R ← end;
    while(L<R) do {
        while(a[L]<a[pivot] and L<R) do L++;
        while(a[R]≥a[pivot] and L<R) do R--;
        if(L<R) then { // L의 원소와 R의 원소 교환
            temp ← a[L];
            a[L] ← a[R];
            a[R] ← temp;
        }
    }
    temp ← a[pivot]; // R의 원소와 피봇 원소 교환
    a[pivot] ← a[R];
    a[R] ← temp;
    return L;
end partition()
```

❖ 퀵 정렬 알고리즘 분석

▪ 메모리 사용공간

- n개의 원소에 대하여 n개의 메모리 사용

▪ 연산 시간

- 최선의 경우
  - ➢ 피봇에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 n/2개씩 이등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우

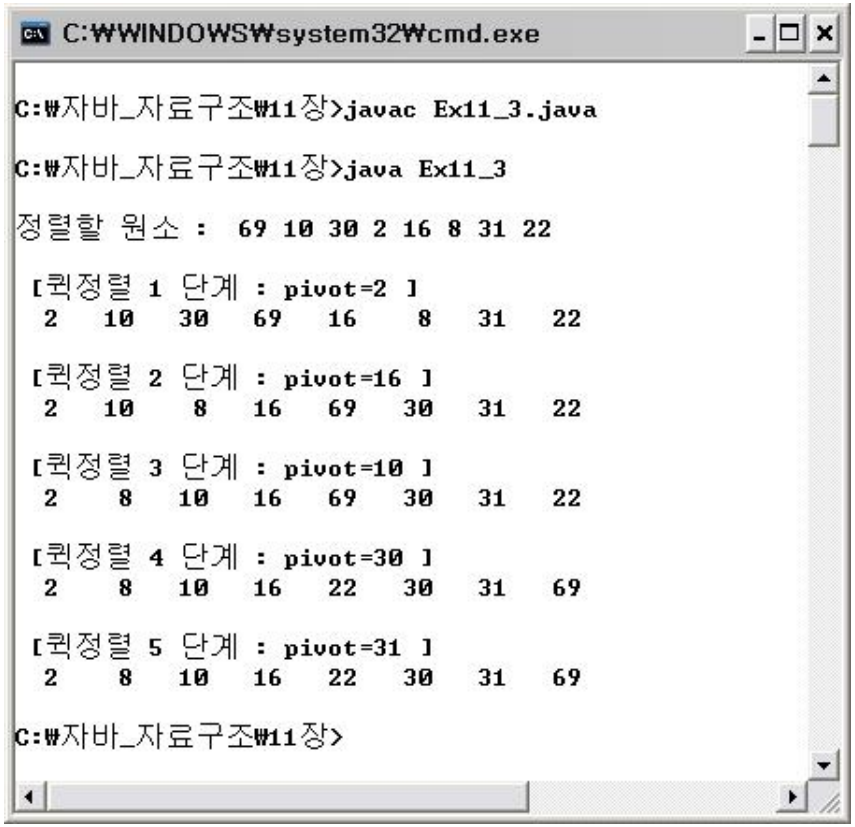
- 최악의 경우 $O(n^2)$
  - ➢ 피봇에 의해 원소들을 분할하였을 때 1개와 n-1개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우

- 평균 시간 복잡도 : $O(n \log_2 n)$ 단계의 수
  - ➢ 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법

❖ 퀵 정렬 프로그램

[예제 11-3]

❖ 실행 결과



>> 계속

❑ 삽입 정렬

❖ 삽입 정렬(insert sort)

- **정렬되어있는 부분집합에 정렬할 새로운 원소의 위치를 찾아 삽입하는 방법**
- **정렬할 자료를 두 개의 부분집합 S와 U로 가정**
  - 부분집합 S : 정렬된 앞부분의 원소들
  - 부분집합 U : 아직 정렬되지 않은 나머지 원소들
  - 정렬되지 않은 부분집합 U의 원소를 하나씩 꺼내서 이미 정렬되어있는 부분집합 S의 마지막 원소부터 비교하면서 위치를 찾아 삽입
  - 삽입 정렬을 반복하면서 부분집합 S의 원소는 하나씩 늘리고 부분집합 U의 원소는 하나씩 감소하게 한다. 부분집합 U가 공집합이 되면 삽입 정렬이 완성된다.

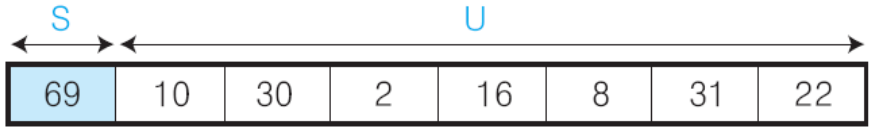
❖ 삽입 정렬 수행 과정

- **정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 삽입 정렬 방법으로 정렬하는 과정을 살펴보자.**
  - 초기 상태 : 첫 번째 원소는 정렬되어있는 부분 집합 S로 생각하고 나머지 원소들은 정렬되지 않은 원소들의 부분 집합 U로 생각한다.

S={69}, U={10, 30, 2, 16, 8, 31, 22}

초기 상태 : 첫 번째 원소는 정렬되어 있는 부분집합 S로 생각하고, 나머지 원소들은 정렬되지 않은 원소들의 부분집합 U로 생각한다.

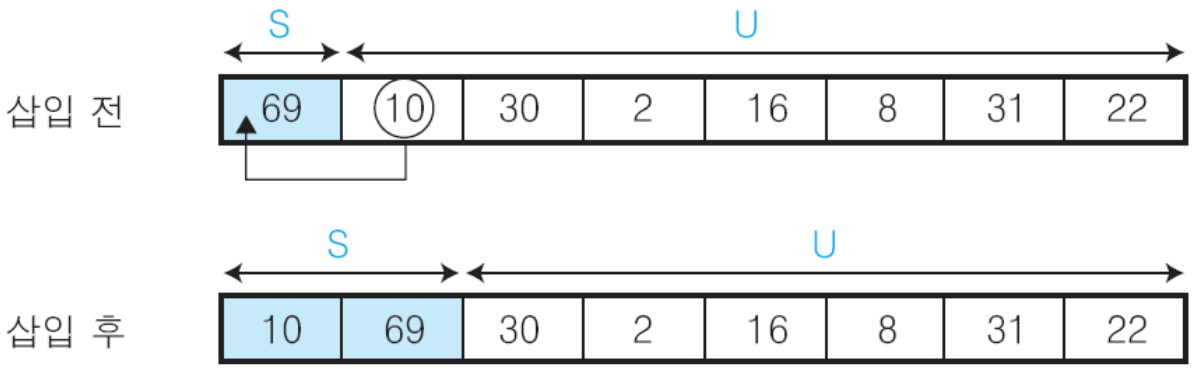
S={69}, U={10, 30, 2, 16, 8, 31, 22}



❑ 삽입 정렬

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.

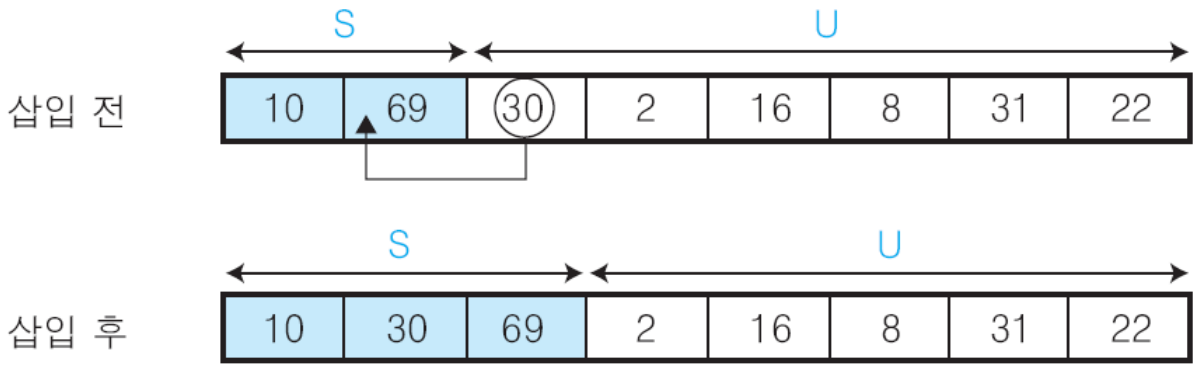
S={10, 69}, U={30, 2, 16, 8, 31, 22}



❑ 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.

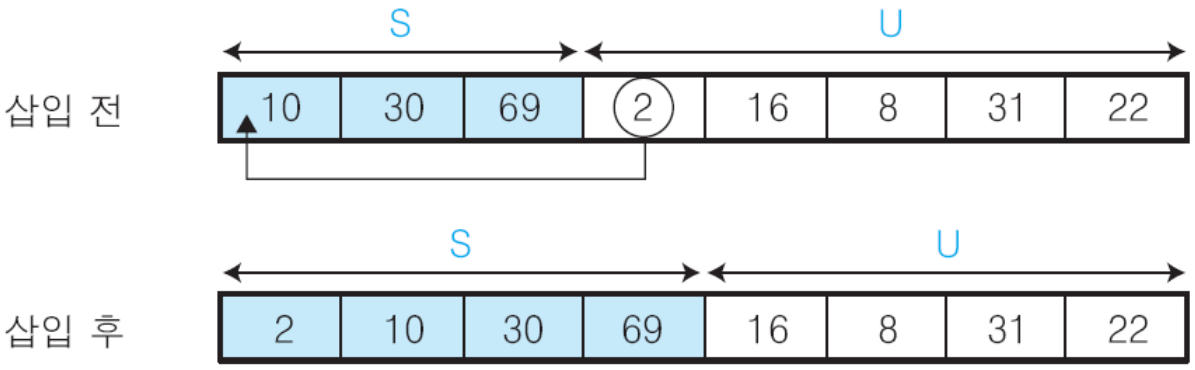
S={10, 30, 69}, U={2, 16, 8, 31, 22}



❑ 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.

S={2, 10, 30, 69}, U={16, 8, 31, 22}



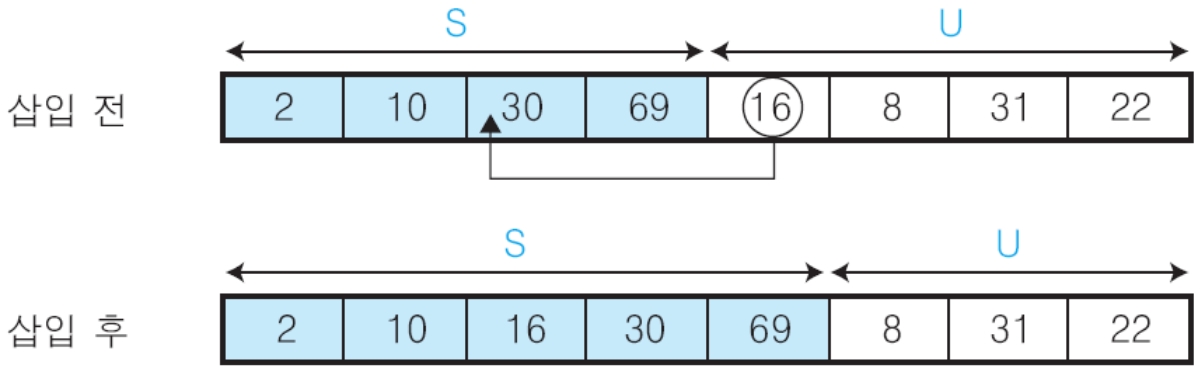
❑ 삽입 정렬

④ U의 첫 번째 원소 16을 S의 마지막 원소 69와 비교하여 (16 < 69) 이므로 그 앞자리 원소 30과 비교한다.

(16 < 30) 이므로 다시 그 앞자리 원소 10과 비교하여,

(16 > 10)이므로 원소 10과 30 사이에 삽입한다.

S={2, 10, 16, 30, 69}, U={8, 31, 22}



❑ 삽입 정렬

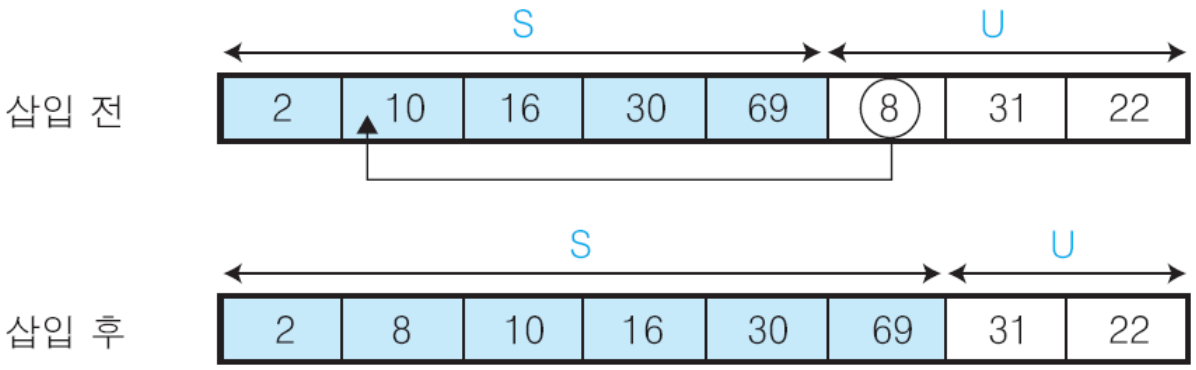
⑤ U의 첫 번째 원소 8을 S의 마지막 원소 69와 비교하여 (8 < 69) 이므로 그 앞자리 원소 30과 비교한다.

(8 < 30) 이므로 그 앞자리 원소 10과 비교하여,

(8 < 10) 이므로 다시 그 앞자리 원소 2와 비교하는데,

(8 > 2)이므로 원소 2와 10 사이에 삽입한다.

S={2, 8, 10, 16, 30, 69}, U={31, 22}

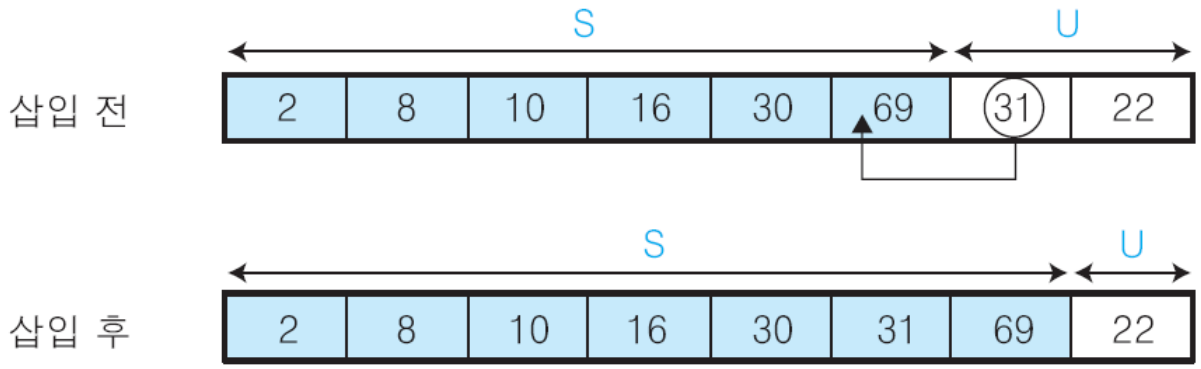


❑ 삽입 정렬

⑥ U의 첫 번째 원소 31을 S의 마지막 원소 69와 비교하여 (31 < 69) 이므로 그 앞자리 원소 30과 비교한다.

(31 > 30) 이므로 원소 30과 69 사이에 삽입한다.

S={2, 8, 10, 16, 30, 31, 69}, U={22}



❑ 삽입 정렬

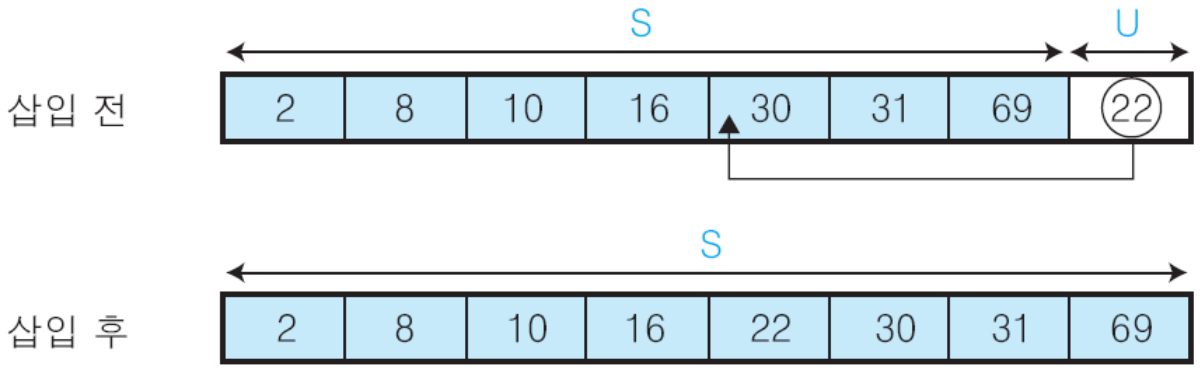
⑦ U의 첫 번째 원소 22를 S의 마지막 원소 69와 비교하여 (22 < 69) 이므로 그 앞자리 원소 31과 비교한다.

(22 < 31) 이므로 그 앞자리 원소 30과 비교하고,

(22 < 30) 이므로 다시 그 앞자리 원소 16과 비교하여,

(22 > 16) 이므로 원소 16과 30 사이에 삽입한다.

S={2, 8, 10, 16, 22, 30, 31, 69}, U={}



U가 공집합이 되었으므로 실행을 종료하고 삽입 정렬이 완성된다.

❖ 삽입 정렬 알고리즘

[알고리즘 11-5]

```
insertionSort(a[],n)
    for (i←1; i<n; i←i+1) do {
        temp ← a[i];
        j ← i;
        if (a[j-1]>temp) then move ← true;
        else move ← false;
        while (move) do {
            a[j] ← a[j-1];
            j ← j-1;
            if (j>0 and a[j-1]>temp) then move ← true;
            else move ← false;
        }
        a[j] ← temp;
    }
end insertionSort()
```

❑ 삽입 정렬

❖ 삽입 정렬 알고리즘 분석

▪ 메모리 사용공간

- n개의 원소에 대하여 n개의 메모리 사용

▪ 연산 시간

- 최선의 경우 : 원소들이 이미 정렬되어있어서 비교횟수가 최소인 경우
  - ➢ 이미 정렬되어있는 경우에는 바로 앞자리 원소와 한번만 비교한다.
  - ➢ 전체 비교횟수 = $n-1$
  - ➢ 시간 복잡도 : $O(n)$

- 최악의 경우 : 모든 원소가 역순으로 되어있어서 비교횟수가 최대인 경우
  - ➢ 전체 비교횟수 = $1+2+3+\cdots+(n-1) = n(n-1)/2$
  - ➢ 시간 복잡도 : $O(n^2)$

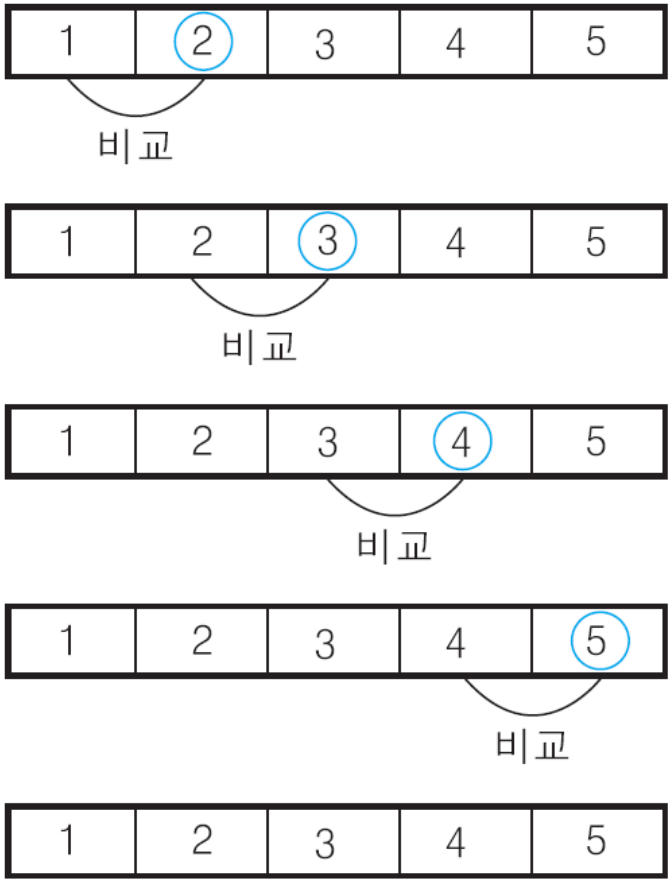
- 삽입 정렬의 평균 비교횟수 = $n(n-1)/4$

$$= \frac{\frac{n(n-1)}{2} + (n-1)}{2}$$

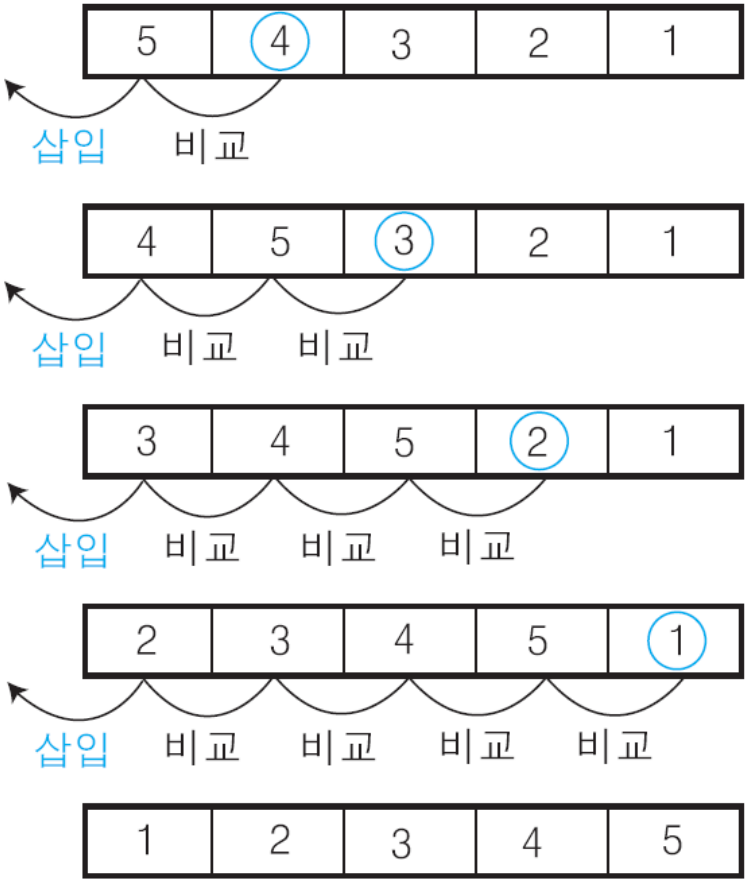
- 평균 시간 복잡도 : $O(n^2)$

❑ 삽입 정렬

- 최선의 경우:
  이미 정렬된 상태에서의 삽입



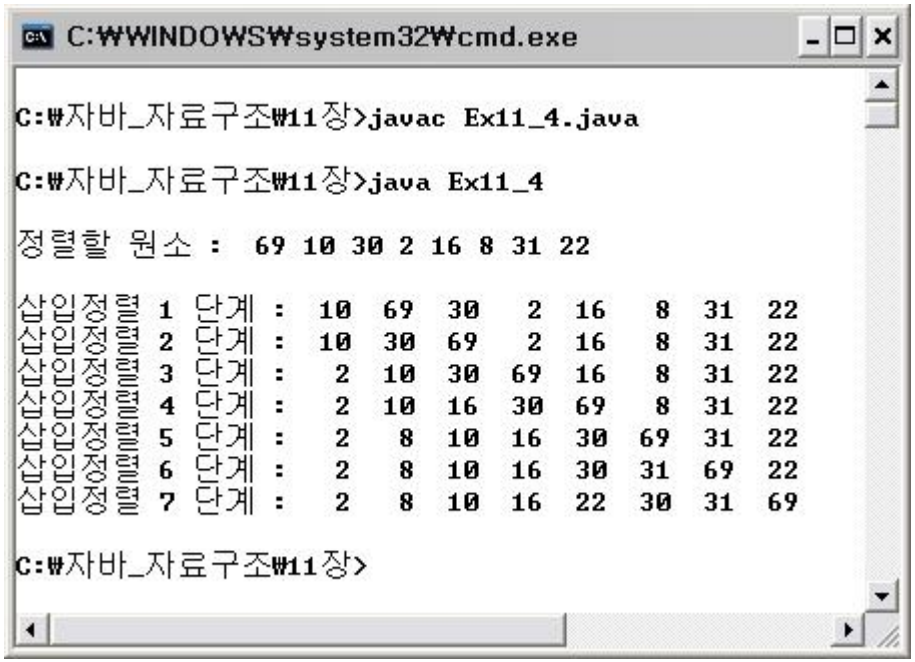
- 최악의 경우:
  역순으로 정렬된 상태에서의 삽입



❖ 삽입 정렬 프로그램

[예제 11-4]

❖ 실행 결과



>> 계속

❖ 셸 정렬(shell sort)

- **일정한 간격(interval)으로 떨어져있는 자료들끼리 부분집합을 구성하고 각 부분집합에 있는 원소들에 대해서 삽입 정렬을 수행하는 작업을 반복하면서 전체 원소들을 정렬하는 방법**
  - 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분집합으로 나누어 정렬하게 되면 비교연산과 교환연산 감소
- **셸 정렬의 부분집합**
  - 부분집합의 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 달라진다.**
  - 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

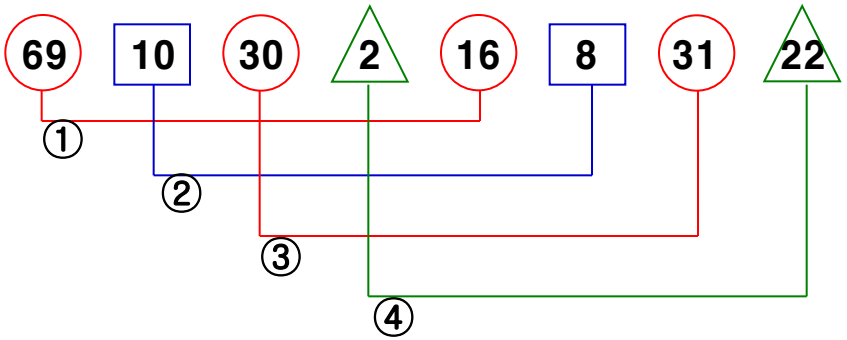
❑ 셸 정렬

❖ 셸 정렬 수행 과정

- **정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셸 정렬 방법으로 정렬하는 과정을 살펴보자.**

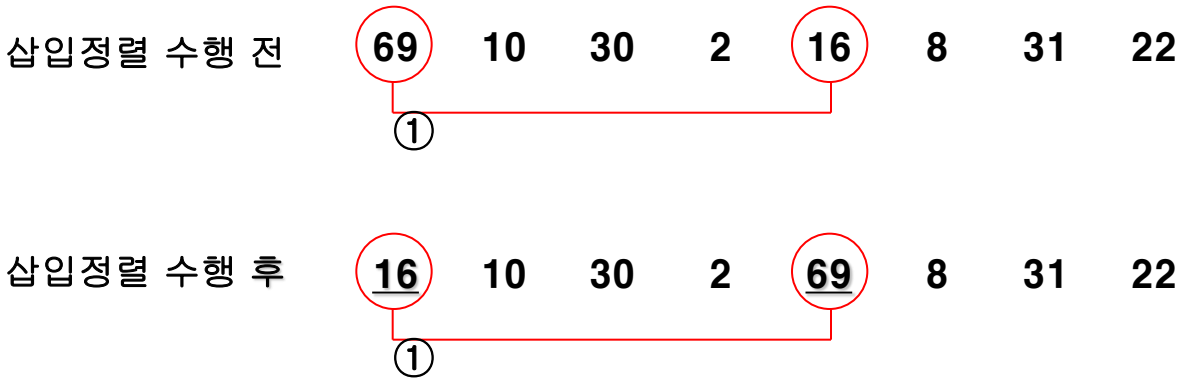
① 원소의 개수가 8개이므로 매개변수 h는 4에서 시작한다.

h=4 이므로 간격이 4인 원소들을 같은 부분 집합으로 만들면 4개의 부분 집합이 만들어진다.

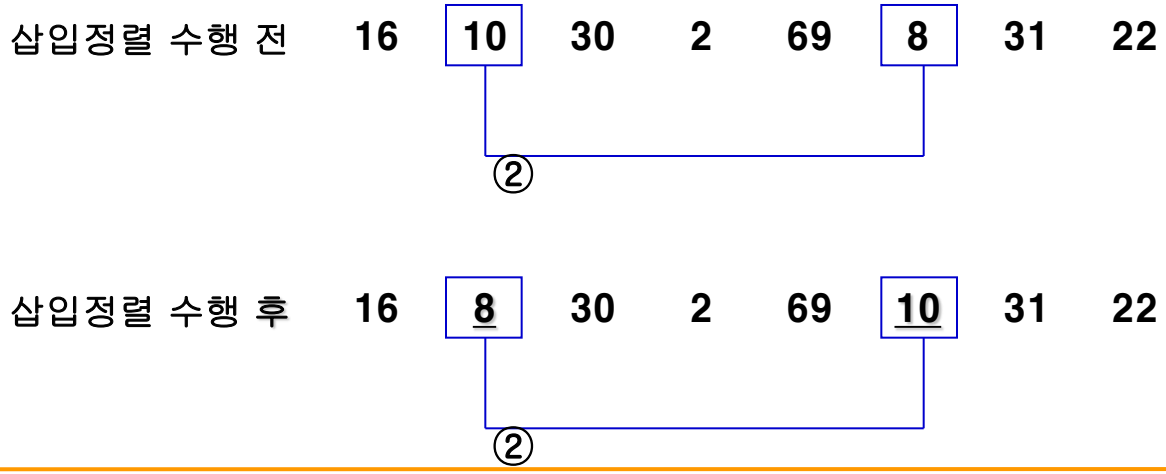


❑ 셸 정렬

- 첫 번째 부분 집합 {69, 16}에 대해서 삽입 정렬을 수행하여 정렬.

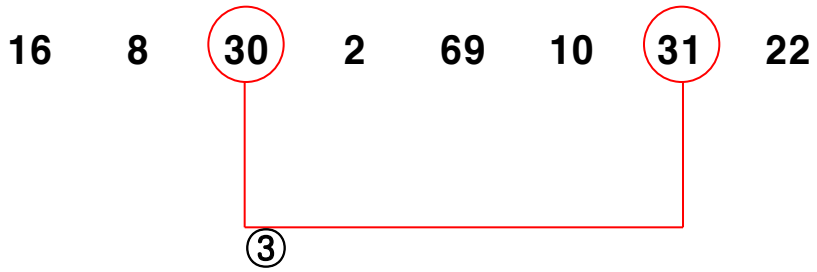


- 두 번째 부분 집합 {10, 8}에 대해서 삽입 정렬 수행.

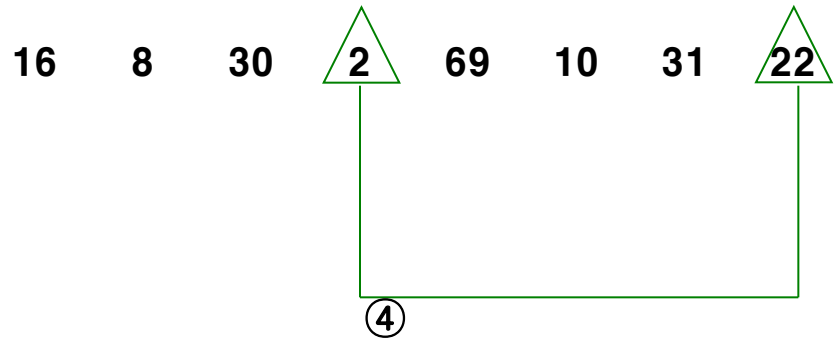


❑ 셸 정렬

- 세 번째 부분 집합 {30, 31}에 대해서 삽입 정렬을 수행하는데, (30<31) 이므로 자리 교환은 이루어지지 않는다.



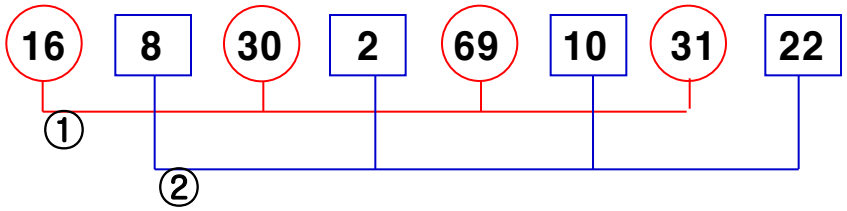
- 네 번째 부분 집합 {2, 22}에 대해서 삽입 정렬을 수행하는데, (2<22) 이므로 자리 교환은 이루어지지 않는다.



❑ 셸 정렬

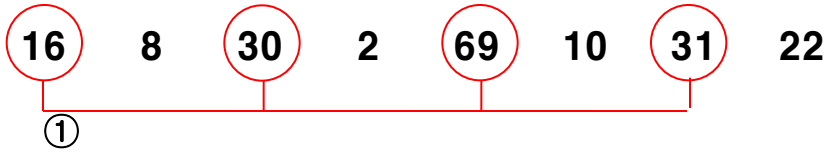
② 이제 h를 2로 변경하고 다시 셸 정렬 시작.

h=2 이므로 간격이 2인 원소들을 같은 부분 집합으로 만들면 2개의 부분 집합이 만들어진다.

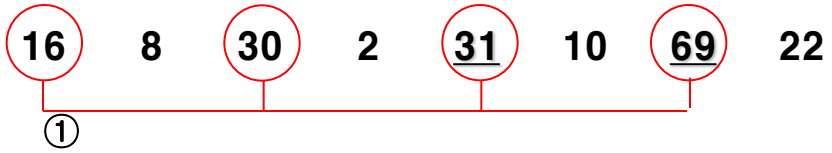


- 첫 번째 부분 집합 {16, 30, 69, 31}에 대해서 삽입 정렬을 수행하여 정렬

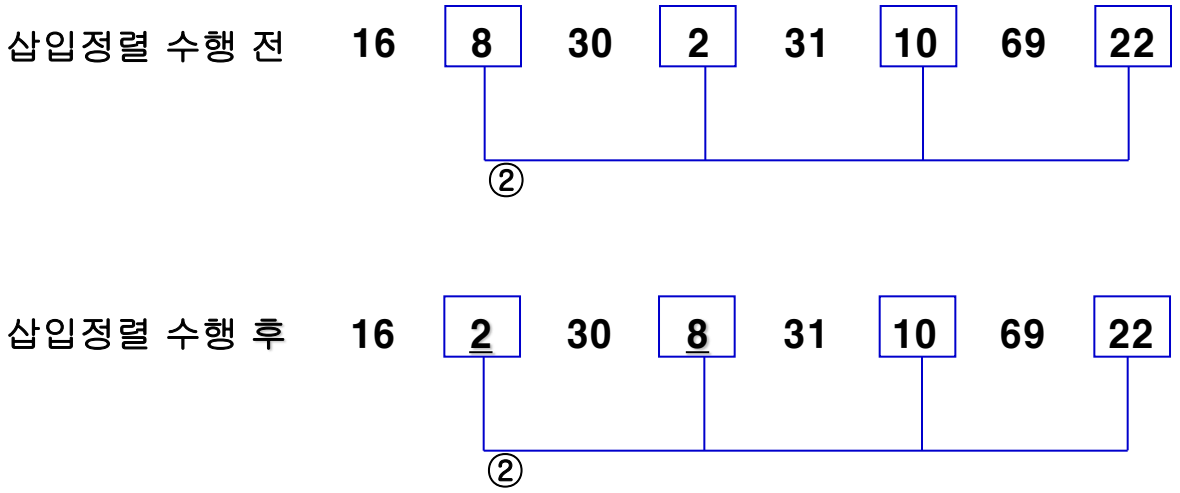
삽입정렬 수행 전



삽입정렬 수행 후



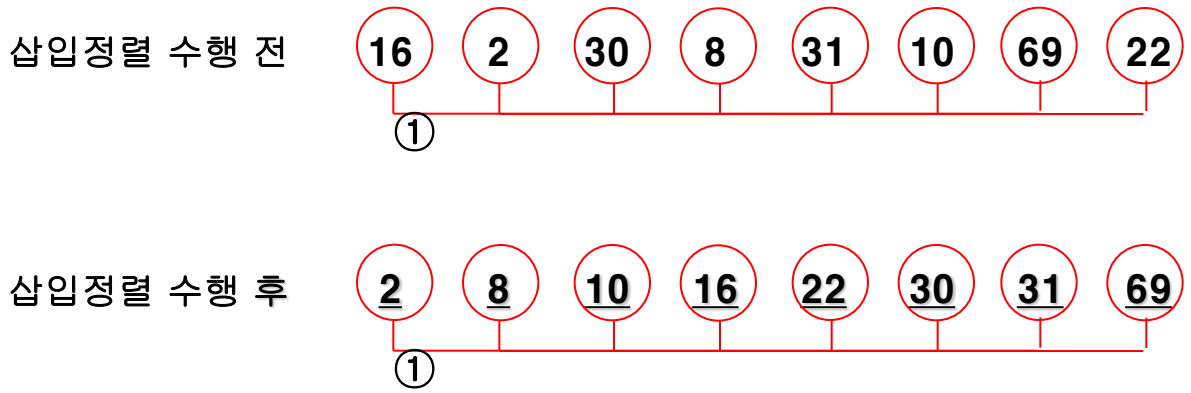
- 두 번째 부분 집합 {8, 2, 10, 22}에 대해서 삽입 정렬을 수행하여 정렬.



③ 이제 h를 1로 변경하고 다시 셸 정렬 시작.

h=1 이므로 간격이 1인 원소들을 같은 부분 집합으로 만들면 1개의 부분 집합이 만들어진다.

즉, 전체 원소에 대해서 삽입 정렬을 수행하고 셸 정렬이 완성된다.



❖ 셸 정렬 알고리즘

[알고리즘 11-6]

```
shellSort(a[],n)
    interval ← n;
    while (interval ≥ 1) do {
        interval ← interval/2;
        for (i←0; i<interval; i←i+1) do {
            intervalSort(a[], i, n, interval);
        }
    }
end shellSort()
```

❖ 셸 정렬 알고리즘 분석

▪ 메모리 사용공간

- n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장공간 사용

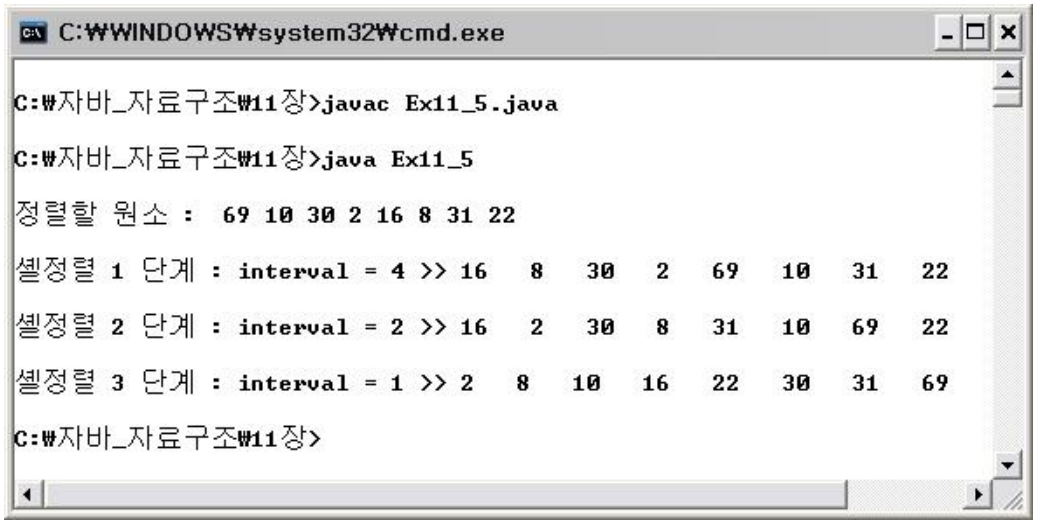
▪ 연산 시간

- 비교횟수
  - ➢ 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

❖ 셸 정렬 프로그램

[예제 11-5]

❖ 실행 결과



>> 계속

❑ 병합 정렬

❖ 병합 정렬(merge sort)

- 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법
- 부분집합으로 분할(divide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분집합들을 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

❑ 병합 정렬

- **2-way 병합 정렬 : 세 가지 기본 작업을 반복 수행하면서 완성**

⑴ 분할(divide) : 입력 자료를 같은 크기의 부분집합 2개로 분할한다.

⑵ 정복(conquer) : 부분집합의 원소들을 정렬한다. 부분집합의 크기가 충분히 작지 않으면 순환호출을 이용하여 다시 분할 정복 기법을 적용한다.

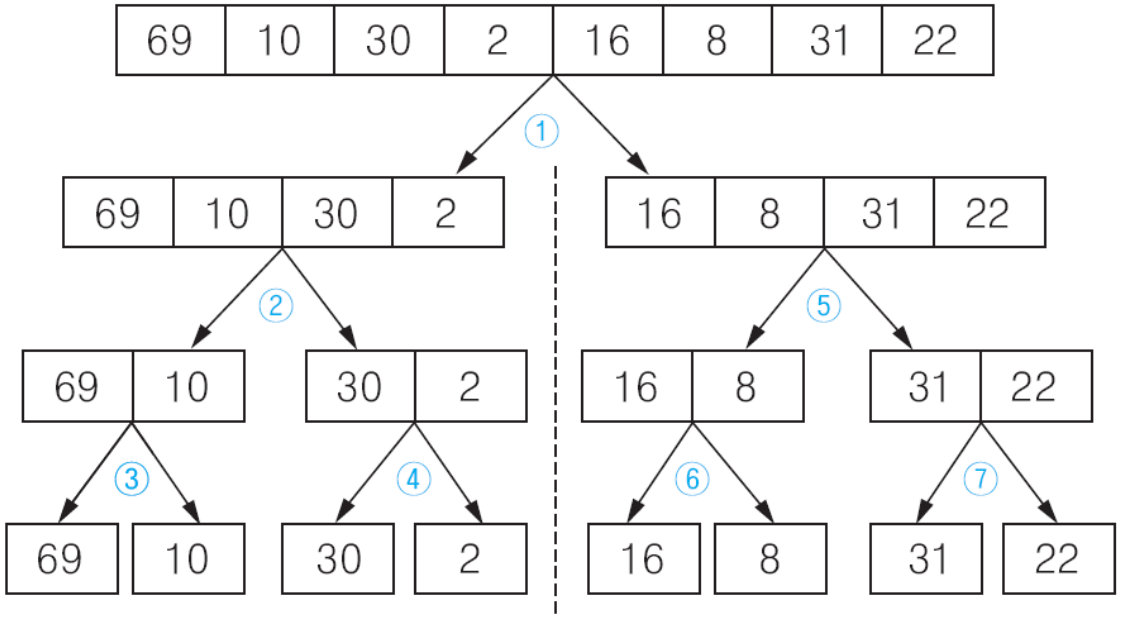
⑶ 결합(combine) : 정렬된 부분집합들을 하나의 집합으로 통합한다.

❑ 병합 정렬

❖ 병합 정렬 수행 과정

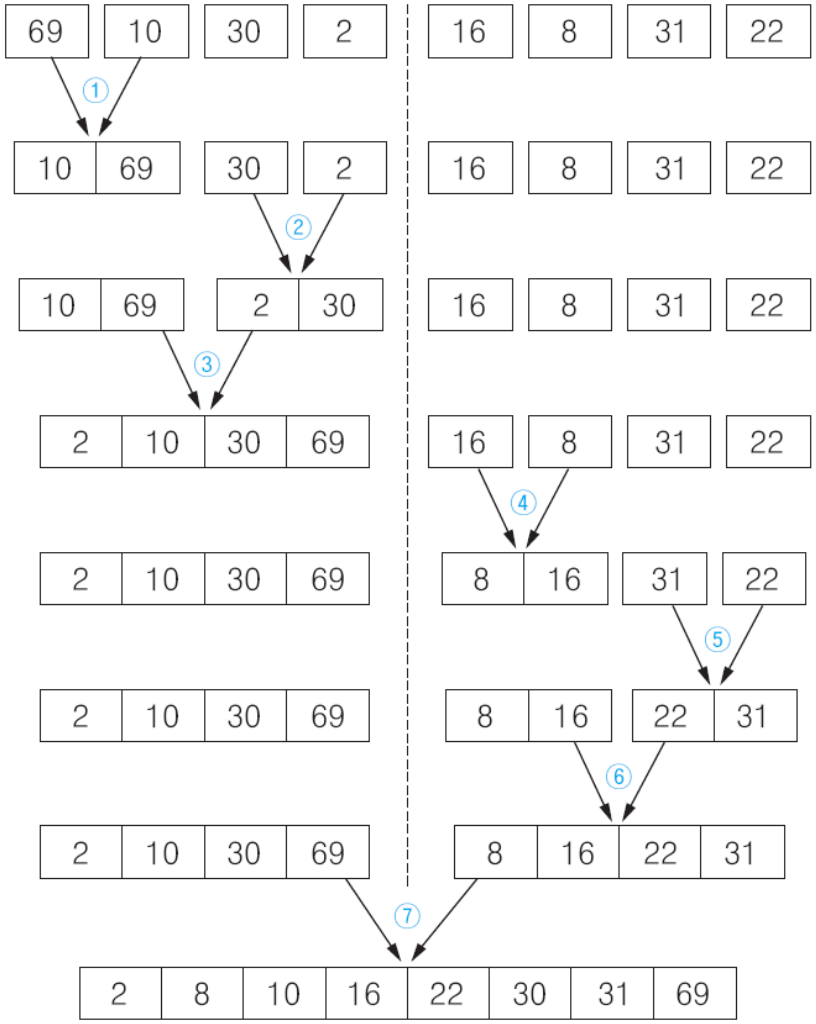
- **정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 병합 정렬 방법으로 정렬하는 과정을 살펴보자.**

① 분할 단계 : 정렬할 전체 자료의 집합에 대해서 최소 원소의 부분집합이 될 때까지 분할작업을 반복하여 1개의 원소를 가진 부분집합 8개를 만든다.



❑ 병합 정렬

② 병합단계 : 2개의 부분집합을 정렬하면서 하나의 집합으로 병합한다.
8개의 부분집합이 1개로 병합될 때까지 반복한다.



❖ 병합 정렬 알고리즘

[알고리즘 11-7]

```
mergeSort(a[],m,n)
    if (a[m:n]의 원소수 > 1) then {
        전체 집합을 두개의 부분집합으로 분할;
        mergeSort(a[],m,middle);
        mergeSort(a[],middle+1,n);
        merge(a[m:middle],a[middle+1:n]);
    }
end mergeSort()
```

❑ 병합 정렬

❖ 병합 정렬 알고리즘 분석

▪ 메모리 사용공간

- 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
- 원소 n개에 대해서 (2 x n)개의 메모리 공간 사용

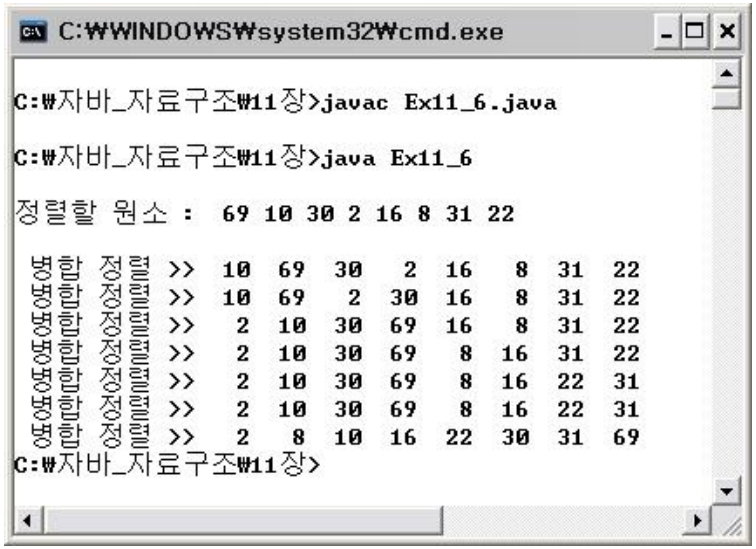
▪ 연산 시간

- 분할 단계 : n개의 원소를 분할하기 위해서 $\log_2 n$번의 단계 수행
- 병합 단계 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n번의 비교연산 수행
- 전체 병합 정렬의 시간 복잡도 : $O(n \log_2 n)$

❖ 병합 정렬 프로그램

[예제 11-8]

❖ 실행 결과



>> 계속

Thank You !