

4. 정렬

한국외국어대학교
고 석 훈

목차

4.1 정렬

4.2 선택 정렬

4.4 삽입 정렬

4.6 병합 정렬

4.8 기수 정렬

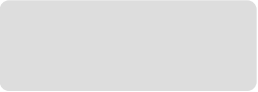
4.3 버블 정렬

4.5 쉘 정렬

4.7 퀵 정렬

4.9 트리 정렬

4.1 정렬(Sort)

- 정렬(sort)
 - 2개 이상의 자료를 오름차순(ascending)이나 내림차순(descending)으로 재배열하는 것
- 
 - 자료를 정렬하는데 사용하는 기준이 되는 특정 값

정렬 방법의 분류

- 정렬 방법의 분류
 - 실행 방법에 따른 분류: 비교식/분산식
 - 정렬 장소에 따른 분류: 내부 정렬/외부 정렬
- 비교식 정렬(comparative sort)
 - 비교하고자 하는 각 키 값들을
한번에 두 개씩 비교하여 교환하는 방식
- 분산식 정렬(distribute sort)
 - 키 값을 기준으로 자료를 여러 개의 부분 집합으로 분해하고,
각 부분집합을 정렬함으로써 전체를 정렬하는 방식

- 내부 정렬(internal sort)

- 정렬할 자료를 메인 메모리에 올려서 정렬하는 방식
- 정렬 속도가 빠르지만 메모리 용량의 제한을 받음

- 내부 정렬 방식

- 교환 방식: 키를 비교하고 교환하는 방식(선택, 버블, 퀵 정렬)
- 삽입 방식: 키를 비교하고 삽입하는 방식(삽입 정렬, 쉘 정렬)
- 병합 방식: 키를 비교하고 병합하는 방식(2-way병합, n-way병합)
- 분배 방식: 여러 부분집합에 분배하여 정렬하는 방식(기수정렬)
- 선택 방식: 이진 트리를 사용하는 방식(힙 정렬, 트리 정렬)

- 외부 정렬(external sort)
 - 정렬할 자료를 보조 기억장치에서 정렬하는 방식
 - 대용량의 보조 기억 장치를 사용하기 때문에
내부 정렬보다 속도는 떨어지지만 대용량의 자료의 정렬 가능
- 외부 정렬 방식
 - 병합 방식: 파일을 부분 파일로 분리하여 각각을 내부 정렬하고,
병합하는 정렬 방식 (2-way 병합, n-way 병합)

정렬의 종류

- 선택 정렬(selection sort)
- 버블 정렬(bubble sort)
- 삽입 정렬(insert sort)
- 쉘 정렬(shell sort)
- 퀵 정렬(quick sort)
- 병합 정렬(merge sort)
- 기수 정렬(radix sort)
- 트리 정렬(tree sort)

4.2 선택 정렬(Selection Sort)

- 선택 정렬(selection sort)
 - 전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 자리를 교환하는 방식으로 정렬
- 수행 방법
 - 전체 원소 중 가장 작은 원소를 찾아, 첫 번째 원소와 자리를 교환
 - 그 다음 두 번째로 작은 원소를 찾아, 두 번째 원소와 자리를 교환
 - 그 다음 세 번째로 작은 원소를 찾아, 세 번째 원소와 자리를 교환
 - 이 과정을 반복하면서 정렬을 완성

선택 정렬 알고리즘

selectionSort($a[], n$) {

입력: 정수 배열 a , 배열 크기 n

출력: 오름차순으로 정렬된 배열 a

for ($i \leftarrow 0; i < n - 1; i++$) {

$\text{min} \leftarrow i;$

for ($j \leftarrow i + 1; j < n; j++$)

if ($a[j] < a[\text{min}]$)

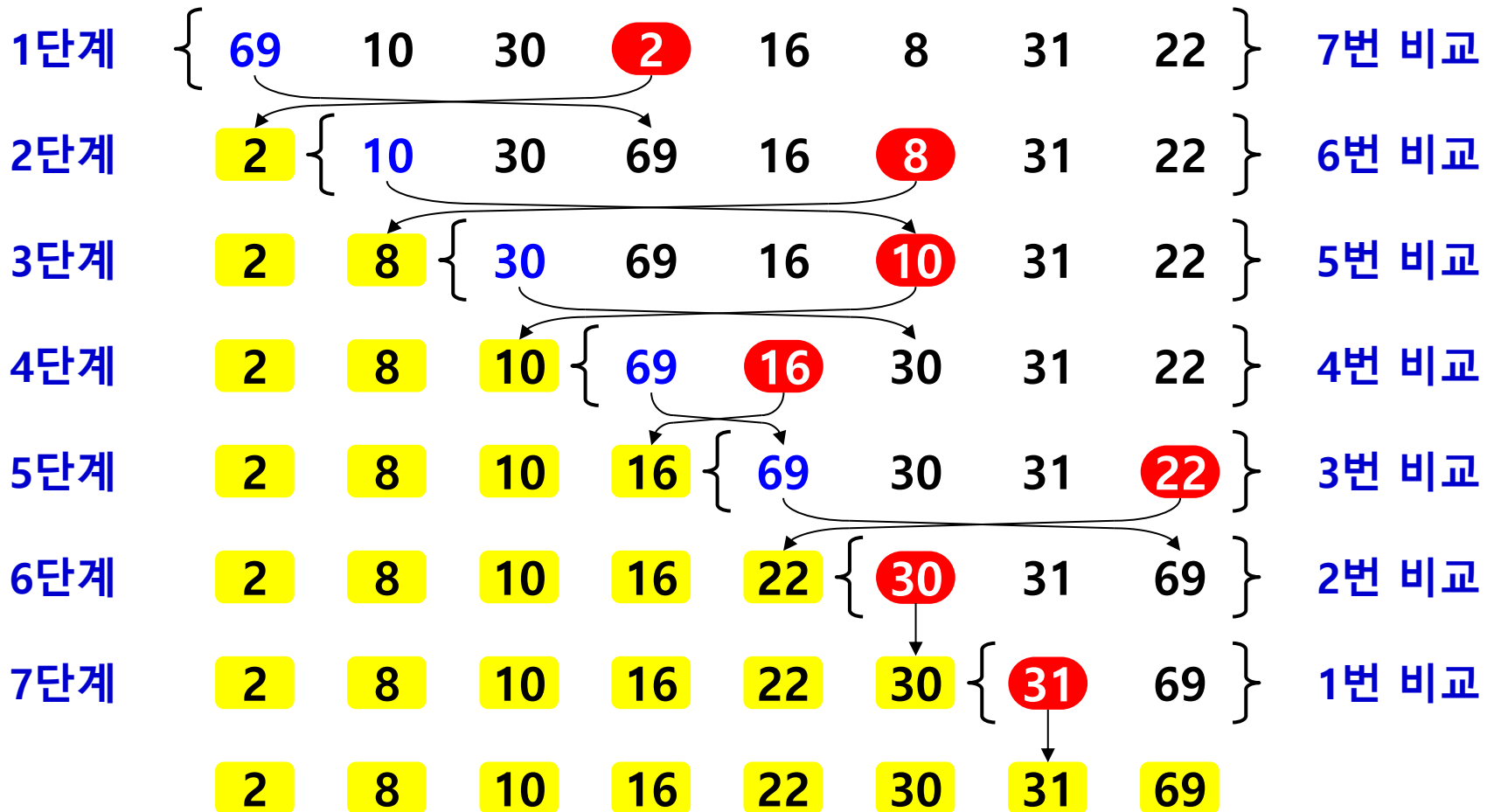
$\text{min} \leftarrow j;$

swap($a[i], a[\text{min}]$); // 두 변수의 값을 교환

 }

}

선택 정렬의 예



선택 정렬 알고리즘 분석

- 메모리 사용공간

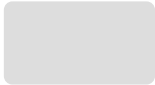
- n 개의 원소에 대하여 n 개의 메모리 사용

- 비교횟수

- 단계별 비교 회수:

1단계	$(n - 1)$ 번 비교
2단계	$(n - 2)$ 번 비교
...	...
i 단계	$(n - i)$ 번 비교
...	...
$(n - 2)$ 단계	2번 비교
$(n - 1)$ 단계	1번 비교

- 전체 비교 회수: $\sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2}$

- 시간 복잡도: 

4.3 버블 정렬(Bubble Sort)

- 버블 정렬(bubble sort)
 - 차례로 인접한 두 개의 원소를 비교하여 자리를 교환하는 방식
 - 첫 번째 원소부터 마지막 원소까지 인접 원소 비교를 반복하여 한 단계가 끝나면 가장 큰 원소를 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 부름

버블 정렬 알고리즘

bubbleSort($a[], n$) {

입력: 정수 배열 a , 배열 크기 n

출력: 오름차순으로 정렬된 배열 a

for ($i \leftarrow n - 1; i \geq 0; i --$)

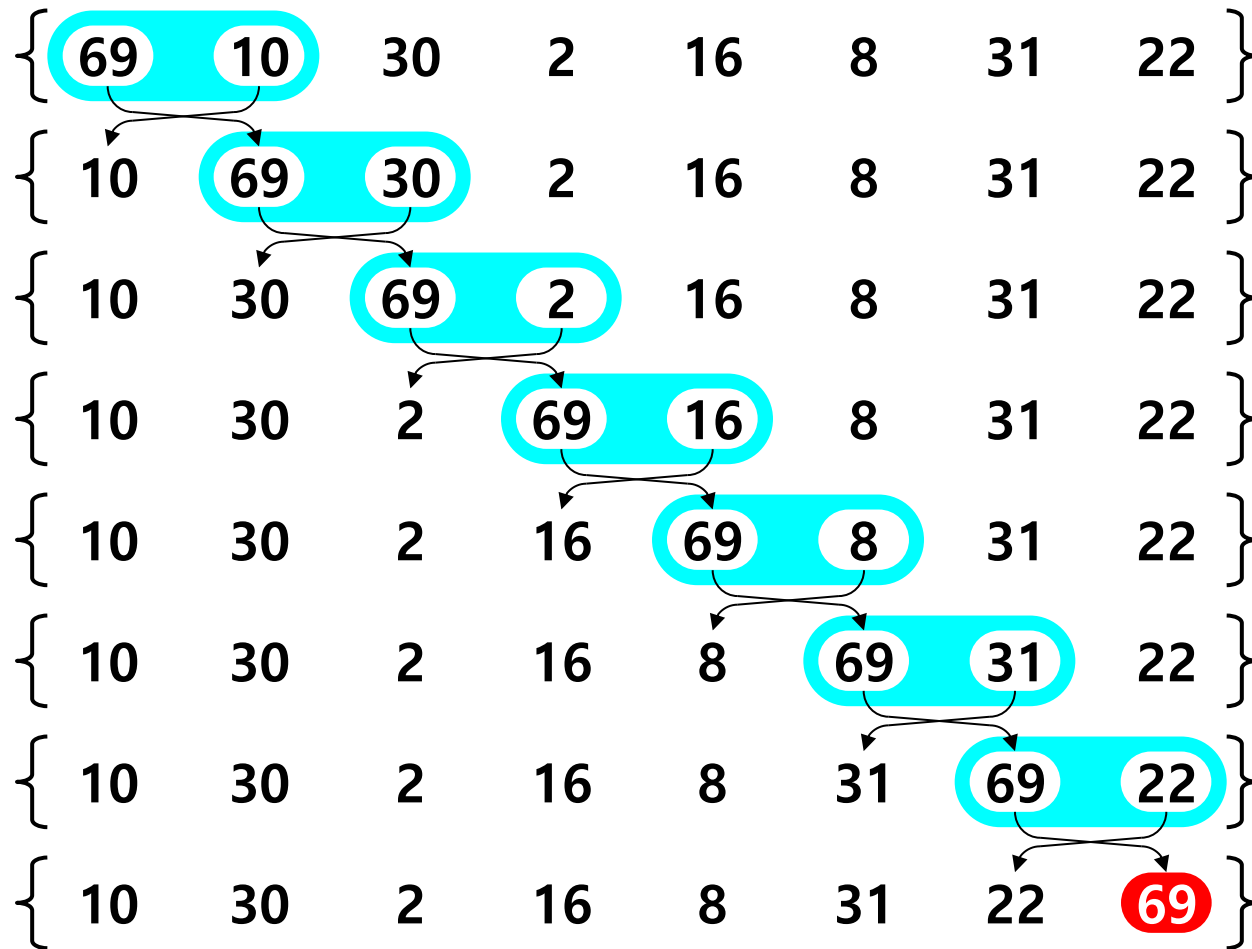
for ($j \leftarrow 0; j < i; j ++$)

if ($a[j] > a[j + 1]$)

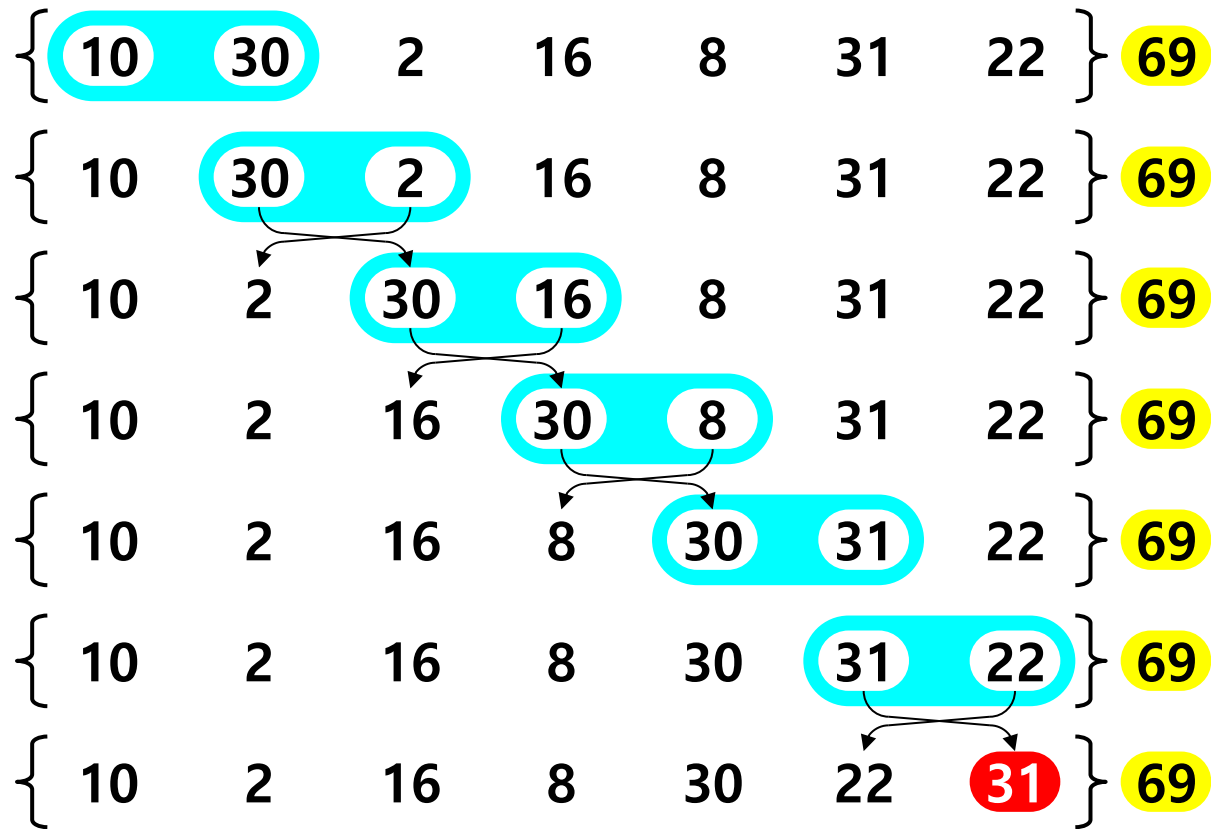
swap($a[j], a[j + 1]$);

}

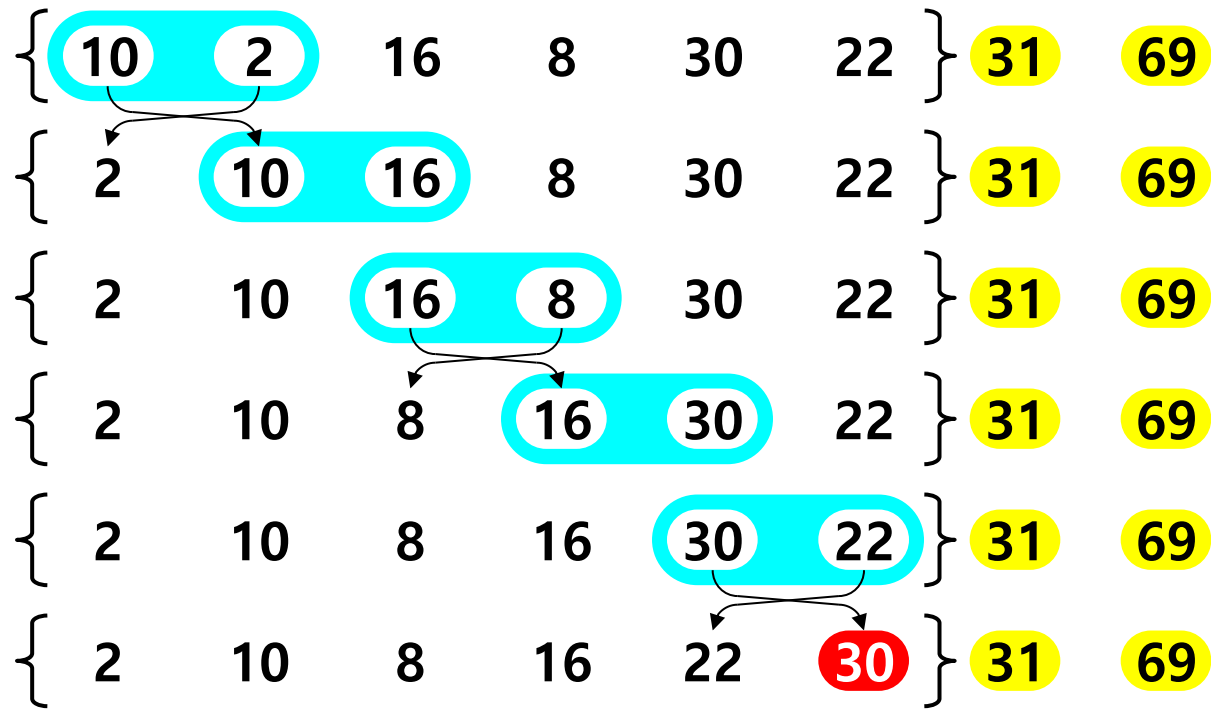
버블 정렬의 예 [1/7]



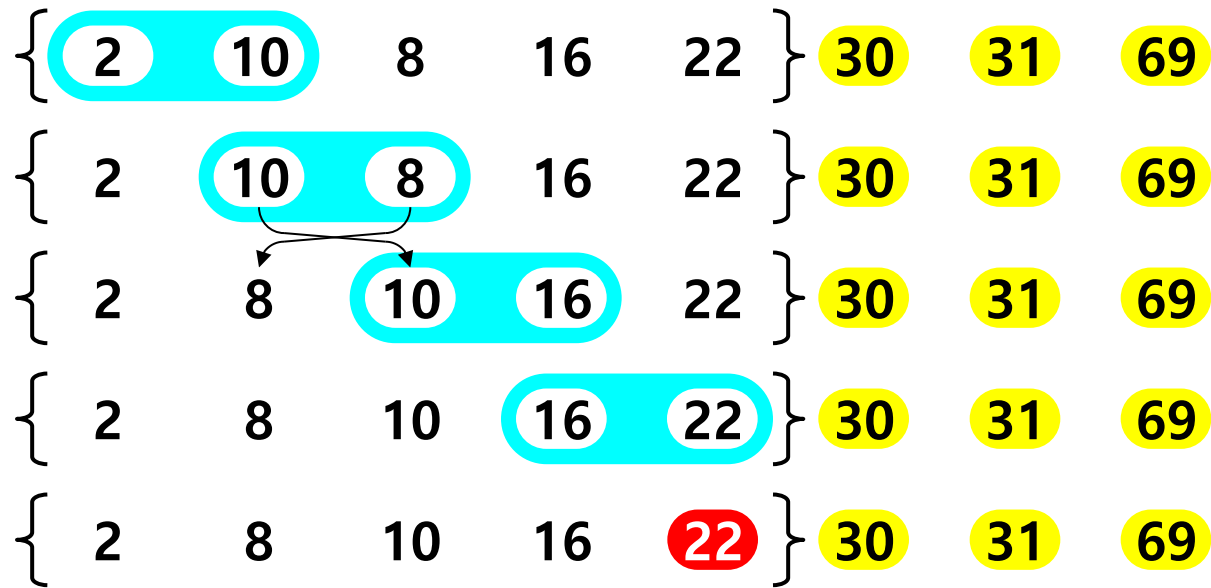
버블 정렬의 예 [2/7]



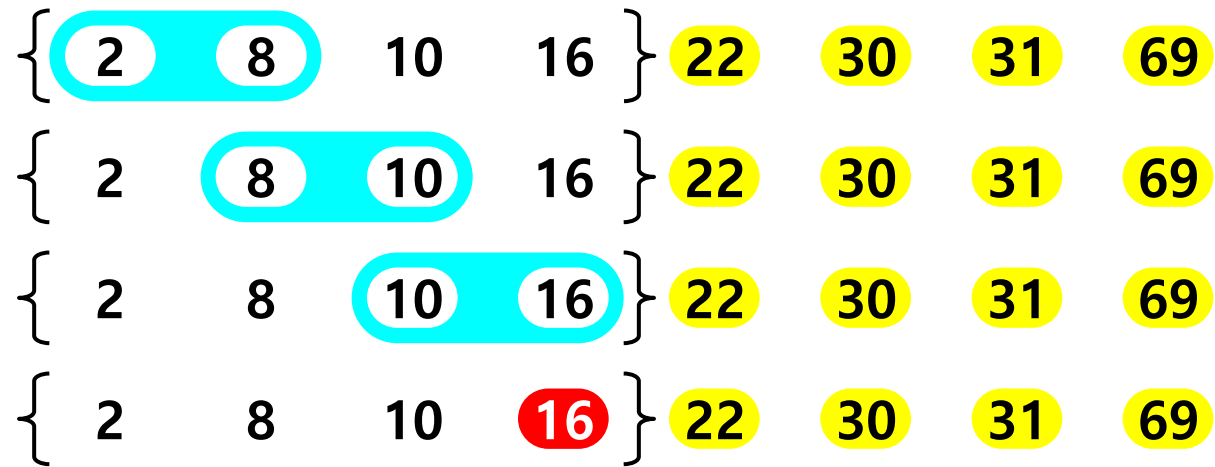
버블 정렬의 예 [3/7]



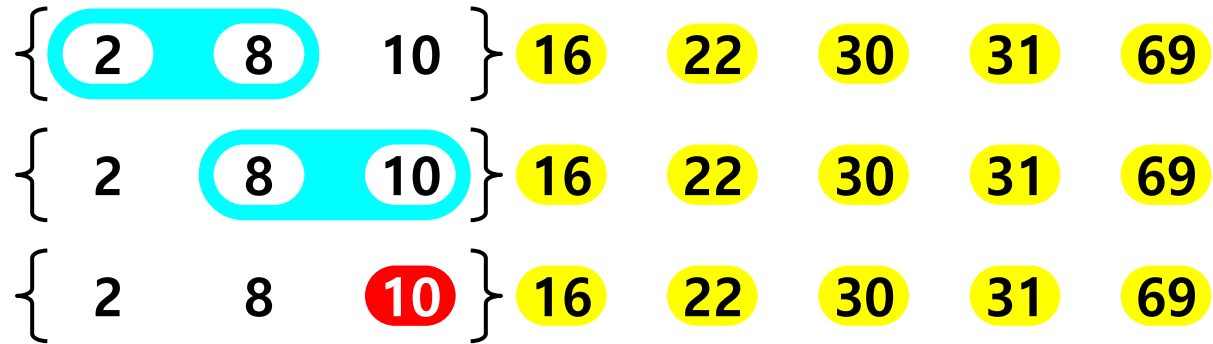
버블 정렬의 예 [4/7]



버블 정렬의 예 [5/7]



버블 정렬의 예 [6/7]



버블 정렬의 예 [7/7]

{ 2 8 } 10 16 22 30 31 69

{ 2 8 } 10 16 22 30 31 69

2 8 10 16 22 30 31 69

버블 정렬 알고리즘 분석

- 메모리 사용공간

- n 개의 원소에 대하여 n 개의 메모리 사용

- 연산 시간

- 최선의 경우: 자료가 이미 정렬되어있는 경우

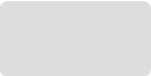
- ◆ 비교횟수: i 번째 실행 시 $(n - i)$ 번 비교하므로, $\frac{n(n-1)}{2}$ 번

- ◆ 자리교환횟수: 자리교환이 발생하지 않는다.

- 최악의 경우: 자료가 역순으로 정렬되어있는 경우

- ◆ 비교횟수: i 번째 실행 시 $(n - i)$ 번 비교하므로, $\frac{n(n-1)}{2}$ 번

- ◆ 자리교환횟수: 모든 i 번째 실행 시 $(n - i)$ 번 교환하므로, $\frac{n(n-1)}{2}$ 번

- 시간 복잡도: 

버블 정렬 알고리즘 개선

- 버블 정렬 알고리즘의 특징
 - 실행 도중 모든 자료가 정렬되면 더 이상 자리교환을 하지 않는다.
 - 반대로, i 번째 실행에서 한번도 자리교환을 하지 않았다면 모든 자료가 정렬되었음을 알 수 있다.

버블 정렬 알고리즘 개선

bubbleSort2($a[], n$) {

입력: 정수 배열 a , 배열 크기 n

출력: 오름차순으로 정렬된 배열 a

for ($i \leftarrow n - 1; i \geq 0; i--$) {

change $\leftarrow false$;

for ($j \leftarrow 0; j < i; j++$)

if ($a[j] > a[j + 1]$) {

swap($a[j], a[j + 1]$);

change $\leftarrow true$;

}

if (***change*** = $false$) *break*;

}

}

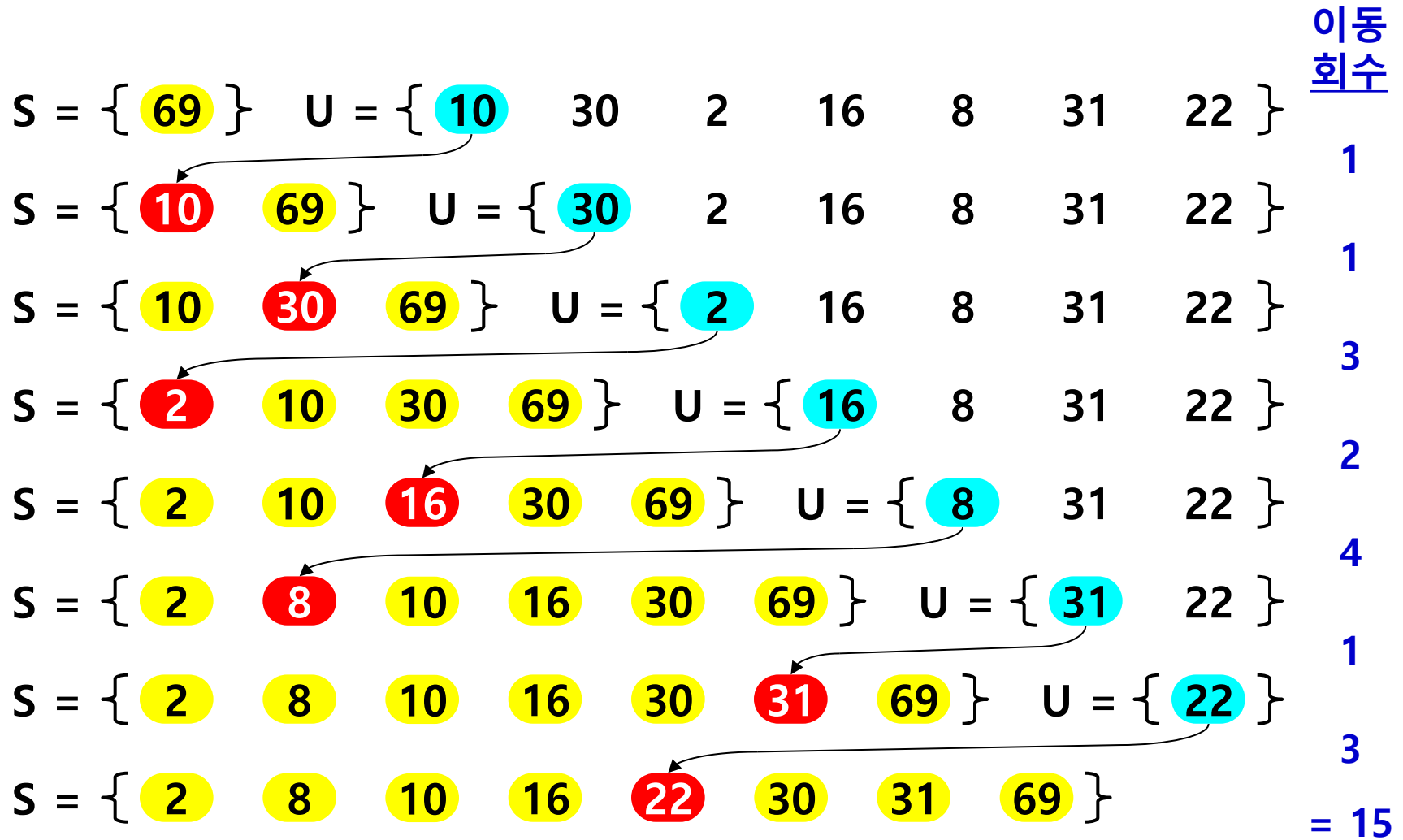
4.4 삽입 정렬(Insert Sort)

- 삽입 정렬(insert sort)
 - 정렬되어있는 부분집합에
정렬할 새로운 원소의 위치를 찾아 삽입하는 방법
 - 정렬할 자료를 두 개의 부분집합 S 와 U 로 가정
 - ◆ 부분집합 S : 정렬된(sorted) 앞부분의 원소들
 - ◆ 부분집합 U : 아직 정렬되지 않은(unsorted) 나머지 원소들
 - ◆ 정렬되지 않은 부분집합 U 의 원소를 하나씩 꺼내서, 정렬되어 있는 부분집합 S 의 마지막 원소부터 비교하여 위치를 찾아 삽입
 - ◆ 삽입 정렬을 반복하면서 부분집합 S 의 원소는 하나씩 증가하고 부분집합 U 의 원소는 하나씩 감소
 - ◆ 부분집합 U 가 공집합이 되면 삽입 정렬 종료

삽입 정렬 알고리즘

```
insertionSort(a[], n) {  
    입력: 정수 배열 a, 배열 크기 n  
    출력: 오름차순으로 정렬된 배열 a  
    for (i ← 1; i < n; i++) {  
        val ← a[i];  
        for (pos ← i; pos > 0; pos--) {  
            if (val < a[pos-1])  
                a[pos] ← a[pos-1];  
            else  
                break;  
        }  
        a[pos] ← val;  
    }  
}
```

삽입 정렬의 예




삽입 정렬 알고리즘의 분석

- 메모리 사용공간

- n 개의 원소에 대하여 n 개의 메모리 사용

- 연산 시간

- 최선의 경우: 원소들이 이미 정렬되어있어서 비교횟수가 최소
 - ◆ 이미 정렬되어있는 경우에는 바로 앞자리 원소와 한번만 비교한다.
 - ◆ 전체 비교횟수: $n - 1$
- 최악의 경우: 모든 원소가 역순으로 되어있어서 비교횟수가 최대
 - ◆ 전체 비교횟수: $1 + 2 + 3 + \dots + (n - 1) = n(n - 1)/2$
- 시간 복잡도: 

4.5 셸 정렬(Shell Sort)

- 셸 정렬(shell sort)
 - 일정 간격(interval) 떨어져있는 자료들끼리 부분집합을 구성하고, 각 부분집합의 원소들에 대해서 삽입 정렬을 수행하는 작업을 간격을 줄여가며 반복하여 전체 원소들을 정렬하는 방법
 - 삽입정렬보다 복잡하지만, 비교연산과 교환연산 회수 감소
 - 그 이유는? 자료의 이동이 빨라지기 때문!
- 셸 정렬의 부분집합
 - 부분집합의 기준이 되는 간격을 매개변수 h 에 저장
 - 한 단계가 수행될 때마다 h 의 값을 $1/2$ 감소시키고, h 가 1이 될 때까지 셸 정렬을 순환 호출

셸 정렬 알고리즘

insertionSort2($a[], first, last, h$) {

입력: 정수 배열 a , 배열의 범위 $first, last$, 간격 h

출력: 지정된 부분이 오름차순으로 정렬된 배열 a

```
for ( $i \leftarrow first + h; i \leq last; i += h$ ) {  
     $val \leftarrow a[i];$   
    for ( $pos \leftarrow i; pos > first; pos -= h$ ) {  
        if ( $val < a[pos-h]$ )  
             $a[pos] \leftarrow a[pos-h];$   
        else  
            break;  
    }  
     $a[pos] \leftarrow val;$   
}
```

```
}
```

shellSort($a[], n$) {

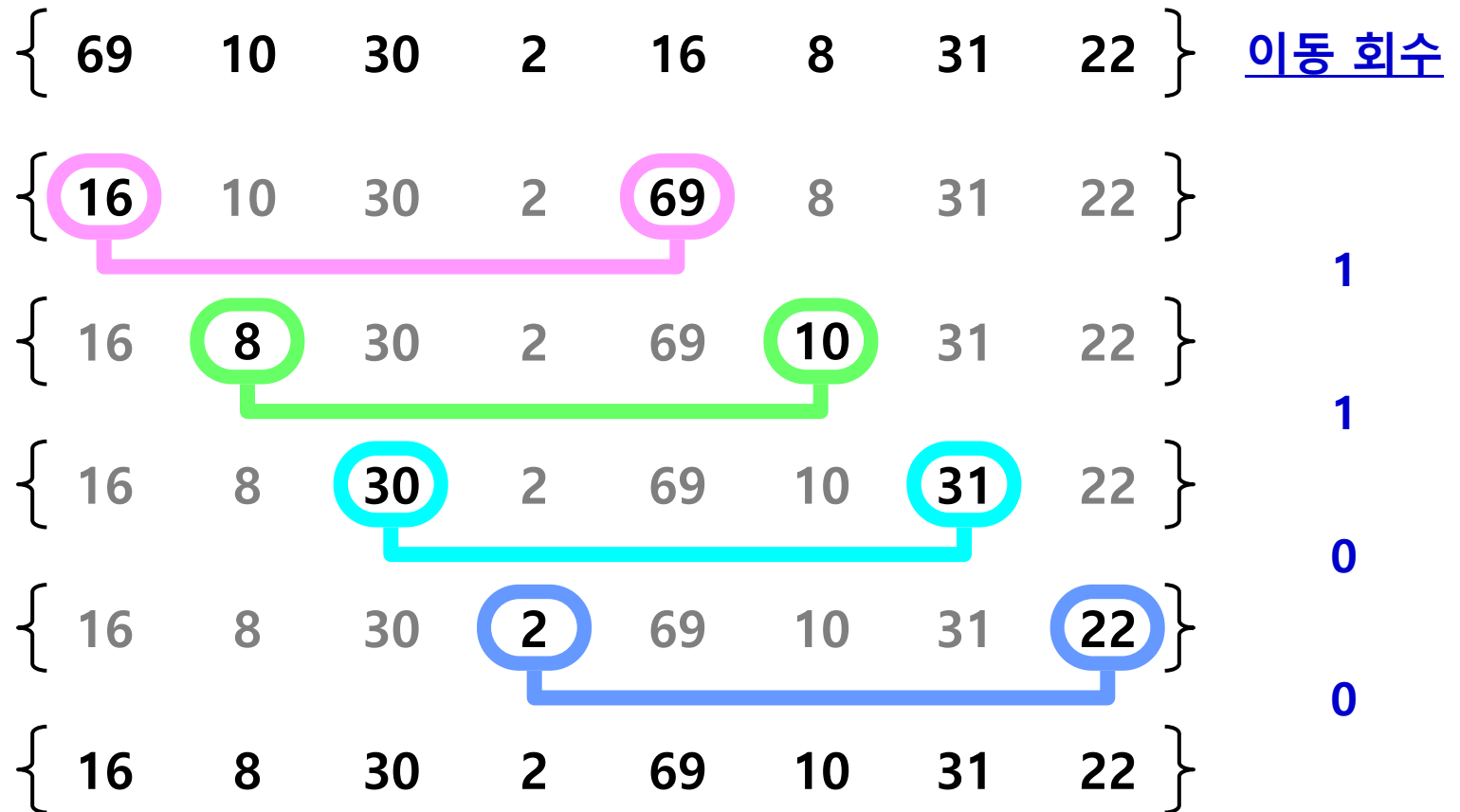
입력: 정수 배열 a , 배열 크기 n

출력: 오름차순으로 정렬된 배열 a

```
for ( $h \leftarrow n/2; h > 0; h = h/2$ )  
    for ( $i \leftarrow 0; i < h; i ++$ )  
        insertionSort2( $a, i, n - 1, h$ );  
}
```

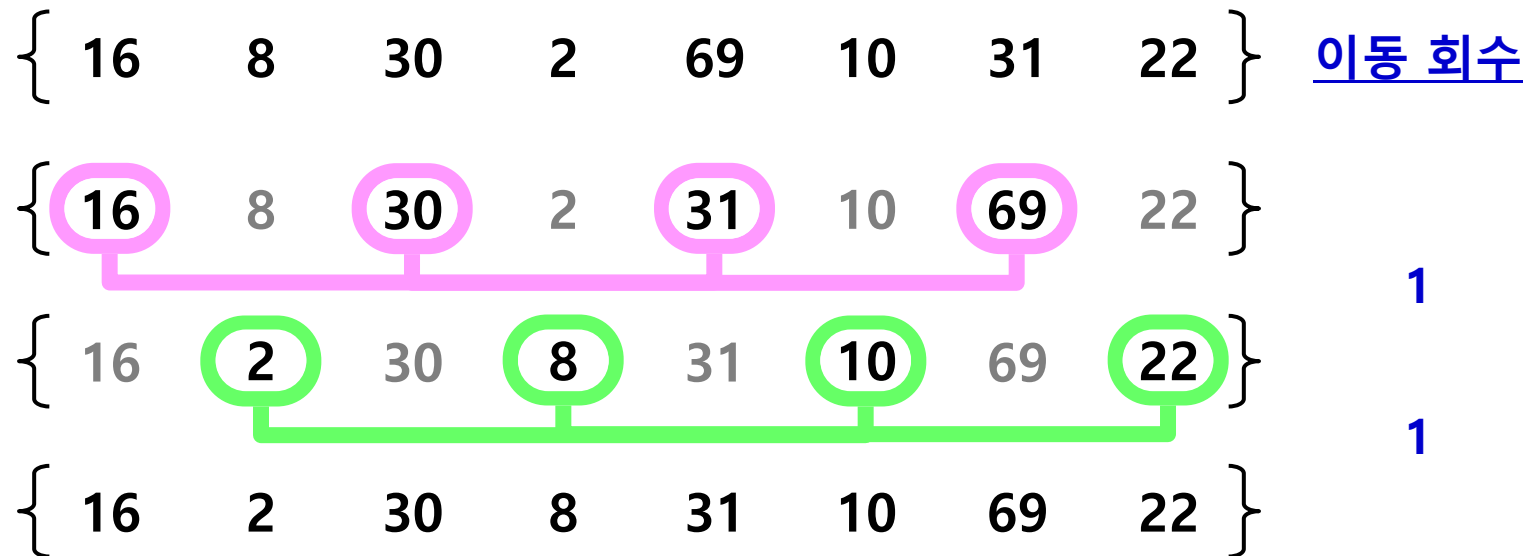
셸 정렬의 예 [1/3]

interval $h = 8/2 = 4$



셸 정렬의 예 [2/3]

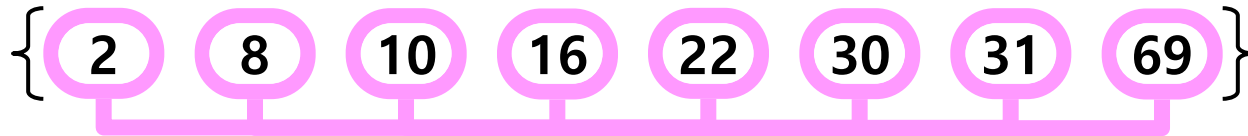
interval $h = 4/2 = 2$



셸 정렬의 예 [3/3]

interval $h = 2/2 = 1$

{ 16 2 30 8 31 10 69 22 } 이동 회수

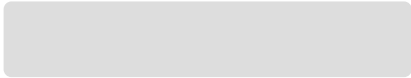


{	16	}	{	2	30	8	31	10	69	22	}	0
{	2	16	}	{	30	8	31	10	69	22	}	1
{	2	16	30	}	{	8	31	10	69	22	}	0
{	2	8	16	30	}	{	31	10	69	22	}	2
{	2	8	16	30	31	}	{	10	69	22	}	0
{	2	8	10	16	30	31	}	{	69	22	}	3
{	2	8	10	16	30	31	69	}	{	22	}	0
{	2	8	10	16	22	30	31	69	}			3

9

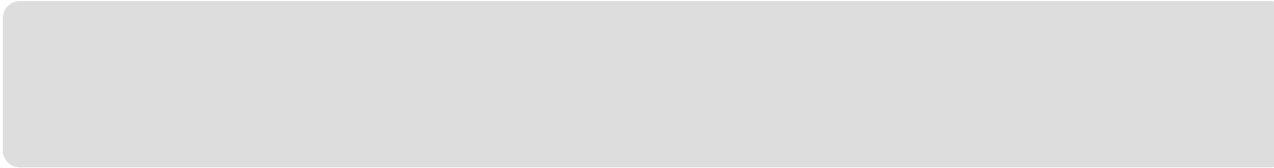
{ 2 8 10 16 22 30 31 69 } 총 이동회수
 $= 2 + 2 + 9$
 $= 13$

셸 정렬 알고리즘의 분석

- 메모리 사용공간
 - n 개의 원소에 대하여
 n 개의 메모리와 매개변수 h 에 대한 저장공간 사용
- 연산 시간
 - 비교횟수는 원소의 상태에 상관없이 매개변수 h 에 의해 결정
 - 최악의 경우 시간복잡도: $O(n^2)$
 - 실험을 통해 확인한 시간 복잡도: 
 - 셸 정렬이 삽입정렬에 비해 빠른 이유
 - ◆ 셸 정렬은 삽입정렬에 비해 최종 위치로 이동하는 속도가 빠름
 - ◆ 예) 69의 경우, 삽입정렬은 7회 이동, 셸 정렬은 3회 이동

분할 정복(Divide-and-Conquer) 기법

- 분할 정복(divide-and-conquer) 알고리즘



- **분할(divide)**: 주어진 문제를 부분 문제(sub-problem)로 분할
 - ◆ 더 이상 분할 할 수 없는 상태까지 반복해서 분할
 - ◆ 일반적으로 재귀 호출 방식 사용
- **정복(conquer)**: 분할된 문제의 부분 해(sub-solution)를 구함
 - ◆ 일반적으로 각 부분 해를 **병합(combine)**하는 작업 필요
- 예) 병합정렬

분할 정복 예제

```
findMax(a[ ], m, n) {  
    입력: 정수 배열  $a$ , 배열의 범위  $m, n$   
    출력: 배열  $a[m:n]$ 의 원소 중의 최대값  
    if ( $m < n$ ) {  
        // divide  
        mid =  $(m + n) / 2$ ;  
        max1 = findMax(a[ ], m, mid);  
        max2 = findMax(a[ ], mid + 1, n);  
        // conquer and combine  
        if (max1 > max2)  
            return max1;  
        else  
            return max2;  
    }  
    return a[m];  
}
```

4.6 병합 정렬(Merge Sort)

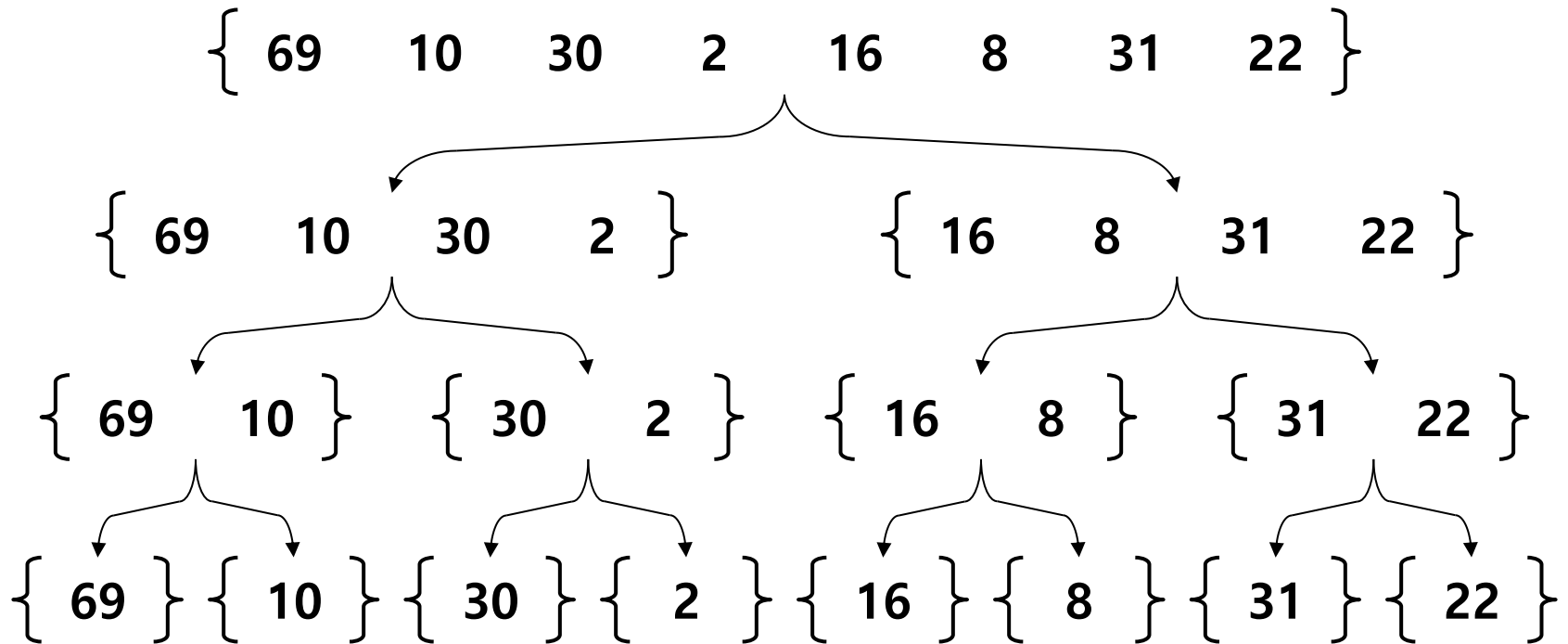
- 병합 정렬(merge sort)
 - 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법
 - 입력 자료를 부분집합으로 분할(divide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer) 한 다음에 정렬된 부분집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법
- 병합 정렬의 종류
 - 2-way 병합: 2개의 정렬된 자료를 결합하여 하나의 집합으로 합체
 - n-way 병합: n개의 정렬된 자료를 결합하여 하나의 집합으로 합체

병합 정렬 알고리즘

```
mergeSort(a[], m, n) {  
    입력: 정수 배열 a, 배열의 범위 m, n  
    출력: 오름차순으로 정렬된 배열 a[m:n]  
    if (a[m:n]의 원소수  $\geq 2$ ) {  
        // 전체 집합을 두개의 부분집합으로 분할  
        mid = (m + n)/2;  
        mergeSort(a[], m, mid);  
        mergeSort(a[], mid + 1, n);  
        merge(a[m:mid], a[mid + 1:n]);  
    }  
    // 원소가 1개 이하면 자연적인 정렬 상태  
}
```

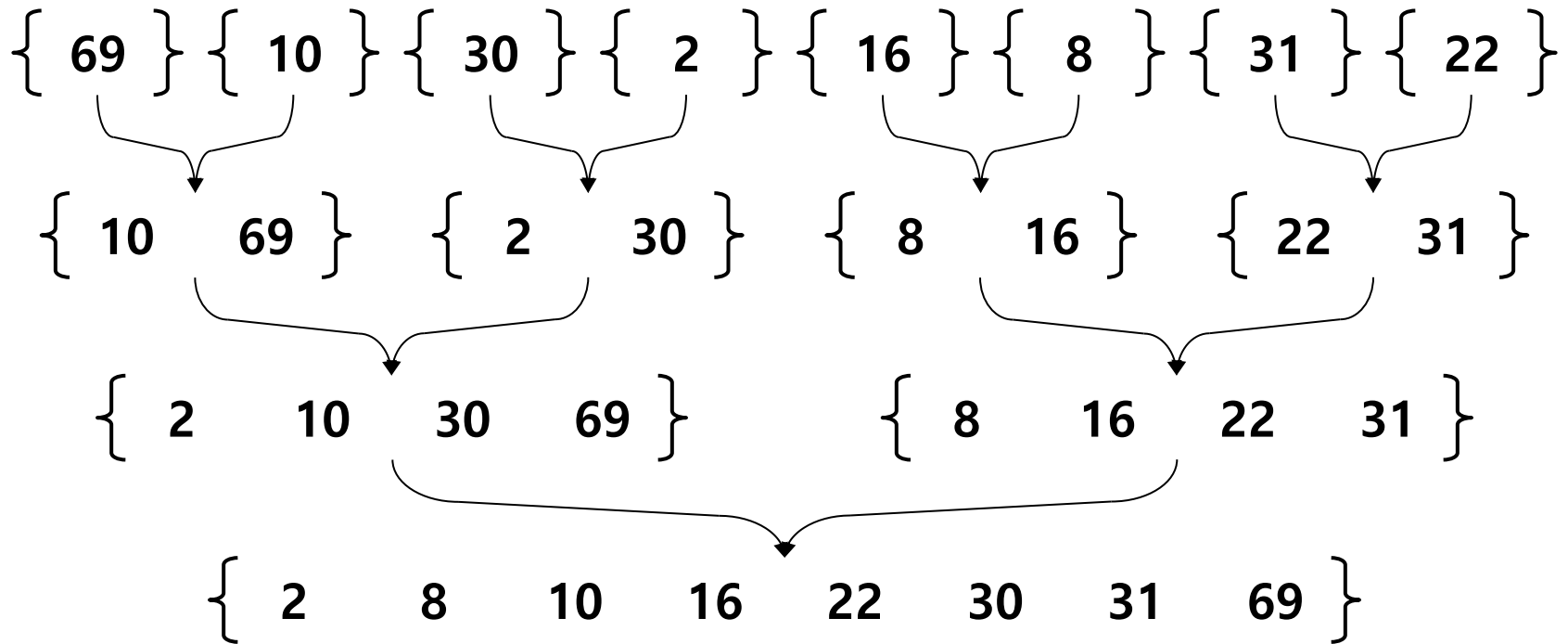
병합 정렬의 예 [1/2]

- 분할 단계




병합 정렬의 예 [2/2]

- 병합 단계




병합 정렬 알고리즘 분석

- 메모리 사용공간
 - 각 단계에서 병합하여 만든 부분집합을 저장할 공간이 필요
 - 원소 n 개에 대해서 $2n$ 개의 메모리 공간 사용
- 연산 시간
 - 분할 단계: n 개의 원소를 분할하기 위해서 $\log_2 n$ 번의 단계 수행
 - 병합 단계: 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n 번의 비교연산 수행
 - 전체 병합 정렬의 시간 복잡도: 

4.7 퀵 정렬(Quick Sort)

- 퀵 정렬(quick sort)

- 기준 값을 중심으로 작은 원소들은 왼쪽 부분집합으로 큰 원소들은 오른쪽 부분집합으로 분할하여 정렬
- 기준 값: 
 - ◆ 좌, 우, 중앙, 또는 평균값으로 선택
- 퀵 정렬은 정복하며 분할하는 원리로 실행
 - ◆ 정복(conquer): 기준 값보다 작은 원소들은 왼쪽 부분집합으로 기준 값보다 큰 원소들은 오른쪽 부분집합으로 구분
 - ◆ 분할(divide): 기준 값을 중심으로 2개의 부분 집합으로 분할
 - ◆ 부분 집합의 크기가 1보다 크면 재귀호출을 이용하여 다시 분할

퀵 정렬 알고리즘 [1/2]

```
quickSort(a[], begin, end) {
```

입력: 정수 배열 *a*, 정렬 범위 *begin*, *end*

출력: 오름차순으로 정렬된 배열 *a*

```
  if (begin < end) {
```

```
    pivot ← partition(a, begin, end);
```

```
    quickSort(a[], begin, pivot - 1);
```

```
    quickSort(a[], pivot + 1, end);
```

```
  }
```

```
}
```

퀵 정렬 알고리즘 [2/2]

```
partiton(a[], begin, end) {
```

입력: 정수 배열 *a*, 정렬 범위 *begin*, *end*

출력: 새로운 *pivot*

```
pivot ← begin; // 제일 왼쪽 원소를 pivot으로 지정
```

```
L ← begin;
```

```
R ← end;
```

```
while (L < R) {
```

```
    while (a[L] ≤ a[pivot] and L < end) L ++;
```

```
    while (a[R] > a[pivot]) R --;
```

```
    if (L < R) swap(a[L], a[R]); // L의 원소와 R의 원소 교환
```

```
}
```

```
swap(a[pivot], a[R]); // R의 원소와 pivot의 원소 교환
```

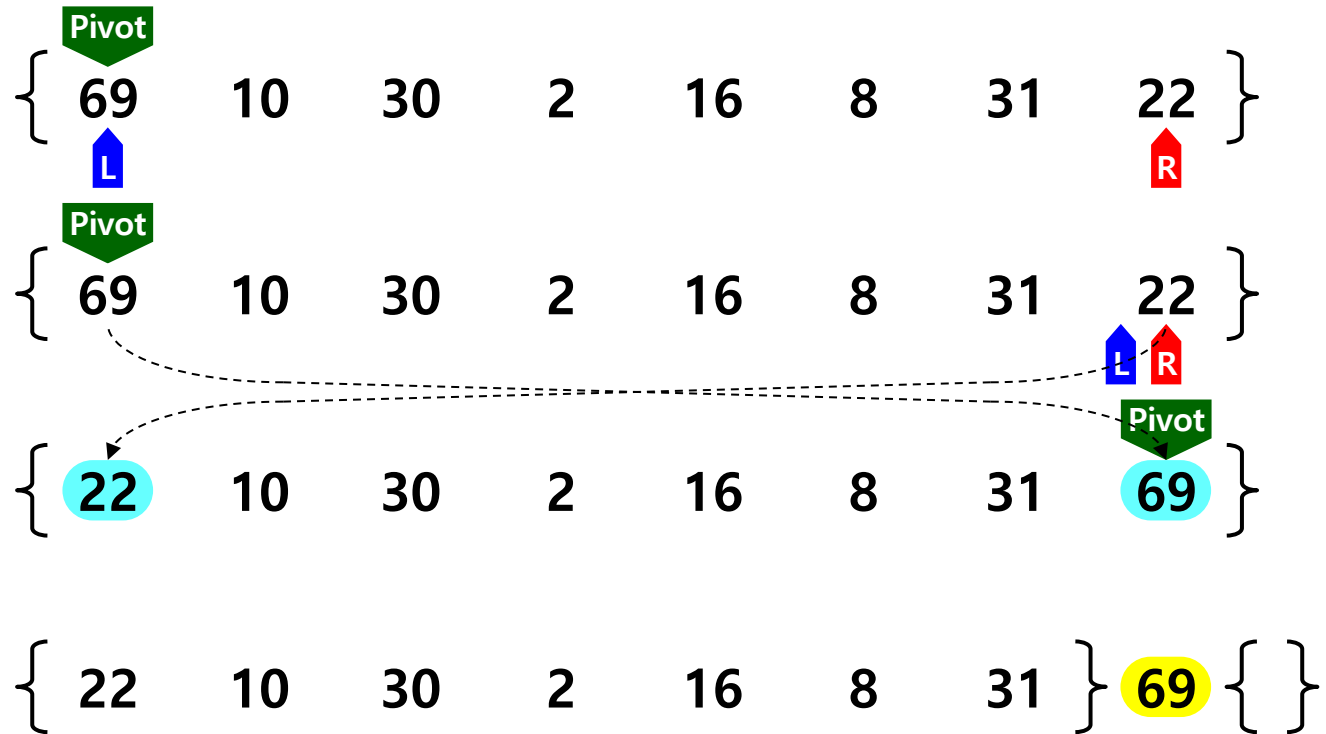
```
return R; // 새로운 pivot 위치 리턴
```

```
}
```

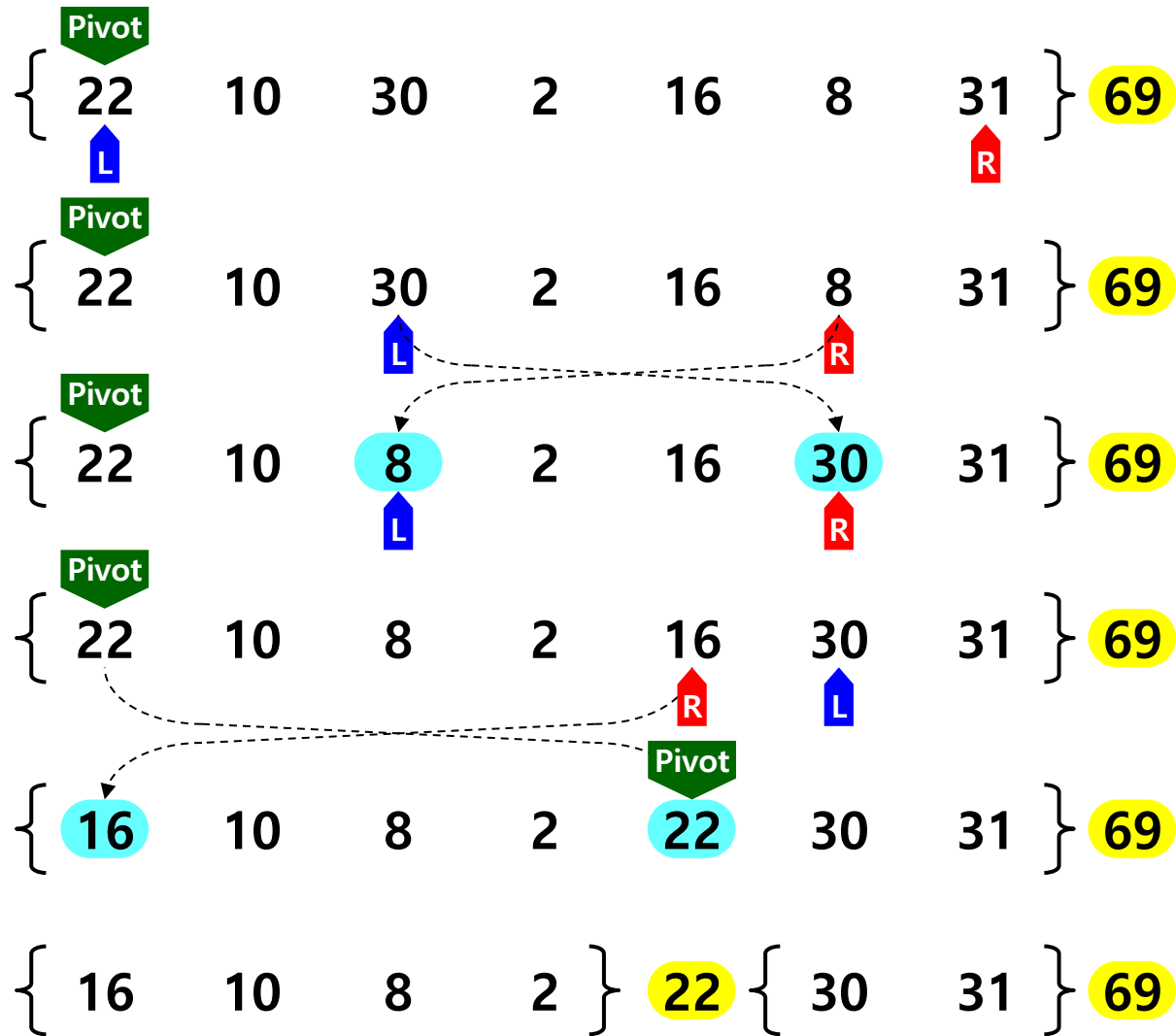
퀵 정렬 알고리즘 설명

- (1) 가장 왼쪽 원소를 pivot으로 지정,
L인덱스를 가장 왼쪽, R인덱스를 가장 오른쪽에 위치
- (2) L인덱스는 pivot보다 큰 값이 나올 때까지 오른쪽으로 이동,
R인덱스는 pivot보다 작거나 같은 값이 나올 때까지 왼쪽으로 이동
- (3) L과 R이 만나지 않았으면, 두 인덱스의 값을 교환하고 (2)로 이동
 - 교환하는 이유는 L인덱스의 원소 값이 pivot보다 크므로 오른쪽으로 보내고, R인덱스의 원소 값이 pivot보다 작거나 같으므로 왼쪽으로 보내기 위함이다.
- (4) L인덱스와 R인덱스가 교차하였으면, pivot과 R인덱스의 값을 교환
 - (2)의 이동 규칙에 따라, L인덱스의 왼쪽 원소는 모두 pivot보다 작거나 같고, R인덱스의 오른쪽 원소는 모두 pivot보다 크다.
 - L과 R이 교차하였으므로 R인덱스의 왼쪽 원소 모두 pivot보다 작거나 같다.
 - pivot과 R인덱스의 위치를 교환하고, 새로운 pivot의 위치를 기준으로 좌우로 pivot보다 작은 원소들의 부분집합과 큰 원소들의 부분집합으로 분할
- (5) 좌우의 부분집합에 대해 퀵 정렬 알고리즘을 재귀 호출

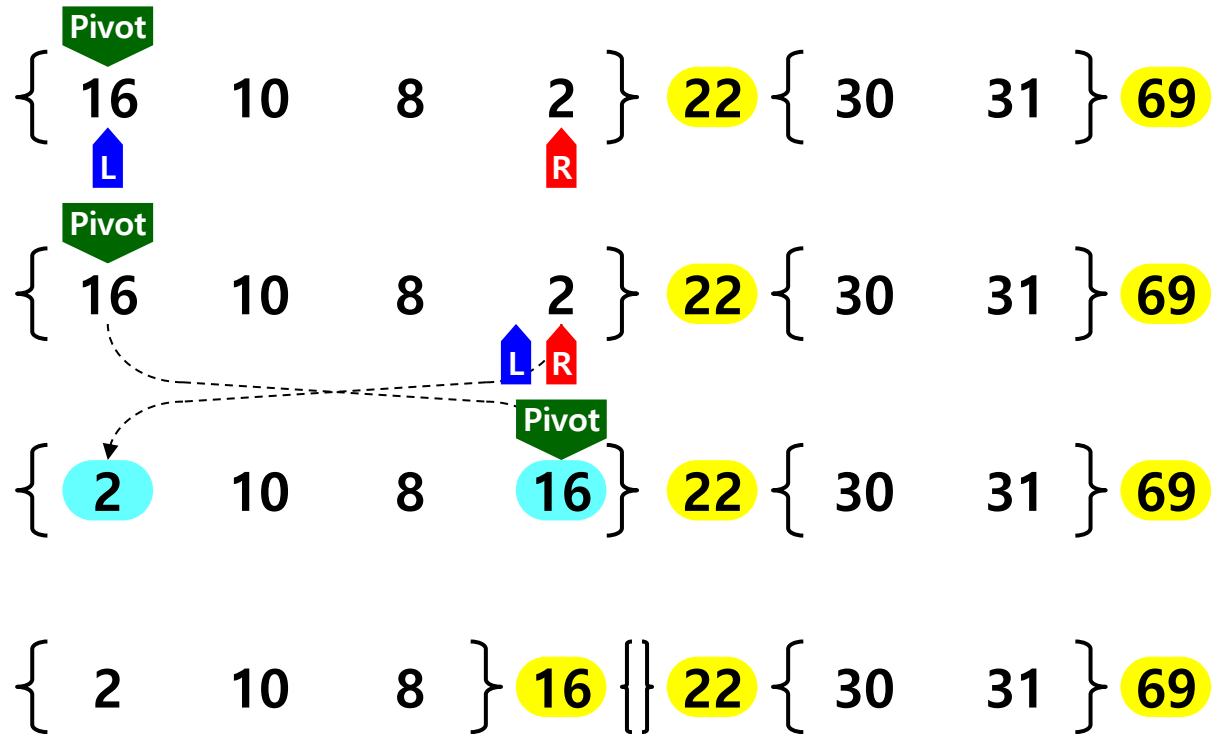
퀵 정렬의 예 [1/6]



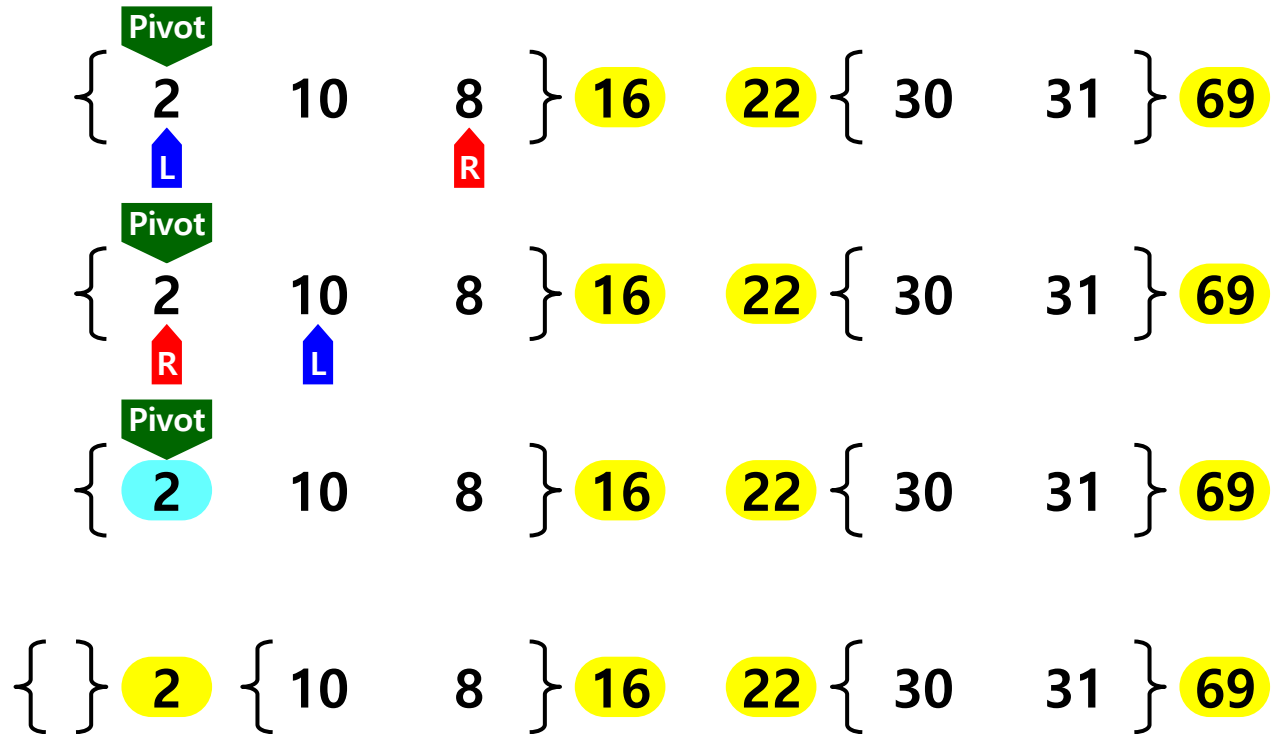
퀵 정렬의 예 [2/6]



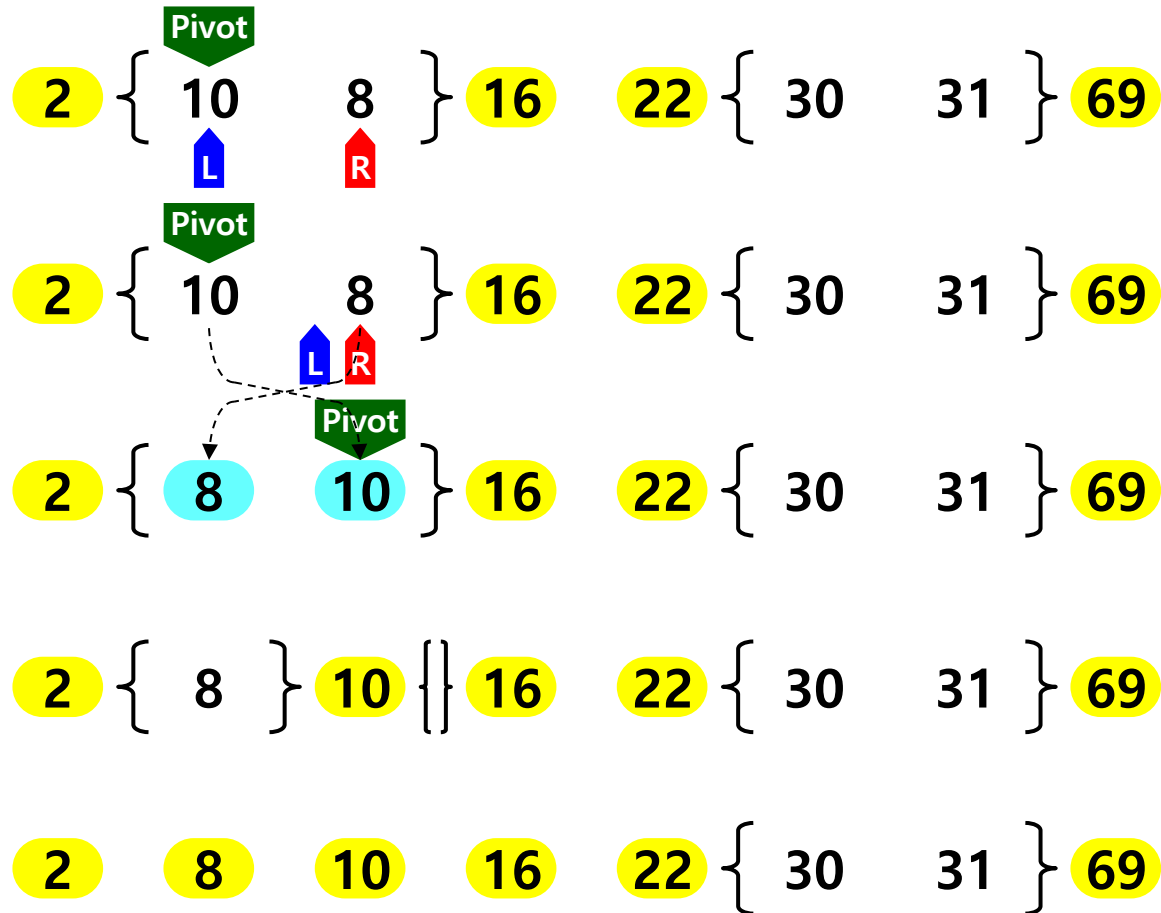
퀵 정렬의 예 [3/6]



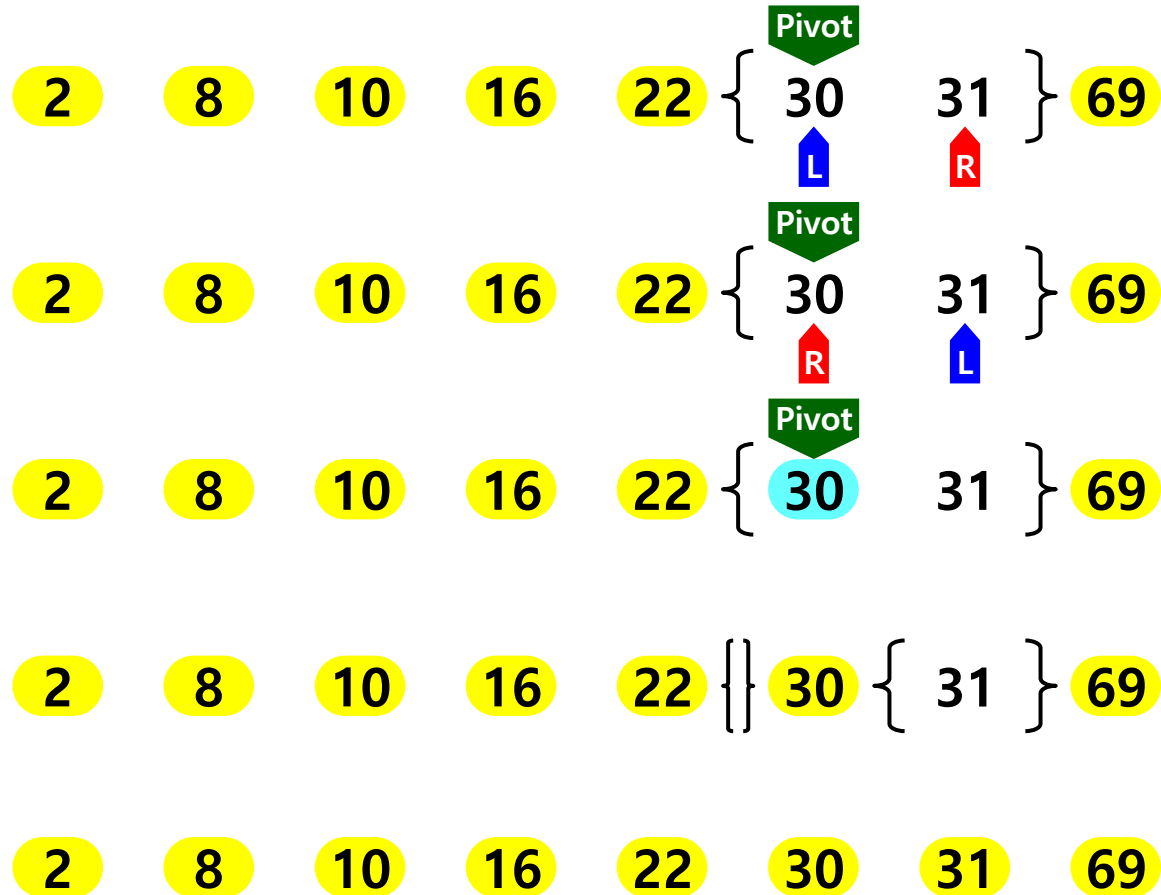
퀵 정렬의 예 [4/6]



퀵 정렬의 예 [5/6]



퀵 정렬의 예 [6/6]



퀵 정렬 알고리즘 분석

- 메모리 사용공간
 - n 개의 원소에 대하여 n 개의 메모리 사용
- 연산 시간
 - 최선의 경우
 - ◆ 왼쪽 부분 집합과 오른쪽 부분 집합 분할이 정확히 이등분 되는 경우
 - 최악의 경우: $O(n^2)$
 - ◆ 부분집합이 1개와 $n - 1$ 개로 치우쳐 분할되는 경우가 반복되는 경우
 - 시간 복잡도:
 - ◆ 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법

4.8 기수 정렬(Radix Sort)

- 기수 정렬(radix sort)

- 원소의 키값을 나타내는 기수를 이용한 정렬 방법

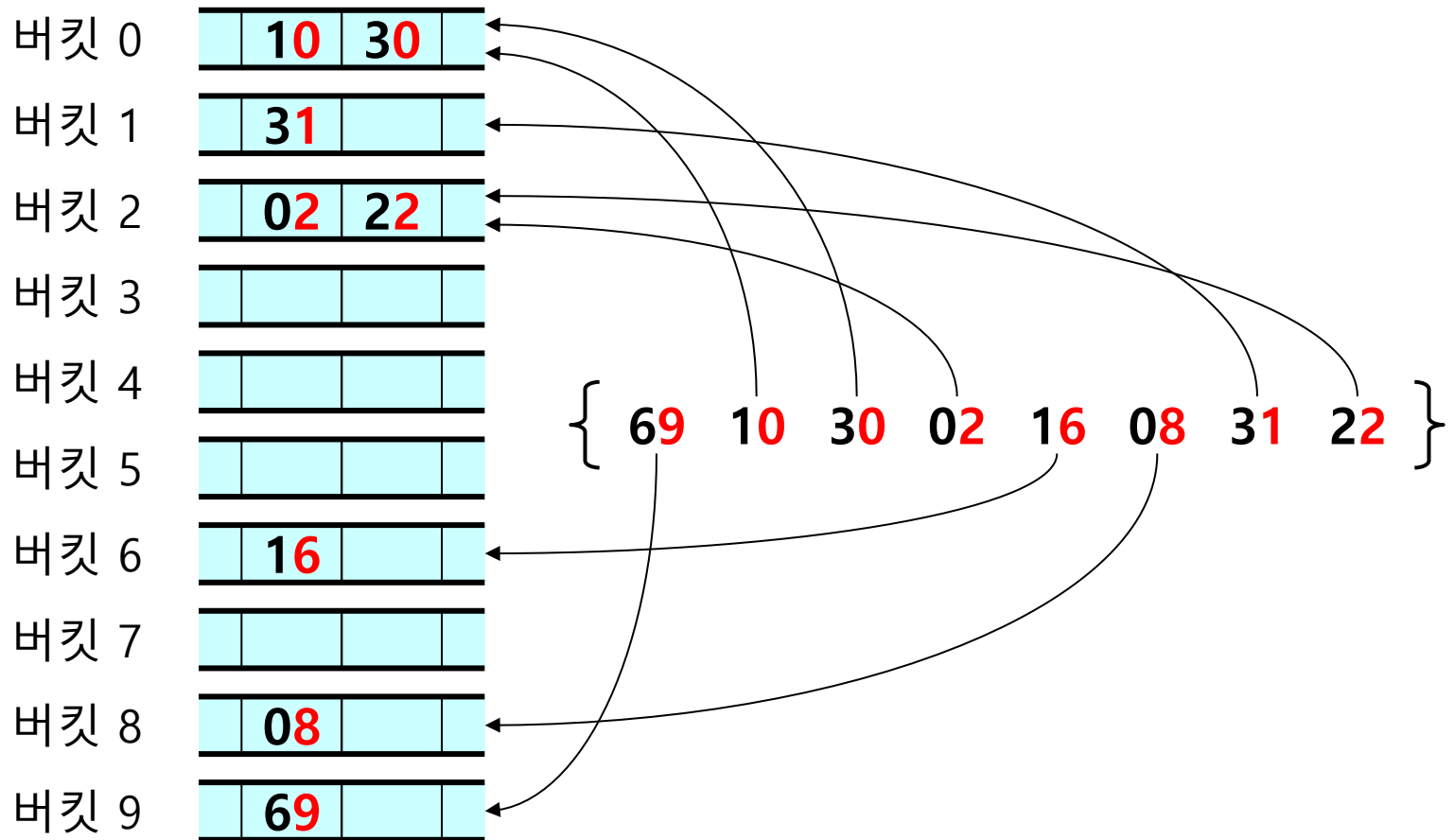
- ◆ 정렬할 원소의 키 값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소를 꺼내는 방법을 반복하면서 정렬
- ◆ 원소의 키를 표현하는 기수만큼의 버킷 사용. 예) 10진수로 표현된 키 값을 가진 원소들을 정렬할 때에는 0부터 9까지 10개의 버킷 사용

- 키 값의 자리수 만큼 기수 정렬을 반복

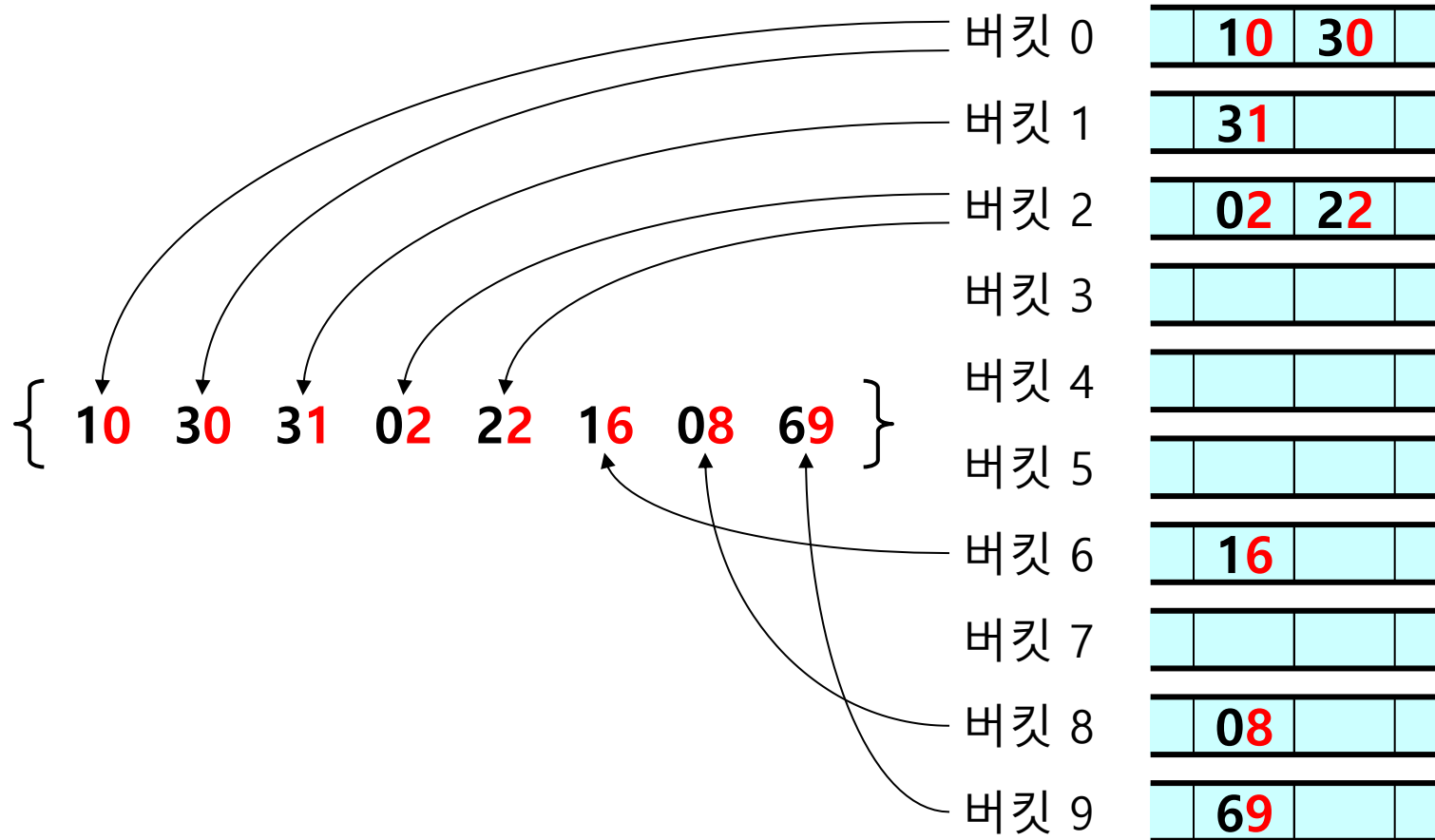
- ◆ 키 값의 일의 자리에 대해 기수 정렬, 십의 자리에 대해 기수 정렬, 백의 자리에 대해 기수 정렬, ...

- 한 단계가 끝날 때마다 버킷에 분배된 원소들을 버킷에 넣은 순서대로 꺼내야 하므로 큐(queue)를 사용하여 버킷을 만든다.

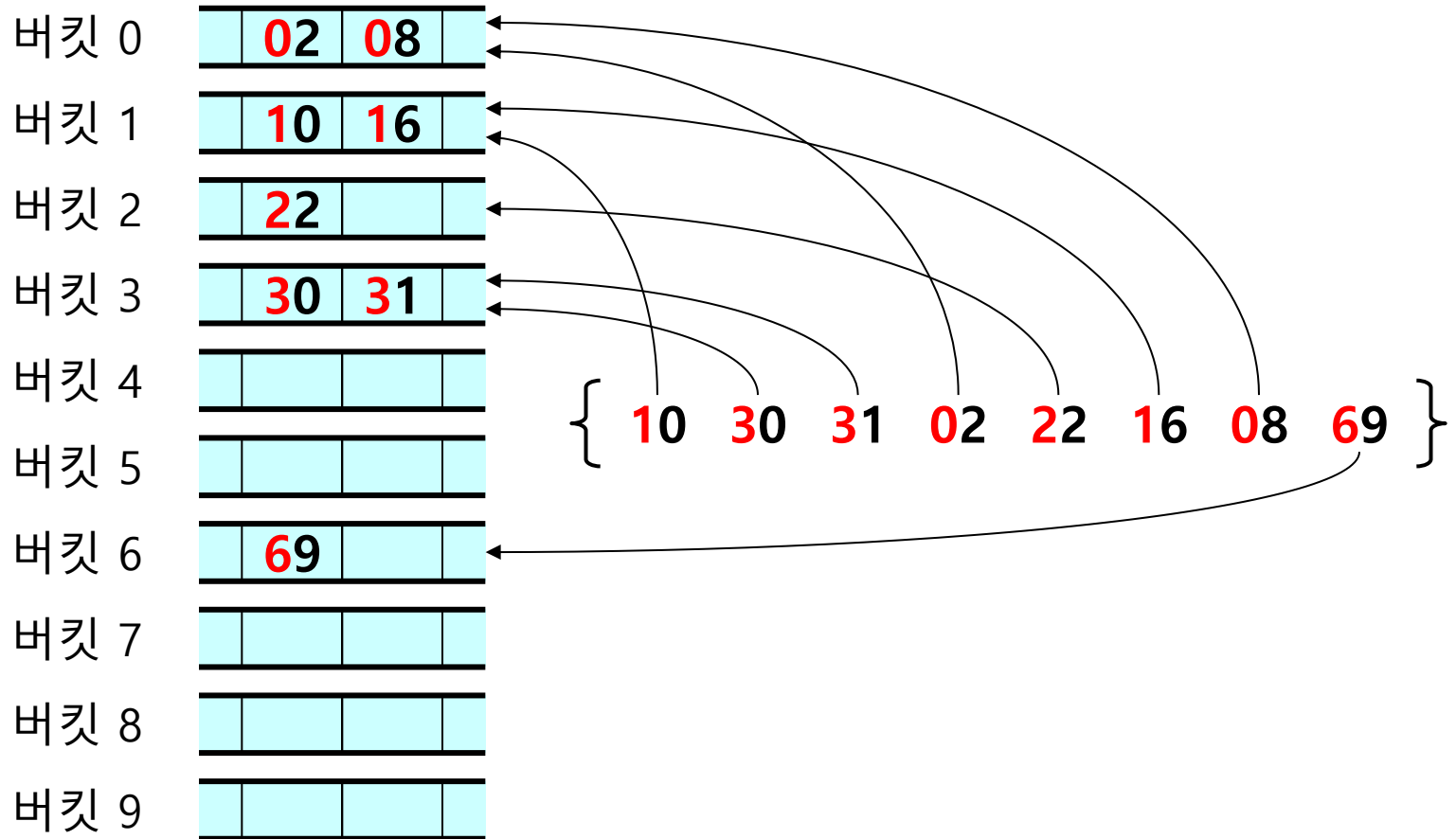
기수 정렬의 예 [1/4]



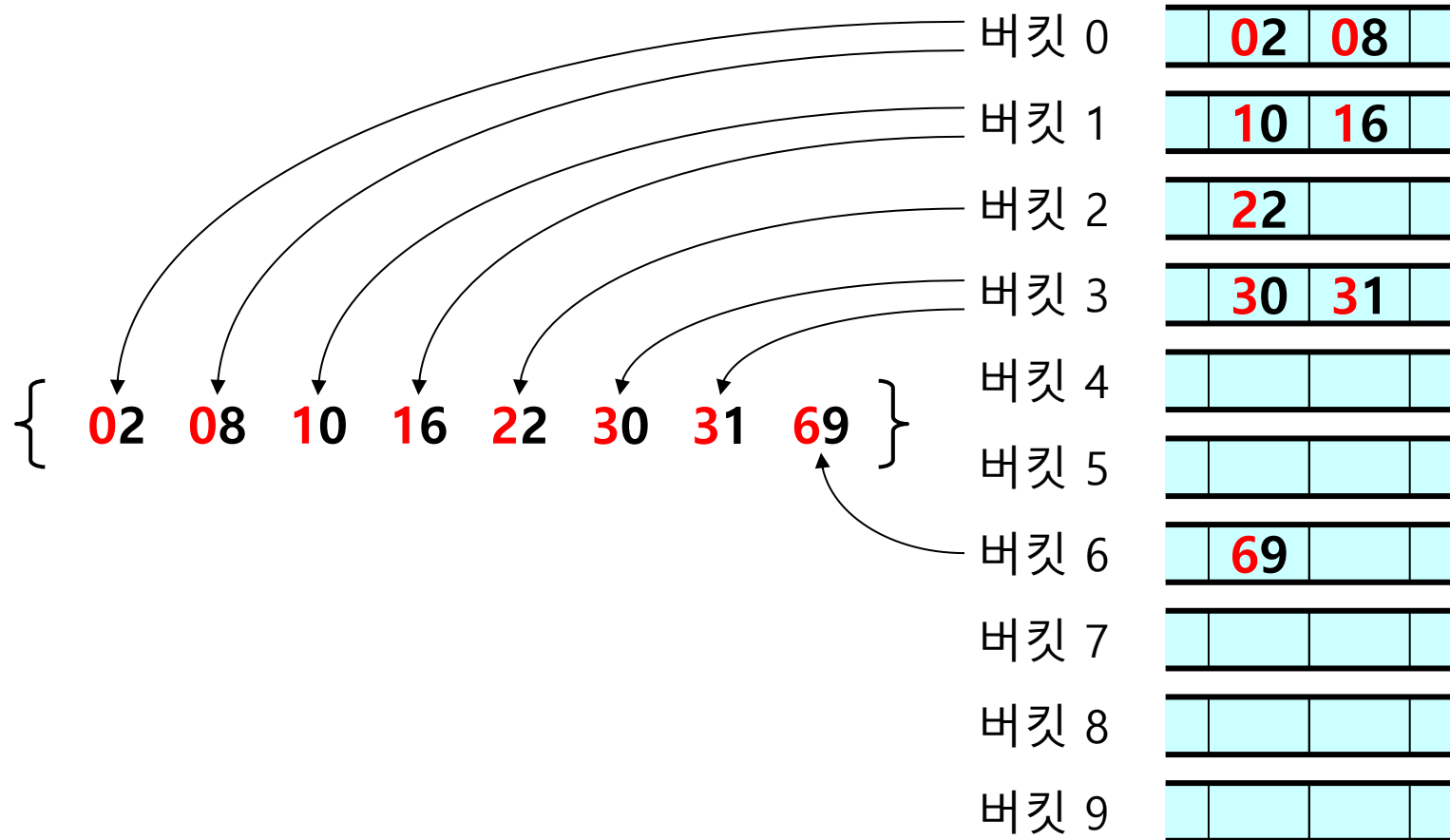
기수 정렬의 예 [2/4]



기수 정렬의 예 [3/4]



기수 정렬의 예 [4/4]

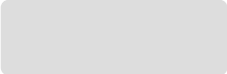



기수 정렬 알고리즘 분석

- 메모리 사용공간

- 원소 n 개에 대해서 n 개의 메모리 공간 사용
- 기수 r 에 따라 버킷 공간이 추가로 필요

- 연산 시간

- 연산 시간은 정렬할 원소의 수 n 과 키 값의 자릿수 d 와 버킷의 수를 결정하는 기수 r 에 따라서 달라진다.
 - ◆ 정렬할 원소 n 개를 r 개의 버킷에 분배하는 작업: $(n + r)$
 - ◆ 이 작업을 자릿수 d 만큼 반복
- 수행할 전체 작업: 
- 시간 복잡도: 

4.9 트리 정렬(Tree Sort)

- 트리 정렬(tree sort)
 - 이진 탐색 트리(BST: Binary Search Tree)를 이용하여 정렬
- 트리 정렬 수행 방법
 1. 정렬할 원소들을 이진 탐색 트리로 구성한다.
 2. 중위 순회(inorder traverse) 경로가 오름차순 정렬이 된다.

트리 정렬 알고리즘

```
treeSort( $a[ ]$ ,  $n$ ) {
```

입력: 정수 배열 a , 배열 크기 n

출력: 오름차순으로 정렬된 배열 a

```
for ( $i \leftarrow 0$ ;  $i < n$ ;  $i++$ )
```

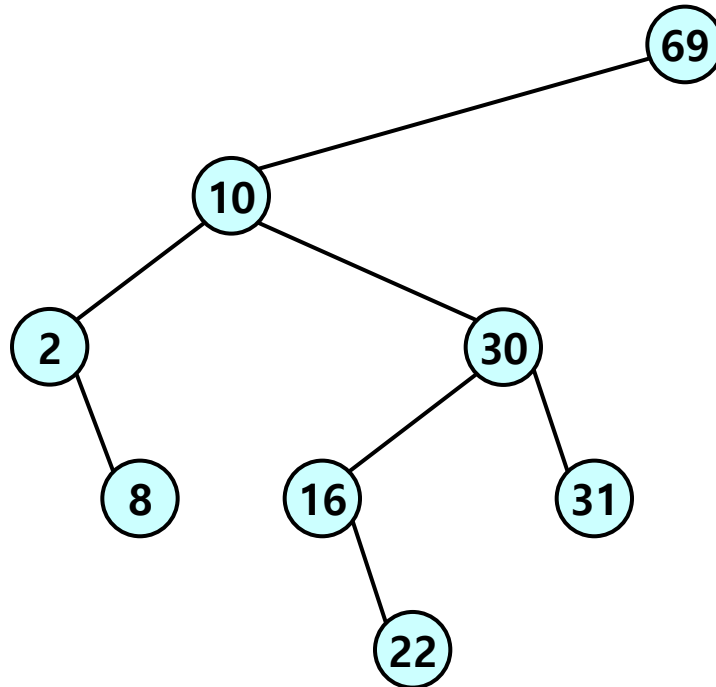
```
    insert( $BST$ ,  $a[i]$ ); // 이진 탐색 트리 삽입
```

```
    inorder( $BST$ ); // 중위 순회 연산
```

```
}
```

트리 정렬의 예

{ 69 10 30 2 16 8 31 22 }



Inorder Traverse: { 2 8 10 16 22 30 31 69 }

트리 정렬 알고리즘의 분석

- 메모리 사용공간
 - 원소 n 개에 대해서 n 개의 메모리 공간 사용
 - 크기 n 의 이진 탐색 트리 저장 공간
- 연산 시간
 - 노드 한 개에 대한 이진 탐색 트리 구성 시간:
 - n 개의 노드에 대한 시간 복잡도:

요약

정렬 방법	메모리 사용공간	연산 시간
선택 정렬(selection sort)	n	$O(n^2)$
버블 정렬(bubble sort)	n	$O(n^2)$
퀵 정렬(quick sort)	n	$O(n \log_2 n)$
삽입 정렬(insert sort)	n	$O(n^2)$
셸 정렬(shell sort)	n	$O(n^{1.25})$
병합 정렬(merge sort)	$2n$	$O(n \log_2 n)$
기수 정렬(radix sort)	$n + r$ (버킷)	$O(d(n + r))$
트리 정렬(tree sort)	n (트리)	$O(n \log_2 n)$

Q&A

