

자료구조 & 알고리즘

for(A;B;C)
D;

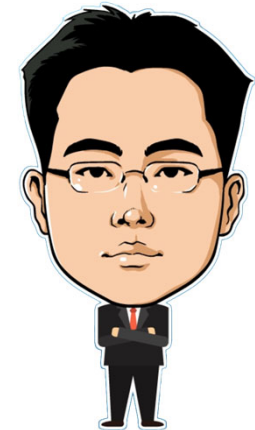


정렬과 탐색
(Sort and Search)

Seo, Doo-Ok

Clickseo.com

clickseo@gmail.com



목 차



- 기초적인 정렬 알고리즘
- 고급 정렬 알고리즘
- 특수 정렬 알고리즘
- 탐색 알고리즘



기초적인 정렬 알고리즘



- 기초적인 정렬 알고리즘

- 선택 정렬
- 버블 정렬
- 삽입 정렬
- 쉘 정렬

- 고급 정렬 알고리즘

- 특수 정렬 알고리즘

- 탐색 알고리즘



정렬 (1/3)

● 정렬(Sort)

○ 순서 없이 배열되어 있는 자료들을 재배열 하는 것

- 정렬의 대상: 레코드
- 정렬의 기준: 정렬 키(sort key) 필드

○ 정렬 방법의 분류

- 실행 방법에 따른 분류: 비교식 정렬, 분산식 정렬
- 정렬 장소에 따른 분류
 - 내부 정렬: 컴퓨터 메모리 내부에서 정렬
 - » 정렬 속도는 빠르지만 자료의 양이 메인 메모리의 용량에 따라 제한된다.
 - » 교환방식, 삽입 방식, 병합 방식, 분배 방식, 선택 방식
 - 외부 정렬: 메모리의 외부인 보조 기억 장치에서 정렬
 - » 내부 정렬로 처리할 수 없는 대용량의 자료를 정렬
 - » 병합 방식: 2-way 병합, n-way 병합

정렬 (2/3)

● 정렬: 알고리즘 성능 비교

○ 점근적 분석

- 항상 입력의 크기가 충분히 크다고 가정하고 수행 시간을 분석한다.

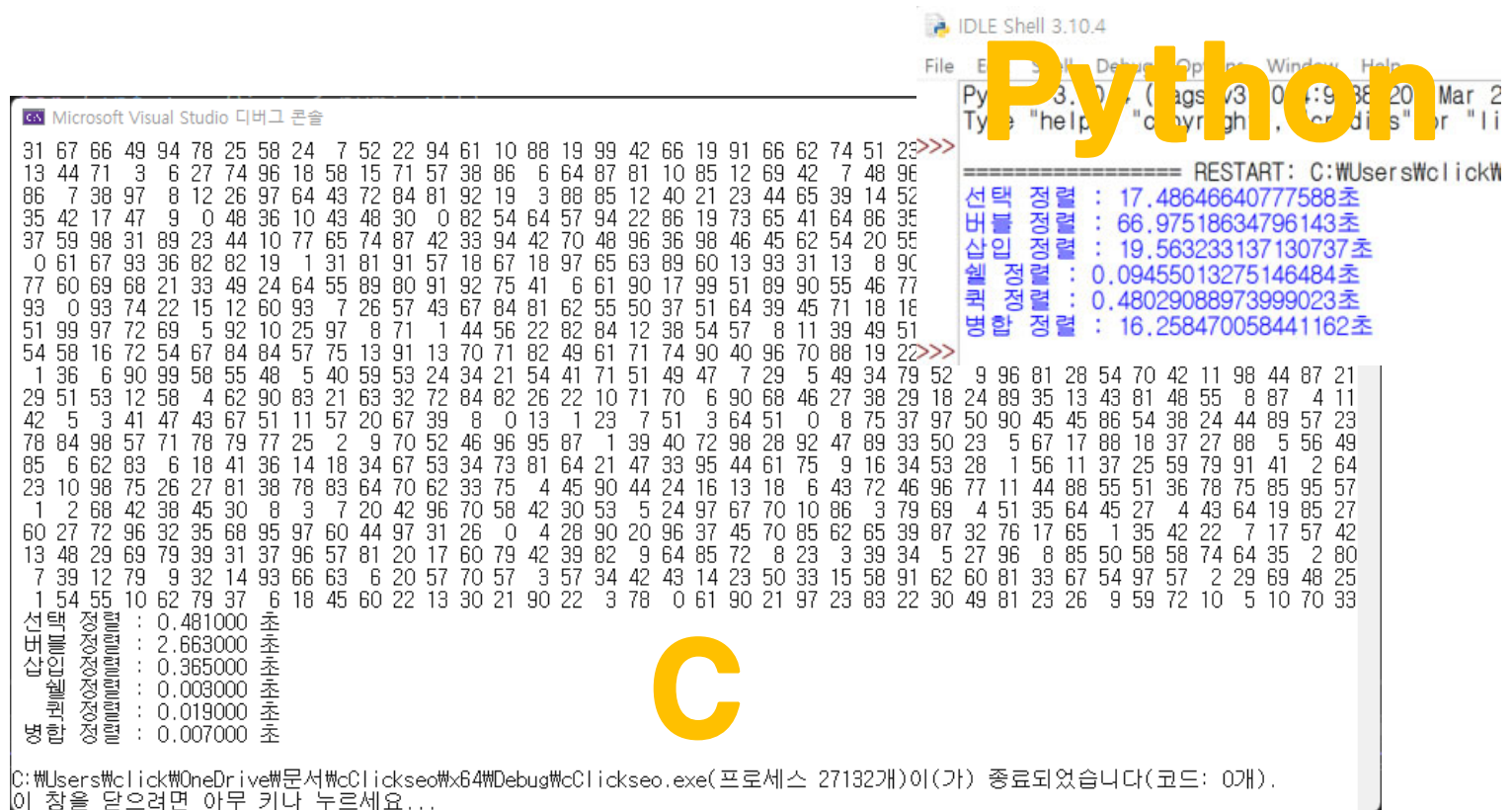
알고리즘	시간 복잡도			공간 복잡도
	최악	평균	최선	
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
버블 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
삽입 정렬	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
셸 정렬	$O(n \log n)$	$O(n^{1.25})$	$O(n^{1.25})$	$O(1)$
퀵 정렬	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
병합 정렬	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
힙 정렬	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
계수 정렬	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n)$
기수 정렬	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$
버킷 정렬	$O(n^2)$	$O(n+k)$	$O(n+k)$	$O(n)$

정렬 (3/3)

- 정렬: 알고리즘 성능 비교

- 경험적 분석

- 알고리즘을 프로그래밍 언어로 구현 후에 실행 시간을 비교해 보는 것



```
Python
===== RESTART: C:\Users\WclickW
선택 정렬 : 17.48646640777588초
버블 정렬 : 66.97518634796143초
삽입 정렬 : 19.563233137130737초
셸 정렬 : 0.09455013275146484초
퀵 정렬 : 0.48029088973999023초
병합 정렬 : 16.258470058441162초
```

정렬 알고리즘	실행 시간 (초)
선택 정렬	17.48646640777588
버블 정렬	66.97518634796143
삽입 정렬	19.563233137130737
셸 정렬	0.09455013275146484
퀵 정렬	0.48029088973999023
병합 정렬	16.258470058441162

C

C:\Users\WclickW\OneDrive\문서\WclickW\Debug\WclickW.exe (프로세스 27132개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...

기초적인 정렬 알고리즘

선택 정렬

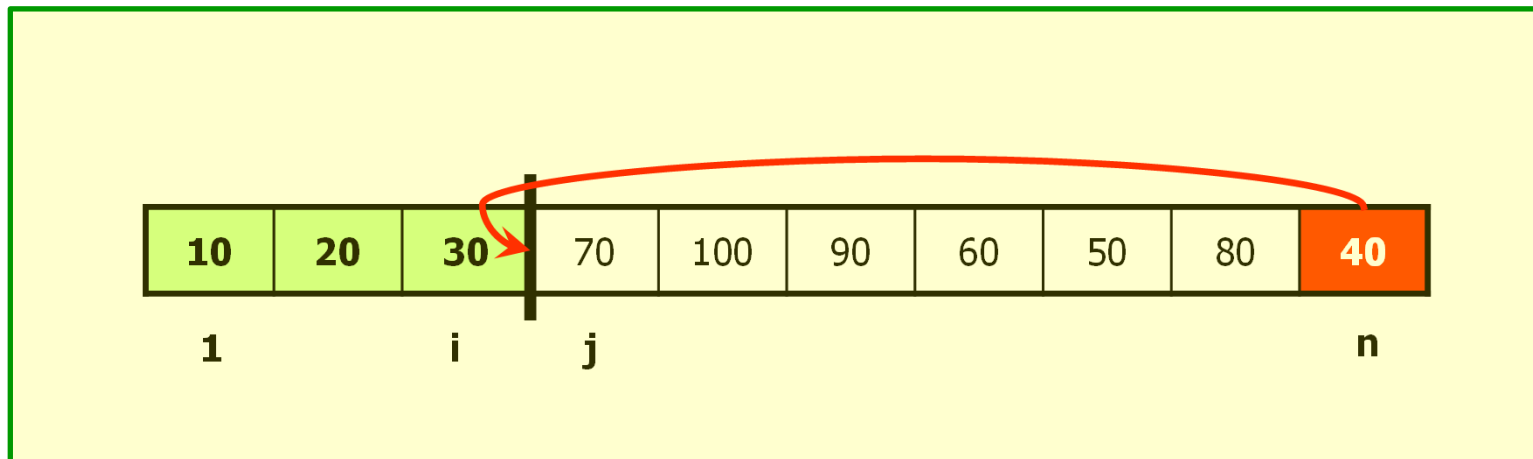


선택 정렬 (1/4)

- **선택 정렬**(Selection Sort)

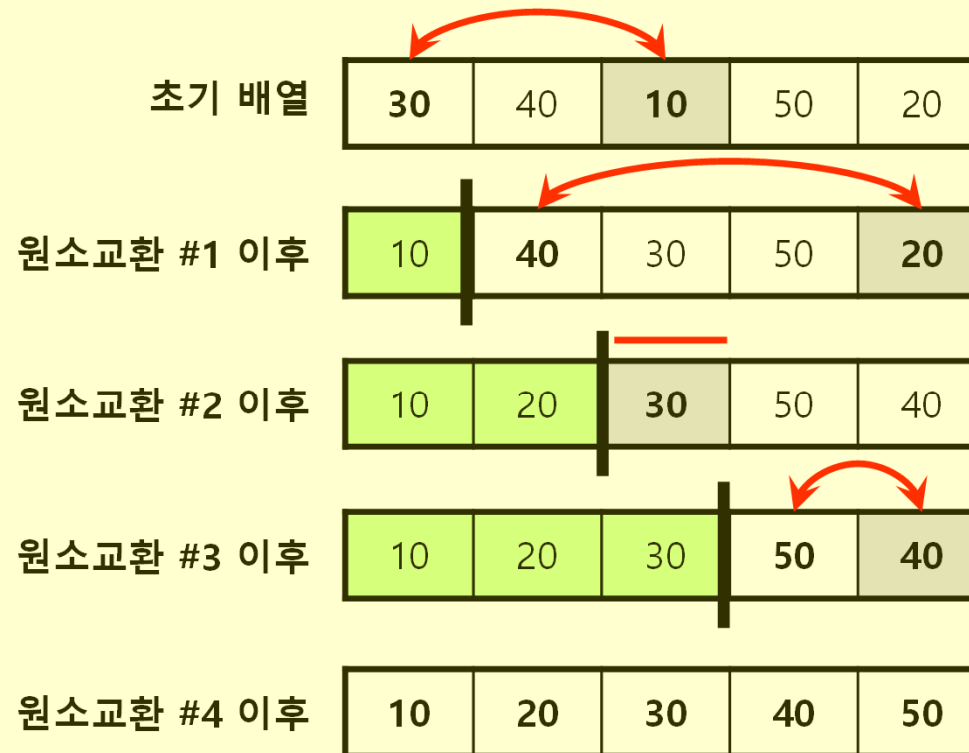
- 배열 원소에 대한 선택 정렬 과정

1. 먼저 정렬되지 않은 리스트에서 가장 작은 원소의 위치 탐색한다.
2. 정렬되지 않은 리스트의 시작 위치에 있는 원소와 교환한다.
3. 각각의 비교 및 교환 후에, 리스트의 경계를 한 개의 원소만큼 **이동**한다.



선택 정렬 (2/4)

선택 정렬 동작 과정



수행시간: $(n - 1) + (n - 2) + \dots + 2 + 1 = O(n^2)$

Worst case

Average case

선택 정렬 (3/4)

● 선택 정렬: 알고리즘

```
selectionSort( A[], n )           // 배열 A[ 1, ... , n ]을 정렬
{
    for i ← 1 to n - 1             ①
    {
        smallest ← i;
        for j ← i + 1 to n
            if (A[j] < A[smallest]) then smallest ← j;  ②
        A[i] ↔ A[smallest];      ③
    }
}
```

수행시간: $1 + 2 + \dots + (n - 1) + n = O(n^2)$

- ① 의 for 루프는 $n - 1$ 번 반복
- ② 에서 가장 작은 수를 찾기 위한 비교 횟수: $1, 2, \dots, n - 1$
- ③ 의 교환은 상수 시간 작업

선택 정렬 (4/4)

- **선택 정렬: 알고리즘 분석**

- **메모리 사용공간:** **n** 개의 원소에 대하여 **n** 개의 메모리 사용

- **원소 비교 횟수**

- 1 단계: 첫 번째 원소를 기준으로 **n** 개의 원소 비교
- 2 단계: 두 번째 원소를 기준으로 마지막 원소까지 **n - 1** 개의 원소 비교
- 3 단계: 세 번째 원소를 기준으로 마지막 원소까지 **n - 2** 개의 원소 비교
- (생략)
- **i** 단계: **i** 번째 원소를 기준으로 **n - i** 개의 원소 비교

$$C_{min} = C_{ave} = C_{max} = (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^{n-1} i$$

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

어떤 경우에서나 원소 비교 횟수가 같기 때문에...

시간 복잡도는 **$O(n^2)$**

기초적인 정렬 알고리즘

버블 정렬

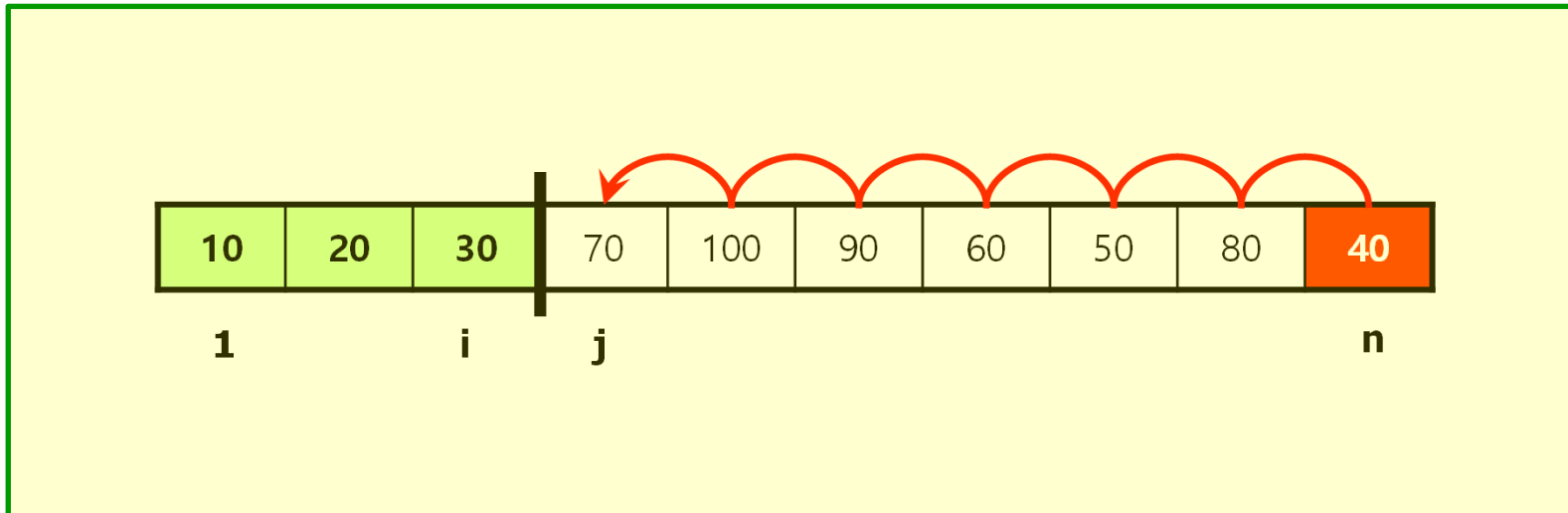


버블 정렬 (1/5)

- **버블 정렬**(Bubble Sort)

- 배열 원소에 대한 버블 정렬 과정

1. 정렬되지 않은 리스트의 가장 작은 원소가 정렬된 서브 리스트로 이동한다.
2. 각각의 비교 및 교환 후에 리스트의 경계를 한 개의 원소만큼 이동한다.



버블 정렬 (2/5)

버블 정렬 동작 과정

초기 배열

30	40	10	50	20
----	----	----	----	----

30	40	10	20	50
----	----	----	----	----

30	40	10	20	50
----	----	----	----	----

30	10	40	20	50
----	----	----	----	----

1단계 완료 이후

10	30	40	20	50
----	----	----	----	----

수행시간: $1 + 2 + \dots + (n - 1) + n = O(n^2)$

Worst case

Average case

버블 정렬 (3/5)

● 버블 정렬: 알고리즘

```
bubbleSort ( A[], n )           // A[ 1 ... n ]을 정렬
{
    for i ← 1 to n - 1           ①
        for j ← n downto i - 1    ②
            if ( A[j] < A[j-1] ) then A[j] ↔ A[j-1]; ③
}
```

수행시간: $(n - 1) + (n - 2) + \dots + 2 + 1 = O(n^2)$

- ① 의 for 루프는 $n - 1$ 번 반복
- ② 에서 가장 큰 수를 찾기 위한 비교 횟수: $n - 1, n - 2, \dots, 2, 1$
- ③ 의 교환은 상수 시간 작업

버블 정렬 (4/5)

- 버블 정렬: 변형(향상)된 알고리즘

```
bubbleSort ( A[], n )           // A[ 1, ... , n ] 을 정렬
{
    for i ← 1 to n - 1
    {
        state ← TRUE;
        for j ← n downto i - 1
        {
            if (A[j] < A[j-1]) then A[j] ↔ A[j-1];
            state ← FALSE;
        }
        if (state = TRUE) then return;
    }
}
```


버블 정렬 (5/5)

- **버블 정렬: 알고리즘 분석**

- **메모리 사용공간:** n 개의 원소에 대하여 n 개의 메모리 사용

- **연산 시간**

- **최선의 경우:** 자료가 이미 정렬되어 있는 경우
 - 원소 비교 횟수: i 번째 원소를 $(n - i)$ 번 비교하기 때문에 $n(n - 1)/2$
 - 원소 교환 횟수: 자리교환이 발생하지 않는다.
- **최악의 경우:** 자료가 역순으로 정렬되어 있는 경우
 - 원소 비교 횟수: i 번째 원소를 $(n - i)$ 번 비교하기 때문에 $n(n - 1)/2$
 - 원소 교환 횟수: i 번째 원소를 $(n - i)$ 번 교환하기 때문에 $n(n - 1)/2$

평균 시간 복잡도는 $O(n^2)$

기초적인 정렬 알고리즘

삽입 정렬, 쉘 정렬

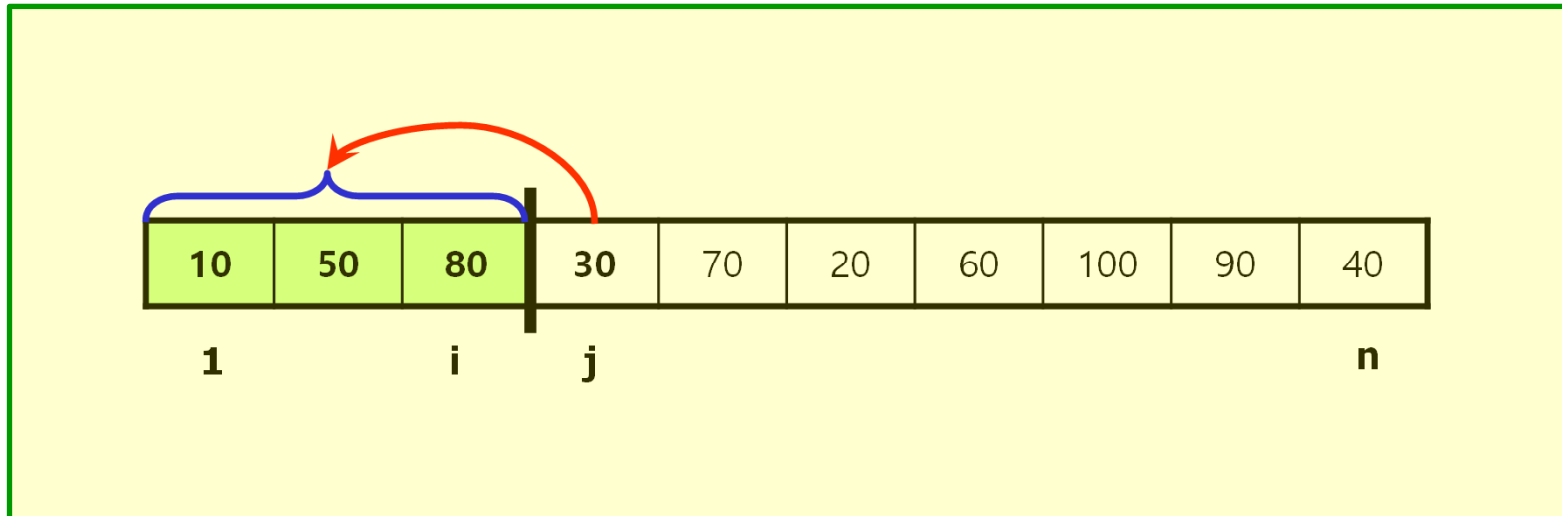


삽입 정렬 (1/4)

- **삽입 정렬**(Insertion Sort)

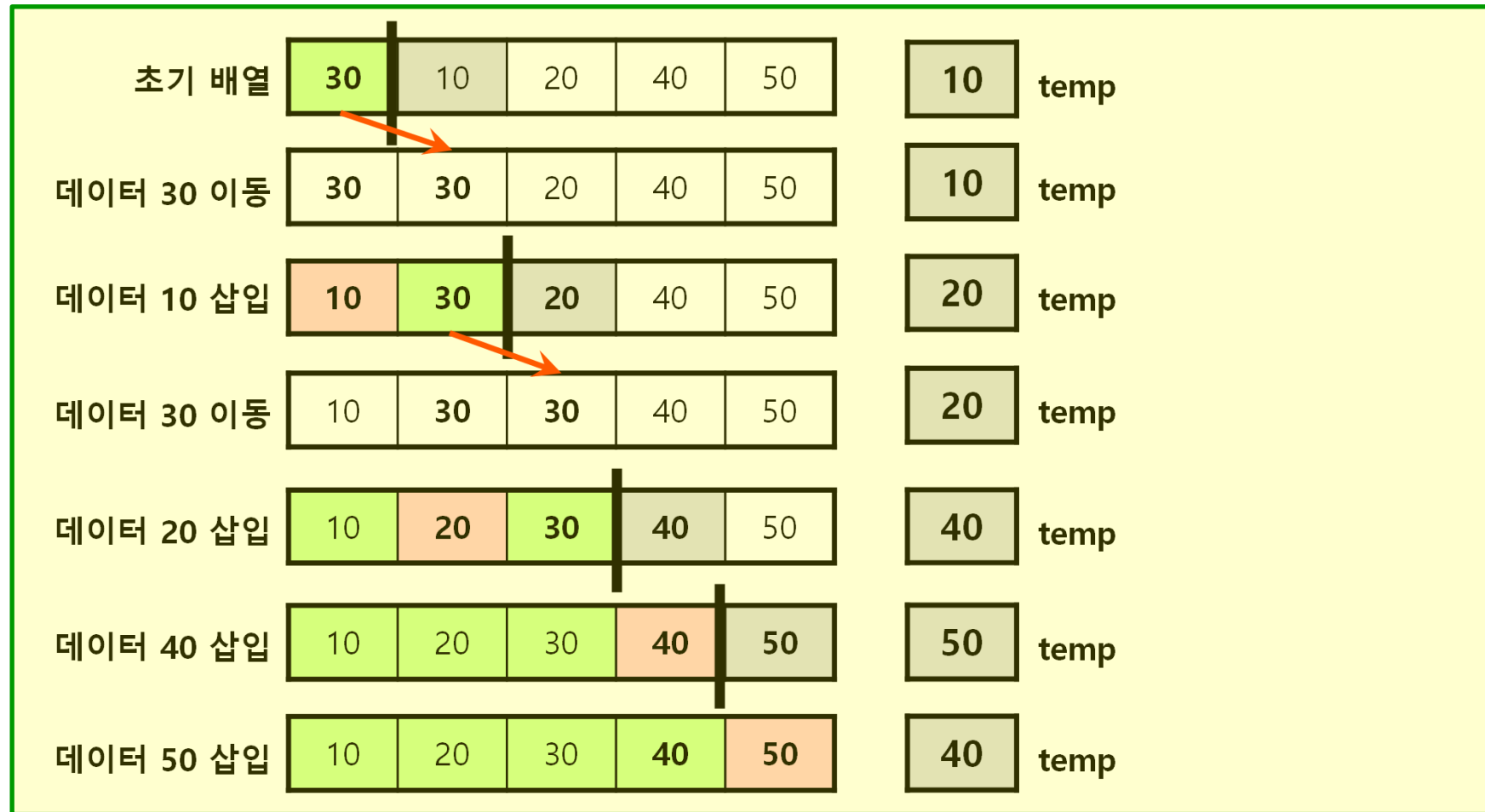
- 배열 원소에 대한 삽입 정렬 과정

1. 각 단계에서 정렬되지 않은 리스트의 첫 번째 원소를 **선택**한다.
2. 선택된 원소를 정렬된 리스트의 적절한 위치로 **삽입**한다.



삽입 정렬 동작 과정

삽입 정렬 (2/4)



수행시간: $O(n^2)$

Worst case: $1 + 2 + \dots + (n-2) + (n-1)$

Average case: $\frac{1}{2}(1 + 2 + \dots + (n-2) + (n-1))$

삽입 정렬 (3/4)

● 삽입 정렬: 알고리즘

insertionSort (A[], n)

// A[1, ... , n] 을 정렬

```
{  
  for i ← 2 to n ①  
  {  
    j ← i - 1;  
    temp ← A[i];  
    while ( j ≥ 1 and temp < A[j] ) ②  
    {  
      A[j + 1] ← A[j];  
      j--;  
    }  
    A[j + 1] ← temp;  
  }  
}
```

“배열이 거의 정렬되어 있는 상태 일 때
가장 매력적인 알고리즘”

수행시간:

① 의 for 루프는 $n - 1$ 번 반복

② 의 삽입은 최악의 경우 $i - 1$ 회 비교

Worst case: $1 + 2 + \dots + (n - 2) + (n - 1) = O(n^2)$

Average case: $\frac{1}{2}(1 + 2 + \dots + (n - 2) + (n - 1)) = O(n^2)$

삽입 정렬 (4/4)

- 삽입 정렬: 알고리즘 분석

- 메모리 사용 공간: n 개의 원소에 대하여 n 개의 메모리 사용

- 연산 시간

- 최선의 경우: 원소들이 이미 정렬되어 있을 때 원소 비교 횟수가 최소
 - 이미 정렬되어 있는 경우에는 바로 앞자리 원소와 한번만 비교
 - 전체 원소 비교 횟수 = $n - 1$
 - 시간 복잡도: $O(n)$
- 최악의 경우: 모든 원소가 역순으로 되어있을 경우 원소 비교 횟수가 최대
 - 전체 원소 비교 횟수 = $1 + 2 + 3 + \dots + (n - 1) = n(n - 1)/2$
 - 시간 복잡도: $O(n^2)$
- 삽입 정렬의 평균 원소 비교 횟수 = $n(n - 1) / 4$

평균 시간 복잡도는 $O(n^2)$

셸 정렬 (1/3)

- 셸 정렬(Shell Sort)

- 일정한 간격(interval)으로 데이터들끼리 부분집합을 구성하고, 각 부분집합에 있는 원소들에 대해서 삽입 정렬을 수행한다.

- 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분집합으로 나누어 정렬하면 **비교와 교환 연산을 감소**시킬 수 있다.
- 셸 정렬에서는 7-정렬, 4-정렬 등의 용어를 주로 사용

- 4-정렬의 예

30	75	15	40	10	65	35	20	90	55	95	25
----	----	----	----	----	----	----	----	----	----	----	----

셸 정렬 (2/3)

- 셸 정렬: 알고리즘

```
shellSort (A[], n)    // A[1, ... , n] 을 정렬
{
    interval ← n;
    while (interval ≥ 1) do
    {
        interval ← interval / 2;
        for (i ← 0; i < interval; i ← i + 1) do
        {
            // interval 간격 만큼의 원소들끼리 셸 정렬 수행
            intervalSort(A[], i, n, interval);
        }
    }
}
```


셸 정렬 (3/3)

- 셸 정렬: 알고리즘 분석

- 메모리 사용 공간

- n 개의 원소에 대하여 n 개의 메모리와 매개변수 h 에 대한 저장공간 사용

- 연산 시간

- 원소 비교 횟수: 처음 원소의 상태에 상관없이 매개변수 h 에 의해 결정
 - 일반적인 시간 복잡도: $O(n^{1.25})$
 - 셸 정렬은 삽입 정렬의 시간 복잡도 $O(n^2)$ 보다 개선된 정렬 방법

고급 정렬 알고리즘



- 기초적인 정렬 알고리즘
- 고급 정렬 알고리즘
 - 퀵 정렬
 - 병합 정렬
- 특수 정렬 알고리즘
- 탐색 알고리즘



퀵 정렬 (1/4)

- **퀵 정렬**(Quick Sort)

- 정렬할 전체 원소에 대해서 정렬을 수행하지 않고, 기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로 분할하여 정렬

- 기준 값: Pivot

- 왼쪽 부분 집합: 기준 값보다 작은 원소들을 이동
 - 오른쪽 부분 집합: 기준 값보다 큰 원소들을 이동

- 퀵 정렬은 다음의 두 가지 기본 작업을 반복 수행

- 분할(Divide)

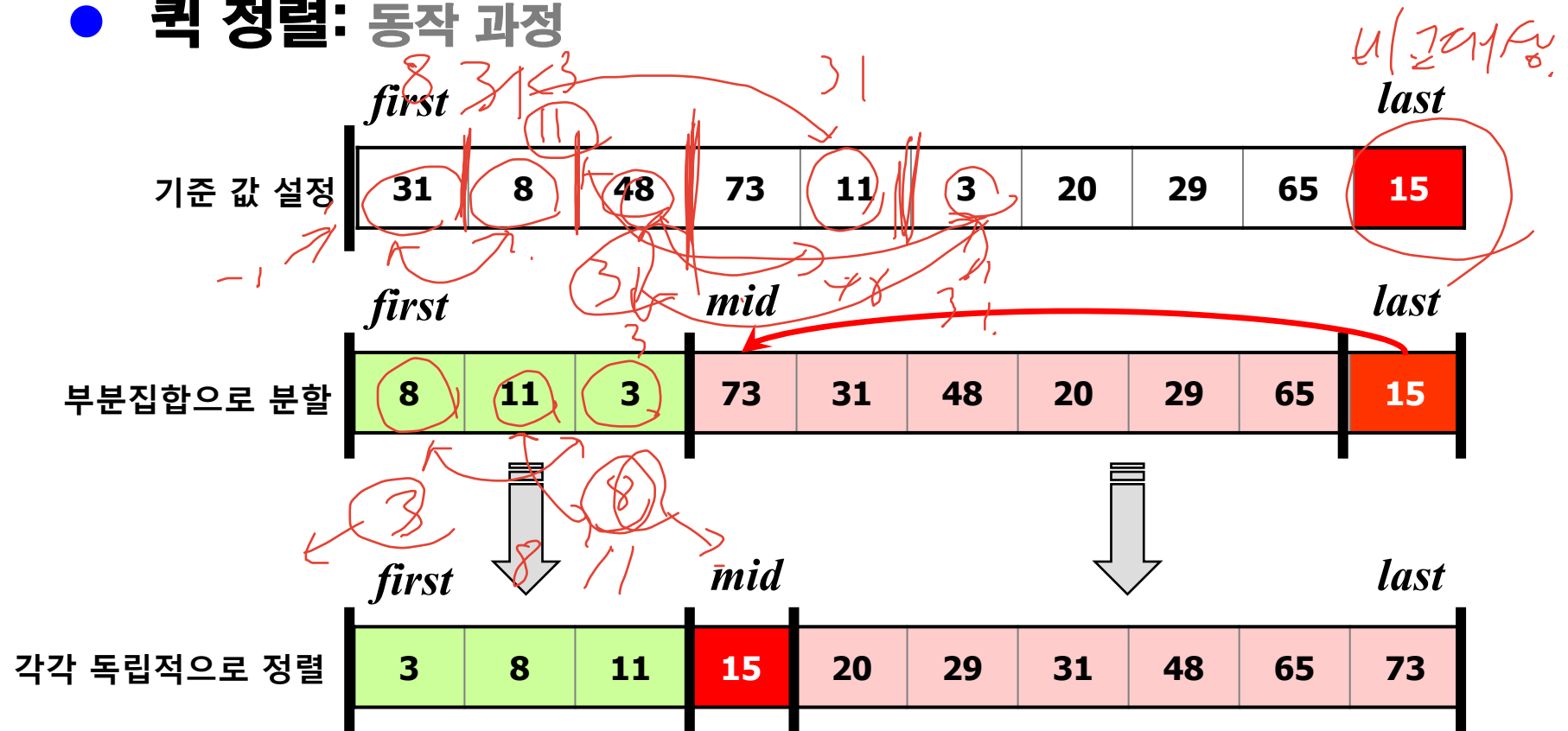
- 정렬할 자료들을 기준 값을 중심으로 두 개의 부분 집합으로 분할

- 정복(Conquer)

- 부분 집합의 원소들 중에서 기준 값보다 작은 원소들은 왼쪽 부분 집합으로, 기준 값보다 큰 원소들은 오른쪽 부분 집합으로 정렬
 - 부분 집합의 크기가 1 이하로 충분히 작지 않으면 순환호출을 이용하여 다시 분할

퀵 정렬 (2/4)

- 퀵 정렬: 동작 과정



평균 수행시간: $O(n \log n)$

최악의 경우 수행시간: $O(n^2)$

퀵 정렬 (3/4)

- 퀵 정렬: 알고리즘

```
Quick_Sort(A[], first, last)    // A[ first , ... , last ] 을 정렬
{
    if (first < last) then
    {
        mid = Partition(A, first, last);    // 분할 후 기준 값의 위치 값을 반환
        Quick_Sort(A, first, mid-1);        // 왼쪽 부분 정렬
        Quick_Sort(A, mid+1, last);         // 오른쪽 부분 정렬
    }
}

Partition(A[], first, last)
{
    pivot ← A[last];    // 마지막 원소를 기준 값으로 선택
    i ← first - 1;
    for j ← first to last - 1
        if (A[j] ≤ pivot) then A[++i] ↔ A[j];
    A[i+1] ↔ A[last];    // 기준 값을 가운데로 위치 시킨다.

    return i + 1;    // 기준 값의 위치 값을 반환
}
```

퀵 정렬 (4/4)

- **퀵 정렬: 알고리즘 분석**

- **메모리 사용공간:** n 개의 원소에 대하여 n 개의 메모리 사용

- **연산 시간**

- **최선의 경우**

- 기준 값에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 $n/2$ 개씩 이등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우

- **최악의 경우**

- 기준 값에 의해 원소들을 분할하였을 때 1 개와 $n-1$ 개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우

n^2

- **평균 시간 복잡도: $O(n \log n)$**

- 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법

고급 정렬 알고리즘

병합 정렬



병합 정렬 (1/4)

- **병합 정렬**(Merge Sort)

- 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법

- 병합 정렬 방법의 종류

- 2-way 병합: 2 개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 방법
- n-way 병합: n 개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 방법

// 2-way 병합 정렬: 세 가지 기본 작업을 반복 수행

1) 분할(Divide): 입력 자료를 같은 크기의 부분집합 2개로 분할한다.

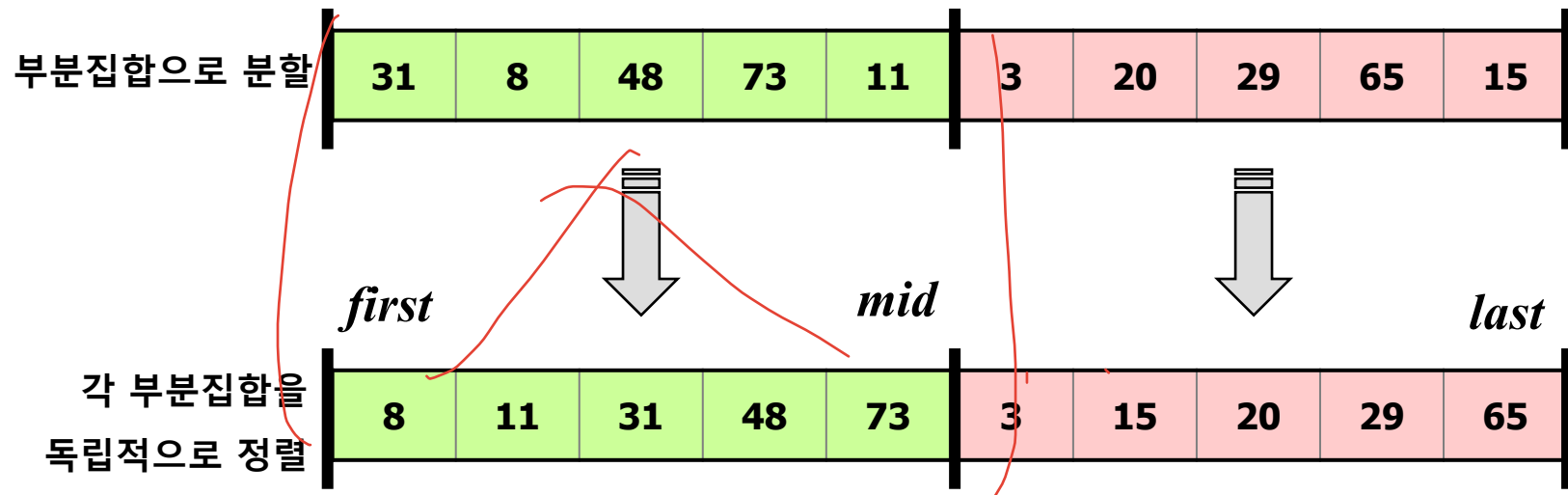
2) 정복(Conquer): 부분집합의 원소들을 정렬한다.

만약 부분집합의 크기가 충분히 작지 않으면, 순환호출을 이용하여 다시 분할 정복 기법을 적용한다.

3) 결합(Combine): 정렬된 부분집합들을 하나의 집합으로 통합한다.

병합 정렬 (2/4)

- 병합 정렬: 동작 과정



정렬된 두 부분집합을 병합

3	8	11	15	20	29	31	48	65	73
---	---	----	----	----	----	----	----	----	----

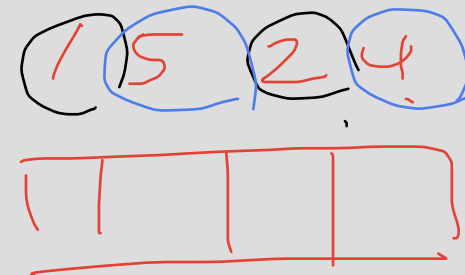
병합 정렬 (3/4)

● 병합 정렬: 알고리즘

```
mergeSort(A[ ], first, last)      // A[first , ... , last]을 정렬
{
    if (first < last) then
    {
        mid ← (first+last)/2 ;      // first와 last 사이의 중간 원소의 위치
        mergeSort(A, first, mid);  // 왼쪽 부분집합 정렬
        mergeSort(A, mid+1, last); // 오른쪽 부분집합 정렬
        merge(A, first, mid, last); // 정렬된 두 부분집합 병합
    }
}

merge(A[ ], first, mid, last)
// A[first ... mid]와 A[mid+1 ... last]를 병합하여 A[first ... last]을 정렬된 상태로 재구성
// 단, A[first ... mid]와 A[mid+1 ... last]는 이미 정렬 부분집합이다.
{
    i ← first; j ← mid+1; t ← 1;
    while (i ≤ mid and j ≤ last) {
        if (A[i] ≤ A[j]) then temp[t++] ← A[i++];
        else temp[t++] ← A[j++];
    }
    while (i ≤ mid) temp[t++] ← A[i++];
    while (j ≤ last) temp[t++] ← A[j++];

    // 정렬된 상태로 재구성된 temp 배열을 원본 배열 A 에 복사
    i ← p; t ← 1;
    while (i ≤ last) A[i++] ← temp[t++];
}
```



병합 정렬 (4/4)

- 병합 정렬: 알고리즘 분석

- 메모리 사용공간

- 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
- (원소 n 개에 대해서 $(2 * n)$ 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: $O(n \log_2 n)$

특수 정렬 알고리즘



- 기초적인 정렬 알고리즘
- 고급 정렬 알고리즘
- **특수 정렬 알고리즘**
 - 계수 정렬
 - 기수 정렬
 - 버킷 정렬
- 탐색 알고리즘



특수 정렬 알고리즘

- 특수 정렬 알고리즘

- 비교 정렬

- 두 원소를 비교하는 정렬의 하한선은 $\Omega(n \log n)$

“최악의 경우 정렬 시간이 $O(n \log n)$ 보다 더 빠를 수는 없는가?”

- 그러나 원소들이 특수한 성질을 만족하면 $O(n)$ 정렬도 가능하다.
 - 계수 정렬(Counting Sort): 원소들의 크기가 모두 $-O(n) \sim O(n)$ 범위에 있을 때...
 - 기수 정렬(Radix Sort): 원소들이 모두 k 이하의 자리 수를 가졌을 때(k : 상수)
 - 버킷 정렬(Bucket Sort): 원소들이 균등 분포(Uniform distribution)를 이룰 때...

특수 정렬 알고리즘

계수 정렬



계수 정렬 (1/3)

● 계수 정렬(Counting Sort)

- 항목들의 순서를 결정하기 위해 집합에 각 항목이 몇 개씩 있는지 세는 작업을 하면서 선형 시간에 정렬하는 효율적인 알고리즘
 - 속도가 빠르며 안정적이다.
 - 제한 사항
 - 정수나 정수로 표현할 수 있는 자료에 대해서만 동작
 - 카운트들을 위한 충분한 공간을 할당하려면 집합 내의 가장 큰 정수를 알아야 한다.

```
countingSort(A[], B[], n)
// A[1...n]: 입력 배열
// B[1...n]: 배열 A 를 정렬한 결과
{
    for i ← 1 to k      C[i] ← 0;
    for j ← 1 to n      C[A[j]]++; // 이 시점에서 C[i] : 값이 i 원소의 총 수
    for i ← 1 to k      C[i] ← C[i] + C[i-1]; // 누적 합 계산
    // 이 시점에서 C[i] : i 보다 작거나 같은 원소의 총 개수
    for j ← n downto 1 {
        B[C[ A[j]] ] ← A[j];
        C[A[j]]--;
    }
}
```

계수 정렬 (2/2)

● 계수 정렬: 동작과정

○ 1단계

- ① data에서 각 항목들의 발생 횟수를 센다.
- ② 발생 횟수들은 정수 항목들로 직접 인덱스 되는 카운트 배열(counts)에 저장한다.

처음의 정렬되지 않은 집합

data

0	4	1	3	1	2	4	1
---	---	---	---	---	---	---	---

data의 각 정수의 발생 횟수 counts

1	3	1	1	2
---	---	---	---	---

$1+3$ $1+3+1$ $1+3+1+1$ $1+3+1+1+1$
 counts[0] counts[1] counts[2] counts[3] counts[4]

1	4	5	6	8
---	---	---	---	---

counts

정렬된 집합에서 각 항목의 앞에 위치할 항목의 개수를 반영하기 위하여 카운트들을 조정한다.

○ 2단계: 정렬된 집합

↓ j = 0

↓ j = 7

temp

0	1	1	1	2	3	4	4
---	---	---	---	---	---	---	---

[0] [1] [2] [3] [4] [5] [6] [7]

40

특수 정렬 알고리즘

기수 정렬



기수 정렬 (1/3)

- **기수 정렬**(Radix Sort)

- 입력이 모두 k 이하의 자리 수를 가진 특수한 경우에(자연수가 아닌 제한된 종류를 가진 알파벳 등도 해당) 사용할 수 있는 방법

- $O(n)$ 시간이 소요되는 알고리즘

```
radixSort(A[ ], n, k)
```

```
// 원소들이 각각 최대 k 자리수인 A[1...n]을 정렬한다
```

```
// 가장 낮은 자리 수를 1번째 자리수라 한다
```

```
{
```

```
    for i ← 1 to k
```

```
        i 번째 자리 수에 대해 A[1...n] 을 안정을 유지하면서 정렬한다;
```

```
}
```

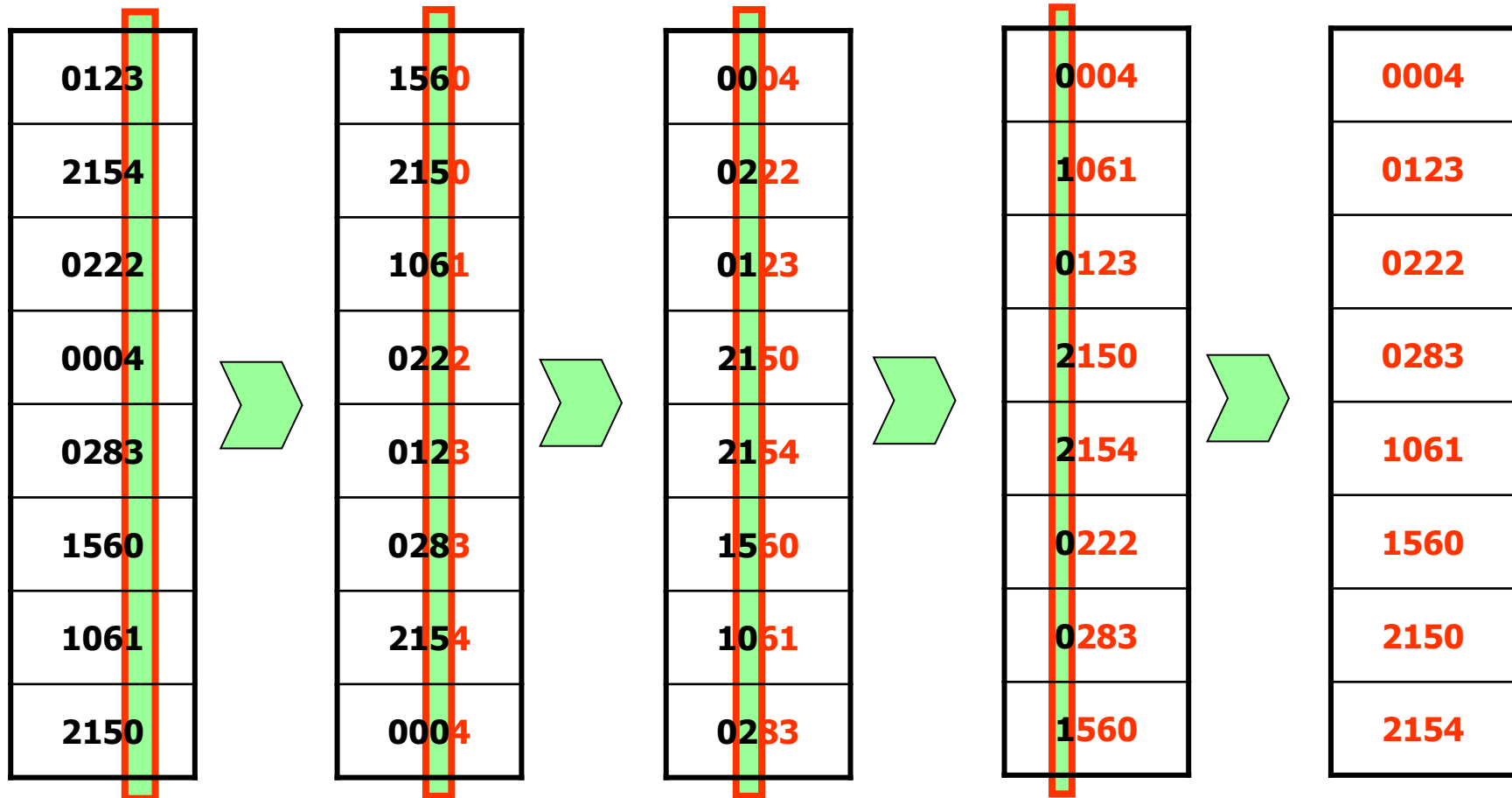
- **안정성 정렬**(Stable Sort)

- 같은 값을 가진 원소들은 정렬 후에도 원래의 순서가 유지되는 성질을 가진 정렬을 일컫는다.

기수 정렬 (2/3)

- 기수 정렬: 동작 과정

자릿수에 따라 정렬하는 과정 //



기수 정렬 (3/3)

- 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)와 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: $d(n+r)$
 - 시간 복잡도: $O(d(n+r))$

특수 정렬 알고리즘

버킷 정렬



버킷 정렬 (1/3)

- 버킷 정렬(Bucket Sort)

- 원소들이 균등 분포(Uniform distribution)를 하는 $[0, 1)$ 범위의 실수인 경우
 - $[0, 1)$ 범위는 아니어도 쉽게 $[0, 1)$ 범위로 변환할 수 있다

```
bucketSort(A[ ], n)
```

```
// A[1...n] :  $[0, 1)$  범위의 균등 분포를 한 실수 값 리스트
```

```
{
```

```
    for i ← 1 to n
```

```
        A[i] 를 리스트 B[n*A[i]] 에 삽입한다;           // B[0,..., n-1]: 각각이 리스트인 리스트
```

```
    for i ← 1 to n
```

```
        리스트 B[i] 에 있는 원소들을 정렬;           // 삽입 정렬이면 충분하다.
```

```
    B[0], B[1], ... , B[n-1] 의 원소들을 차례대로 A[0, ... , n-1] 로 복사한다;
```

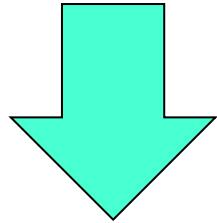
```
}
```

버킷 정렬 (2/3)

- 버킷 정렬: 동작 과정

(a) $A[0...14]$: 정렬할 배열

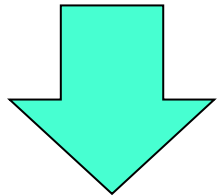
.38	.94	.48	.73	.99	.43	.55	.15	.85	.84	.81	.71	.17	.10	.02
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



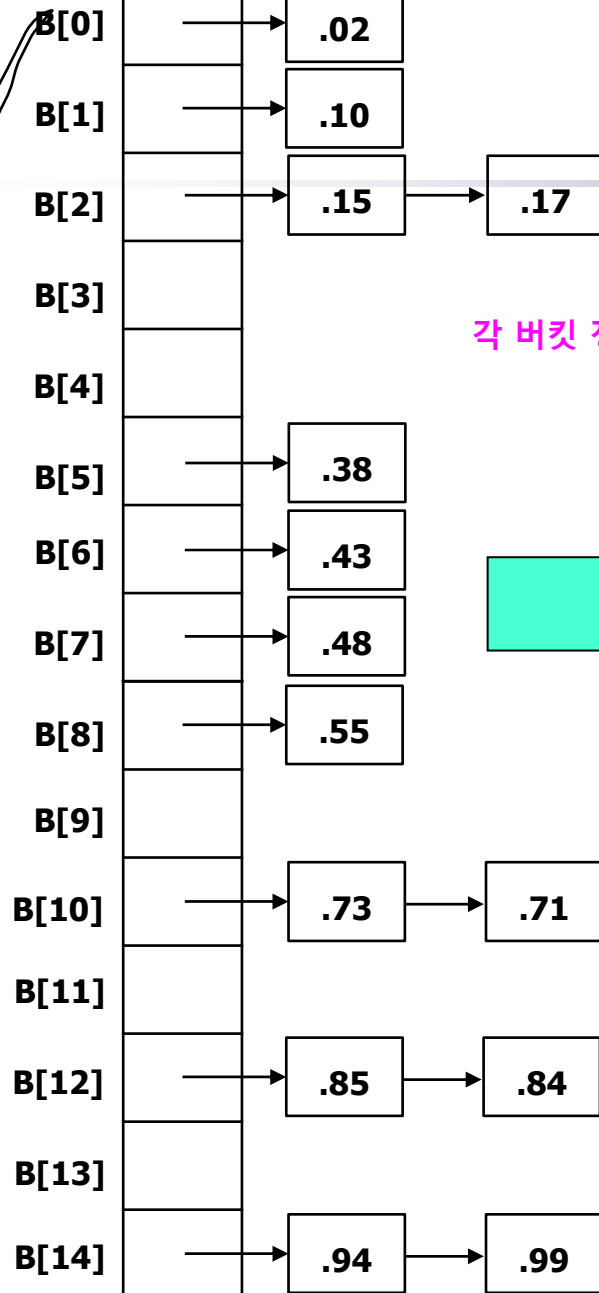
$A[0...14]$ 각각에 15를 곱하여 정수부만 취함.

(b) 버킷 리스트 위치

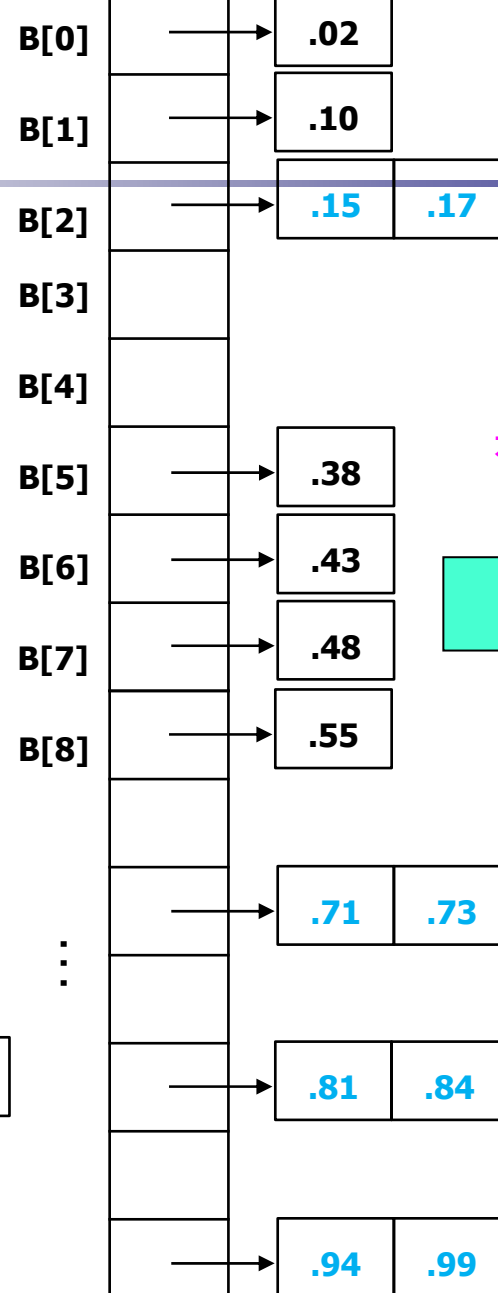
5	14	7	10	14	6	8	2	12	12	12	10	2	1	0
---	----	---	----	----	---	---	---	----	----	----	----	---	---	---



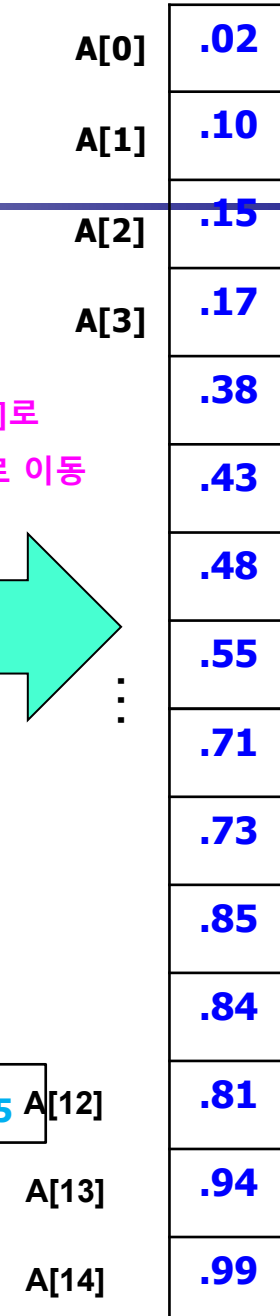
(c)



(d)



(e)



각 버킷 정렬

A[]로
차례로 이동

탐 색



- 기초적인 정렬 알고리즘
- 고급 정렬 알고리즘
- 특수 정렬 알고리즘
- **탐색 알고리즘**
 - 순차 탐색
 - 이진 탐색



탐 색

- **탐색(Search)**

- 레코드의 집합에서 주어진 키를 지닌 레코드를 찾는 작업 탐색

- 주어진 키 값: **목표 키(target key)** 또는 **탐색 키(search key)**

- 탐색의 분류

- 수행되는 위치에 따른 분류: **내부 탐색, 외부 탐색**

- 검색 방법에 따른 분류

- **비교 탐색:** 검색 대상의 키를 비교하여 탐색
 - » 순차 탐색, 이진 탐색, 트리 탐색
- **계산 탐색:** 계수적 성질을 이용한 계산으로 탐색
 - » 해싱

탐색

순차 탐색



순차 탐색 (1/3)

- 순차 탐색(Sequential Search)

- 선형 탐색(Linear Search)

- 순차 탐색 알고리즘

- 목표치를 찾기 위해 리스트의 처음부터 탐색을 시작해서, 목표치를 찾거나 리스트에 목표치가 없다는 것이 밝혀질 때까지 탐색을 계속한다.
 - 순차 탐색은 순서가 없는 리스트일 때 사용
 - 순차 탐색은 리스트가 작거나, 가끔 한번씩 탐색할 경우에만 사용

```
sequentialSearch( A[ ], n, key )
{
    i ← 0;
    while ( i < n )
    {
        if ( A[i] = key ) then
            return i;
        i ← i + 1;
    }
    return -1;
}
```

순차 탐색 (2/3)

- 순차 탐색: 동작 과정 -- 탐색 성공

목표 데이터: 73

- 순서 없는 리스트에 위치한 데이터

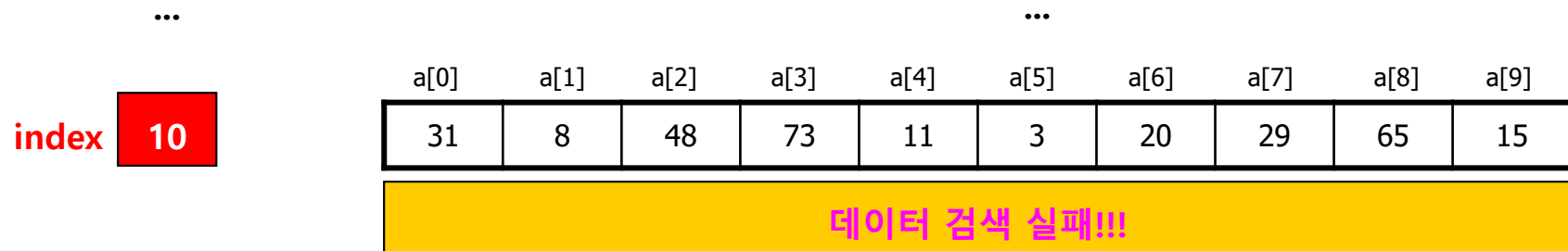
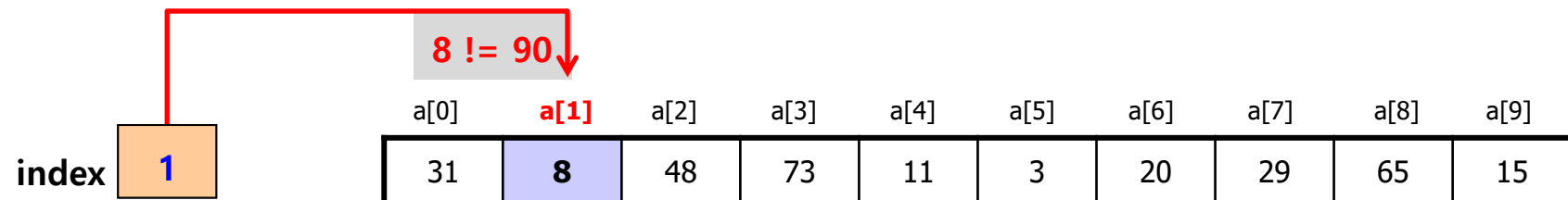
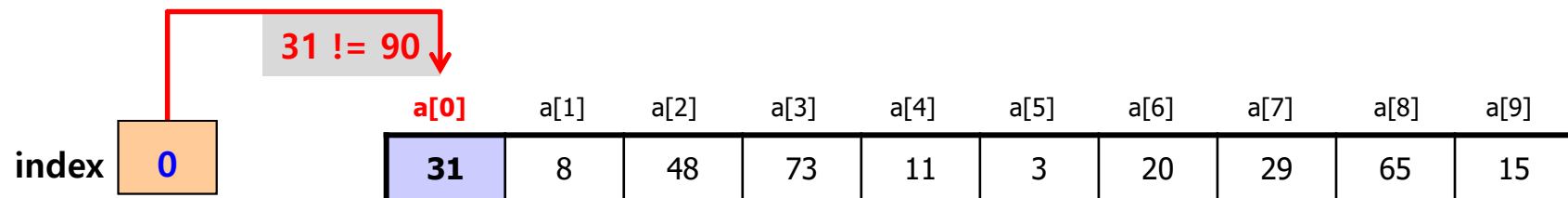


순차 탐색 (3/3)

- 순차 탐색: 동작 과정 -- 탐색 실패

목표 데이터: 90

- 순서 없는 리스트에 탐색 실패



탐색

이진 탐색



이진 탐색 (2/3)

- **이진 탐색(Binary Search)**

- 이진 탐색은 배열이 정렬되어 있을 때 효율적인 알고리즘
 - 순차 탐색은 매우 느리다.
- 이진 탐색 알고리즘의 조건
 - 탐색할 데이터들은 정렬된 상태 이다.
 - 주어진 데이터들은 유일한 키 값을 가지고 있다.

```
binarySearch(A[ ], first, last, key)
{
    if (first > last) then
        return -1;

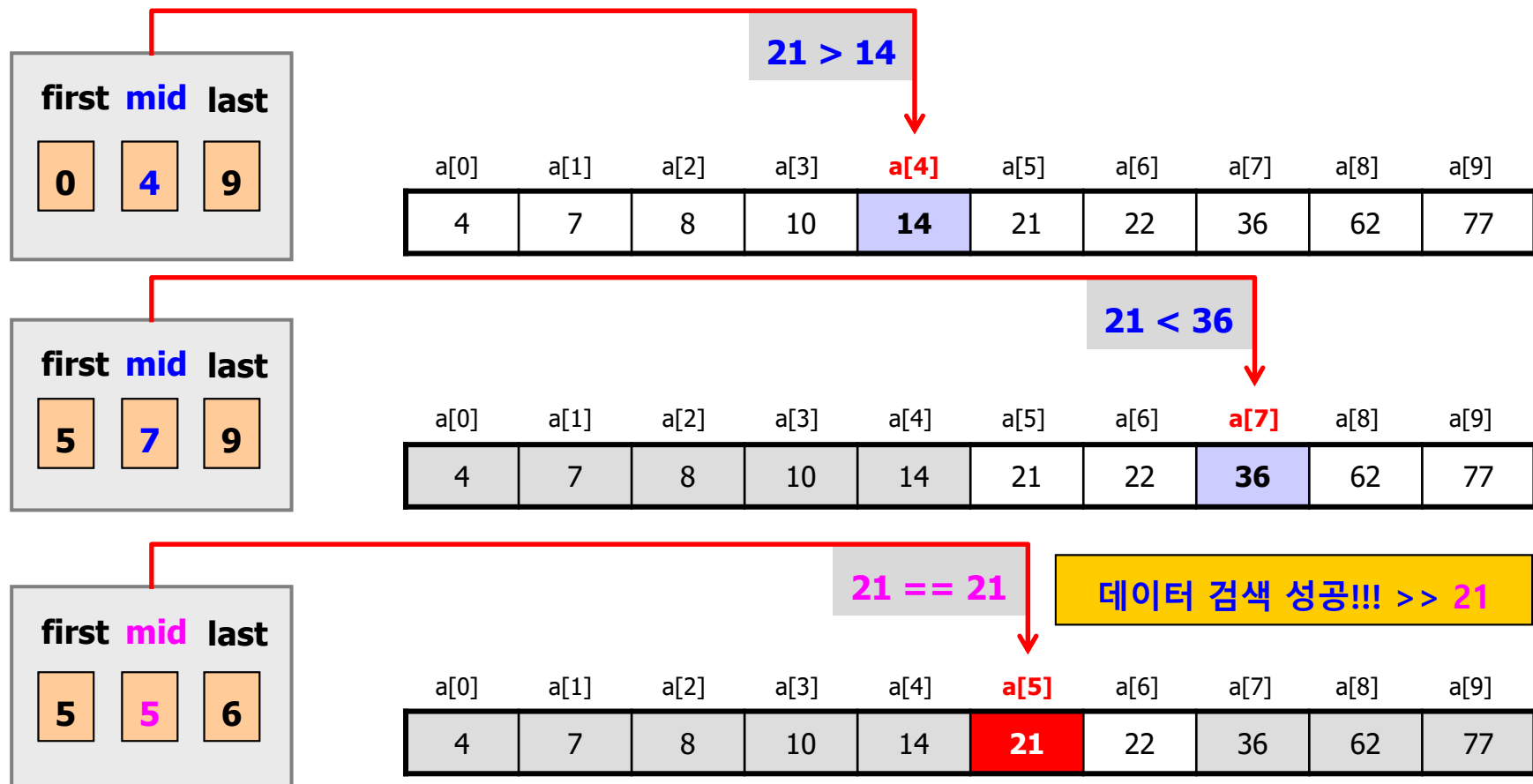
    // 검색 범위의 중간 원소의 위치 값 계산
    mid ← (first + last) / 2;

    if (key = A[mid])      return mid;
    else if (key < A[mid]) then index ← binarySearch(A[], first, mid-1, key);
    else if (key > A[mid]) then index ← binarySearch(A[], mid+1, last, key);
    return index;
}
```


이진 탐색 (2/3)

- 이진 탐색: 동작 과정 -- 탐색 성공

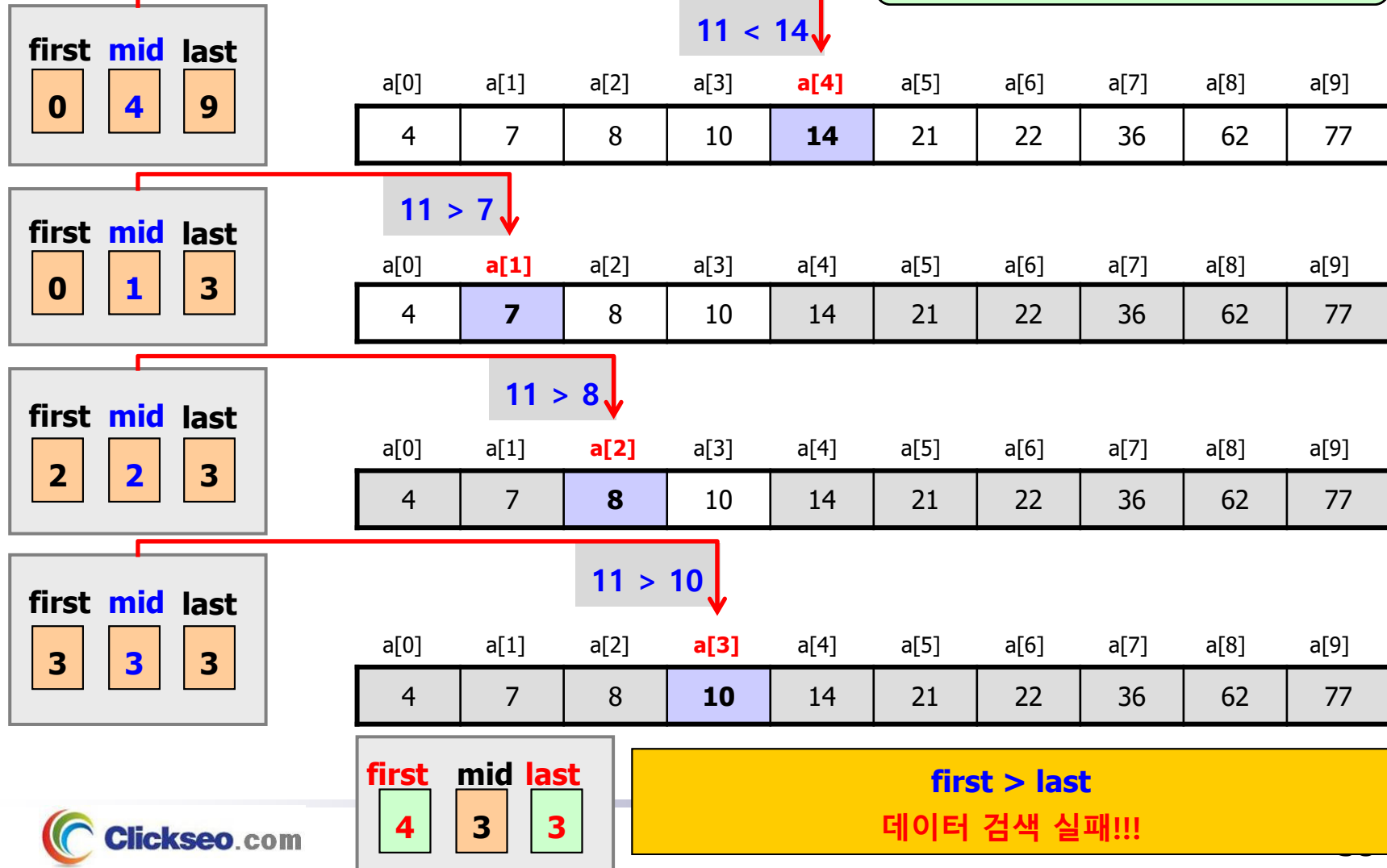
검색 데이터: 21



이진 탐색 (3/3)

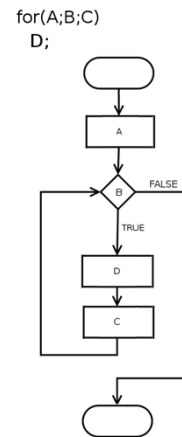
● 이진 탐색: 동작 과정 -- 탐색 실패

검색 데이터: 11



참고문헌

- [1] Michael T. Goodrich 외 2인 지음, 김유성 외 2인 옮김, "C++로 구현하는 자료구조와 알고리즘", 한티에듀, 2020.
- [2] "프로그래밍 대회 공략을 위한 알고리즘과 자료 구조 입문", 와타노베 유타카 저, 윤인성 역, 인사이트, 2021.
- [3] "IT CookBook, 쉽게 배우는 자료구조 with 파이썬", 문병로, 한빛아카데미, 2022.
- [4] "이것이 취업을 위한 코딩 테스트다 with 파이썬", 나동빈, 한빛미디어, 2020.
- [5] 문병로, "IT CookBook, 쉽게 배우는 알고리즘: 관계 중심의 사고법"(개정판), 개정판, 한빛아카데미, 2018.
- [6] Richard E. Neapolitan, 도경구 역, "알고리즘 기초", 도서출판 홍릉, 2017.
- [7] 주우석, "IT CookBook, C · C++ 로 배우는 자료구조론", 한빛아카데미, 2019.
- [8] 이지영, "C 로 배우는 쉬운 자료구조", 한빛아카데미, 2022.



이 강의자료는 저작권법에 따라 보호받는 저작물이므로 무단 전제와 무단 복제를 금지하며,
내용의 전부 또는 일부를 이용하려면 반드시 저작권자의 서면 동의를 받아야 합니다.

Copyright © Clickseo.com. All rights reserved.

