ALGORITHM
알고리즘

Lecture 03

정렬 (1)

컴퓨터과학과 | 이관용 교수

한국방송통신대학교

학습목차

01. 기본 개념

정렬?

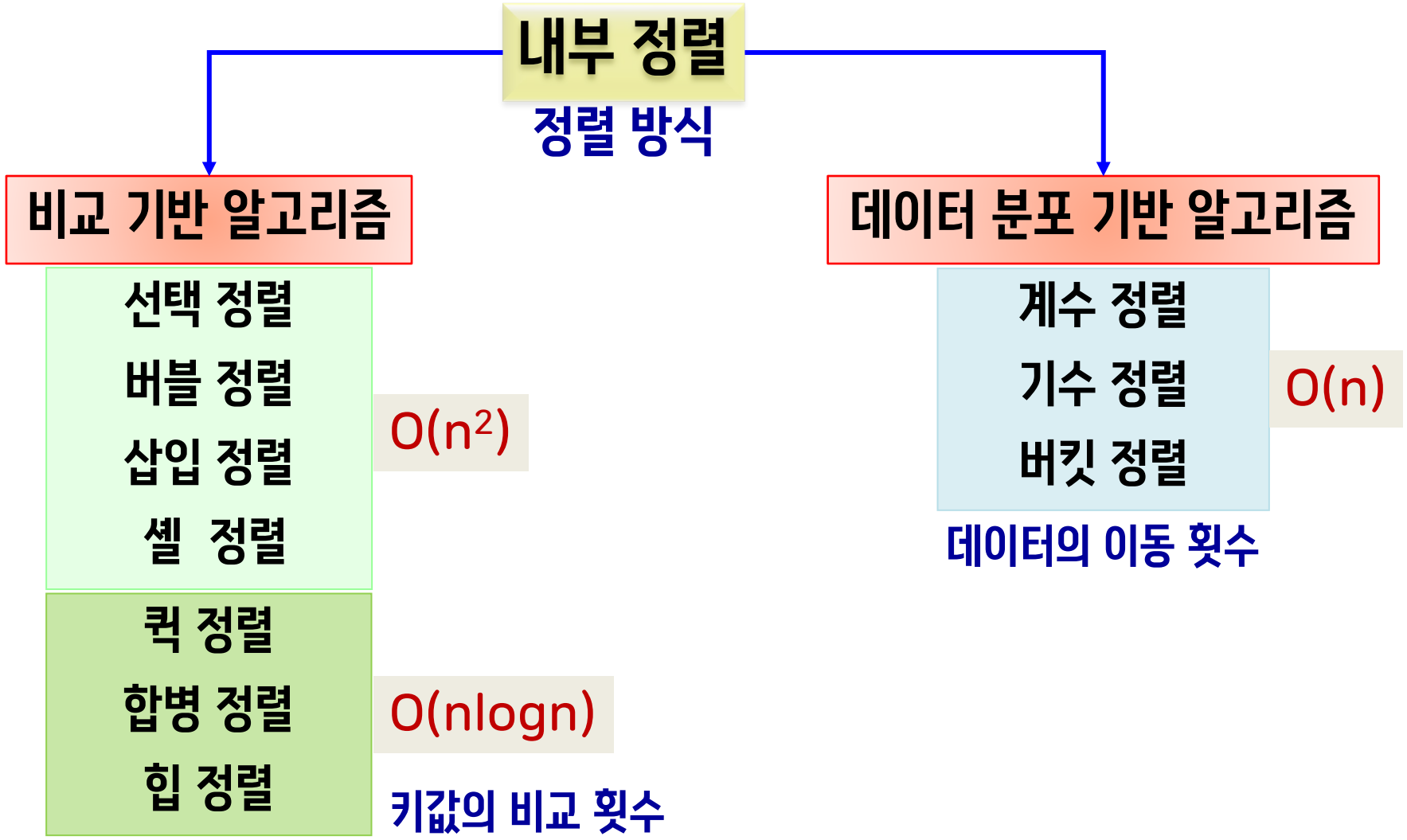
주어진 데이터를 값의 크기 순서에 따라 재배치하는 것

- 오름차순, 내림차순

정렬 구분 → "정렬 수행 시점에 데이터가 어디에 저장되어 있는가?"

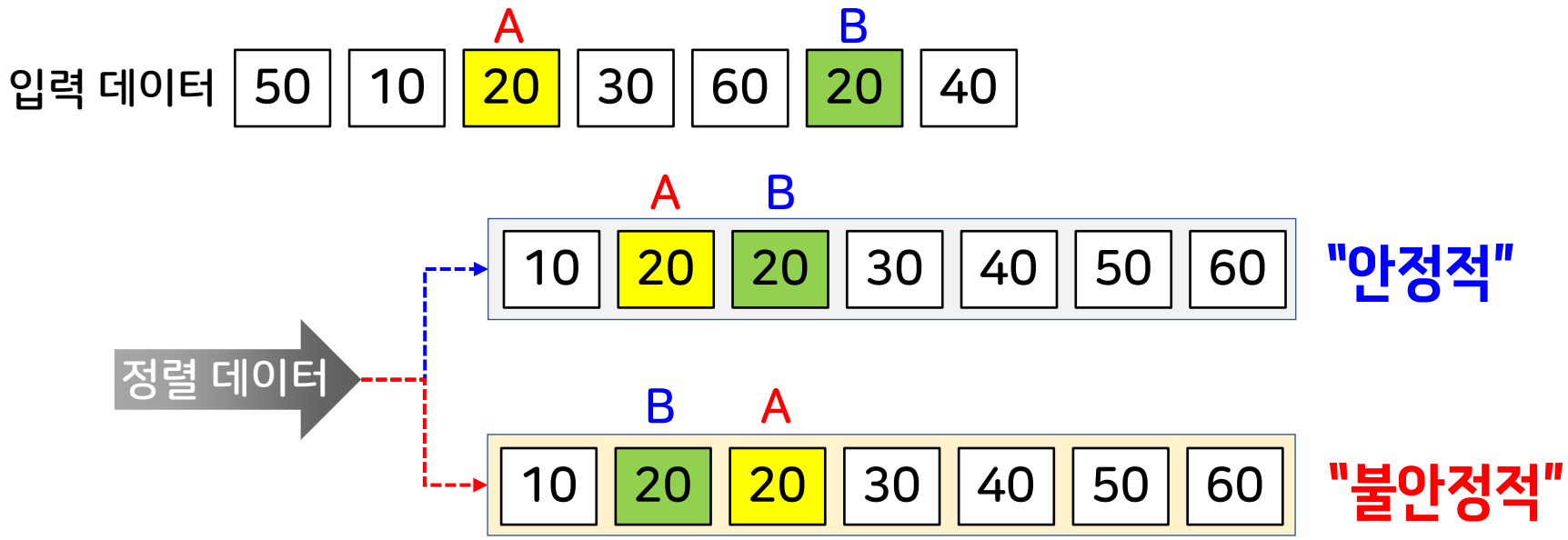
- 내부 정렬
  - ✓ 전체 데이터를 주기억장치에 저장한 후 정렬을 수행하는 방식
- 외부 정렬
  - ✓ 모든 데이터를 주기억장치에 저장할 수 없는 경우,
    모든 데이터를 보조기억장치에 저장해 두고
    그중 일부 데이터만을 반복적으로 주기억장치로 옮겨와서 정렬을 수행하는 방식

내부 정렬 알고리즘


내부 정렬
정렬 방식
비교 기반 알고리즘
선택 정렬
버블 정렬
삽입 정렬
셸 정렬
O(n²)
퀵 정렬
합병 정렬
힙 정렬
O(nlogn)
키값의 비교 횟수
데이터 분포 기반 알고리즘
계수 정렬
기수 정렬
버킷 정렬
O(n)
데이터의 이동 횟수

정렬 관련 개념

안정적 stable 정렬



- 동일한 값을 갖는 데이터가 여러 개 있을 때
  정렬 전의 상대적 위치가 정렬 후에도 그대로 유지되는 정렬

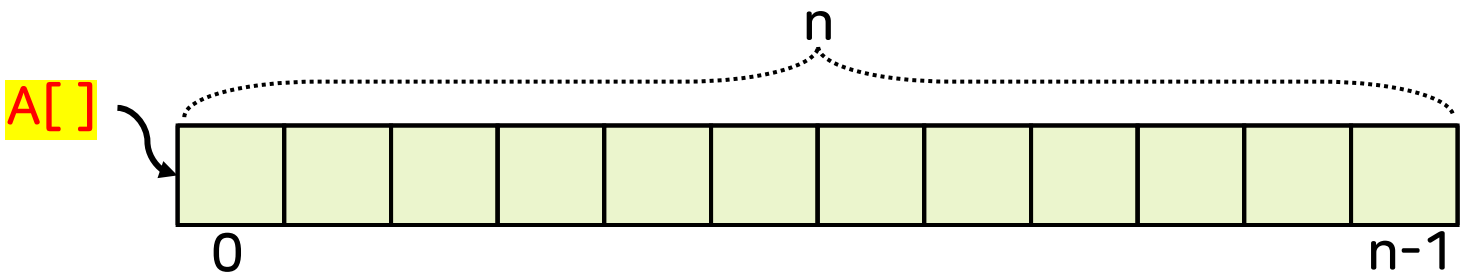
정렬 관련 개념

제자리 in-place 정렬

- 입력 배열 이외에 별도로 필요한 저장 공간이 상수 개를 넘지 않는 정렬

  ✓ 입력 크기 n이 증가함에도 불구하고 추가적인 저장 공간은 증가하지 않음

정렬 알고리즘의 기본 가정



A[i] > 0, (0≤i≤n-1)

정렬 결과 → if i < j then A[i]≤A[j], (0≤ i, j ≤n-1)

✓ 입력 배열 → A[0..n-1]

✓ 입력 크기 → n

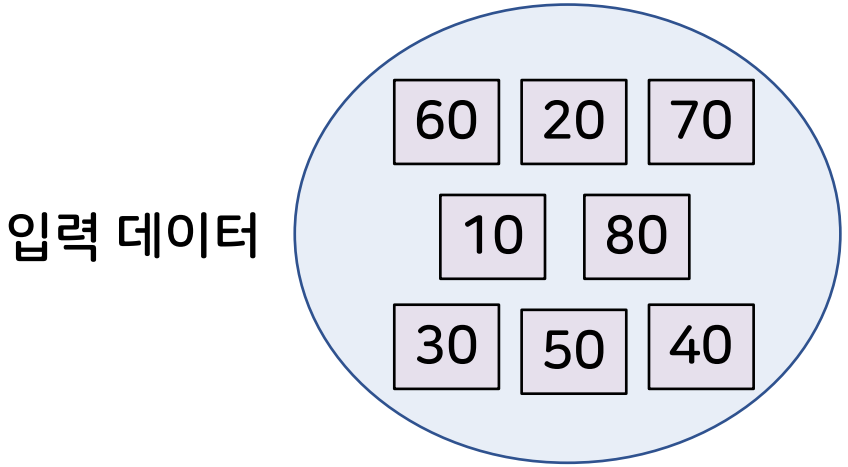
✓ 입력 데이터 → 양의 정수

✓ 정렬 방식 → 오름차순

02. 선택 정렬

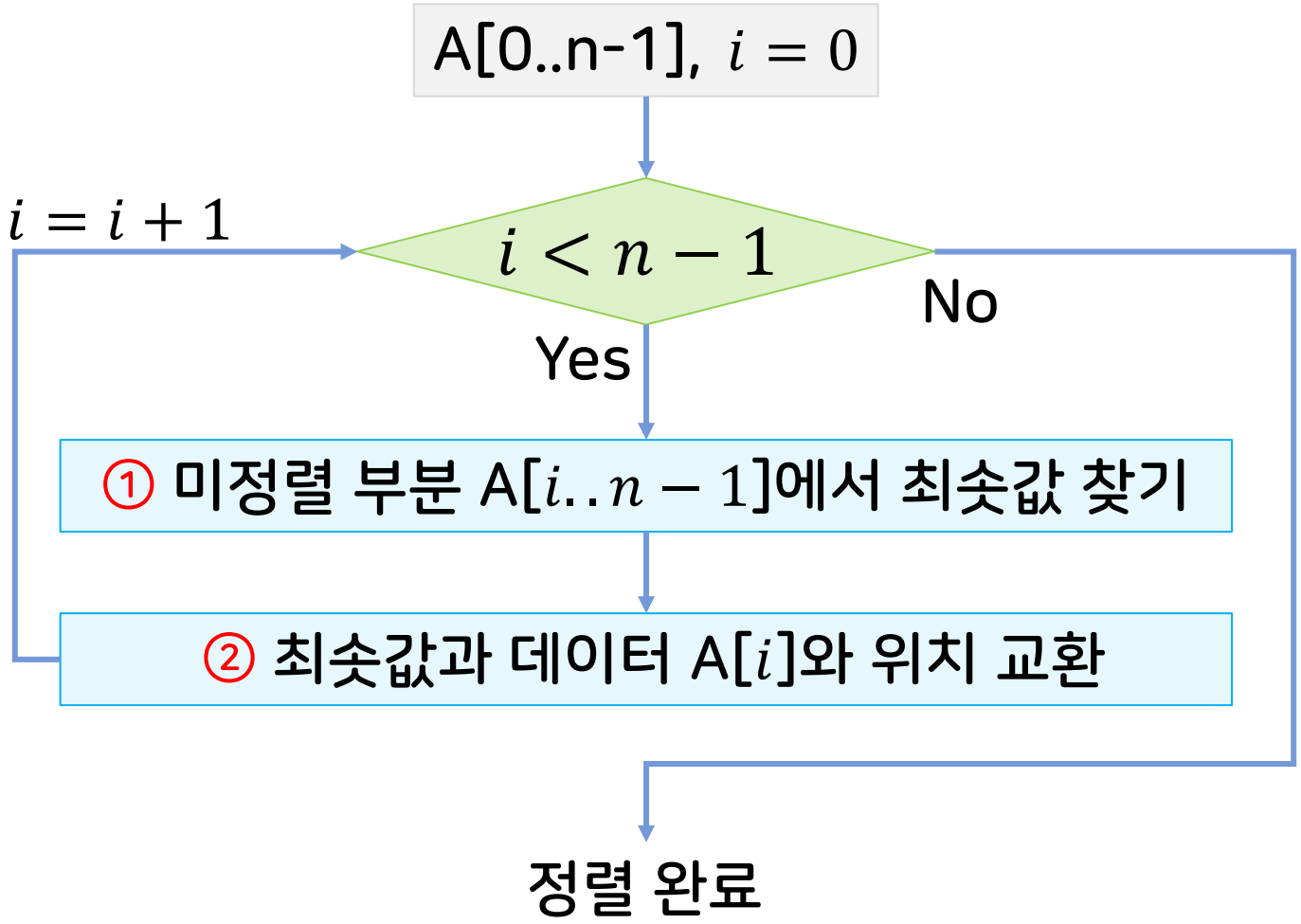
선택 정렬?

입력 배열에서 가장 작은 값부터 순서대로 '선택'해서 나열하는 방식



정렬 결과

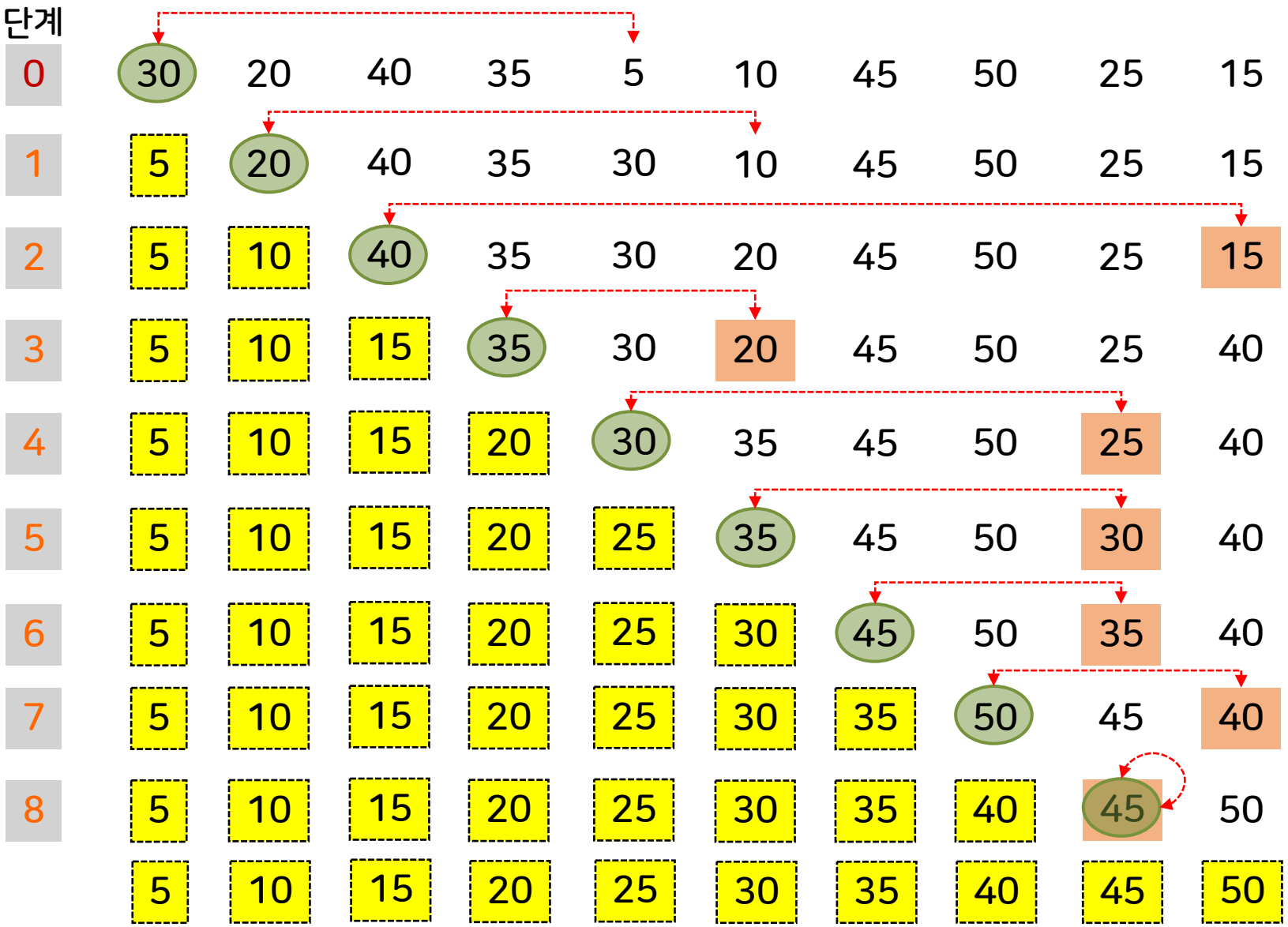
선택 정렬의 처리 과정


A[0..n-1], $i = 0$
$i < n - 1$
No
Yes
① 미정렬 부분 A[$i..n-1$]에서 최솟값 찾기
② 최솟값과 데이터 A[i]와 위치 교환
$i = i + 1$
정렬 완료

선택 정렬 알고리즘


```
SelectionSort (A[ ], n)
{
    for (i=0; i < n-1; i++) {          // (n-1)번 반복
      min = i;
      for (j=i+1; j < n; j++)          // ① A[i..n-1]에서 최솟값 찾기
        if (A[min] > A[j])
          min = j;
      A[i]와 A[min]의 자리바꿈;        // ② 최소값과 A[i]의 위치 교환
    }
    return (A);
}
```

선택 정렬의 예


단계
0 30 20 40 35 5 10 45 50 25 15
1 5 20 40 35 30 10 45 50 25 15
2 5 10 40 35 30 20 45 50 25 15
3 5 10 15 35 30 20 45 50 25 40
4 5 10 15 20 30 35 45 50 25 40
5 5 10 15 20 25 35 45 50 30 40
6 5 10 15 20 25 30 45 50 35 40
7 5 10 15 20 25 30 35 50 45 40
8 5 10 15 20 25 30 35 40 45 50
5 10 15 20 25 30 35 40 45 50

성능과 특징


```
SelectionSort (A[ ], n)
{
    for (i=0; i < n-1; i++) {
      min = i;
      for (j=i+1; j < n; j++)
        if (A[min] > A[j])
          min = j;
      A[i]와 A[min]의 자리바꿈;
    }
    return (A);
}
```

루프 i	0	1	2	…	n-2
루프 j의 비교 횟수	n-1	n-2	n-3	…	1

총 비교 횟수

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

$O(n^2)$

성능과 특징

입력 데이터의 순서에 민감하지 않음

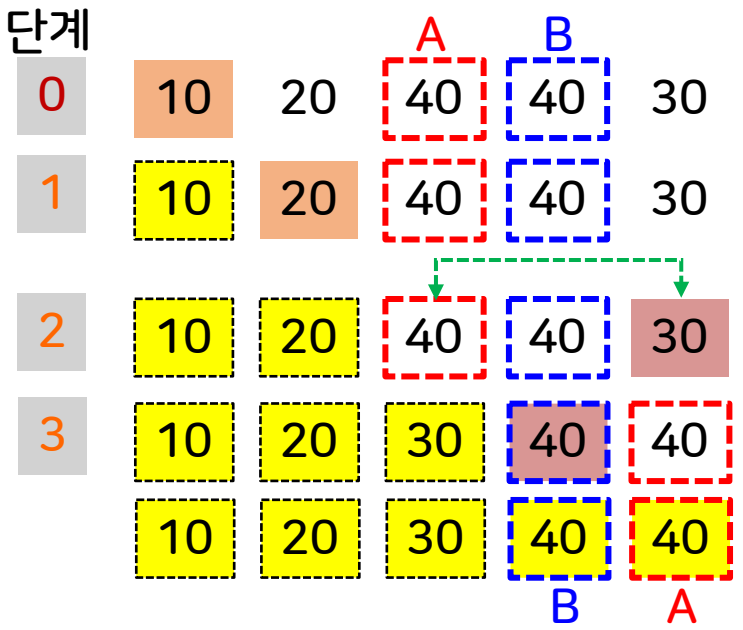
- 최소값 찾기 → 미정렬 부분 A[i..n-1]에서 항상 (n-1)-i번의 비교가 필요

→ 입력 데이터의 상태와 상관없이 항상 동일한 성능 $O(n^2)$을 가짐

제자리 정렬 알고리즘

- 입력 배열 A[ ] 이외에 상수 개의 저장 공간(예: i, j, min, tmp)만 필요

안정적이지 않은 정렬 알고리즘



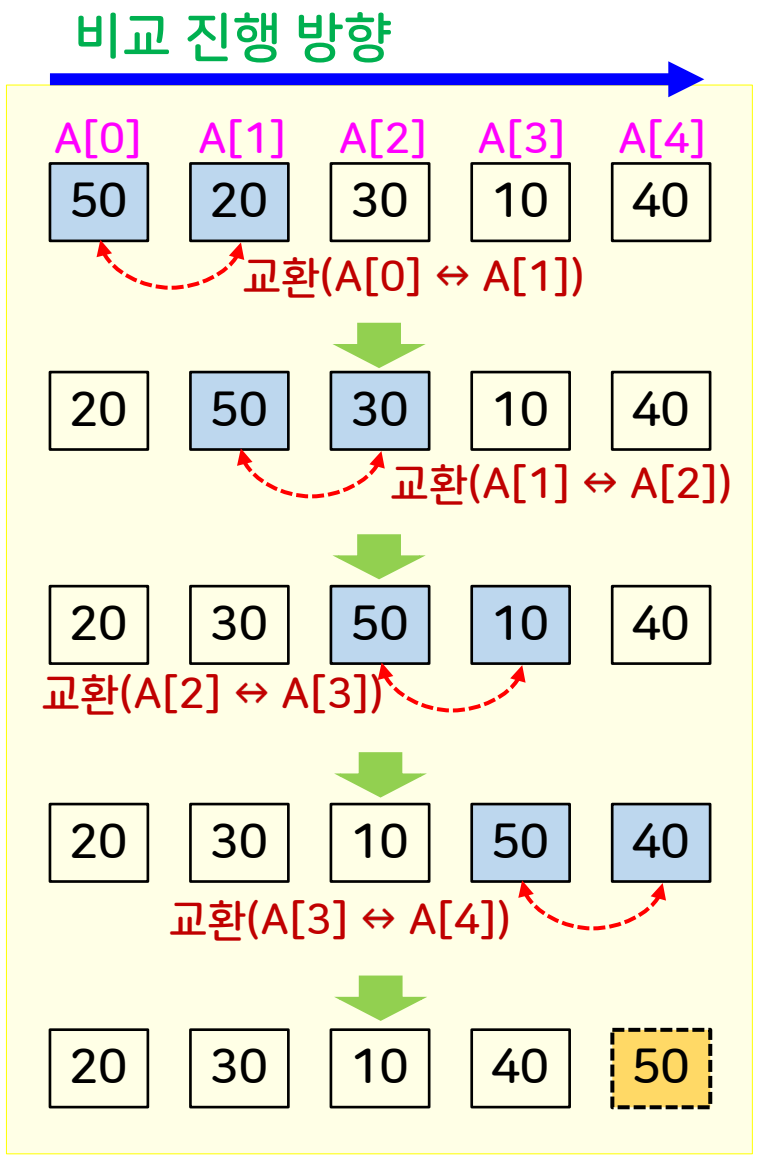
03. 버블 정렬

버블 정렬?

모든 인접한 두 데이터를 차례대로 비교해서
왼쪽 데이터가 더 큰 경우에는 오른쪽 데이터와
자리를 바꾸는 과정을 반복해서 정렬을 수행하는 방식

- 비교를 진행하는 방향
  - ✓ 왼쪽에서 오른쪽으로 진행
    - → 가장 큰 값부터 찾아서 오른쪽 끝에서부터 위치시킴
  - ✓ 오른쪽에서 왼쪽으로 진행
    - → 가장 작은 값부터 찾아서 왼쪽 끝에서부터 위치시킴

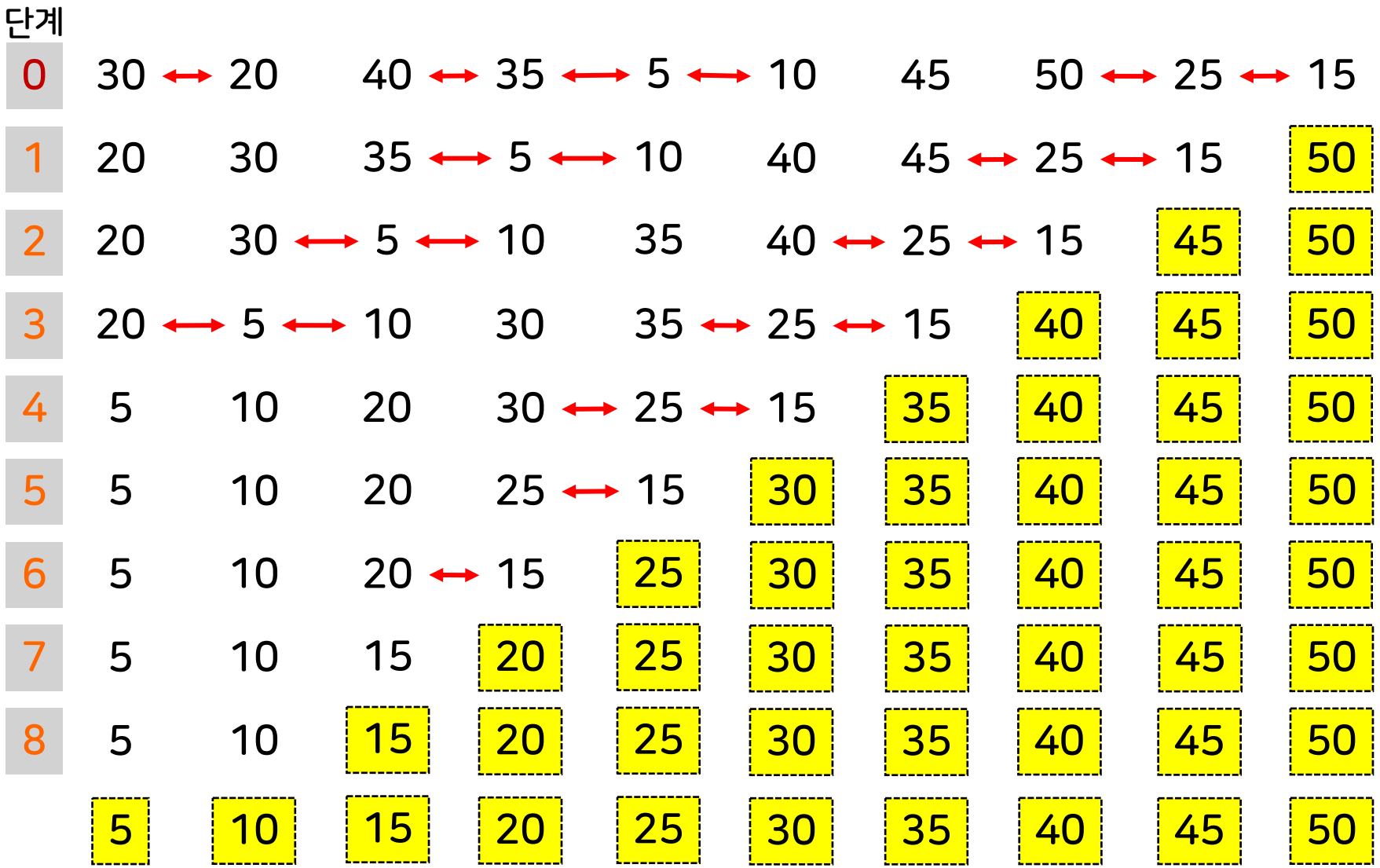
버블 정렬의 비교 진행 방향



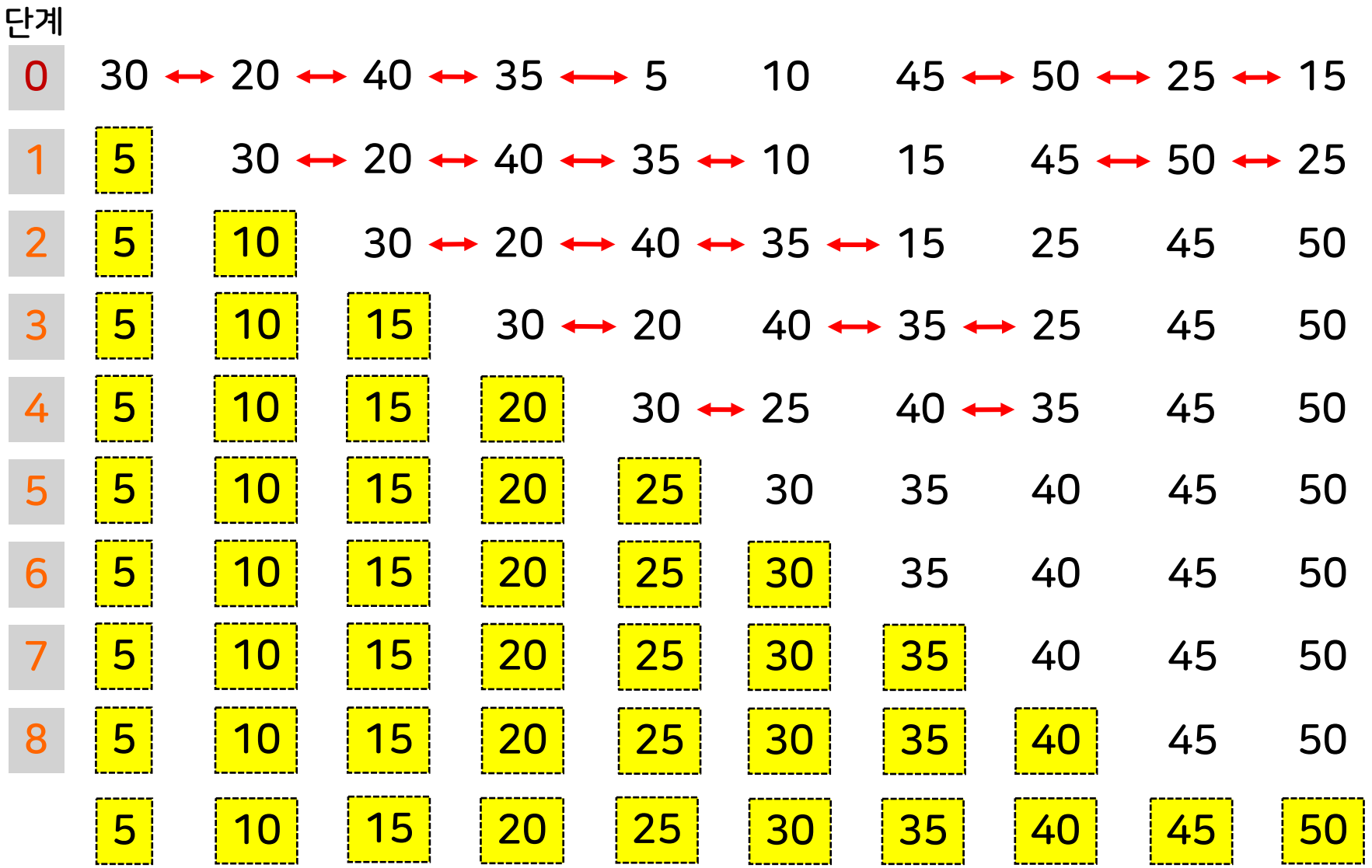
기본 형태의 버블 정렬 알고리즘


```
BubbleSort (A[ ], n)
{
   for (i=0; i < n-1; i++)      // 단계: (n-1)번 반복
     for (j=0; j < n-1; j++)    // 왼쪽에서 오른쪽으로 진행하는 경우
       if (A[j] > A[j+1])       // '왼쪽 데이터 > 오른쪽 데이터'이면
           A[j]와 A[j+1]의 자리바꿈;
   return (A);
}
```

버블 정렬의 예_1 (왼쪽 → 오른쪽)



버블 정렬의 예_2 (오른쪽 → 왼쪽)



성능과 특징


```
BubbleSort (A[ ], n)
{
   for (i=0; i < n-1; i++)
     for (j=0; j < n-1; j++)
        if (A[j] > A[j+1])
            A[j]와 A[j+1]의 자리바꿈;
   return (A);
}
```

for (i=0; i < n-1; i++) → 0,…, (n-2) → (n-1)회

for (j=0; j < n-1; j++) → 0,…, (n-2) → (n-1)회

총 비교 횟수

$O(n^2)$

성능과 특징

안정적인 정렬 알고리즘

- 인접한 두 데이터가 동일한 경우 → 위치 교환이 미발생

제자리 정렬 알고리즘

- 입력 배열 A[ ] 이외에 상수 개의 저장 공간(예: i, j, tmp)만 필요

선택 정렬에 비해 데이터 교환이 많이 발생

- 선택 정렬보다 비효율적

개선된 버블 정렬 알고리즘

각 루프의 반복 횟수를 줄여서 개선 가능

- 처리 단계의 수
  - ✓ 자리바꿈이 발생하지 않으면 이미 정렬된 상태이므로 이후의 처리 단계를 수행하지 않고 종료
- 인접한 두 데이터의 비교 횟수
  - ✓ 각 단계에서 무조건 오른쪽/왼쪽 끝까지 이동하면서 인접한 두 데이터의 비교가 불필요
    - → 이미 제자리를 잡은 데이터에 대해서는 비교를 수행하지 않도록 함

```
BubbleSort (A[ ], n)
{
① for (i=0; i < n-1; i++)
② for (j=0; j < n-1; j++)
        if (A[j] > A[j+1])
            A[j]와 A[j+1]의 자리바꿈;
    return (A);
}
```

60	20	70	10	80	30	50	40
20	60	10	70	30	50	40	80
20	10	60	30	50	40	70	80
10	20	30	50	40	60	70	80
10	20	30	40	50	60	70	80
10	20	30	40	50	60	70	80
10	20	30	40	50	60	70	80
10	20	30	40	50	60	70	80

개선된 버블 정렬 알고리즘


```
Advanced_BubbleSort (A[ ], n)
{
   for (i=0; i < n-1; i++) {          // 단계: 0, 1, …, (n-2)
      Sorted = TRUE;                  // 이미 정렬된 상태라고 가정
      for (j=0; j < (n-1)-i; j++)     // 왼쪽에서 오른쪽으로 진행하는 경우
         if (A[j] > A[j+1]) {
            A[j]와 A[j+1]의 자리바꿈;
            Sorted = FALSE;           // 자리바꿈 발생 → 미정렬 상태
         }
         if (Sorted == TRUE) break;      // 이미 정렬된 상태이므로 종료
   }   return (A);
}
```

개선된 알고리즘의 성능과 특징

시간 복잡도 $O(n^2)$

- 총 비교 횟수 → $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$

입력 데이터의 상태에 따라 성능이 달라짐

역순으로 정렬된 경우

단계					
0	50 ↔ 40 ↔ 30 ↔ 20 ↔ 10				
1	40 ↔ 30 ↔ 20 ↔ 10	50			
2	30 ↔ 20 ↔ 10	40	50		
3	20 ↔ 10	30	40	50	
	10	20	30	40	50

최악의 경우: $O(n^2)$

원하는 순서로 이미 정렬된 경우

단계					
0	10	20	30	40	50
	10	20	30	40	50

최선의 경우: $O(n)$

ALGORITHM
알고리즘

04. 삽입 정렬

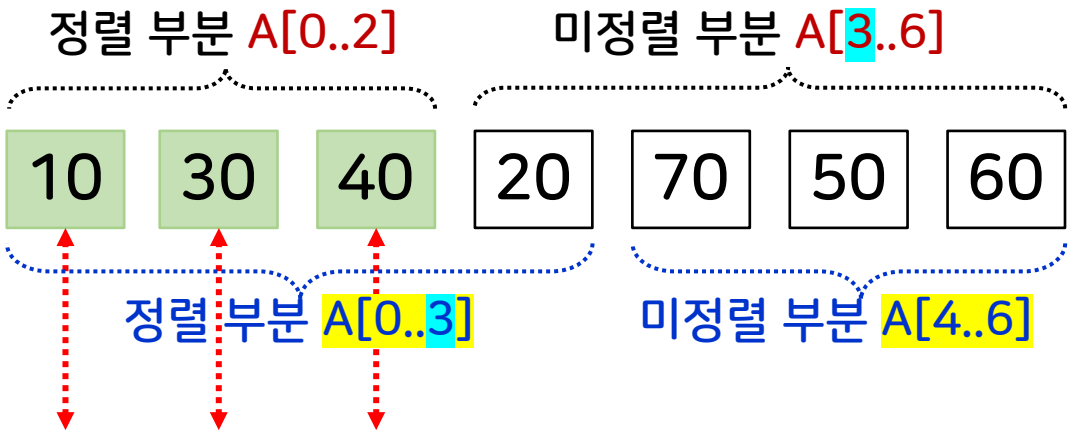
삽입 정렬?

주어진 데이터를 하나씩 뽑은 후 이미 나열된 데이터가 항상 정렬된 상태를 유지하도록 뽑은 데이터를 바른 위치에 삽입해서 나열하는 방식

- 입력 배열을 정렬 부분 A[0..k-1]과 미정렬 부분 A[k..n-1]으로 구분해서 처리
  - ✓ A[0]를 정렬 부분, A[1..n-1]은 미정렬 부분으로 취급하여 시작
  - ✓ k=1, …, n-1까지 반복
    - 미정렬 부분 A[k..n-1]의 첫 번째 데이터 A[k]를 뽑고,
    - 정렬 부분 A[0..k-1]에서 제자리를 찾아 A[k]를 삽입해서 A[0..k]가 정렬 상태를 유지하도록 만듦

삽입 정렬

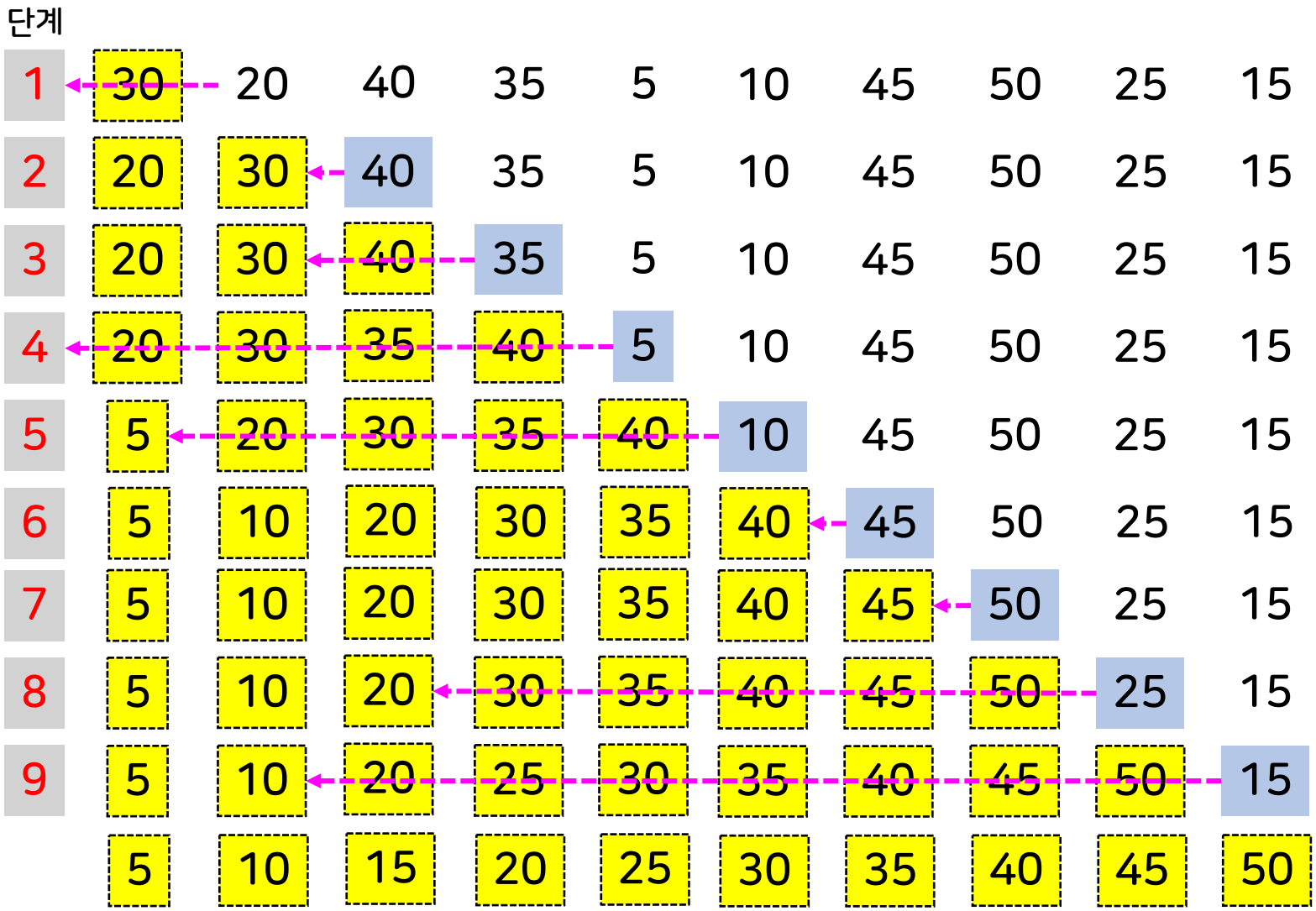
정렬 부분에서 제자리를 찾는 과정



삽입 정렬 알고리즘


```
InsertionSort (A[ ], n)
{
  for (i=1; i < n; i++) {       // A[0] 정렬 부분;  1, …, (n-1)까지 (n-1)번 반복
    val = A[i];                 // 미정렬 부분 A[i..n-1]의 첫 번째 데이터 선택
    for (j=i; j > 0 && A[j-1] > val; j--)      // 삽입할 위치 찾기
      A[j] = A[j-1];            // 정렬 부분의 A[j-1]이 크면 뒤로 한 칸 이동
    A[j] = val;                 // 찾아진 위치에 선택된 데이터 삽입
  }
  return (A);
}
```

삽입 정렬의 예


단계
1 30 20 40 35 5 10 45 50 25 15
2 20 30 40 35 5 10 45 50 25 15
3 20 30 40 35 5 10 45 50 25 15
4 20 30 35 40 5 10 45 50 25 15
5 5 20 30 35 40 10 45 50 25 15
6 5 10 20 30 35 40 45 50 25 15
7 5 10 20 30 35 40 45 50 25 15
8 5 10 20 30 35 40 45 50 25 15
9 5 10 20 25 30 35 40 45 50 15
5 10 15 20 25 30 35 40 45 50

성능과 특징


```
InsertionSort (A[ ], n)
{
  for (i=1; i < n; i++) {
    val = A[i];
    for (j=i; j > 0 && A[j-1] > val; j--)
      A[j] = A[j-1];
    A[j] = val;
  }
  return (A);
}
```

바깥 루프 i	1	2	3	…	n-1
루프 j의 비교 횟수	1	2	3	…	n-1

총 비교 횟수

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

$O(n^2)$

내림차순으로 정렬할 경우

```
for (j=i; j > 0 && A[j-1] < val; j--)
```

성능과 특징

안정적인 정렬 알고리즘

- 인접한 두 데이터가 정렬되지 않은 경우에만 위치 교환이 발생

제자리 정렬 알고리즘

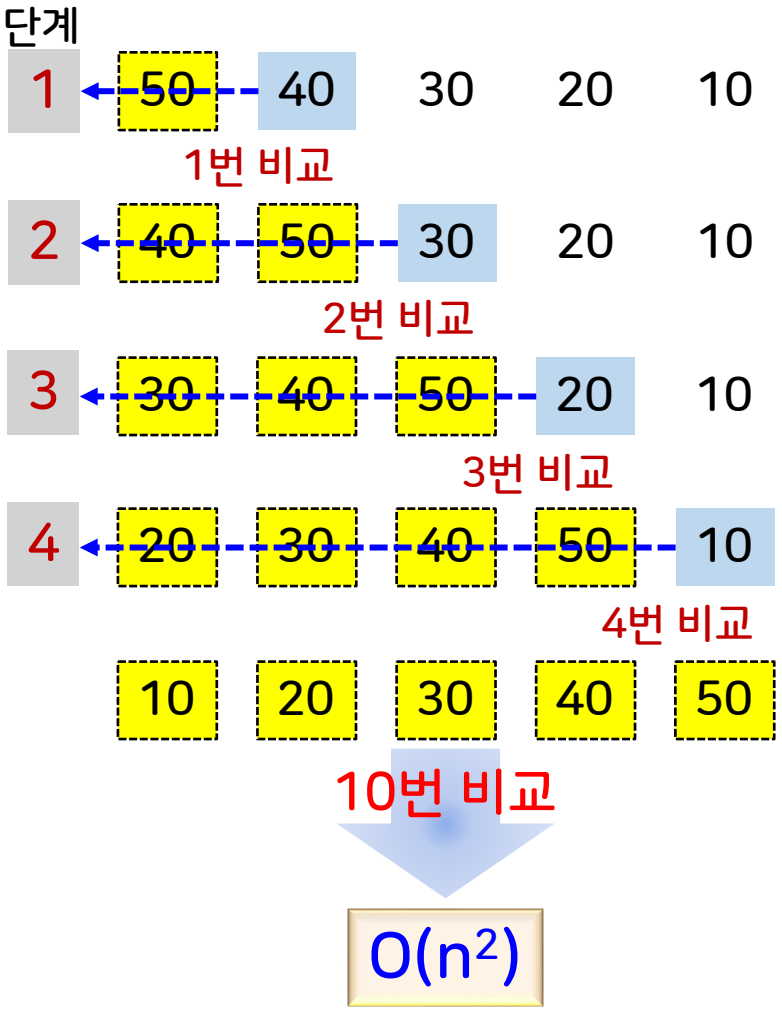
- 입력 배열 A[ ] 이외에 상수 개의 저장 공간(예: i, j, val)만 필요

입력 데이터의 원래 순서에 민감

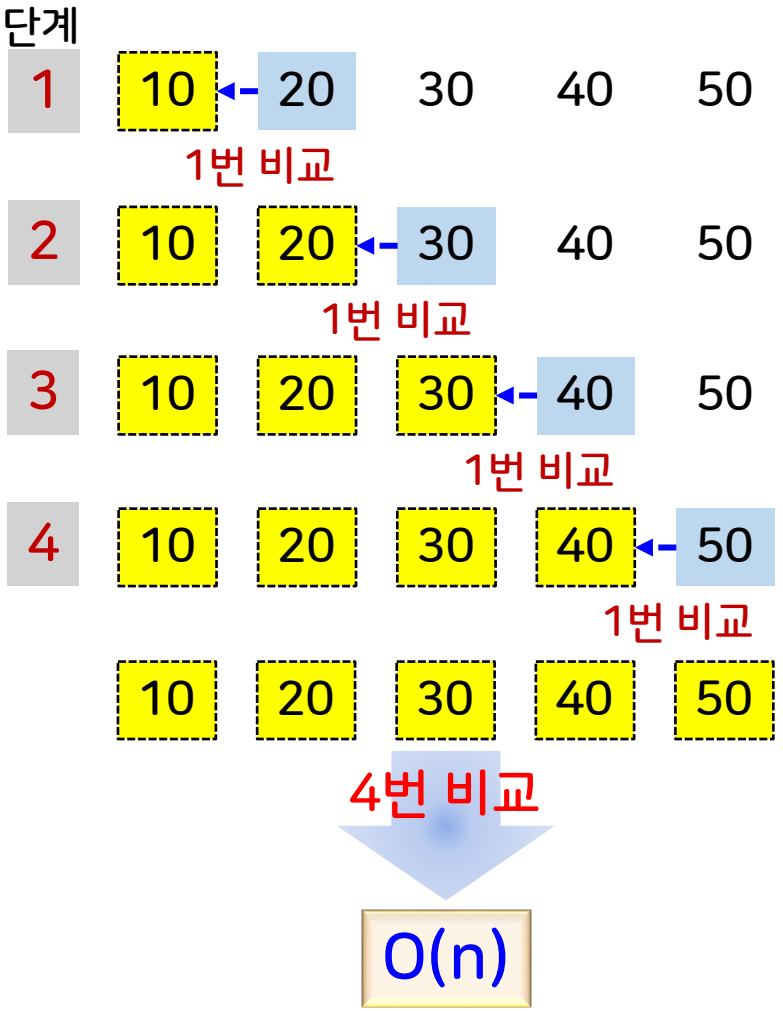
- 원하는 정렬 순서의 역순으로 주어지는 경우 → $O(n^2)$
- 원하는 순서의 정렬된 상태로 주어지는 경우 → $O(n)$

성능과 특징

역순으로 정렬된 경우



제순서대로 정렬된 경우



ALGORITHM
알고리즘

05. 셸 정렬

셸 정렬?

삽입 정렬의 단점 보완 by Donald L. Shell

- 현재 삽입하고자 하는 데이터가 삽입될 제 위치에서
  많이 벗어나 있어도 한 번에 한 자리씩만 이동해서 찾아가야 함

기본 아이디어

- 멀리 떨어진 데이터와의 비교·교환으로
  한 번에 이동할 수 있는 거리를 늘려서 처리 속도 향상

✓ 처음에는 멀리 떨어진 두 데이터를 비교해서 필요시 위치를 교환하고,
점차 가까운 위치의 데이터를 비교·교환한 뒤,
맨 마지막에는 인접한 데이터를 비교·교환하는 방식

✓ 입력 배열을 부분배열로 나누어 삽입 정렬을 수행하는 과정을
부분배열의 크기와 개수를 변화시켜 가면서 반복

부분배열의 개수를 정하는 방법

- 양수로 이루어진 임의의 순열 $h_1, \cdots, h_{k-1}, h_k$을 사용
  - ✓ $1 < i < k$인 i에 대하여 $h_{i-1} < h_i < h_{i+1}$를 항상 만족
  - ✓ 임의의 i, j에 대하여 $i < j$이면 h_j는 h_i의 배수가 되지 않음
  - ✓ 항상 $h_1 = 1$이 되어야 함
- 순열의 역순으로 적용 → $h_k \rightarrow h_{k-1} \rightarrow \cdots \rightarrow h_1$
  - ✓ 첫 번째 단계 → 전체 배열을 h_k개의 부분배열로 나누어 처리(삽입 정렬 수행),
    두 번째 단계 → 부분 정렬된 전체 배열을 h_{k-1}개의 부분배열로 나누어 처리,
    …
    마지막 단계 → 부분 정렬된 전체 배열을 $h_1 = 1$개의 부분배열, 즉 전체에 대해서 처리

순열값 h_k의 의미

- 부분배열의 개수
  - ✓ 전체 배열을 h_k개의 부분배열로 나누어 각 부분배열에 대해서 삽입 정렬을 적용
- 각 부분배열 내에서 이웃한 데이터 간의 거리
  - ✓ i번째 부분배열을 구성하는 데이터
    - → 배열 인덱스를 h_k로 나누었을 때 나머지가 $i-1$인 데이터
  - ✓ 각 부분배열은 전체 배열에서 h_k씩 떨어진 데이터로 구성

셸 정렬 알고리즘

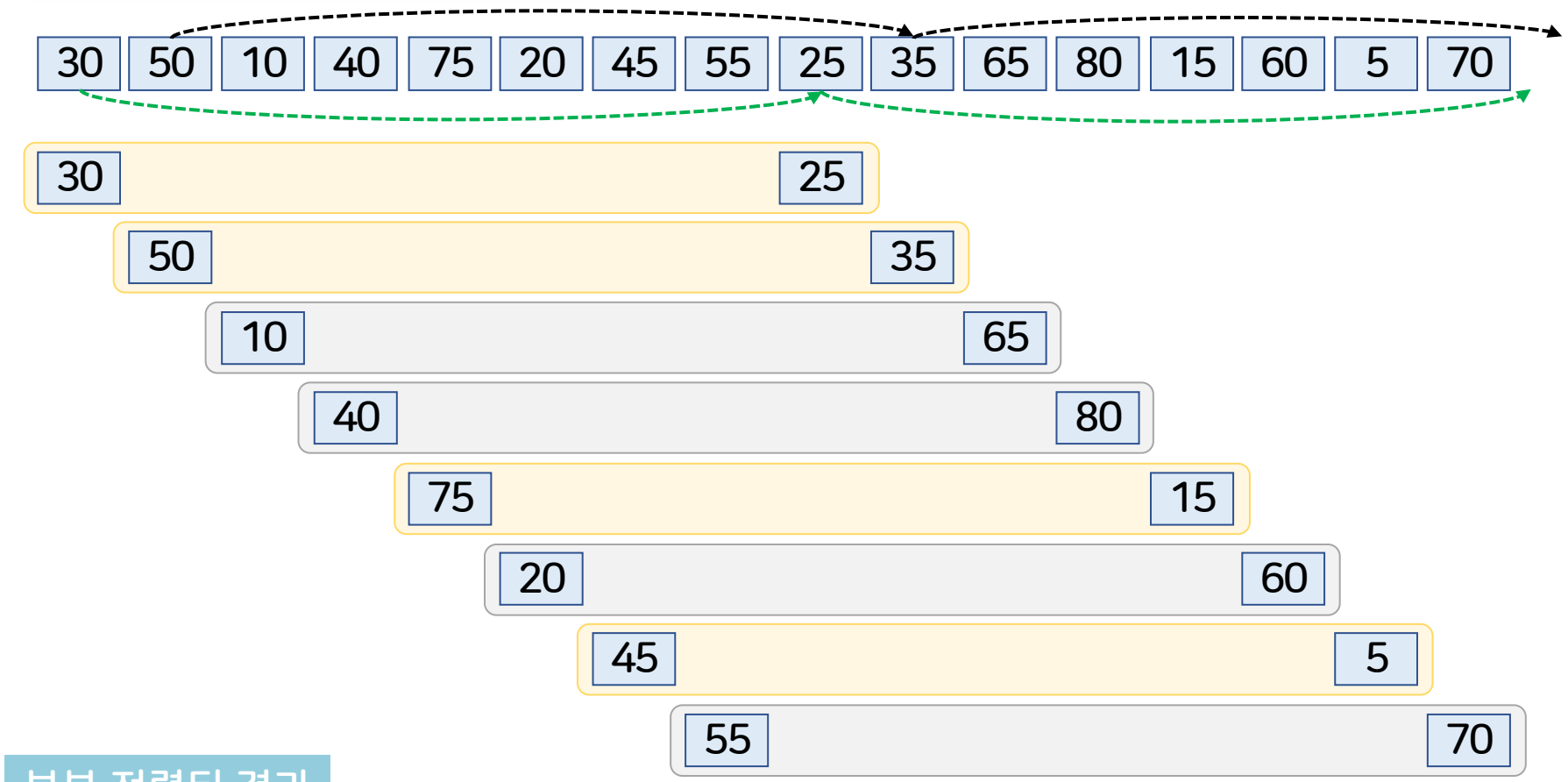

```
ShellSort (A[ ], n)
{
  for (D=⌊n/2⌋; D>=1; D= ⌊D/2⌋) {    // D: 부분배열의 개수 & 간격의 크기
    for (i=D; i < n; i++) {
      val = A[i];
      for (j=i; j>=D && A[j-D] > val; j=j-D)
        A[j] = A[j-D];
      A[j] = val;
    }
  }
  return (A);
}
```

D개의 부분배열에 대한 삽입 정렬 과정

- 하나의 입력 배열을 물리적으로 여러 개의 부분배열로 분할하지 않음
- 각 부분배열을 번갈아 가면서 미정렬 부분의 첫 번째 데이터를 뽑은 후 D만큼씩 떨어진 정렬 부분에서 제자리를 찾아서 삽입하는 방식

셸 정렬의 예_1

$D = \lfloor n/2 \rfloor = \lfloor 16/2 \rfloor = 8$

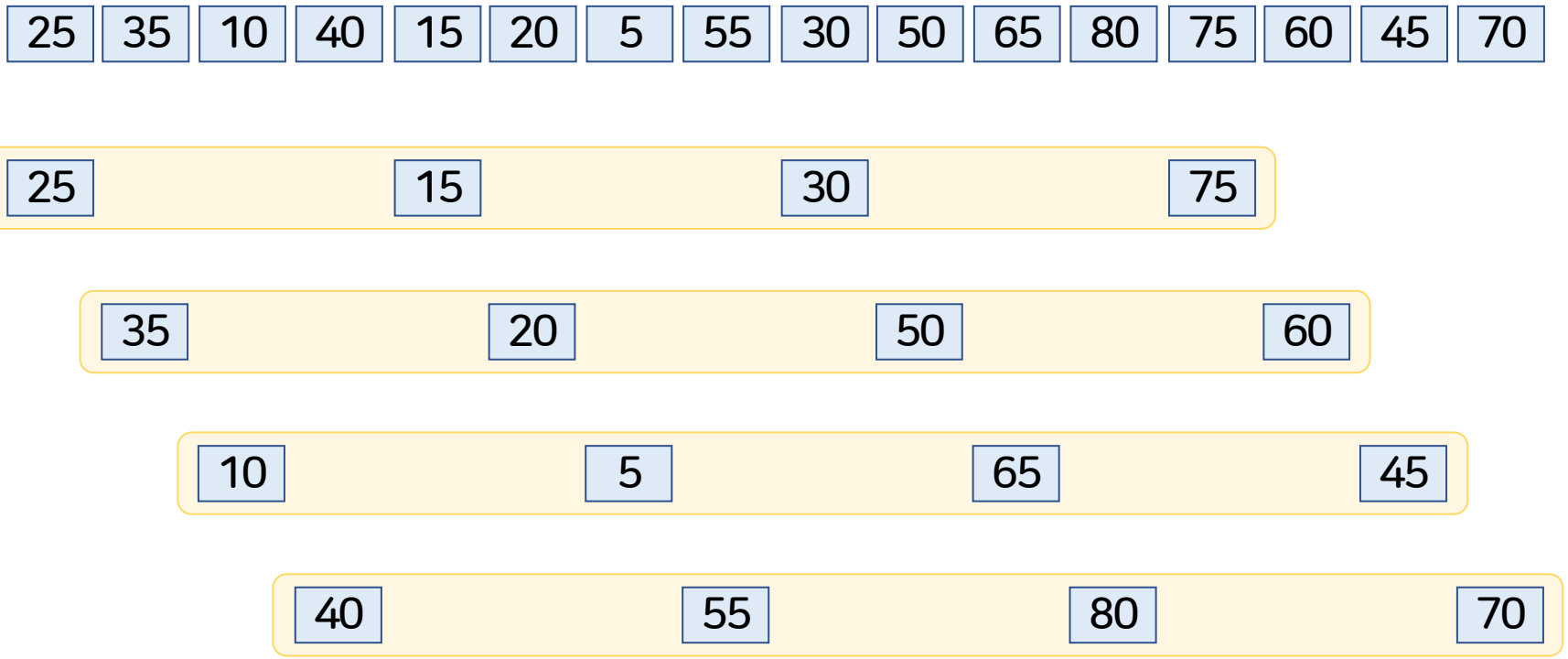


부분 정렬된 결과

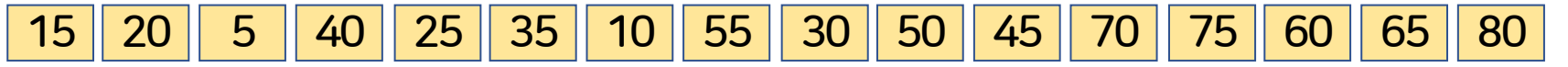
25 35 10 40 15 20 5 55 30 50 65 80 75 60 45 70

셸 정렬의 예_1

$D = \lfloor D/2 \rfloor = \lfloor 8/2 \rfloor = 4$



부분 정렬된 결과



셸 정렬의 예_1

$D = \lfloor D/2 \rfloor = \lfloor 4/2 \rfloor = 2$

15	20	5	40	25	35	10	55	30	50	45	70	75	60	65	80

15		5		25		10		30		45		75		65	

	20		40		35		55		50		70		60		80

부분 정렬된 결과

5	20	10	35	15	40	25	50	30	55	45	60	65	70	75	80

셸 정렬의 예_1

D= ⌊D/2⌋ = ⌊2/2⌋ = 1

5	20	10	35	15	40	25	50	30	55	45	60	65	70	75	80

5	20	10	35	15	40	25	50	30	55	45	60	65	70	75	80

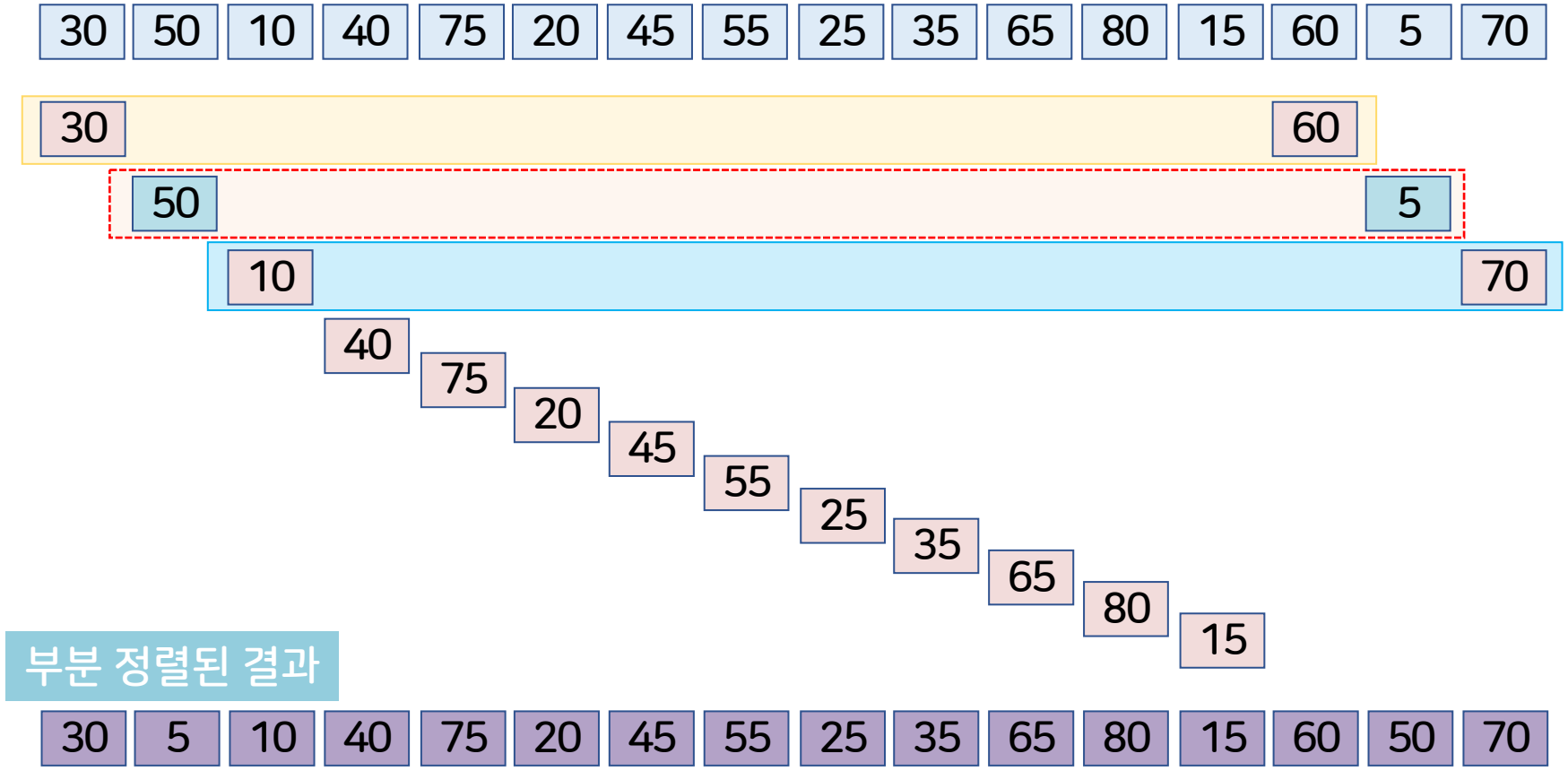
최종 정렬 결과

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80

셸 정렬의 예_2

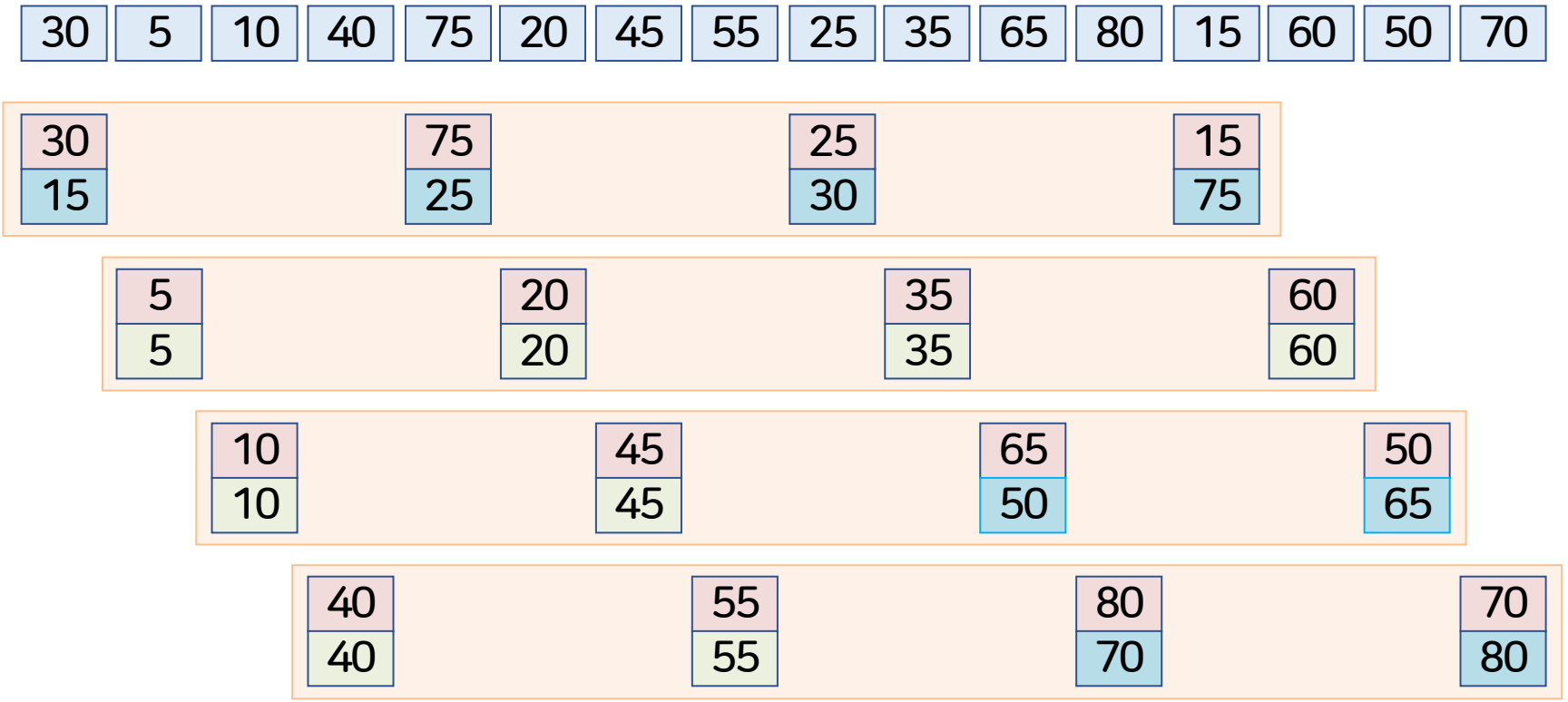
$h_{i+1} = 3h_i + 1, \quad h_1 = 1 \rightarrow h_i = 1,\ 4,\ 13,\ 40,\ 121,\ 364, \cdots$

$D = (h_3 = 13)$의 경우

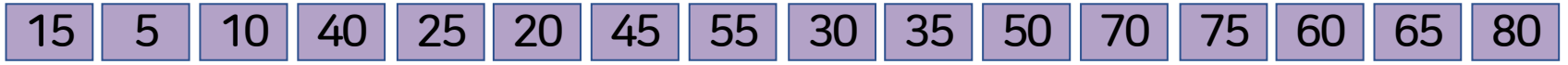


셸 정렬의 예_2

$D = (h_2 = 4)$의 경우



부분 정렬된 결과



셸 정렬의 예_2

$D = (h_1 = 1)$의 경우

15	5	10	40	25	20	45	55	30	35	50	70	75	60	65	80

15	5	10	40	25	20	45	55	30	35	50	70	75	60	65	80

최종 정렬 결과

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80

성능과 특징

사용하는 순열에 따라 성능이 달라짐

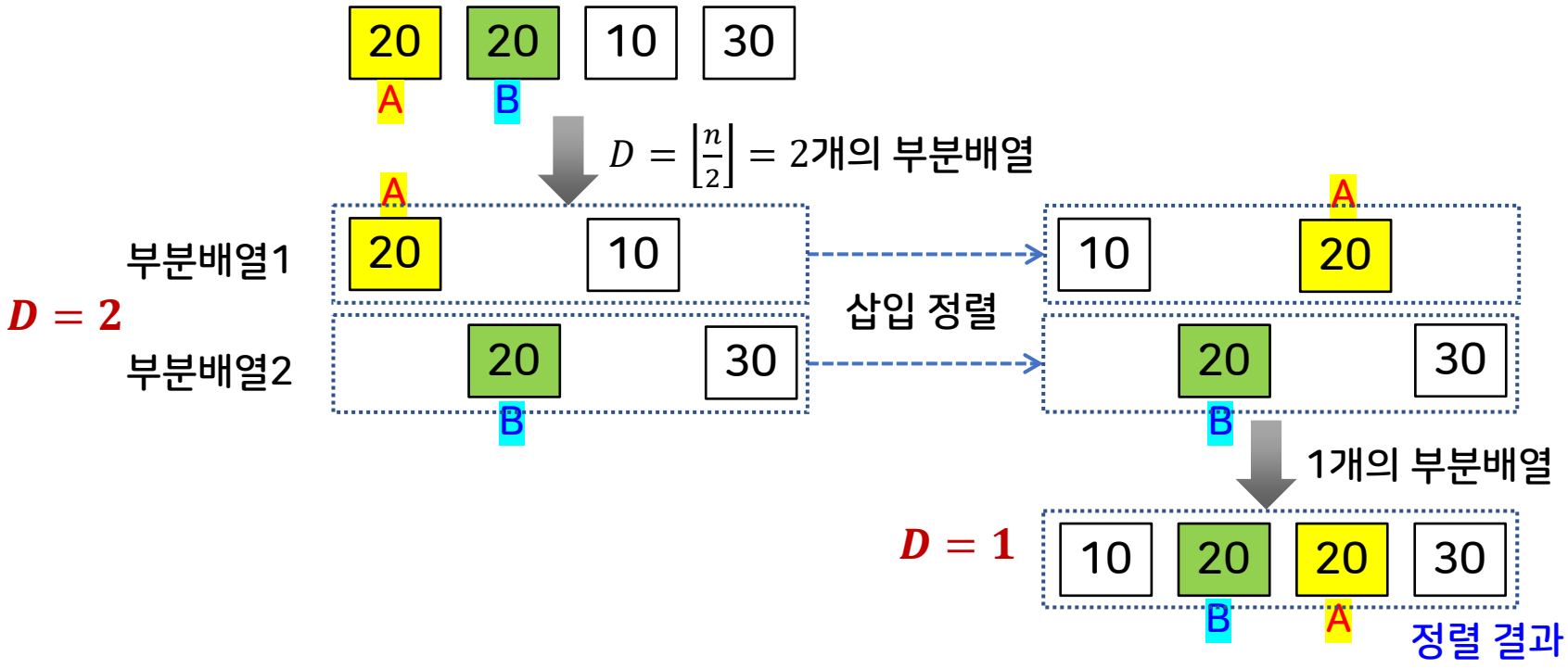
- D= $\lfloor n/2^i \rfloor$ (n: 데이터 개수, $i = 1,2,3,\cdots$)
- 가장 좋은 간격을 찾는 것은 아직 미해결 과제
  - ✓ 1, 4, 13, 40, 121, 364, 1093, … ($h_{i+1} = 3h_i + 1,\ h_1 = 1$)
  - ✓ 1, 3, 7, 15, 31, 63, … ($2^i - 1$)
  - ✓ 1, 3, 7, 21, 48, 112, 336, 861, 1968, 4592, …
  - ✓ 1, 4, 10, 23, 57, 132, 391, 701
- 순열값의 역순으로 차례대로 적용 → $h_k \rightarrow h_{k-1} \rightarrow \cdots \rightarrow h_1$
- 최선 O(nlogn) ~ 최악 $O(n^2)$

성능과 특징

제자리 정렬 알고리즘

- 입력 배열 A[ ] 이외에 상수 개의 저장 공간(예: D, i, j, val)만 필요

안정적이지 않은 정렬 알고리즘



정리하기

1. 기본 개념

- 내부 정렬, 비교 기반 ↔ 데이터 분포 기반, 안정적, 제자리

2. 선택 정렬

- $O(n^2)$, 불안정적, 제자리, 언제나 동일한 시간 복잡도를 가짐

3. 버블 정렬

- $O(n^2)$, 안정적, 제자리
- 개선된 알고리즘(→ 데이터의 입력 상태에 따라 성능이 달라짐)

4. 삽입 정렬

- $O(n^2)$, 안정적, 제자리, 입력 상태의 순서에 민감

5. 셸 정렬

- 삽입 정렬 단점 보완→부분배열의 크기/개수를 변화시키면서 삽입 정렬 수행
- $O(n^2)$, 불안정적, 제자리, 사용하는 순열에 따라 성능이 달라짐

ALGORITHM
알고리즘

다음시간에는

Lecture 04

정렬 (2)

컴퓨터과학과 | 이관용 교수

한국방송통신대학교