Chapter 6. 정렬 알고리즘

정렬 알고리즘

- 기본적인 정렬 알고리즘
  - 버블 정렬
  - 선택 정렬
  - 삽입 정렬
- 효율적인 정렬 알고리즘
  - 쉘 정렬
  - 힙 정렬
  - 합병 정렬
  - 퀵 정렬
- 입력이 제한된 크기 이내에 숫자로 구성되어 있을 때
  - 기수 정렬

내부정렬 vs. 외부정렬

- 내부정렬 (Internal sort)
  - 입력의 크기가 주기억 장치(main memory)의 공간보다 크지 않은 경우에 수행되는 정렬이다.
  - 이전 슬라이드의 모든 정렬 알고리즘들이 내부정렬 알고리즘들이다.

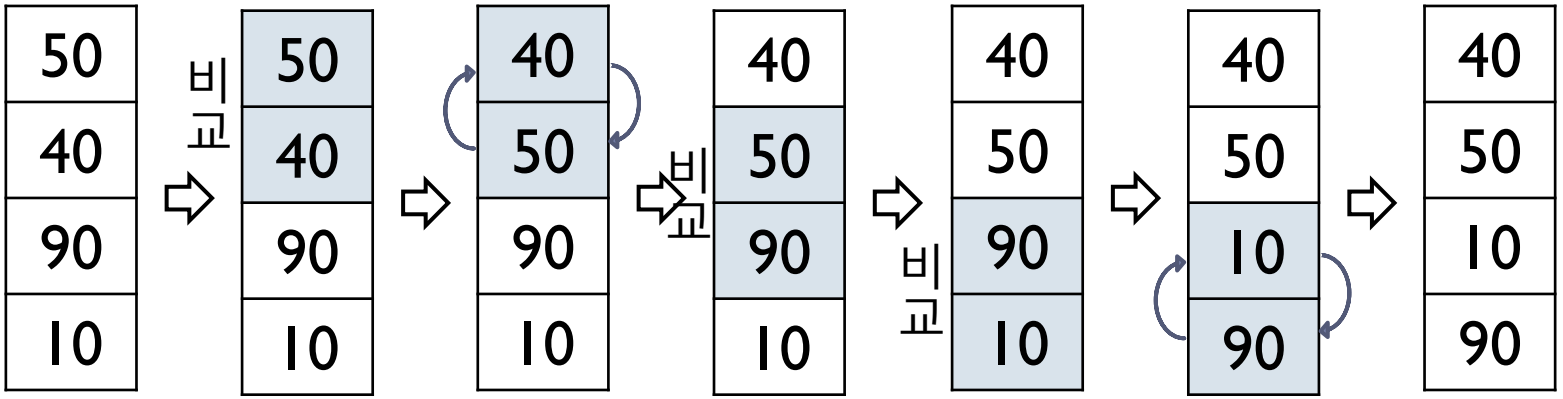
- 외부정렬 (External sort)
  - 입력의 크기가 주기억 장치 공간보다 큰 경우, 보조 기억 장치에 있는 입력을 여러 번에 나누어 주기억 장치에 읽어 들인 후 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복해야 한다.
  - 6.8절에서 자세히 학습한다.

6.1 버블 정렬

- 버블 정렬(Bubble Sort)은 이웃하는 숫자를 비교하여 작은 수를 앞쪽으로 이동시키는 과정을 반복하여 정렬하는 알고리즘이다.
- 오름차순으로 정렬한다면 작은 수는 배열의 앞부분으로 이동하는데, 배열을 좌우가 아니라 상하로 그려보면 정렬하는 과정에서 작은 수가 마치 '거품'처럼 위로 올라가는 것을 연상케 한다.

버블 정렬

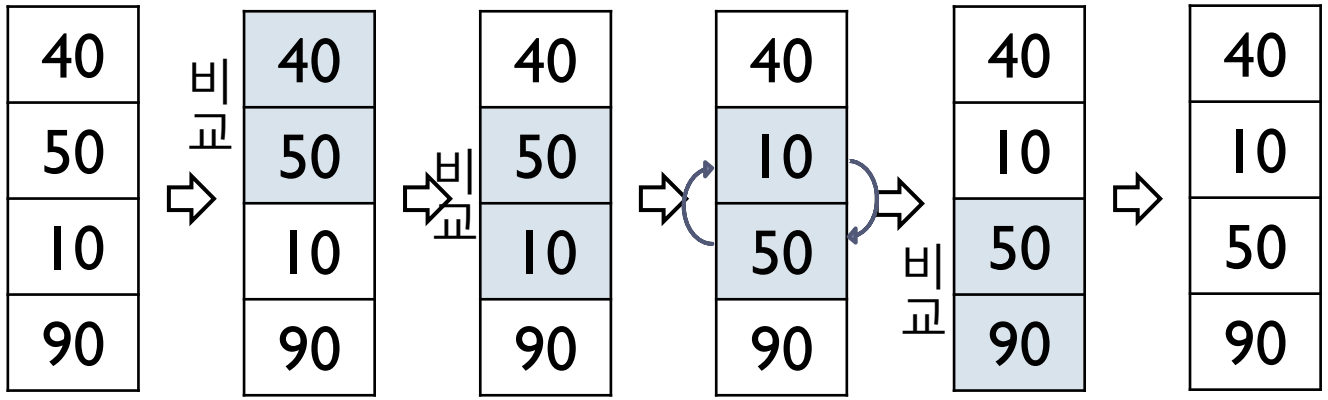
- 배열에 50, 40, 90, 10이 차례로 저장되어 있다고 가정하자.
- 입력을 전체적으로 한 번 처리하는 것을 패스(pass)라고 한다.
- 아래 그림과 같이 첫 번째 패스 후에 그 결과를 살펴보면, 작은 수는 버블처럼 위로 1칸씩 올라갔다.'무거운' 수(즉, 큰 수)의 측면에서 관찰해보면, 가장 큰 수가 '바닥'(즉, 배열의 가장 마지막 원소)에 위치하게 된다.



버블 정렬

결과를 보여주었다.

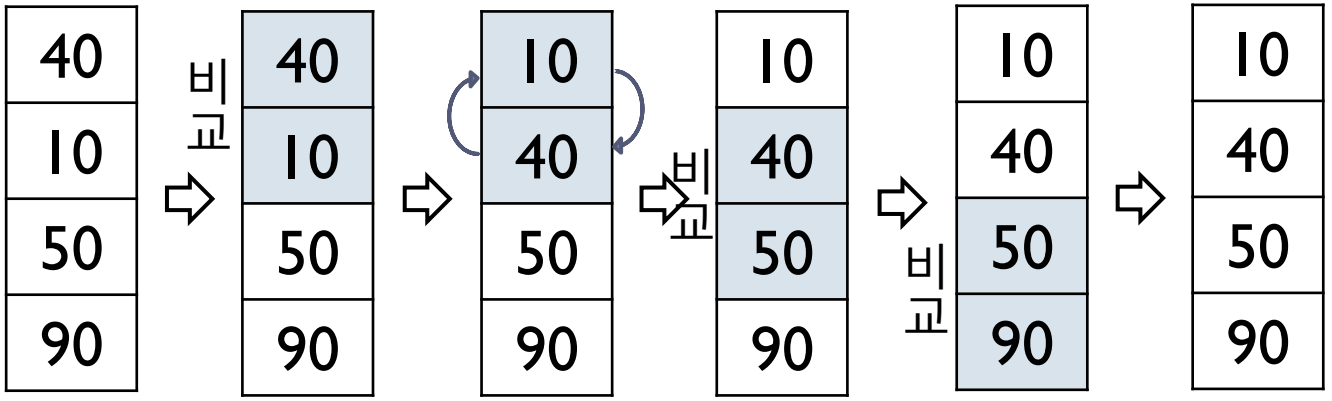
- 두 번째 패스가 수행되는 과정이다.
- 이웃하는 원소간의 비교를 통해 40-50과 50-90은 그대로 그 자리에 있고, 50과 10이 서로의 자리를 바꾸었다. 역시 눈여겨보아야 할 것은 두 번째로 큰 수인 50이 가장 큰 수인 90의 위로 '가라앉았다.'



  - 참고: 50과 90의 비교는 불필요하다.

버블 정렬

- 마지막 패스가 수행되는 과정이다.
- 이웃하는 원소간의 비교를 통해 40과 10이 서로 자리를 바꾸었고, 40-50과 50-90은 그대로 그 자리에 있다. 세 번째로 큰 수인 40이 두 번째로 큰 수인 50의 위로 '가라앉았다.'



  - 참고: 40과 50, 50과 90의 비교는 불필요하다.
- n개의 원소가 있으면 (n–1)번의 패스가 수행된다.

버블 정렬 알고리즘

BubbleSort	
입력: 크기가 n인 배열 A 출력: 정렬된 배열 A	
1	for pass = 1 to n–1
2	for i = 1 to n–pass
3	if (A[i–1] > A[i]) // 위의 원소가 아래의 원소보다 크면
4	A[i–1] ↔ A[i] // 서로 자리를 바꾼다
5	return 배열 A

버블 정렬 알고리즘

Line 1	for-루프가 (n–1)번의 패스를 수행한다.
Line 2	for-루프는 배열의 이웃하는 원소를 A[0]부터 A[n–pass]까지 비교하기 위함이다. 배열 인덱스 (n–pass)는 pass=1이면 A[n–1]까지 비교하고, pass=2이면 A[n–2]까지 비교하는데, 이는 pass=1이 수행되고 나면 가장 큰 숫자가 A[n–1]에 저장되므로 A[n–2]까지만 비교하기 위한 것이다. 마찬가지로 pass=3이면, 2번째로 큰 숫자가 A[n–2]에 저장되어 있기 때문에 A[n–3]까지만 비교한다.
Line 3~4	if-조건인 (A[i–1] > A[i])가 '참'이면, 즉, 위의 원소가 아래의 원소보다 크면, A[i–1]과 A[i]를 서로 바꾼다. 만일 if-조건이 '거짓'이면 아무 일도 하지 않고 다음 i값에 대해 알고리즘이 진행된다.

BubbleSort 예제

- 입력

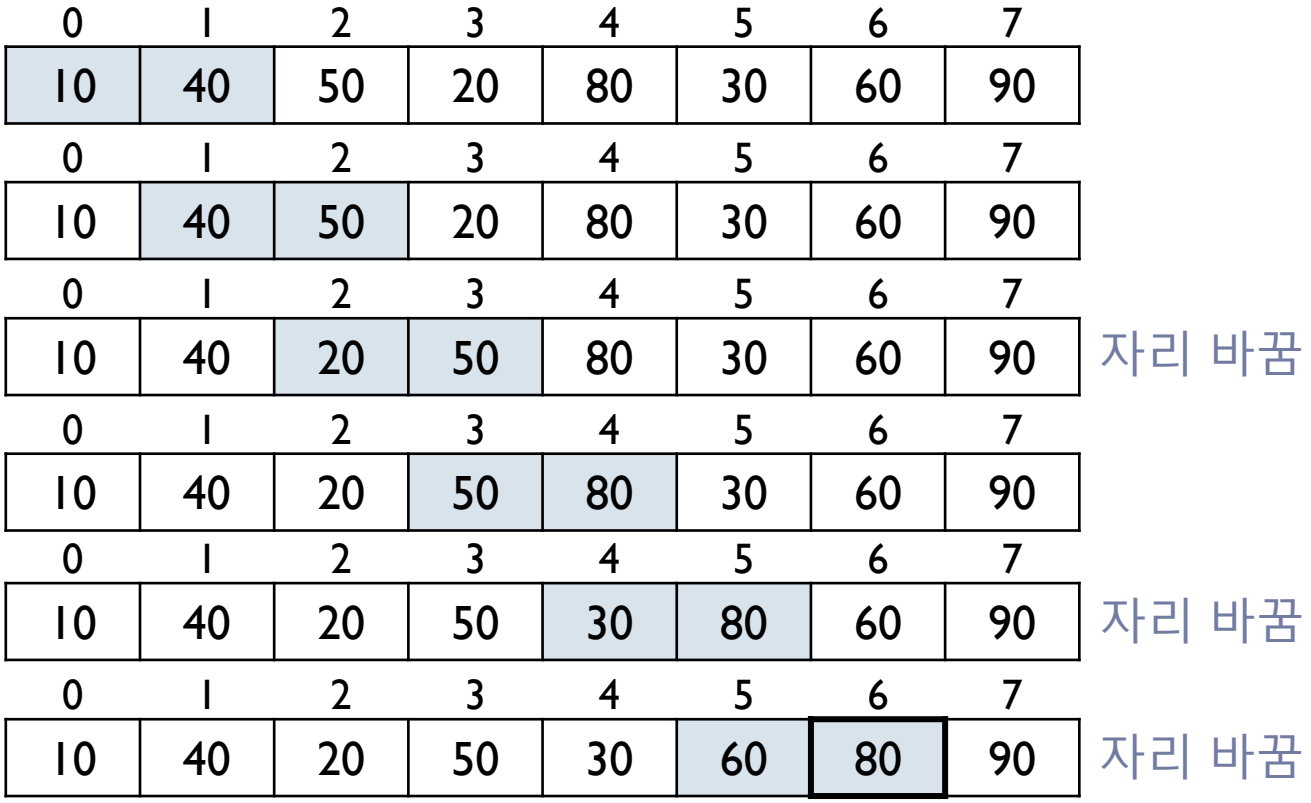
0	1	2	3	4	5	6	7
40	10	50	90	20	80	30	60

- 패스 1

0	1	2	3	4	5	6	7	
10	40	50	90	20	80	30	60	자리 바꿈
10	40	50	90	20	80	30	60	
10	40	50	90	20	80	30	60	
10	40	50	20	90	80	30	60	자리 바꿈
10	40	50	20	80	90	30	60	자리 바꿈
10	40	50	20	80	30	90	60	자리 바꿈
10	40	50	20	80	30	60	90	자리 바꿈

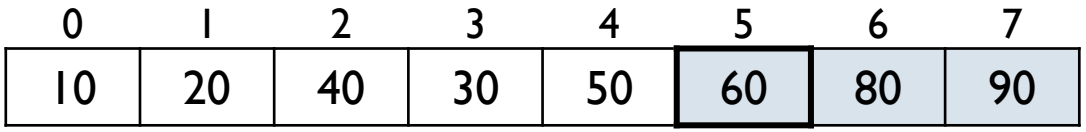
BubbleSort 예제

- 패스 2

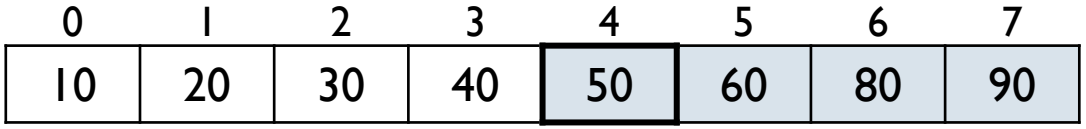


BubbleSort 예제

- 패스 3의 결과



- 패스 4의 결과



- 패스 5~7의 결과는 패스 4의 결과와 동일하다.

시간복잡도

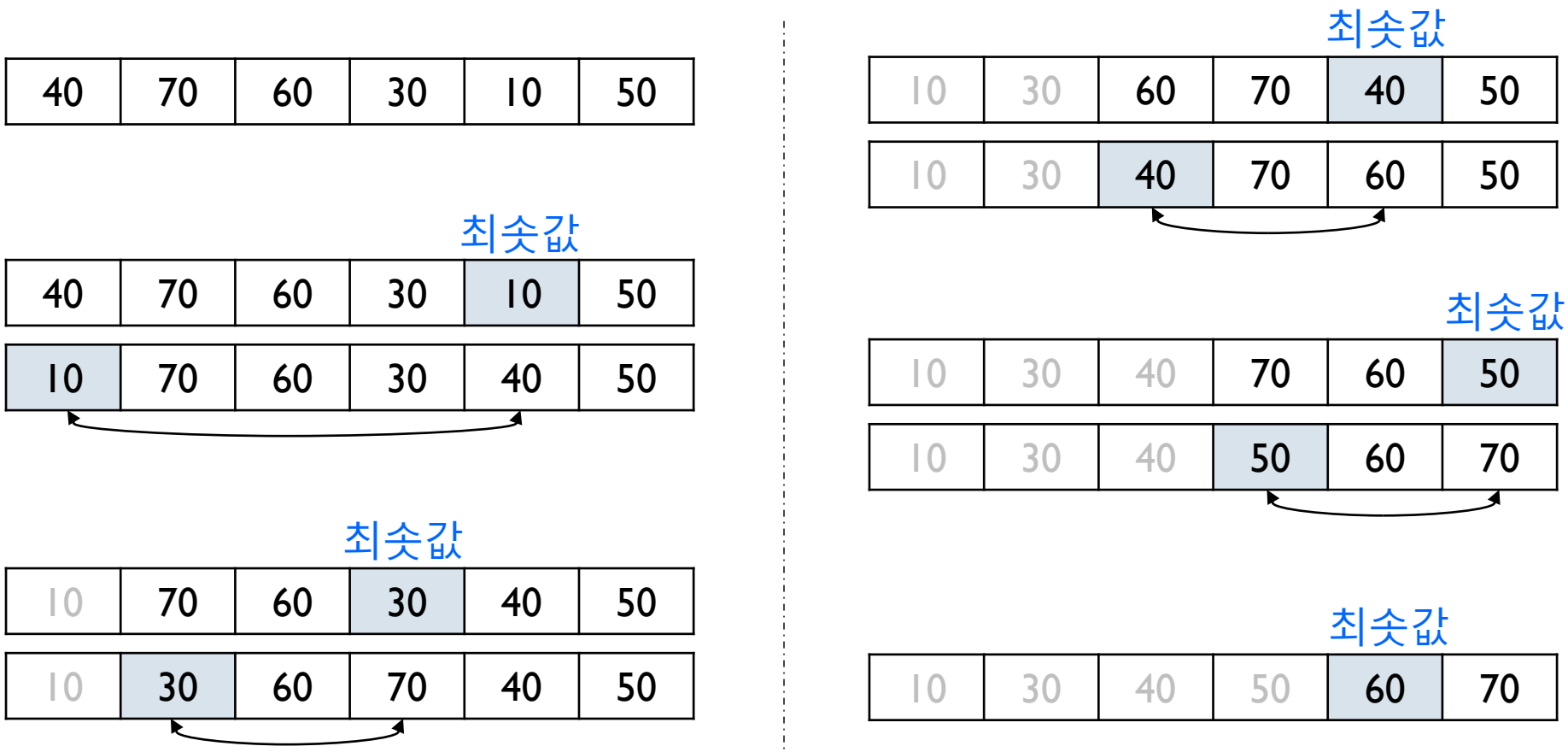
- 버블 정렬은 for-루프 속에서 for-루프가 수행된다.
  - pass=1이면 (n–1)번 비교하고,
  - pass=2이면 (n–2)번 비교하고,
  - …
  - pass=n–1이면 1번 비교한다.

  총 비교 횟수는 $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$이다.
- 안쪽 for-루프의 if-조건이 '참'일 때의 자리바꿈은 O(1) 시간이 걸린다.
- 버블 정렬 알고리즘의 시간복잡도는 $n(n-1)/2 \times O(1) = O(n^2) \times O(1) = O(n^2)$이다.

6.2 선택 정렬

- 선택 정렬(Selection Sort)은 입력 배열 전체에서 최솟값을 '선택'하여 배열의 0번 원소와 자리를 바꾸고, 다음에는 0번 원소를 제외한 나머지 원소에서 최솟값을 선택하여, 배열의 1번 원소와 자리를 바꾼다.
- 이러한 방식으로 마지막에 2개의 원소 중 최솟값을 선택하여 자리를 바꿈으로써 오름차순의 정렬을 마친다.

선택 정렬


40 70 60 30 10 50
최솟값
40 70 60 30 10 50
10 70 60 30 40 50
최솟값
10 70 60 30 40 50
10 30 60 70 40 50
최솟값
10 30 60 70 40 50
10 30 40 70 60 50
최솟값
10 30 40 70 60 50
10 30 40 50 60 70
최솟값
10 30 40 50 60 70

선택 정렬

- 앞의 예제를 살펴보면 10이 배열 전체에서 최솟값이다. 그래서 10을 첫 번째 원소인 40과 교환한다.
- 그 다음에는 10을 제외한 배열의 나머지 원소들 중에서 최솟값인 30을 찾는다. 30은 두 번째 원소인 70과 교환된다.
- 다음은 이러한 과정을 반복 하는 선택 정렬 알고리즘이다.

선택 정렬 알고리즘

최솟값을 찾고 보내기

SelectionSort	
입력: 크기가 n인 배열 A 출력: 정렬된 배열 A	
1	for i = 0 to n–2 {
2	min = i
3	for j = i+1 to n–1 { // A[i]~A[n–1]에서 최솟값을 찾는다.
4	if (A[j] < A[min])
5	min = j
	}
6	A[i] ↔ A[min] // min이 최솟값이 있는 원소의 인덱스이다.
	}
7	return 배열 A

선택 정렬 알고리즘

Line 1	for-루프에서는 i가 0부터 (n–2)까지 변하는데, 이는 A[i]에서부터 A[n–1]까지의 숫자들 중에서 최솟값을 찾기 위함이다. (n–1)까지 반복하지 않는 것은 (n–1)까지 수행할 경우 line 3의 j 값이 i+1 = (n–1)+1 = n이 되어 배열의 범위를 벗어나기 때문이다.
Line 2	A[i]를 최솟값으로 놓고, 즉 min = i로 놓고. Line 3의 for-루프에서는 A[i+1]부터 1개의 원소씩 A[min]과 비교하고, A[min]보다 작은 원소가 발견되면 min = j로 갱신하며, 최종적으로 A[n–1]까지 검사한 후에 최솟값이 있는 원소의 인덱스를 min에 저장한다.
Line 6	line 3~5의 for-루프에서 찾은 최솟값 A[min]을 A[i]와 교환한다.
Line 7	line 1~6의 for-루프가 끝나면, 정렬된 배열 A를 리턴한다.

SelectionSort 예제

0	1	2	3	4	5	6	7
40	10	50	90	20	80	30	60

- i=0일 때, A[0]~A[7]에서 최솟값은 10이다. 즉 min=1이다.

0	1	2	3	4	5	6	7
40	10	50	90	20	80	30	60

0	1	2	3	4	5	6	7
10	40	50	90	20	80	30	60

자리 바꿈

- i=1일 때, A[1]~A[7]에서 최솟값은 20이다. 즉 min=4이다.

0	1	2	3	4	5	6	7
10	40	50	90	20	80	30	60

0	1	2	3	4	5	6	7
10	20	50	90	40	80	30	60

자리 바꿈

SelectionSort 예제

- i=2일 때, A[2]~A[7]에서 최솟값은 30이다. 즉 min=6이다.

0	1	2	3	4	5	6	7
10	20	50	90	40	80	30	60

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

자리 바꿈

…

- i=6일 때, A[6]~A[7]에서 최솟값은 80이다. 즉 min=7이다.

0	1	2	3	4	5	6	7
10	20	30	40	50	60	90	80

0	1	2	3	4	5	6	7
10	20	30	40	50	60	80	90

자리 바꿈

시간복잡도

- 선택 정렬은 line 1의 for-루프가 (n–1)번 수행되는데,
  - i=0일 때 line 3의 for-루프는 (n–1)번 수행되고,
  - i=1일 때 line 3의 for-루프는 (n–2)번 수행되고, …
  - 마지막으로 i=n–2일 때 line 3의 for-루프는 1번 수행되므로,
- 루프 내부의 line 4~5가 수행되는 총 횟수는 (n–1)+(n–2)+(n–3)+… +2+1 = n(n–1)/2이다.
- 루프 내부의 if-조건이 '참'일 때의 자리바꿈은 O(1) 시간이 걸리므로, 시간복잡도는 n(n–1)/2 × O(1) = $O(n^2)$이다.

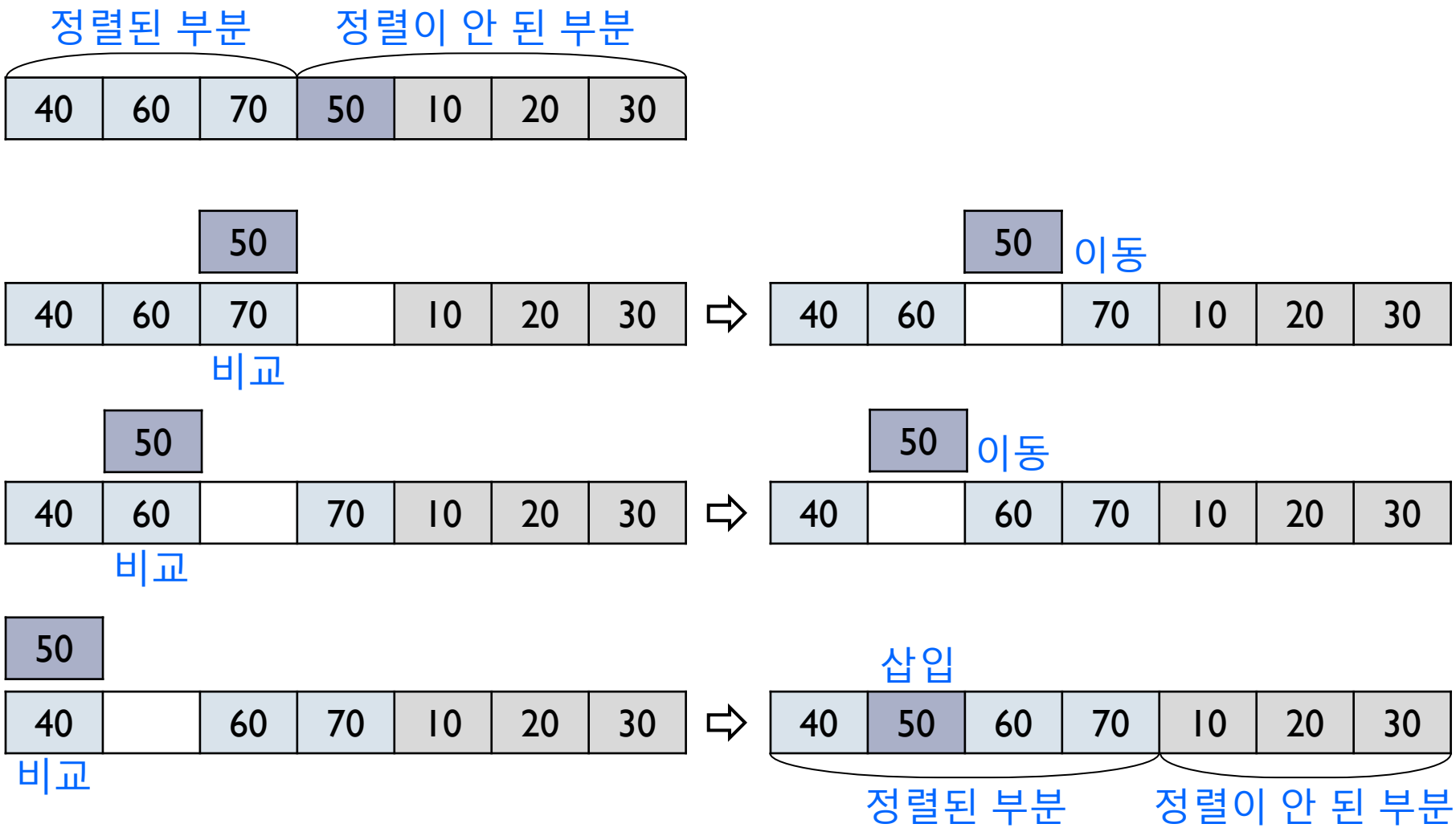
- 선택 정렬의 특징
  - 입력이 거의 정렬되어 있든지, 역으로 정렬되어 있든지, 랜덤하게 되어있든지 관계없이, 항상 일정한 시간복잡도를 나타낸다. 즉, 입력에 민감하지 않은(input insensitive) 알고리즘이다.

6.3 삽입 정렬

- 삽입 정렬(Insertion Sort)은 배열을 정렬된 부분(앞부분)과 정렬이 안 된 부분(뒷부분)으로 나누고, 정렬이 안 된 부분의 가장 왼쪽 원소를 정렬된 부분의 적절한 위치에 삽입하여 정렬되도록 하는 과정을 반복한다.

- 다음 그림은 정렬이 아직 안 된 부분의 가장 왼쪽 원소인 50을 정렬된 부분에 삽입하는 과정을 보이고 있다.

삽입 정렬


정렬된 부분
정렬이 안 된 부분
40 60 70 50 10 20 30
50
40 60 70 10 20 30
비교
50
이동
40 60 70 10 20 30
50
40 60 70 10 20 30
비교
50
이동
40 60 70 10 20 30
50
40 60 70 10 20 30
비교
삽입
40 50 60 70 10 20 30
정렬된 부분
정렬이 안 된 부분

삽입 정렬

- 정렬이 안 된 부분의 숫자 하나가 정렬된 부분에 삽입됨으로써, 정렬된 부분의 원소 수가 1개 늘어나고, 동시에 정렬이 안 된 부분의 원소 수는 1개 줄어든다.
- 이를 반복하여 수행하면, 마지막에는 정렬이 안 된 부분에는 아무 원소도 남지 않고, 정렬된 부분에 모든 원소가 있게 된다.
- 단, 삽입 정렬은 배열의 첫 번째 원소만이 정렬된 부분에 있는 상태에서 정렬을 시작한다.

삽입 정렬 알고리즘

InsertionSort

입력: 크기가 n인 배열 A
출력: 정렬된 배열 A

```
1   for i = 1 to n–1 {
2       CurrentElement = A[i]     // 정렬이 안 된 부분의 가장 왼쪽 원소
3       j ← i–1     // 정렬된 부분의 가장 오른쪽 원소로부터 왼쪽 방향으로
                        삽입할 곳을 탐색하기 위하여
4       while (j >= 0) and (A[j] > CurrentElement) {
5           A[j+1] = A[j]      // 자리 이동
6           j ← j–1
        }
7       A [j+1] ← CurrentElement
    }
8   return A
```

삽입 정렬 알고리즘

Line 1	삽입 정렬은 정렬된 부분에는 A[0]만이 있는 상태에서 정렬이 시작되므로, line 1에서는 CurrentElement의 배열 인덱스 i를 위해 for-루프를 이용하여 1부터 (n–1)까지 변한다.
Line 2	정렬이 안 된 부분의 가장 왼쪽에 있는 원소인 A[i]를 CurrentElement로 놓는다.
Line 3	'j=i–1'은 j가 정렬된 부분의 가장 오른쪽 원소의 인덱스가 되어 왼쪽 방향으로 삽입할 곳을 탐색하기 위함이다.

삽입 정렬 알고리즘

Line 4~6	CurrentElement가 삽입될 곳을 찾는다. while-루프의 조건 (j >= 0)은 배열 인덱스로 사용되는 j가 배열의 범위를 벗어나는 것을 방지하기 위함이고, 두 번째 조건 (A[j] > CurrentElement)는 A[j]가 CurrentElement보다 크면 A[j]를 오른쪽으로 한 칸 이동시키기 위함이다. 이러한 자리이동은 line 5에서 수행된다. Line 6에서는 j를 1 감소시켜 바로 왼쪽 원소에 대해 while-루프를 반복적으로 수행하기 위함이다.
Line 7	A[j+1]에 CurrentElement를 저장하는데, 이는 while-루프가 끝난 직후에는 A[j]가 CurrentElement보다 크지 않으므로 A[j]의 오른쪽 (즉, A[j+1])에 CurrentElement를 삽입해야 하기 때문이다.

InsertionSort 예제

0	1	2	3	4	5	6	7
40	10	50	90	20	80	30	60

- i=1일 때, CurrentElement=A[1]=10, j=i–1=0

0	1	2	3	4	5	6	7
40	10	50	90	20	80	30	60

A[j]=A[0] > CurrentElement=10이므로

0	1	2	3	4	5	6	7
	40	50	90	20	80	30	60

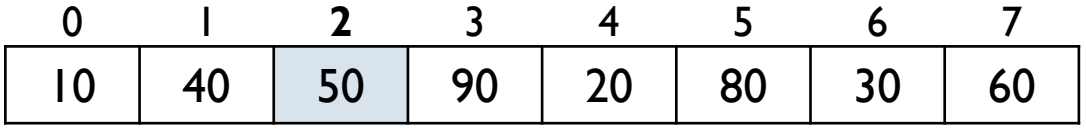
한 칸 이동

j=j–1=0–1=–1이므로, A[j+1]=A[–1+1]=A[0]에 CurrentElement=10을 저장한다.

0	1	2	3	4	5	6	7
10	40	50	90	20	80	30	60

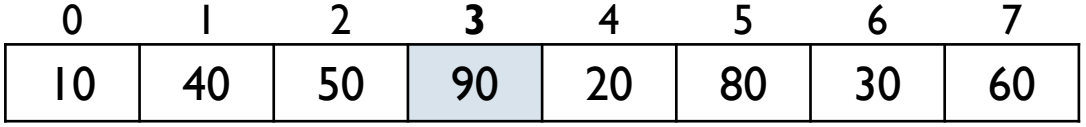
InsertionSort 예제

- i=2일 때, CurrentElement=A[2]=50, j=i–1=1



A[j]=A[1] < CurrentElement=50이므로, 자리이동 없이 50이 그 자리에 위치한다.

- i=3일 때, CurrentElement=A[3]=90, j=i–1=2



A[j]=A[2] < CurrentElement=90이므로, 자리이동 없이 90이 그 자리에 위치한다.

InsertionSort 예제

- i=4일 때, CurrentElement=A[4]=20, j=i–1=3

0	1	2	3	**4**	5	6	7
10	40	50	90	20	80	30	60

A[j]=A[3] > CurrentElement=20이므로

0	1	2	3	4	5	6	7	
10	40	50		90	80	30	60	한 칸 이동

j=j–1=3–1=2이고, A[j]=A[2] > CurrentElement=20이므로

0	1	2	3	4	5	6	7	
10	40		50	90	80	30	60	한 칸 이동

j=j–1=2–1=1이고, A[j]=A[1] > CurrentElement=20이므로

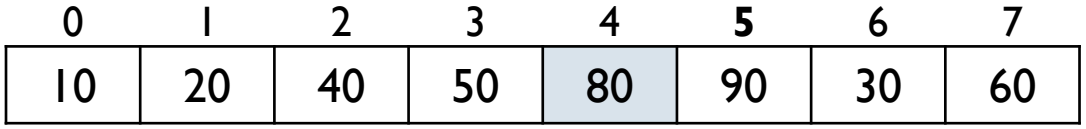
0	1	2	3	4	5	6	7	
10		40	50	90	80	30	60	한 칸 이동

j=j–1=1–1=0이고, A[j]=A[0] < CurrentElement=20이므로,
A[j+1]=A[0+1]=A[1]에 CurrentElement=20을 저장한다.

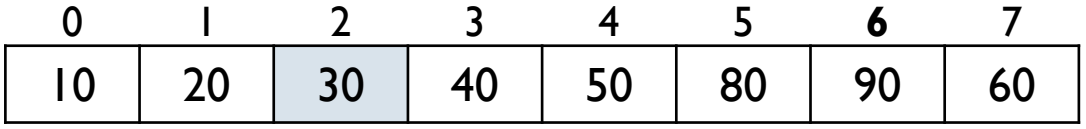
0	1	2	3	4	5	6	7
10	20	40	50	90	80	30	60

InsertionSort 예제

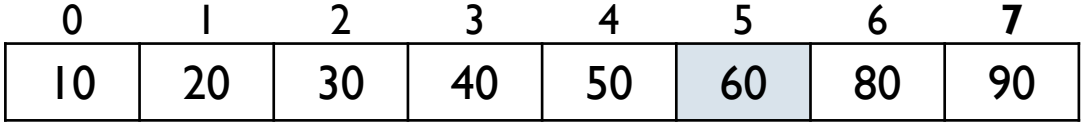
- i=5일 때, CurrentElement=A[5]=80일 때의 결과



- i=6일 때, CurrentElement=A[6]=30일 때의 결과



- i=7일 때, CurrentElement=A[7]=60일 때의 결과



시간복잡도

- 삽입 정렬은 line 1의 for-루프가 (n–1)번 수행되는데,
  - i=1일 때 while-루프는 1번 수행되고,
  - i=2일 때 최대 2번 수행되고,
  - …
  - 마지막으로 i=n–1일 때 최대 (n–1)번 수행되므로
- while-루프 내부의 line 5~6이 수행되는 총 횟수는 1+2+3+…+(n–2) +(n–1) = n(n–1)/2이다.
- while-루프 내부의 수행시간은 O(1)이므로, 삽입 정렬의 시간복잡도는 n(n–1)/2 x O(1) = $O(n^2)$이다.

시간복잡도

- 삽입 정렬은 입력의 정렬 상태에 따라 수행 시간이 달라질 수 있다.

1. 입력이 이미 정렬되어 있으면, 항상 각각 CurrentElement가 자신의 왼쪽 원소와 비교 후 자리이동 없이 원래 자리에 있게 되고, while-루프의 조건이 항상 '거짓'이 되므로 원소의 이동도 없다. 따라서 (n–1)번의 비교만 하면 정렬을 마치게 된다. 이때가 삽입 정렬의 최선 경우이고 시간복잡도는 O(n)이다. 따라서 삽입 정렬은 거의 정렬된 입력에 대해서 다른 정렬 알고리즘보다 빠르다.
2. 반면에 역으로(반대로) 정렬된 입력에 대해서는 앞의 시간복잡도 분석대로 $O(n^2)$ 시간이 걸린다.
3. 삽입 정렬의 평균 경우 시간복잡도는 최악 경우와 같다.

6.4 쉘 정렬

- 버블 정렬, 삽입 정렬
  - 버블 정렬이나 삽입 정렬이 수행되는 과정을 살펴보면, 이웃하는 원소끼리의 자리이동으로 정렬이 이루어진다.
  - 버블 정렬이 오름차순으로 정렬하는 과정을 살펴보면, 작은(가벼운) 숫자가 배열의 앞부분으로 매우 느리게 이동하는 것을 알 수 있다.
  - 삽입 정렬의 경우 만일 배열의 마지막 원소가 입력에서 가장 작은 숫자라면, 그 숫자가 배열의 맨 앞으로 이동해야 하므로, 모든 다른 숫자들이 1칸씩 오른쪽으로 이동해야 한다.
- 쉘 정렬(Shell Sort)
  - 쉘 정렬은 이러한 단점을 보완하기 위해서 삽입 정렬을 이용하여 배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게' 이동시키고, 동시에 앞부분의 큰 숫자는 뒷부분으로 이동시키며, 가장 마지막에는 삽입 정렬을 수행한다.

쉘 정렬

- 다음의 예제를 통해 쉘 정렬의 아이디어를 이해해보자.

  30 60 90 10 40 80 40 20 10 60 50 30 40 90 80

- 먼저 간격(gap)이 5가 되는 숫자끼리 그룹을 만든다.
  - 총 15개의 숫자가 있으므로, 첫 번째 그룹은 첫 숫자인 30, 첫 숫자에서 간격이 5가 되는 숫자인 80, 그리고 80에서 간격이 5인 50으로 구성된다. 즉, 첫 번째 그룹은 [30, 80, 50]이다.
  - 두 번째 그룹은 [60, 40, 30]이고, 나머지 그룹은 각각 [90, 20, 40], [10, 10, 90], [40, 60, 80]이다.

쉘 정렬

h=5

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	30					80					50				
그 2		60					40					30			
룹 3			90					20					40		
4				10					10					90	
5					40					60					80

- 각 그룹 별로 삽입 정렬을 수행한 결과를 한 줄에 나열해보면 다음과 같다.

30 30 20 10 40 50 40 40 10 60 80 60 90 90 80

쉘 정렬

A		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	1	30					50					80				
그	2		30					40					60			
룹	3			20					20					90		
	4				10					10					90	
	5					40					60					80

그룹별 정렬 후

- 간격이 5인 그룹 별로 정렬한 결과를 살펴보면, 80과 90 같은 큰 숫자가 뒷부분으로 이동하였고, 20과 30 같은 작은 숫자가 앞부분으로 이동한 것을 관찰할 수 있다.

쉘 정렬

- 그 다음에는 간격을 5보다 작게 하여, 예를 들어 3으로 하여, 3개의 그룹으로 나누어 각 그룹별로 삽입 정렬을 수행한다. 이때에는 각 그룹에 5개의 숫자가 있다.

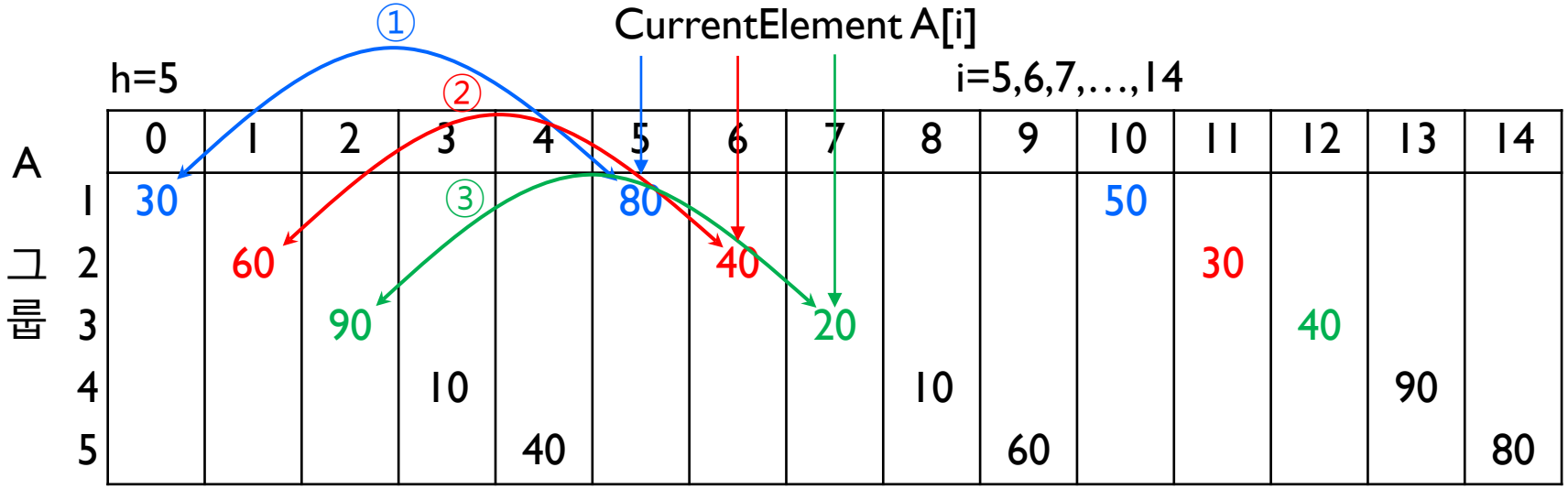
- 마지막에는 반드시 간격을 1로 놓고 수행해야 한다. 왜냐하면 다른 그룹에 속하여 서로 비교되지 않은 숫자가 있을 수 있기 때문이다. 즉, 모든 원소를 1개의 그룹으로 여기는 것이고, 이는 삽입 정렬 그 자체이다.

쉘 정렬 알고리즘

ShellSort	
입력: 크기가 n인 배열 A 출력: 정렬된 배열 A	
1	for each gap h = [$h_0 > h_1 > \ldots > h_k = 1$] // 큰 gap부터 차례로
2	for i = h to n–1 {
3	CurrentElement = A[i]
4	j = i
5	while (j ≥ h) and (A[j–h] > CurrentElement) {
6	A[j] = A[j–h]
7	j = j–h
	}
8	A[j] = CurrentElement
	}
9	return 배열 A

쉘 정렬 알고리즘

Line 1	쉘 정렬은 간격 [$h_0 > h_1 > \ldots > h_k = 1$]이 미리 정해져야 한다. 가장 큰 간격 h_0부터 차례로 간격에 따른 삽입 정렬이 line 2~8에서 수행된다. 마지막 간격 h_k는 반드시 1이어야 한다. 이는 간격에 대해서 그룹별로 삽입 정렬을 수행하였기 때문에, 아직 비교되지 않은 다른 그룹의 숫자가 있을 수 있기 때문이다.
Line 2~8	for-루프에서는 간격 h에 대하여 삽입 정렬이 수행된다. 앞에서 설명한대로 그룹별로 삽입 정렬을 수행하지만, 자리바꿈을 위한 원소 간의 비교는 다음과 같은 순서로 진행된다.

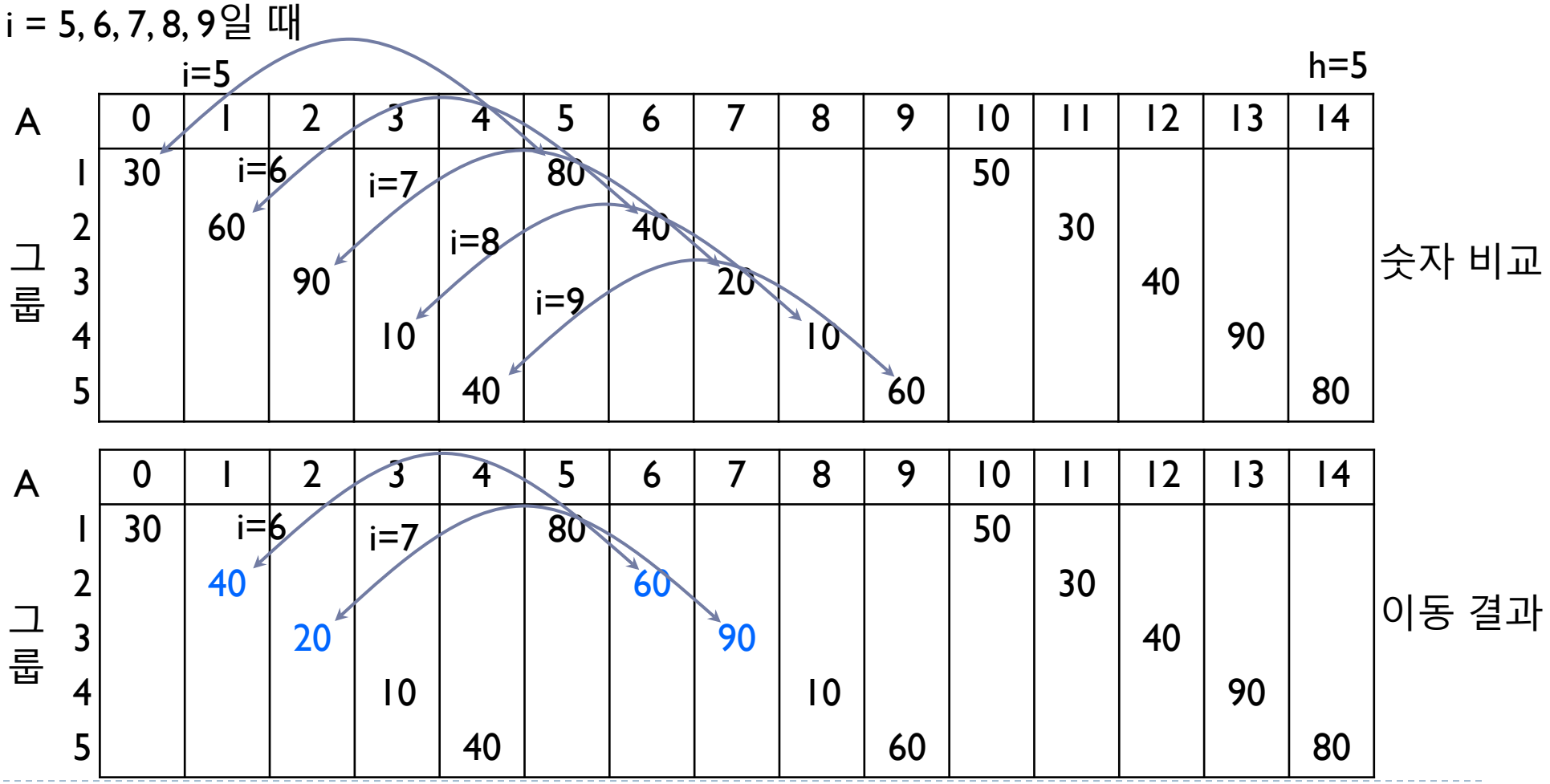


쉘 정렬 알고리즘

	즉, line 2의 for-루프가 i를 h부터 1씩 증가시켜서 CurrentElement A[i]를 자신의 그룹에 속한 원소끼리 비교하도록 조절한다.
Line 5~7	while-루프에서 CurrentElement를 정렬이 되도록 앞부분에 삽입한다. while-루프의 첫 번째 조건 (j ≥ h)는 j가 h보다 작으면 배열 인덱스 (j–h)가 음수가 되어, 배열의 범위를 벗어나는 것을 검사하기 위한 것이다. 두 번째 조건인 (A[j–h] > CurrentElement)는 (j–h)가 음수가 아니면 CurrentElement를 자신의 그룹 원소인 A[j–h]와 비교하여 A[j–h]가 크면 line 6에서 A[j–h]를 h만큼 뒤로 이동(A[j]=A[j–h])시킨다.
Line 8	while-루프의 조건이 '거짓'이 되면 line 8에서 CurrentElement를 A[j]에 저장한다. 여기서 while-루프의 조건이 '거짓'이 될 때, 첫 번째 경우는 (j–h)가 음수인 경우인데, 이는 A[j] 앞에 같은 그룹의 원소가 없다는 뜻이다. 두 번째 경우는 A[j–h]가 CurrentElement과 같거나 작은 경우이다. 따라서 두 경우 모두 line 8에서 CurrentElement를 A[j]에 저장하면 알맞게 삽입되는 것이다.

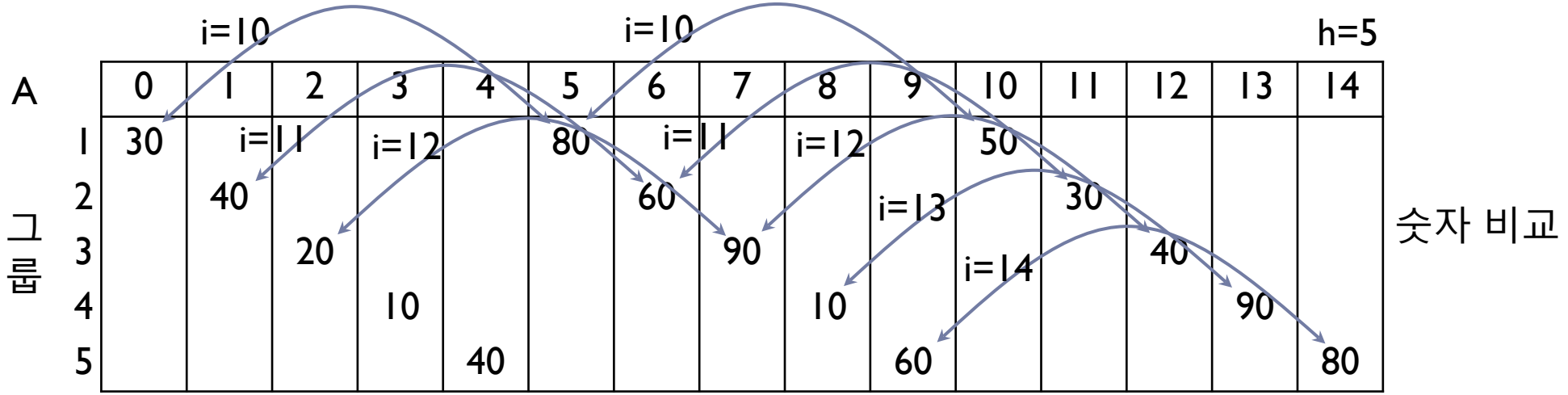
ShellSort 예제

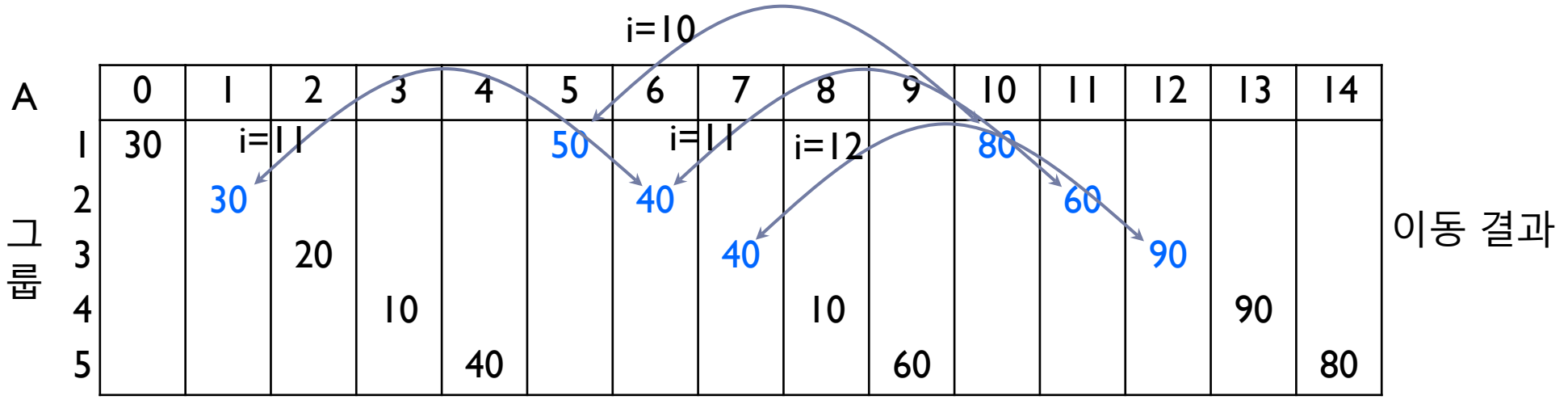
- 앞의 예제에 대해 간격이 5일 때 쉘 정렬이 수행되는 과정



ShellSort 예제

i = 10, 11, 12, 13, 14일 때





ShellSort 예제

- h=5일 때의 결과를 한 줄에 나열해보면 다음과 같다.

  30 30 20 10 40 50 40 40 10 60 80 60 90 90 80

- 이제 간격을 줄여서 h=3이 되면, 배열의 원소가 3개의 그룹으로 나누어진다.
  - 그룹1은 0번째, 3번째, 6번째, 9번째, 12번째 숫자이고,
  - 그룹2는 1번째, 4번째, 7번째, 10번째, 13번째 숫자이고,
  - 그룹3은 2번째, 5번째, 8번째, 11번째, 14번째 숫자이다.

ShellSort 예제

- 각 그룹 별로 삽입 정렬하면, 그 결과는 다음과 같다.

그룹1	그룹2	그룹3		그룹1	그룹2	그룹3
30	30	20		10	30	10
10	40	50	⇨	30	40	20
40	40	10		40	40	50
60	80	60		60	80	60
90	90	80		90	90	80

- 각 그룹 별로 정렬한 결과를 한 줄에 나열해보면 다음과 같다. 즉, 이것이 h=3일 때의 결과이다.

10 30 10 30 40 20 40 40 50 60 80 60 90 90 80

ShellSort 예제

- 마지막으로 앞의 배열에 대해 h=1일 때 알고리즘을 수행하면 아래와 같이 정렬된 결과를 얻는다. h=1일 때는 간격이 1(즉, 그룹이 1개)이므로, 삽입 정렬과 동일하다.

  10 10 20 30 30 40 40 40 50 60 60 80 80 90 90

- 쉘 정렬의 수행 속도는 간격 선정에 따라 좌우된다.

- 지금까지 알려진 가장 좋은 성능을 보인 간격은 1, 4, 10, 23, 57, 132, 301, 701이고, 701 이후는 아직 밝혀지지 않았다.

시간복잡도와 응용

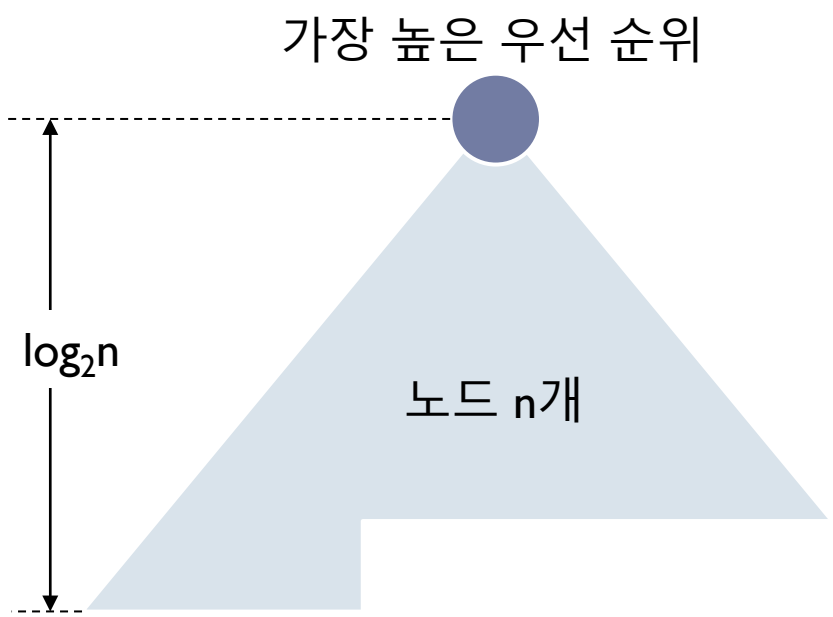
- 쉘 정렬의 최악 경우의 시간복잡도는 $O(n^2)$이다.
  - 히바드(Hibbard)의 간격인 2^k-1 (즉, $2^k-1, \ldots, 15, 7, 5, 3, 1$)을 사용하면, 시간복잡도는 $O(n^{1.5})$이 된다고 밝혀졌다.
  - 또한 다양한 실험을 통한 쉘 정렬의 시간복잡도는 $O(n^{1.25})$으로 알려지고 있다.
  - 그러나 쉘 정렬의 시간복잡도는 아직 풀리지 않은 문제로 남아 있다. 이는 가장 좋은 간격을 알아내야 하는 것이 선행되어야 하기 때문이다.
- 쉘 정렬은 입력 크기가 매우 크지 않은 경우에 매우 좋은 성능을 보인다.
  - 쉘 정렬은 임베디드(Embedded) 시스템에서 주로 사용되는데, 쉘 정렬의 특징인 간격에 따른 그룹별 정렬 방식이 하드웨어로 정렬 알고리즘을 구현하는데 매우 적합하기 때문이다.

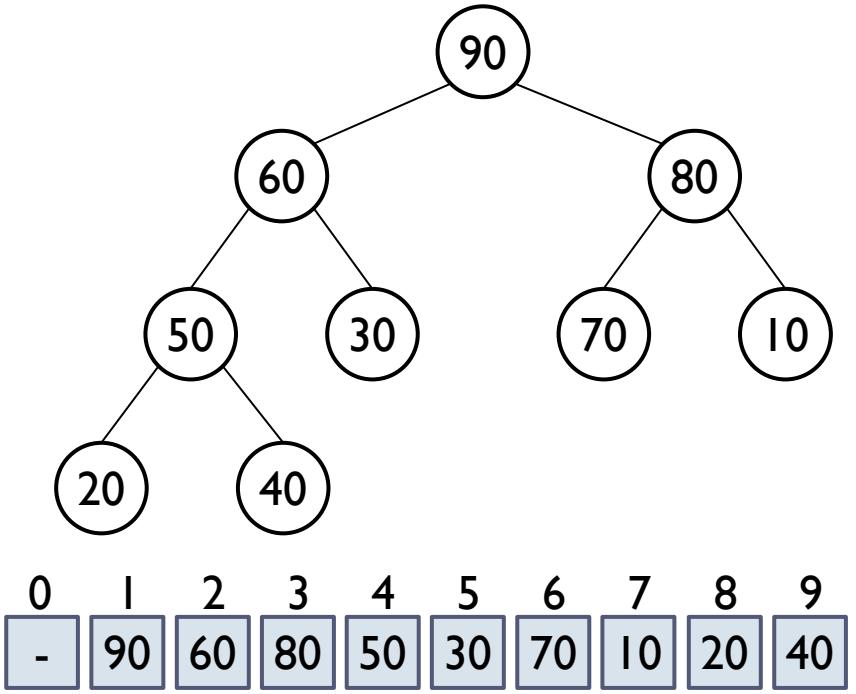
6.5 힙 정렬

- 힙(Heap)은 힙 조건을 만족하는 완전 이진트리(Complete Binary Tree)이다.
  - 힙 조건이란 각 노드의 값이 자식 노드의 값보다 커야 한다는 것을 말한다.
- 노드의 값은 우선순위(priority)라고 일컫는다.
  - 힙의 루트에는 가장 높은 우선순위가 저장되어 있다.
  - 값이 작을수록 우선순위가 높은 경우에는 가장 작은 값이 루트에 저장된다.

힙 (Heap)

- n개의 노드를 가진 힙은 완전 이진트리이므로, 힙의 높이가 $\log_2 n$이며, 노드들을 빈 공간 없이 배열에 저장할 수 있다.
- 아래의 그림은 힙의 노드들이 배열에 저장된 모습을 보여주고 있다.





힙 (Heap)

- 배열 A에 힙을 저장한다면, A[0]은 비워 두고, A[1]부터 A[n]까지에 힙 노드들을 트리의 층별로 좌우로 저장한다.
  - 루트의 90이 A[1]에 저장되고,
  - 그 다음 층의 60과 80이 각각 A[2]와 A[3]에 저장되며,
  - 그 다음 층의 50, 30, 70, 10이 A[4]에서 A[7]에 각각 저장되고,
  - 마지막으로 20과 40이 A[8]과 A[9]에 저장된다.
- 트리에서 부모 노드와 자식 노드의 관계를 배열의 인덱스로 쉽게 표현할 수 있다.
  - A[i]의 부모 노드는 A[i/2]이다.
    - 단, i가 홀수일 때, i/2에서 정수 부분만을 취한다. 예를 들어, A[7]의 부모 노드는 A[7/2] = A[3]이다.
  - A[i]의 왼쪽 자식 노드는 A[2i]이고, A[i]의 오른쪽 자식 노드는 A[2i+1]이다. 예를 들면, A[4]의 왼쪽 자식 노드는 A[2i] = A[2x4] = A[8]이고, 오른쪽 자식 노드는 A[2i+1] = A[2x4+1] =A[9]이다.

힙 정렬: 아이디어

- 힙 정렬(Heap Sort)은 힙 자료 구조를 이용하는 정렬 알고리즘이다.
  1. 오름차순의 정렬을 위해 입력 배열을 먼저 큰 숫자가 높은 우선순위를 가지는 최대힙(maximum heap)을 만든다.
  2. 힙의 루트에는 가장 큰 수가 저장되므로, 루트의 숫자를 힙의 가장 마지막 노드에 있는 숫자와 바꾼다. 즉, 가장 큰 수를 배열의 가장 끝으로 이동시킨 것이다.
  3. 루트에 새로 저장된 숫자로 인해 위배된 힙 조건을 해결하여 힙 조건을 만족시키고, 힙 크기를 1개 줄인다.
  4. 이 과정을 반복하여 나머지 숫자를 정렬한다.

- 다음은 이러한 과정에 따른 힙 정렬 알고리즘이다.

힙 정렬 알고리즘

HeapSort	
입력: 입력이 A[1]부터 A[n]까지 저장된 배열 A 출력: 정렬된 배열 A	
1	배열 A의 숫자에 대해서 힙 자료 구조를 만든다.
2	heapSize = n // 힙의 크기를 조절하는 변수
3	for i = 1 to n–1
4	A[1] ↔ A[heapSize] // 루트와 힙의 마지막 노드를 교환한다.
5	heapSize = heapSize–1 // 힙의 크기를 1 감소시킨다.
6	DownHeap( ) // 위배된 힙 조건을 만족시킨다.
7	return 배열 A

힙 정렬 알고리즘

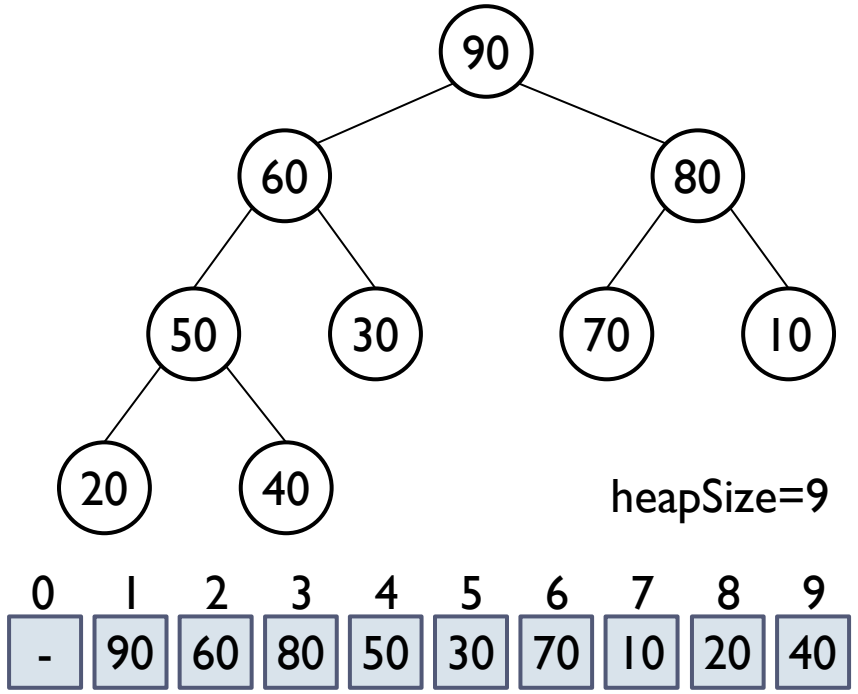
Line 1	배열 A[1..n]을 힙으로 만든다.
Line 2	현재의 힙의 크기를 나타내는 변수인 heapSize를 n으로 초기화시킨다.
Line 3~6	for-루프는 (n–1)번 수행되는데, 이는 루프가 종료된 후에는 루트인 A[1] 홀로 힙을 구성하고 있고, 또 가장 작은 수이므로 루프를 수행할 필요가 없기 때문이다.
Line 4	루트와 힙의 마지막 노드와 교환한다. 즉, 현재의 힙에서 가장 큰 수와 현재 힙의 맨 마지막 노드에 있는 숫자와 교환하는 것이다.
Line 5	힙의 크기를 1 줄인다.

힙 정렬 알고리즘

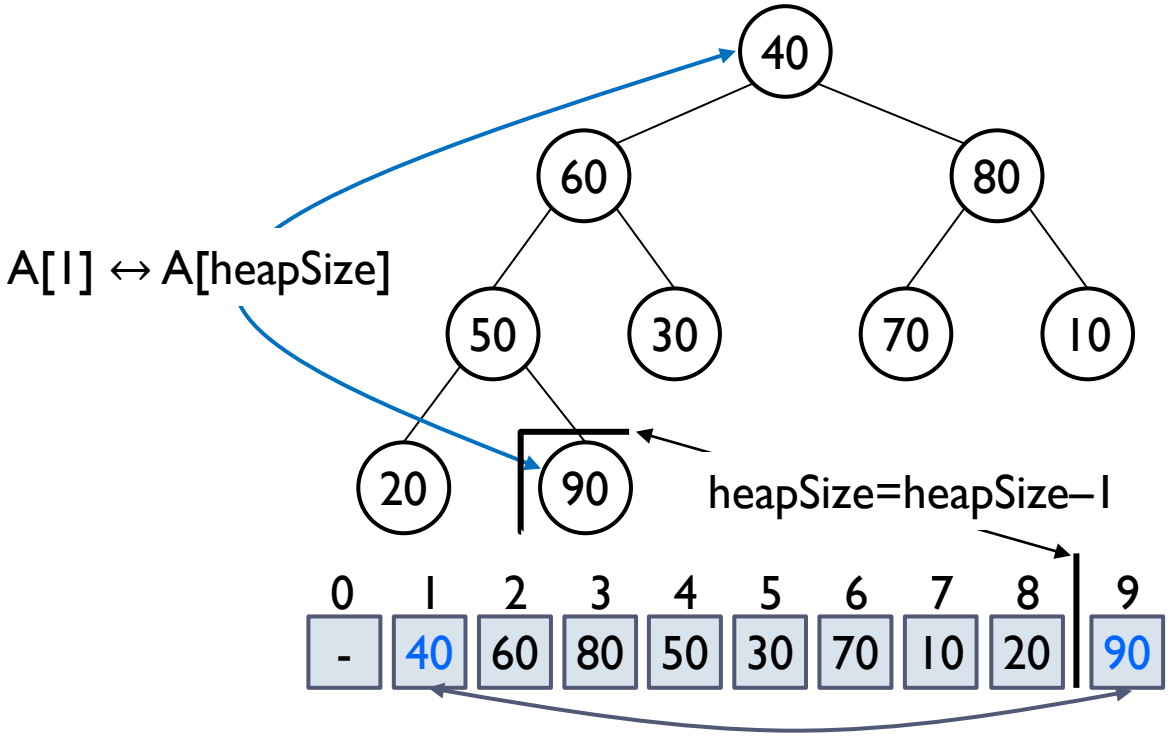
Line 6	Line 4에서 힙의 마지막 노드와 힙의 루트를 바꾸어 놓았기 때문에 새로이 루트에 저장된 값이 루트의 자식 노드의 값보다 작아서 힙 조건이 위배된다. Line 6 에서는 위반된 힙 조건을 DownHeap을 수행시켜서 해결한다.

- DownHeap
  - 루트가 r을 가지고 있다고 가정하자.
  - 먼저 루트의 r과 자식 노드들 중에서 큰 것을 비교하여 큰 것과 r을 바꾼다.
  - 다시 r을 자식 노드들 중에서 큰 것과 비교하여 힙 조건이 위배되면, 앞서 수행한 대로 큰 것과 r을 교환한다.
  - 힙 조건이 만족될 때까지 이 과정을 반복한다.

HeapSort 예제

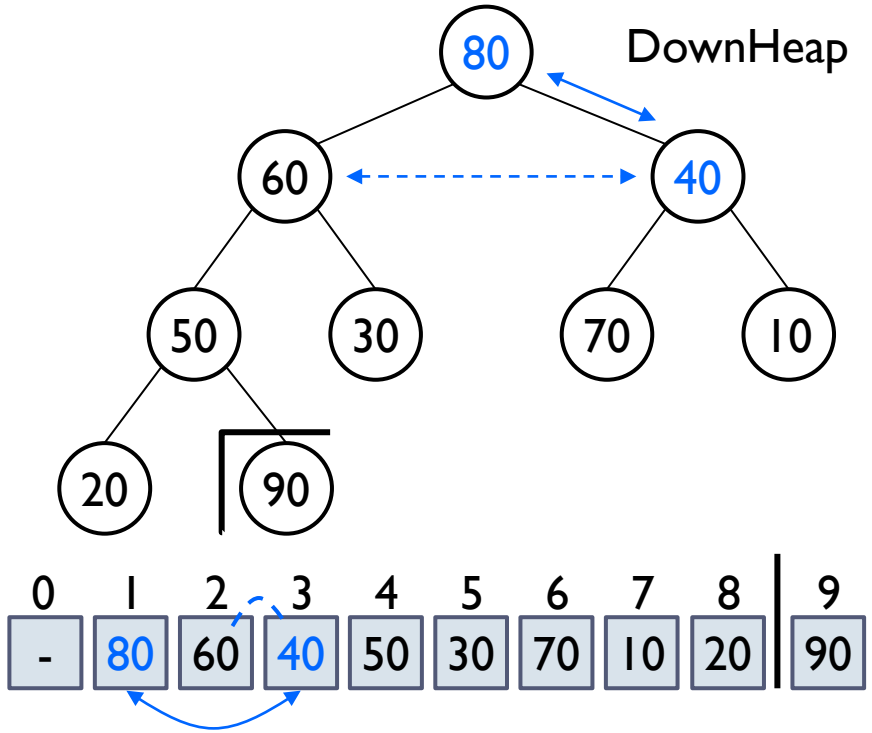

90
60
80
50
30
70
10
20
40
heapSize=9
0 1 2 3 4 5 6 7 8 9
- 90 60 80 50 30 70 10 20 40

HeapSort 예제



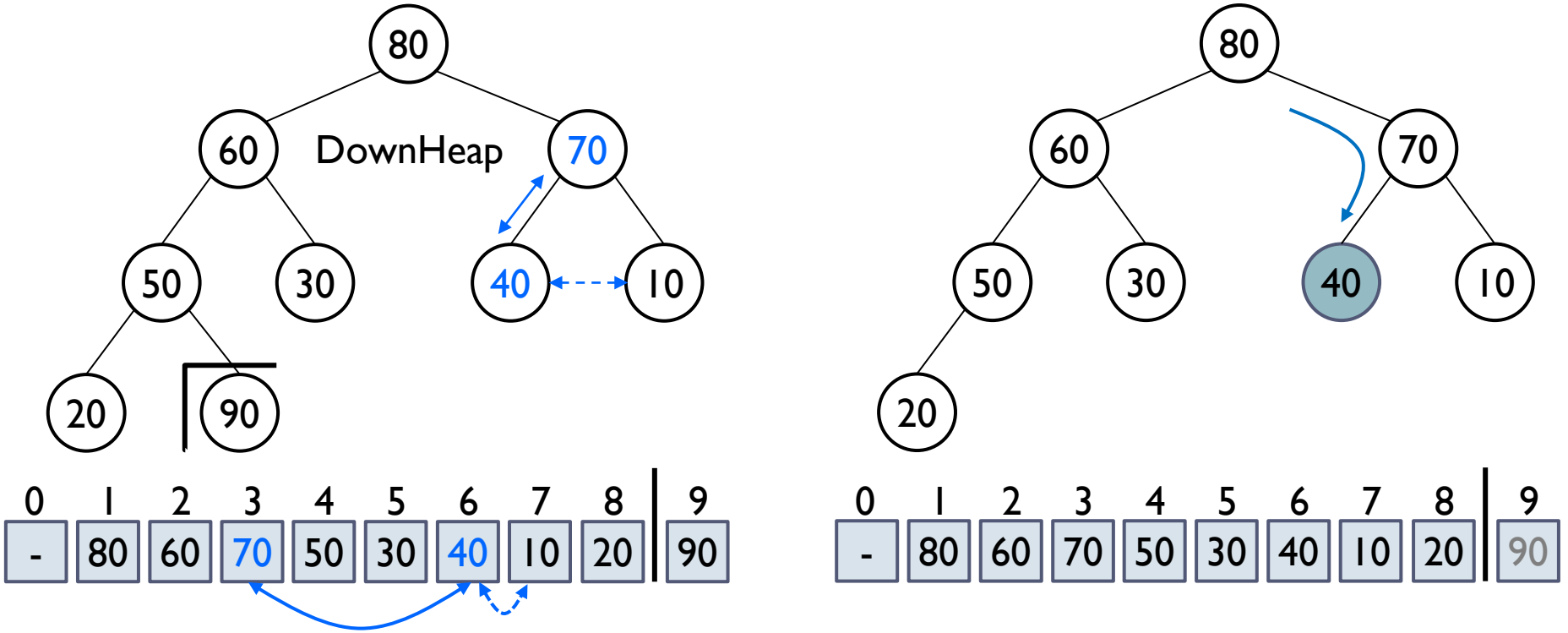
- Line 4에서 힙의 마지막 노드 40과 루트 90을 바꾸고, Line 5에서 힙의 노드 수(heapsize)가 1개 줄어든 것을 보이고 있다.

HeapSort 예제



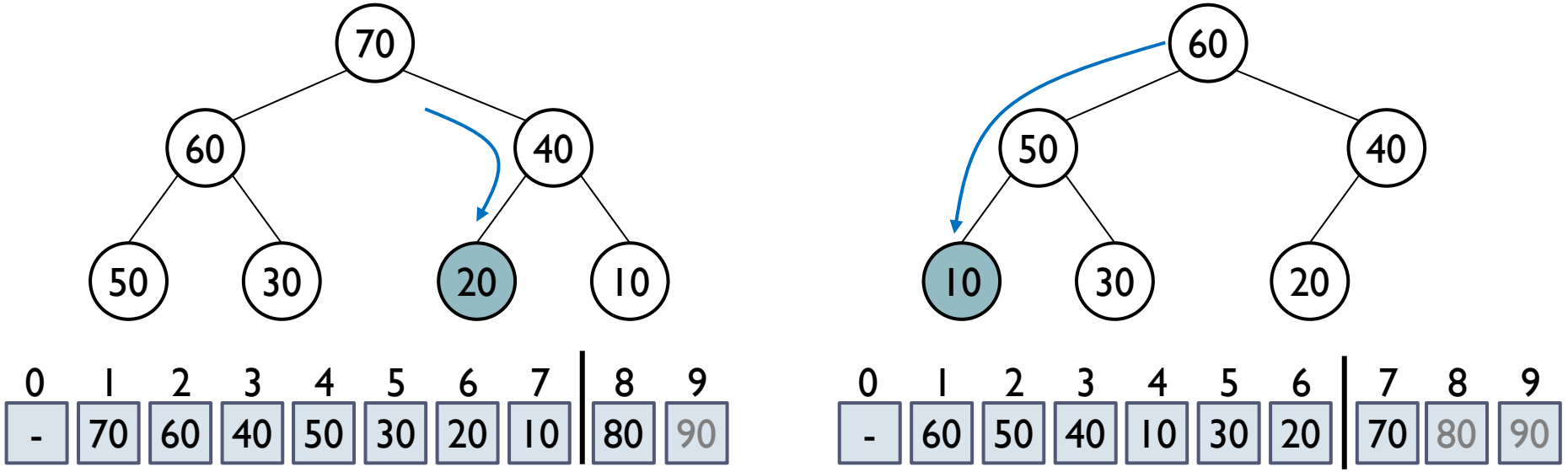
- 새로이 루트에 저장된 40이 루트의 자식 노드들(60과 80)보다 작아서 힙 조건이 위배되므로 자식 노드들 중에서 큰 자식 노드 80과 루트 40이 교환된 것을 보여준다.

HeapSort 예제



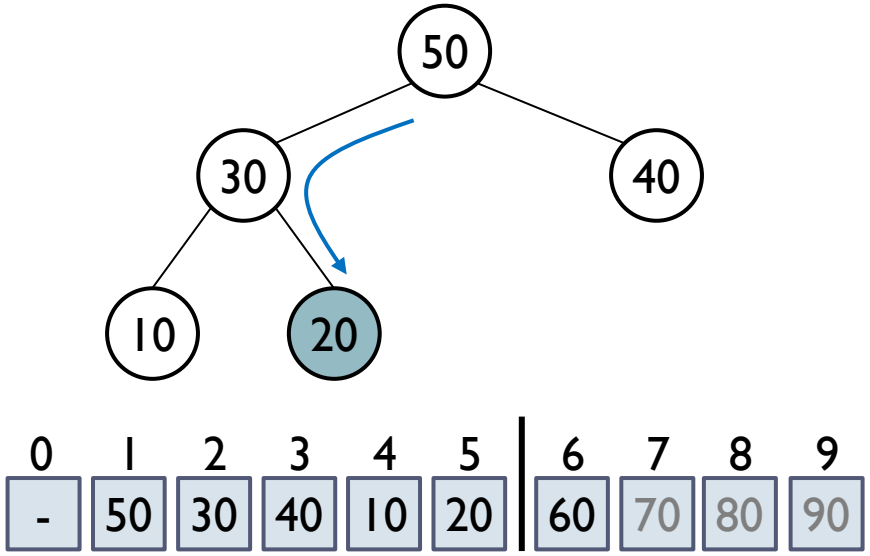
- 40은 다시 자식 노드들 (70과 10) 중에서 큰 자식 70과 비교하여, 힙 조건이 위배되므로 70과 40을 서로 바꾼다.
- 그 다음에는 더 이상 자식 노드가 없으므로 힙 조건이 만족되어 DownHeap을 종료한다.

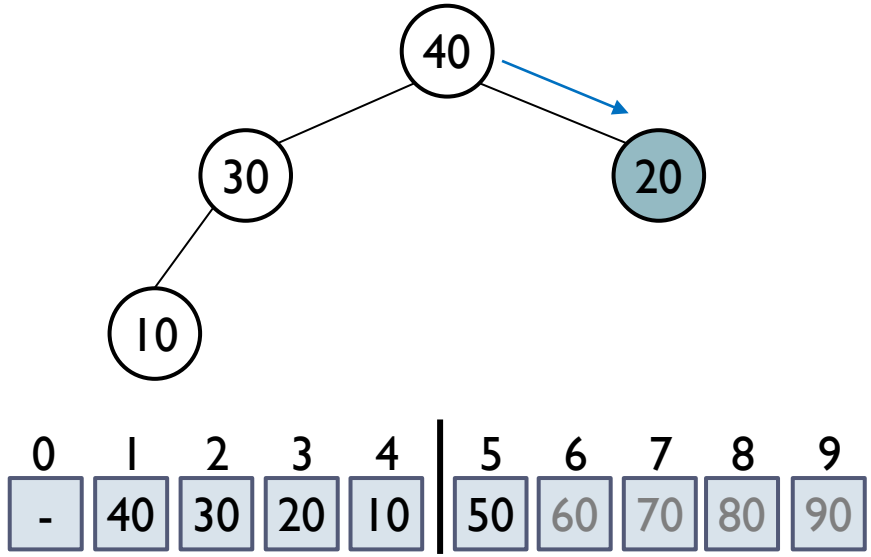
HeapSort 예제



- 80이 20과 교환된 후 DownHeap을 수행한 결과(왼쪽)
- 70이 10과 교환된 후 DownHeap을 수행한 결과(오른쪽)

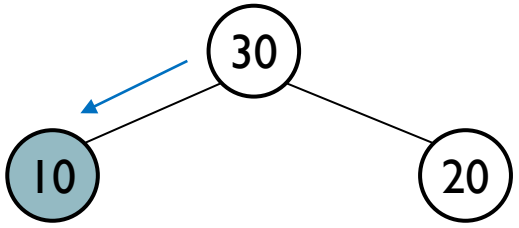
HeapSort 예제

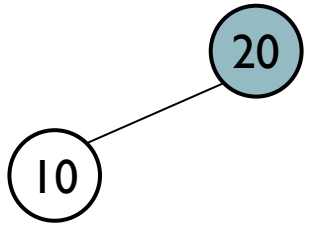




- 60이 20과 교환된 후 DownHeap을 수행한 결과(왼쪽)
- 50이 20과 교환된 후 DownHeap을 수행한 결과(오른쪽)

HeapSort 예제



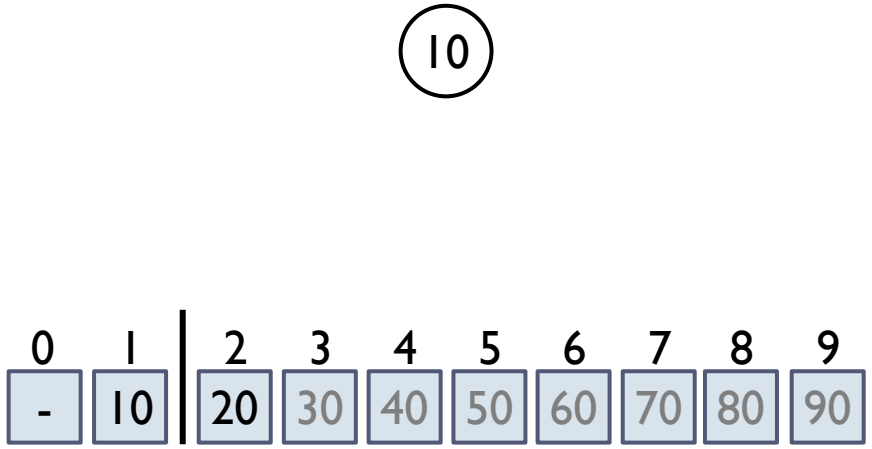


0	1	2	3	4	5	6	7	8	9
-	30	10	20	40	50	60	70	80	90

0	1	2	3	4	5	6	7	8	9
-	20	10	30	40	50	60	70	80	90

- 40이 10과 교환된 후 DownHeap을 수행한 결과(왼쪽)
- 30이 10과 교환된 후 DownHeap을 수행한 결과(오른쪽)

HeapSort 예제



- 마지막에 힙의 크기가 1이 되면 힙 정렬을 마친다.

HeapSort 요약

- for-루프를 반복할 때마다 힙에서 가장 큰 수를 힙의 마지막 노드에 있는 수와 교환하고, 힙 크기를 1개 줄임으로써 힙에 속하지 않는 배열의 뒷부분에는 가장 큰 수부터 차례로 왼쪽 방향으로 저장된다.
- 선택 정렬에서 최솟값을 찾는 대신 최댓값을 찾아서 배열의 뒷부분으로부터 정렬하는 것과 같다.
- 선택 정렬과의 차이점은 힙 정렬은 힙 자료구조를 이용하여 최댓값을 찾으나 선택 정렬은 순차 탐색으로 최댓값을 찾는 것이다.

시간복잡도

- Line 1에서 힙을 만드는 데에는 O(n) 시간이 걸린다.
- Line 2는 변수를 초기화하는 것이므로 O(1) 시간이 걸린다.
- Line 3~6의 for-루프는 (n–1)번 수행되고, 루프 내부에서는 line 4~5가 각각 O(1) 시간이 걸리고 DownHeap은 최악의 경우 리프 노드까지 내려가며 교환할 때이고, 힙의 높이는 $\log_2 n$를 넘지 않기 때문에 O(logn) 시간이 걸린다.
- 힙 정렬의 시간복잡도는 O(n)+(n–1)xO(logn) = O(nlogn)이다.

6.6 정렬 문제의 하한

- 비교정렬 (Comparison Sort)
  - 버블 정렬, 선택 정렬, 삽입 정렬, 쉘 정렬, 힙 정렬, 합병 정렬, 퀵 정렬의 공통점은 숫자의 비교가 부분적이 아닌 숫자 대 숫자로 이루어진다. 즉, 두 수 a, b 를 비교하기 위해서 사용할 수 있는 연산은 다음과 같다.

    $$a > b, \quad a \geq b, \quad a = b, \quad a \leq b, \quad a < b$$

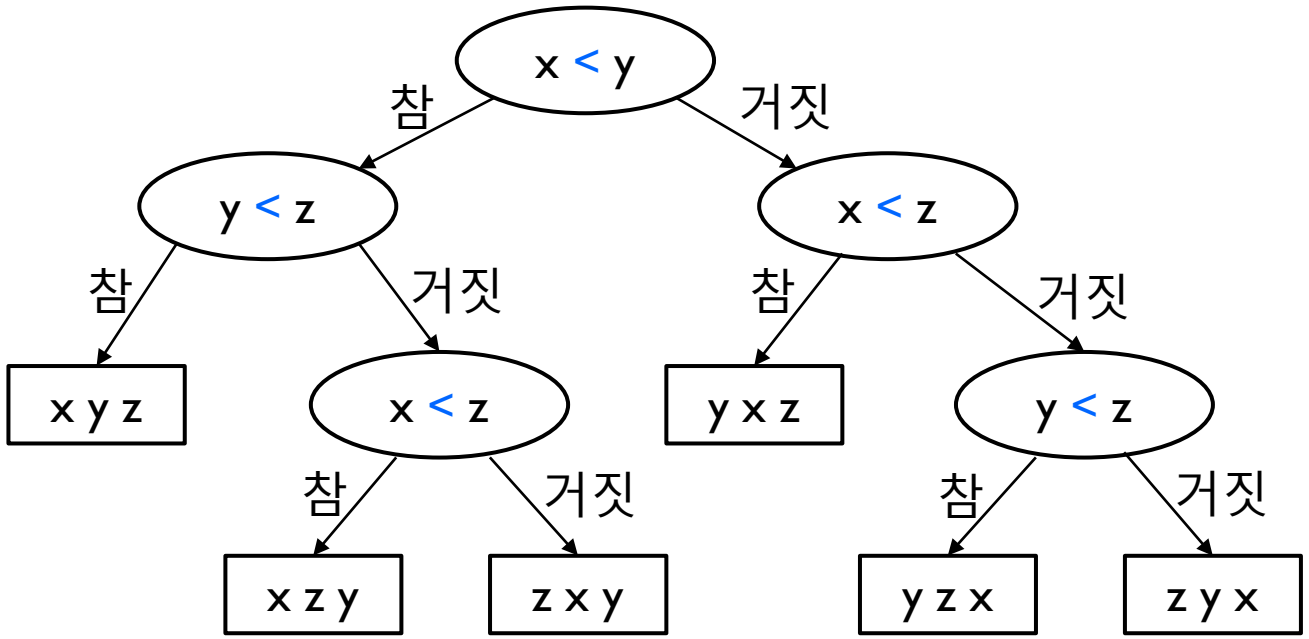
    이러한 정렬을 비교정렬이라고 한다.

- 비교정렬의 하한: Ω(nlogn)

- 선형 시간복잡도 정렬 방법 (비교정렬이 아님)
  - Radix Sort (6.7 기수 정렬): 숫자들을 한 자리씩 부분적으로 비교
  - Counting Sort
  - Bucket Sort

비교정렬의 하한

- 결정트리 (decision tree)
  - 다음 그림은 3개의 서로 다른 숫자 x, y, z에 대해서 비교정렬에 필요한 모든 경우의 숫자 대 숫자 비교를 보이고 있다. 결정트리의 단말(leaf) 노드 수는 3! = 6 이다.



비교정렬의 하한

- n개의 서로 다른 숫자에 대해서 비교 정렬을 수행할 경우
  - 결정트리의 단말 노드 수는 총 n! 이다.
  - 결정트리의 높이(height)를 h라고 하자. 높이 h인 이진트리는 최대 2^h개의 단말 노드를 가질 수 있기 때문에 $2^h \geq n!$ 가 성립한다.
    - 높이 h인 포화(Perfect) 이진트리가 2^h개의 단말 노드를 가짐
  - $2^h \geq n!$ 에서 $h \geq \log(n!) \geq \log(n/2)^{n/2} = (n/2)\log(n/2) = \Omega(n\log n)$
  - 따라서, 비교 정렬의 시간복잡도 하한은 $\Omega(n\log n)$이다.
  - 즉, $O(n\log n)$보다 빠른 시간복잡도를 가진 비교정렬 알고리즘은 존재하지 않는다.

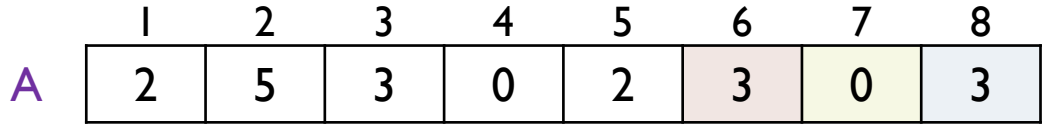
Counting Sort 알고리즘

CountingSort

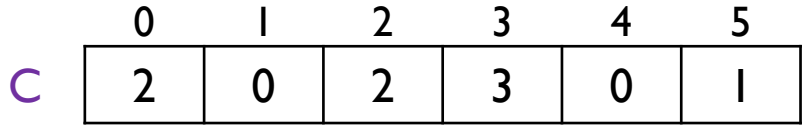
입력: 0~r 범위의 숫자 n개가 저장된 배열 A (단, A의 인덱스는 1~n)
출력: 정렬된 배열 B

```
1   for i = 0 to r    // 카운터 배열 C를 0으로 초기화한다.
2       C[i] = 0
3   for j = 1 to n    // C[i]는 값이 i인 배열 A의 원소 개수를 저장한다.
4       C[A[j]] = C[A[j]] + 1
5   for i = 1 to r    // C[i]는 값이 i보다 같거나 작은 A의 원소 개수를 저장한다.
6       C[i] = C[i] + C[i–1]
7   for j = n downto 1          // 입력 배열 A의 원소를 출력 배열 B에 정렬한다.
8       B[C[A[j]]] = A[j]       // A[j]의 원소를 B에 삽입한다.
9       C[A[j]] = C[A[j]]–1     // A[j]값에 해당하는 카운터를 감소시킨다.
10  return 배열 B
```

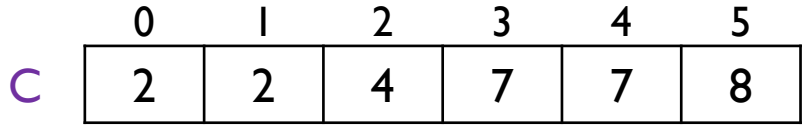
CountingSort 예제



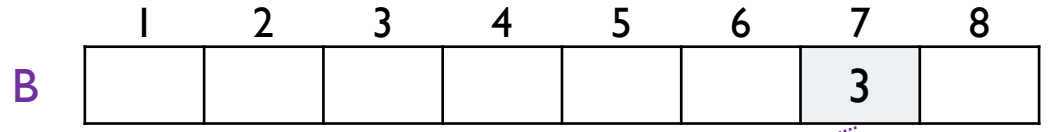
입력



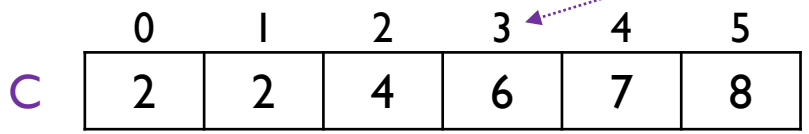
Line 3~4 실행 후



Line 5~6 실행 후

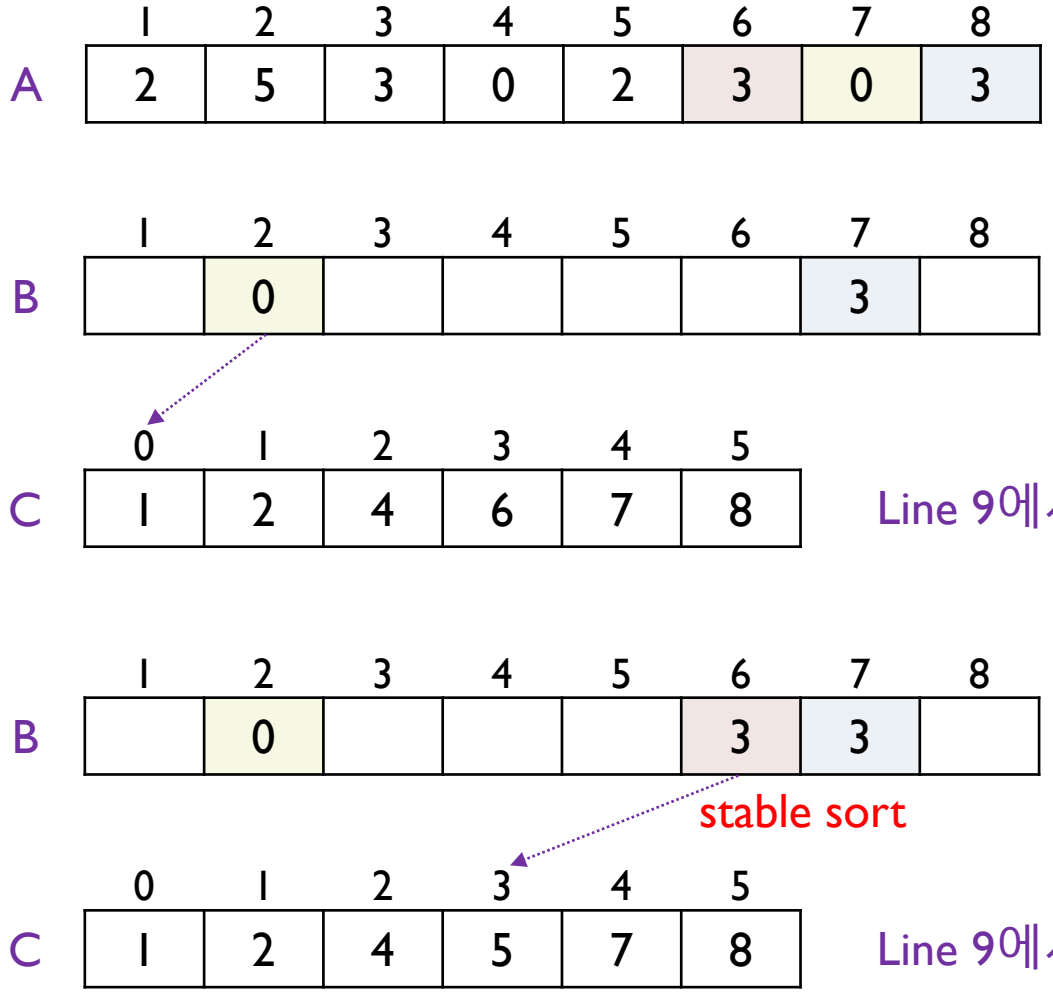


Line 8에서 A[8]의 원소를
B[C[A[8]]] = B[C[3]] = B[7]에 삽입



Line 9에서 C[A[8]] = C[3] 카운터를 감소시킴

CountingSort 예제



시간복잡도

- 최종 출력

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

출력

- 시간복잡도
  - 4개의 for-루프는 각각 O(n) 또는 O(r)의 시간복잡도를 가진다.
  - 따라서 Counting sort의 시간복잡도는 O(n+r)이다.

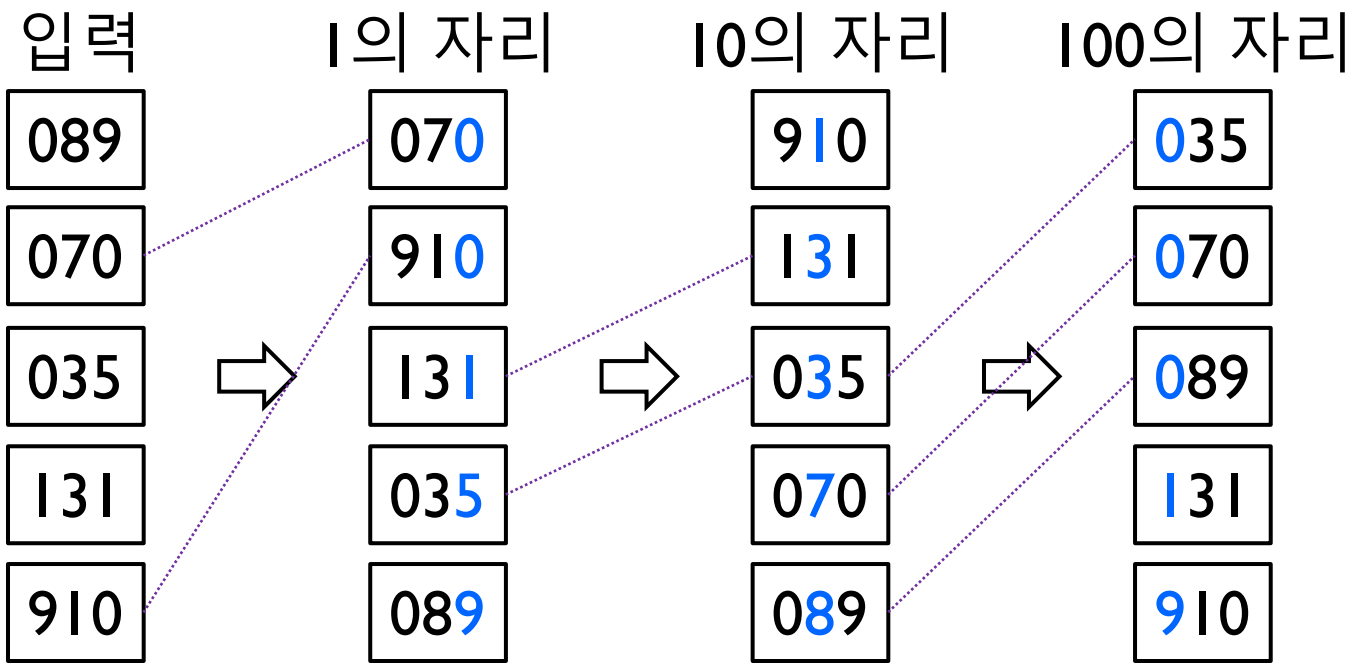
- 안정한 정렬(stable sort) 알고리즘
  - 입력에 중복된 숫자가 있을 때, 정렬 후에도 중복된 숫자의 순서가 입력에서의 순서와 동일하다.

6.7 기수 정렬

- 기수 정렬(Radix Sort)이란 비교정렬이 아니고, 숫자를 부분적으로 비교하는 정렬 방법이다.
- 기(radix)는 특정 진수를 나타내는 숫자들이다.
  - 예를 들어, 10진수의 기는 0, 1, 2, …, 9이고, 2진수의 기는 0, 1이다.
- 기수 정렬은 제한적인 범위 내에 있는 숫자에 대해서 각 자릿수별로 정렬하는 알고리즘이다.
- 기수 정렬은 비교정렬보다 빠른 성능을 얻을 수 있다.

LSD 기수 정렬

- Least significant digit 부터 각 자리별로 stable sort 수행



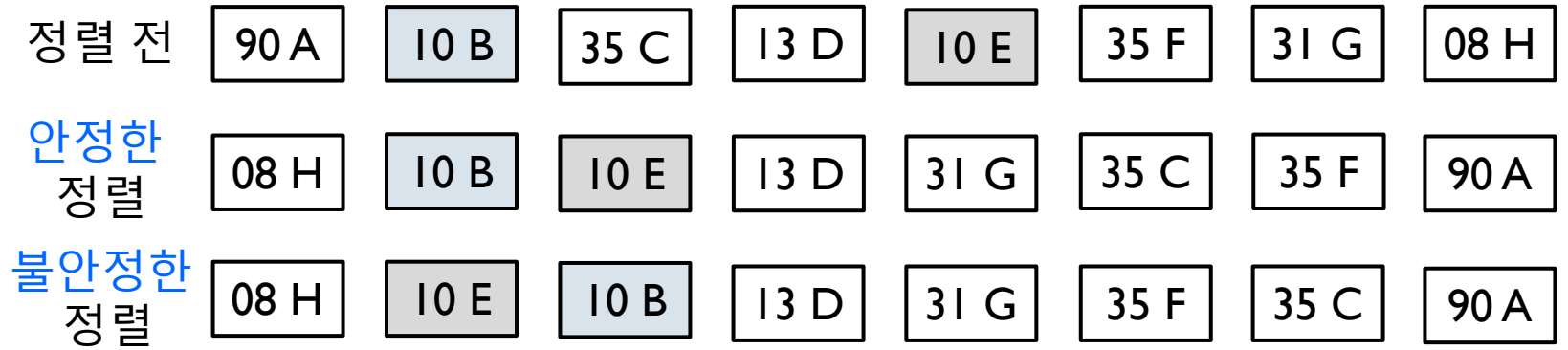
  - 안정한 정렬(stable sort) 알고리즘: 입력에 중복된 숫자가 있을 때, 정렬 후에도 중복된 숫자의 순서가 입력에서의 순서와 동일하다.

LSD 기수 정렬

- 앞의 예제와 같이 5개의 3자리 십진수가 입력으로 주어지면, 가장 먼저 각 숫자의 1의 자리만 비교하여 작은 수부터 큰 수를 정렬한다.
- 그 다음에는 10의 자리만을 각각 비교하여 정렬한다. 이때 반드시 지켜야 할 순서가 있다.
- 예제에서 10의 자리가 3인 131과 035가 있는데, 10의 자리에 대해 정렬될 때 131이 반드시 035 위에 위치하여야 한다.
  - 10의 자리가 같은데 왜 035가 131 위에 위치하면 안 되는 것일까?
  - 그 답은 1의 자리에 대해 정렬해 놓은 것이 아무 소용이 없게 되기 때문이다.

Stable sort

- 입력에 중복된 숫자가 있을 때, 정렬된 후에도 중복된 숫자의 순서가 입력에서의 순서와 동일하면 정렬 알고리즘이 안정성(stability)을 가진다고 한다.
- 다음의 예에서 2개의 10이 입력에 있을 때, 안정한 정렬(stable sort) 알고리즘은 중복된 숫자에 대해 입력에서 앞서 있던 숫자가 정렬 후에도 앞서 있고, 불안정한 정렬 알고리즘은 정렬 후에 그 순서가 반드시 지켜지지 않는다.



LSD 기수 정렬 알고리즘

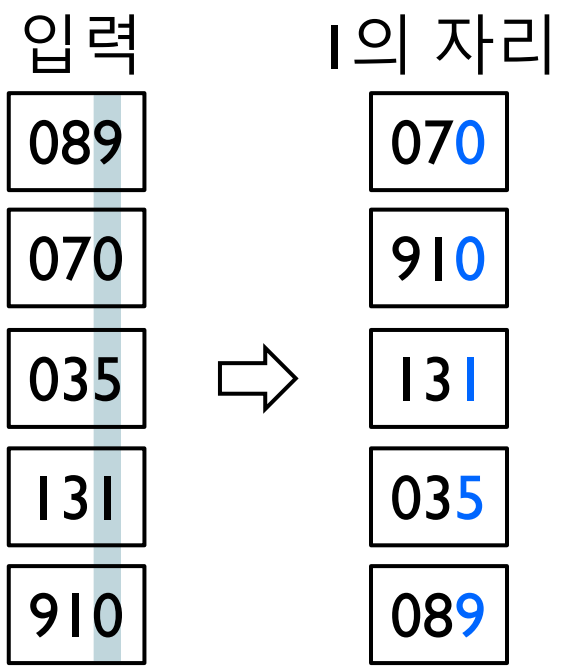
RadixSort	
입력: n개의 r진수의 k자리 숫자가 저장된 배열 A 출력: 정렬된 배열 A	
1 2 3	for i = 1 to k 　　각 숫자의 i자리 숫자에 대해 안정한 정렬(stable sort)을 수행한다. return 배열 A

LSD 기수 정렬 알고리즘

Line 1	for-루프에서는 1의 자리부터 k자리까지 차례로 안정한 정렬을 반복한다.
Line 2	각 숫자의 i자리 수만에 대해 다음과 같이 정렬한다. 입력 숫자가 r진수라면, i자리가 • '0'인 수의 개수 • '1'인 수의 개수 • … • '(r–1)'인 수의 개수를 각각 계산하여 i자리가 '0'인 숫자로부터 '(r–1)'인 숫자까지 차례로 안정성에 기반을 두어 정렬한다.

LSD 기수 정렬 알고리즘

- 예제에서 i=1일 때, 입력의 각 숫자의 1의 자리 수만을 보면, 9, 0, 5, 1, 0이므로 1의 자리가 '0'인 숫자가 2개, '1'인 숫자가 1개, '5'인 숫자가 1개, '9'인 숫자가 1개이다.
- 따라서 1의 자리가 '0'인 숫자 070과 910, '1'인 숫자 131, '5'인 숫자 035, 마지막으로 '9'인 숫자 089 순으로 입력의 숫자들이 정렬 된다.



시간복잡도

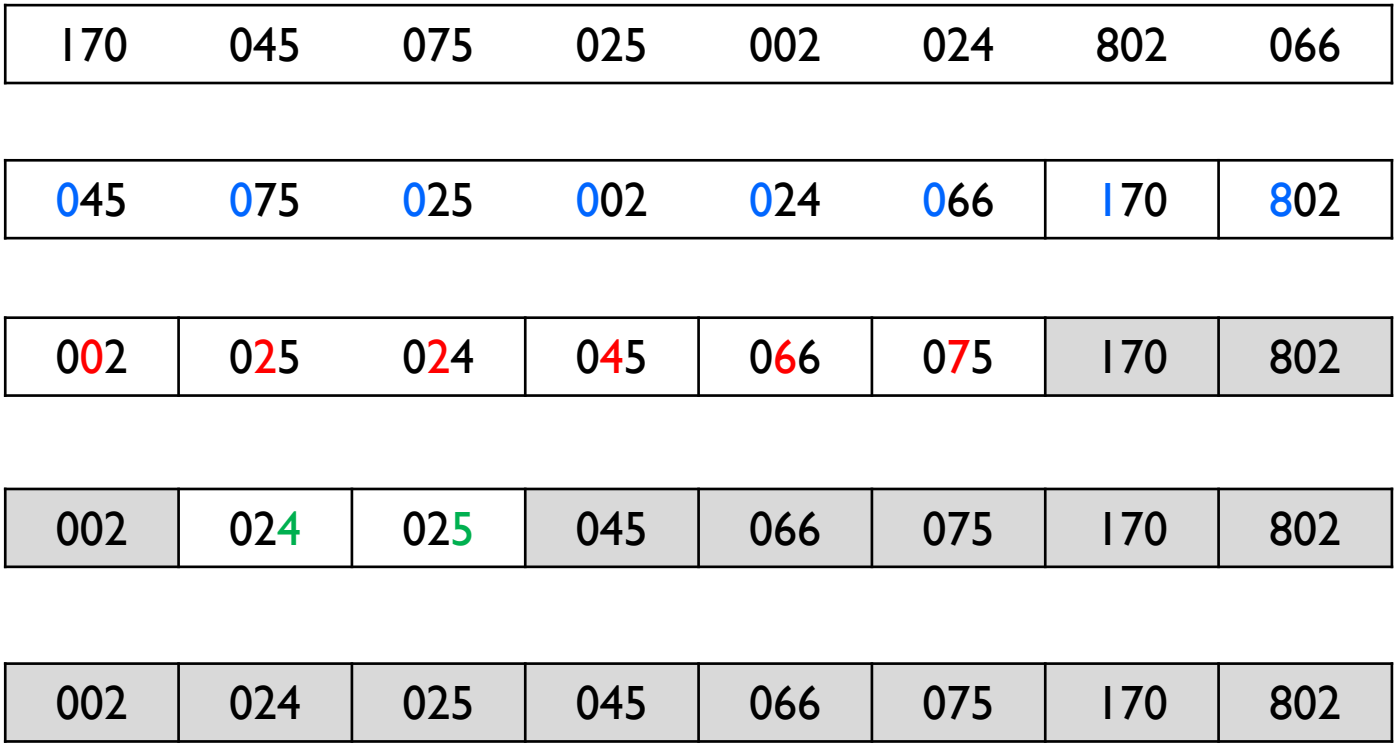
- 먼저 for-루프가 k번 반복된다. 단, k는 입력 숫자의 최대 자릿수이다.
- (Line 2에서 counting sort를 사용할 경우) 한 번 루프가 수행될 때 n개의 숫자의 i자리 수를 읽으며, r개로 분류하여 개수를 세고, 그 결과에 따라 숫자가 이동하므로 O(n+r) 시간이 걸린다.
- 따라서 시간복잡도는 O(k(n+r))이다.
- 여기서 k나 r이 입력 크기인 n보다 크지 않으면, 시간복잡도는 O(n)이 된다.

시간복잡도

- 예를 들어, 4바이트 숫자를 정렬할 때, $n=2^{32}$이라고 가정하고, $r=2^4$, 즉 16진수라고 하면, 입력 숫자는 k=8, 즉 8자리의 16진수가 된다.
- 시간복잡도 $O(k(n+r)) = O(8\times(2^{32}+16))$에서 8이나 16은 2^{32}과 비교하면 매우 작은 수이므로, 즉, 8이나 16을 상수로 취급할 수 있으므로, 시간복잡도는 $O(2^{32}) = O(n)$이다.
- 따라서, 기수 정렬은 계좌 번호, 날짜, 주민등록번호 등 대용량의 상용 데이터베이스 정렬에 응용된다.

MSD 기수 정렬

- Most significant digit 부터 크기 순으로 partition을 반복

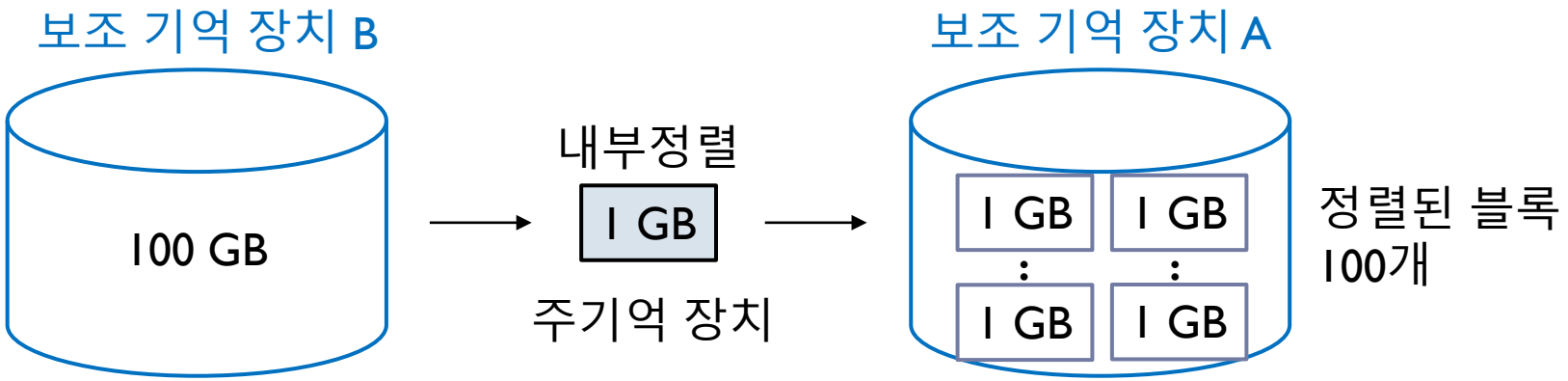


6.8 외부정렬

- 외부정렬(External Sort)은 입력 크기가 매우 커서 읽고 쓰는 시간이 오래 걸리는 보조 기억 장치에 입력을 저장할 수 밖에 없는 상태에서 수행되는 정렬을 일컫는다.
- 반면에 앞서 언급된 모든 정렬 알고리즘들은 내부정렬(Internal Sort)이라고 하는데, 이는 입력이 주기억 장치(내부 메모리)에 있는 상태에서 정렬이 수행되기 때문이다.
- 예를 들면, 주기억 장치의 용량이 1GB이고, 입력 크기가 100GB라면, 내부정렬 알고리즘으로 정렬할 수 없다.

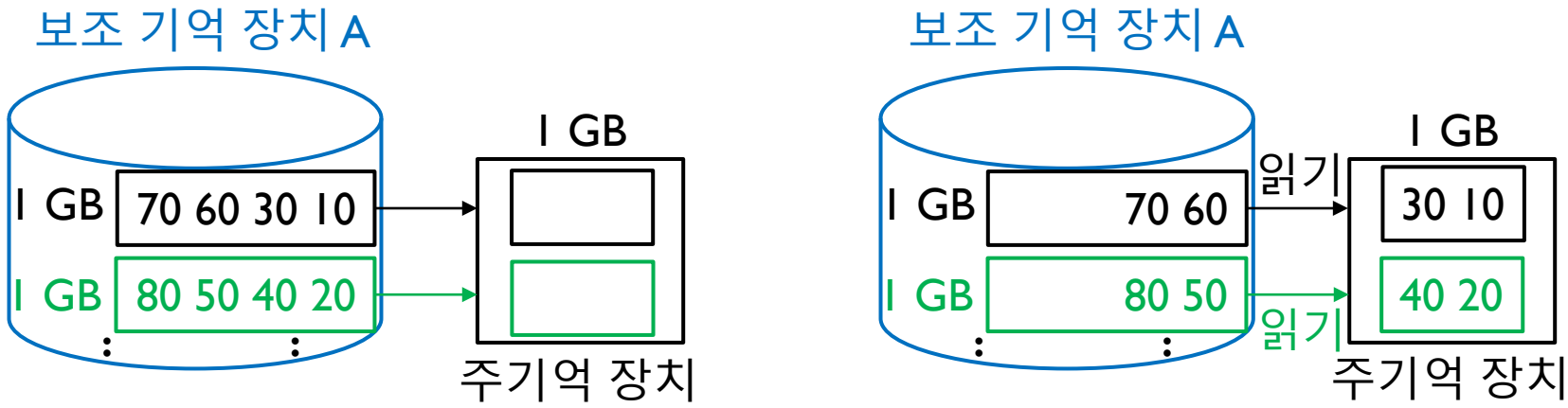
외부정렬

- 외부정렬은 입력을 분할하여 주기억 장치에 수용할 만큼의 데이터에 대해서만 내부정렬을 수행하고, 그 결과를 보조 기억 장치에 일단 다시 저장한다.
  - 즉, 100GB의 데이터를 1GB 만큼씩 주기억 장치로 읽어 들이고, 퀵 정렬과 같은 내부정렬 알고리즘을 통해 정렬한 후, 다른 보조 기억 장치에 저장한다.
  - 이를 반복하면, 원래의 입력이 100개의 정렬된 블록으로 분할되어 보조 기억 장치에 저장된다.



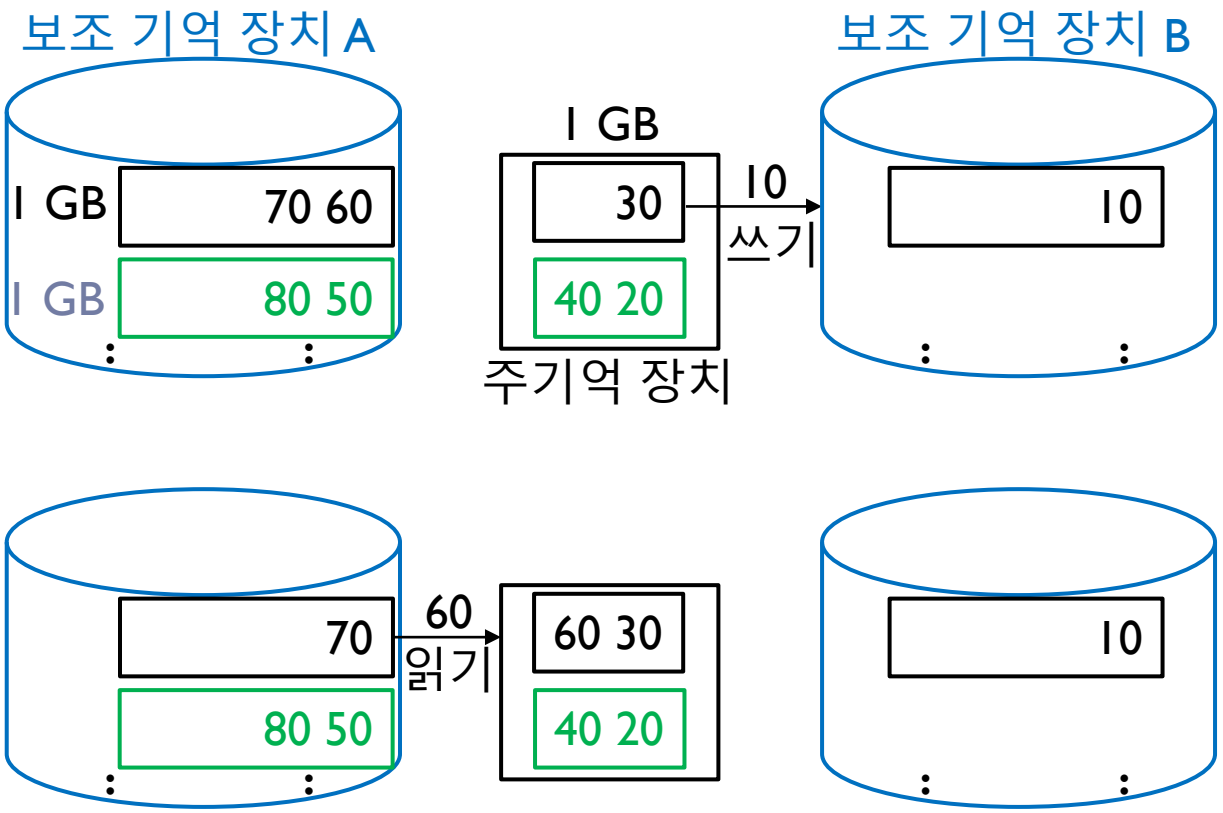
외부정렬

- 그 다음 과정은 정렬된 블록들을 하나의 정렬된 거대한 (크기가 100GB인) 블록으로 만드는 것이다.
- 이를 위해 합병(merge)을 반복 수행한다. 즉, 블록들을 부분적으로 주기억 장치에 읽어 들이고 합병을 수행하여 부분적으로 보조 기억 장치에 쓰는 과정을 반복한다.
- 아래의 그림은 블록을 부분적으로 읽어 들인 상황을 보이고 있다.

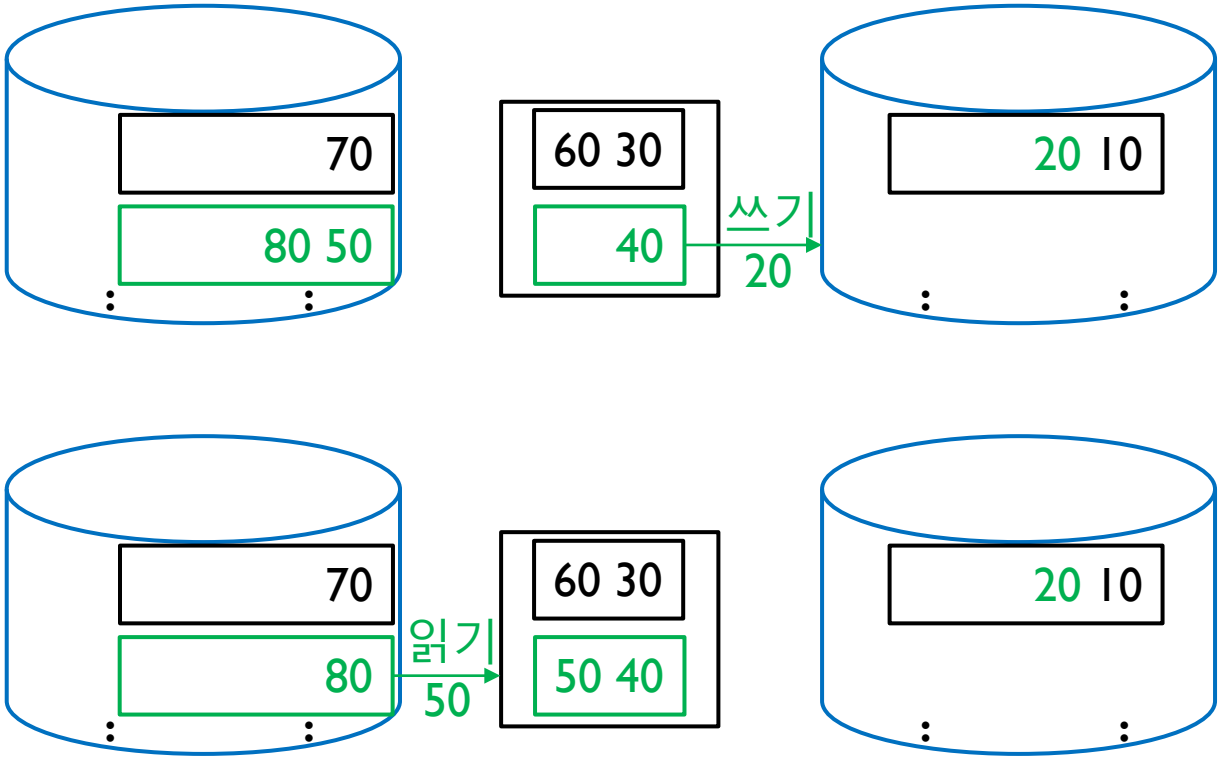


외부정렬

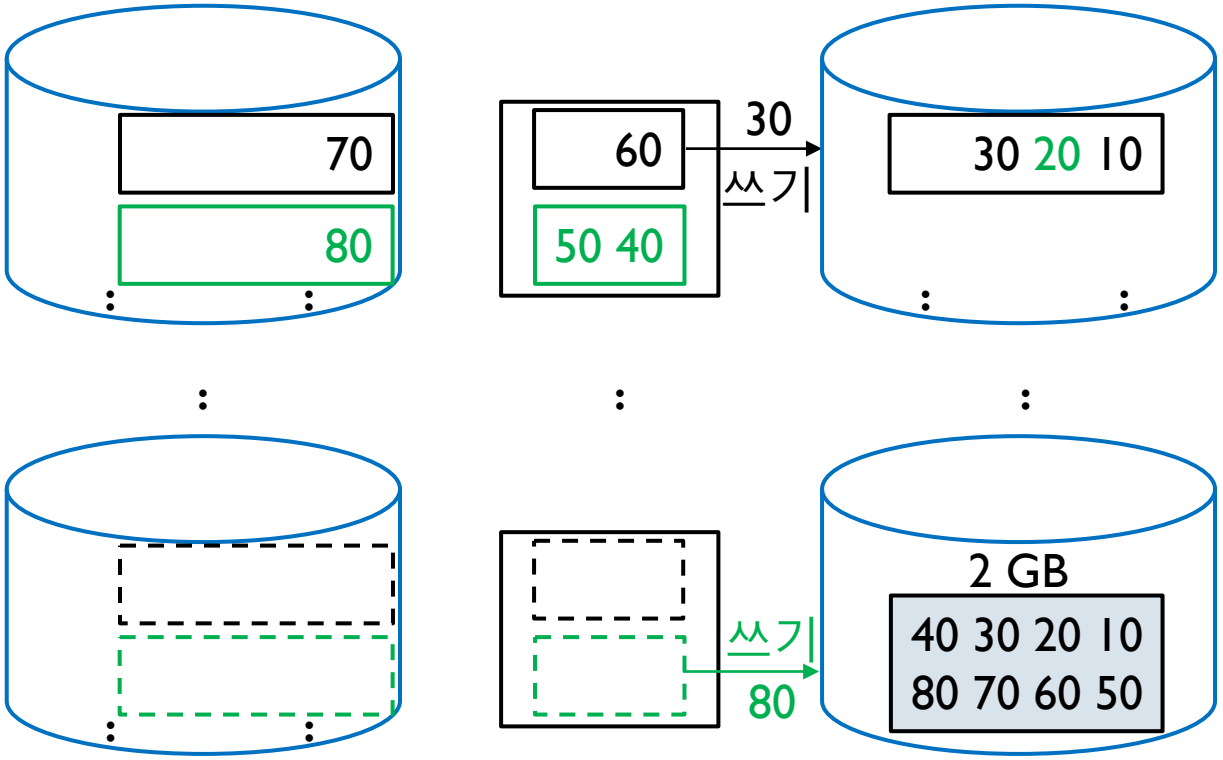
- 1GB 블록 2개가 2GB 블록 1개로 합병되는 과정



외부정렬



외부정렬



외부정렬

- 나머지 98개의 블록에 대해서 위와 같이 49회를 반복하면, 2GB 블록이 총 50개 만들어진다.
- 그 다음에는 2GB 블록 2개씩 짝을 지워 합병시키는 과정을 총 25회 수행하면, 4GB 블록 25개가 만들어진다.
- 이러한 방식으로 계속 합병을 진행하면, 블록 크기가 2배로 커지고 블록의 수는 1/2로 줄어들게 되어 결국에는 100GB 블록 1개만 남는다.

외부정렬

- 외부정렬 알고리즘은 보조 기억 장치에서의 읽고 쓰기를 최소화하는 것이 매우 중요하다.
- 왜냐하면 보조 기억 장치의 접근 시간(access time)이 주기억 장치의 접근 시간보다 매우 오래 걸리기 때문이다.

- 다음은 외부정렬 알고리즘이다.
  - 주기억 장치의 용량을 M이라고 가정한다.
  - 외부정렬 알고리즘은 입력이 저장된 보조 기억 장치 외에 별도의 보조 기억 장치를 사용한다.
  - 알고리즘에서 보조 기억 장치는 'HDD'로 표기한다.

외부정렬 알고리즘

ExternalSort	
입력: 입력 데이터가 저장된 입력 HDD 출력: 정렬된 데이터가 저장된 출력 HDD	
1	입력 HDD에 저장된 입력을 크기가 M만큼씩 주기억 장치에 읽어 들인 후 내부정렬 알고리즘으로 정렬하여 별도의 HDD에 저장한다. 다음 단계에서 별도의 HDD는 입력 HDD로 사용되고, 입력 HDD는 출력 HDD로 사용된다.
2	while (입력 HDD에 저장된 블록 수 > 1) {
3	입력 HDD에 저장된 블록을 2개씩 선택하여, 각각의 블록으로부터 데이터를 부분적으로 주기억 장치에 읽어 들여서 합병을 수행한다. 이때 합병된 결과는 출력 HDD에 저장한다. 단, 입력 HDD에 저장된 블록 수가 홀수일 때에는 마지막 블록은 그대로 출력 HDD에 저장한다.
4	입력과 출력 HDD의 역할을 바꾼다. }
5	return 출력 HDD

외부정렬 알고리즘

Line 1	입력 HDD에 저장된 입력을 크기가 M 만큼씩 주기억 장치로 읽어 들여서 내부정렬 알고리즘으로 정렬한 후에 별도의 HDD에 저장한다. 입력 크기가 N이라면, N/M개의 블록이 만들어진다. 단, 편의상 N이 M의 배수라고 가정하자. 다음 단계를 위해 별도의 HDD는 입력 HDD로 사용되고, 입력 HDD는 출력 HDD로 사용된다.
Line 2	while-루프에서는 입력 HDD에 저장된 블록 수가 2개 이상이면, line 3~4를 수행한다.

외부정렬 알고리즘

Line 3	입력 HDD에 저장된 블록을 2개씩 선택하여, 각 블록의 데이터를 부분적으로 주기억 장치에 읽어 들여서 합병을 수행한다. 처음에 line 3이 수행되면, 출력 HDD에 있는 각각의 블록 크기는 2M이 된다. 그 다음에 line 3이 수행되면 각각의 블록 크기는 4M이 된다. 즉, line 3이 수행되기 직전의 블록 크기의 2배가 되어 출력 HDD에 저장된다.
Line 4	입력과 출력으로 사용된 HDD의 역할을 서로 바꾼다. 즉, 출력 HDD가 입력 HDD가 되고, 입력 HDD가 출력 HDD로 사용된다.

외부정렬 알고리즘 수행과정

- 128GB 입력과 1GB의 주기억 장치 대한 ExternalSort의 수행 과정을 블록의 크기와 개수로 살펴보자.
  - Line 1: 1GB의 정렬된 블록 128개가 별도의 HDD에 저장된다.
  - Line 3: 2GB의 정렬된 블록 64개가 출력 HDD에 만들어진다.
  - Line 3: 4GB의 정렬된 블록 32개가 출력 HDD에 만들어진다.
  - Line 3: 8GB의 정렬된 블록 16개가 출력 HDD에 만들어진다.

  …
  - Line 3: 64GB의 정렬된 블록 2개가 출력 HDD에 만들어진다.
  - Line 3: 128GB의 정렬된 블록 1개가 출력 HDD에 만들어지고, while-루프의 조건이 '거짓'이 되어 line 5에서 출력 HDD를 리턴한다.

시간복잡도

- 외부정렬은 전체 데이터를 몇 번 처리(읽고 쓰기)하는가를 가지고 시간복잡도를 측정한다.
- 전체 데이터를 읽고 쓰는 것을 패스(pass)라고 한다.
- ExternalSort 알고리즘은 line 3에서 전체 데이터를 입력 HDD에서 읽고 합병하여 출력 HDD에 저장한다. 즉, 1 패스가 수행된다.
- 그러므로 while-루프가 수행된 횟수가 ExternalSort 알고리즘의 시간복잡도가 된다.

시간복잡도

- 입력 크기가 N이고, 메모리 크기가 M이라고 하면, line 3이 수행될 때마다 블록 크기가 2M, 4M, …, 2^kM으로 2배씩 증가한다.
- 만일 마지막에 만들어진 1개의 블록 크기가 2^kM이라고 하면 이 블록은 입력 전체가 합병된 결과를 가지고 있으므로, $2^k M = N$이다.
- 여기서 k는 while-루프가 수행된 횟수가 된다. 따라서 $2^k = N/M$, $k = \log_2(N/M)$이다.

  ∴ 외부정렬의 시간복잡도는 O(log(N/M))이다.