Chapter 6-1

Sorting Algorithms

Sorting Algorithm

- 내부 정렬 (Internal sort)
  - 내부 정렬은 입력의 크기가 주기억 장치(main memory)의 공간보다 크지 않은 경우에 수행되는 정렬
  - 버블 정렬, 선택 정렬, 삽입 정렬, 합병 정렬, 퀵 정렬, 힙 정렬, 쉘 정렬, 기수 정렬, 이중 피봇 퀵 정렬, Time sort
- 외부 정렬 (External sort)
  - 입력의 크기가 주기억 장치 공간보다 큰 경우에 보조 기억 장치에 있는 입력을 여러 번에 나누어 주기억 장치에 읽어 들인 후, 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복
  - 다방향 합병 (p-way Merge), 다단계 합병 (Polyphase Merge)

내부 정렬 알고리즘

Comparison sorts

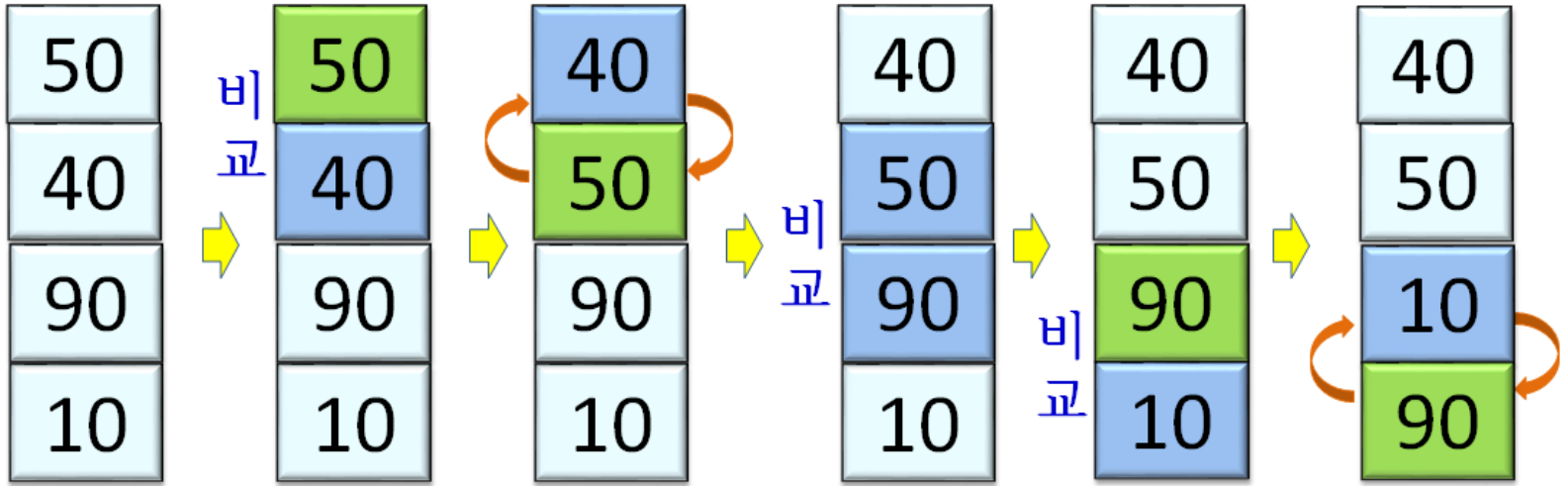
Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes
In-place merge sort	—	—	$n \log^2 n$	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Block sort	n	$n \log n$	$n \log n$	1	Yes
Quadsort	n	$n \log n$	$n \log n$	n	Yes
Timsort	n	$n \log n$	$n \log n$	n	Yes
Selection sort	n^2	n^2	n^2	1	No
Cubesort	n	$n \log n$	$n \log n$	n	Yes
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Tree sort	$n \log n$	$n \log n$	$n \log n$(balanced)	n	Yes
Cycle sort	n^2	n^2	n^2	1	No
Library sort	n	$n \log n$	n^2	n	Yes
Patience sorting	n	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[12]	No
Cocktail shaker sort	n	n^2	n^2	1	Yes
Comb sort	$n \log n$	n^2	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
UnShuffle Sort[13]	n	kn	kn	n	No
Franceschini's method[14]	—	$n \log n$	$n \log n$	1	Yes
Odd–even sort	n	n^2	n^2	1	Yes
Zip sort	$n \log n$	$n \log n$	$n \log n$	1	Yes

Bubble Sort

- 버블 정렬 (Bubble Sort)
  - 이웃하는 숫자를 비교하여 작은 수를 앞쪽으로 이동시키는 과정을 반복하여 정렬
  - 작은 수는 배열의 앞부분으로 이동하는데, 배열을 좌우가 아니라 상하로 그려보면 정렬하는 과정에서 작은 수가 마치 거품처럼 위로 올라가는 것을 연상케 함

Bubble Sort

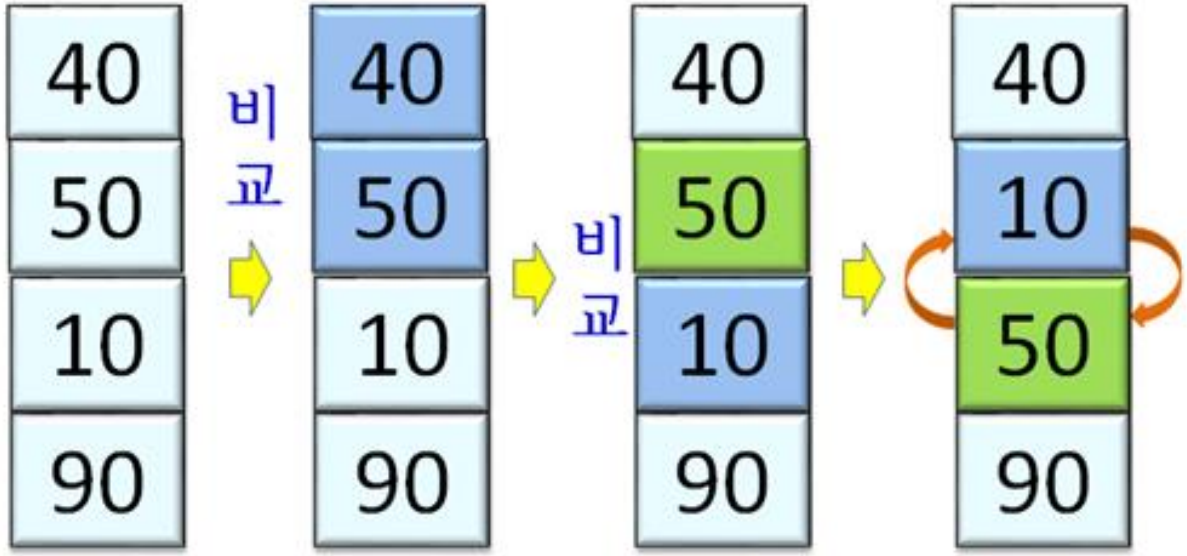
- Bubble Sort



- 패스 (pass): 입력은 전반적으로 1번 처리하는 것
- 1번째 pass 결과를 살펴보면, 작은 수는 버블처럼 위로 1칸씩 올라감
- 가장 큰 수인 90은 맨 밑에, 즉, 배열의 가장 마지막에 위치

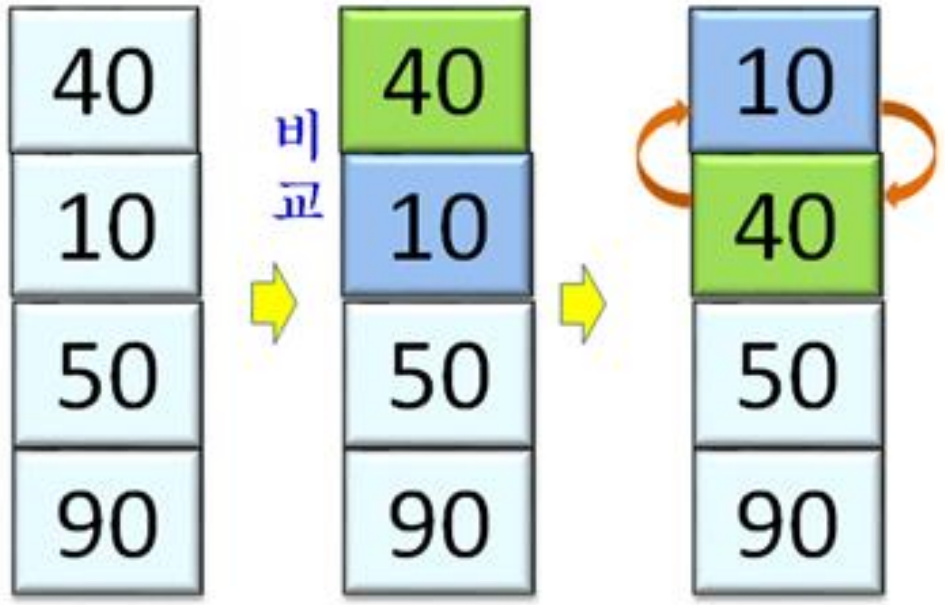
Bubble Sort

- 2번째 패스
  - 이웃하는 원소 간의 비교를 통해 40-50은 그대로 그 자리에 있고, 50과 10이 서로의 자리를 바꿈
  - 두 번째로 큰 수인 50이 가장 큰 수인 90의 위에 위치



Bubble Sort

- 3번째 패스
  - 이웃하는 원소 간의 비교를 통해 40과 10이 서로 자리를 바꿈
  - 세 번째로 큰 수인 40이 두 번째로 큰 수인 50의 위에 위치
  - n개의 원소가 있으면 (n-1)번의 패스가 수행



Bubble Sort

- 알고리즘

  BubbleSort

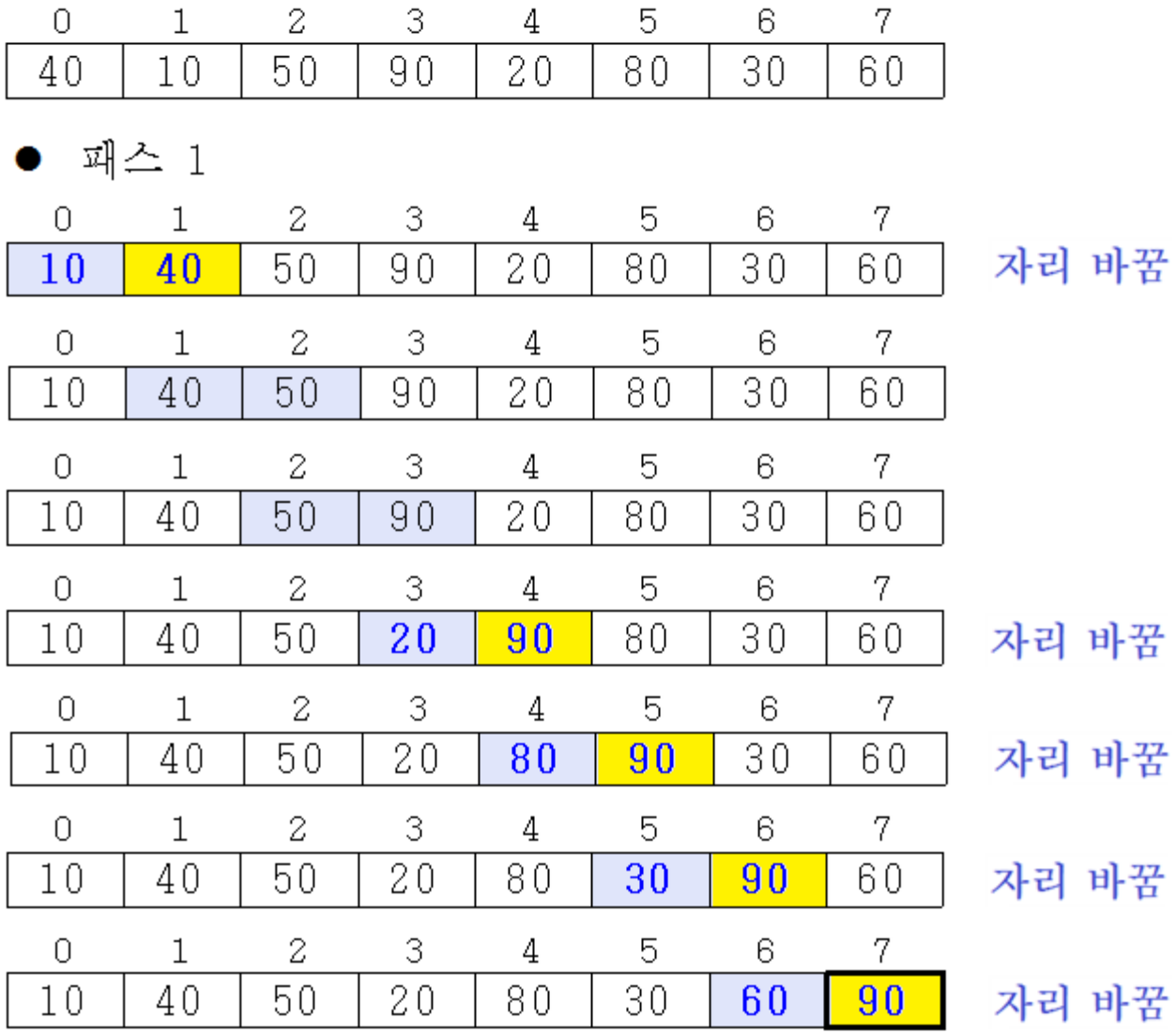
  입력: 크기가 n인 배열 A

  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]       // 자리 바꿈
5.  return A
```

Bubble Sort

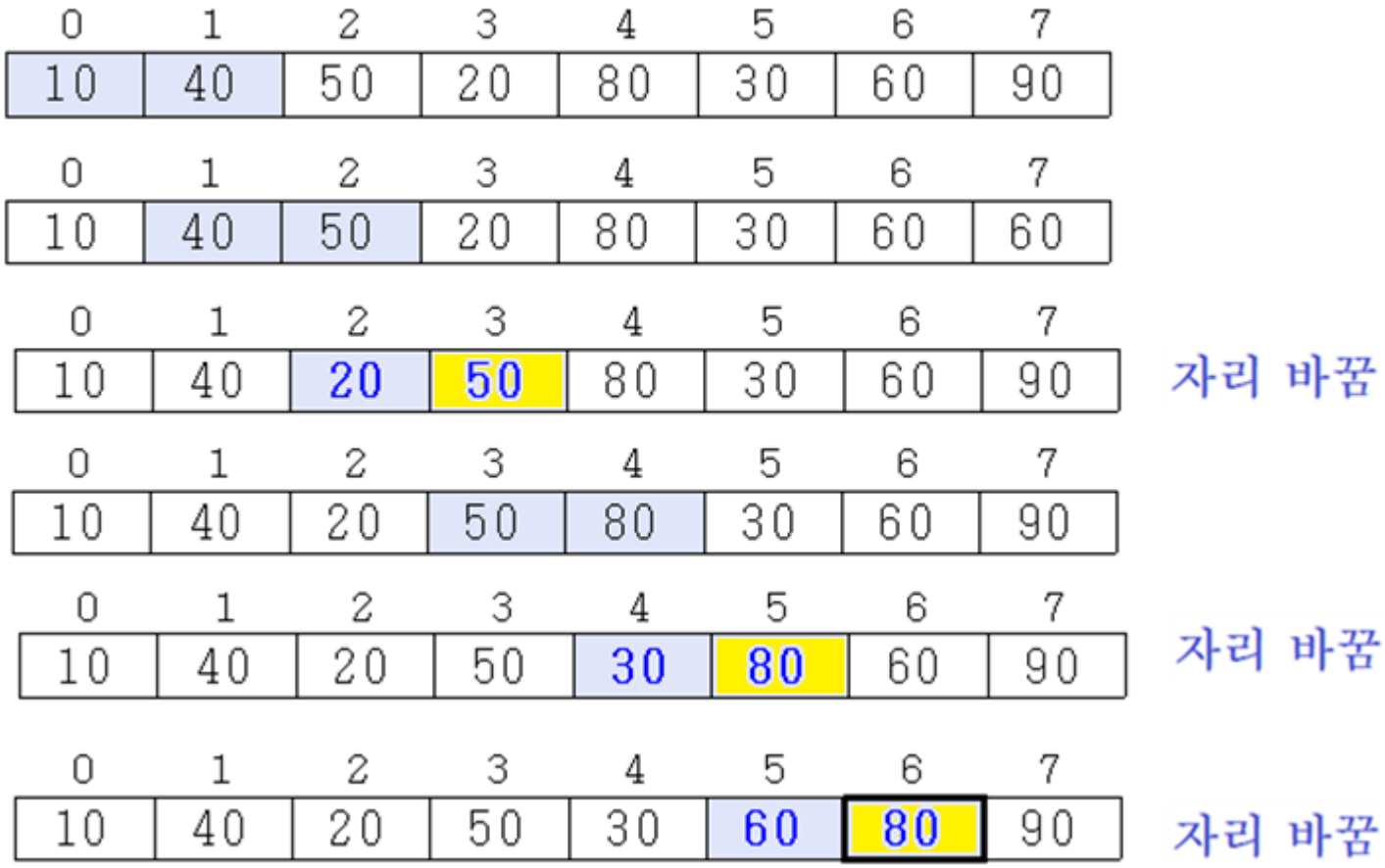
- 버블 정렬의 수행 과정



Bubble Sort

- 버블 정렬의 수행 과정

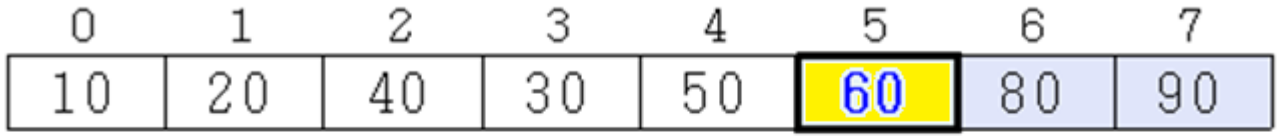
● 패스 2



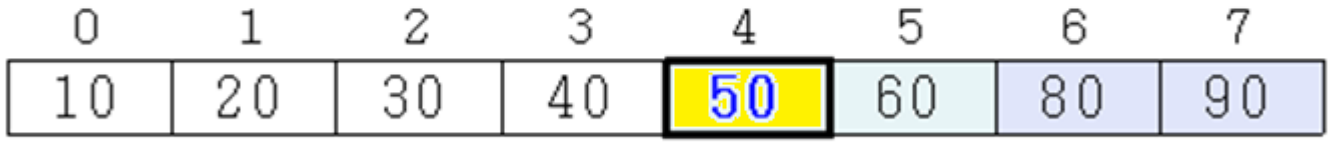
Bubble Sort

- 버블 정렬의 수행 과정

● 패스 3 결과



● 패스 4 결과



● 패스 5~7 결과는 패스 4 결과와 동일

Bubble Sort

- Time Complexity
  - 버블 정렬은 for-loop 속에서 for-loop 수행
    - Pass=1이면 (n-1)번 비교
    - Pass=2이면 (n-2)번 비교

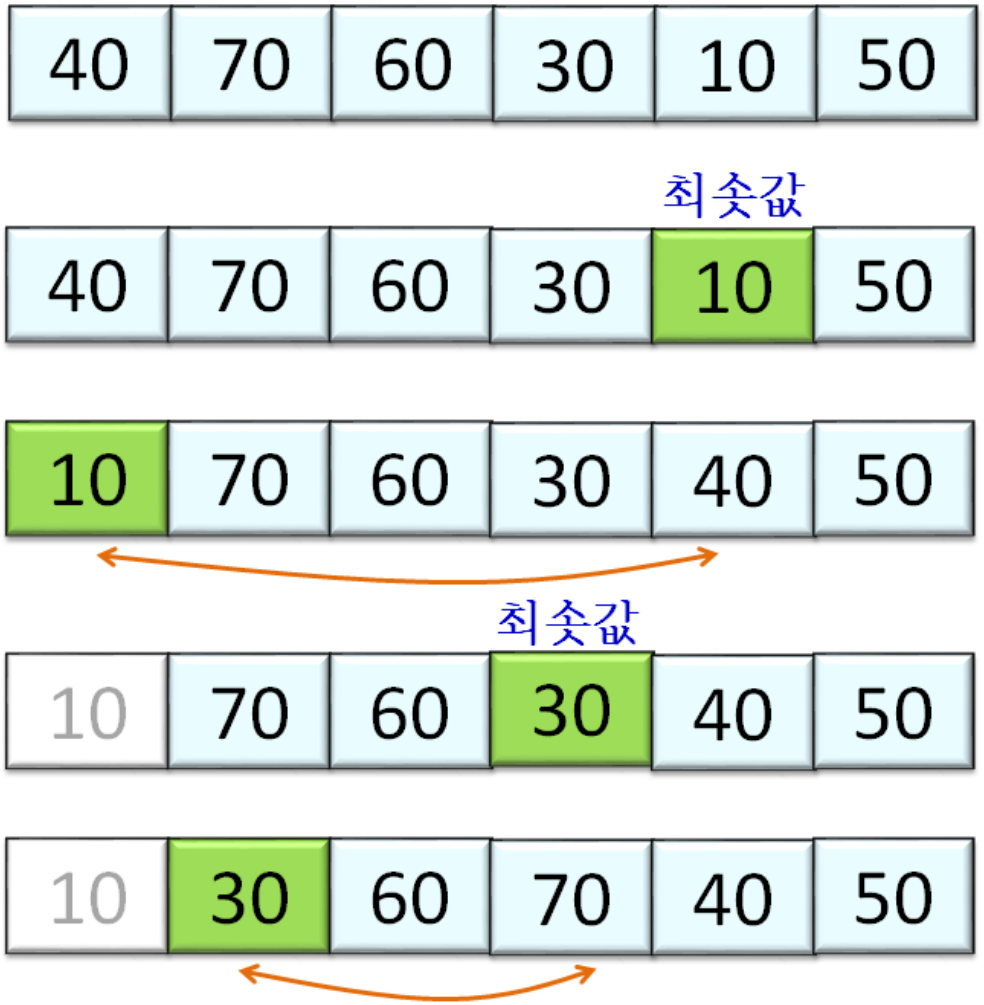
      …
    - Pass=n-1이면 1번 비교
    - 총 비교 횟수: (n-1) + (n-2) + … + 1 = n(n-1)/2
    - 안쪽 for-loop의 if-조건이 True일 때의 자리바꿈은 O(1) 시간
  - n(n-1)/2 X O(1) = $O(n^2)$

Selection Sort

- 선택 정렬 (Selection Sort)
  - 입력 배열 전체에서 최솟값을 선택하여 배열의 0번 원소와 자리를 바꾸고, 다음엔 0번 원소를 제외한 나머지 원소에서 최솟값을 선택하여, 배열의 1번 원소와 자리를 바꿈
  - 이런 방식으로 마지막에 2개의 원소 중 작은 것을 선택하여 자리를 바꿈

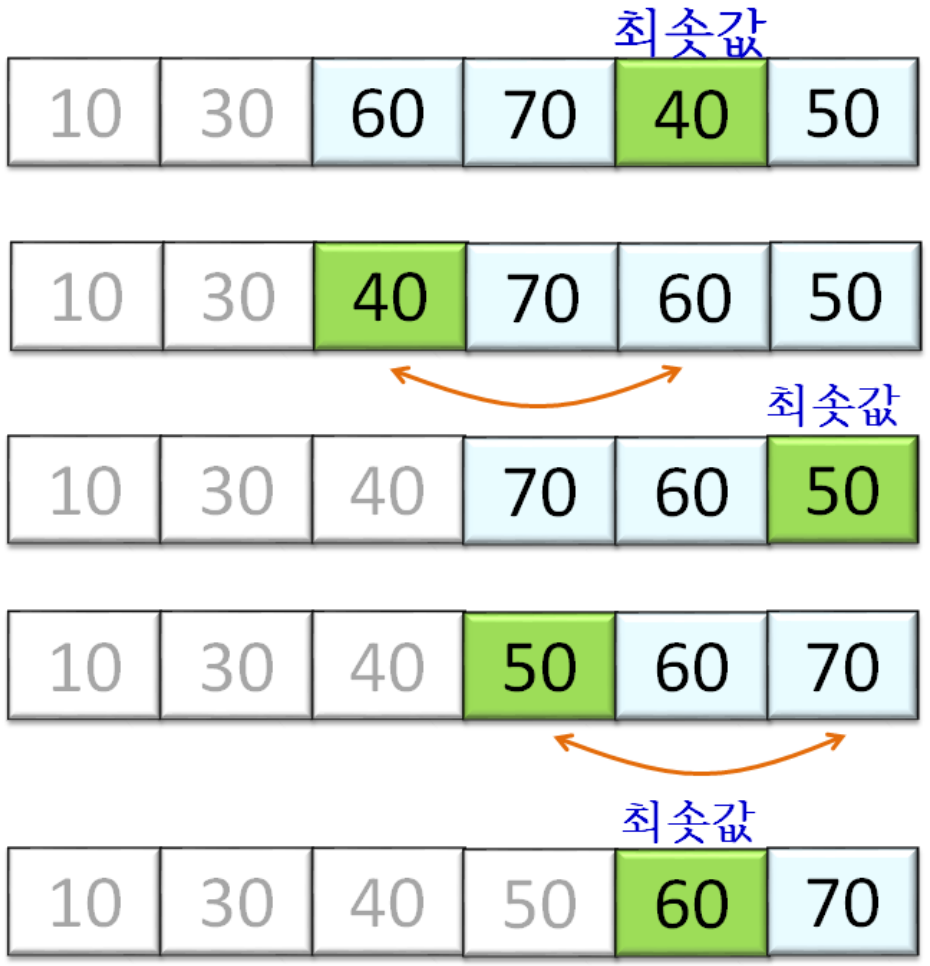
Selection Sort

- 선택 정렬 수행 과정



Selection Sort

- 선택 정렬 수행 과정



Selection Sort

- 알고리즘

SelectionSort

입력: 크기가 n인 배열 A

출력: 정렬된 배열 A

```
1.  for i = 0 to n-1
2.      min = i
3.      for j = i+1 to n-1    // A[i]~A[n-1]
4.          if A[j] < A[min]
5.              min = j
6.      A[i] ↔ A[min]         // min이 최솟값이 있는 원소의 인덱스
7.  return A
```

Selection Sort

- 선택 정렬의 수행 과정

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

● i=0 A[0]~A[7]에서 최소값은 10, min=1

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=1 A[1]~A[7]에서 최소값은 20, min=4

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

Selection Sort

- 선택 정렬의 수행 과정

● i=2 A[2]~A[7]에서 최소값은 30 , min=6

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	40	80	50	60

자리 바꿈

⋮

● i = 6 A[6]~A[7]에서 최소값은 80 , min=7

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

Selection Sort

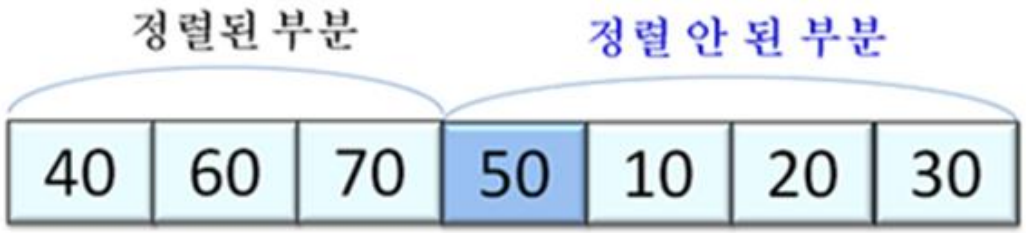
- Time Complexity
  - 외부 for-loop는 (n-1)번 수행
    - i=0일 때, 내부 for-loop는 (n-1)번 수행
    - i=1일 때, 내부 for-loop는 (n-2)번 수행
    - ⋮
    - 마지막으로 1번 수행
  - 내부의 for-loop가 수행되는 총 횟수
    - (n-1)+(n-2)+(n-3)+…+2+1 = n(n-1)/2
  - Loop 내부의 if-조건이 True일 때의 자리바꿈은 O(1)
  - n(n-1)/2 X O(1) = $O(n^2)$

Selection Sort

- 선택 정렬의 특징
  - 입력이 거의 정렬되어 있든지, 역으로 정렬되어 있든지, 랜덤하게 되어 있든지 항상 일정한 시간 복잡도를 나타냄
  - 입력에 민감하지 않은 (input insensitive) 알고리즘
  - 원소 간의 자리바꿈 횟수가 최소인 정렬

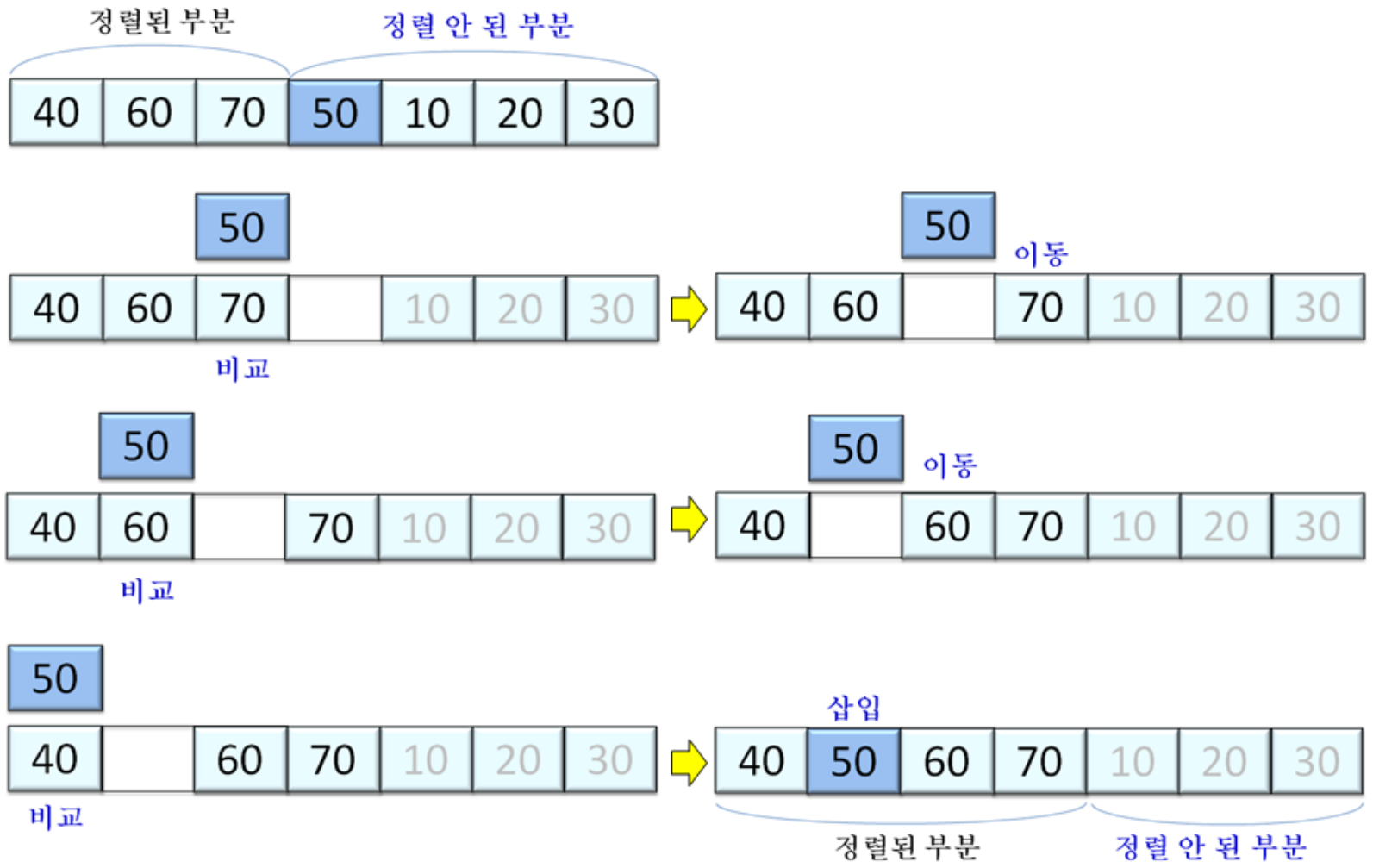
Insertion Sort

- 삽입 정렬 (Insertion Sort)
  - 배열을 정렬된 부분(앞부분)과 정렬 안된 부분(뒷부분)으로 나누고, 정렬 안된 부분의 가장 왼쪽 원소를 정렬된 부분의 적절한 위치에 삽입하여 정렬되도록 하는 과정을 반복



Insertion Sort

- 삽입 정렬 (Insertion Sort)



Insertion Sort

- 삽입 정렬 (Insertion Sort)
  - 정렬 안된 부분의 숫자 하나가 정렬된 부분에 삽입됨으로써, 정렬된 부분의 원소의 개수가 1개 늘어나고, 동시에 정렬이 안된 부분의 원소의 개수는 1개 줄어듦
  - 이를 반복하여 수행하면, 마지막엔 정렬이 안된 부분엔 아무 원소도 남지 않고, 입력이 정렬됨
  - 정렬은 배열의 첫 번째 원소만이 정렬된 부분에 있는 상태에서 정렬을 시작

Insertion Sort

- 알고리즘

InsertionSort

입력: 크기가 n인 배열 A

출력: 정렬된 배열 A

```
1.  for i = 1 to n-1
2.      CurrentElement = A[i]      // 정렬안된 부분의 가장 왼쪽 원소
3.      j = i - 1
4.      while (j >= 0) and (A[j] > CurrentElement)
5.          A[j+1] = A[j]     // 자리이동
6.          j = j - 1
7.      A[j+1] <- CurrentElement
8.  return A
```

Insertion Sort

- 삽입 정렬의 수행 과정

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

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

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

한 칸 앞으로 이동

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

A[0]에 CurrentElement=10 저장

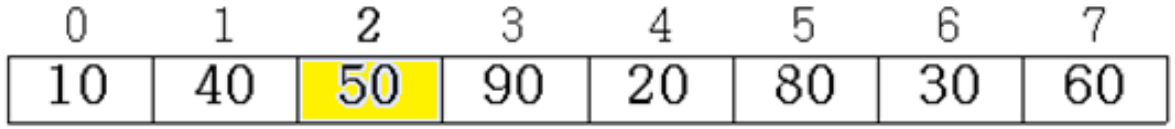
0	1	2	3	4	5	6	7
10	40	50	90	20	80	30	60

Insertion Sort

- 삽입 정렬의 수행 과정

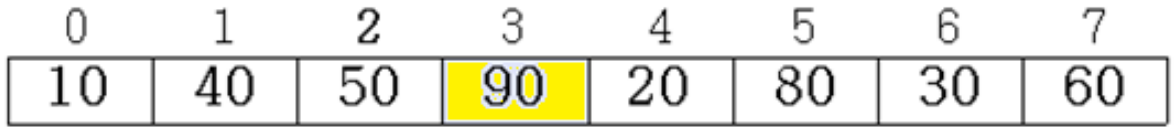
  - i=2, CurrentElement=A[2]=50, j=i-1=1

    자리이동 없이 50이 그 자리에



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

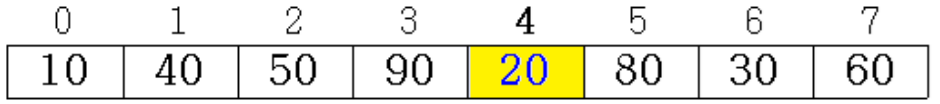
    자리이동 없이 90이 그 자리에



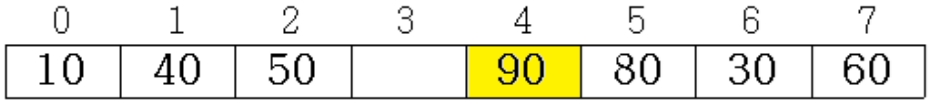
Insertion Sort

- 삽입 정렬의 수행 과정

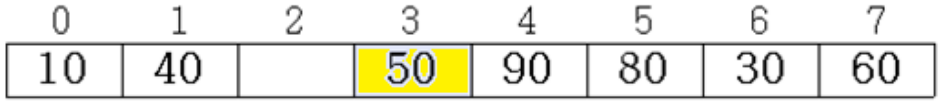
● i=4, CurrentElement=A[4]=20, j=i-1=3



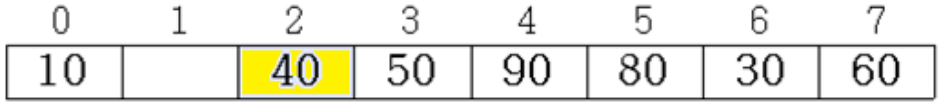
한 칸 앞으로 이동



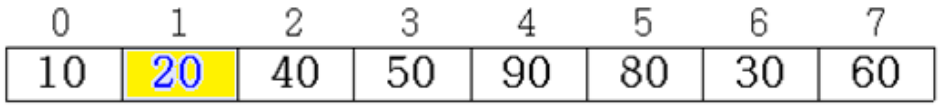
한 칸 앞으로 이동



한 칸 앞으로 이동

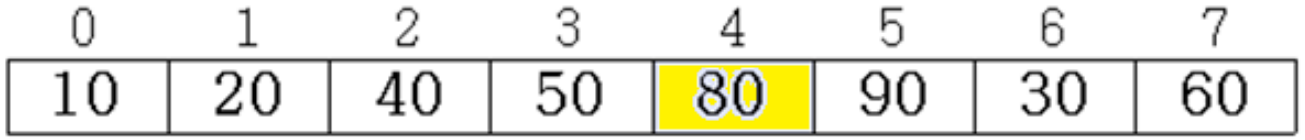


A[1]에 CurrentElement=20 저장

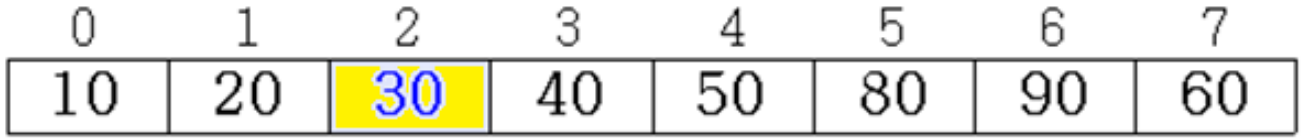


Insertion Sort

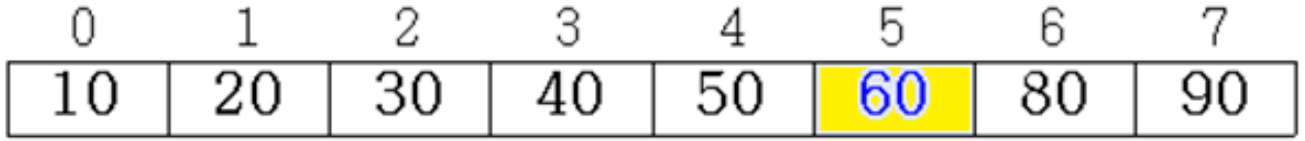
- 삽입 정렬의 수행 과정
  - i=5, CurrentElement=A[5]=80일 때의 결과



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



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



Insertion Sort

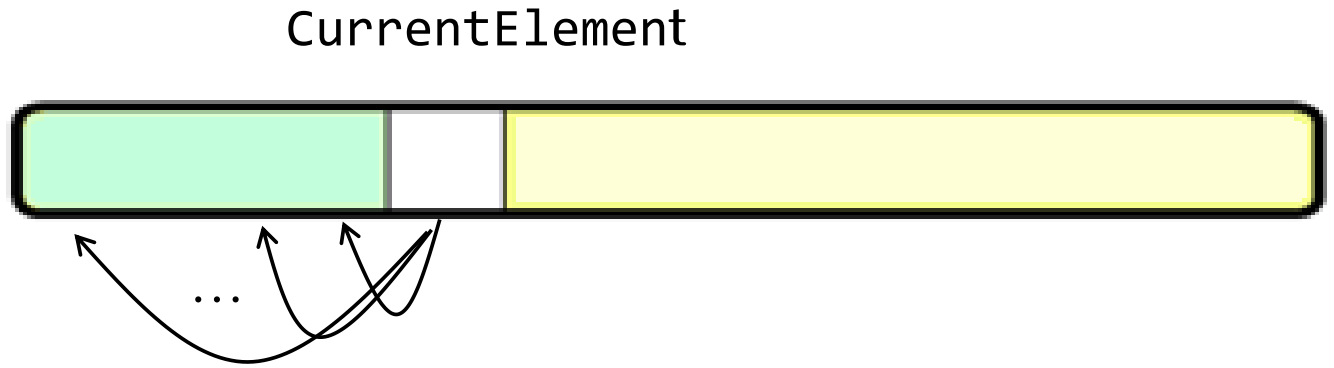
- Time Complexity for Worst Case
  - for-loop가 (n-1)회 수행
    - i=1일 때, while-loop는 1회 수행
    - i=2일 때, 최대 2회 수행
    - ⋮
    - 마지막으로 최대 (n-1)회 수행
  - Loop 내부의 line 5-6이 수행되는 총 횟수
    - 1+2+3+…+(n-2)+(n-1) = n(n-1)/2
  - Loop 내부 시간은 O(1) 시간
  - n(n-1)/2 X O(1) = $O(n^2)$

Insertion Sort

- Time Complexity for Best Case
  - 입력이 이미 정렬되어 있으면, 항상 각각 CurrentElement가 자신의 왼쪽 원소와 비교 후 자리이동 없이 원래 자리에 위치하고, while-loop의 조건이 항상 False이므로 원소의 이동도 전혀 없음
    - 따라서 (n-1)번의 비교만에 정렬을 마치게 됨
    - 이때가 삽입 정렬의 best case이고 time complexity는 O(n)

Insertion Sort

- Time Complexity for Average Case
  - CurrentElement가 자신의 자리 포함 앞부분의 각 원소에 자리잡을 확률이 같다고 가정하여 분석하면
    - $O(n^2/4) = O(n^2)$



현재 원소가 각각 앞부분 각각 원소에 자리 잡을 확률이 같다고 가정(균등 분포)

Insertion Sort

- 삽입 정렬의 특성
  - 삽입 정렬은 거의 정렬된 입력에 대해서 다른 정렬 알고리즘보다 빠름
    - 예를 들면, 입력이 앞부분은 정렬되어 있고 뒷부분(전체 입력의 20% 이하)에 새 데이터가 있는 경우
  - 입력의 크기가 작을 때 매우 좋은 성능을 보임
  - Quick sort, merge sort에서 입력 크기가 작아지면 순환 호출을 중단하고 insertion sort를 사용

Shell Sort

- 쉘 정렬 (Shell Sort) Motivation
  - Bubble sort나 insertion sort가 수행되는 과정
    - '기껏해야' 이웃하는 원소의 자리바꿈
  - Bubble sort의 수행 과정
    - 작은(가벼운) 숫자가 배열의 앞부분으로 매우 느리게 이동
  - Insertion sort에서 마지막 원소가 가장 작은 숫자라면
    - 그 숫자가 배열의 맨 앞으로 이동해야 하므로, 모든 다른 숫자들이 1칸씩 오른쪽으로 이동해야 함

Shell Sort

- 쉘 정렬 (Shell Sort) 아이디어
  - Insertion sort를 이용하여 배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게 ' 이동시키고
  - 동시에 앞부분의 큰 숫자는 뒷부분으로 '빠르게 ' 이동시킴

Shell Sort

- 쉘 정렬 (Shell Sort) 아이디어
  - 간격(gap)을 이용

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

    - 간격(gap)이 5인 숫자 그룹
      - 그룹 1 [30, 80, 50]
      - 그룹 2 [60, 40, 30]
      - 그룹 3 [90, 20, 40]
      - 그룹 4 [10, 10, 90]
      - 그룹 5 [40, 60, 80]

h=5

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

Shell Sort

- 쉘 정렬 (Shell Sort) 아이디어
  - 간격(gap)을 이용

h=5

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

  - 각 그룹 별로 insertion sort를 수행한 결과를 1줄에 나열해 보면 다음과 같음

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

그룹별 정렬 후

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

Shell Sort

- 쉘 정렬 (Shell Sort) 아이디어
  - 간격이 5인 그룹 별로 정렬한 결과
    - 80과 90같은 큰 숫자가 뒷부분으로 이동
    - 20과 30같은 작은 숫자가 앞부분으로 이동

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

Shell Sort

- 쉘 정렬 (Shell Sort) 아이디어
  - 간격 조절
    - 다음엔 간격을 5보다 작게 하여 (예: 3) 3개의 그룹으로 나누어 각 그룹별로 insertion sort를 수행
  - 마지막에는 반드시 간격을 1로 놓고 수행
    - 왜냐하면 다른 그룹에 속해 서로 비교되지 않은 숫자가 있을 수 있기 때문
    - 모든 원소를 1개의 그룹으로 여기는 것이고, 이는 insertion sort 그 자체

Shell Sort

- 알고리즘

ShellSort

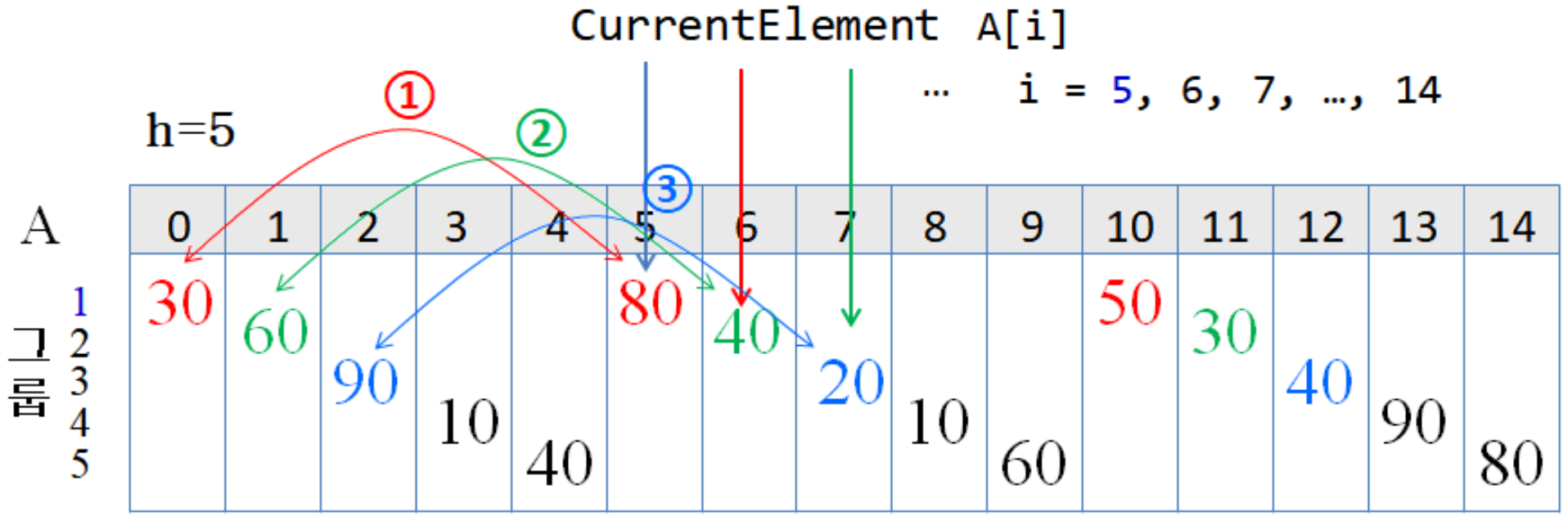
입력: 크기가 n인 배열 A

출력: 정렬된 배열 A

```
1.  for each gap h = [h0>h1>...>hk=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 A
```

Shell Sort

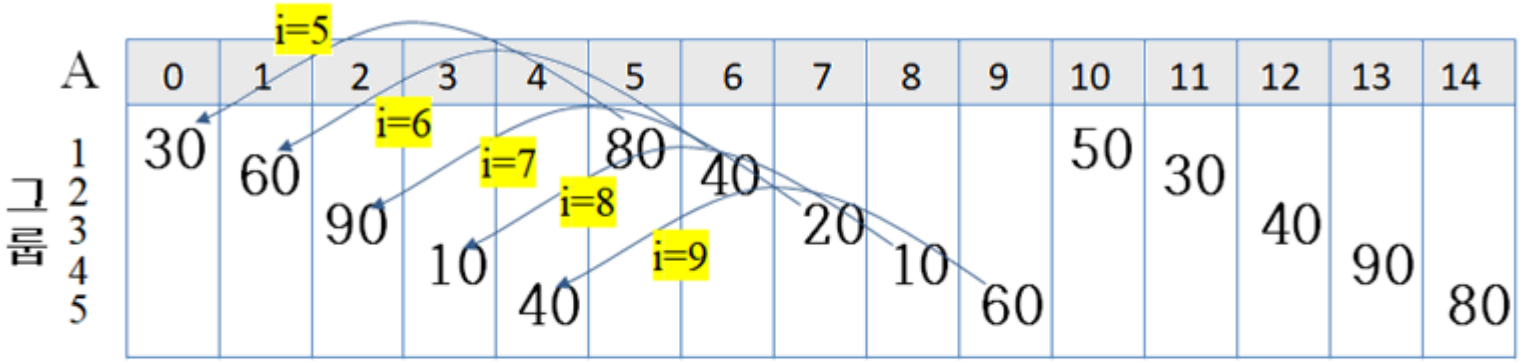
- Shell sort 알고리즘 수행과정
  - Line 2-8: for-loop에서 간격 h에 대해 그룹 별로 insertion sort가 수행되는 데, 실제 자리바꿈을 위한 원소 간의 비교는 아래와 같은 순서로 진행



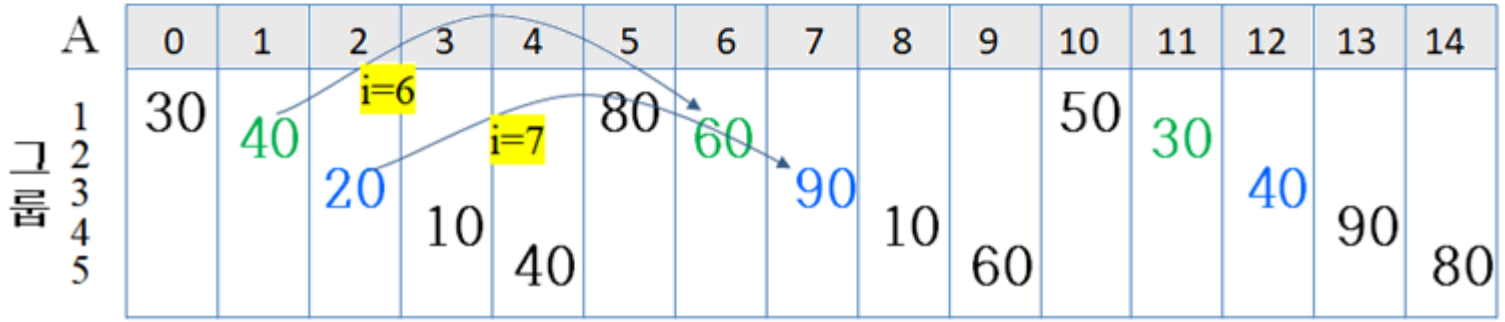
Shell Sort

- Shell sort 알고리즘 수행과정

h=5: i= 5, 6, 7, 8, 9일 때



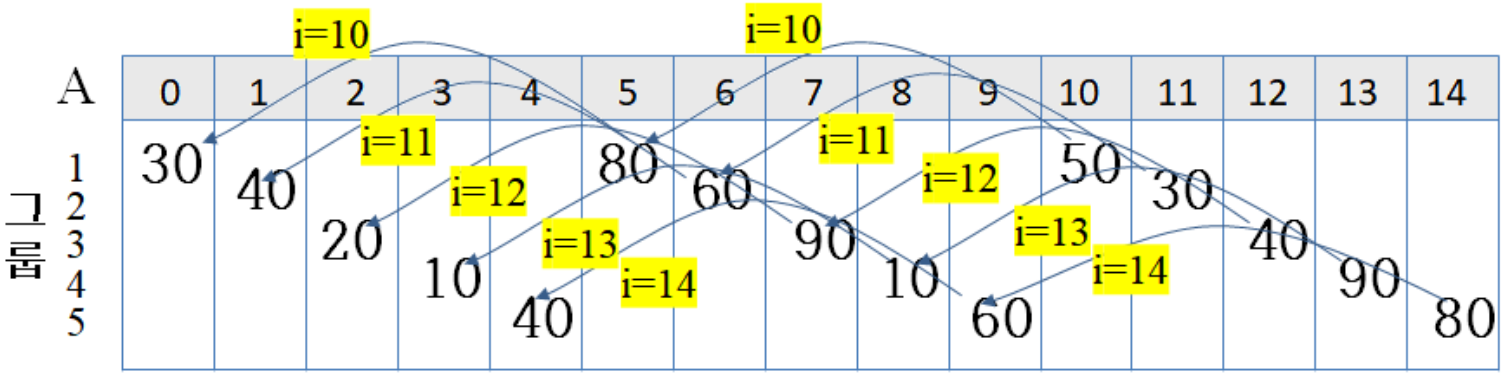
i=5, 6, 7, 8, 9일 때 이동 결과



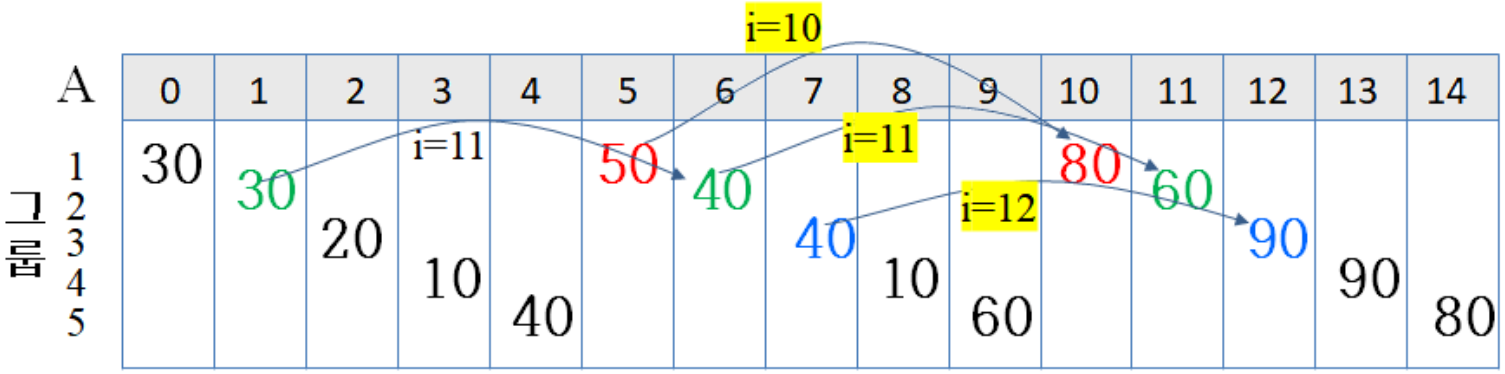
Shell Sort

- Shell sort 알고리즘 수행과정

h=5: i=10, 11, 12, 13, 14



i=10, 11, 12, 13, 14일 때 이동 결과



Shell Sort

- Shell sort 알고리즘 수행과정
  - h = 5일 때의 결과를 한 줄에 나열하면

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

  - h = 3이 되면, 배열의 원소가 3개의 그룹으로 나뉨
    - 그룹 1은 0번째, 3번째, 6번째, 9번째, 12번째 숫자
    - 그룹 2는 1번째, 4번째, 7번째, 10번째, 13번째 숫자
    - 그룹 3은 2번째, 5번째, 8번째, 11번째, 14번째 숫자

Shell Sort

- Shell sort 알고리즘 수행과정
  - h = 3일 때
    - 각 그룹 별로 insertion sort를 수행하면,

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

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

    - 각 그룹 별로 정렬한 결과를 한 줄에 나열하면

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

Shell Sort

- Shell sort 알고리즘 수행과정
  - h = 1일 때
    - Insertion sort와 동일

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

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

Shell Sort

- Time Complexity
  - Shell sort의 time complexity는 사용하는 간격에 따라 분석해야 함
  - 최악 경우의 시간 복잡도: 히바드(Hibbard)의 간격
    - 2^k-1 (2^k-1, ..., 15, 7, 5, 3, 1)을 사용하면, $O(n^{1.5})$
  - 지금까지 알려진 가장 좋은 성능을 보인 간격
    - 1, 4, 10, 23, 57, 132, 301, 701, 1750 (Marcin Ciura)
  - Shell sort의 time complexity는 아직 풀리지 않은 문제
    - 가장 좋은 간격을 알아내야 하는 것이 선행되어야 하기 때문

Shell Sort

- Shell sort의 특성
  - Shell sort는 입력 크기가 매우 크지 않은 경우에 매우 좋은 성능을 보임
  - Shell sort는 임베디드 (embedded) 시스템에서 주로 사용
    - Shell sort의 특징인 간격에 따른 그룹 별 정렬 방식이 H/W로 정렬 알고리즘을 구현하는데 매우 적합함