Chapter 6
정렬 알고리즘

차례

정렬 알고리즘

- 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 퀵정렬, Tim 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

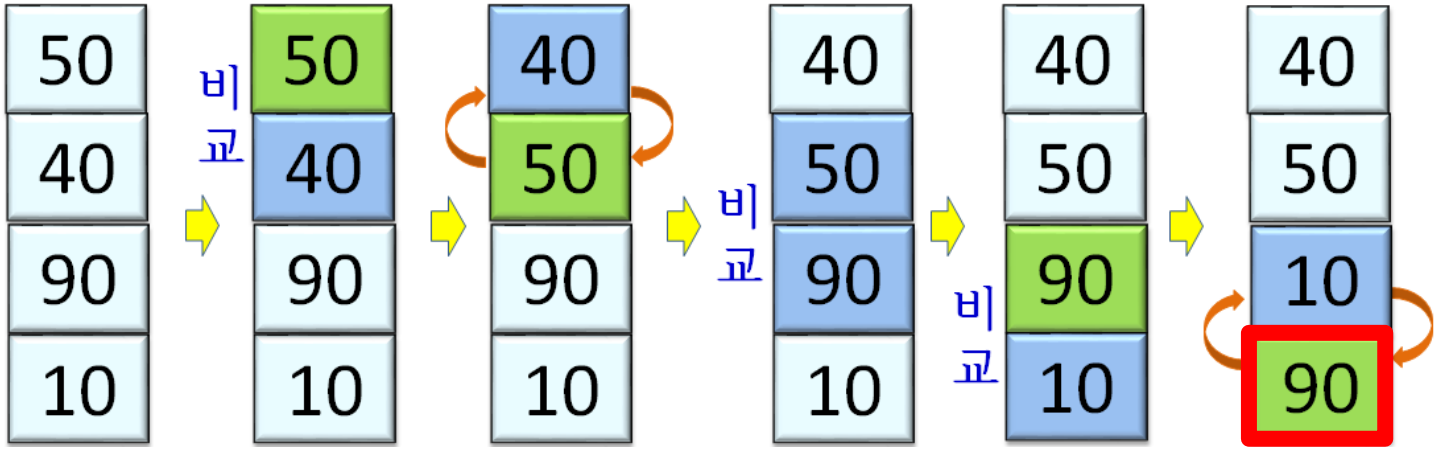
Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes
In-place merge sort	—	—	$n \log^2 n$	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	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

https://en.wikipedia.org/wiki/Sorting_algorithm

6.1 버블 정렬

- 버블 정렬 (Bubbble 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 한다.

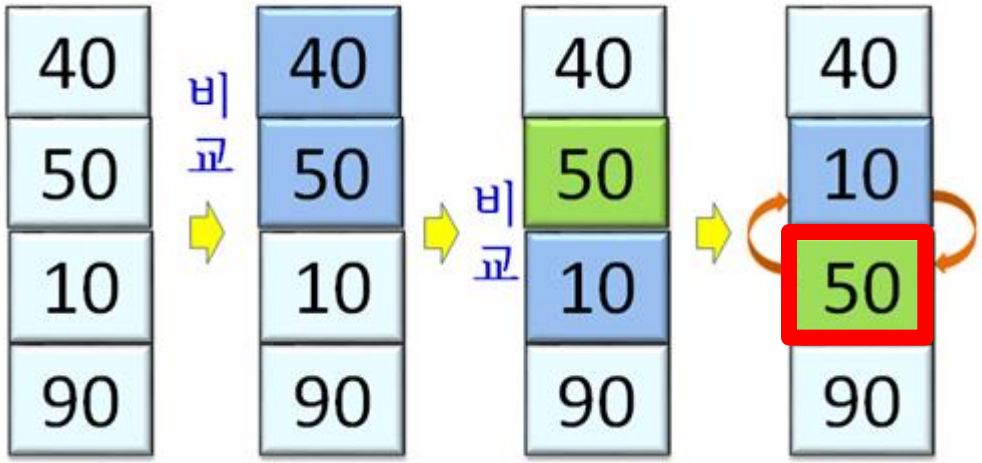
버블 정렬



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

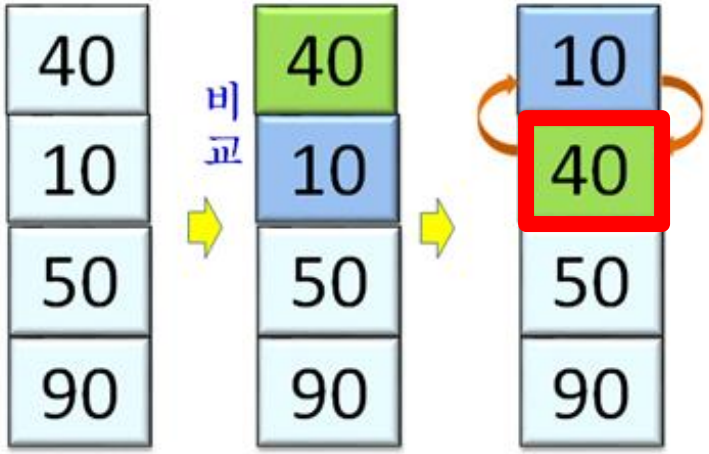
2번째 패스

- 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



3번째 패스

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



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]    // 자리바꿈
5.  return A
```

버블 정렬의 수행 과정

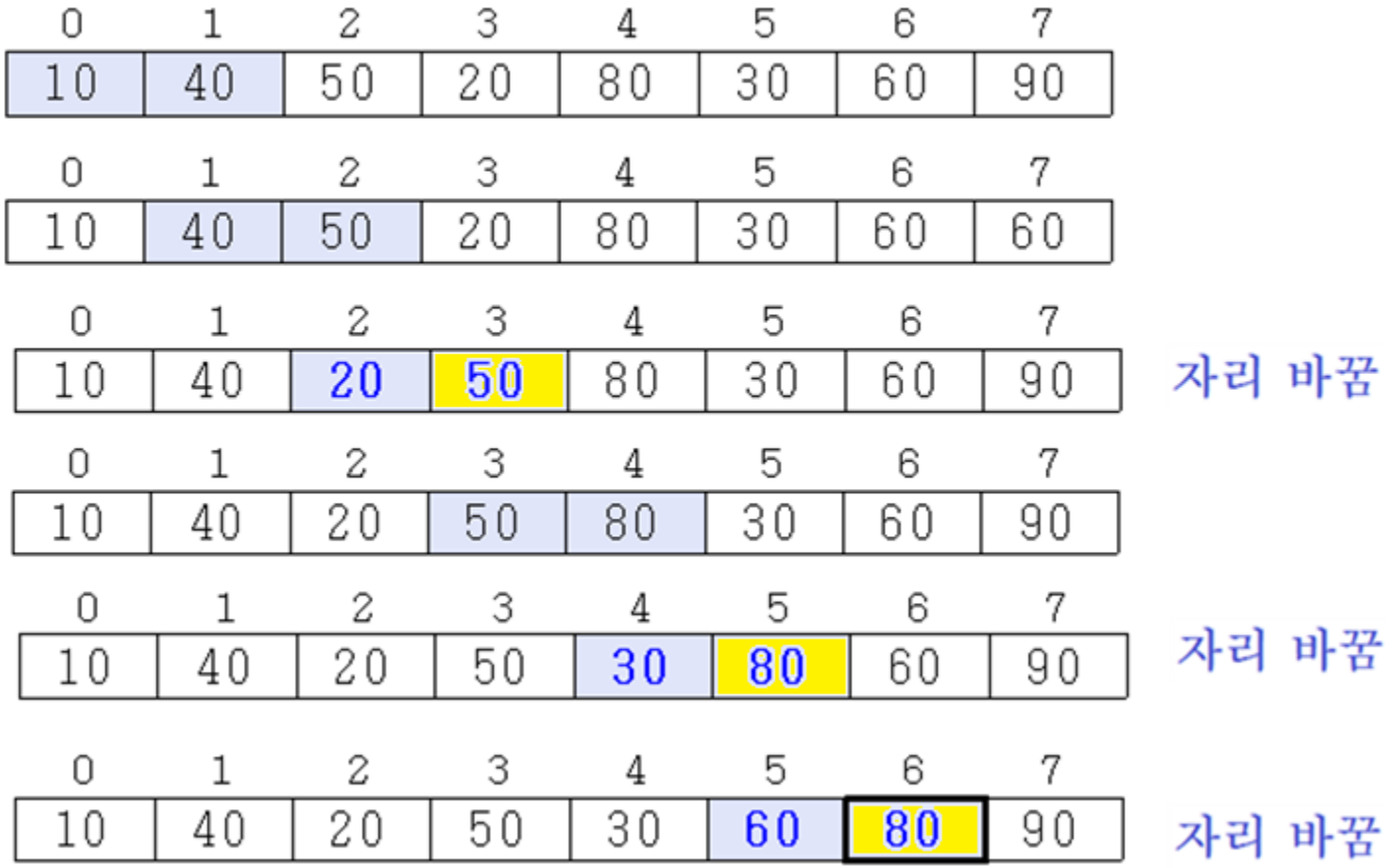
0	1	2	3	4	5	6	7
40	10	50	90	20	80	30	60

● 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	자리 바꿈
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

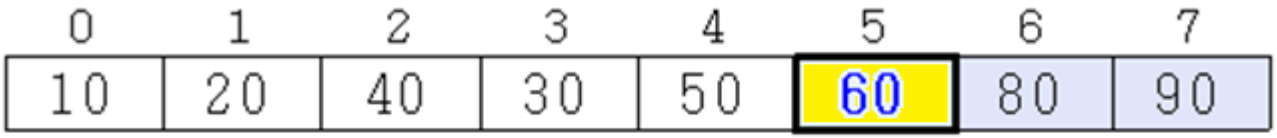
수행 과정

● 패스 2

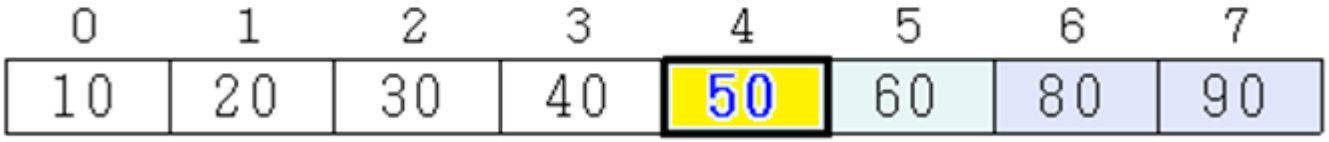

0 1 2 3 4 5 6 7
10 40 50 20 80 30 60 90
0 1 2 3 4 5 6 7
10 40 50 20 80 30 60 60
0 1 2 3 4 5 6 7
10 40 20 50 80 30 60 90
자리 바꿈
0 1 2 3 4 5 6 7
10 40 20 50 80 30 60 90
0 1 2 3 4 5 6 7
10 40 20 50 30 80 60 90
자리 바꿈
0 1 2 3 4 5 6 7
10 40 20 50 30 60 80 90
자리 바꿈

수행 과정

- 패스 3 결과



- 패스 4 결과



- 패스 5~7 결과는 패스 4 결과와 동일

시간 복잡도

- 버블 정렬은 for-루프 속에서 for-루프 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-루프의 if-조건이 True일 때의 자리바꿈은 O(1) 시간
- $n(n-1)/2 \times O(1) = O(n^2) \times O(1) = O(n^2)$

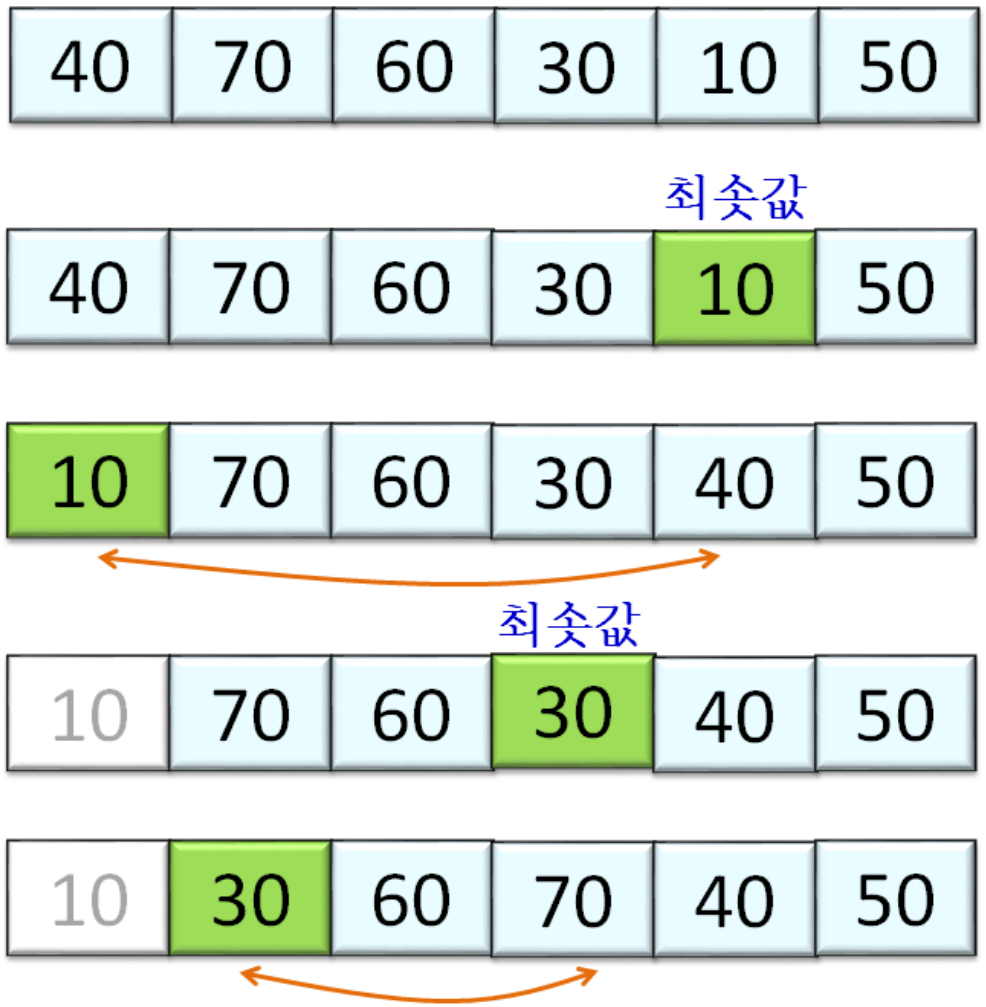
6.2 선택 정렬

선택 정렬 (Selection Sort)

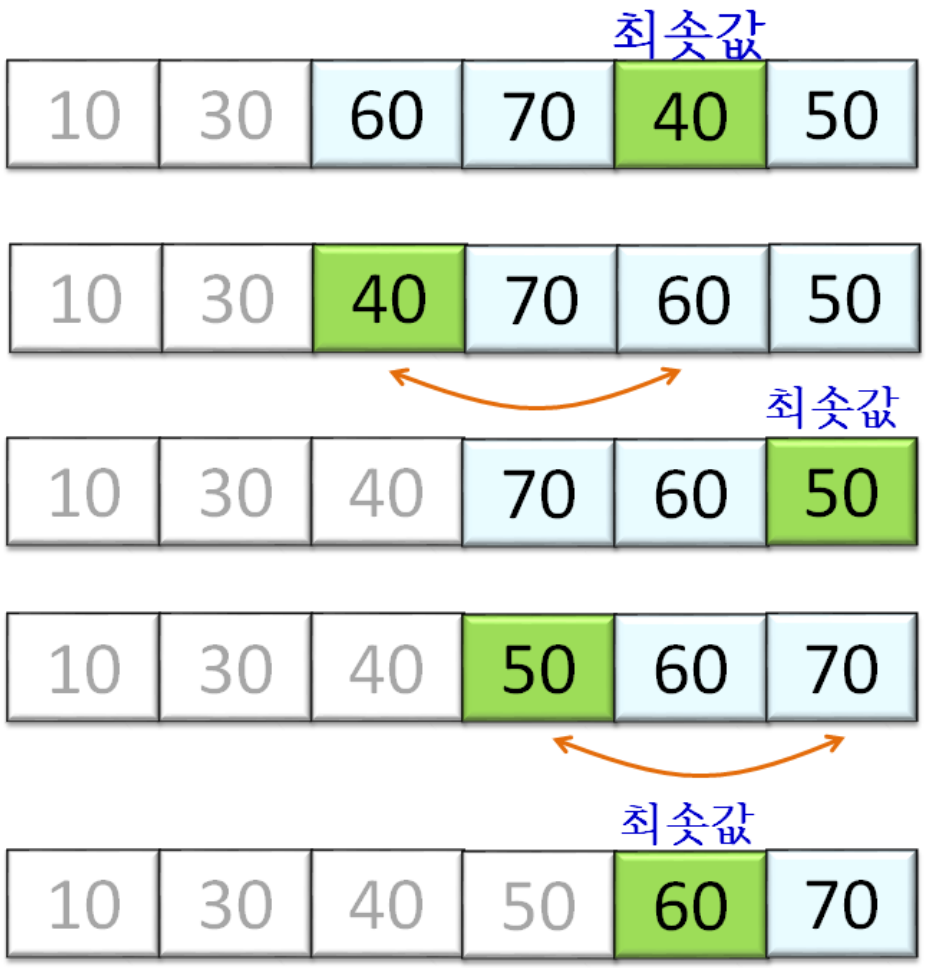
- 입력 배열 전체에서 최솟값을 선택하여 배열의 0번 원소와 자리를 바꾸고, 다음엔 0번 원소를 제외한 나머지 원소에서 최솟값을 선택하여, 배열의 1번 원소와 자리를 바꾼다.

- 이러한 방식으로 마지막에 2개의 원소 중 작은 것을 선택하여 자리를 바꾼다.

선택 정렬 수행 과정


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

수행 과정


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

알고리즘

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 인덱스
7. return  A
```

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

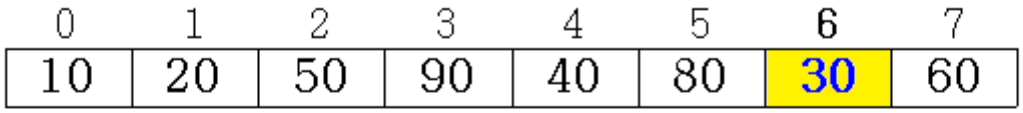
0	1	2	3	**4**	5	6	7
10	40	50	90	**20**	80	30	60

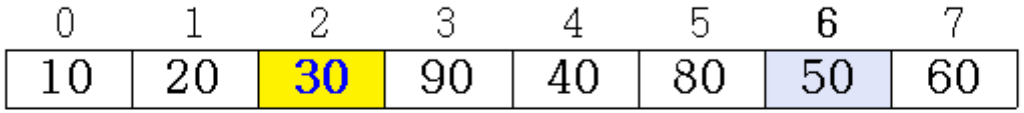
0	1	2	3	4	5	6	7
10	**20**	50	90	40	80	30	60

자리 바꿈

수행 과정

- i=2 A[2]~A[7]에서 최소값은 30 , min=6

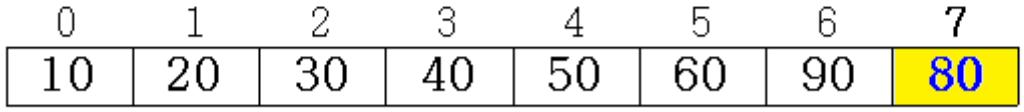


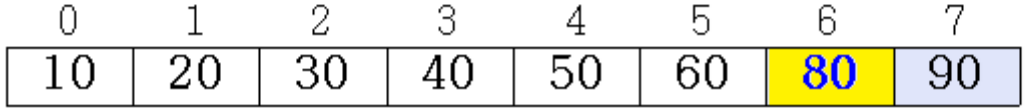


자리 바꿈

⋮

- i = 6 A[6]~A[7]에서 최소값은 80 , min=7





자리 바꿈

시간 복잡도

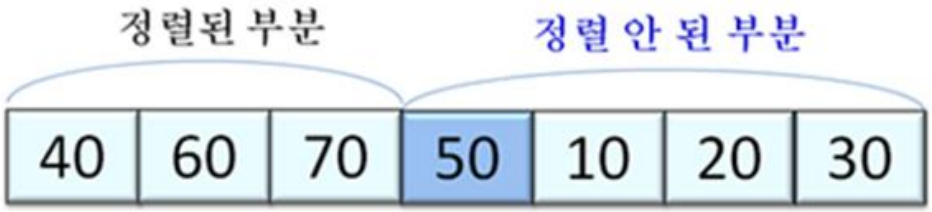
- 외부 for-루프는 (n-1)번 수행
  - i=0일 때 내부 for-루프는 (n-1)번 수행
  - i=1일 때 내부 for-루프는 (n-2)번 수행
  - ⋮
  - 마지막으로 1번 수행
- 내부의 for-루프가 수행되는 총 횟수
  - (n-1)+(n-2)+(n-3)+…+2+1 = n(n-1)/2
- 루프 내부의 if-조건이 True일 때의 자리바꿈은 O(1)
- n(n-1)/2 x O(1) = $O(n^2)$

선택 정렬의 특징

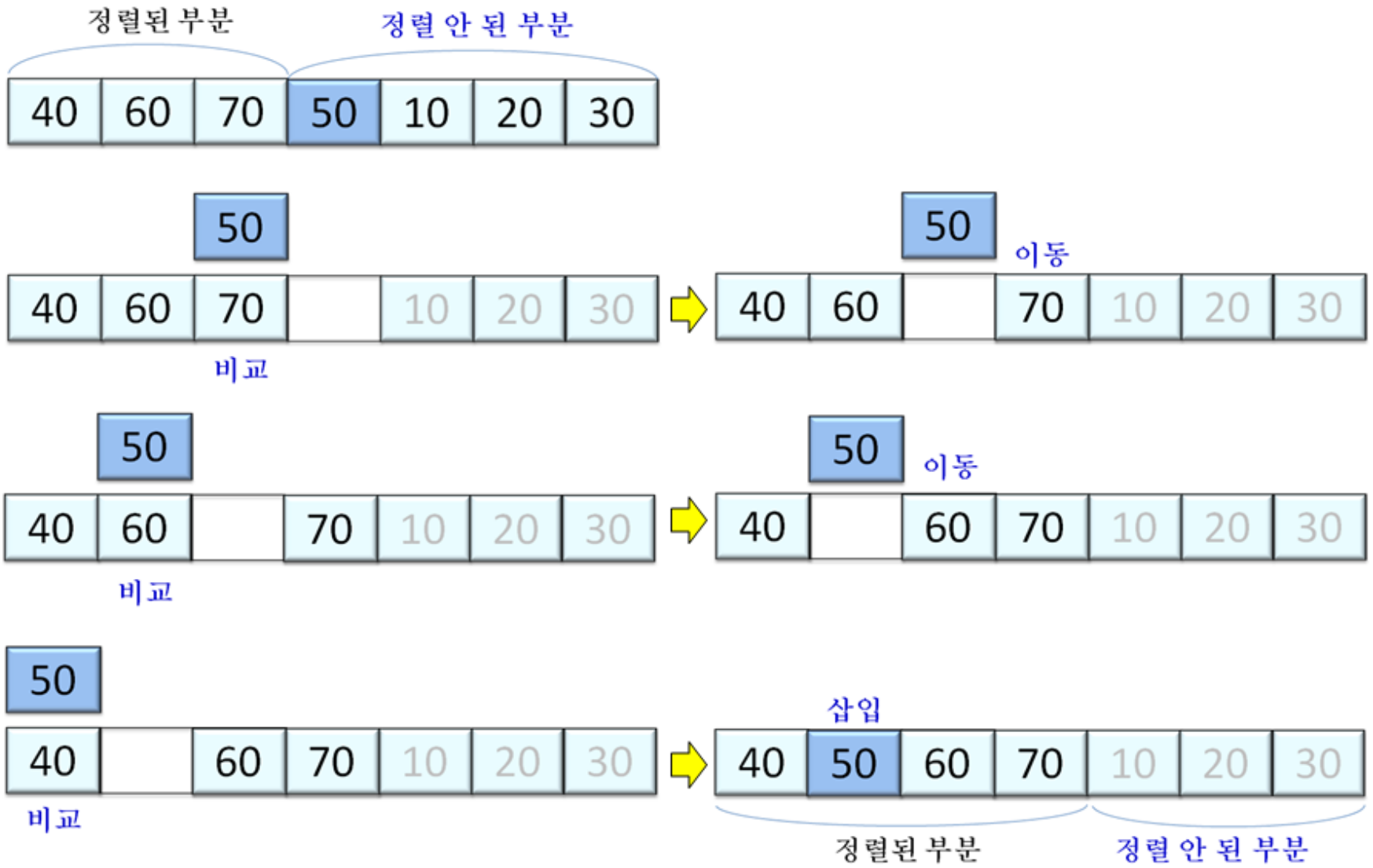
- 입력이 거의 정렬되어 있든지, 역으로 정렬되어 있든지, 랜덤하게 되어있든지 항상 일정한 시간 복잡도를 나타낸다
- 입력에 민감하지 않은(input insensitive) 알고리즘
- 원소 간의 자리바꿈 횟수가 최소인 정렬

6.3 삽입 정렬

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



삽입 정렬


정렬된 부분
정렬 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
정렬 안 된 부분

삽입 정렬

- 정렬 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, 마지막엔 정렬이 안 된 부분엔 아무 원소도 남지 않고, 입력이 정렬된다.
- 정렬은 배열의 첫 번째 원소만이 정렬된 부분에 있는 상태에서 정렬을 시작한다.

알고리즘

입력: 크기가 n인 배열 A

출력: 정렬된 배열 A

```
1. for i = 1 to n-1
2.     CurrentElement = A[i]    // 정렬 안된 부분의 가장 왼쪽 원소
3.     j ← i - 1
4.     while (j >= 0) and (A[j] > CurrentElement)
5.         A[j+1] = A[j]    // 자리 이동
7.     A [j+1] ← CurrentElement
8. return A
```

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

수행 과정

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

  자리이동 없이 50이 그 자리에

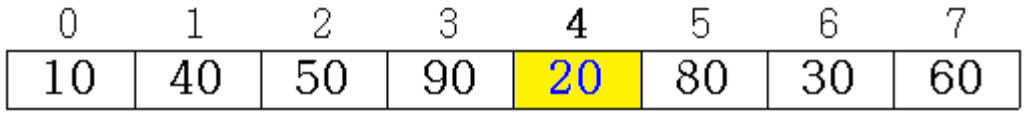
0	1	2	3	4	5	6	7
10	40	50	90	20	80	30	60

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

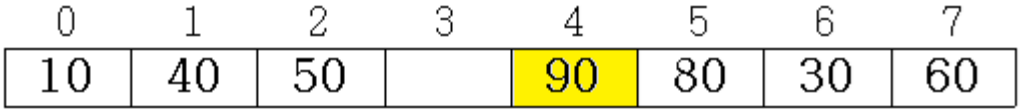
  자리이동 없이 90이 그 자리에

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

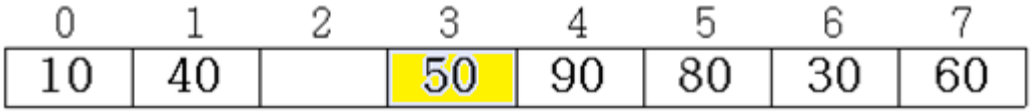
- i=4, CurrentElement=A[4]=20, j=i-1=3



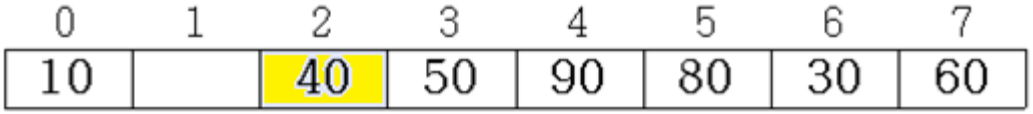
한 칸 앞으로 이동



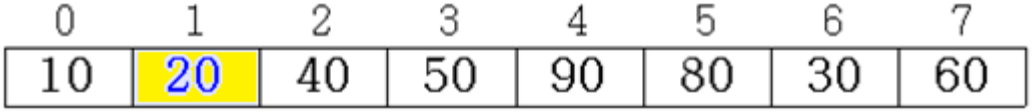
한 칸 앞으로 이동



한 칸 앞으로 이동

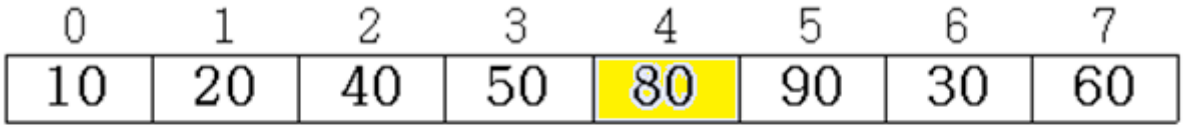


A[1]에 CurrentElement=20 저장

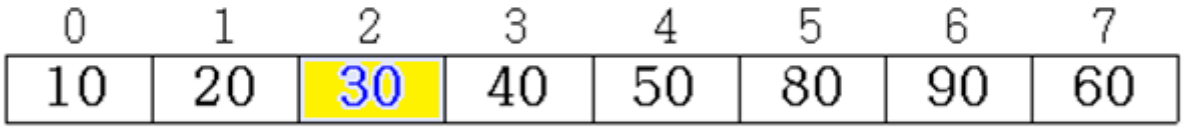


수행 과정

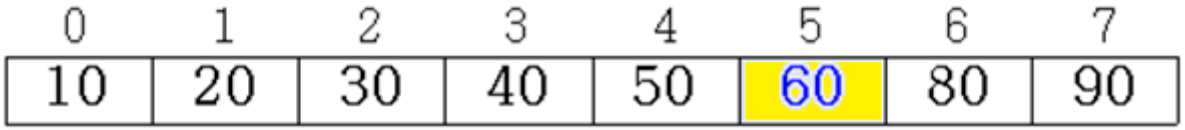
- i=5, CurrentElement=A[5]=80일 때의 결과



- i=6, CurrentElement=A[6]=30일 때의 결과



- i=7, CurrentElement=A[7]= 60일 때의 결과



최악 경우 시간 복잡도

- for-루프가 (n-1)회 수행
  - i=1일 때 while-루프는 1회 수행
  - i=2일 때 최대 2회 수행
  - ⋮
  - 마지막으로 최대 (n-1)회 수행
- 루프 내부의 line 5~6이 수행되는 총 횟수:

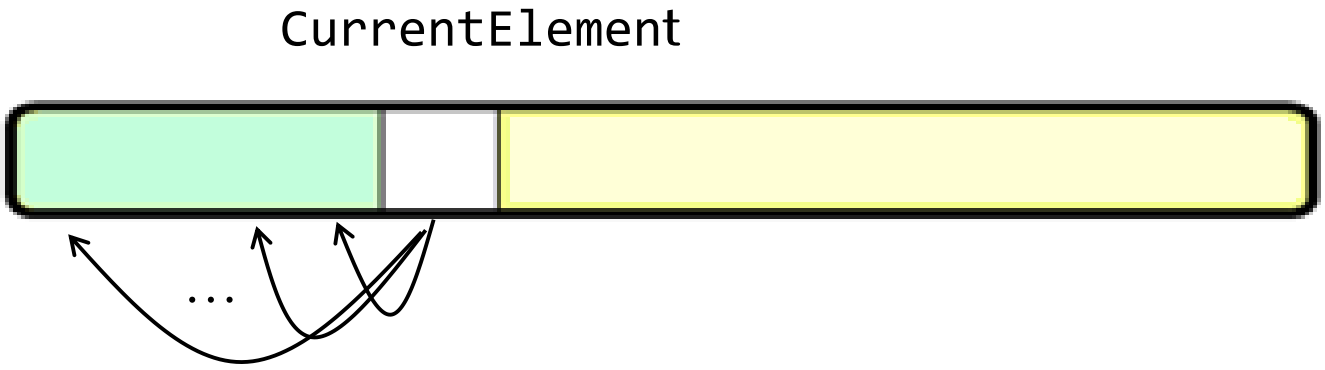
  $1 + 2 + 3 + \cdots + (n-2) + (n-1) = n(n-1)/2$
- 루프 내부는 O(1) 시간
- $n(n-1)/2 \times O(1) = O(n^2)$

최선 경우 시간 복잡도

- 입력이 이미 정렬되어 있으면, 항상 각각 CurrentElement가 자신의 왼쪽 원소와 비교 후 자리이동 없이 원래 자리에 위치하고, while-루프의 조건이 항상 False이므로 원소의 이동도 전혀 없다.
  - 따라서 (n-1)번의 비교만에 정렬을 마치게 된다.
  - 이때가 삽입 정렬의 최선 경우이고 시간 복잡도는 O(n)

평균 경우 시간 복잡도

- CurrrentElement가 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)

삽입 정렬의 특성

- 삽입 정렬은 거의 정렬된 입력에 대해서 다른 정렬 알고리즘보다 빠르다.
  - 예를 들면, 입력이 앞부분은 정렬되어 있고 뒷부분(전체 입력의 20% 이하)에 새 데이터가 있는 경우
- 입력의 크기가 작을 때 매우 좋은 성능을 보임
- 퀵 정렬, 합병 정렬에서 입력 크기가 작아지면 순환 호출을 중단하고 삽입 정렬을 사용
- Tim sort에서는 입력 크기가 64이하이면 삽입 정렬을 호출한다.

6.4 쉘 정렬

Motivation

- 버블 정렬이나 삽입 정렬이 수행되는 과정
  - '기껏해야' 이웃하는 원소의 자리바꿈
- 버블 정렬의 수행 과정
  - 작은(가벼운) 숫자가 배열의 앞부분으로 매우 느리게 이동
- 삽입 정렬에서 마지막 원소가 가장 작은 숫자라면
  - 그 숫자가 배열의 맨 앞으로 이동해야 하므로, 모든 다른 숫자들이 1칸 씩 오른쪽으로 이동해야

아이디어

- 삽입 정렬을 이용하여 배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게' 이동시키고,
- 동시에 앞부분의 큰 숫자는 뒷부분으로 '빠르게' 이동시키자.

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

h=5

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

- 각 그룹 별로 삽입 정렬 수행한 결과를 1줄에 나열해 보면 다음과 같다.

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	30	20	10	40	50	40	40	10	60	80	60	90	90	80
그룹 2															
그룹 3															
그룹 4															
그룹 5															

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

간격 조절

- 다음엔 간격을 5보다 작게 하여, 예를 들어, 3으로 하여, 3개의 그룹으로 나누어 각 그룹별로 삽입 정렬을 수행

- 마지막에는 반드시 간격을 1로 놓고 수행해야
  - 왜냐하면 다른 그룹에 속해 서로 비교되지 않은 숫자가 있을 수 있기 때문
  - 모든 원소를 1개의 그룹으로 여기는 것이고, 이는 삽입 정렬 그 자체

알고리즘

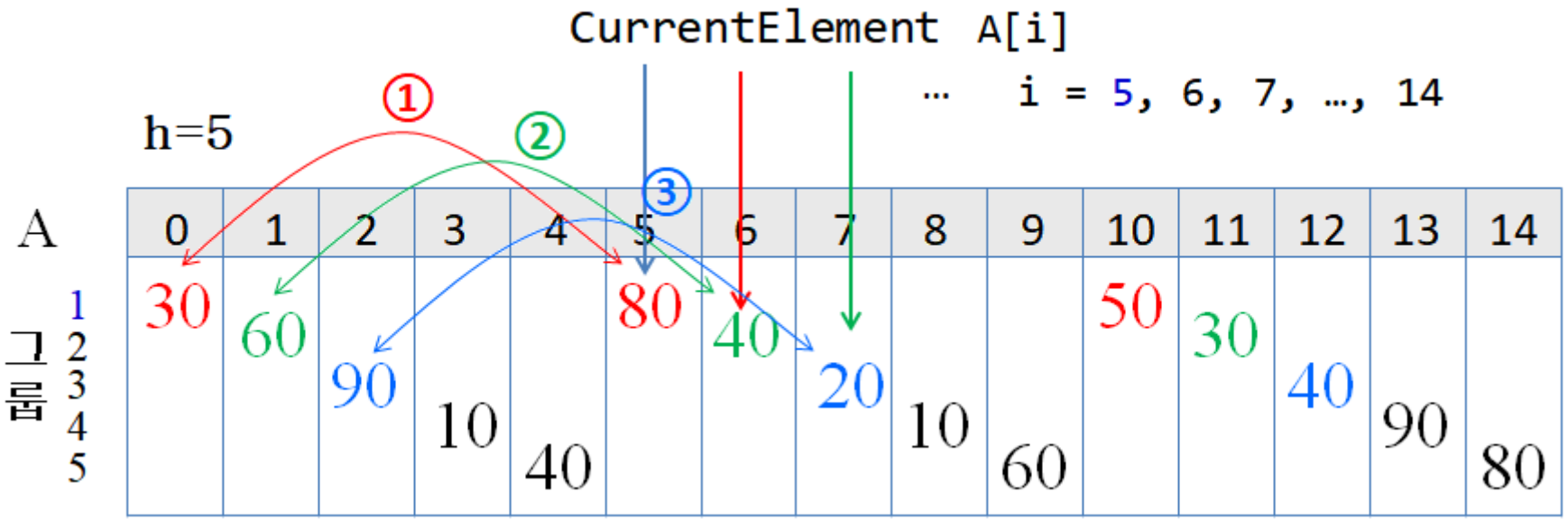
입력: 크기가 n인 배열 A

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
```

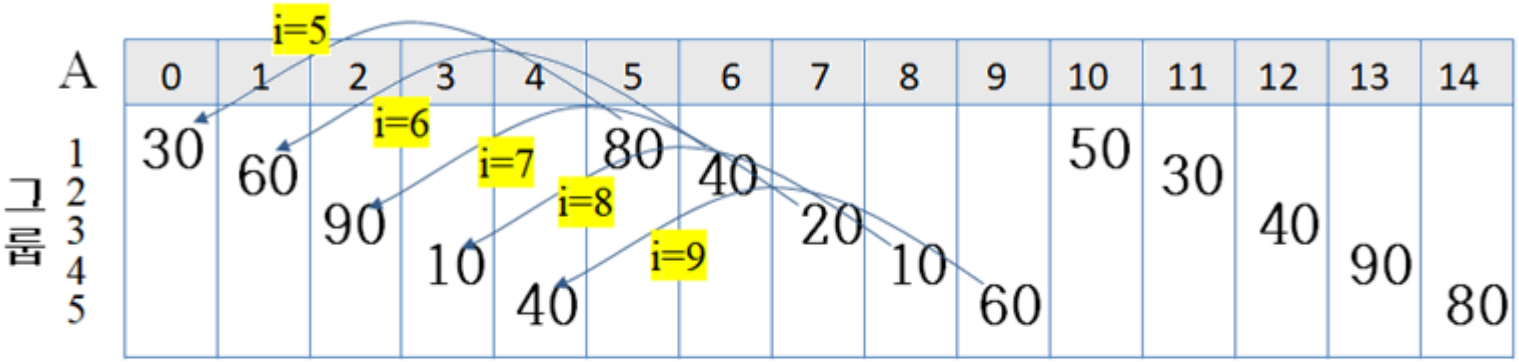
자리바꿈을 위한 원소 간의 비교 순서

- Line 2~8: for-루프에서 간격 h에 대해 그룹 별로 삽입 정렬이 수행되는데, 실제 자리바꿈을 위한 원소 간의 비교는 아래와 같은 순서로 진행

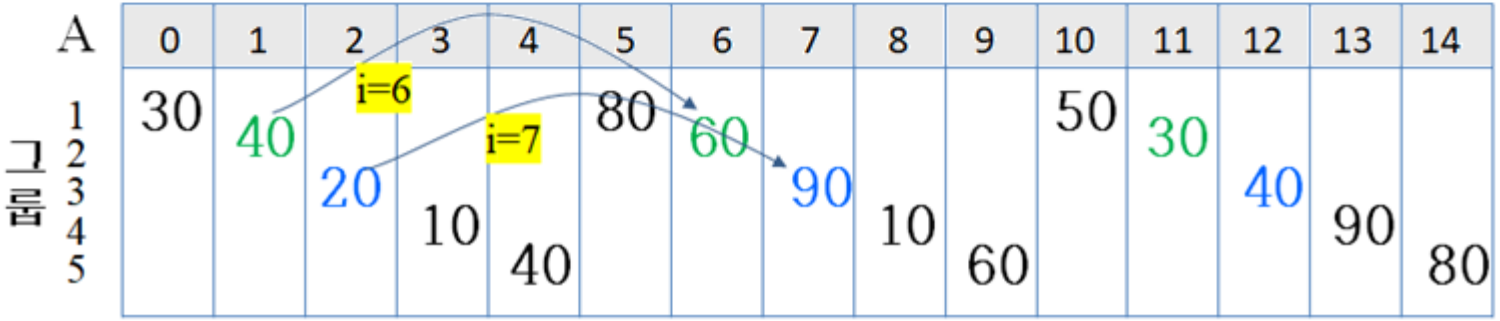


간격이 5일 때

h=5: i= 5, 6, 7, 8, 9일 때

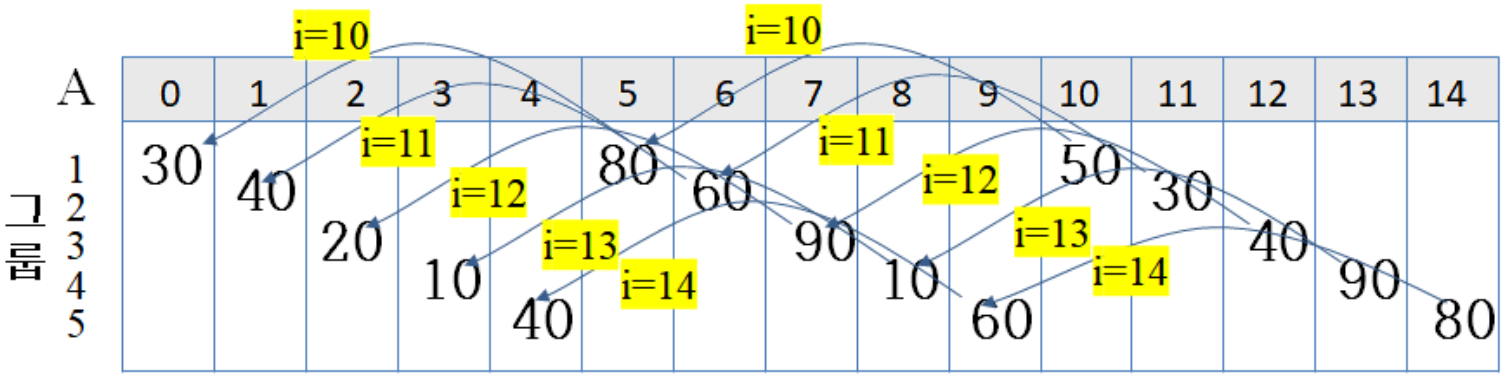


i=5, 6, 7, 8, 9일 때 이동 결과

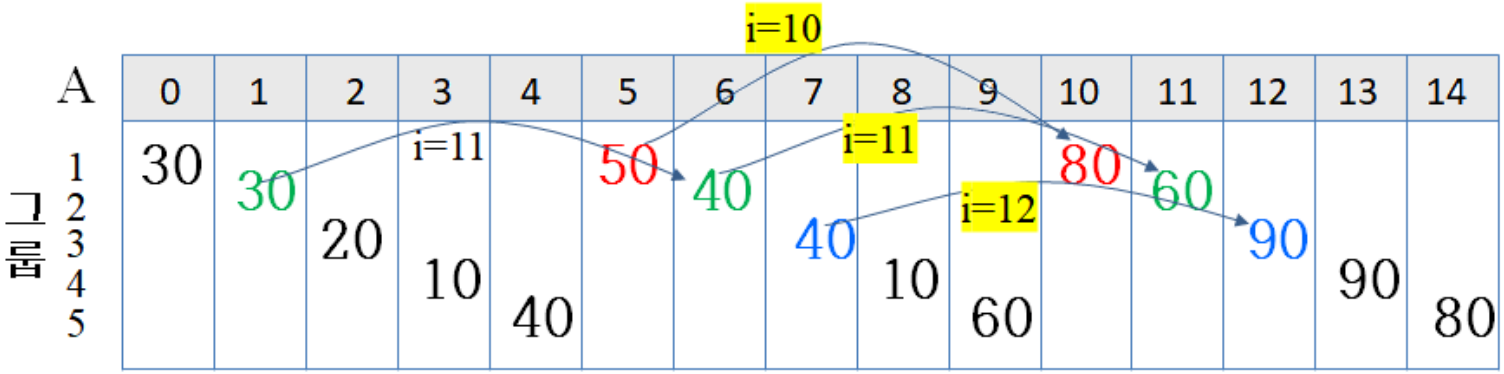


간격이 5일 때

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



i=10, 11, 12, 13, 14일 때 이동 결과



ShellSort 수행 과정

- 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
  - 그룹1은 0번째, 3번째, 6번째, 9번째, 12번째 숫자
  - 그룹2는 1번째, 4번째, 7번째, 10번째, 13번째 숫자
  - 그룹3은 2번째, 5번째, 8번째, 11번째, 14번째 숫자

h = 3일 때

- 각 그룹 별로 삽입 정렬하면,

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

h = 1일 때

- 삽입 정렬과 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

시간 복잡도

- 쉘 정렬의 시간 복잡도는 사용하는 간격에 따라 분석해야 한다.
- 최악 경우 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)
- 쉘 정렬의 시간 복잡도는 아직 풀리지 않은 문제
  - 가장 좋은 간격을 알아내야 하는 것이 선행되어야 하기 때문

쉘 정렬의 특성

- 쉘 정렬은 입력 크기가 매우 크지 않은 경우에 매우 좋은 성능을 보임

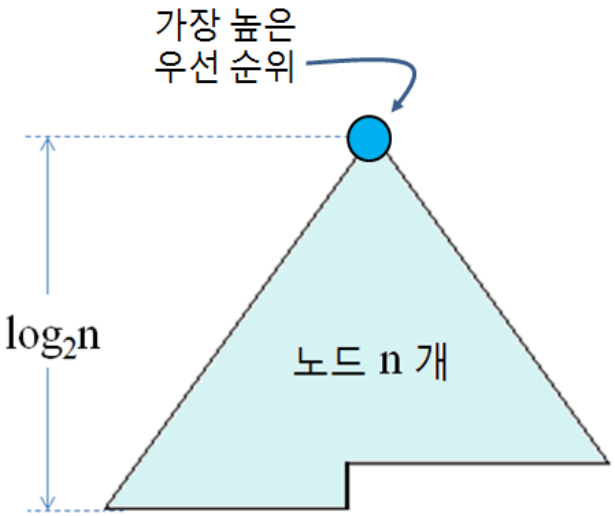
- 쉘 정렬은 임베디드(Embedded) 시스템에서 주로 사용, 쉘 정렬의 특징인 간격에 따른 그룹 별 정렬 방식이 H/W로 정렬 알고리즘을 구현하는데 매우 적합하므로

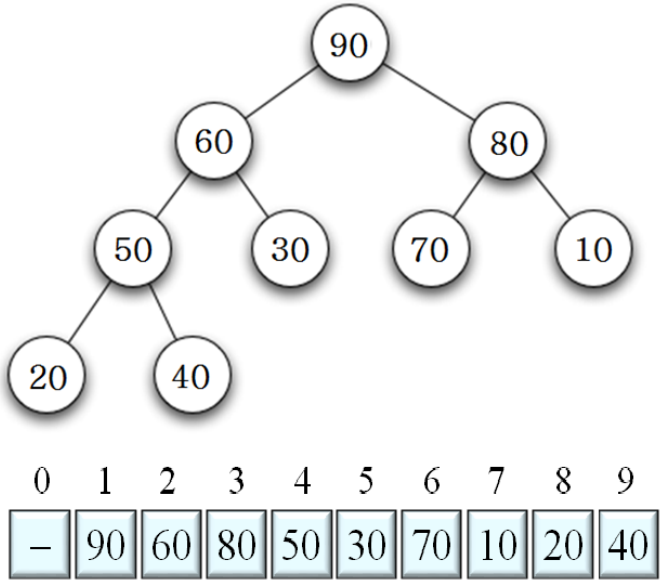
6.5 힙 정렬

- 이진 힙 (Binary Heap)
  - 힙 조건을 만족하는 완전 이진 트리 (Complete Binary Tree)
  - 힙 조건: 각 노드의 우선 순위(priority)가 자식 노드의 우선 순위보다 높다.
  - 최대 힙(Maximum Heap): 가장 큰 값이 루트에 저장
  - 최소 힙(minimum Heap): 가장 작은 값이 루트에 저장

이진 힙

- n개의 노드를 가진 힙
  - 완전 이진 트리이므로, 힙의 높이가 $\log_2 n$이며, 노드들을 빈 공간 없이 배열에 저장





힙의 노드 관계

- 힙에서 부모와 자식의 관계
  - A[i]의 부모 = A[i/2]
    - 단, i가 홀수이면 i/2에서 정수 부분만
  - A[i]의 왼쪽 자식 = A[2i]
  - A[i]의 오른쪽 자식= A[2i+1]

힙 정렬

- 힙 정렬(Heap Sort)
  - 정렬할 입력으로 최대 힙(Max heap)을 만든다.
  - 힙 루트에 가장 큰 수가 있으므로, 루트와 힙의 가장 마지막 노드를 교환한다.
    - 가장 큰 수를 배열의 맨 뒤로 옮긴 것
  - 힙 크기를 1개 줄인다.
  - 루트에 새로 저장된 숫자로 인해 위배된 힙 조건을 해결하여 힙 조건을 만족시킨다.
  - 이 과정을 반복하여 정렬한다.

알고리즘

입력: 입력이 A[1]부터 A[n]까지 저장된 배열 A

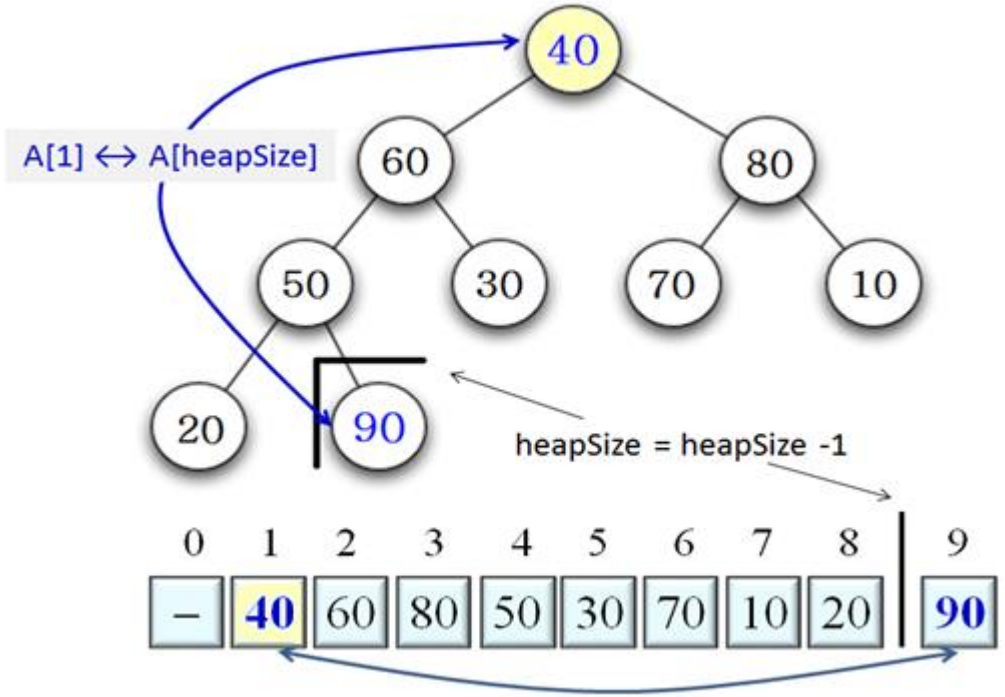
출력: 정렬된 배열 A

1. A의 숫자에 대해서 최대 힙을 만든다.
2. heapSize = n // 힙의 크기를 조절하는 변수
3. for i = 1 to n-1
4. A[1] ↔ A[heapSize] // 루트와 힙의 마지막 노드 교환
5. heapSize = heapSize -1 // 힙의 크기 1 감소
6. DownHeap() // 위배된 힙 조건을 만족시킨다.
7. return A

DownHeap()

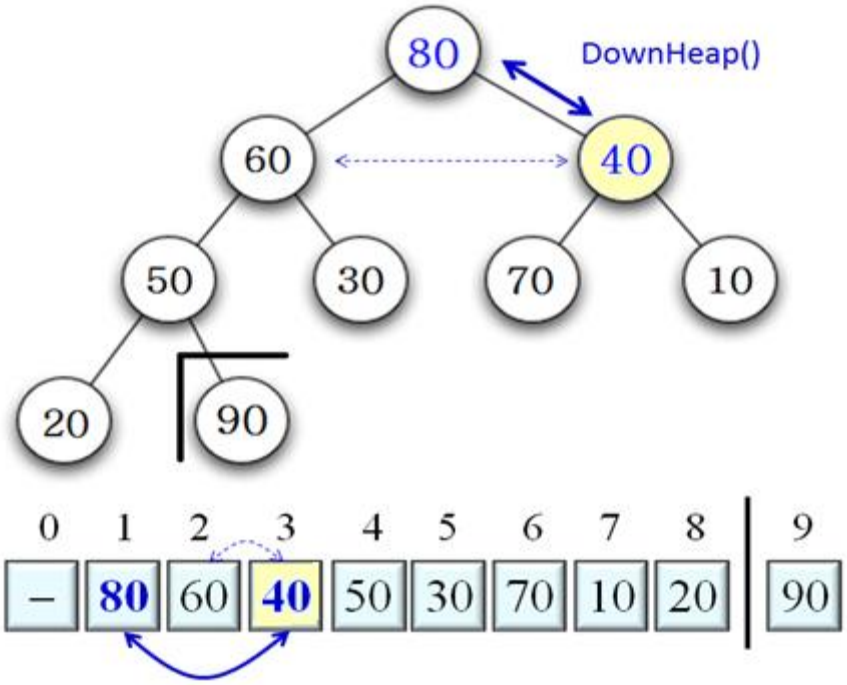
- 루트로부터 자식들 중에서 큰 값을 가진 자식과 비교하여 힙 속성이 만족될 때까지 숫자를 교환하며 이파리 방향으로 진행

HeapSort 알고리즘 수행 과정



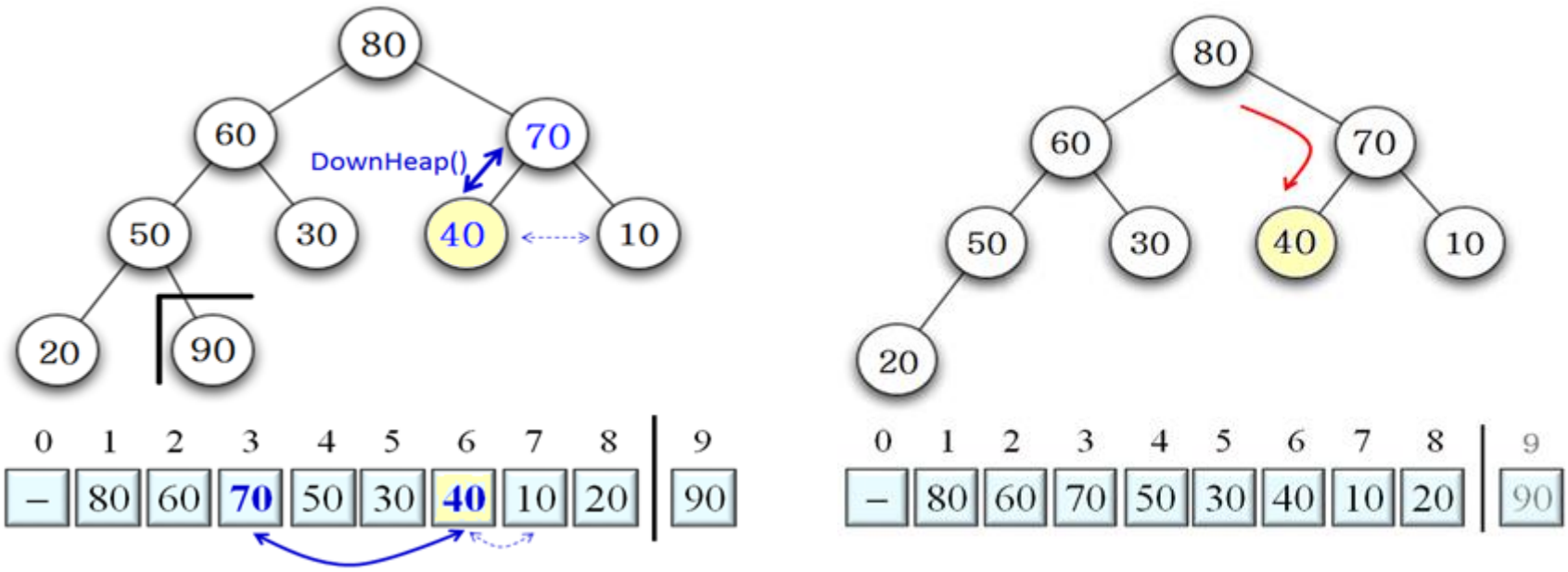
- 힙의 마지막 노드 40과 루트 90을 바꾸고, 힙의 노드 수(heapSize) 1 감소

DownHeap()



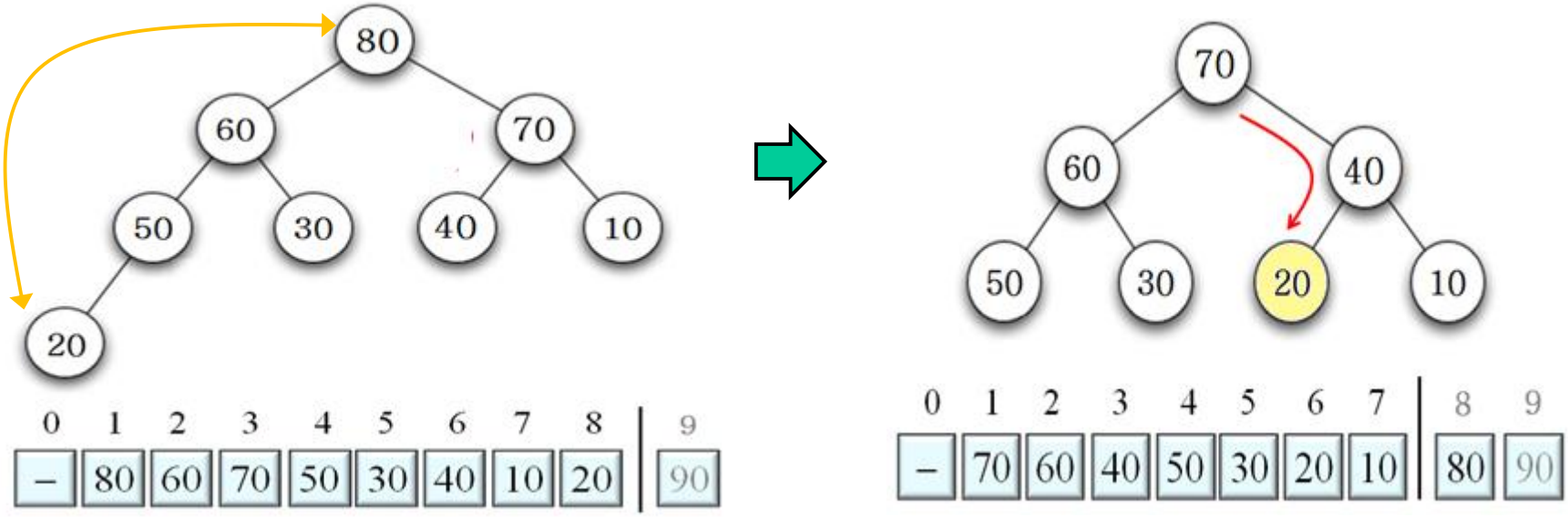
- 새로이 루트에 저장된 40이 루트의 자식 노드 60과 80보다 작으므로 자식 중에서 큰 자식 80과 루트 40을 교환

DownHeap()

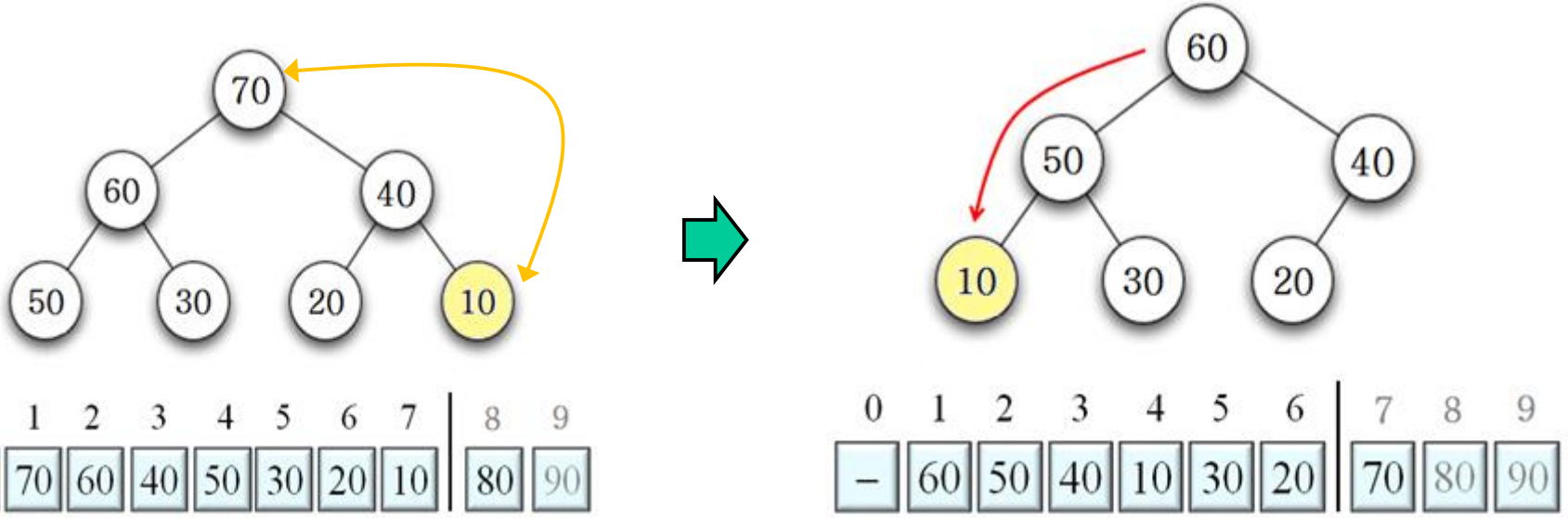


- 40은 다시 자식 노드 70과 10 중에서 큰 자식 70과 비교하여, 70과 40을 교환

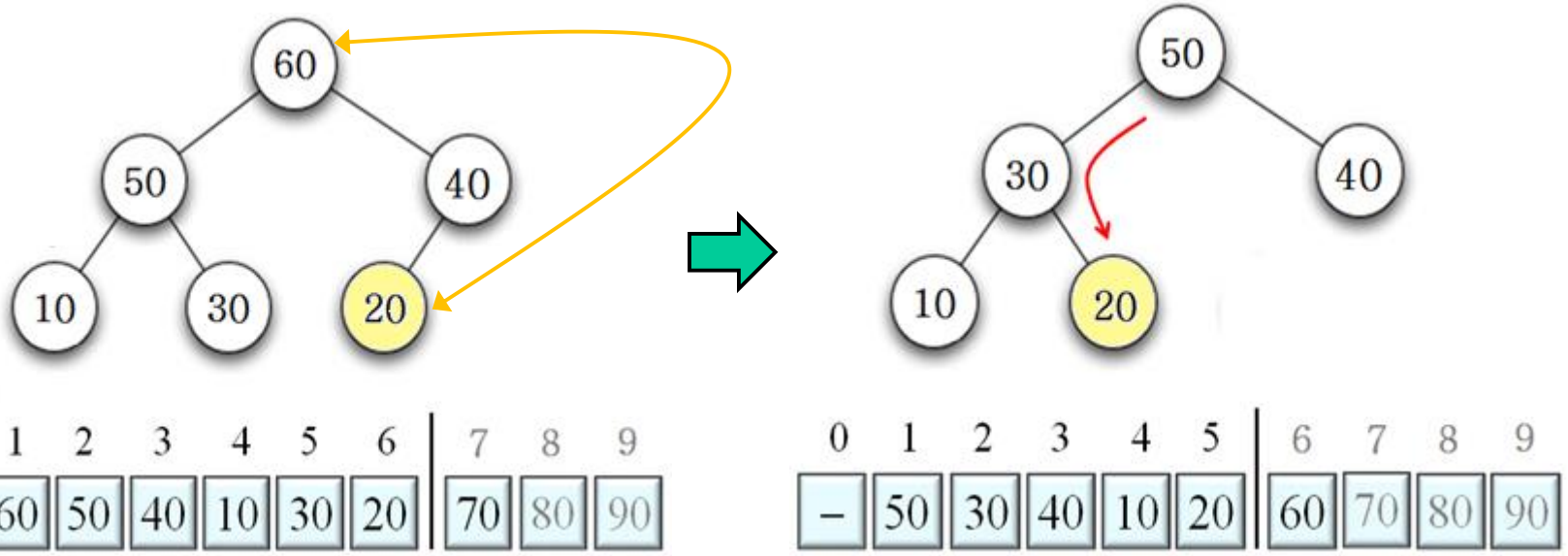
80 ↔ 20 교환 후 DownHeap 수행 결과



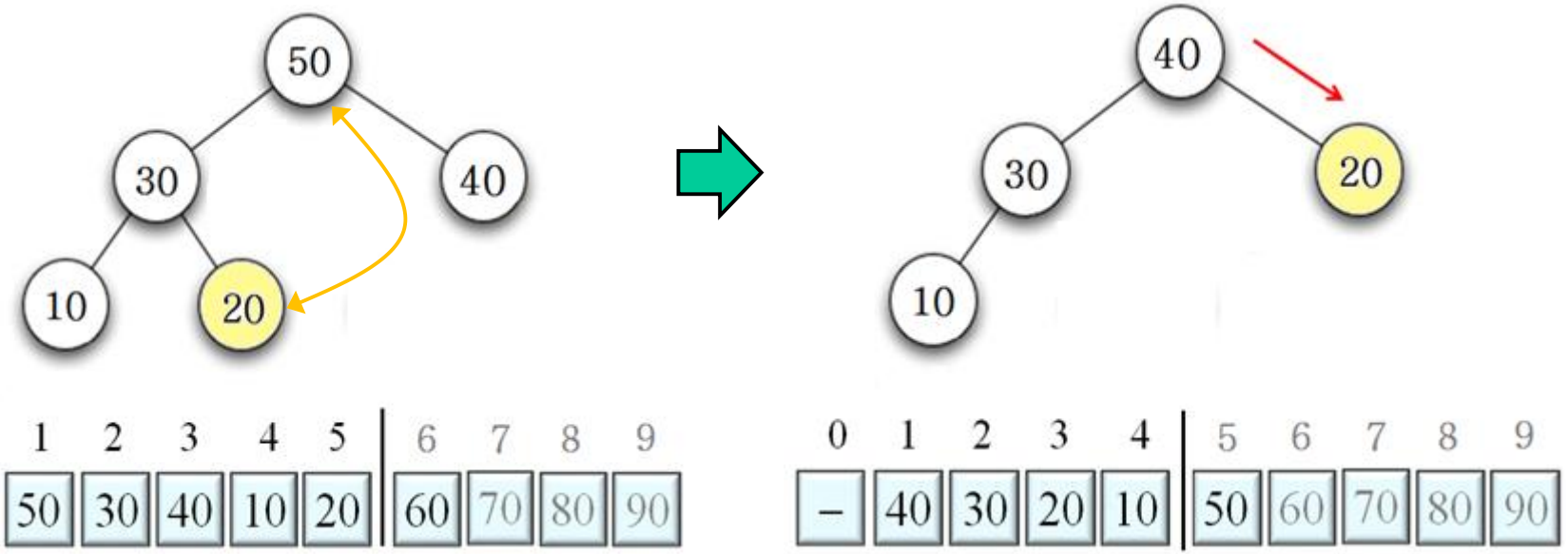
70 ↔ 10 교환 후 DownHeap 수행 결과



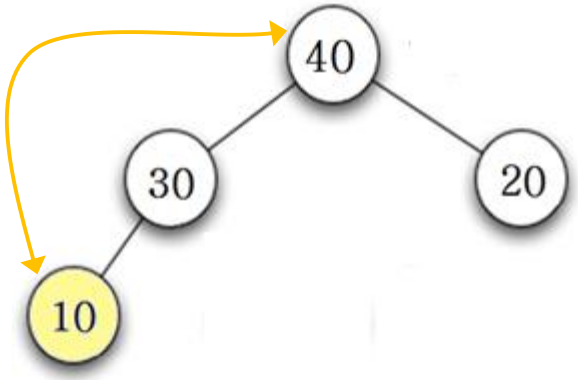
60 ↔ 20 교환 후 DownHeap 수행 결과

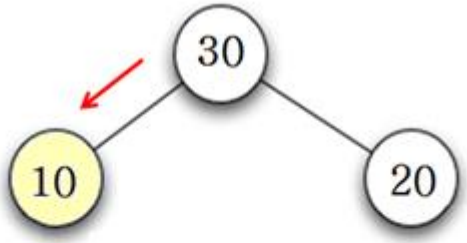


50 ↔ 20 교환 후 DownHeap 수행 결과



40 ↔ 10 교환 후 DownHeap 수행 결과

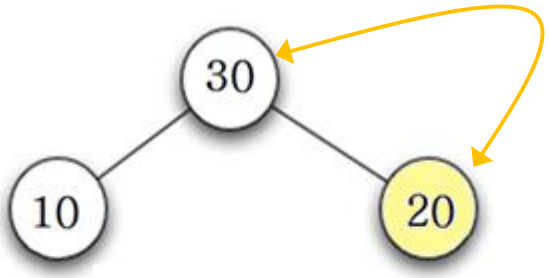


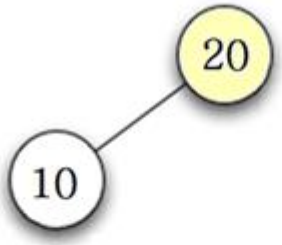


0	1	2	3	4	5	6	7	8	9
–	40	30	20	10	50	60	70	80	90

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

30 ⇔ 20 교환 후 DownHeap 수행 결과

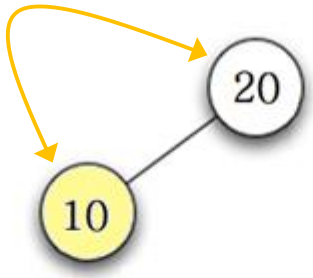


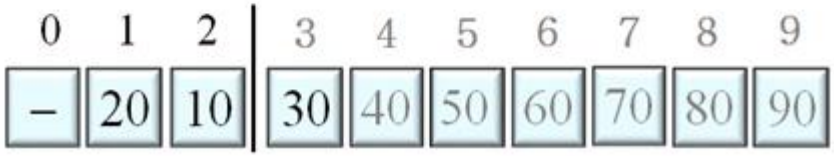


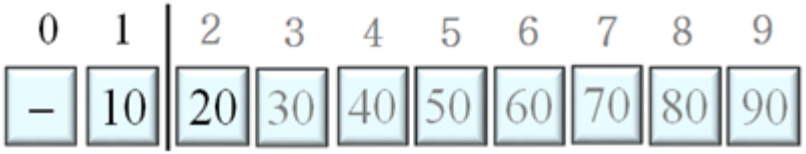
0	1	2	3	4	5	6	7	8	9
–	30	10	20	40	50	60	70	80	90

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

20 ↔ 10 교환 후 수행 결과







시간 복잡도

- 힙 만드는 데 O(n) 시간
- for-루프는 (n-1)번 수행
  - 루프 내부는 O(1) 시간
- DownHeap은 O(logn) 시간
- O(n) + (n-1) x O(logn) = **O(nlogn)**

힙 정렬의 특성

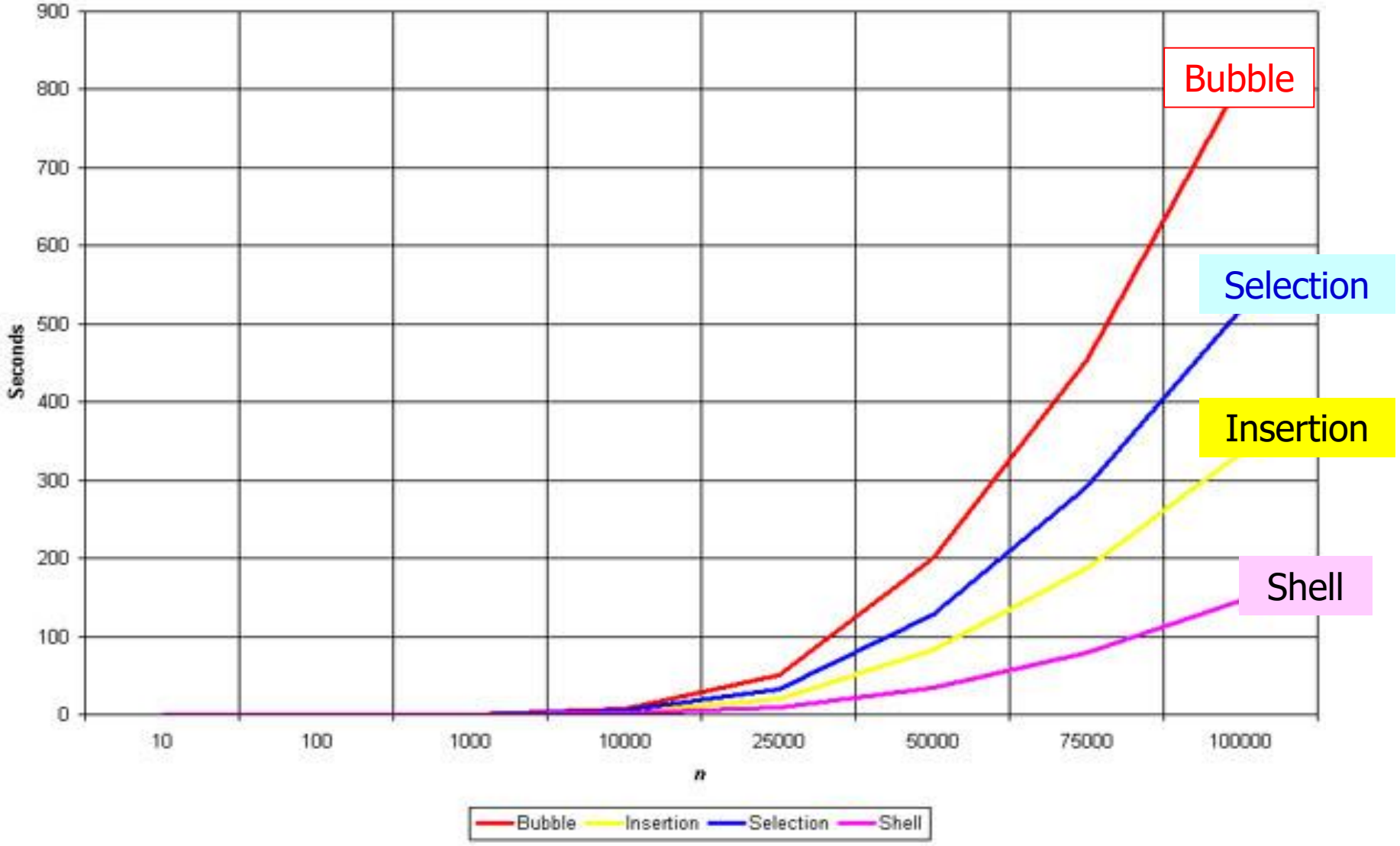
- 큰 입력에 대해 DownHeap()을 수행할 때 자식을 찾아야 하므로 너무 많은 캐시 미스로 인해 페이지 부재(page fault)를 야기시킴
- 최선, 최악, 평균 시간 복잡도가 동일 `O(nlogn)`

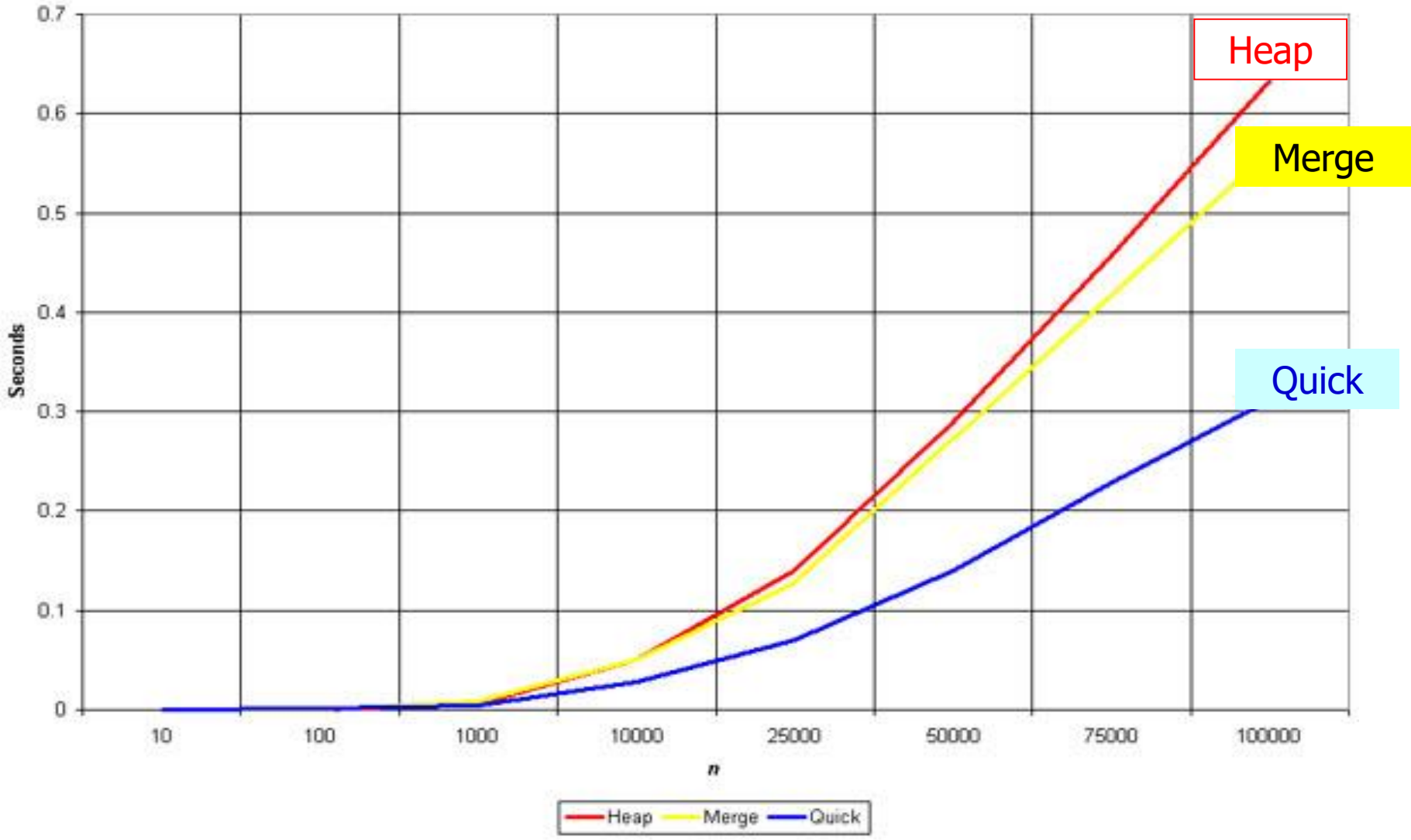
내부 정렬 알고리즘 성능 비교

	최선 경우	평균 경우	최악 경우	추가 공간	안정성
선택 정렬	n^2	n^2	n^2	$O(1)$	X
삽입 정렬	n	n^2	n^2	$O(1)$	○
쉘 정렬	$nlogn$	?	$n^{1.5}$	$O(1)$	X
힙 정렬	$nlogn$	$nlogn$	$nlogn$	$O(1)$	X
합병 정렬	$nlogn$	$nlogn$	$nlogn$	n	○
퀵 정렬†	$nlogn$	$nlogn$	n^2	$O(1)$*	X
Tim Sort	n	$nlogn$	$nlogn$	n	○

* 퀵 정렬에서 수행되는 순환 호출까지 고려한 추가 공간은 $O(\log n)$
† 이중 피벗 퀵 정렬의 이론적인 성능은 퀵 정렬과 같다.

Tim Sort에 대한 상세한 설명은 부록 V


900
800
700
600
500
400
300
200
100
0
Seconds
10
100
1000
10000
25000
50000
75000
100000
n
Bubble
Insertion
Selection
Shell
Bubble
Selection
Insertion
Shell


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Seconds
10
100
1000
10000
25000
50000
75000
100000
n
Heap
Merge
Quick
Heap
Merge
Quick

6.6 정렬 문제의 하한

- 비교 정렬 (Comparison Sort)
  - 버블 정렬, 선택 정렬, 삽입 정렬, 쉘 정렬, 힙 정렬, 합병 정렬, 퀵 정렬의 공통점은 비교가 부분적이 아닌 숫자 대 숫자로 이루어진다.

- 기수 정렬(Radix sort)은 비교 정렬이 아님
  - 숫자들을 한 자리씩 부분적으로 비교

정렬 문제의 하한

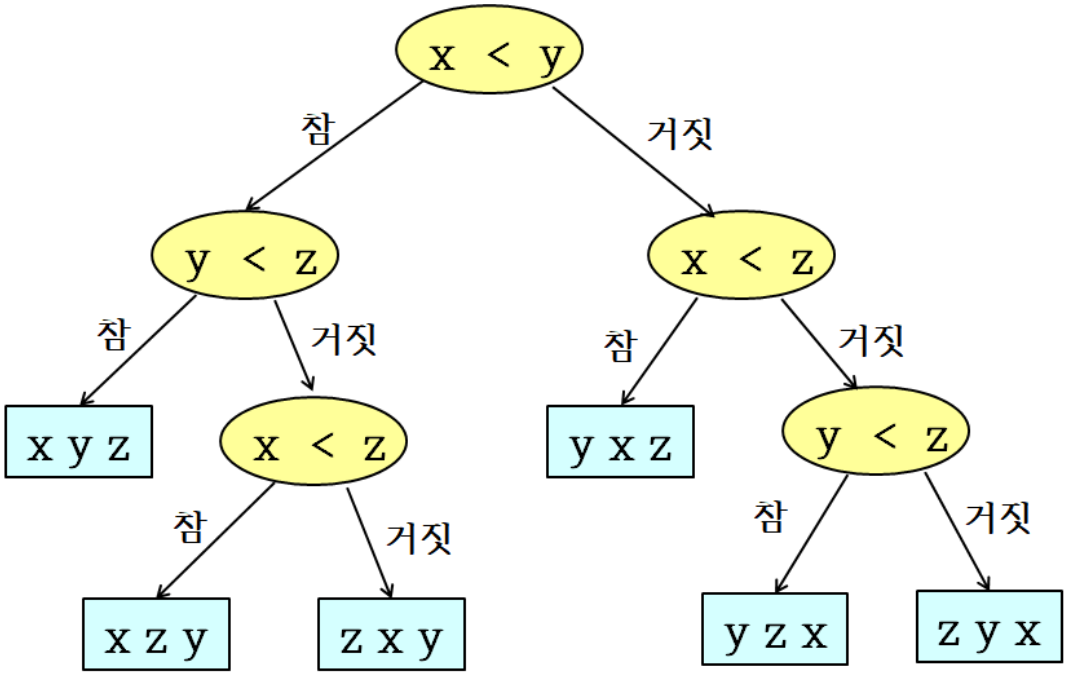
- 어떤 주어진 문제에 대해 시간 복잡도의 하한(lower bound)이라 함은 어떠한 알고리즘도 문제의 하한보다 빠르게 해를 구할 수 없음을 의미
  - 문제의 하한은 어떤 특정 알고리즘에 대한 시간 복잡도의 하한을 뜻하는 것이 아님
  - 문제가 지닌 고유한 특성 때문에 어떠한 알고리즘일지라도 해를 찾으려면 적어도 하한의 시간 복잡도만큼 시간이 걸린다는 뜻

최댓값 찾는 문제의 하한

- 최댓값을 찾기 위해 숫자들을 적어도 몇 번 비교해야 하는가?
- 어떤 방식으로 탐색하든지 적어도 (n-1)번의 비교가 필요
  - 왜냐하면 어떤 방식이라도 각 숫자를 적어도 한 번 비교해야
  - (n-1)보다 작은 비교 횟수가 의미하는 것은 n개의 숫자 중에서 적어도 1개의 숫자는 비교되지 않았다는 것
  - 비교 안 된 숫자가 가장 큰 수일 수도 있기 때문에, (n-1)보다 적은 비교 횟수로는 최댓값을 항상 찾을 수는 없다.

정렬 문제의 하한

- 3개의 서로 다른 숫자 x, y, z에 대해서, 정렬에 필요한 모든 경우의 숫자 대 숫자 비교



결정 트리 (Decision Tree)

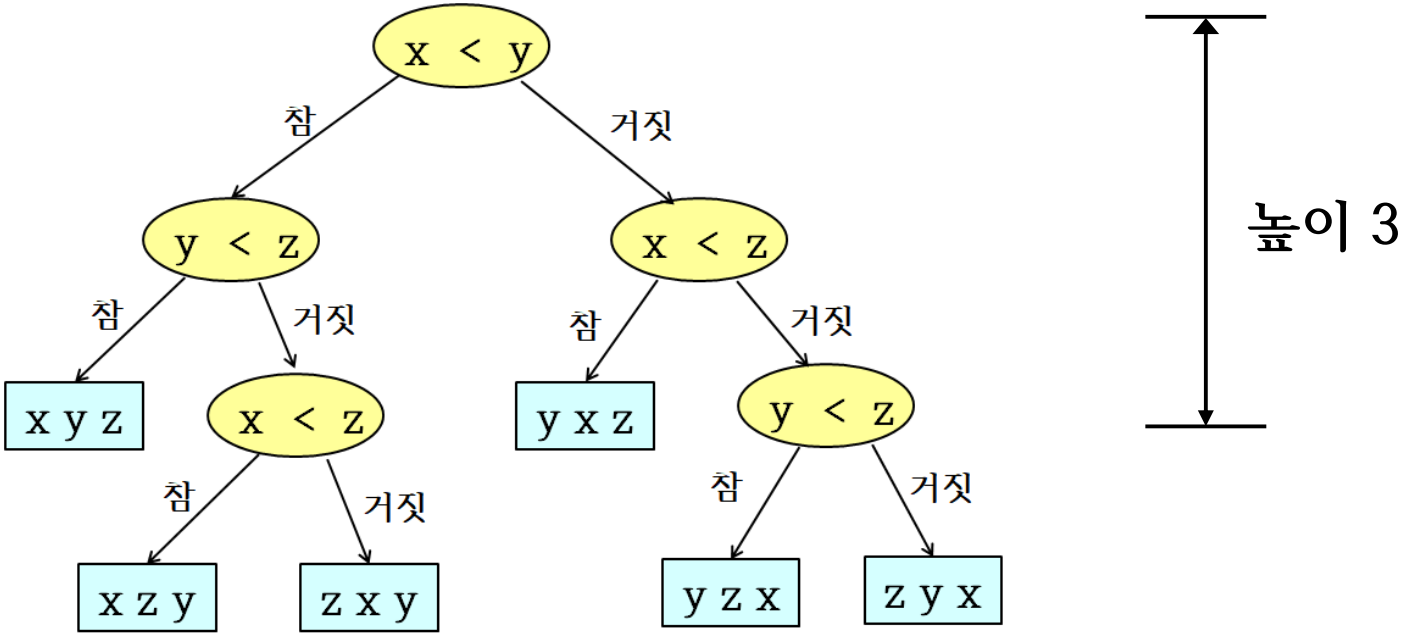
- 각 내부 노드에서는 2개의 숫자가 비교
- 비교 결과가 참이면 왼쪽으로 거짓이면 오른쪽으로 분기
- 각 이파리에는 정렬된 결과 저장

결정 트리의 특징

- 이파리의 수는 3! = 6
- 결정 트리는 이진 트리 (binary tree)이다.
- 결정 트리에는 정렬을 하는데 불필요한 내부 노드가 없다.
  - 중복 비교를 하는 노드들이 있으나, 이들은 루트로부터 각 이파리 노드의 정렬된 결과를 얻기 위해서 반드시 필요한 노드들이다.

정렬 문제의 하한

- 어느 경우에도 서로 다른 3개의 숫자가 정렬되기 위해서는 적어도 3번의 비교가 필요하다.
  - 3번의 횟수는 앞의 결정 트리의 높이



정렬 문제의 하한

- n개의 서로 다른 숫자를 비교 정렬하는 결정 트리의 높이가 비교 정렬의 하한이다.

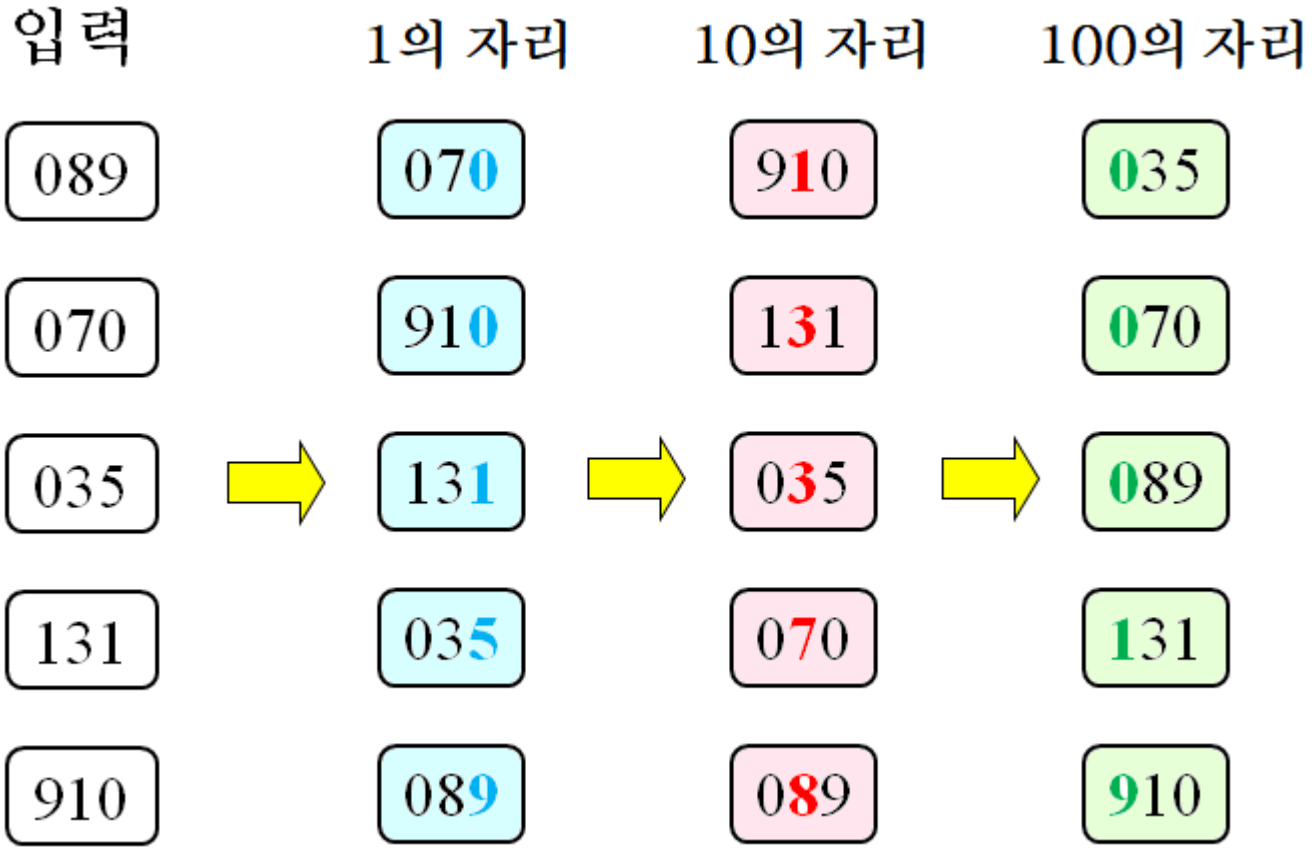
- k개의 이파리가 있는 이진 트리의 높이는 logk보다 크다.
- 따라서 n!개의 이파리를 가진 결정 트리의 높이는 log(n!)보다 크다.
- log(n!) = O(nlogn)이므로, 비교 정렬의 하한은 O(nlogn)
  - $n! \geq (n/2)^{n/2}$ 이므로 $\log(n!) \geq \log(n/2)^{n/2} = (n/2)\log(n/2) = O(n\log n)$
- 즉, O(nlogn)보다 빠른 시간 복잡도를 가진 비교 정렬 알고리즘은 존재하지 않는다.

- 점근적 표기 방식으로 하한 표기하면 $\Omega(n\log n)$

6.7 기수 정렬

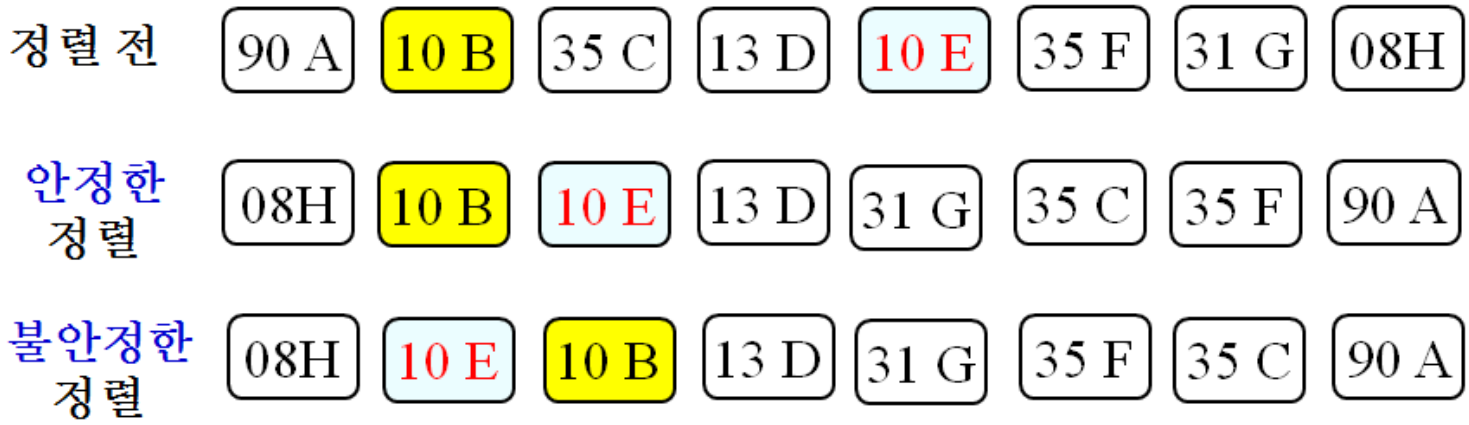
- 기수 정렬 (Radix Sort)
  - 숫자를 부분적으로 비교하며 정렬
  - 기(radix)는 특정 진수를 나타내는 숫자들
    - 10진수의 기는 0, 1, 2, …, 9
    - 2진수의 기는 0, 1
  - 기수 정렬은 제한적인 범위 내에 있는 숫자에 대해서 각 자릿수 별로 정렬하는 알고리즘

기수 정렬


입력
1의 자리
10의 자리
100의 자리
089
070
035
131
910
070
910
131
035
089
910
070
035
089
131

안정성(Stability)

- 입력에 중복된 숫자가 있을 때, 정렬된 후에도 중복된 숫자의 순서가 입력에서의 순서와 동일하면 정렬 알고리즘이 안정성 (stability)을 가진다고 한다.



알고리즘

입력: n개의 r진수의 k자리 숫자

출력: 정렬된 숫자

1. for i = 1 to k
2. 　각 숫자의 i 자리 숫자에 대해 안정한 정렬을 수행
3. return A

LSD 기수 정렬

- RadixSort는 1의 자리부터 k자리로 진행하는 경우, Least Significant Digit(LSD) 기수 정렬 또는 RL (Right-to-Left) 기수 정렬이라고 부른다.

LSD

A			temp		
c	f	b	e	d	a
e	d	a	f	e	a
d	a	b	a	b	a
f	e	a	c	f	b
f	f	b	d	a	b
e	a	b	f	f	b
a	a	c	e	a	b
a	b	a	a	a	c
b	c	e	a	c	c
a	c	c	b	c	e
c	a	e	c	a	e
b	b	f	b	b	f

A			temp		
e	d	a	d	a	b
f	e	a	e	a	b
a	b	a	a	a	c
c	f	b	c	a	e
d	a	b	a	b	a
f	f	b	b	b	f
e	a	b	a	c	c
a	a	c	b	c	e
a	c	c	e	d	a
b	c	e	f	e	a
c	a	e	c	f	b
b	b	f	f	f	b

A			temp		
d	a	b	a	a	c
e	a	b	a	b	a
a	a	c	a	c	c
c	a	e	b	c	e
a	b	a	b	b	f
b	b	f	c	a	e
a	c	c	c	f	b
b	c	e	d	a	b
e	d	a	e	a	b
f	e	a	e	d	a
c	f	b	f	e	a
f	f	b	f	f	b

MSD 기수 정렬

- k자리부터 1의 자리로 진행하는 방식은 Most Significant Digit(MSD) 기수 정렬 또는 LR (Left-to-right) 기수 정렬 이라고 부른다.

MSD

A			temp		
c	f	b	a	a	c
e	d	a	a	b	a
d	a	b	a	c	c
f	e	a	b	c	e
f	f	b	b	b	f
e	a	b	c	f	b
a	a	c	c	a	e
a	b	a	d	a	b
b	c	e	e	d	a
a	c	c	e	a	b
c	a	e	f	e	a
b	b	f	f	f	b

A			temp		
a	a	c	a	a	c
a	b	a	a	b	a
a	c	c	a	c	c
b	c	e	b	b	f
b	b	f	b	c	e
c	f	b	c	a	e
c	a	e	c	f	b
d	a	b	d	a	b
e	d	a	e	a	b
e	a	b	e	d	a
f	e	a	f	e	a
f	f	b	f	f	b

A			temp		
a	a	c	a	a	c
a	b	a	a	b	a
a	c	c	a	c	c
b	b	f	b	b	f
b	c	e	b	c	e
c	a	e	c	a	e
c	f	b	c	f	b
d	a	b	d	a	b
e	a	b	e	a	b
e	d	a	e	d	a
f	e	a	f	e	a
f	f	b	f	f	b

시간 복잡도

- for-루프가 k번 반복
  - k는 입력의 최대 자릿수
  - 루프가 1회 수행될 때 n개의 숫자의 i자리 수를 읽으며, r개로 분류하여 개수를 세고, 그 결과에 따라 숫자가 이동하므로 O(n+r) 시간 소요
- 시간 복잡도는 O(k(n+r))
  - k나 r이 입력 크기인 n보다 크지 않으면, 시간 복잡도는 O(n)

Applications

- 기수 정렬은 계좌 번호, 날짜, 주민등록번호 등으로 대용량의 상용 데이터베이스 정렬, 랜덤 128 비트 숫자로 된 초대형 파일 (예, 인터넷 주소)의 정렬, 지역 번호를 기반한 대용량의 전화 번호 정렬에 활용
- 다수의 프로세서들이 사용되는 병렬 (Parallel) 환경에서의 정렬 알고리즘에 기본 아이디어로 사용

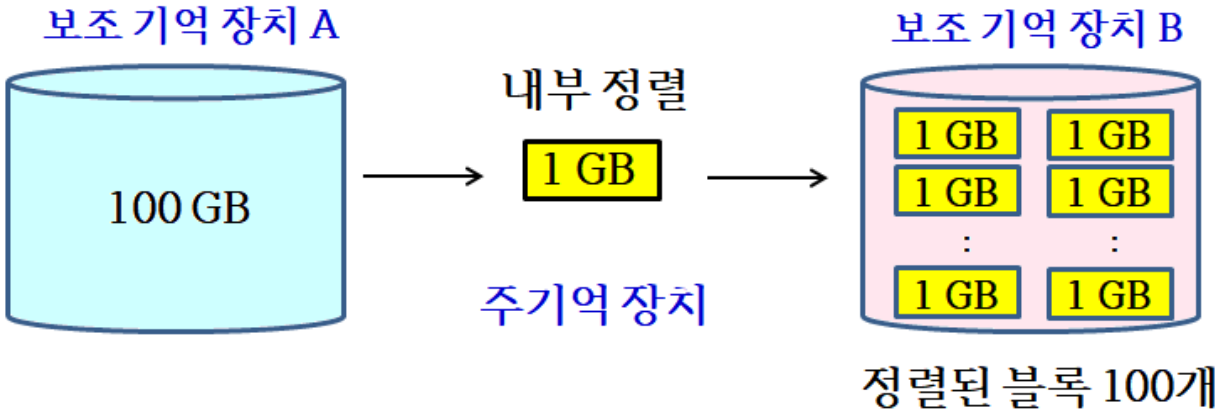
6.8 외부 정렬

- 내부 정렬 (Internal Sort)
  - 입력이 주기억 장치 (내부 메모리)에 있는 상태에서 정렬이 수행되는 정렬
- 외부 정렬 (External Sort)
  - 입력 크기가 매우 커서 읽고 쓰는 시간이 오래 걸리는 보조 기억 장치에 입력을 저장할 수밖에 없는 상태에서 수행되는 정렬
  - 주기억 장치의 용량이 1GB이고, 입력 크기가 100GB라면, 어떤 내부 정렬 알고리즘으로도 직접 정렬할 수 없다.

주기억 장치에 수용할 만큼 Read/Sort

- 외부 정렬은 입력을 분할하여 주기억 장치에 수용할 만큼의 데이터에 대해서만 내부 정렬을 수행하고, 그 결과를 보조 기억 장치에 일단 다시 저장
  - 100GB의 데이터를 1GB 만큼씩 주기억 장치로 읽어 들이고, 퀵 정렬과 같은 내부 정렬 알고리즘을 통해 정렬한 후, 다른 보조 기억 장치에 저장

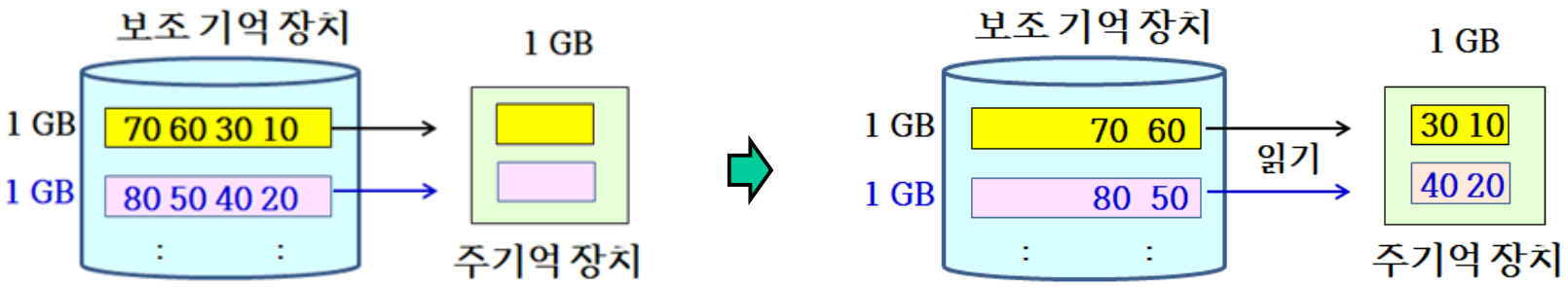
- 이를 반복하면, 원래의 입력이 100개의 정렬된 블록으로 분할되어 보조 기억 장치에 저장



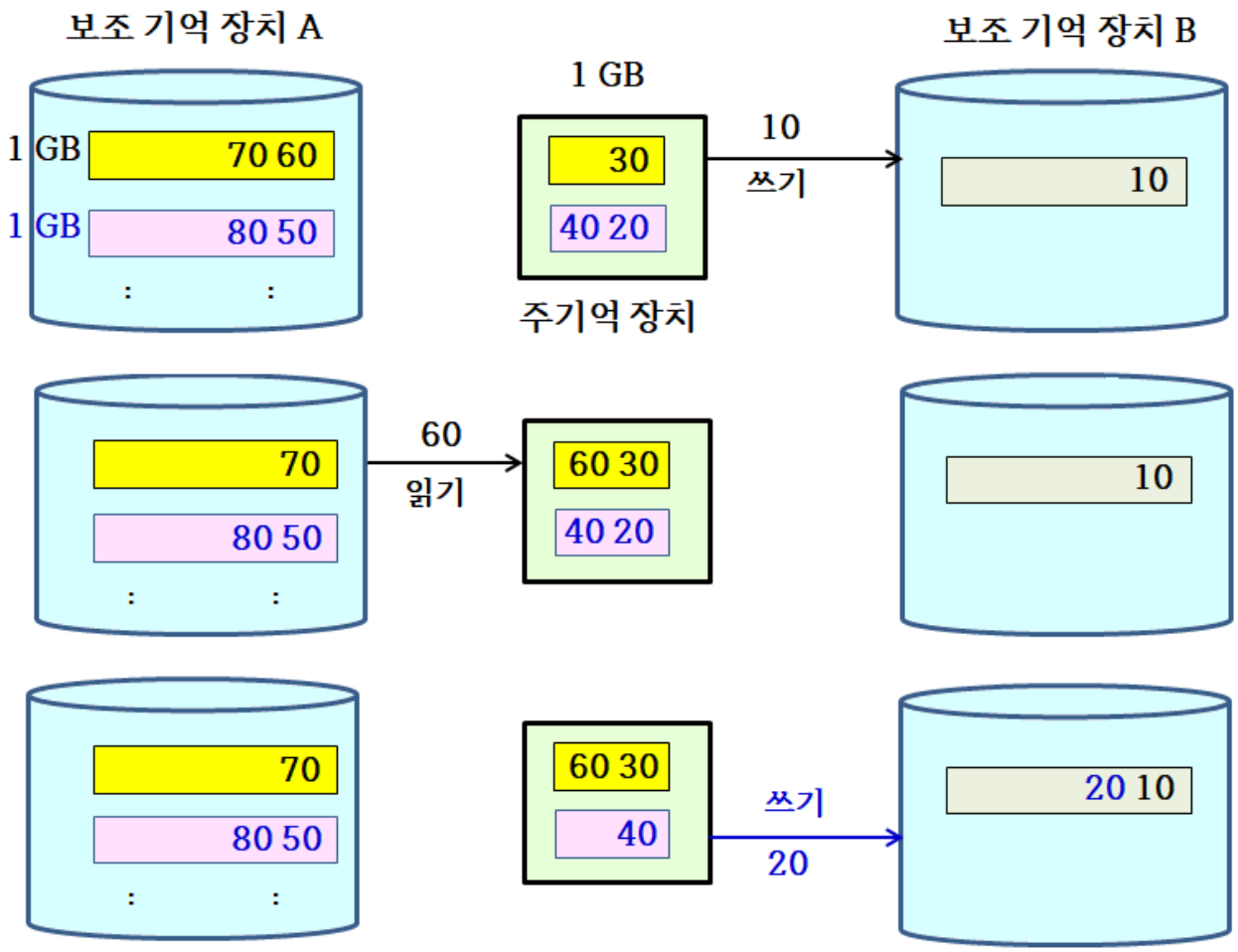
정렬된 블록의 합병

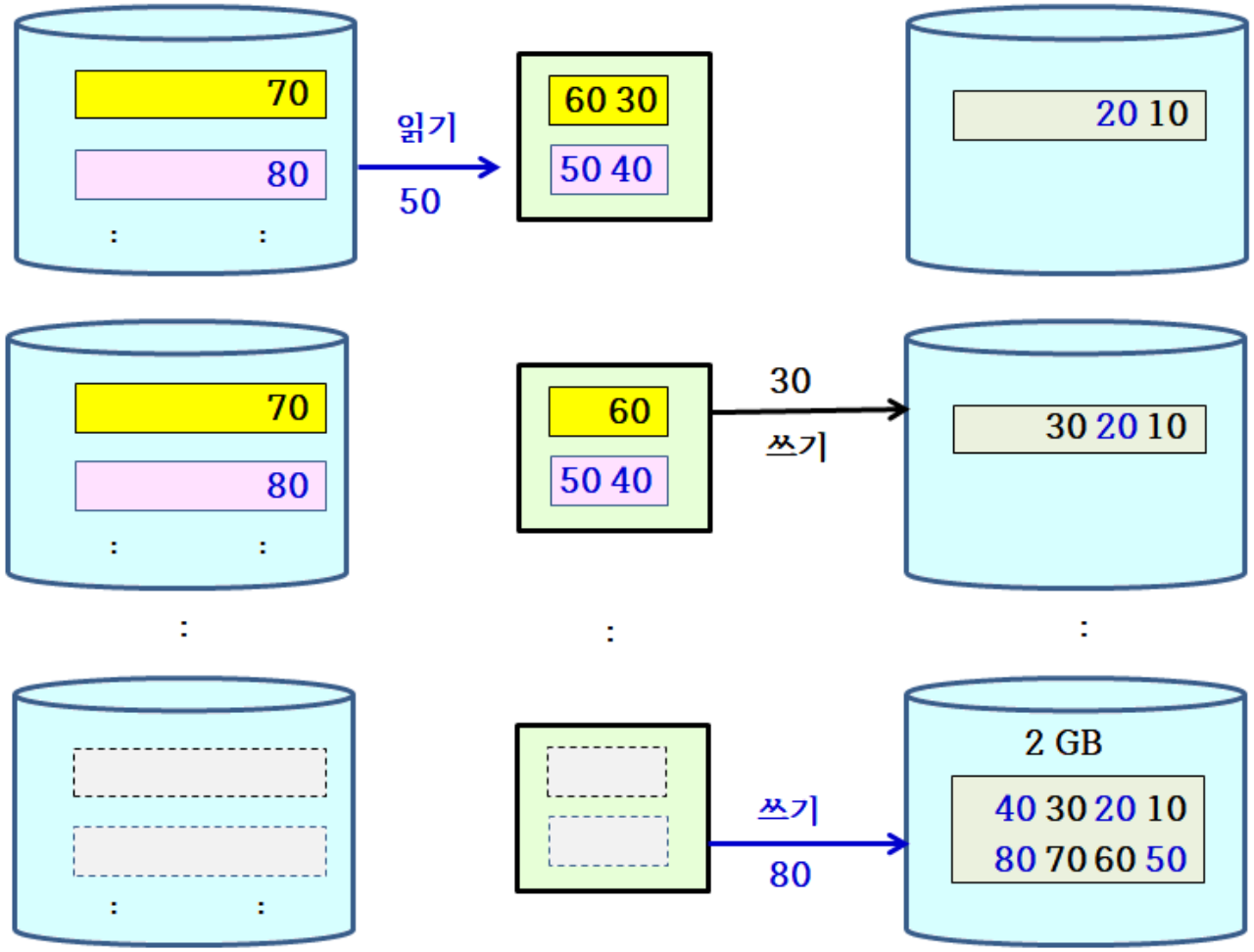
- 정렬된 블록들을 반복적인 합병(merge)을 통해서 하나의 정렬된 거대한 (100GB 크기의) 블록으로 만든다.
  - 블록들을 부분적으로 주기억 장치에 읽어 들여서, 합병을 수행하여 부분적으로 보조 기억 장치에 쓰는 과정 반복

- 블록을 부분적으로 읽어 들인 상황



1GB 블록 2개가 합병되는 과정




70
80
:
:
읽기
50
60 30
50 40
20 10
70
80
:
:
60
50 40
30
쓰기
30 20 10
:
:
:
2 GB
40 30 20 10
80 70 60 50
쓰기
80
:
:

- 나머지 98개의 블록에 대해서 이와 같이 49회를 반복하면, 2GB 블록이 총 50개 만들어지고
- 그 다음엔 2GB 블록 2개씩 짝을 지워 합병하는 과정을 총 25회 수행하면, 4GB 블록 25개가 만들어진다.
- 이러한 방식으로 계속 합병을 진행하면, 블록 크기가 2배로 커지고 블록의 수는 ½로 줄어들게 되어 결국에는 100GB 블록 1개만 남는다.

- 외부 정렬 알고리즘은 보조 기억 장치에서의 읽고 쓰기를 최소화하는 것이 매우 중요
  - 왜냐하면 보조 기억 장치의 접근 시간이 주기억 장치의 접근 시간보다 매우 오래 걸리기 때문.

- ExternalSort()
  - M = 주기억 장치의 용량
  - 외부 정렬 알고리즘은 입력이 저장된 보조 기억 장치 외에 별도의 보조 기억 장치 사용
  - 알고리즘에서 보조 기억 장치는 'HDD'로

ExternalSort 알고리즘

입력: 입력 데이터 저장된 입력 HDD

출력: 정렬된 데이터가 저장된 출력 HDD

1. 입력 HDD에 저장된 입력을 M만큼씩 주기억 장치에 읽어 들인 후 내부 정렬 알고리즘으로 정렬하여 별도의 HDD에 저장한다. 다음 단계에서 별도의 HDD는 입력 HDD로 사용되고, 입력 HDD는 출력 HDD로 사용
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 저장
4. 입력과 출력 HDD의 역할을 바꾼다.
5. `return` 출력 HDD

ExternalSort의 수행 과정

- 128GB 입력과 1GB의 주기억 장치 대한 ExternalSort의 수행 과정
  - 1GB의 정렬된 블록 128개를 별도의 HDD에 저장
  - 2GB의 정렬된 블록 64개가 출력 HDD에 만들어진다.
  - 4GB의 정렬된 블록 32개가 출력 HDD에 만들어진다.
  - 8GB의 정렬된 블록 16개 출력 HDD에 만들어진다.
    ⋮
  - 64GB의 정렬된 블록 2개가 출력 HDD에 만들어진다.
  - 128GB의 정렬된 블록 1개가 출력 HDD에 만들어진다.
  - 출력 HDD를 리턴

시간 복잡도

- 외부 정렬은 전체 데이터를 몇 번 처리 (읽고 쓰기)하는가를 가지고 시간 복잡도를 측정
- 패스(pass): 전체 데이터를 1회 처리하는 것
- 외부 정렬 알고리즘에서는 line 3에서 전체 데이터를 입력 HDD에서 읽고 합병하여 출력 HDD에 저장한다. 즉, 1 패스가 수행된다.
- while-루프가 수행된 횟수가 알고리즘의 시간 복잡도

시간 복잡도

- 입력 크기가 N이고, 메모리 크기가 M이라고 하면, line 3이 수행될 때마다 블록 크기가 2M, 4M, …, 2^kM으로 (2배) 증가
- 마지막에 만들어진 1개의 블록 크기가 2^kM이면 이 블록은 입력 전체가 합병된 결과를 가지므로, $2^k M = N$
  - k는 while-루프가 수행된 횟수

    $2^k = N/M$

    $k = \log_2(N/M)$
- 시간 복잡도: `O(log(N/M))`

2-way 합병

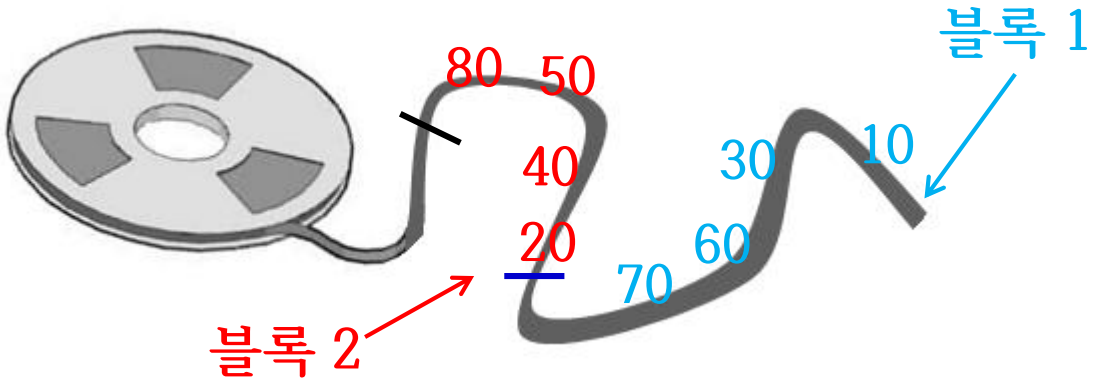
- ExternalSort 알고리즘에서는 하나의 보조 기억 장치에서 2개의 블록을 동시에 주기억 장치로 읽어 들일 수 있다는 가정

- 2개의 블록이 각각 다른 보조 기억 장치에서 읽어 들여야 하는 경우도 있다.

- 테이프 드라이브 (Tape Drive) 저장 장치는 블록을 순차적으로만 읽고 쓰는 장치이므로, 2개의 블록을 동시에 주기억 장치로 읽어 들일 수 없다.

2-way 합병

- 블록 1이 [10 30 60 70], 블록 2가 [20 40 50 80], 이 두 블록을 합병하려면 블록 1의 10을 읽고, 블록 2의 20을 읽어서 비교해야

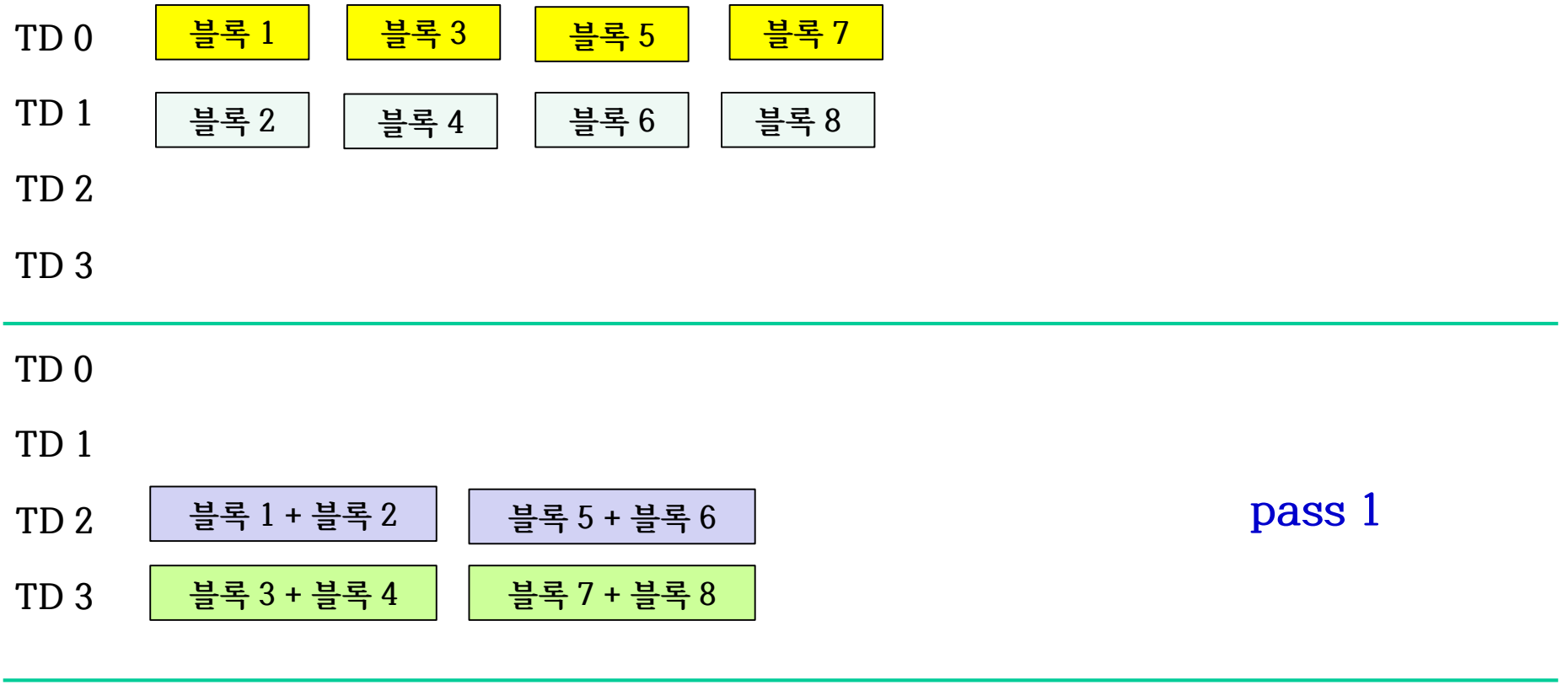


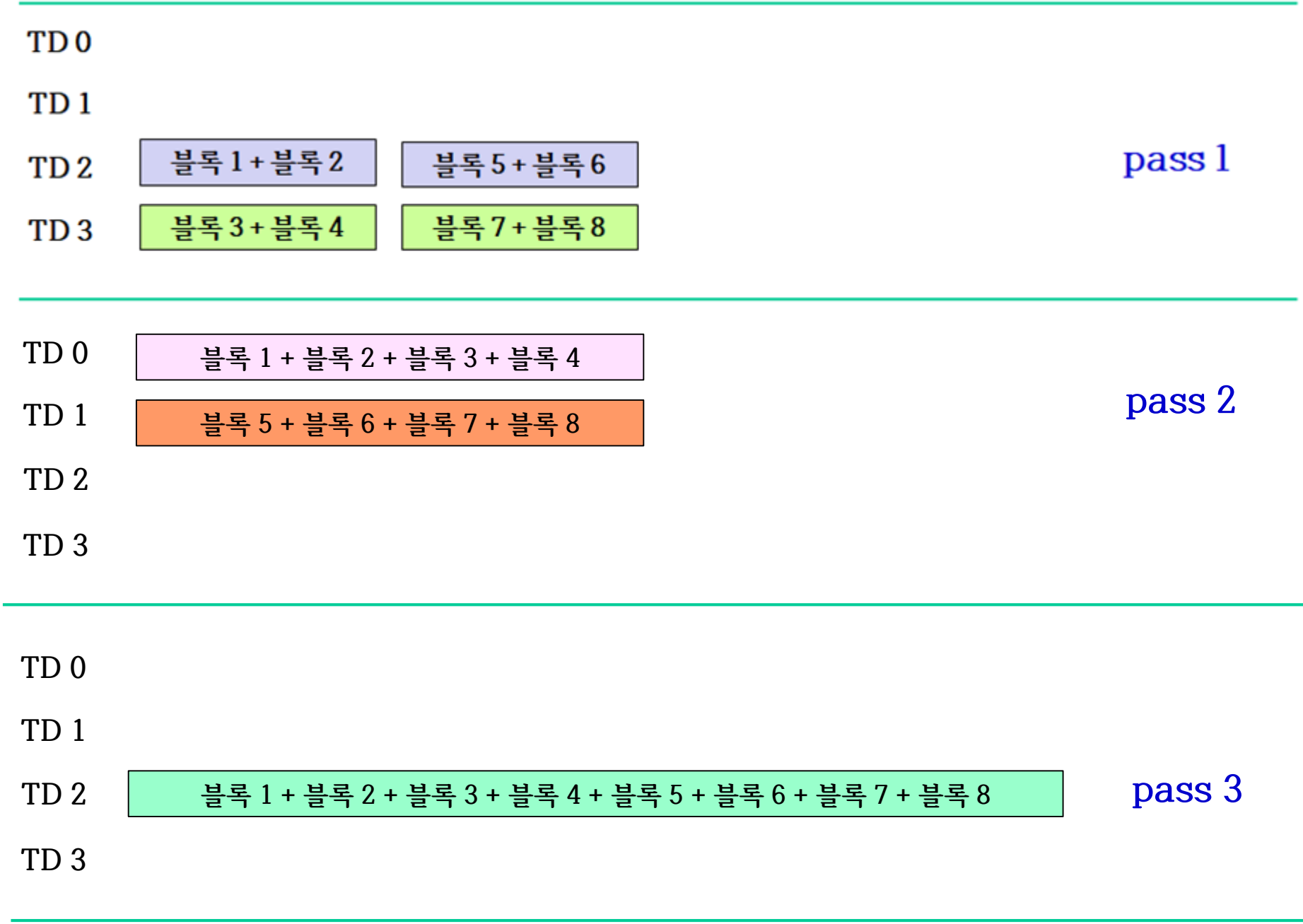
- 테이프를 한쪽 방향으로만 테이프가 감기므로, 합병하려면 블록 2의 20을 읽은 후 다시 되감아 블록 1의 두 번째 숫자인 30을 읽을 수 없다.

테이프 드라이브에서 ExternalSort 알고리즘

- ExternalSort 알고리즘의 line 3에서 2개의 블록을 읽어 들여 합병하면서 만들어지는 블록들을 2개의 저장 장치에 번갈아 저장

2-way 합병 수행 과정


TD 0
블록 1
블록 3
블록 5
블록 7
TD 1
블록 2
블록 4
블록 6
블록 8
TD 2
TD 3
TD 0
TD 1
TD 2
블록 1 + 블록 2
블록 5 + 블록 6
TD 3
블록 3 + 블록 4
블록 7 + 블록 8
pass 1

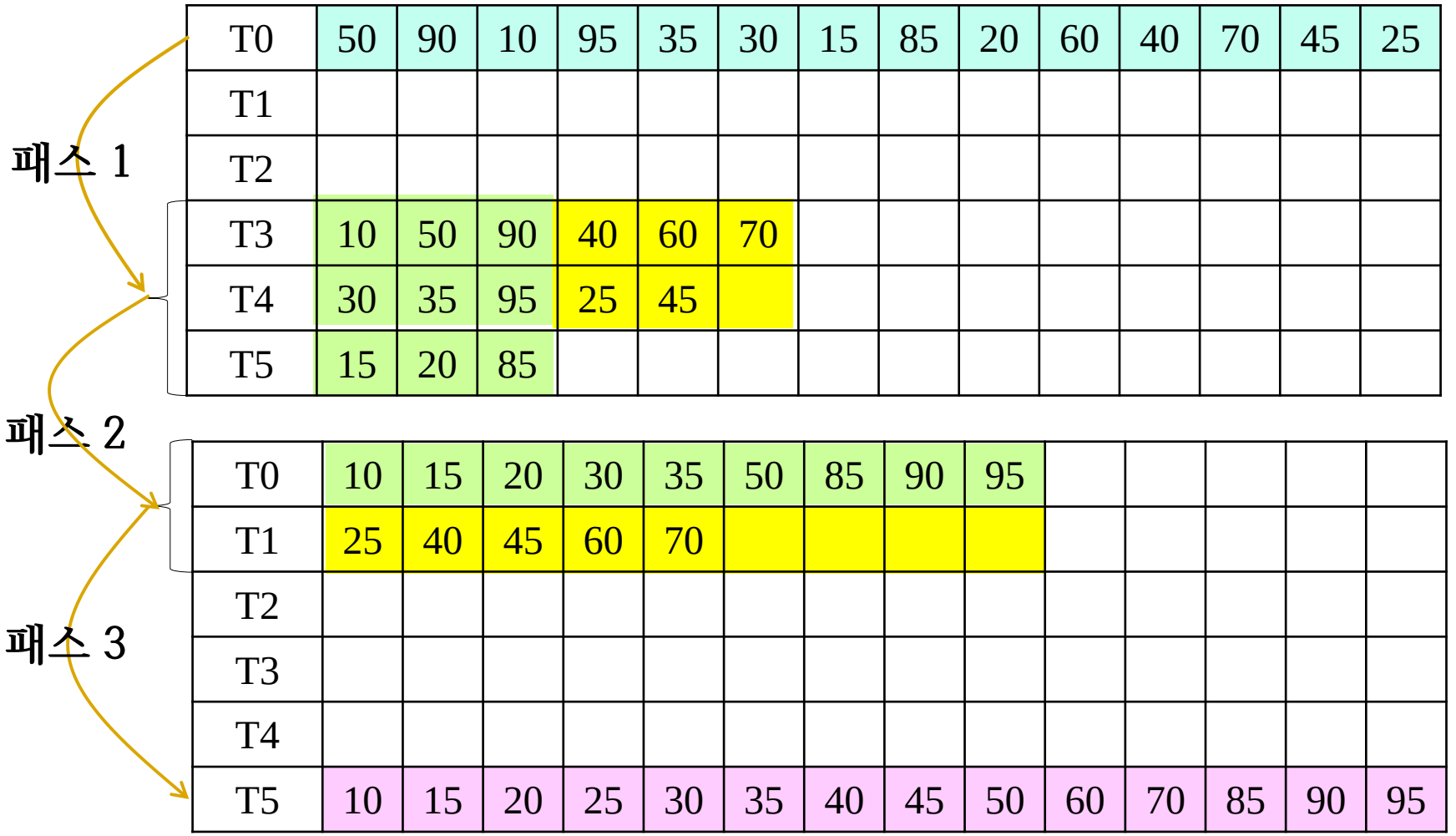

TD 0
TD 1
TD 2
블록 1 + 블록 2
블록 5 + 블록 6
TD 3
블록 3 + 블록 4
블록 7 + 블록 8
pass 1
TD 0
블록 1 + 블록 2 + 블록 3 + 블록 4
TD 1
블록 5 + 블록 6 + 블록 7 + 블록 8
TD 2
TD 3
pass 2
TD 0
TD 1
TD 2
블록 1 + 블록 2 + 블록 3 + 블록 4 + 블록 5 + 블록 6 + 블록 7 + 블록 8
TD 3
pass 3

다방향 (Multi-way) 합병

- 2-way 합병보다 빠르게 합병
- 시간 복잡도는 $O(\log_p(N/M))$
- 그러나 2p 개의 보조 기억 장치가 필요

p-way Merge

p=3, M(주기억 장치 크기)=3

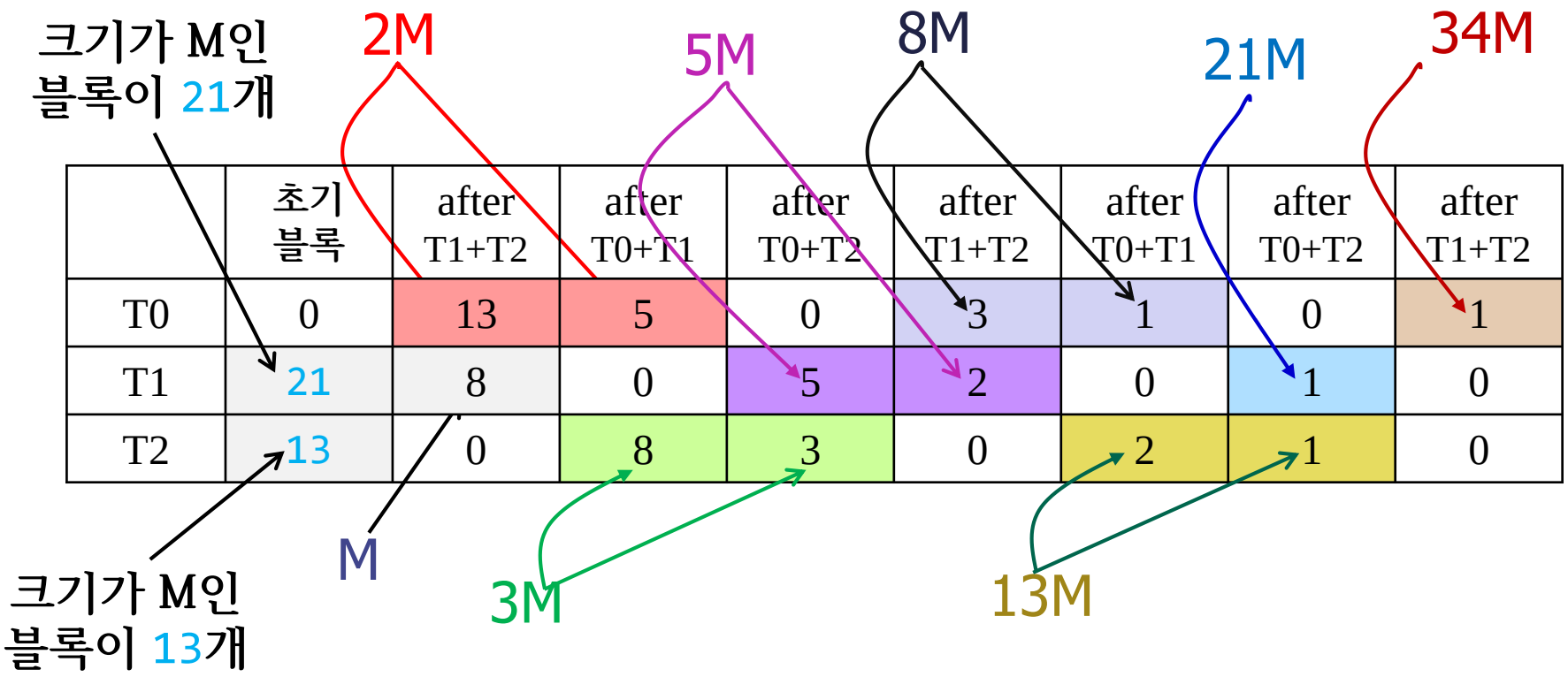


다단계 (Polyphase) 합병

- (p+1)개의 보조 기억 장치만을 가지고 p-way 합병을 하는 알고리즘
- M = 주기억 장치 크기
- 입력을 정렬된 크기가 M인 F_n 개의블록을 만든다.
  - F_n은 피보나치 수이고, 입력으로 F_n 개를 만들 수 없으면 입력 데이터보다 매우 큰 데이터로 블록을 만들어 채운다.
- p = 2일 때 F_{n-1}과 F_{n-2}로 나누어 블록들을 2개의 디바이스에 저장

p = 2

F_9 = 34 = 21+13

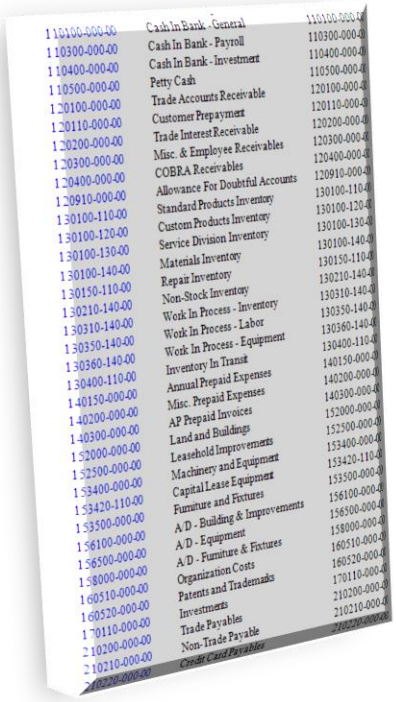


	초기 블록	after T1+T2	after T0+T1	after T0+T2	after T1+T2	after T0+T1	after T0+T2	after T1+T2
T0	0	13	5	0	3	1	0	1
T1	21	8	0	5	2	0	1	0
T2	13	0	8	3	0	2	1	0

총 패스 수 = 1+(n-2) =1+(9-2) 패스

Applications

- 물품/재고 DB, 인사 DB 등의 갱신을 위해 사용
- 통신/전화 회사의 전화 번호 정렬
- 인터넷의 IP주소 관리
- 은행에서의 계좌 관리
- 일반적인 DB의 중복된 데이터 제거
- 프로세서 스케줄링
- 소셜 네트워크
- 기하 (Geometric) 알고리즘
- 그리디 알고리즘
- 근사 알고리즘 등



요약

- 정렬 알고리즘은 내부 정렬 (Internal sort)과 외부정렬 (External sort)로 분류한
- 내부 정렬은 입력의 크기가 주기억 장치의 공간보다 크지 않은 경우에 수행되는 정렬이
- 외부 정렬은 입력의 크기가 주기억 장치 공간보다 큰 경우에는, 보조 기억 장치에 있는 입력을 여러 번에 나누어 주기억 장치에 읽어 들인 후, 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복하는 정렬
- 버블 정렬(Bubble Sort)은 이웃하는 숫자를 비교하여 작은 수를 앞쪽으로 이동시키는 과정을 반복하여 정렬이고, 시간 복잡도는 $O(n^2)$

요약

- 선택 정렬(Selection Sort)은 매번 최소값을 선택하여 정렬하며, 시간 복잡도는 $O(n^2)$
- 삽입 정렬(Insertion Sort)은 정렬 안 된 부분에 있는 원소 하나를 정렬된 부분의 알맞은 위치에 삽입하여 정렬하며, 시간 복잡도는 $O(n^2)$이다. 또한 최선 경우 시간 복잡도는 $O(n)$이고, 평균 경우 시간 복잡도는 $O(n^2)$
- 쉘 정렬(Shell Sort)은 삽입 정렬을 이용하여 배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게' 이동시키고, 동시에 앞부분의 큰 숫자는 뒷부분으로 이동시키는 과정을 반복하여 정렬하는 알고리즘. 시간 복잡도는 $O(n^2)$

요약

- 힙 정렬(Heap Sort)은 힙 자료 구조를 이용하는 정렬 알고리즘이고, 시간 복잡도는 $O(n\log n)$
- 정렬 문제의 하한(Lower Bound)은 $\Omega(n\log n)$
- 기수 정렬(Radix Sort)은 숫자를 부분적으로 비교하는 정렬 방법이다. 시간 복잡도는 $O(k(n+r))$. 단, k는 자릿수, r은 기(radix, 진수)
- 외부 정렬(External Sort)은 내부 정렬을 이용하여 부분적으로 데이터를 읽어 합병하는 과정을 반복하는 정렬 방법이다. 다방향 (p-way Merge) 합병과 다단계 합병 (Polyphase Merge) 방법이 있다.