

정렬 Sorting

I. 기초적 정렬

1. 선택 정렬
2. 버블 정렬
3. 삽입 정렬

II. 고급 정렬

4. 병합 정렬
5. 퀵 정렬
6. 힙 정렬
7. 셀 정렬

III. 특수 정렬

8. 계수 정렬
9. 기수 정렬
10. 버킷 정렬

정렬 알고리즘 개요

대부분의 정렬 알고리즘은 $\Omega(n \log n)$ 과 $O(n^2)$ 사이
입력이 특수한 성질을 만족하는 경우에는 $\Theta(n)$ 도 가능

예: 모든 원소가 $0 \sim n$ 사이의 정수, ...

I. 기초적 정렬

1. 선택 정렬 Selection sort
2. 버블 정렬 Bubble Sort
3. 삽입 정렬 Insertion Sort

1. 선택 정렬 Insertion Sort

단위 순환

최대 원소를 찾는다

최대 원소와 맨 오른쪽 원소를 자리 바꾼다

맨 오른쪽 자리를 관심 대상에서 제외한다

원소가 1 개 남을 때까지 위의 순환을 반복

Animation



알고리즘 selectionSort()

```
selectionSort(A[], n):  ◀ 배열 A[0...n-1]을 정렬한다
    for last ← n-1 downto 1
        A[0...last] 중 가장 큰 수 A[k]를 찾는다
        A[k] ↔ A[last]  ◀ A[k]와 A[last]의 값을 교환
```

또는

```
selectionSort2(A[], n):  ◀ 배열 A[0...n-1]을 정렬한다
    for last ← n-1 downto 1
        k ← theLargest(A, last)  ◀ A[0...last] 중 가장 큰 수 A[k] 찾기
        ◀ A[k] ↔ A[last]
        tmp ← A[k]; A[k] ← A[last]; A[last] ← tmp

theLargest(A[], last):  ◀ A[0...last]에서 가장 큰 수의 인덱스를 리턴한다
    largest ← 0
    for i ← 1 to last
        if (A[i] > A[largest])
            largest ← i
    return largest
```

```

selectionSort2(A[], n):  ◀ 배열 A[0...n-1]을 정렬한다
    for last ← n-1 downto 1 ①
        k ← theLargest(A, last)  ◀ A[0...last] 중 가장 큰 수 A[k] 찾기
        tmp ← A[k]; A[k] ← A[last]; A[last] ← tmp  ◀ A[k] ↔ A[last]

theLargest(A[], last):  ◀ A[0...last]에서 가장 큰 수의 인덱스를 리턴한다
    largest ← 0
    for i ← 1 to last
        if (A[i] > A[largest]) ②
            largest ← i
    return largest

```

✓ Running time: 두 함수의 **for** loop의 iteration 수의 합이 좌우
 — theLargest()가 n-1 번 call 되고,
 call 될 때마다 theLargest()의 수행시간은 한 단계씩 가벼워진다.

✓ 결국 ②의 비교 작업의 총 수가 수행시간을 좌우

✓ $(n - 1) + (n - 2) + \dots + 2 + 1 = \theta(n^2)$ ← Worst case
 ← Average case

정렬할 배열이 주어짐

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

가장 큰 수를 찾는다 (73)

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

73을 맨 오른쪽 수(15)와 자리 바꾼다

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

①의 첫번째 loop

맨 오른쪽 수를 제외한 나머지에서 가장 큰 수를 찾는다 (65)

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

65를 맨 오른쪽 수(11)와 자리 바꾼다

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

①의 두번째 loop

맨 오른쪽 두 수를 제외한 나머지에서 가장 큰 수를 찾는다 (48)

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

...

8을 맨 오른쪽 수(3)와 자리 바꾼다

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

최종 배열

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

2. 버블 정렬 Bubble Sort

단위 순환

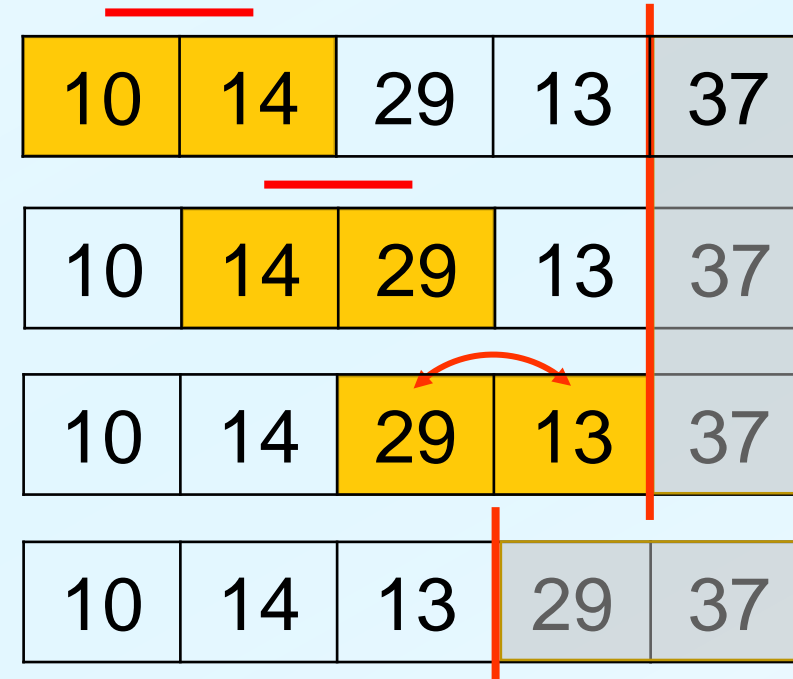
맨 앞부터 한 칸씩 이동하면서,

두 원소가 순서대로 되어 있지 않으면 자리 바꾼다

맨 오른쪽 자리를 관심 대상에서 제외한다

원소가 1 개 남을 때까지 위의 순환을 반복

Animation



알고리즘 bubbleSort()

```
bubbleSort(A[], n): ◀ A[0...n-1]을 정렬한다
    for last ← n-1 downto 1 ❶
        for i ← 0 to last-1 ❷
            if (A[i] > A[i+1]) ❸
                A[i] ↔ A[i+1]
```

기본적으로 selection sort와 비슷한 착상.

다만, 가장 큰 수를 맨 오른쪽으로 보내는 방법이 다를 뿐

✓ Running time: 두 for loop의 iteration 수의 합이 좌우
— ❶의 for loop이 call 될 때마다 ❷의 for loop은 한 바퀴씩 덜 돈다

✓ 결국 ❸의 비교 작업의 총 수가 수행시간을 좌우

✓ $(n-1) + (n-2) + \dots + 2 + 1 = \theta(n^2)$ ← Worst case
← Average case

정렬할 배열이 주어짐

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

왼쪽부터 시작해 이웃한 쌍들을 비교해간다

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

순서대로 되어 있지 않으면 자리 바꾼다

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

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

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

...

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

맨 오른쪽 수(73)를 대상에서 제외한다

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

왼쪽부터 시작해 이웃한 쌍들을 비교해간다

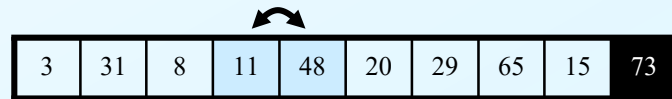


3	31	48	8	11	20	29	65	15	73
---	----	----	---	----	----	----	----	----	----

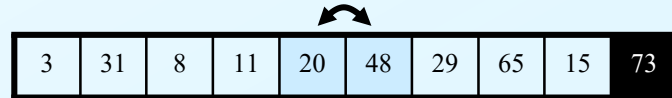
순서대로 되어 있지 않은 경우에는 자리 바꾼다



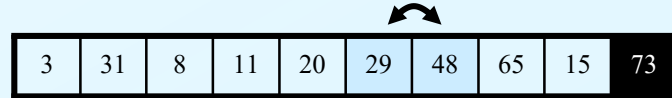
3	31	8	48	11	20	29	65	15	73
---	----	---	----	----	----	----	----	----	----



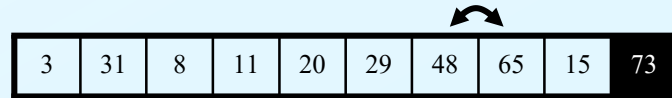
3	31	8	11	48	20	29	65	15	73
---	----	---	----	----	----	----	----	----	----



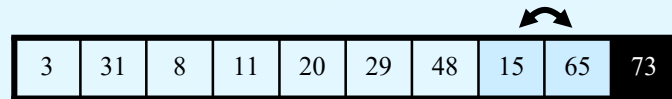
3	31	8	11	20	48	29	65	15	73
---	----	---	----	----	----	----	----	----	----



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



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



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

맨 오른쪽 수(65)를 대상에서 제외한다

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

앞의 작업을 반복하면서 계속 제외해 나간다

...

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

두개짜리 배열의 처리를 끝으로 정렬이 완료된다

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

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

3. 삽입 정렬 Insertion Sort

정렬할 배열이 주어짐

3	31	48	73	8	33	11	15	20	65	29	28	65	25	4
---	----	----	----	---	----	----	----	----	----	----	----	----	----	---

삽입정렬

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	8	31	48	73	33	11	4	20	65	29	28	65	25	15
---	---	----	----	----	----	----	---	----	----	----	----	----	----	----

3	8	31	33	48	73	11	4	20	65	29	28	65	25	15
---	---	----	----	----	----	----	---	----	----	----	----	----	----	----

3	8	11	31	33	48	73	4	20	65	29	28	65	25	15
---	---	----	----	----	----	----	---	----	----	----	----	----	----	----

3	4	8	11	31	33	48	73	20	65	29	28	65	25	15
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----



알고리즘 insertionSort()

```
insertionSort(A[], n): ◀ A[0..n-1]을 정렬한다
    for i ← 1 to n-1 ❶
        A[0...i]의 적합한 자리에 A[i]를 삽입한다 ❷
```

Selection sort와 bubble sort는

정렬되지 않은 배열의 크기가 하나씩 작아지는 알고리즘

Insertion sort는

정렬된 배열의 크기가 하나씩 커지는 알고리즘

✓ Running time:

❶의 for loop이 call 될 때마다 ❷의 삽입 부담은 ~1만큼 늘어난다

✓ 결국 ❷의 삽입에 소요되는 max(비교횟수, shift 횟수)가
수행시간을 좌우

✓ $1 + 2 + \dots + (n - 1) = \theta(n^2)$ ← Worst case

✓ $\frac{1}{2}(1 + 2 + \dots + (n - 1)) = \theta(n^2)$ ← Average case

✓ $\underbrace{1 + 1 + \dots + 1}_{n-1} = \theta(n)$ ← Best case

좀 더 기호적으로 기술한 insertionSort()

```
insertionSort(A[], n): ◀ A[0...n-1]를 정렬한다
  for i ← 1 to n-1
    j ← i - 1
    insertionItem ← A[i]
    ◀ 이 지점에서 A[0...i-1]은 이미 정렬되어 있는 상태임
    while (0 ≤ j and insertionItem < A[j])
      A[j+1] ← A[j]
      j--
    A[j+1] ← insertionItem
```

정렬할 배열이 주어짐

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

...

중간의 한 시점 (다섯번째 자리까지는 정렬되었다)

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

연두색 구간에서 11을 알맞은 자리에 삽입한다

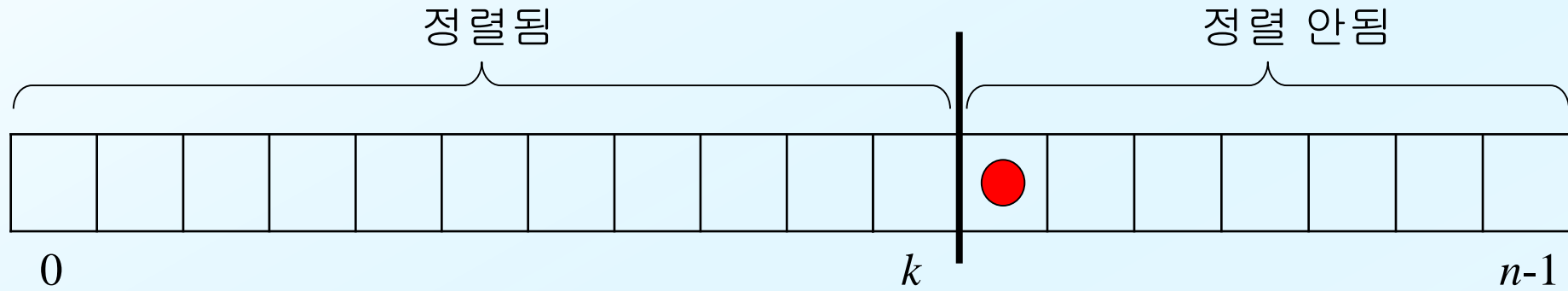
3	8	11	31	48	73	20	29	65	15
---	---	----	----	----	----	----	----	----	----

→
(11보다 큰 수는 모두 한 자리씩 이동)

크기를 하나씩 키워 가면서 계속 진행한다

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

삽입 정렬에서 중간의 한 시점



$i = k$ 로 순환 직후

II. 고급 정렬

4. 병합 정렬

5. 퀵 정렬

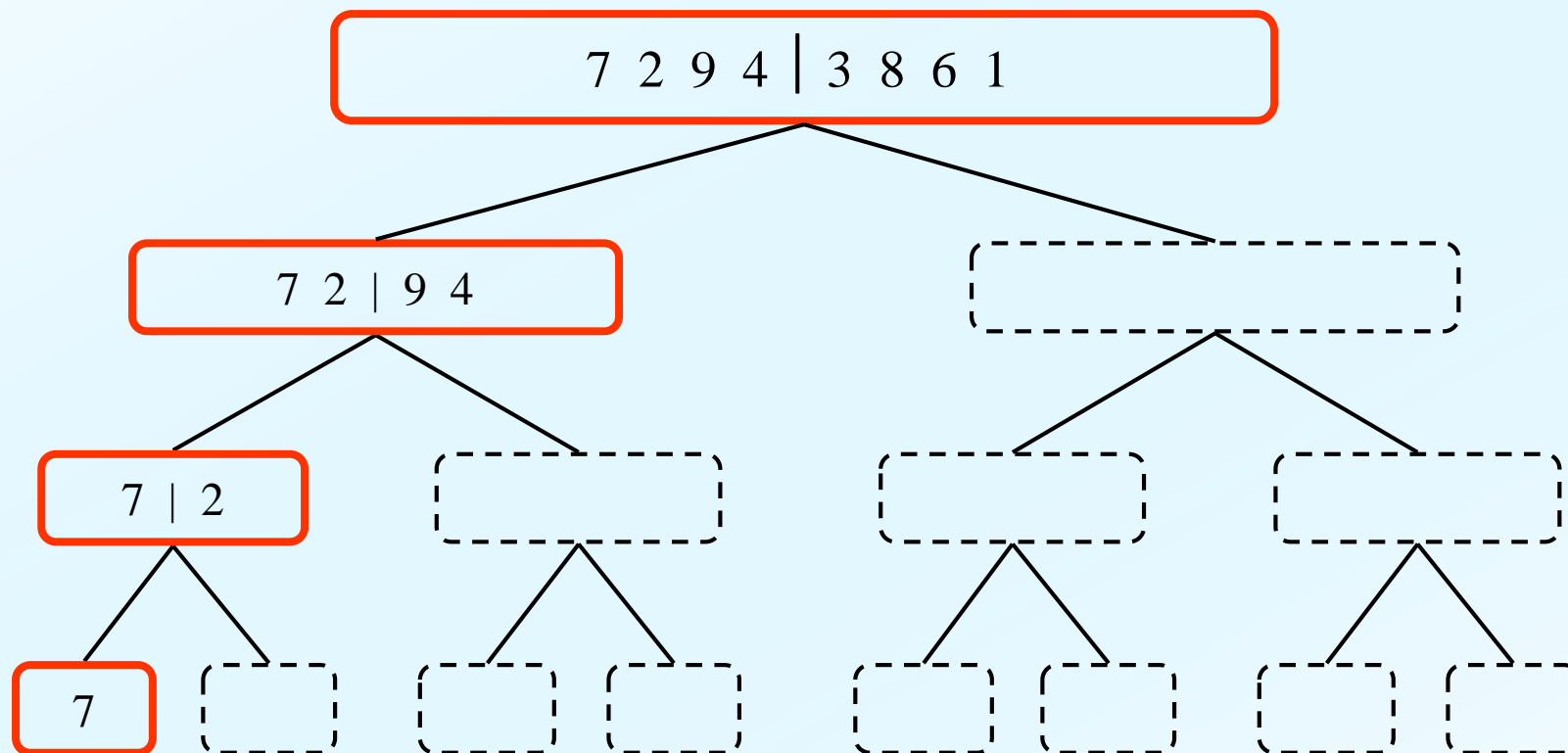
6. 힙 정렬

7. 셀 정렬

4. 병합 정렬 Mergesort

주어진 배열을 이등분하여
각각 (재귀적으로) 정렬한 다음
병합한다

Animation



1 2 3 4 6 7 8 9

✓ Running time: $\Theta(n \log n)$

알고리즘 mergeSort()

mergeSort(A[], p, r): ◀ A[p...r]을 정렬한다

if (p < r)

$q \leftarrow \lfloor (p+r)/2 \rfloor$

◀ p, r의 중간 지점 계산

mergeSort(A, p, q)

◀ 전반부 정렬

mergeSort(A, q+1, r)

◀ 후반부 정렬

merge(A, p, q, r)

◀ 병합

merge(A[], p, q, r):

정렬되어 있는 두 배열 A[p...q]와 A[q+1...r]을 합쳐
정렬된 하나의 배열 A[p...r]을 만든다.

좀 더 기호적으로 기술한 merge()

merge(A[], p, q, r):

◀ A[p...q]와 A[q+1...r]를 병합하여 A[p...r]을 정렬된 상태로 만든다

◀ A[p...q]와 A[q+1...r]는 이미 정렬되어 있다

$i \leftarrow p; j \leftarrow q+1; t \leftarrow 0$

while ($i \leq q$ **and** $j \leq r$)

if ($A[i] \leq A[j]$)

$\text{tmp}[t++] \leftarrow A[i++]$

◀ 의미: $\text{tmp}[t] \leftarrow A[i]; t++; i++;$

else

$\text{tmp}[t++] \leftarrow A[j++]$

◀ 의미: $\text{tmp}[t] \leftarrow A[j]; t++; j++;$

while ($i \leq q$)

◀ 왼쪽 부분 배열이 남은 경우

$\text{tmp}[t++] \leftarrow A[i++]$

while ($j \leq r$)

◀ 오른쪽 부분 배열이 남은 경우

$\text{tmp}[t++] \leftarrow A[j++]$

$i \leftarrow p; t \leftarrow 0$

while ($i \leq r$)

◀ 결과를 A[p...r]에 저장

$A[i++] \leftarrow \text{tmp}[t++]$

병합 정렬의 예

정렬할 배열이 주어짐

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

배열을 반반으로 나눈다

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

 — ①

각각 독립적으로 정렬한다

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

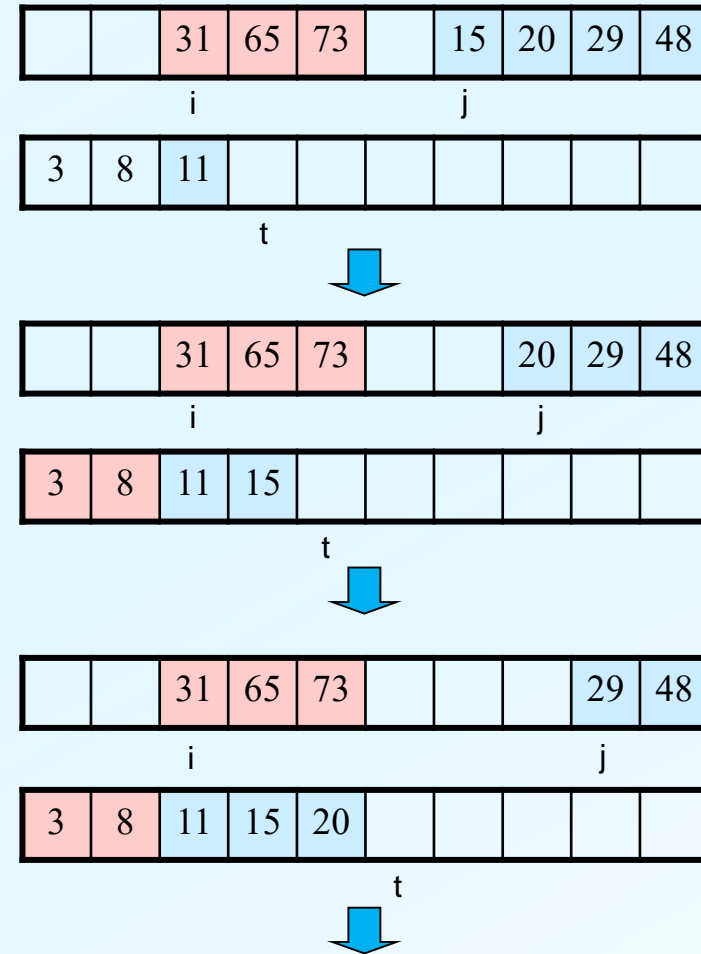
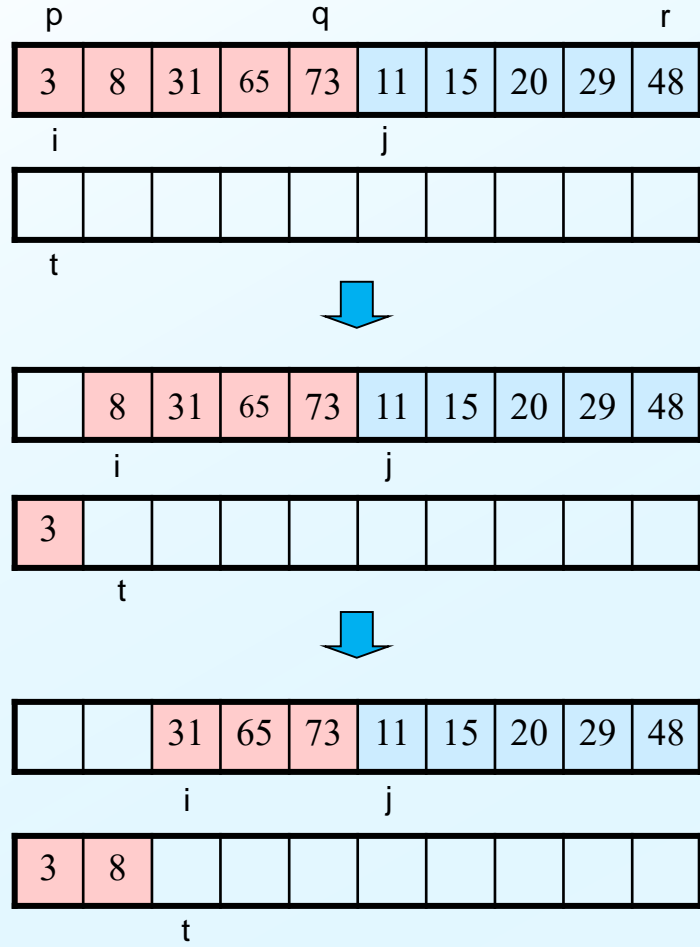
 — ② ③

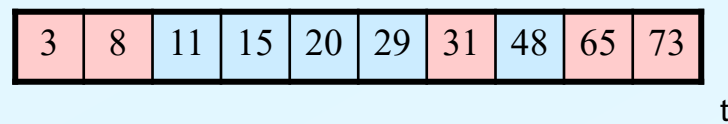
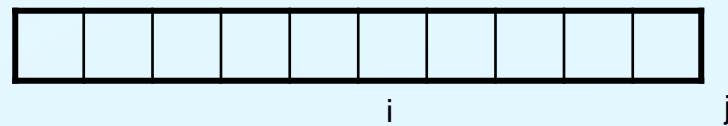
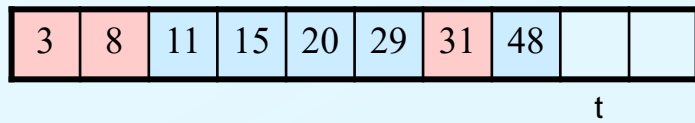
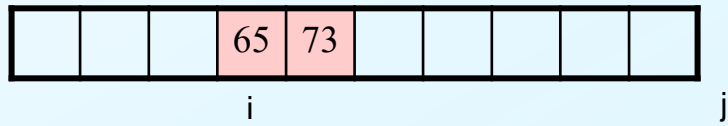
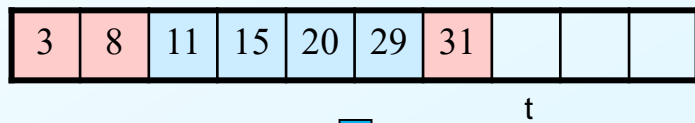
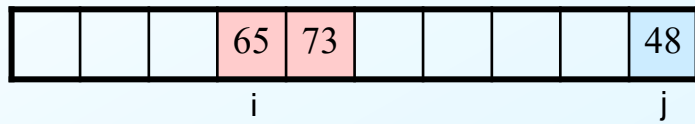
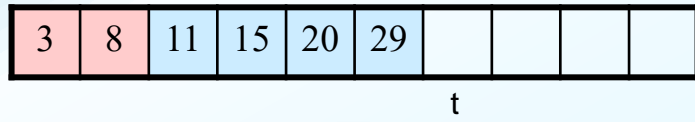
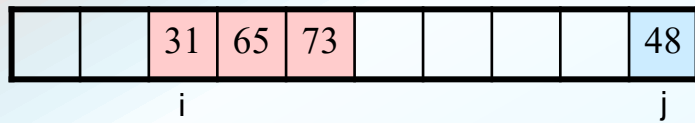
병합한다 (정렬완료)

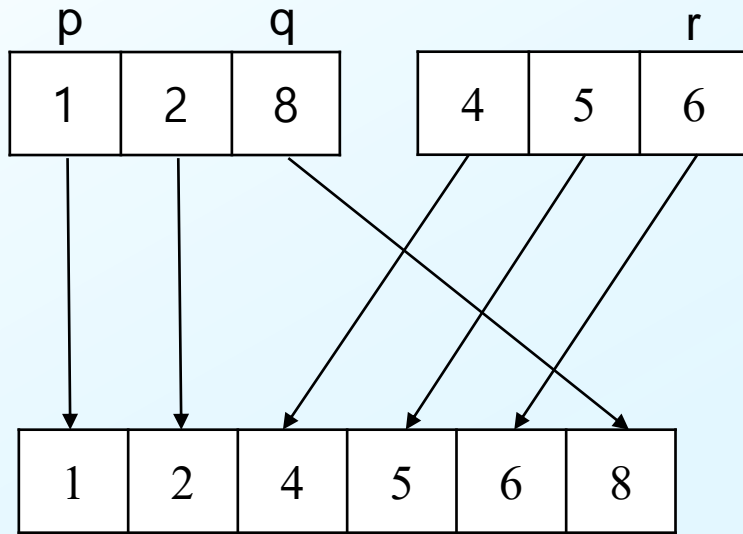
3	8	11	15	20	29	31	48	65	73
---	---	----	----	----	----	----	----	----	----

 — ④

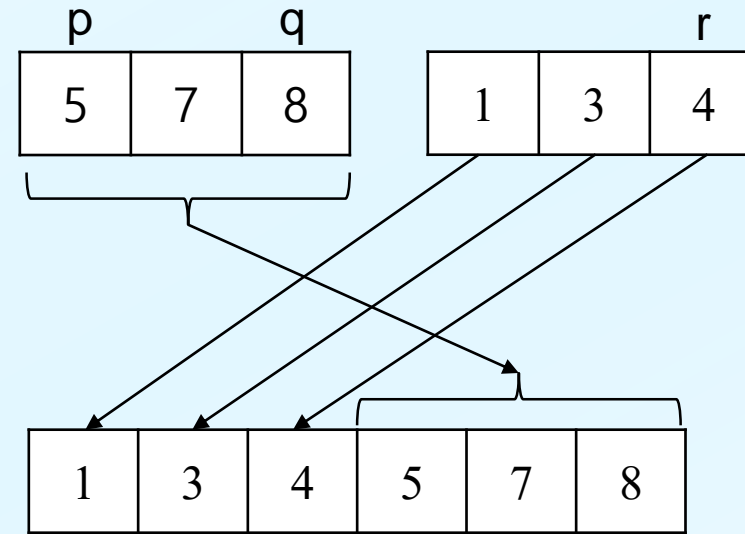
병합의 예







Worst-case merge()



Best-case merge()

병합 정렬의 특성

Mergesort는 항상 $\theta(n \log n)$ 시간이 소요된다

$$\begin{aligned} T(n) &= \Theta(n) + 2T(n/2) \\ &= \Theta(n \log n) \end{aligned}$$

In-place sorting

- 주어진 배열에서 해결되는 정렬

Mergesort는 in-place sorting이 아니다

- merge 과정에서 배열 $A[0 \dots n-1]$ 만큼의 보조 배열 $\text{tmp}[0 \dots n-1]$ 가 필요하다
- 보조 배열 $\text{tmp}[0 \dots n-1]$ 로 저장한 merge 결과를 배열 $A[0 \dots n-1]$ 로 되쓰는 것도 시간 지체 요인

5. 퀵 정렬 Quicksort

정렬할 배열이 주어짐. 기준 원소를 정한다(이 예는 맨 끝 원소).

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

기준보다 작은 수는 기준의 왼쪽에 나머지는
기준의 오른쪽에 오도록 재배치한다

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

기준(31) 왼쪽과 오른쪽을 각각 독립적으로 정렬한다 (정렬완료)

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

알고리즘 quickSort()

quickSort(A[], p, r):

◀ A[p...r]을 정렬한다

if ($p < r$)

$q \leftarrow \text{partition}(A, p, r)$ ◀ 분할

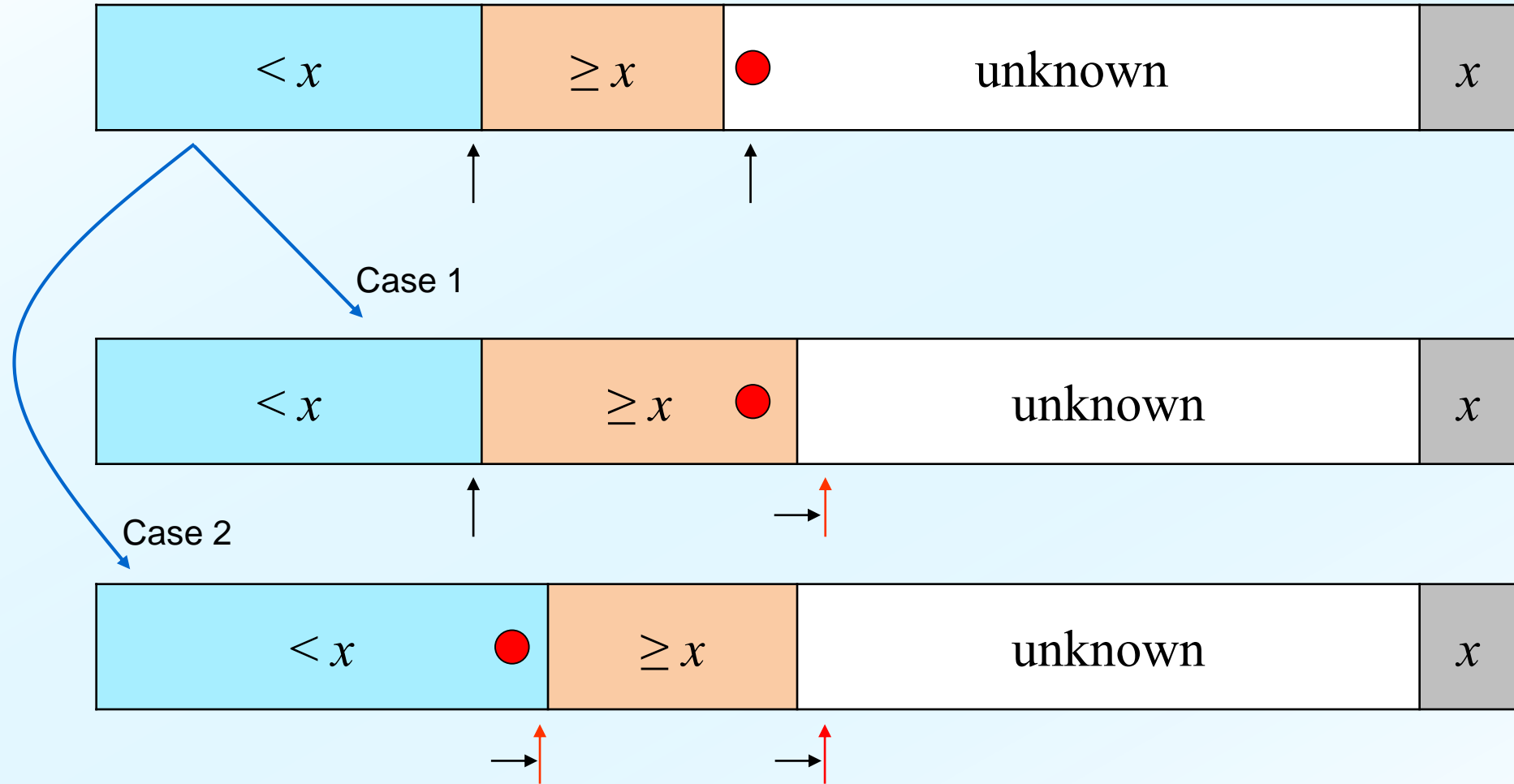
quickSort(A, p, q-1) ◀ 왼쪽 부분 배열 정렬

quickSort(A, q+1, r) ◀ 오른쪽 부분 배열 정렬

partition(A[], p, r):

배열 A[p...r]의 원소들을 기준원소인 A[r]과의 상대적 크기에 따라 양쪽으로 재배치하고, 기준원소가 자리한 위치를 리턴한다

분할의 예. 중간의 한 시점



알고리즘 partition()

partition(A[], p , r):

$x \leftarrow A[r]$ ◀ 기준원소

$i \leftarrow p-1$ ◀ i 는 1구역의 끝 지점

for $j \leftarrow p$ **to** $r-1$ ◀ j 는 3구역의 시작 지점

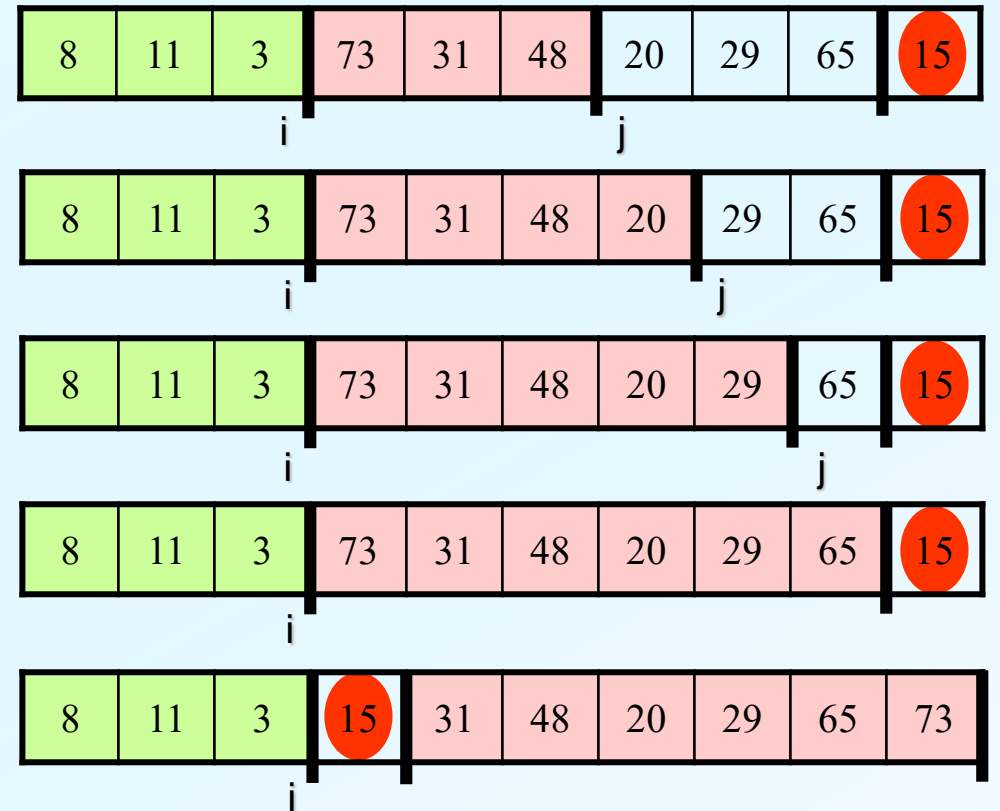
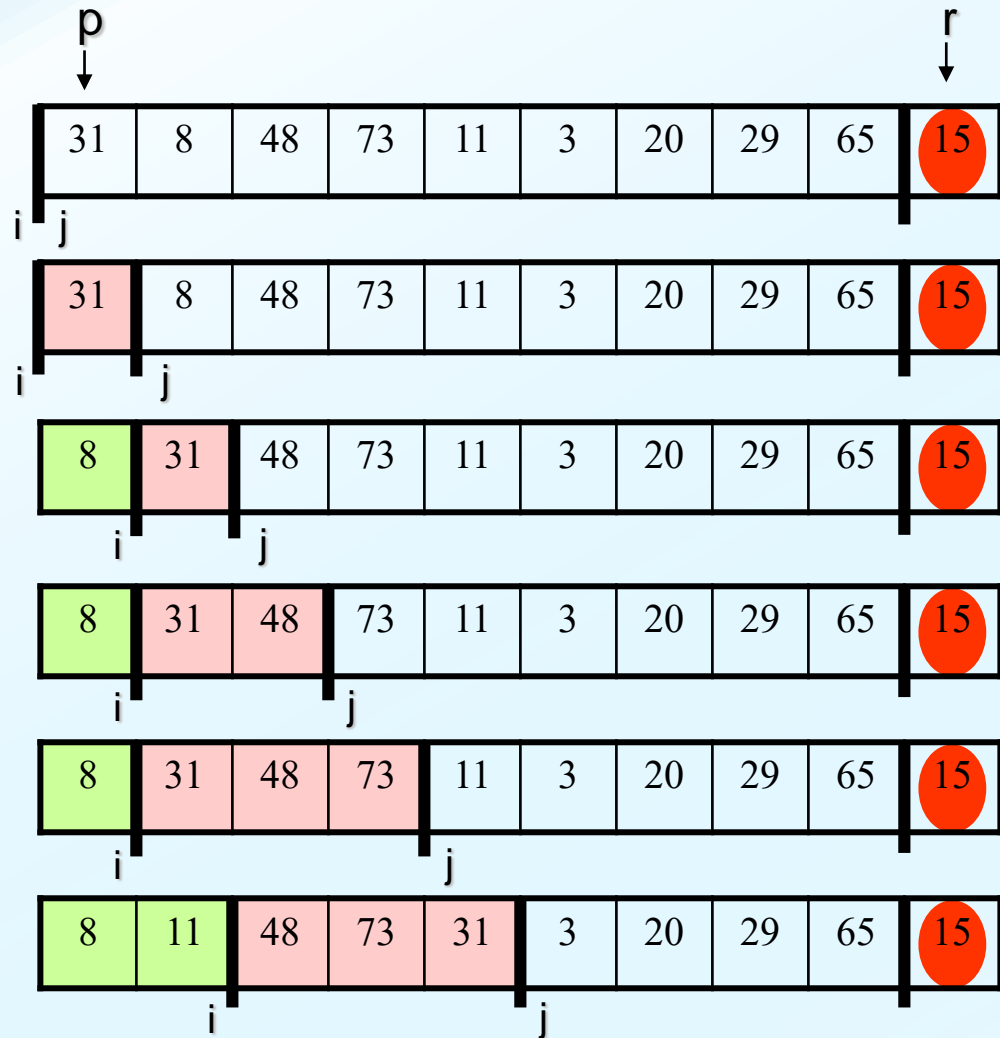
if ($A[j] < x$)

$A[++i] \leftrightarrow A[j]$ ◀ 의미는 i 값 증가 후 교환

$A[i+1] \leftrightarrow A[r]$ ◀ 기준원소와 2구역 첫 원소 교환

return $i+1$

Partition()의 수행 예



퀵 정렬의 특성

Quicksort의 성능은 partition의 균형이 좌우한다

Worst: $0 : n-1$ 또는 $n-1 : 0$

Best: $\sim n/2 : n/2$

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) \\ &= \Theta(n^2) \end{aligned}$$

$$\begin{aligned} T(n) &= \Theta(n) + 2T(n/2) \\ &= \Theta(n \log n) \end{aligned}$$

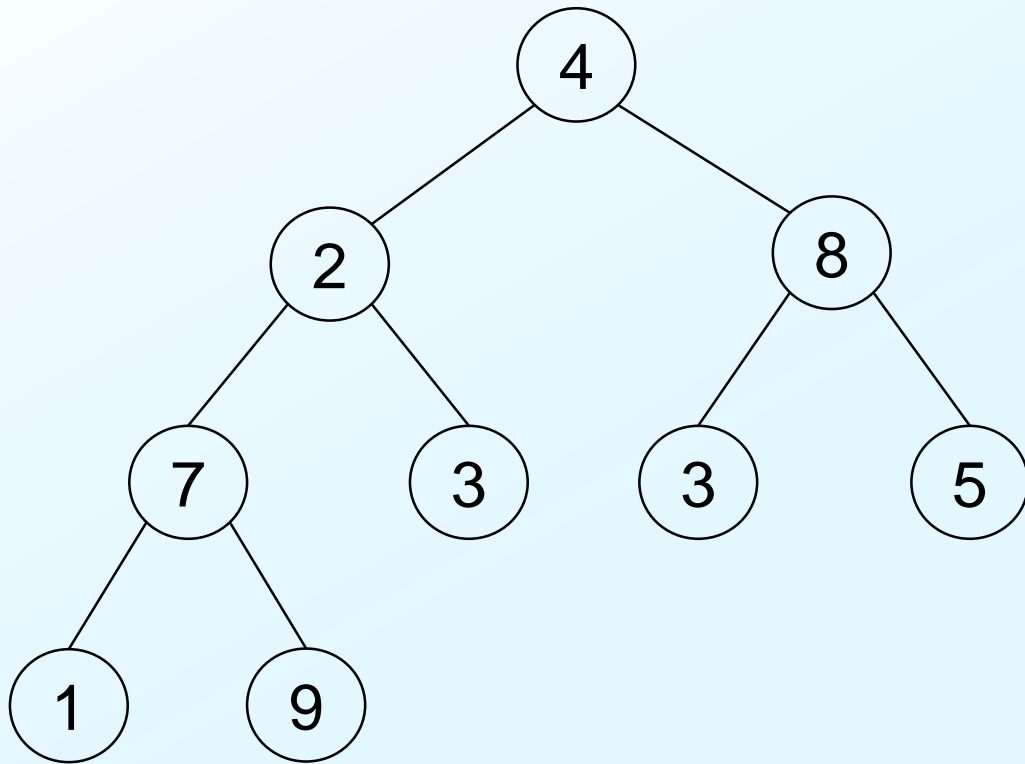
Quicksort은 평균적으로 매우 빠르다

- 필드에서 가장 많이 사용

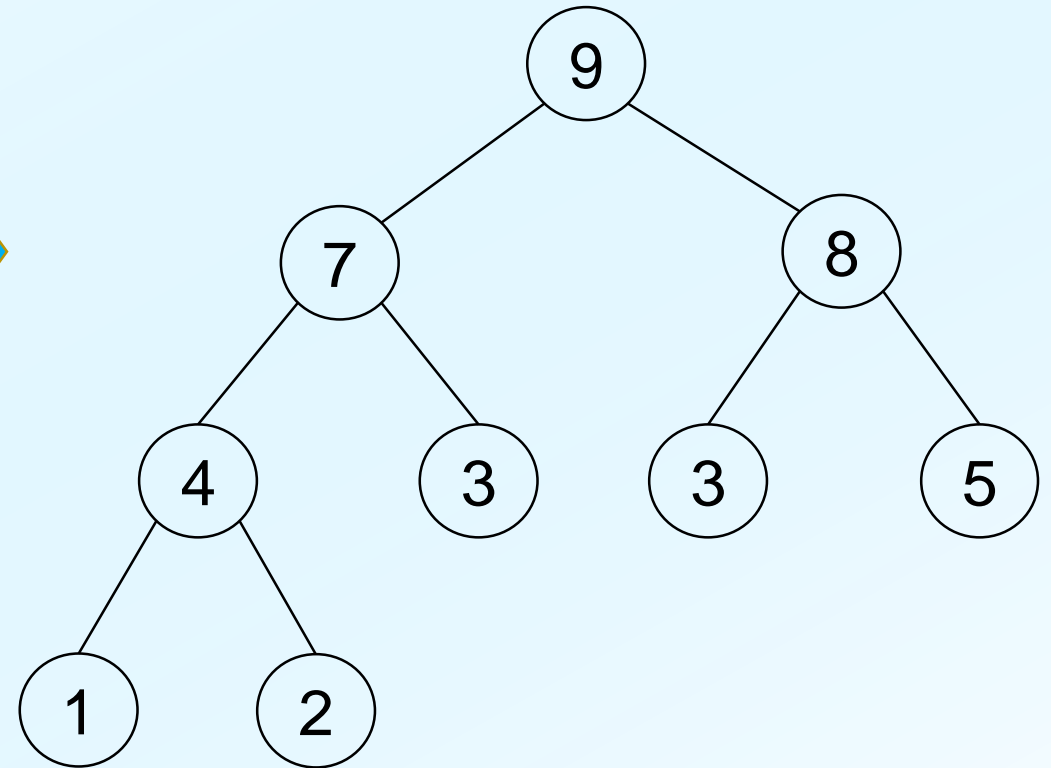
같은 원소가 많으면 그만큼 성능이 떨어진다

- 모든 원소가 동일하면 Selection sort 수준으로 떨어진다
- 같은 원소의 그룹들이 있으면 그룹의 크기가 클수록 성능이 떨어진다

6. 힙 정렬 Heapsort

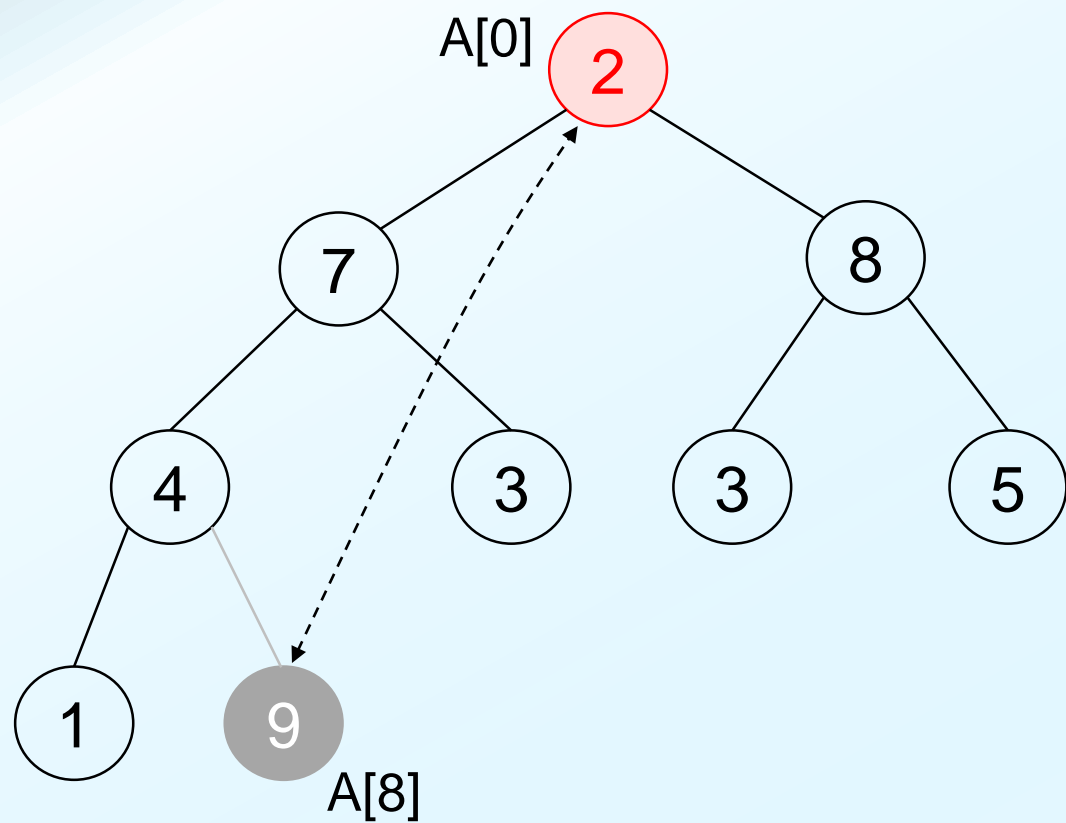


(a)

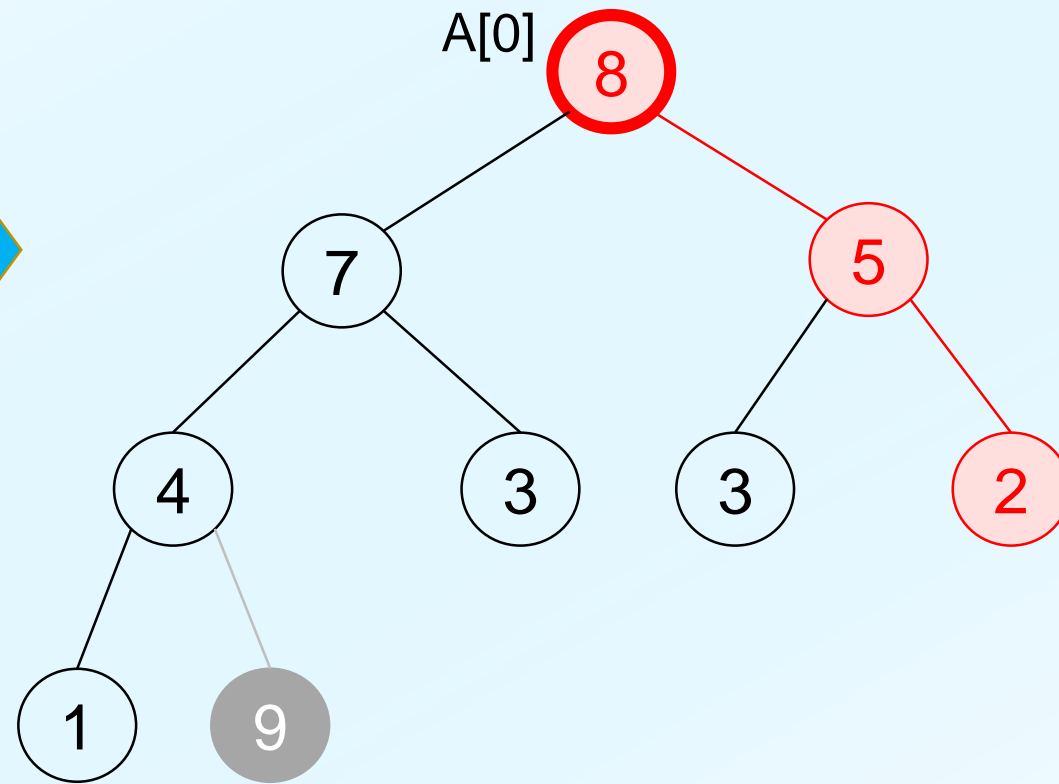


(b)



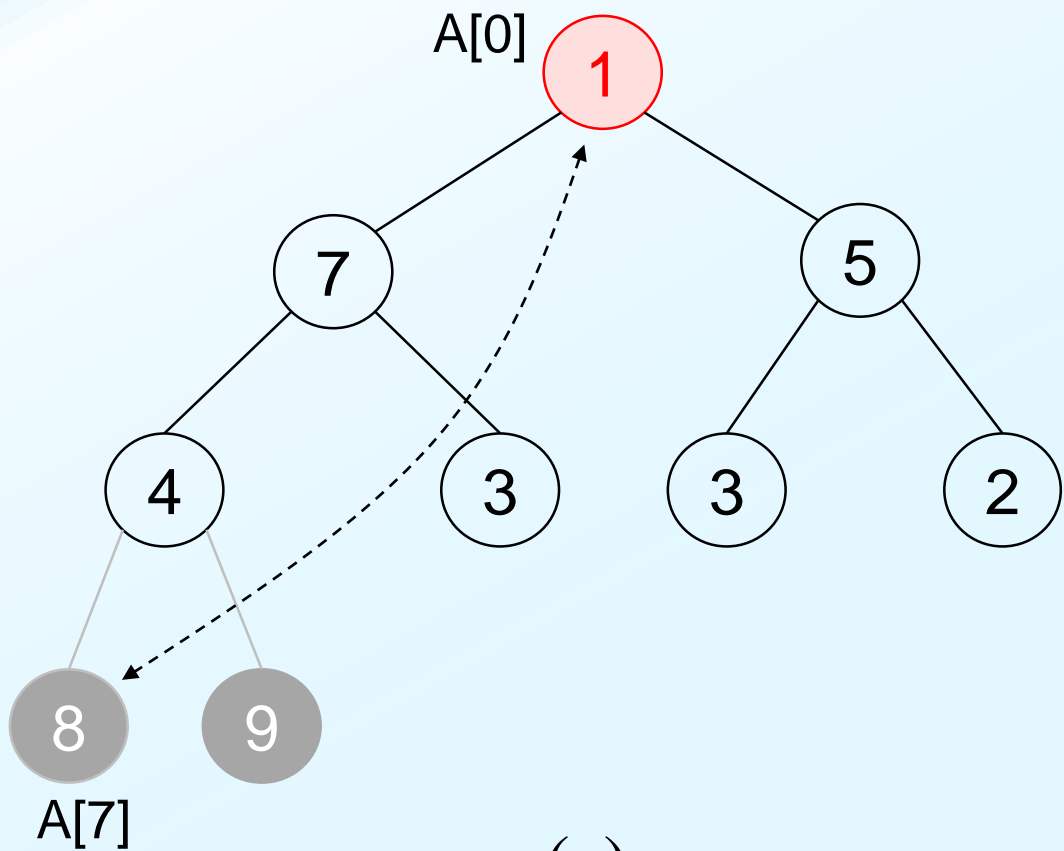


(c)

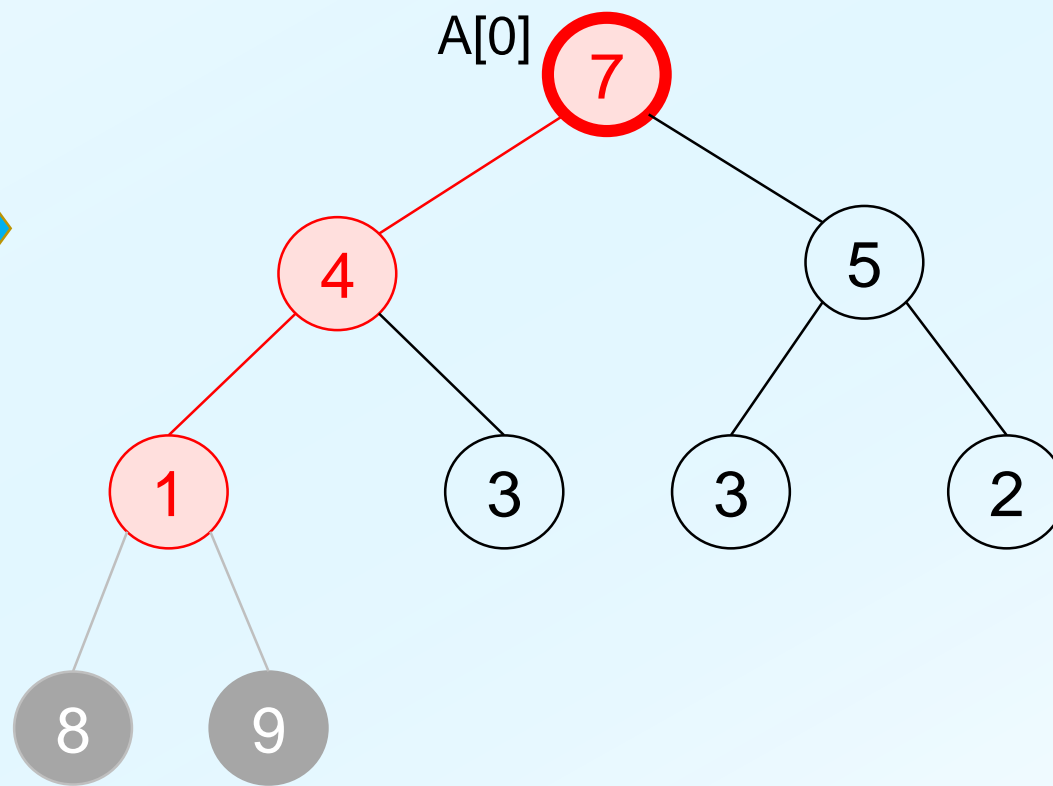


(d)



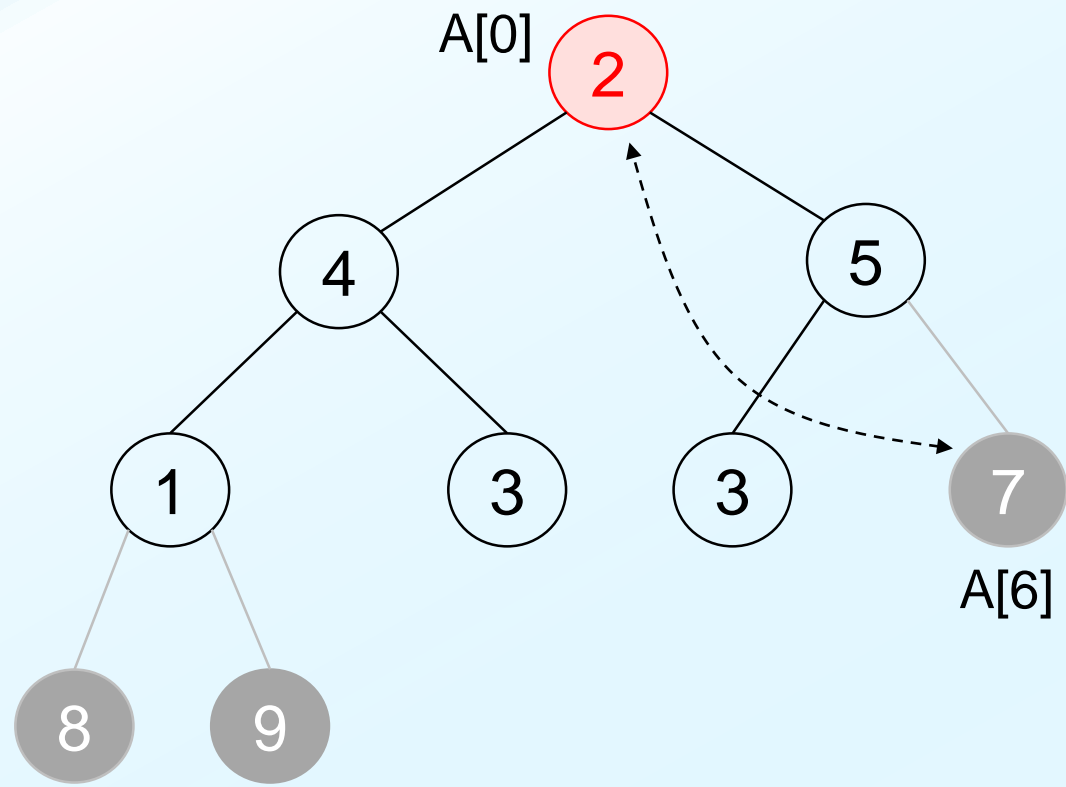


(e)

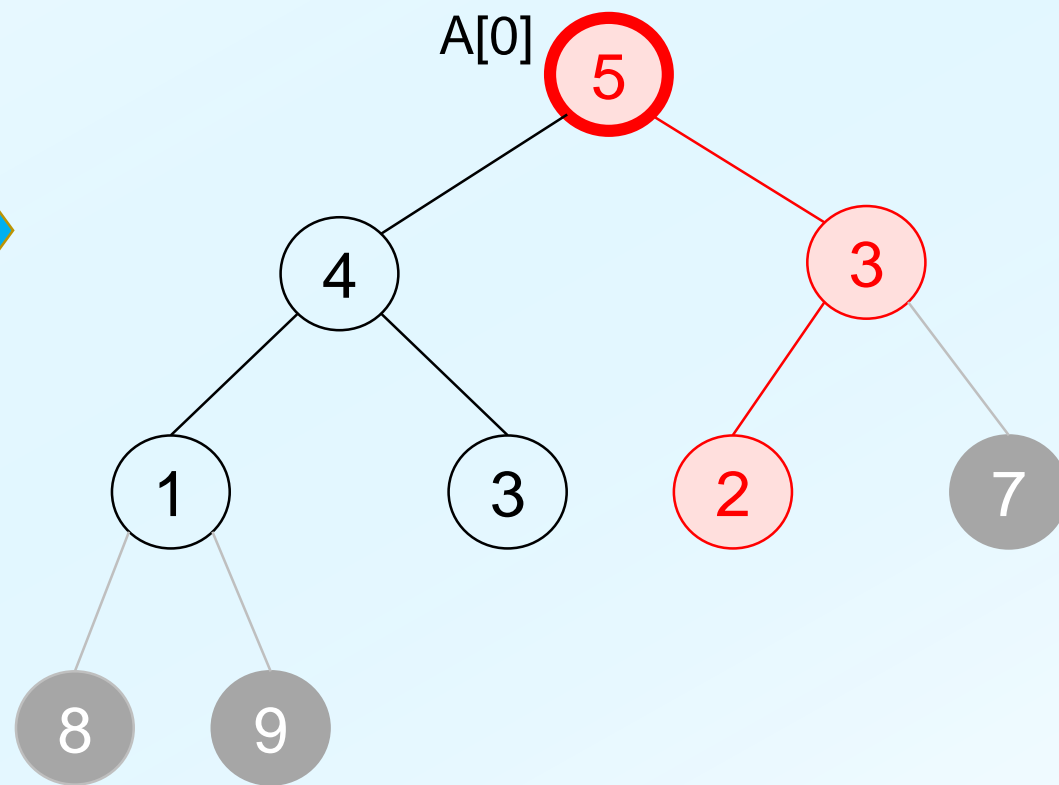


(f)



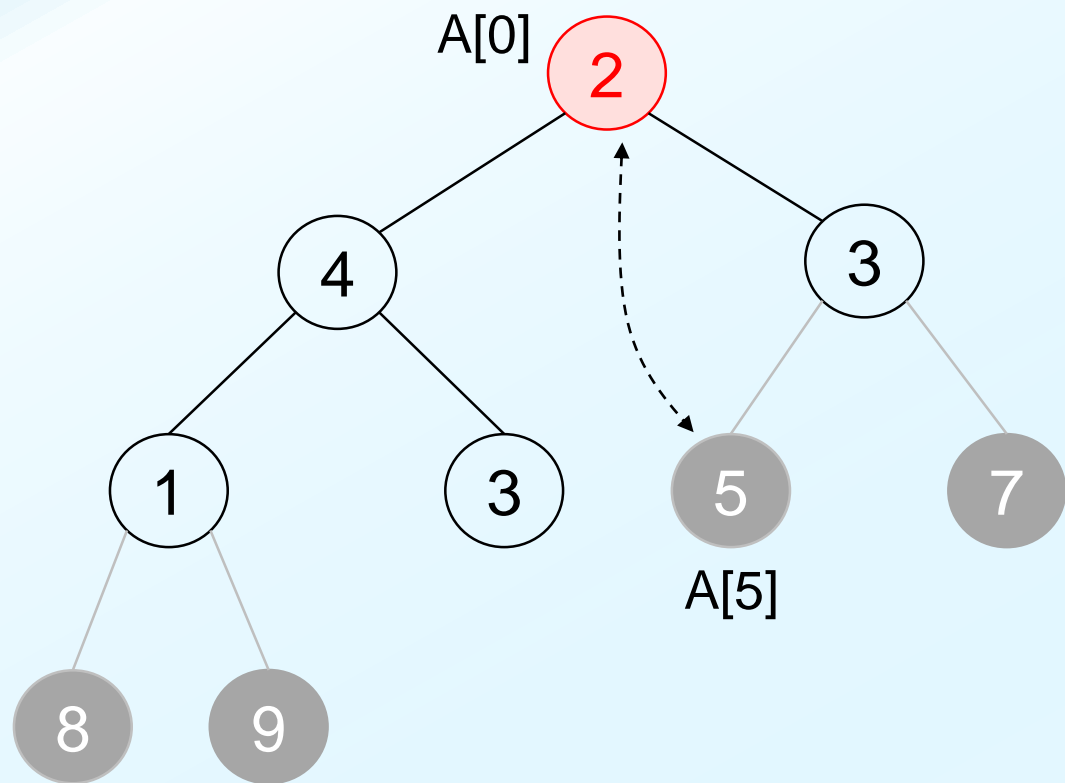


(g)

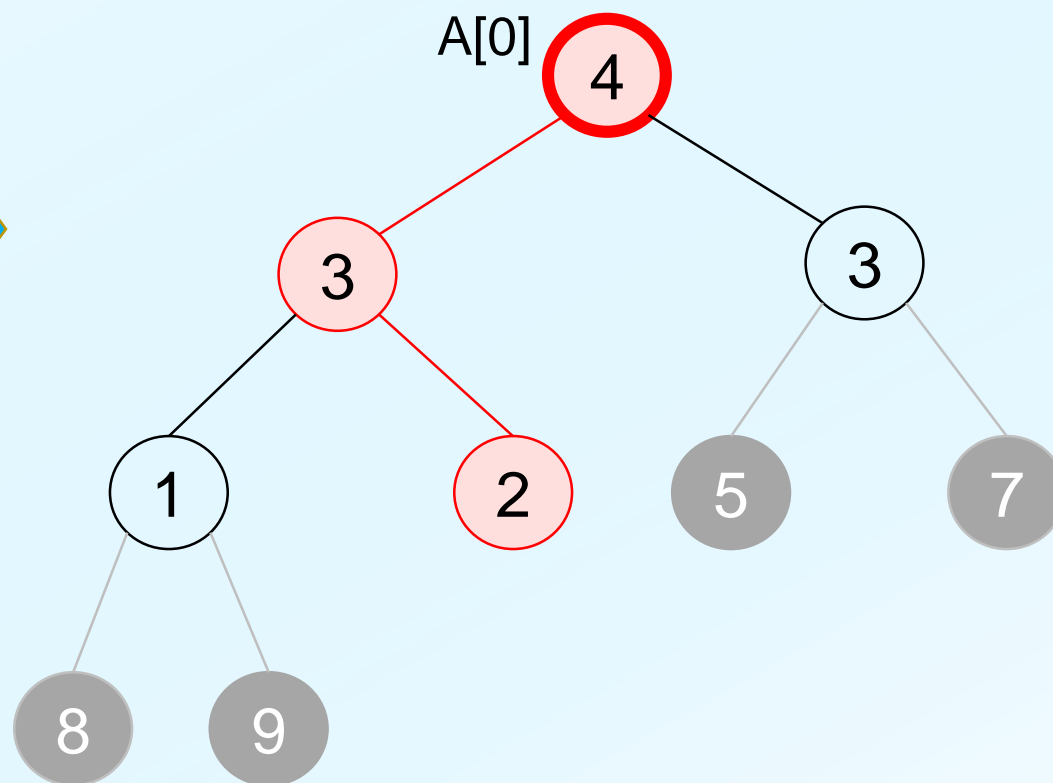


(h)



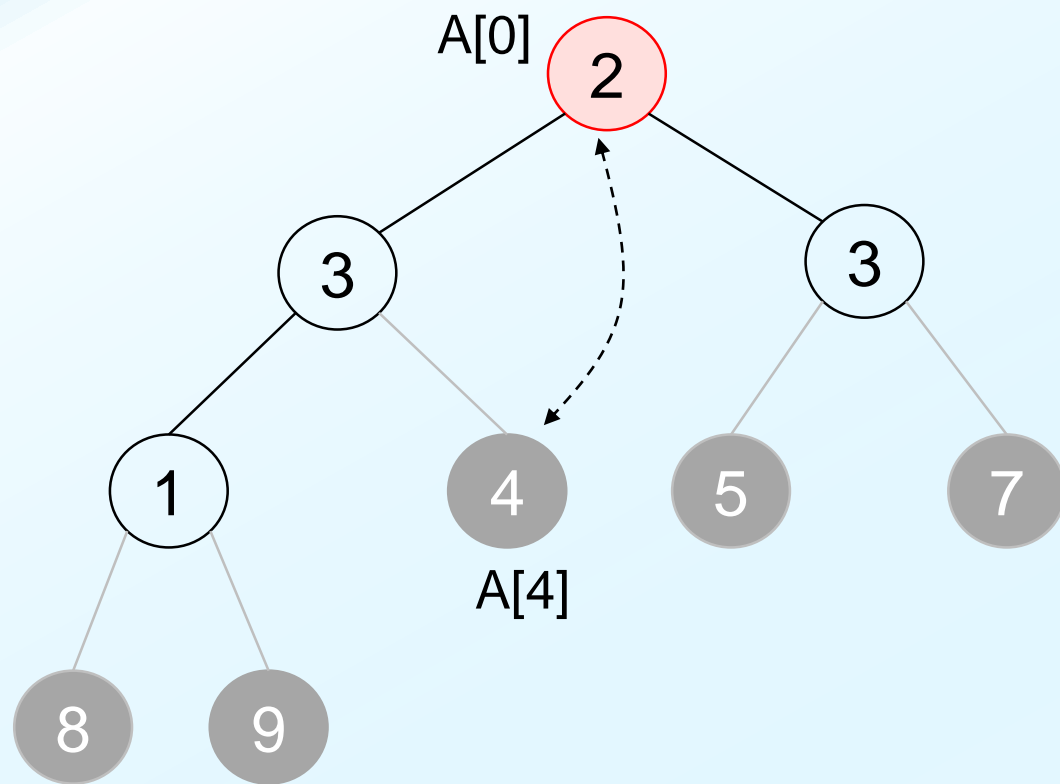


(i)

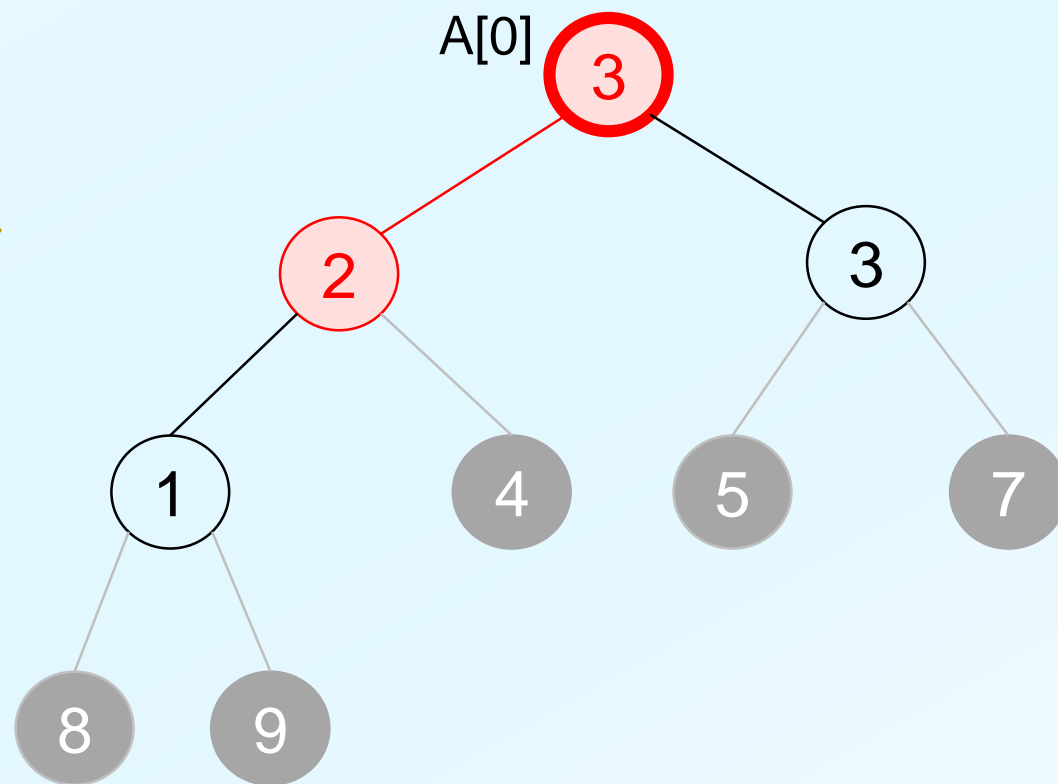


(j)



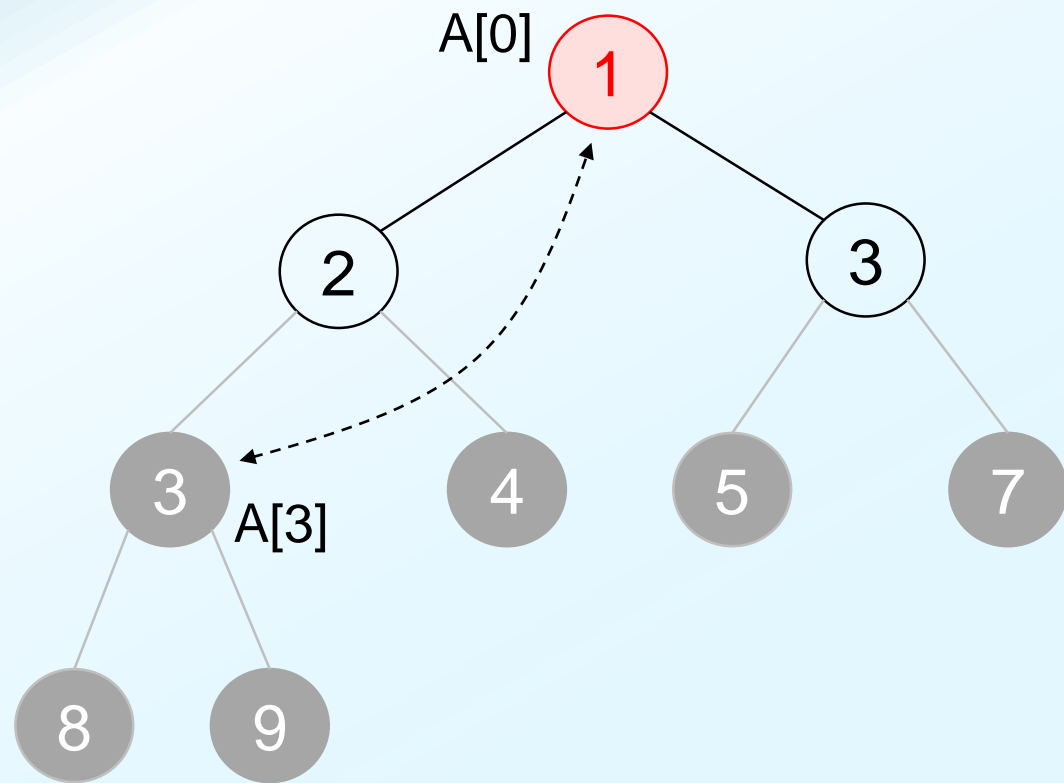


(k)

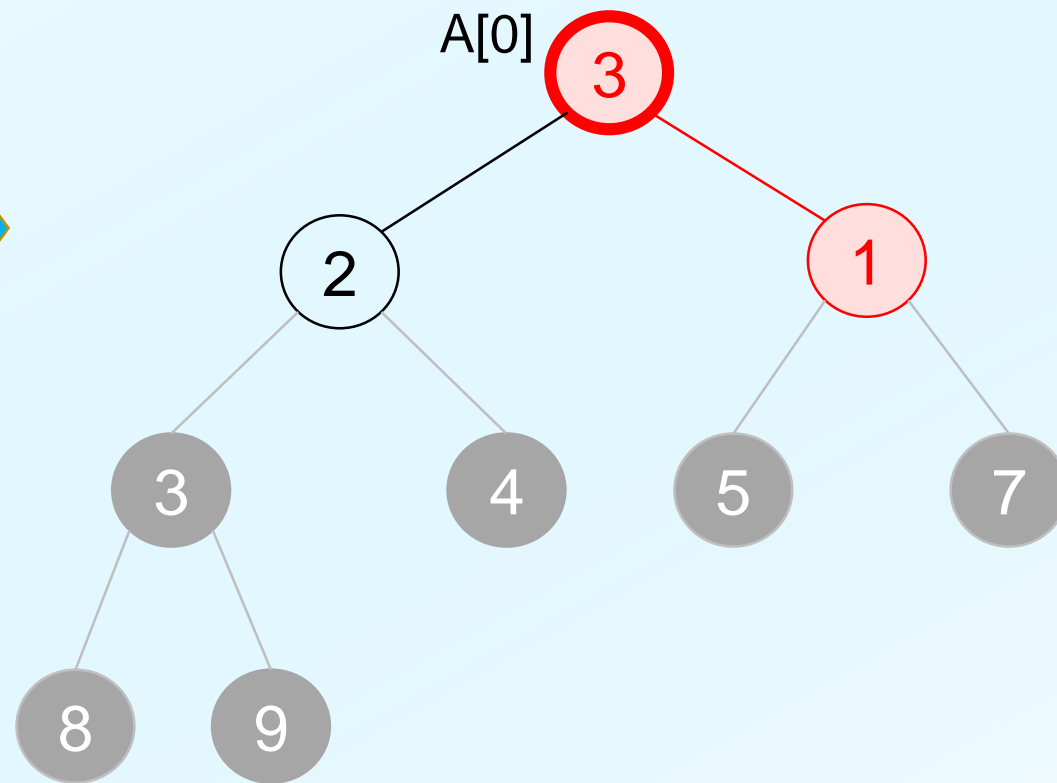


(1)



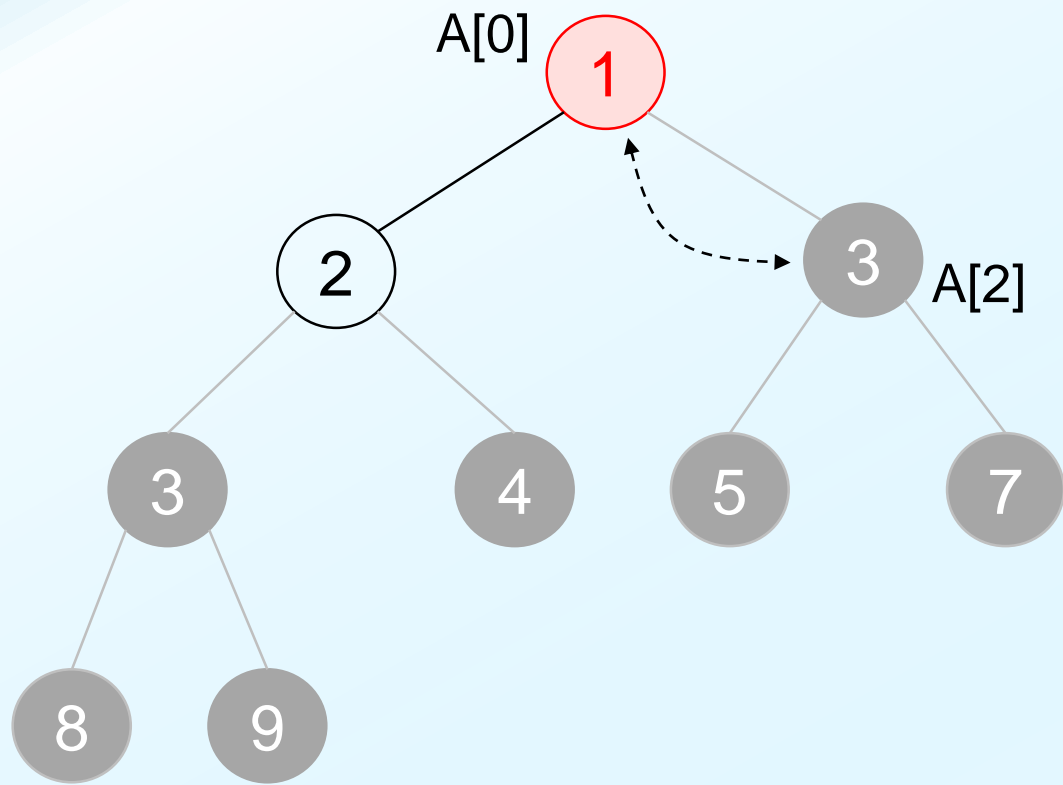


(m)

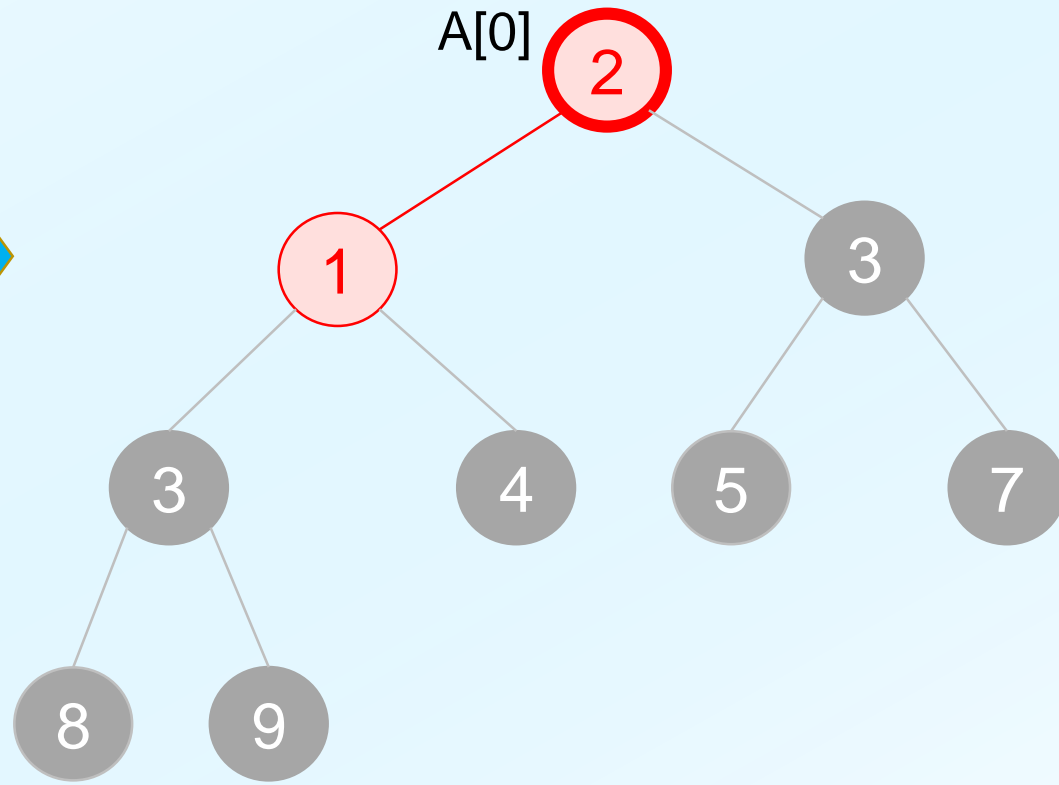


(n)



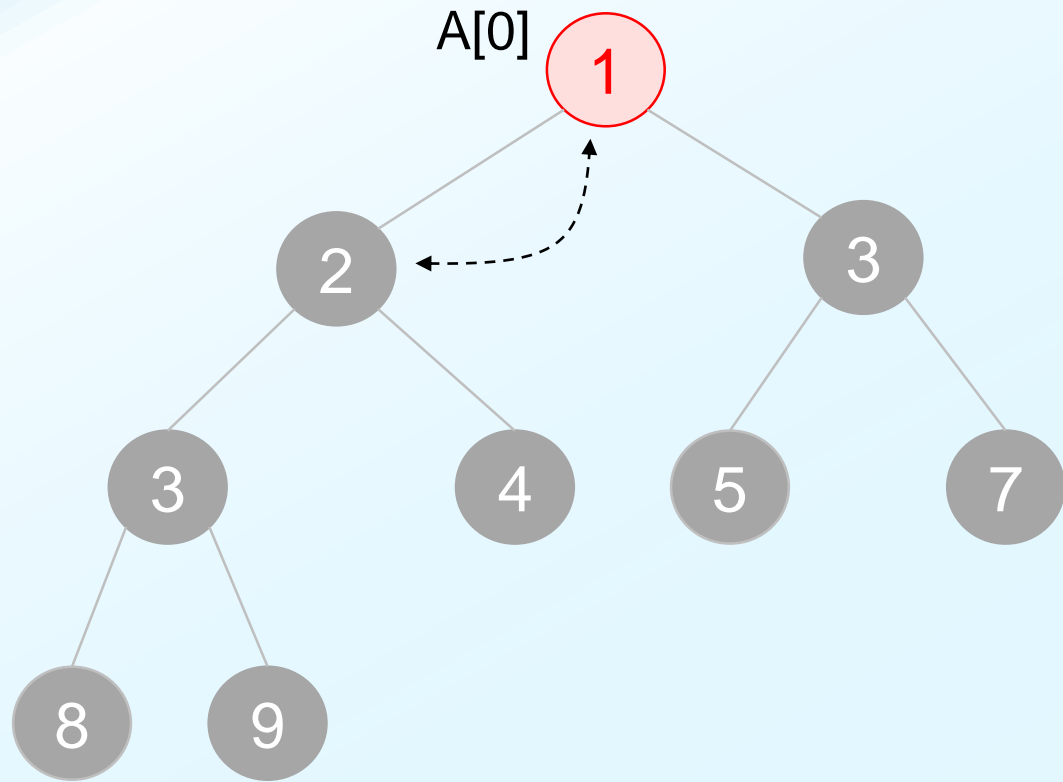


(o)

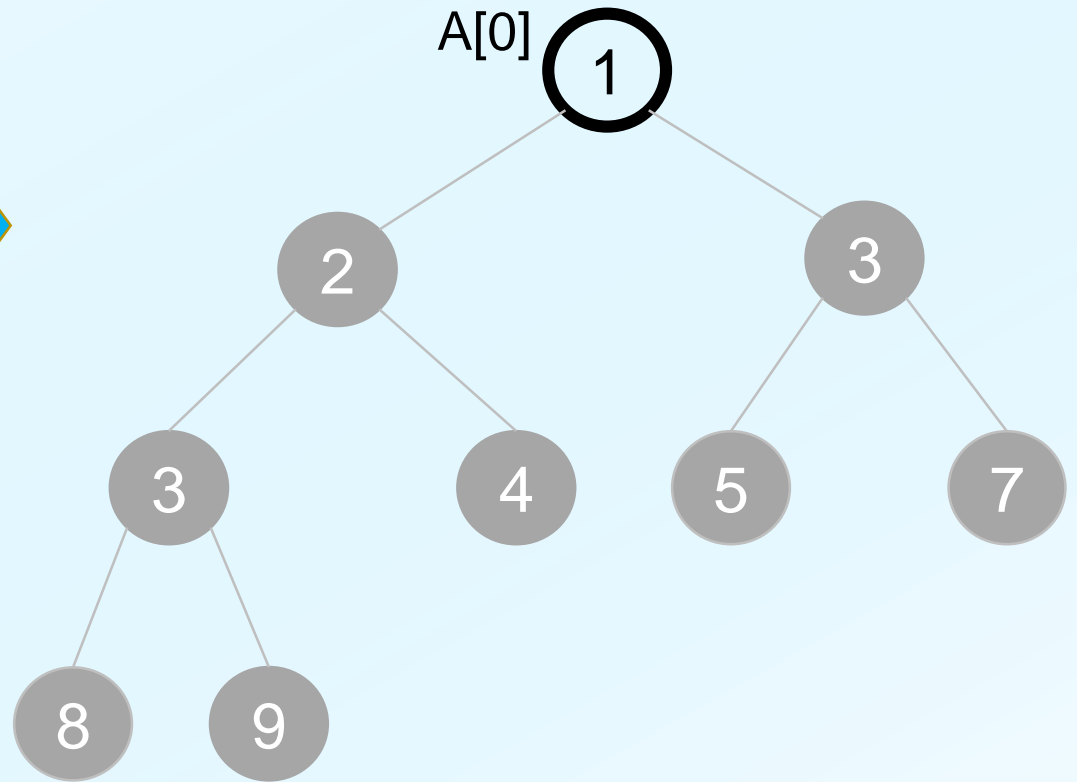


(p)





(q)



(r)

알고리즘 heapsort()

```
heapSort(A[], n):           ◀ 배열 A[0... n-1]을 정렬한다
    buildHeap(A)
    for i ← n-1 downto 1
        A[i] ← deleteMax(A)
```

buildHeap: $\Theta(n)$

for loop: $O(n \log n)$

그냥 말하면: $\Omega(n) \sim O(n \log n)$

Worst: $\Theta(n \log n)$

Best: $\Theta(n)$

7. 셸 정렬 Shell Sort

삽입정렬은 거의 정렬되어 있는 입력에는 탁월한 성능을 보인다

삽입정렬의 성능을 극대화할 수 있게 한 정렬

- 삽입정렬에서 원소의 이동횟수를 극적으로 줄임

통상적인 삽입 정렬

정렬할 배열이 주어짐

3	31	48	73	8	33	11	15	20	65	29	28	65	25	4
---	----	----	----	---	----	----	----	----	----	----	----	----	----	---

일반 삽입정렬

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	8	31	48	73	33	11	4	20	65	29	28	65	25	15
---	---	----	----	----	----	----	---	----	----	----	----	----	----	----

3	8	31	33	48	73	11	4	20	65	29	28	65	25	15
---	---	----	----	----	----	----	---	----	----	----	----	----	----	----

3	8	11	31	33	48	73	4	20	65	29	28	65	25	15
---	---	----	----	----	----	----	---	----	----	----	----	----	----	----

3	4	8	11	31	33	48	73	20	65	29	28	65	25	15
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----



셀 정렬의 예

정렬할 배열이 주어짐

3	31	48	73	8	33	11	15	20	65	29	28	65	25	4
---	----	----	----	---	----	----	----	----	----	----	----	----	----	---

7칸 떨어진 원소들끼리 정렬

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	20	48	73	8	33	11	4	31	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	20	48	73	8	33	11	4	31	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	20	48	29	8	33	11	4	31	65	73	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	20	48	29	8	33	11	4	31	65	73	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	20	48	29	8	33	11	4	20	65	73	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	20	48	29	8	33	11	4	20	65	73	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3칸 떨어진 원소들끼리 정렬

3	20	48	11	8	33	29	4	20	65	73	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

3	4	48	11	8	33	29	20	20	65	25	28	65	73	15
---	---	----	----	---	----	----	----	----	----	----	----	----	----	----

3	4	15	11	8	20	29	20	28	65	25	33	65	73	48
---	---	----	----	---	----	----	----	----	----	----	----	----	----	----

1칸 떨어진 원소들끼리 정렬(전체 배열)

3	4	8	11	15	20	20	25	28	29	33	65	48	65	73
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

전통적인 셀 정렬

7칸 떨어진 원소들끼리 정렬

3	31	48	73	8	33	11	15	20	65	29	28	65	25	4
3	20	48	73	8	33	11	15	31	65	29	28	65	25	4
3	20	48	73	8	33	11	15	31	65	29	28	65	25	4
3	20	48	29	8	33	11	15	31	65	73	28	65	25	4
3	20	48	29	8	33	11	15	31	65	73	28	65	25	4
3	20	48	29	8	33	11	15	20	65	73	28	65	25	4
3	20	48	29	8	33	11	15	20	65	73	28	65	25	4
3	20	48	29	8	33	11	4	20	65	73	28	65	25	15

3칸 떨어진 원소들끼리 정렬

3	20	48	29	8	33	11	4	20	65	73	28	65	25	15
3	8	48	29	20	33	11	4	20	65	73	28	65	25	15
3	8	33	29	20	48	11	4	20	65	73	28	65	25	15
3	8	33	11	20	48	29	4	20	65	73	28	65	25	15
3	4	33	11	8	48	29	20	20	65	73	28	65	25	15
3	4	20	11	8	33	29	20	48	65	73	28	65	25	15
3	4	20	11	8	33	29	20	48	65	73	28	65	25	15
3	4	20	11	8	33	29	20	48	65	73	28	65	25	15

3	4	20	11	8	28	29	20	33	65	73	48	65	25	15
---	---	----	----	---	----	----	----	----	----	----	----	----	----	----

3	4	20	11	8	28	29	20	33	65	73	48	65	25	15
---	---	----	----	---	----	----	----	----	----	----	----	----	----	----

3	4	20	11	8	28	29	20	33	65	25	48	65	73	15
---	---	----	----	---	----	----	----	----	----	----	----	----	----	----

3	4	15	11	8	20	29	20	28	65	25	33	65	73	48
---	---	----	----	---	----	----	----	----	----	----	----	----	----	----

1칸 떨어진 원소들끼리 정렬(전체 배열)

3	4	8	11	15	20	20	25	28	29	33	48	65	65	73
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

알고리즘 shellSort()

shellSort(A[]): ◀ $A[0 \dots n-1]$: 정렬할 배열

for $h \leftarrow h_1, h_2, \dots, 1$ ◀ $h_1, h_2, \dots, 1$: 갭 수열

for $k \leftarrow 0$ **to** $h-1$

 stepInsertionSort(A, k, h)

stepInsertionSort(A[], k, h): ◀ $A[k, k+h, k+2h, \dots]$ 를 정렬한다

for ($i \leftarrow k+h; i \leq n-1; i \leftarrow i+h$)

 newItem $\leftarrow A[i]$

 ◀ 이 지점에서 $A[\dots i-3h, i-2h, i-h]$ 는 이미 정렬되어 있는 상태임

for ($j \leftarrow i-h; 0 \leq j$ **and** newItem $< A[j]; j \leftarrow j-h$)

$A[j+h] \leftarrow A[j]$

$A[j+h] \leftarrow \text{newItem}$

shellSort(A[]): \blacktriangleleft A[0...n-1]: 정렬할 배열

for $h \leftarrow h_1, h_2, \dots, 1$ $\blacktriangleleft h_1, h_2, \dots, 1$: 갭 수열

for $k \leftarrow 0$ **to** $h-1$

stepInsertionSort(A, k, h)

stepInsertionSort(A[], k, h): \blacktriangleleft A[k, k+h, k+2h, ...]를 정렬한다

for ($i \leftarrow k+h; i \leq n-1; i \leftarrow i+h$)

 newItem \leftarrow A[i]

\blacktriangleleft 이 지점에서 A[... i-3h, i-2h, i-h]는 이미 정렬되어 있는 상태임

for ($j \leftarrow i-h; 0 \leq j$ **and** newItem < A[j]; $j \leftarrow j-h$)

 A[j+h] \leftarrow A[j]

 A[j+h] \leftarrow newItem

Given

3	31	48	73	8	33	11	15	20	65	29	28	65	25	4
---	----	----	----	---	----	----	----	----	----	----	----	----	----	---

$k=0, h=7$

3	31	48	73	8	33	11	4	20	65	29	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

$k=1, h=7$

3	20	48	73	8	33	11	15	31	65	29	28	65	25	4
---	----	----	----	---	----	----	----	----	----	----	----	----	----	---

...

$k=6, h=7$

3	20	48	29	8	33	11	4	20	65	73	28	65	25	15
---	----	----	----	---	----	----	---	----	----	----	----	----	----	----

...

삽입 정렬과 셸 정렬 수행 시간 비교

입력 크기 \	1,000 (micro sec)	10,000 (micro sec)	100,000 (milli sec)
Insertion Sort	115	9359	982
Shell Sort	62	647	9

극적으로 개선

셀 정렬에서 준비 과정과 마지막 삽입 정렬 시간 비교

h=1로 수행되는 마지막
stepInsertionSort()를 제외한 나머지 부분

→ 앞쪽 준비 과정의 시간 vs. 맨 마지막 삽입정렬 시간

5 ~ 10 : 1 (배열 크기가 1천)

30 ~ 100 : 1 (배열 크기가 10만 이상)

III. 특수 정렬

8. 계수 정렬

9. 기수 정렬

10. 버킷 정렬

원소들이 특수한 성질을 만족하면 $\Theta(n)$ 정렬도 가능하다

계수 정렬 Counting Sort

- 원소들의 크기가 모두 $-O(n) \sim O(n)$ 정수 범위에 있을 때

기수 정렬 Radix Sort

- 원소들이 모두 k 이하의 자릿수를 가졌을 때 (k : 상수)

버킷 정렬 Bucket Sort

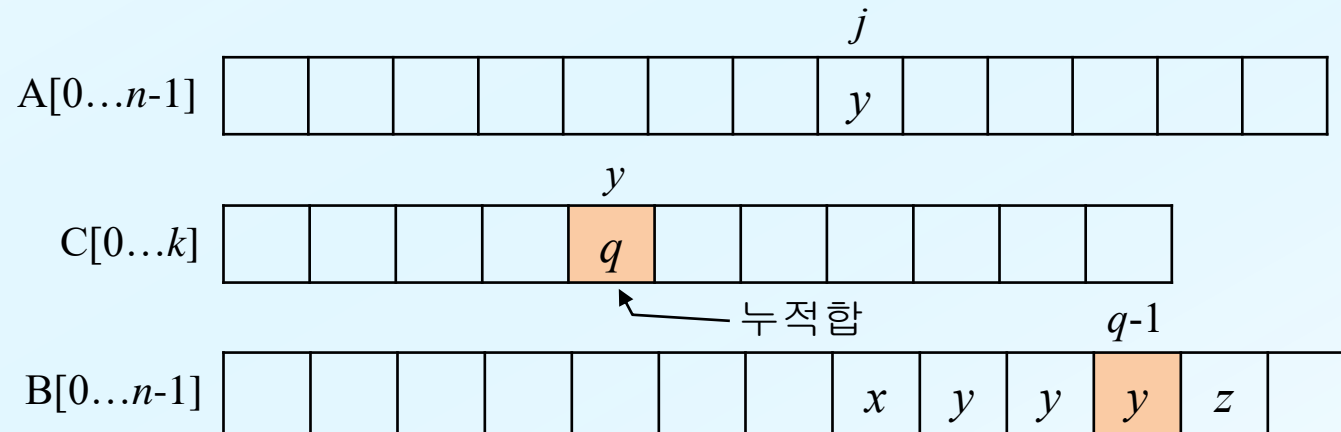
- 원소들이 **균등 분포** Uniform distribution을 이룰 때

8. 계수 정렬 Counting Sort

```
countingSort(A[], B[], n):  
  ◀ A[0...n-1]: 입력 배열 A[i] ∈ {0, 1, 2, ..., k}  
  ◀ B[1...n]: 배열 A[0...n-1]를 정렬한 결과  
    for i ← 0 to k  
      C[i] ← 0  
    for j ← 0 to n-1  
      C[A[j]]++  
    ◀ 이 시점에서의 C[i]: 값이 i인 원소의 총 수  
    for i ← 1 to k  
      C[i] ← C[i] + C[i-1]  ◀ 누적합 계산  
    ◀ 이 시점에서의 C[i]: i보다 작거나 같은 원소의 총 개수  
    for j ← n-1 downto 0  
      B[C[A[j]]-1] ← A[j]  
      C[A[j]]--
```

$k = O(n)$

Running time: $\Theta(n)$



9. 기수 정렬 Radix Sort

모든 원소가 상수 k 이하의 자릿수를 가진 자연수인 특수한 경우
자연수가 아닌 제한된 길이를 가진 알파벳 등도 해당

radixSort(A[], n , k):

- ◀ 원소들이 최대 k -자릿수인 A[0... n -1]을 정렬
- ◀ 가장 낮은 자릿수를 1번째 자릿수라 함

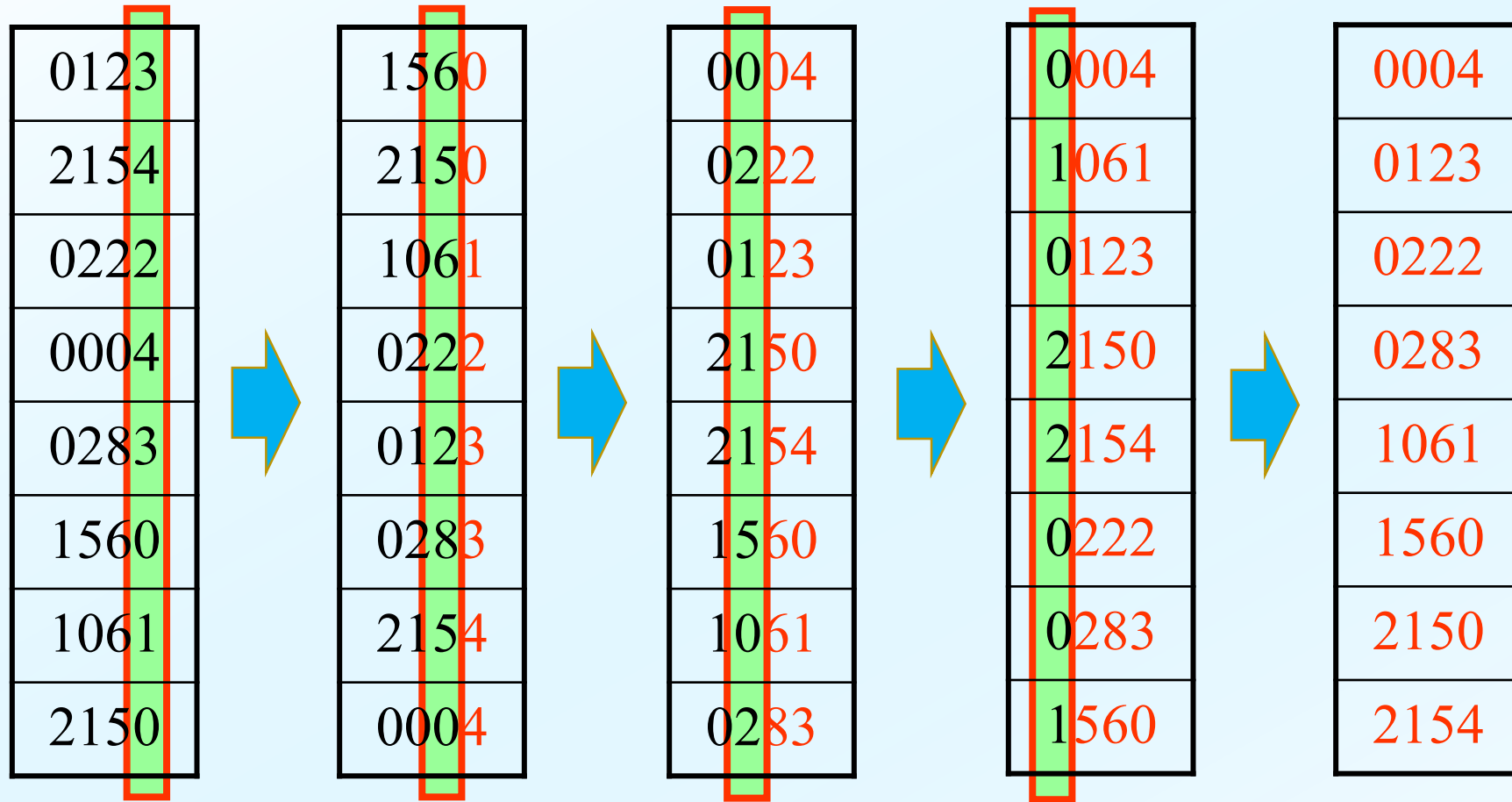
for $i \leftarrow 1$ **to** k

i 번째 자릿수에 대해 A[0... n -1]을 안정성을 유지하면서 정렬한다

✓ 안정성을 유지하는 정렬 Stable sort

- 같은 값을 가진 원소들은 정렬 후에도 원래의 순서가 유지되는 성질을 가진 정렬

기수 정렬의 예



✓ Running time: $\Theta(kn) = \Theta(n) \leftarrow k: \text{a constant}$

10. 버킷 정렬 Bucket Sort

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

bucketSort(A[], n):

◀ $A[0...n-1]$: $[0, 1)$ 범위의 균등 분포를 한 실숫값 리스트

for $i \leftarrow 0$ **to** $n-1$

$A[i]$ 를 리스트 $B[nA[i]]$ 에 삽입한다 ▶ $B[0...n-1]$ 각각이 리스트

for $i \leftarrow 0$ **to** $n-1$

리스트 $B[i]$ 에 있는 원소들을 정렬

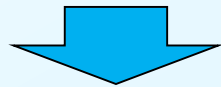
$B[0], B[1], \dots, B[n-1]$ 의 원소들을 차례대로 $A[0...n-1]$ 로 복사한다

Running time: $\Theta(n)$

버킷 정렬의 예

(a) $A[0..14]$: 정렬할 배열

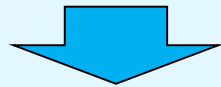
.38	.94	.48	.73	.99	.43	.55	.15	.85	.84	.81	.71	.17	.10	.02
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

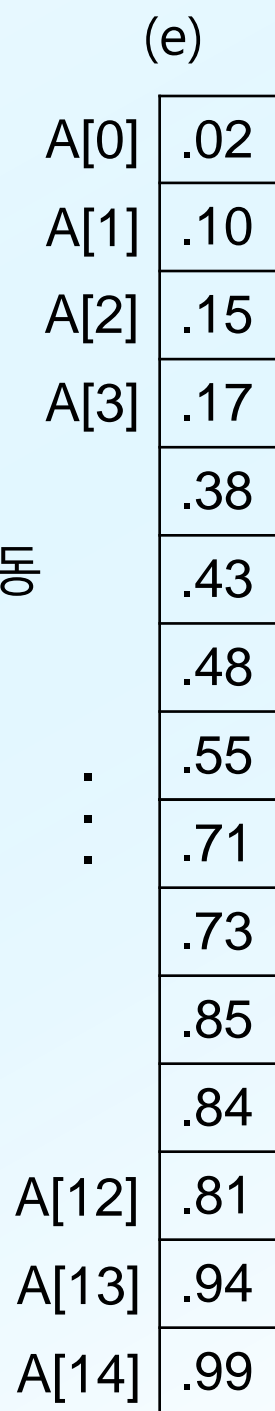
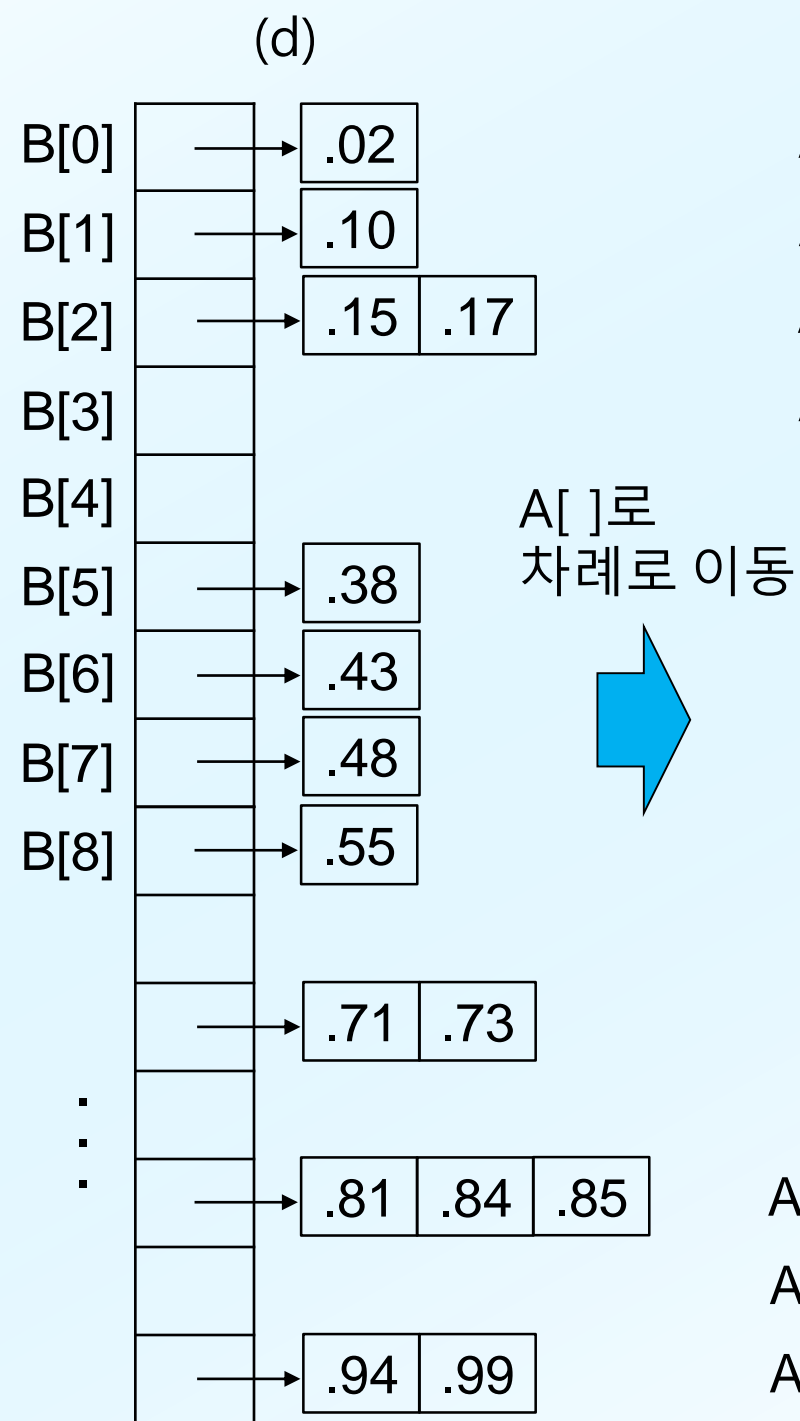
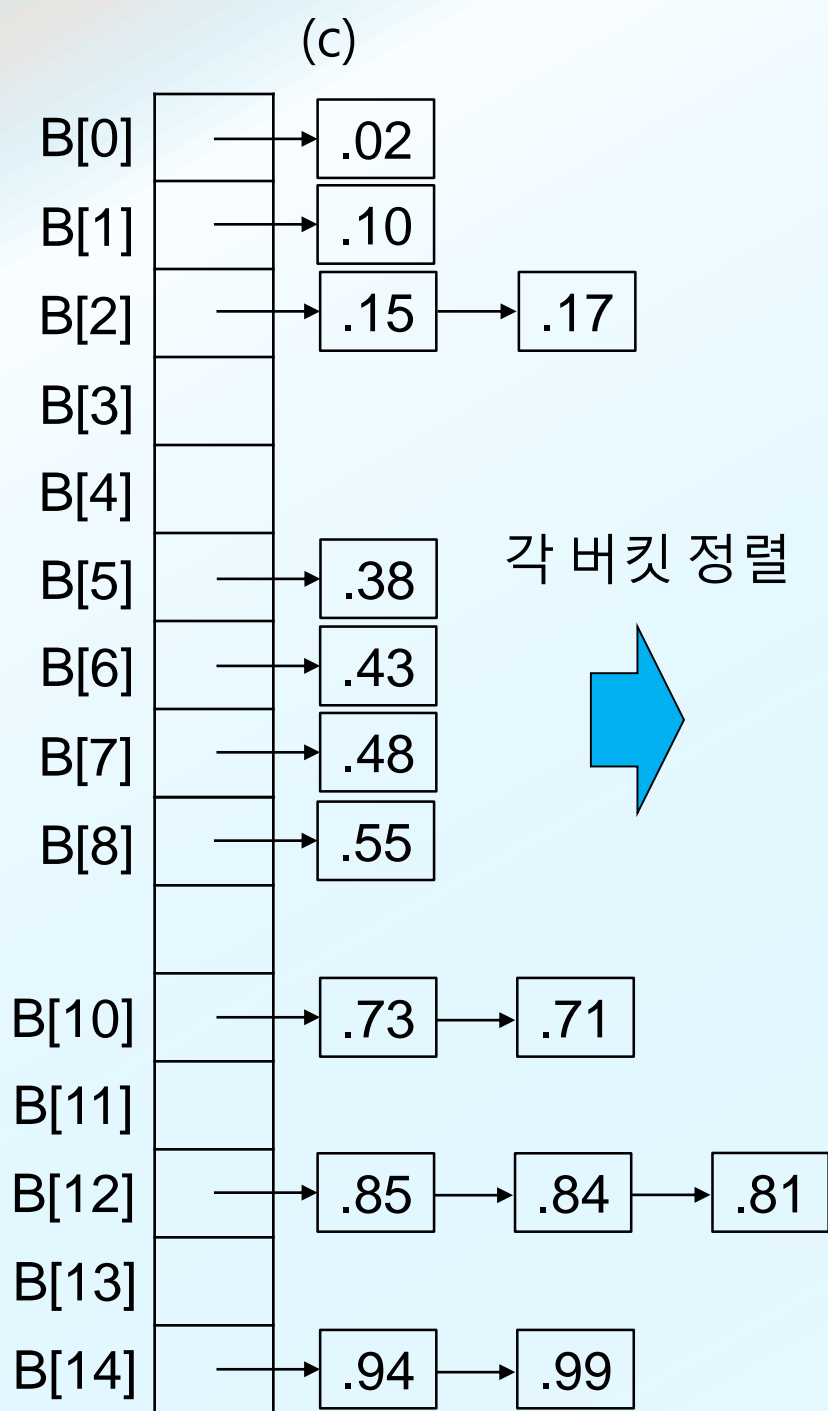


$A[0..14]$ 각각에 15를 곱하여 정수부만 취함

(b) 버킷 리스트 위치

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





시간 복잡도

	Worst Case	Average Case	Best Case
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Counting Sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Radix Sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Bucket Sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

필드 수행 시간 비교

표 9-1 정렬 알고리즘 성능 비교

정렬	점근적 평균 복잡도	배열 크기 10^3 (micro sec) 1000회 평균	배열 크기 10^4 (micro sec) 1000회 평균	배열 크기 10^5 (milli sec) 100회 평균	배열 크기 10^7 (sec)100회 평균 (삽입 정렬은 5회)	배열 크기 10^8 (sec) 10회 평균
선택 정렬	$\Theta(n^2)$	349	24,590	2,603	N/A	N/A
버블 정렬	$\Theta(n^2)$	850	104,195	13,328	N/A	N/A
삽입 정렬	$\Theta(n^2)$	115	9,359	982	23시간	N/A
셸 정렬	$O(n^{1+1/v})$	62	647	9.05	1.65초	23.7
병합 정렬	$\Theta(n \log n)$	67	687	8.51	1.35초	15.5
퀵 정렬	$\Theta(n \log n)$	62	593	6.95	0.93초	10.3
힙 정렬	$\Theta(n \log n)$	75	793	10.93	2.36초	35.9
Room to improve → 버킷 정렬	$\Theta(n)$	103	739	13.6	4.62초	Overflow