알고리즘

4장 정렬(sorting)

학습내용

1. 기본적인 정렬 알고리즘
2. 고급 정렬 알고리즘
3. 비교 정렬 시간의 하한
4. 특수 정렬 알고리즘

학습목표

- 기본 정렬 알고리즘을 이해한다.
- 정렬을 귀납적 관점에서 볼 수 있도록 한다.
- 정렬의 수행시간을 분석할 수 있도록 한다.
- 비교정렬의 한계를 이해하고, 선형시간 정렬이 가능한 조건과 선형시간 정렬 알고리즘을 이해한다.

학습내용

1. 기본적인 정렬 알고리즘

2. 고급 정렬 알고리즘
3. 비교 정렬 시간의 하한
4. 특수 정렬 알고리즘

정렬 알고리즘

- 정렬은 n개의 원소를 기준에 따라 순서대로 배열하는 것
- 여러 다른 알고리즘에 정렬이 포함되어 있음
- 배열 A[1...n]을 정렬하는 알고리즘
  - 대부분 수행시간은 **O($n$log$n$) ~ O(n^2)** 범위
  - 정렬하고자 하는 값이 특수한 성질을 만족하는 경우에는 **O(n)** 시간 정렬도 가능

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - 기수 정렬

선택 정렬(selection sort)

- 원리
  - 입력 원소들 중에서 최대 원소를 **선택**한다.

    29,10,14,37,13   *37*

  - 이 최대 원소를 제외한 나머지 입력 원소들 중 최대 원소를 **선택**한다.

    29,10,14,13   *29 37*

  - 이러한 작업을 반복한다.

선택 정렬

A					
	29	10	14	37	13

선택 정렬

최대 원소

swap

최대 원소를 찾아 가장 마지막 원소와 교환

배열 A의 초기 상태	29	10	14	37	13
첫번째 반복 후	29	10	14	13	**37**
두번째 반복 후	13	10	14	**29**	**37**
세번째 반복 후	13	10	**14**	**29**	**37**
배열 A가 정렬된 상태	**10**	**13**	**14**	**29**	**37**

selectionSort 알고리즘

```
selectionSort(A[], n)      ▷ 배열 A[1...n]을 정렬한다
{
      for last ← n downto 2 {                          ------------------ ①
            A[1...last] 중 가장 큰 수 A[k]를 찾는다;        -------- ②
            A[k] ↔ A[last];  ▷ A[k]와 A[last]의 값을 교환   ------- ③
      }
}
```

✓ 수행시간: $\Theta(n^2)$

- ①의 **for** 루프는 n-1번 반복
- ②에서 가장 큰 수를 찾기 위한 비교횟수 = 첫번째 반복시는 n-1, 두번째 반복시는 n-2, …, n-2번째 반복시는 2, n-1번째 반복시는 1
  ➔ 이 비교하는 횟수가 전체 수행시간을 좌우함
- ③의 교환은 상수 시간 작업

✓ $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = \Theta(n^2)$

selectionSort 알고리즘

```
selectionSort(A[], n)      ▷ 배열 A[1...n]을 정렬한다
{
    for last ← n downto 2 {
        k ← theLargest(A, last) ▷ A[1...last] 중 가장 큰 수의 위치 k 찾기
        A[k] ↔ A[last];         ▷ A[k]와 A[last]의 값을 교환
    }
}

theLargest(A[], last)  ▷ 배열 A[1...last]에서 가장 큰 수의 인덱스 리턴
{
    largest ← 1;
    for i ← 2 to last
        if (A[i] > A[largest]) then largest ← i;
    return largest;
}
```

선택정렬 예

A	8	5	11	2	3	7

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - **버블 정렬**
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - 기수 정렬

버블 정렬(bubble sort)

- 원리
  - 서로 인접한 원소들끼리 비교하여 정렬 순서에 맞지 않으면 교환한다.➔ 최대 원소를 가장 뒤로 보내는 효과

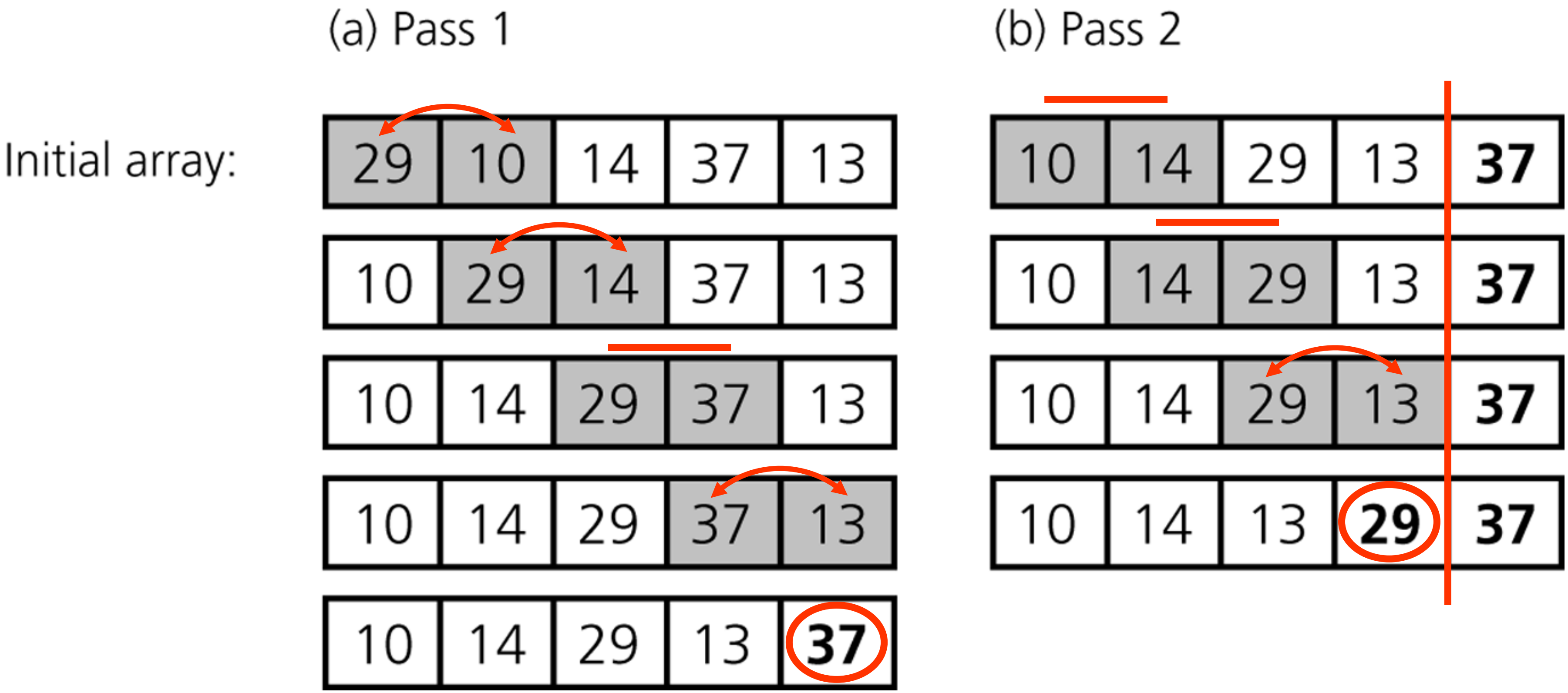
29	*10*	*14*	*37*	*13*
10	*29*	*14*	*37*	*13*
10	*14*	*29*	*37*	*13*
10	*14*	*29*	*37*	*13*
10	*14*	*29*	*13*	***37***

  - 이 최대 원소를 제외한 나머지 원소들을 같은 방법으로 교환한다. ➔ 두 번째 최대 원소를 뒤에서 두 번째로 보내는 효과
  - 이러한 작업을 반복한다.

버블 정렬

A	29	10	14	37	13

버블 정렬



이어서 Pass 3, Pass 4를 수행함

bubbleSort 알고리즘

```
bubbleSort(A[], n)    ▷ A[1...n]을 정렬한다
{
    for last ← n downto 2                                    ------------------ ①
        for i ← 1 to last-1                                  ------------------ ②
            if (A[i] > A[i+1]) then A[i] ↔ A[i+1]; ▷ 원소 교환 -- ③
}
```

✓ 수행시간: $\Theta(n^2)$

- ①의 **for** 루프는 n-1번 반복
- ②의 **for** 루프는 n-1, n-2, …, 2, 1번 반복
- ③은 상수 시간 작업

✓ $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = \Theta(n^2)$

버블 정렬 예

A					
8	5	11	2	3	7

bubbleSort 알고리즘의 변형

```
bubbleSort2(A[], n)    ▷ A[1...n]을 정렬한다
{
    for last ← n downto 2 {                          ----------------- ①
            sorted ← TRUE;
            for i ← 1 to last-1  {                   ----------------- ②
                    if (A[i] > A[i+1]) then {
                            A[i] ↔ A[i+1]; ▷ 원소 교환
                            sorted ← FALSE;
                    }
            }
            if(sorted = TRUE) then return;           ----------------- ③
                            // 루프 ②에서 원소교환이 한번도 일어나지 않으면
    }                       // 정렬이 완료된 상태인 것이므로 종료
}
```

- ✓ best-case 수행시간은 $\Theta(n)$: 이미 정렬된 배열인 경우
  ①의 last가 n일 때 sorted를 TRUE로 만든 후 ② 반복을 수행하는 동안 원소교환이 전혀 일어나지 않으므로 ③에서 알고리즘 종료
- ✓ bubbleSort2의 수행시간은 여전히 $\boldsymbol{O}(n^2)$

버블 정렬 예

A	2	3	5	7	8	9

✓ 위의 배열은 bubbleSort2를 이용하는 경우 best-case

선택 정렬과 버블 정렬

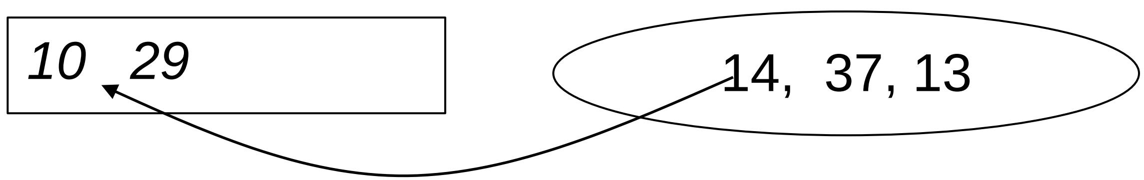
- 선택 정렬과 버블 정렬은 각 단계에서 최대값을 가장 뒤로 보낸다는 점에서는 동일하다. 다음 관점에서 동일한지 차이가 있는지 생각해보세요.
  - 평균 복잡도
  - 각 단계의 비교 횟수
  - 각 단계의 자료 교환 횟수
  - 안정 정렬
  - 필요한 단계 수

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - **삽입 정렬**
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - 기수 정렬

삽입 정렬(insertion sort)

- 원리
  - 이미 정렬되어 있는 배열이 있다고 보고, 다른 한 원소를 그 배열에 **삽입**한다. (배열의 정렬 순서를 유지하도록 적절한 위치에 삽입)

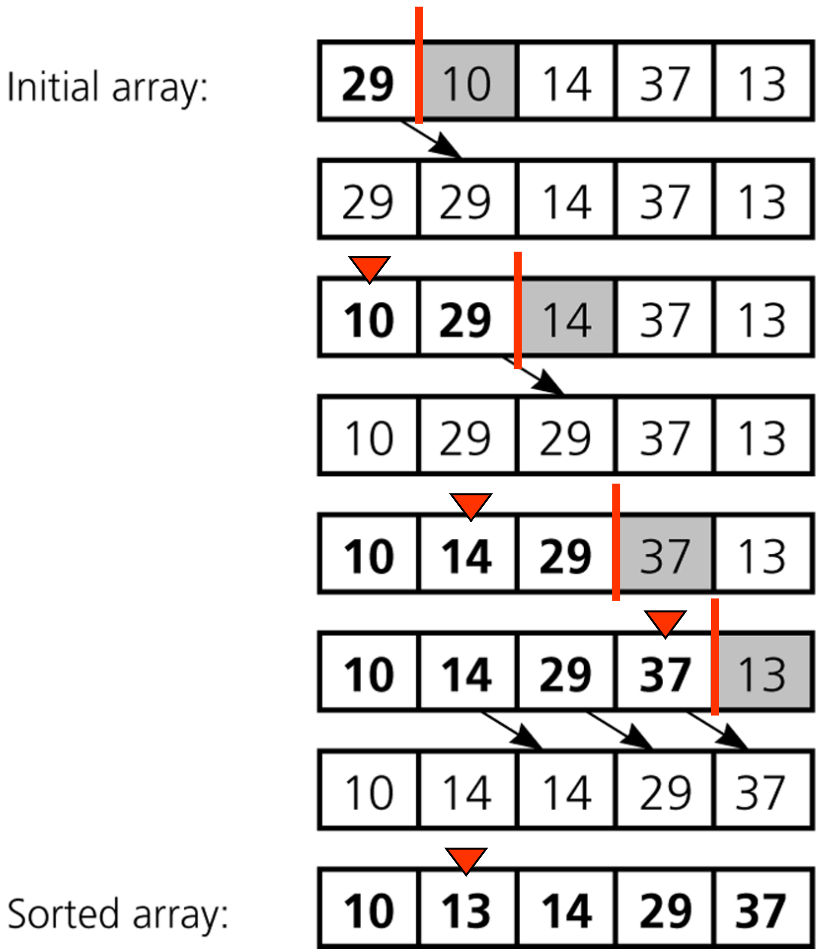


  - 이러한 작업을 반복한다.

삽입 정렬

A					
A	29	10	14	37	13

삽입 정렬



10을 복사해 둠

10보다 큰 29를 shift

10을 삽입; 14를 복사해 둠

14보다 큰 29를 shift

14를 삽입; 37을 복사한 후 제자리에 37 삽입

13을 복사해 둠

13보다 큰 37, 29, 14를 shift

13을 삽입

insertionSort 알고리즘

insertionSort(A[], n)　　▷ A[1...n]을 정렬한다

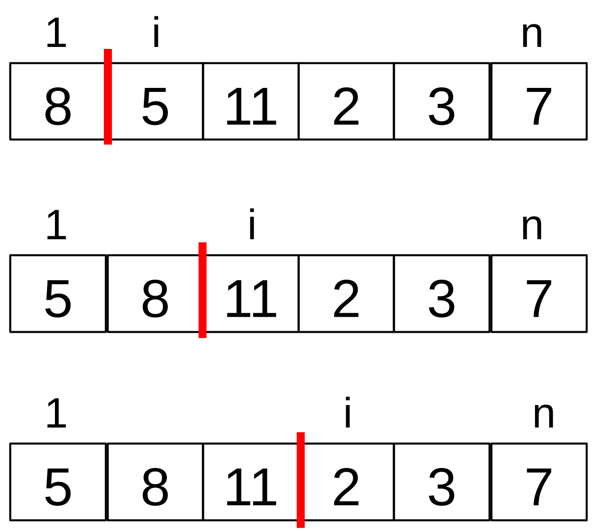
{

　　for i ← 2 **to** n {

　　　　A[1...i]의 적당한 자리에 A[i]를 삽입한다;

　　}

}



……

```
insertionSort(A[], n)        ▷ A[1...n]을 정렬한다
{
      for i ← 2 to n {                              -------------------------- ①
            loc ← i-1;
            newItem ← A[i];
            ▷ 이 지점에서 A[1…i-1]은 이미 정렬된 상태
            while(loc≥1 and newItem < A[loc]) {
                  A[loc+1] ← A[loc];
                  loc--;
            }
            A[loc+1] ← newItem;
      }
}
```

A[1...i]의 적당한 자리에 A[i]를 삽입한다 ------- ②

✓ 수행시간: $\boldsymbol{O}(n^2)$

— ①의 **for** 루프는 n-1번 반복

— ②는 최악의 경우 i-1회 비교, 최선의 경우 1번 비교

✓ Worst case: $1+2+\cdots+(n-2)+(n-1) = \Theta(n^2)$

✓ Average case: $\frac{1}{2}(1+2+\cdots+(n-2)+(n-1)) = \Theta(n^2)$

✓ Best case: $1+1+\cdots+1 = \Theta(n)$

삽입정렬 예

Best-case

2	3	5	7	8	9

Worst-case

11	9	8	7	4	2

동일한 키값이 있는 경우

2	5	7	5’	8	9

요약

- 기본적인 정렬 알고리즘
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬

➔ 평균적인 경우, 최악의 경우에 모두 $\Theta(n^2)$

학습내용

1. 기본적인 정렬 알고리즘
2. **고급 정렬 알고리즘**
3. 비교 정렬 시간의 하한
4. 특수 정렬 알고리즘

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - **병합 정렬**
  - 퀵 정렬
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - 기수 정렬

병합 정렬(merge sort)

- 원리
  - 정렬할 대상을 반으로 나눈다.
  - 이렇게 나눈 전반부와 후반부를 각각 독립적으로 정렬한다.
  - 정렬된 두 부분을 합쳐서 정렬된 배열을 얻는다.

병합 정렬

```
mergeSort(A[], p, r)    ▷ A[p...r]을 정렬한다
{
      if (p < r) then {
            q ← (p+r)/2;               ---------  ①  ▷ p, r의 중간 지점 계산
            mergeSort(A, p, q);        ---------  ②  ▷ 전반부 정렬
            mergeSort(A, q+1, r);      ---------  ③  ▷ 후반부 정렬
            merge(A, p, q, r);         ---------  ④  ▷ 병합
      }
}

merge(A[], p, q, r)
{
      정렬되어 있는 두 부분 배열 A[p...q]와 A[q+1...r]을 병합하여
      정렬된 하나의 배열 A[p...r]을 만든다.
}
```

✓ 병합 정렬의 수행 시간: $\Theta(n \log n)$

mergeSort 작동 예

정렬할 배열이 주어짐

31	3	65	73	8	11	20	29	48	15

배열을 반으로 나눔

31	3	65	73	8	11	20	29	48	15

1

두 부분배열을 독립적으로 정렬 – 이 때 mergeSort 알고리즘 이용(재귀적)

3	8	31	65	73	11	15	20	29	48

2 3

두 부분배열을 병합(정렬완료)

3	8	11	15	20	29	31	48	65	73

4

merge 작동 예

merge(A[], p, q, r) ▷ 상세 알고리즘은 p.99 참조

{

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

}

	p			q				r
A	1	2	3	8	3	5	9	11

tmp								

두 부분배열의 병합에
임시 배열이 필요

병합정렬 예

A							
8	5	11	2	3	8	9	1

병합정렬의 수행시간

- mergeSort는 재귀 알고리즘
  - 수행 시간 = 두 부분배열을 각각 정렬하는 시간 + 병합 시간
  - 수행 시간을 점화식으로 표현할 수 있다.

    $T(n) = 2T(n/2) + \Theta(n)$
  - 마스터 정리를 이용하여 점근적 표기를 구할 수 있다.

    $T(n) = \Theta(n \log n)$

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - **퀵 정렬**
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - 기수 정렬

퀵 정렬(quick sort)

- 원리
  - 정렬할 배열에서 기준원소를 하나 고른다.
  - 이 기준원소를 중심으로, 더 작거나 같은 원소는 왼쪽으로, 큰 원소는 오른쪽으로 재배치함으로써 배열을 분할한다.
  - 이렇게 분할된 왼쪽, 오른쪽 부분 배열을 각각 독립적으로 정렬한다.

퀵 정렬

```
quickSort(A[], p, r)   ▷ A[p...r]을 정렬한다
{
    if (p < r) then {
        q = partition(A, p, r);        ▷ 분할
        quickSort(A, p, q-1);          ▷ 왼쪽 부분배열 정렬
        quickSort(A, q+1, r);          ▷ 오른쪽 부분배열 정렬
    }
}

partition(A[], p, r)
{
    A[r]을 기준값으로 하여 A[p...r]의 원소들을 양쪽으로 재배치(기준값
    이하는 왼쪽, 나머지는 오른쪽에 배치)하고,
    기준값이 저장된 위치를 return 한다;
}
```

✓ 퀵 정렬의 평균 수행 시간: $\Theta(n \log n)$

quickSort 작동 예

정렬할 배열이 주어짐. 가장 끝에 있는 수를 기준(pivot)으로 삼기로 하자.

31	8	48	73	11	3	20	29	65	15

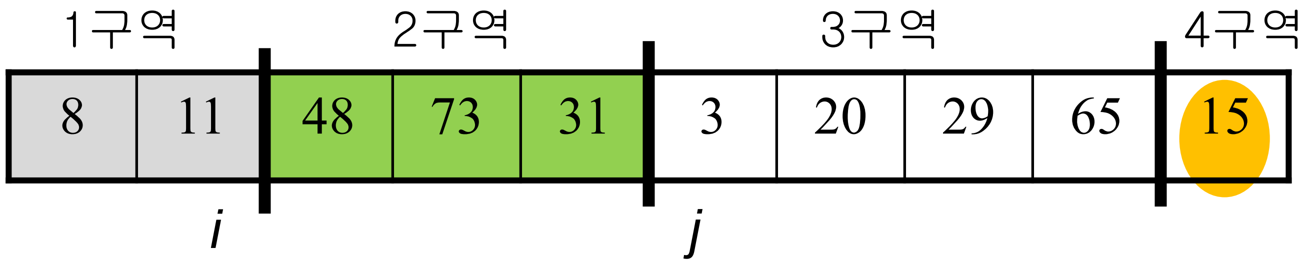
기준 이하인 수는 기준의 왼쪽에, 나머지는 기준의 오른쪽에 오도록 재배치 (이것이 partition에서 하는 일임)

8	11	3	15	31	48	20	29	65	73

기준 왼쪽과 오른쪽을 각각 독립적으로 정렬 – 이 때 quickSort 알고리즘 이용(재귀적)

3	8	11	15	20	29	31	48	65	73

partition 알고리즘



1구역 = $p \sim i$ = 기준보다 작거나 같은 수

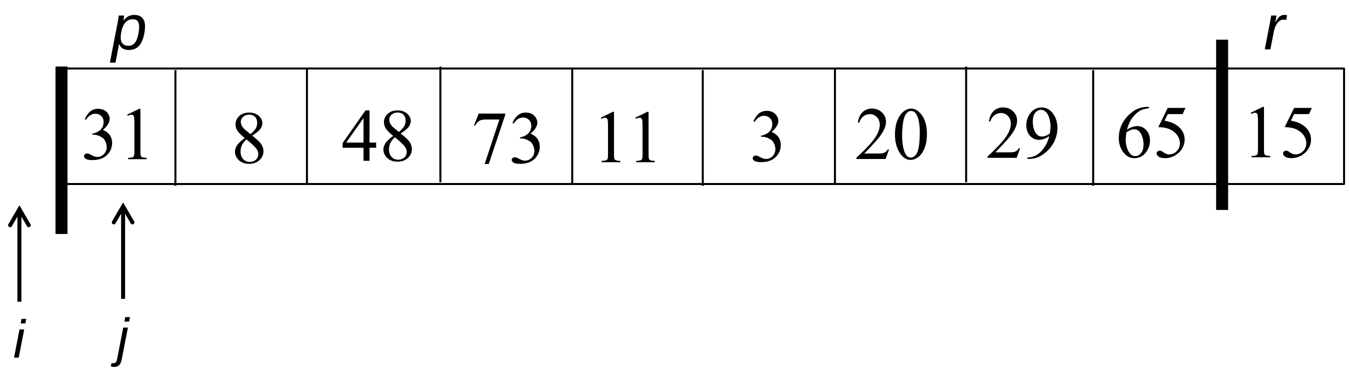
2구역 = $i+1 \sim j-1$ = 기준보다 큰 수

3구역 = $j \sim r-1$ = 아직 검사하지 않은 수

4구역 = r = 기준

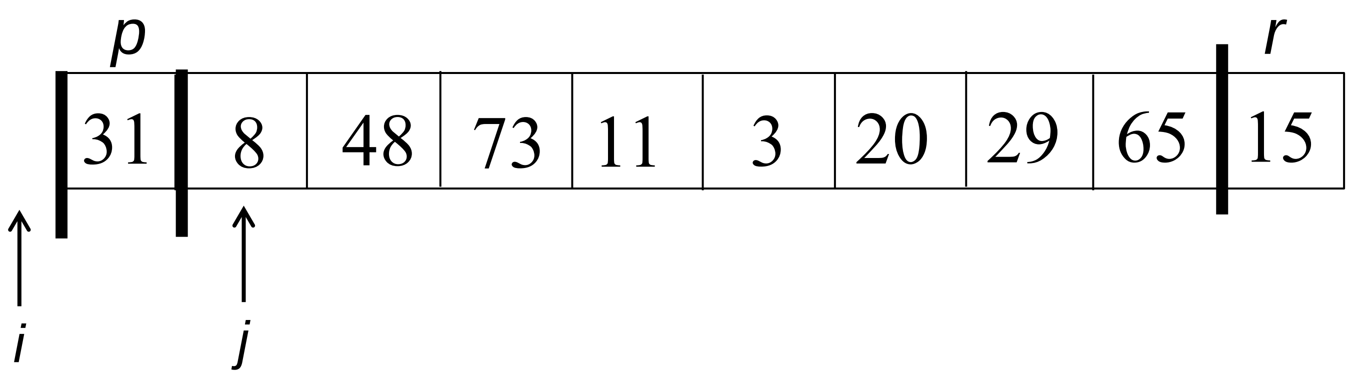
✓ 기준원소와 같은 값은 좌, 우 어느 쪽으로 보내도 됨

partition 알고리즘



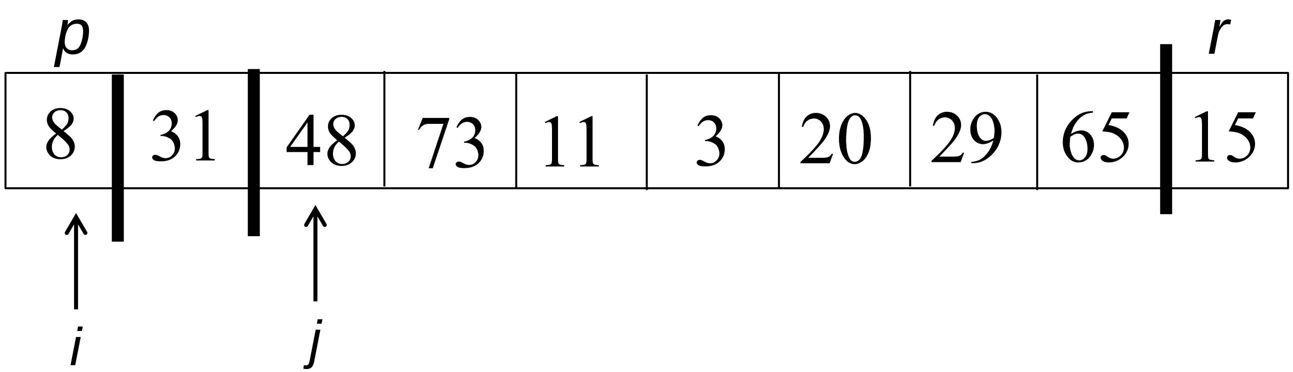

```
partition(A[], p, r)
{
    x ← A[r];                              ▷기준원소(pivot)
    i ← p−1;                               ▷i는 1구역의 끝지점
    for j ← p to r−1                       ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                         ▷기준원소와 2구역 첫 원소 교환
    return i+1;                            ▷기준원소의 위치 리턴
}
```

partition 알고리즘



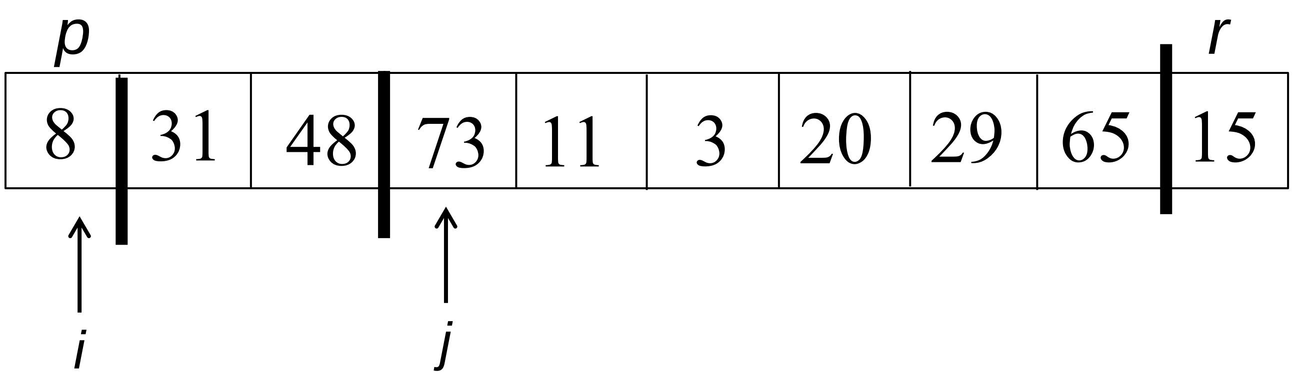

```
partition(A[], p, r)
{
    x ← A[r];                          ▷기준원소(pivot)
    i ← p−1;                           ▷i는 1구역의 끝지점
    for j ← p to r−1                   ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                     ▷기준원소와 2구역 첫 원소 교환
    return i+1;                        ▷기준원소의 위치 리턴
}
```

partition 알고리즘



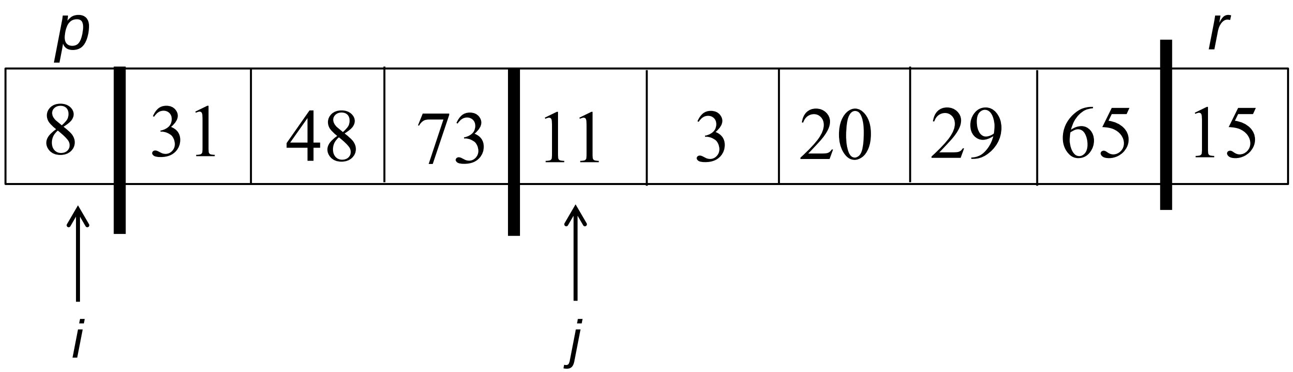

```
partition(A[], p, r)
{
    x ← A[r];                              ▷기준원소(pivot)
    i ← p−1;                               ▷i는 1구역의 끝지점
    for j ← p to r−1                       ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                         ▷기준원소와 2구역 첫 원소 교환
    return i+1;                            ▷기준원소의 위치 리턴
}
```

partition 알고리즘



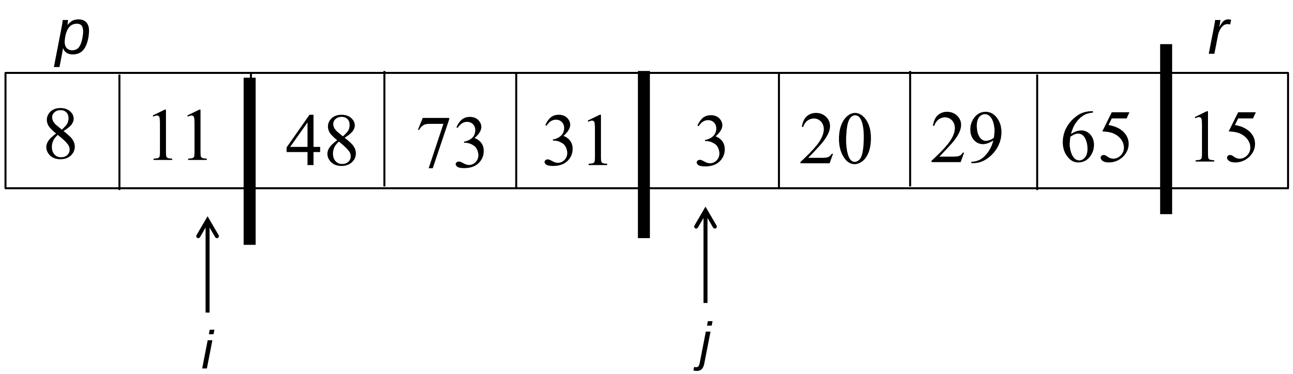

```
partition(A[], p, r)
{
    x ← A[r];                              ▷기준원소(pivot)
    i ← p−1;                               ▷i는 1구역의 끝지점
    for j ← p to r−1                       ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                         ▷기준원소와 2구역 첫 원소 교환
    return i+1;                            ▷기준원소의 위치 리턴
}
```

partition 알고리즘



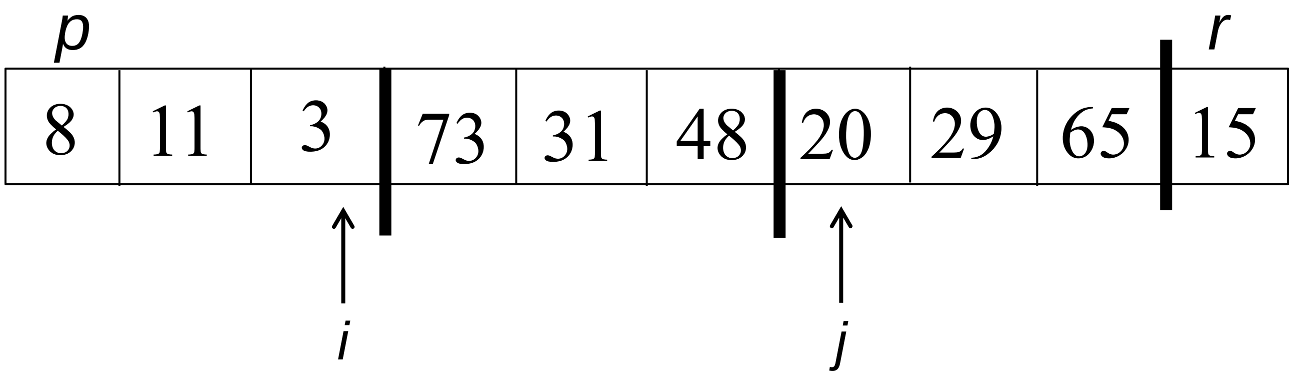

```
partition(A[], p, r)
{
    x ← A[r];                                  ▷기준원소(pivot)
    i ← p−1;                                   ▷i는 1구역의 끝지점
    for j ← p to r−1                           ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                             ▷기준원소와 2구역 첫 원소 교환
    return i+1;                                ▷기준원소의 위치 리턴
}
```

partition 알고리즘



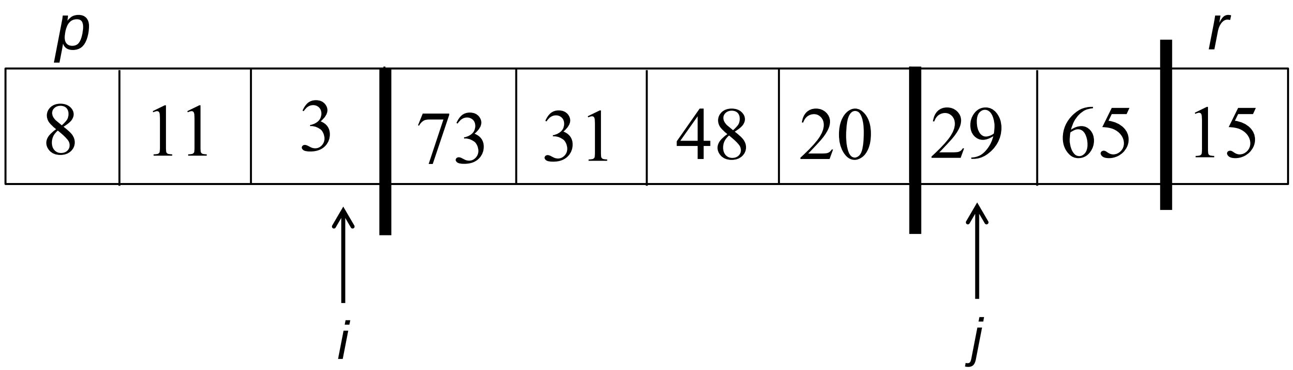

```
partition(A[], p, r)
{
    x ← A[r];                               ▷기준원소(pivot)
    i ← p−1;                                ▷i는 1구역의 끝지점
    for j ← p to r−1                        ▷j는 3구역의 시작지점
          if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                          ▷기준원소와 2구역 첫 원소 교환
    return i+1;                             ▷기준원소의 위치 리턴
}
```

partition 알고리즘



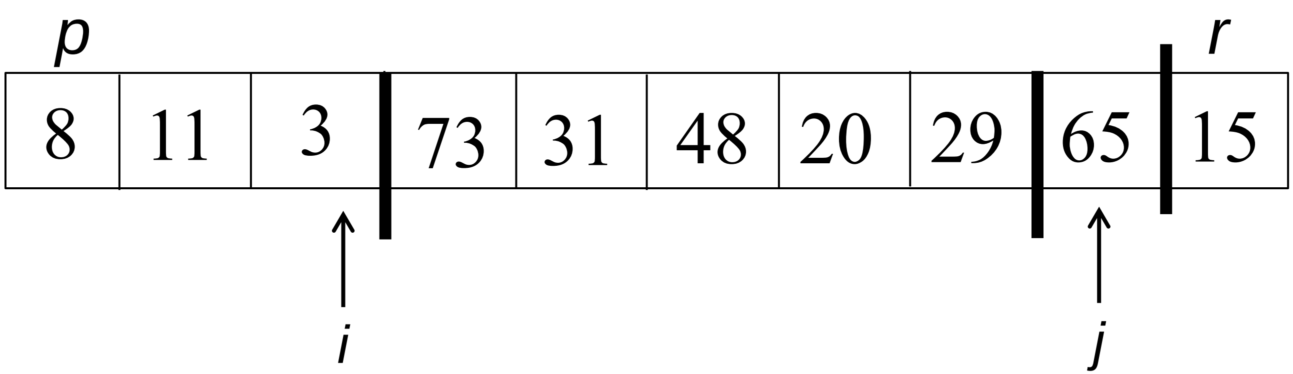

```
partition(A[ ], p, r)
{
    x ← A[r];                                   ▷기준원소(pivot)
    i ← p−1;                                    ▷i는 1구역의 끝지점
    for j ← p to r−1                            ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                              ▷기준원소와 2구역 첫 원소 교환
    return i+1;                                 ▷기준원소의 위치 리턴
}
```

partition 알고리즘



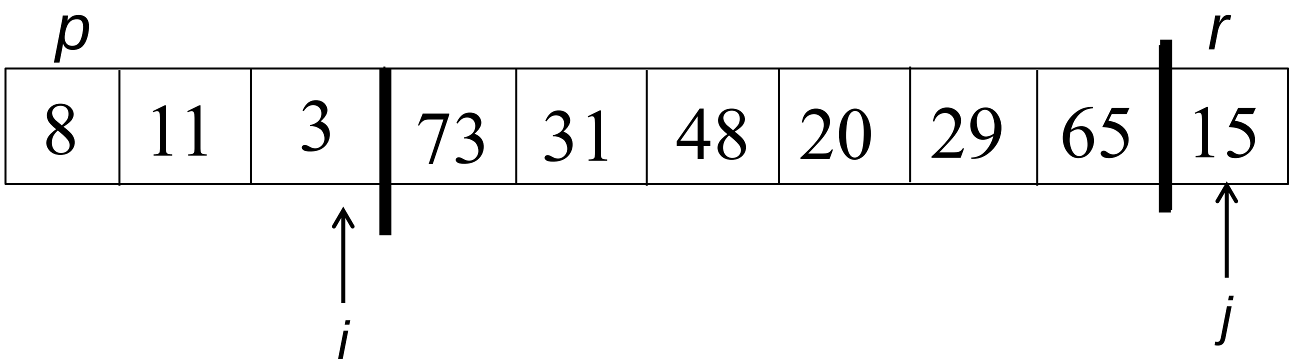

```
partition(A[], p, r)
{
    x ← A[r];                              ▷기준원소(pivot)
    i ← p−1;                               ▷i는 1구역의 끝지점
    for j ← p to r−1                       ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                         ▷기준원소와 2구역 첫 원소 교환
    return i+1;                            ▷기준원소의 위치 리턴
}
```

partition 알고리즘




```
partition(A[], p, r)
{
    x ← A[r];                                  ▷기준원소(pivot)
    i ← p−1;                                   ▷i는 1구역의 끝지점
    for j ← p to r−1                           ▷j는 3구역의 시작지점
          if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                             ▷기준원소와 2구역 첫 원소 교환
    return i+1;                                ▷기준원소의 위치 리턴
}
```

partition 알고리즘




```
partition(A[], p, r)
{
    x ← A[r];                               ▷기준원소(pivot)
    i ← p−1;                                ▷i는 1구역의 끝지점
    for j ← p to r−1                        ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                          ▷기준원소와 2구역 첫 원소 교환
    return i+1;                             ▷기준원소의 위치 리턴
}
```

partition 알고리즘

p									r
8	11	3	15	31	48	20	29	65	73
		i							

```
partition(A[], p, r)
{
    x ← A[r];                                  ▷기준원소(pivot)
    i ← p−1;                                   ▷i는 1구역의 끝지점
    for j ← p to r−1                           ▷j는 3구역의 시작지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
    A[i+1] ↔ A[r];                             ▷기준원소와 2구역 첫 원소 교환
    return i+1;                                ▷기준원소의 위치 리턴
}
```

퀵정렬 예

	1	2	3	4	5	6	7	8
A	1	11	5	2	3	8	9	7

퀵정렬의 수행시간

```
quickSort(A[], p, r)   ▷ A[p...r]을 정렬한다
{
    if (p < r) then {
        q = partition(A, p, r);      ▷ 분할
        quickSort(A, p, q-1);        ▷ 왼쪽 부분배열 정렬
        quickSort(A, q+1, r);        ▷ 오른쪽 부분배열 정렬
    }
}
```

- best-case 수행 시간 : $\Theta(n \log n)$
  - 두 부분배열이 매번 completely balanced 인 경우
  - 각 부분배열 원소 수 = $n/2$

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \log n)$$

퀵정렬의 수행시간

```
quickSort(A[], p, r)   ▷ A[p...r]을 정렬한다
{
    if (p < r) then {
        q = partition(A, p, r);       ▷ 분할
        quickSort(A, p, q-1);         ▷ 왼쪽 부분배열 정렬
        quickSort(A, q+1, r);         ▷ 오른쪽 부분배열 정렬
    }
}
```

- worst-case 수행 시간 : $\Theta(n^2)$
  - 두 부분배열이 매번 completely unbalanced 인 경우 – 한 부분배열은 원소가 $n-1$ 개이고, 다른 부분배열은 원소가 0 개

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \Theta(n^2)$$

퀵정렬의 수행시간

- average-case 수행시간 : $\Theta(n \log n)$
  - worst-case보다 best-case에 더 가깝다.
  - 분할했을 때 모든 가능한 경우를 평균 내어 구할 수 있다.
- 퀵정렬의 수행시간은 분할이 얼마나 균형 잡히게 잘 되느냐에 좌우됨
  - 최악의 경우를 피하려면 기준 원소를 정할 때 매번 가장 크거나 가장 작은 값이 되는 상황을 피해야 함

1 3 4 5 7 8 9

1 3 4 5 7 8 9

요약

- 고급 정렬 알고리즘
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬

➔ 병합 정렬은 평균적인 경우, 최악의 경우 모두 $\Theta(n \log n)$

➔ 퀵 정렬은 평균적인 경우 $\Theta(n \log n)$, 최악의 경우 $\Theta(n^2)$

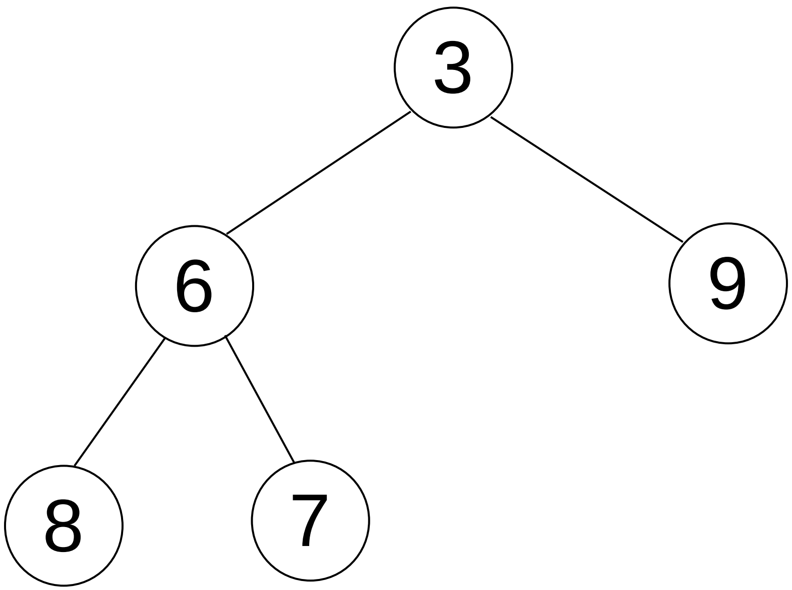
정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - 퀵 정렬
  - **힙 정렬**
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - 기수 정렬

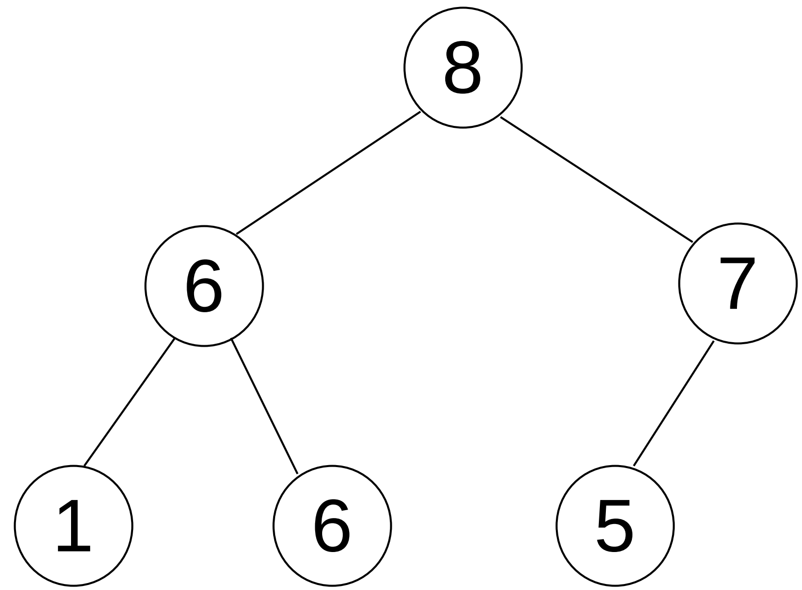
힙 정렬(heap sort)

- 힙(heap)
  - 완전이진트리(complete binary tree)로서 다음 성질을 만족한다.
  - 최소 힙(min heap): 각 노드의 값은 자식(child) 노드의 값보다 작거나 같다. ➔ 루트 노드가 최소값임
  - 최대 힙(max heap): 각 노드의 값은 자식(child) 노드의 값보다 크거나 같다. ➔ 루트 노드가 최대값임
- 힙 정렬
  - 주어진 배열을 힙으로 만든 다음, 차례로 하나씩 힙에서 제거함으로써 정렬한다.

힙(heap)

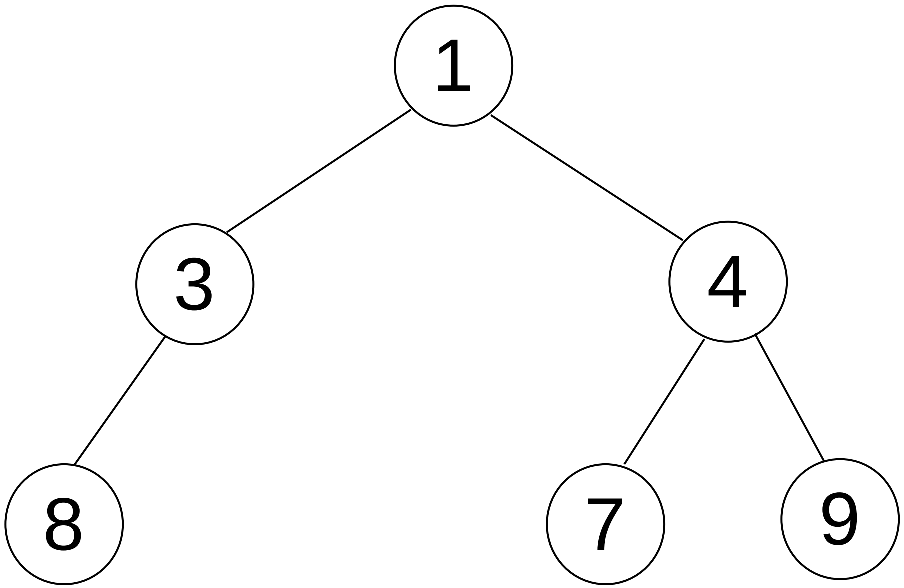


최소 힙(min heap)

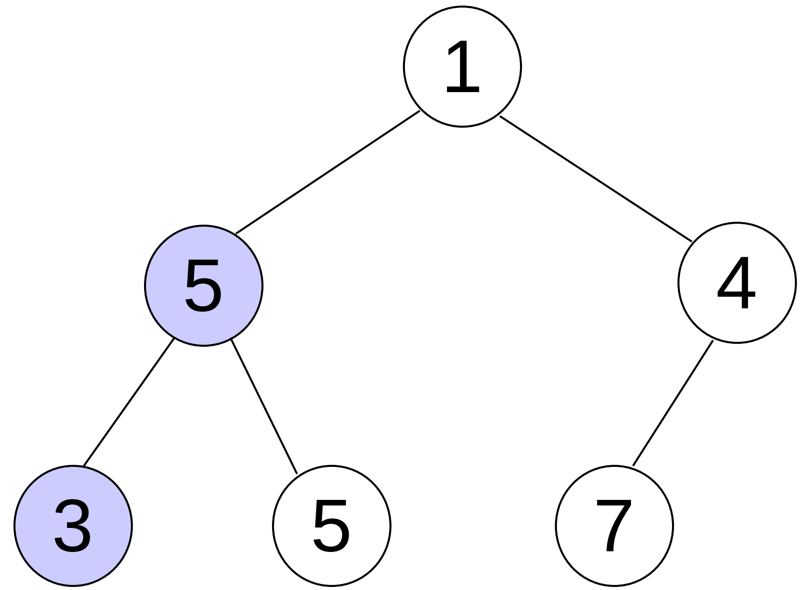


최대 힙(max heap)

힙(heap)

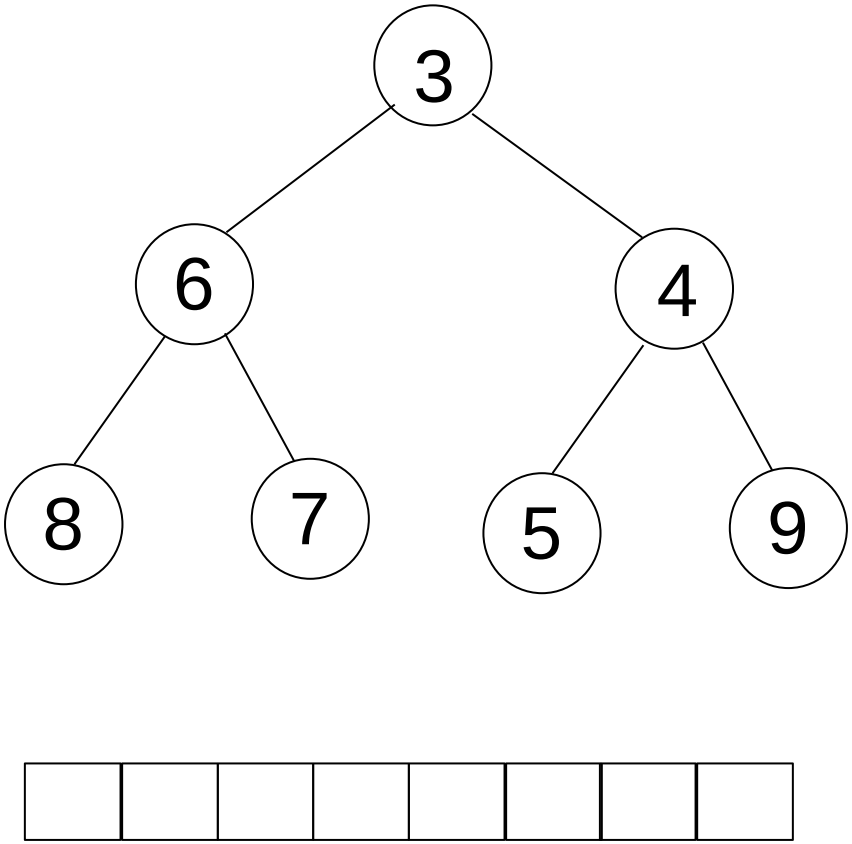


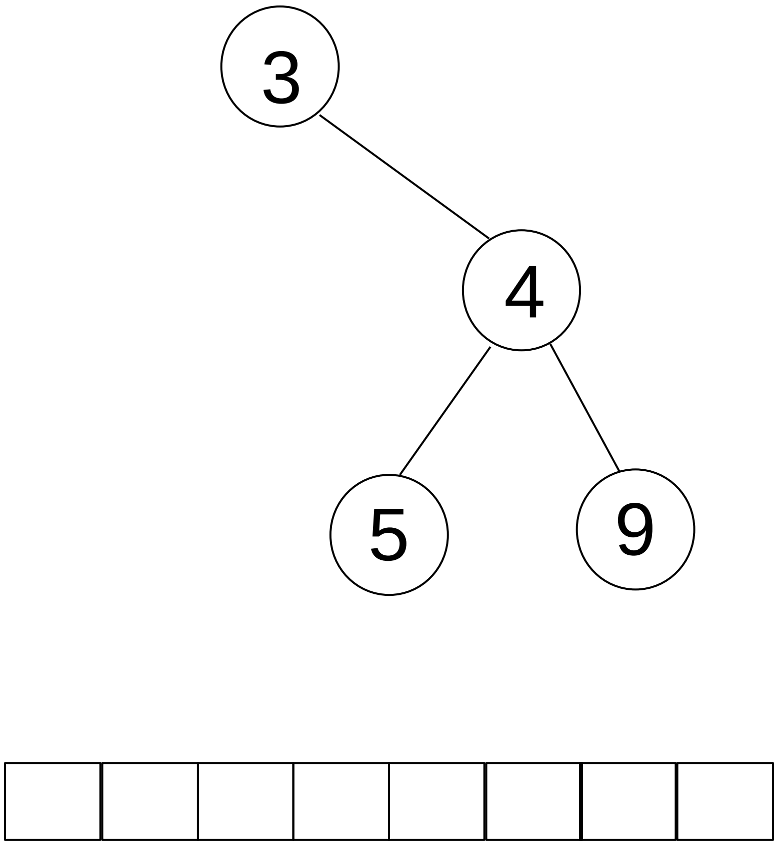
최소 힙 아님



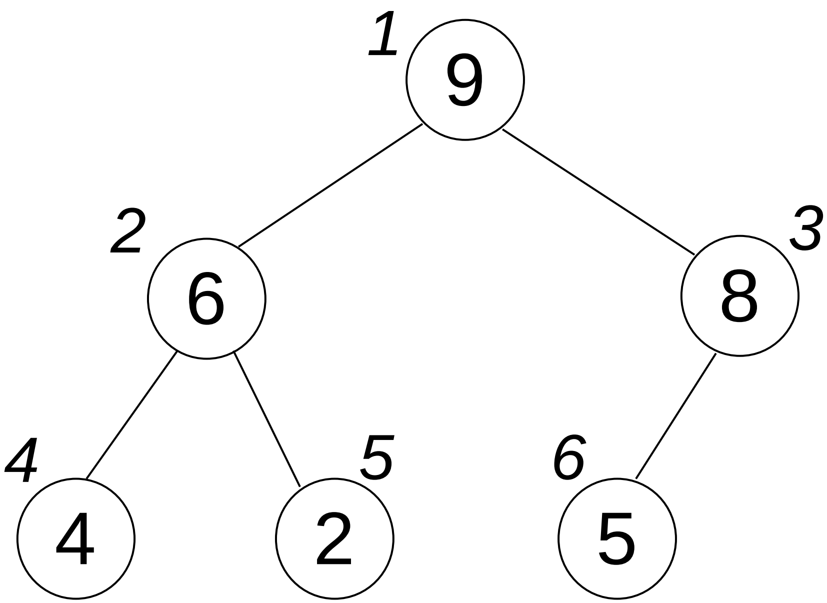
최소 힙 아님

이진트리의 배열 표현

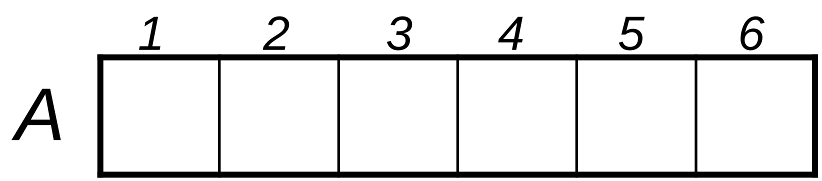




Heap의 배열 표현

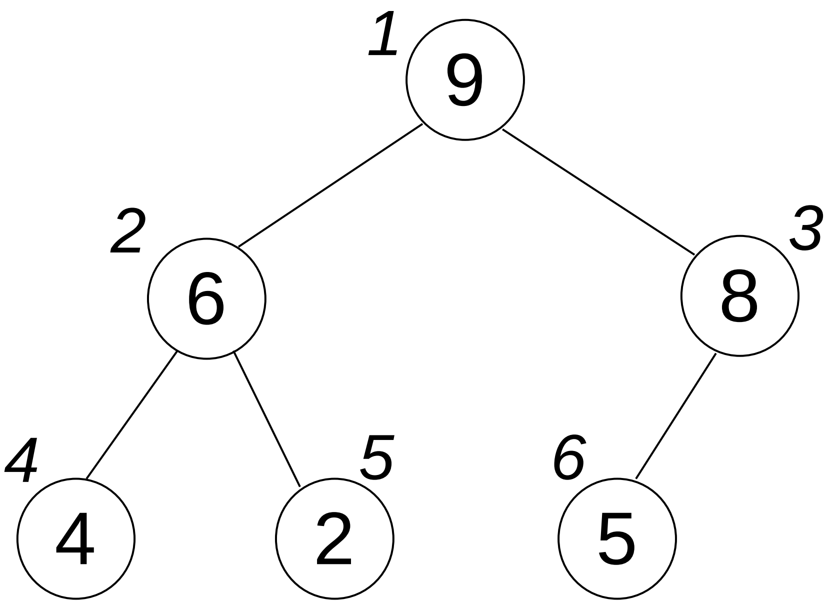


최대 힙



교재에서는 크기가 **n**인 배열의 인덱스는 **1~n**로 설명하며, 최소 힙을 이용한 내림차순 정렬을 다룸(강의 노트 설명은 최대 힙을 이용한 오름차순 정렬임)

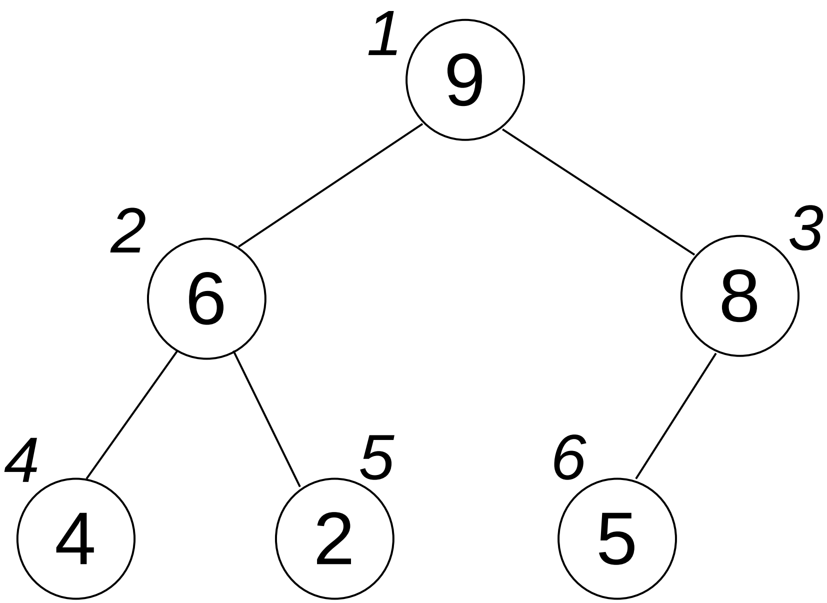
Heap의 삽입



최대 힙

	1	2	3	4	5	6
A	9	6	8	4	2	5

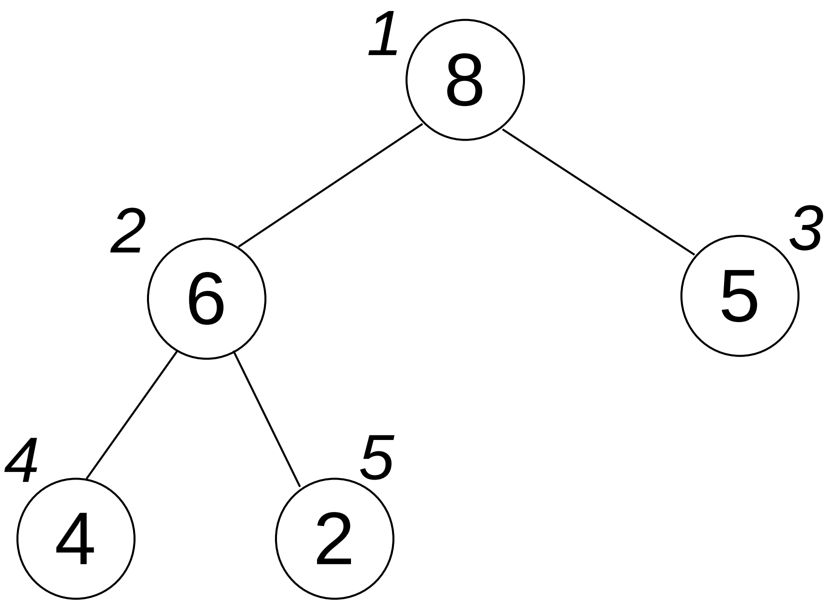
Heap의 삭제



최대 힙

	1	2	3	4	5	6
A	9	6	8	4	2	5

Heap의 삭제



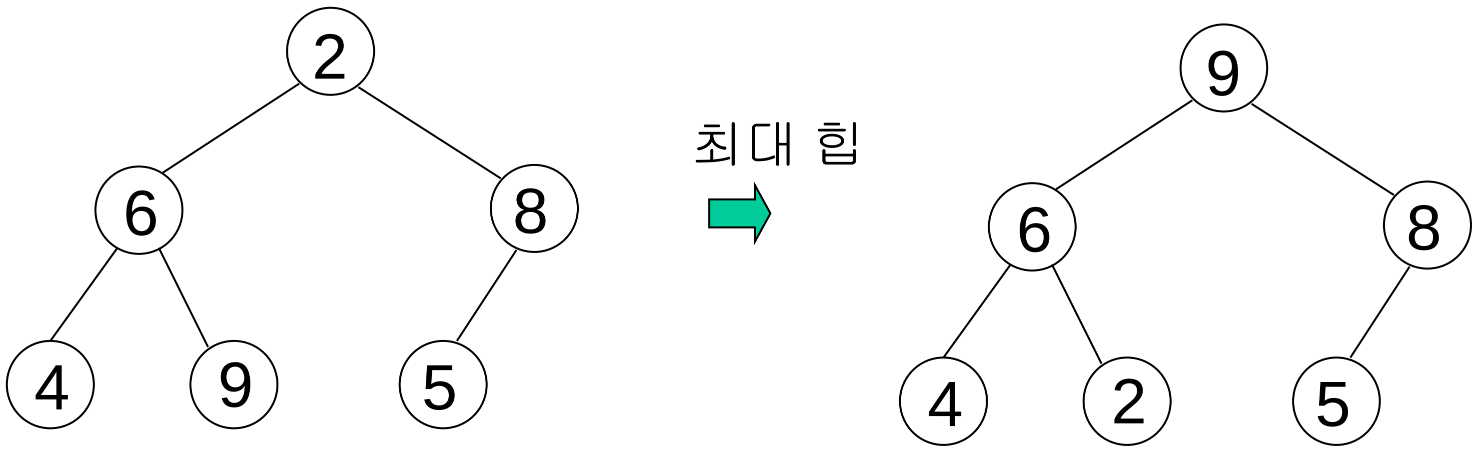
최대 힙

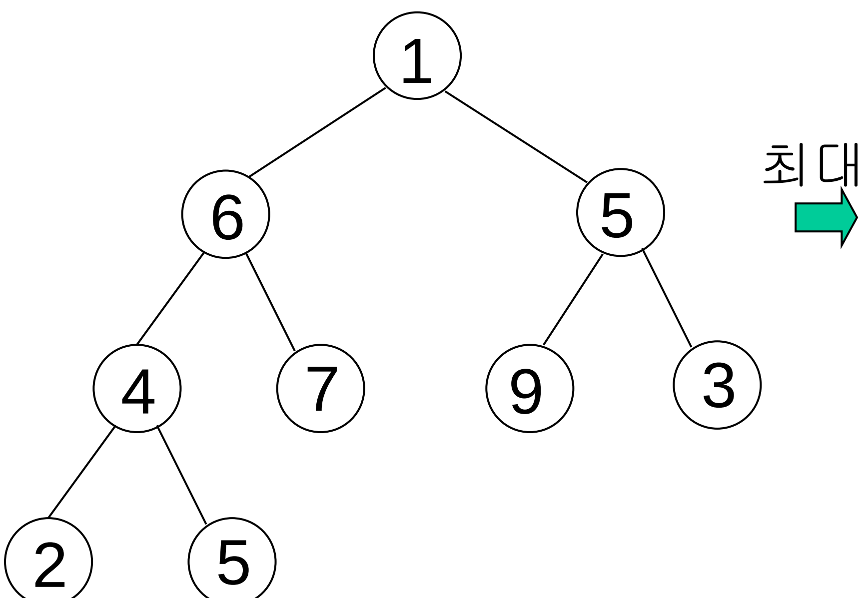
	1	2	3	4	5	6
A	8	6	5	4	2	

힙 만들기

```
buildHeap(A[], n)        ▷ A[1...n]을 힙으로 만든다.
{
      for i ← n/2 downto 1
            heapify(A, i, n);    ▷ 힙 재구성
}
```

buildHeap 예





힙 재구성

```
heapify(A[], k, n)              ▷ n은 최대 인덱스
▷ A[k]의 두 자식을 루트로 하는 서브트리는 힙성질을 이미 만족하고 있다.
▷ A[k]를 루트로 하는 트리를 힙성질을 만족하도록 재구성한다.
{
   ▷ 큰 자식을 고른다.
    left ← 2k;    right ← 2k + 1;
    if (right ≤ n) then {                    ▷ 자식이 둘인 경우
          if (A[left]>A[right]) then bigger←left; else bigger ←right;
    }
    else if (left ≤ n) then bigger ← left;   ▷ 왼쪽 자식만 있는 경우
    else return;                             ▷ 자식이 없는 경우 (종료)

    ▷ 큰 자식이 부모보다 크면 힙성질 위반이므로 재귀적으로 계속 조정
    if (A[bigger] > A[k]) then {
          A[k] ↔ A[bigger];
          heapify(A, bigger, n);
    }
}
```

✓$O(\log n)$

힙 정렬

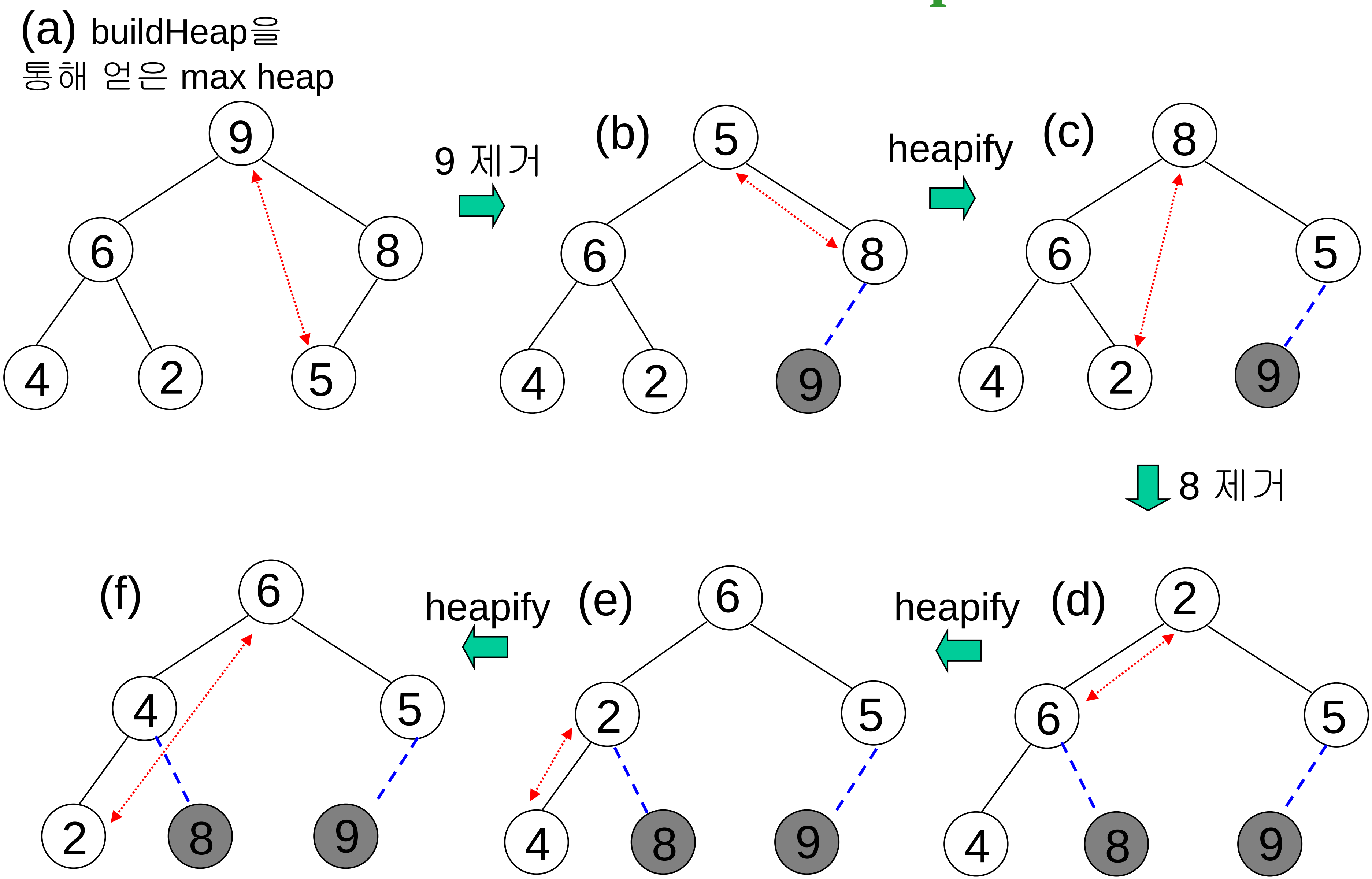
p.114 엄밀하게 계산하면 $\Theta(n)$

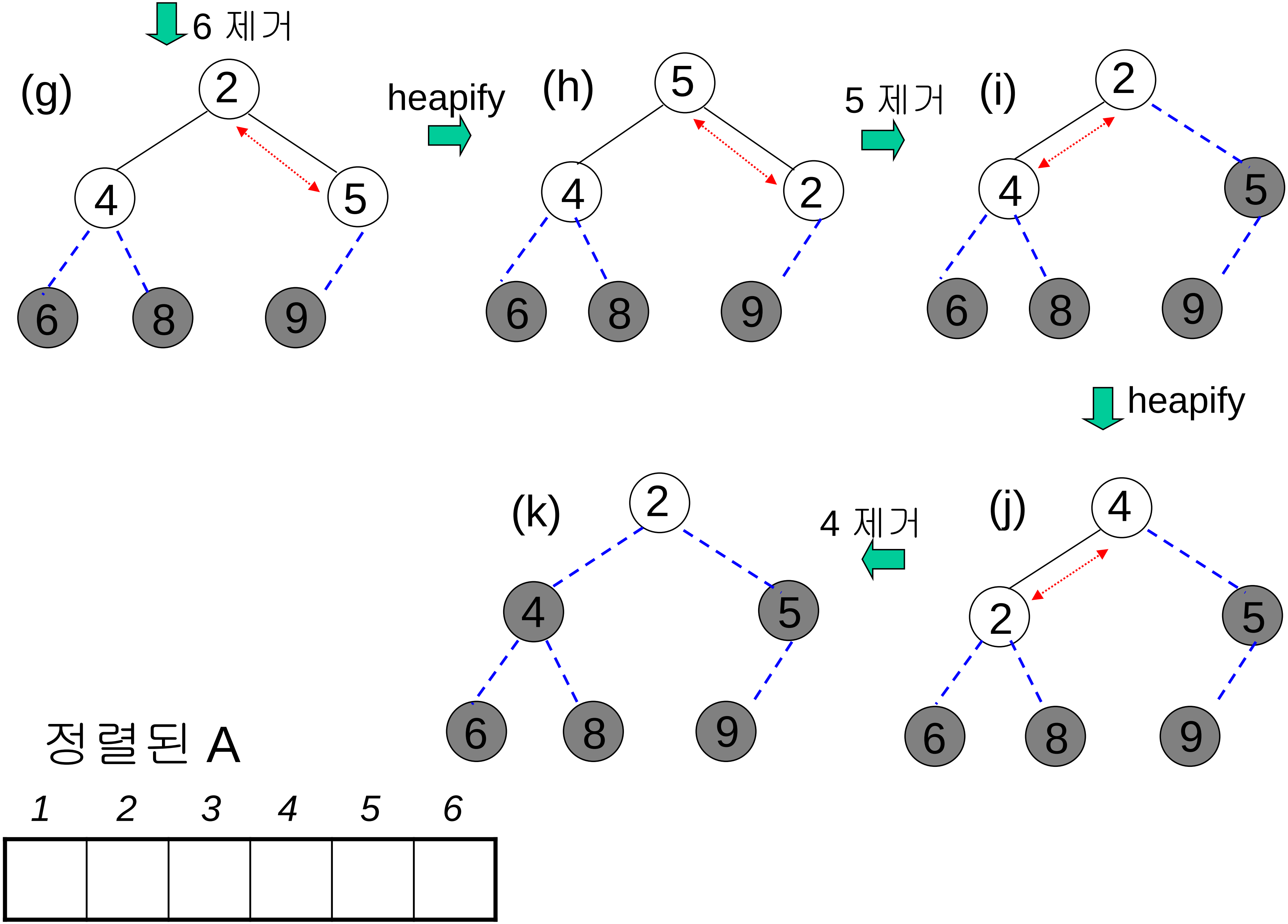
```
heapSort(A[], n)  ▷ A[1...n]을 힙 정렬 한다.
{
        buildHeap(A, n);            ▷ 힙 만들기              : O(n log n)
        for i ← n downto 2 {        ▷                        : n-1 번 수행
                A[1] ↔ A[i];        ▷ 교환(A[1] 제거)        : O(1)
                heapify(A, 1, i-1); ▷ 힙 재구성              : O(log n)
        }
}
```

✓ 최악의 경우에도 $O(n \log n)$ 시간 소요!

점선 - - - - - 은 힙에서 제거된 노드의 원래 간선

heapSort 작동 예




6 제거
(g)
heapify
(h)
5 제거
(i)
heapify
(j)
4 제거
(k)
정렬된 A
1 2 3 4 5 6

Heap implementation of Priority Queue

- 힙 정렬은 실행시간이 **O(n log n)**인 알고리즘이지만, 실제로 퀵 정렬을 잘 구현하는 것이 더 빠름
- heap은 힙 정렬 외의 분야에서도 유용하게 이용됨
  - heap을 이용하여 우선순위 큐(priority queue)를 효율적으로 구현
  - max heap으로 max-priority queue 구현
- heap은 삽입과 삭제에 모두 **O(log n)** 시간이 걸림
  - 다른 구현 방법은 삽입이 빠르면 삭제가 느리고, 삭제가 빠르면 삽입이 느리다.

예) 연결리스트로 priority queue를 구현한 경우와 비교해보자.

	정렬된 리스트	정렬되지 않은 리스트	힙
삽입	**O(n)**	**O(1)**	**O(log n)**
삭제	**O(1)**	**O(n)**	**O(log n)**

문제

- 다음 배열의 원소들을 힙 정렬을 이용하여 오름차순으로 정렬하세요.

1	2	3	4	5	4	5	9	8

요약

- 고급 정렬 알고리즘
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬

➔ 병합 정렬, 힙 정렬은 평균적인 경우, 최악의 경우에 모두 $\Theta(n \log n)$

➔ 퀵 정렬은 평균적인 경우에 $\Theta(n \log n)$, 최악의 경우에 $\Theta(n^2)$ 이지만 실제로 가장 많이 사용되는 정렬 알고리즘

학습내용

1. 기본적인 정렬 알고리즘
2. 고급 정렬 알고리즘
3. **비교 정렬 시간의 하한**
4. 특수 정렬 알고리즘

비교 정렬 시간의 하한

- 원소끼리 비교하는 작업을 통해 정렬하는 방식을 통칭해 비교 정렬(comparison sort)이라 한다.
- 비교 정렬은 최악의 경우 수행시간이 적어도 $\Theta(n \log n)$이다. 즉, $\Omega(n \log n)$ 이다.
  - 선택 정렬, 버블 정렬, 삽입 정렬
  - 병합 정렬, 퀵 정렬, 힙 정렬

학습내용

1. 기본적인 정렬 알고리즘
2. 고급 정렬 알고리즘
3. 비교 정렬 시간의 하한
4. **특수 정렬 알고리즘**

특수 정렬 알고리즘 : O(n) Sort

- 비교 정렬
  - 이제까지 다룬 정렬 기법들은 두 원소를 비교하는 것을 기본 연산으로 하는 비교 정렬임
  - 최악의 경우 $\Theta(n \log n)$ 보다 빠를 수 없다. $O(n^2)$, $O(n \log n)$
    ➔ 즉, $n \log n$ 이 하한. $\boldsymbol{\Omega(n \log n)}$
- 그러나 원소들이 특수한 성질을 만족하면 비교 정렬이 아닌 기법으로 $O(n)$ 정렬도 가능
  - 계수 정렬(Counting Sort)
    - 원소들의 값이 $O(n)$을 넘지 않는 경우 사용 가능
  - 기수 정렬(Radix Sort)
    - 원소들이 모두 k 이하의 자리수인 경우 사용 가능 (k : 상수)

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - **계수 정렬**
  - 기수 정렬

계수 정렬(counting sort)

- 원소들의 값이 모두 $O(n)$ 범위에 있을 때 사용할 수 있는 정렬 방법
  - 또는 $-O(n) \sim O(n)$ 범위에 있을 때로 보아도 됨
- 예) A[1...n]의 원소들이 k 를 넘지 않는 자연수인 경우 (k는 상수)
  1) 배열의 원소 중에서 1부터 k 까지의 자연수가 각각 몇 번씩 나타나는지 센다.
  2) A[1...n]의 각 원소가 몇 번째 놓이면 되는지 계산해 낸다.

계수 정렬

```
countingSort(A[], B[], n)    ▷ A: 입력배열, B:정렬결과, n: 입력크기
                             ▷ A의 모든 원소는 {1, 2, 3, ..., k} 에 속함
{
    for i ← 1 to k
        C[i] ← 0;
    for j ← 1 to n
        C[A[j]]++;
    ▷ 이 지점에서 C[i] : 값이 i인 원소의 총 수
    for i ← 2 to k
        C[i] ← C[i] + C[i-1];
    ▷ 이 지점에서 C[i] : i 보다 작거나 같은 원소의 총 수
    for j ← n downto 1 {
        B[C[A[j]]] ← A[j];
        C[A[j]]--;
    }
}
```

	1	2	3	4	5	6
A	1	2	1	1	3	3

	1	2	3
C			

	1	2	3	4	5	6
B						

✓ 수행시간 : $\Theta(n)$
단, $k = O(n)$ 인 경우에 한함

문제

- 배열 A의 원소들을 counting sort를 이용하여 오름차순 정렬하세요. 단, 모든 원소는 {1, 2, ⋯ , 8}에 속함. 즉, k=8

A

8	2	5	1	5	7	6	5	1	4

C

B

정렬 알고리즘

- 기본적인 정렬 알고리즘 – 평균 $\Theta(n^2)$
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬
- 고급 정렬 알고리즘 – 평균 $\Theta(n\log n)$
  - 병합 정렬
  - 퀵 정렬
  - 힙 정렬
- 특수 정렬 알고리즘 – 평균 $\Theta(n)$
  - 계수 정렬
  - **기수 정렬**

기수 정렬(radix sort)

- 원소들의 값이 모두 k 자릿수 이하의 자연수인 특수한 경우에 사용할 수 있는 정렬 방법
  - 자연수가 아닌 제한된 종류를 가진 알파벳 등도 해당
- 예) A[1...n]의 원소들이 4자리 이하 자연수인 경우
  1) 가장 낮은 1의 자릿수만 가지고 모든 수를 정렬한다.
  2) 그 다음으로 낮은 10의 자릿수만 가지고 모든 수를 정렬한다.
  3) 이 과정을 100, 1000 자릿수까지 반복하면 정렬된 배열을 얻는다.

기수 정렬

radixSort(A[ ], k)
▷ 원소들이 각각 최대 k 자리수인 A[1…n]을 정렬한다
▷ 가장 낮은 자리수를 1번째 자리수라 한다
{
    for $i \leftarrow 1$ to k
        i 번째 자리수에 대해 A[1…n] 을 안정 정렬한다; --------- ①
}

✓ **안정 정렬(stable sort)**

—같은 값을 가진 원소들 간에 정렬 후에도 원래의 순서가 유지되도록 하는 정렬

예) 정렬 전 : 8 3 5 7 5' 6
정렬 후 : 3 5 5' 6 7 8

✓ 수행시간 : $O(n)$

①에서 각 digit에 대해 계수 정렬을 이용하면 O(n)시간이 걸리고, digit 개수 k 는 상수이므로 $k \times O(n) = O(n)$

radixSort 작동 예

0123	1560	0004	0004	0004
2154	2150	0222	1061	0123
0222	1061	0123	0123	0222
0004	0222	2150	2150	0283
0283	0123	2154	2154	1061
1560	0283	1560	0222	1560
1061	2154	1061	0283	2150
2150	0004	0283	1560	2154

문제

- 10진수 3자리수로 제한된 다음 원소들을 radix sort를 이용하여 오름차순 정렬하세요.

123			
871			
109			
205			
176			

요약

- 특수 정렬 알고리즘
  - 계수 정렬
  - 기수 정렬

➔ 두 원소의 비교에 근거하지 않으므로 선형 시간에 정렬 가능하다. 즉, $\Theta(n)$

정렬 알고리즘의 효율성 비교

	worst case	average case	
selection sort	n^2	n^2	
bubble sort	n^2	n^2	**n**이 작을 때 유용
insertion sort	n^2	n^2	
merge sort	$n \log n$	$n \log n$	평균/최악 모두 시간 복잡도 낮음
quick sort	n^2	$n \log n$	가장 많이 사용됨 – 최악의 경우는 회피가능
heap sort	$n \log n$	$n \log n$	힙이라는 자료구조를 기반으로 함
counting sort	n	n	입력 값의 범위에 제한
radix sort	n	n	입력 값의 자리수에 제한