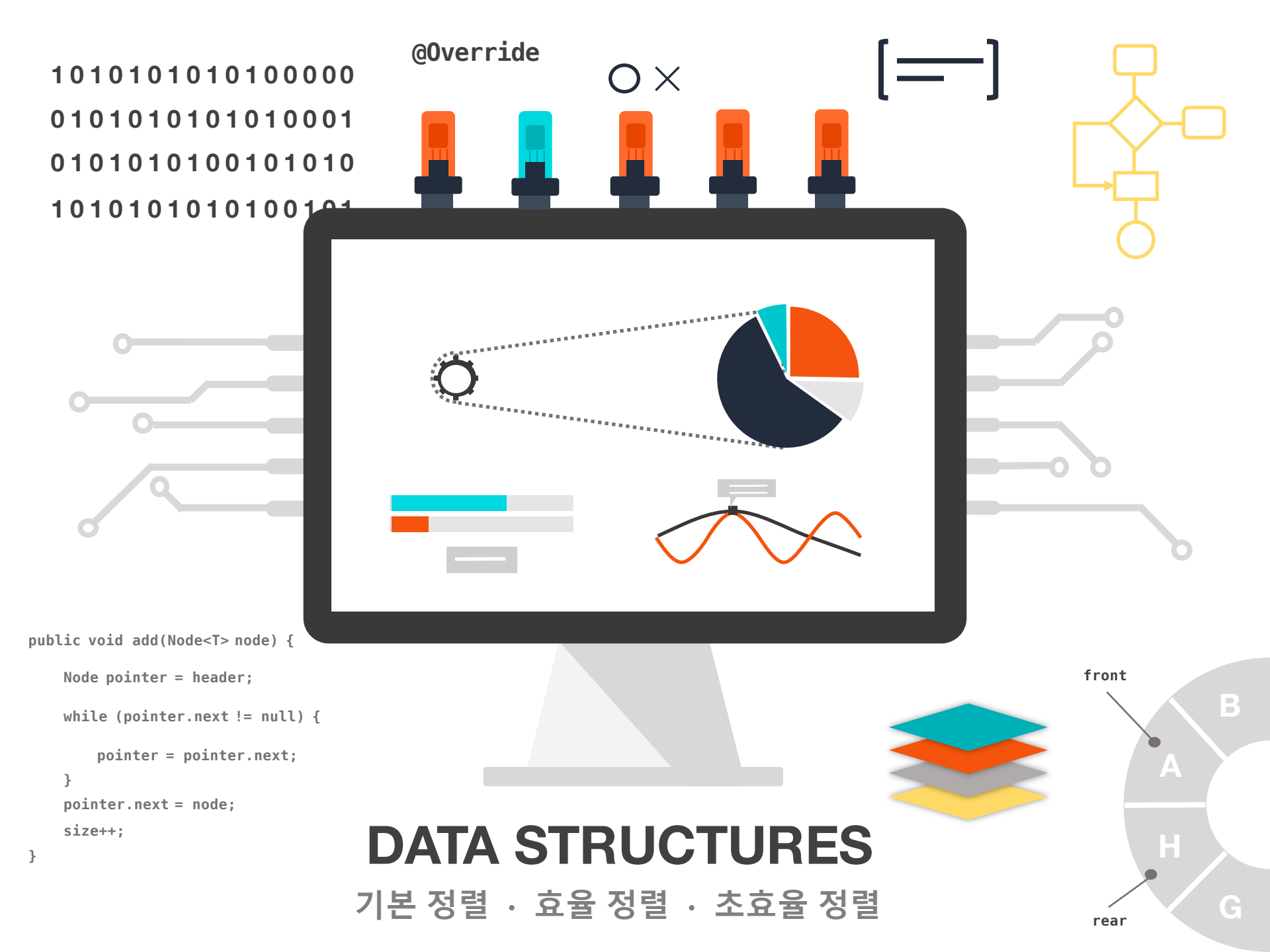


DATA STRUCTURES

기본 정렬 · 효율 정렬 · 초효율 정렬

TABLE OF CONTENTS

선택 정렬 - 개요 006

선택 정렬 - 연산 007

선택 정렬 - 코드 분석 010

선택 정렬 - 성능 분석 011

삽입 정렬 - 개요 013

삽입 정렬 - 연산 014

삽입 정렬 - 코드 분석 016

삽입 정렬 - 성능 분석 017

버블 정렬 - 개요 019

버블 정렬 - 연산 020

버블 정렬 - 코드 분석 023

버블 정렬 – 성능 분석 024

병합 정렬 – 개념 026
병합 정렬 - 수행 과정 027
병합 정렬 - 코드 분석 029
병합 정렬 - 성능 분석 031
퀵 정렬 - 개념 032
퀵 정렬 - 수행 과정 033
퀵 정렬 - 코드 분석 041
퀵 정렬 - 성능 분석 042
기수 정렬 - 개념 045
기수 정렬 - 특징 046
기수 정렬 - 수행 과정 047
기수 정렬 – 코드 분석 061

TABLE OF CONTENTS

기수 정렬 – 적용사례 063
기수 정렬 - 성능 분석 064
계수 정렬 - 개념 066
계수 정렬 - 원리 067
계수 정렬 - 특징 068
계수 정렬 - 수행 과정 069
계수 정렬 - 코드 분석 076
계수 정렬 - 성능 분석 078

TABLE OF CONTENTS

선택 정렬

개념 · 코드 분석 · 성능 분석

선택 정렬

개요

전체 원소들 중 위치에 맞는 원소를 선택해 자리를 교환하는 방식으로 정렬

선택 정렬

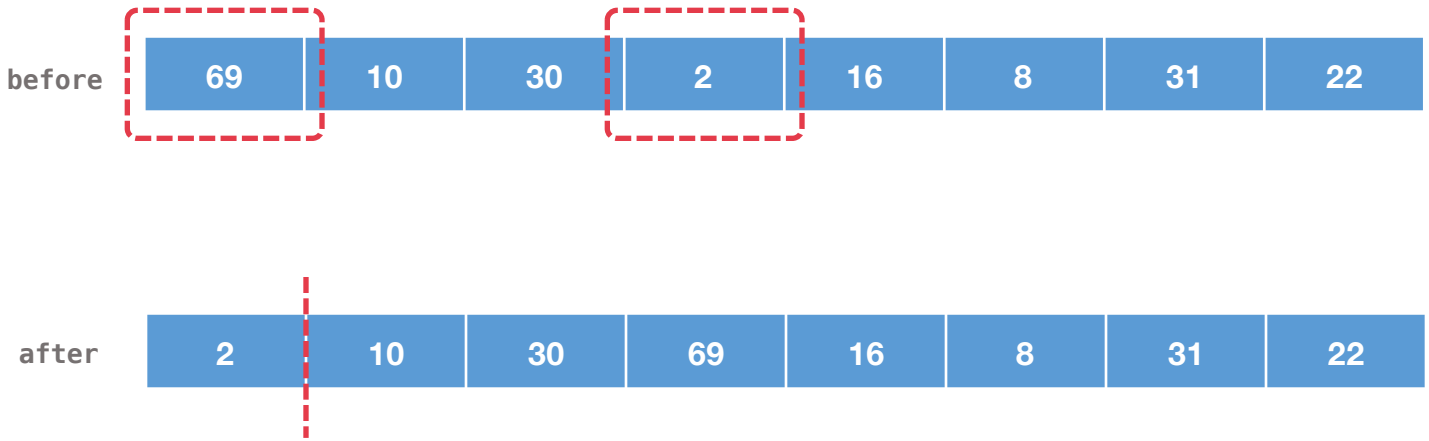
연산

1. 주어진 리스트에서 최소값을 선택

선택 정렬

연산

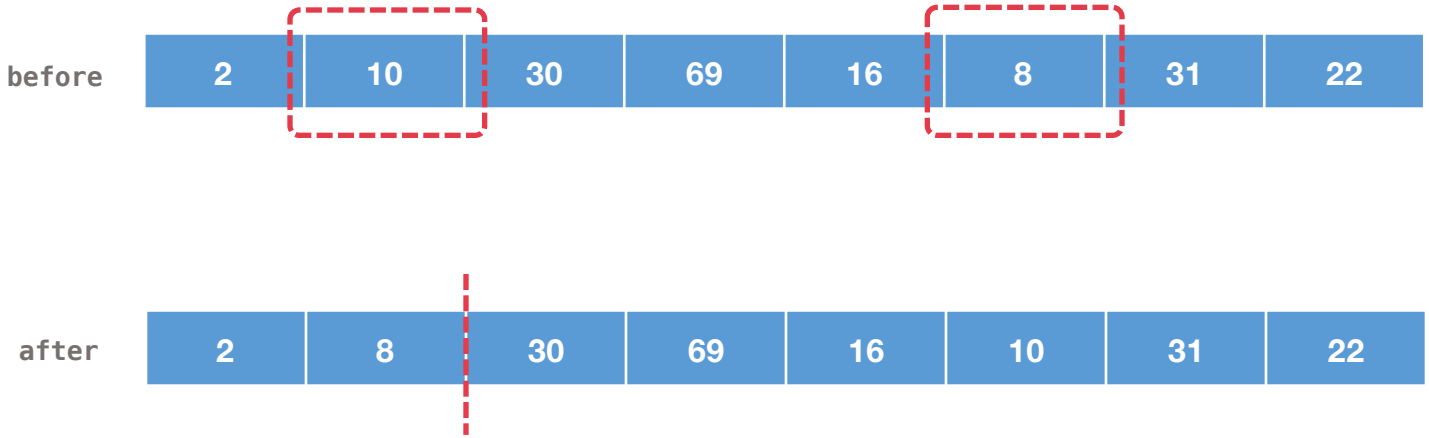
2. 그 값을 맨 앞에 위치한 값과 교체



선택 정렬

연산

3. 정렬된 원소를 제외한 나머지 리스트를 대상으로 교체를 반복



선택 정렬

코드 분석

```
public class Selection_sort<E> {
    //selection sort
    public static <E extends Comparable<E>> void
selectionSort(E[] list) {

        for(int i=0; i<list.length -1; i++) {

            int iSmallest = i;

            for(int j=i+1; j<list.length; j++) {
                if(list[iSmallest].compareTo((list[j])) > 0
) {
                    iSmallest = j;
                }
            }

            E iSwap = list[iSmallest];
            list[iSmallest] = list[i];
            list[i] = iSwap;

        }
    }
}
```

- 주어진 리스트에서 최소값을 찾는다 (isSmallest)
- 찾은 최소값을 리스트의 맨 앞의 원소와 교체
- 교체된 맨 앞의 원소를 제외한 나머지 리스트를 대상으로 반복 실행

선택 정렬

성능 분석

메모리 사용 공간

O (n) : **n** 개의 원소에 대하여 **n** 개의 메모리 사용

연산 시간

1단계 : 첫 번째 원소를 기준으로 **n** 개의 원소 비교
2단계 : 두 번째 원소를 기준으로 마지막 원소까지 **n – 1** 개의 원소 비교
3단계 : 세 번째 원소를 기준으로 마지막 원소까지 **n – 2** 개의 원소 비교

…

i 단계 : i 번째 원소를 기준으로 마지막 원소까지 **n – i** 개의 원소 비교

평균 시간 복잡도

O (n^2)

삽입 정렬

개념 · 코드 분석 · 성능 분석

삽입 정렬

개요

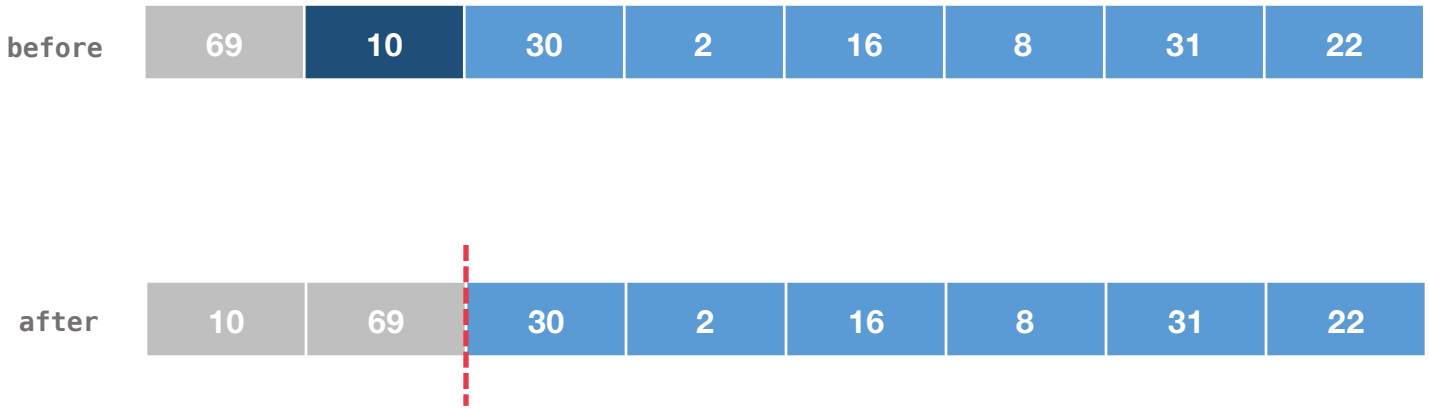
정렬되어 있는 부분집합에 새로운 원소의 위치를 찾아 삽입하는 방식으로 정렬

삽입 정렬

연산

1. 두 번째 리스트를 첫 번째 리스트와 비교하여 적절한 위치에 삽입

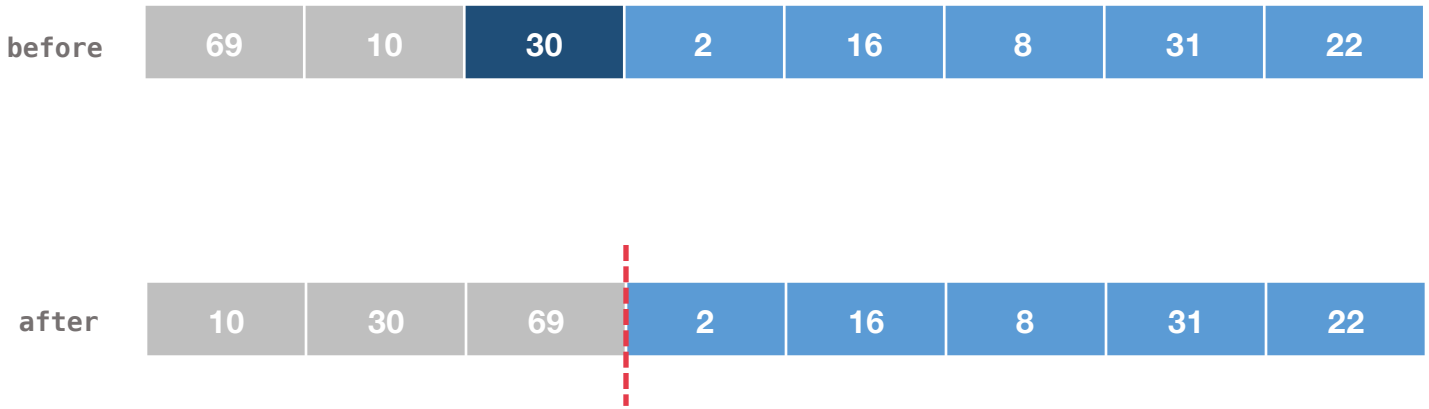
첫 번째 리스트는 하나이기 때문에 정렬이 되어있다고 가정하고 두 번째 리스트부터 시작한다



삽입 정렬

연산

2. 위와 같은 리스트가 끝날 때까지 계속해서 삽입



삽입 정렬

코드 분석

```
public class Insertion_sort<T> {

    public <T extends Comparable<T>> void
doInsertionSort(T[] input) {

        if (input == null) {
            throw new RuntimeException("Input array
cannot be null");
        }
        int length = input.length;
        if (length == 1) return;
        int i, j;
        T temp;
        for (i = 1; i < length; i++) {
            temp = input[i];

            for (j = i; (j > 0 &&
(temp.compareTo(input[j - 1]) < 0)); j--) {

                input[j] = input[j - 1];
            }

            input[j] = temp;
        }

    }
}
```

- 두번째 원소 부터 앞의 정렬된 리스트와 비교하여 적절한 자리에 삽입
  정렬하고자 하는 원소를 temp에 임의 저장
- 삽입 된 후 다음 원소(세번째 원소)를 대상으로 반복 실행

삽입 정렬

성능 분석

메모리 사용 공간

O (n) : **n** 개의 원소에 대하여 **n** 개의 메모리 사용

연산 시간

1. 최선의 경우

   원소들이 이미 정렬되어 있어서 비교 횟수가 최소인 경우, 바로 앞자리 원소와 한 번만 비교한다
   전체 비교 횟수는 **n – 1** 이며, 시간 복잡도는 **O (n)**이다

2. 최악의 경우

   모든 원소가 역순으로 되어있는 경우 비교 횟수가 최대이므로
   전체 비교 횟수는 **n (n – 1) / 2** 이며, 시간 복잡도는 **O (n^2)**이다

평균 시간 복잡도

O (n^2) : 평균 비교 횟수는 **n (n-1) / 4**

버블 정렬

개념 · 코드 분석 · 성능 분석

버블 정렬

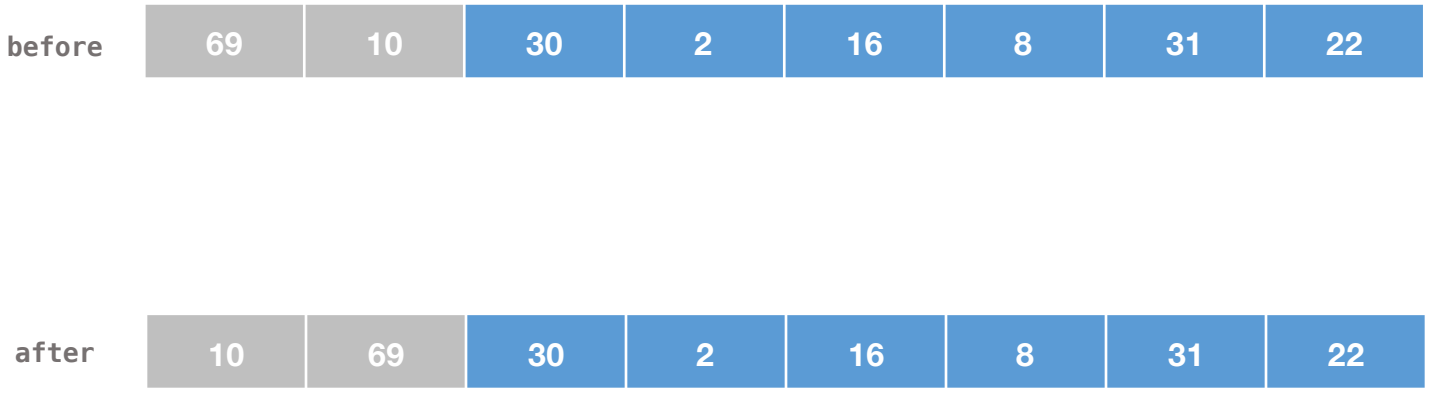
개요

인접한 두 개의 원소를 비교하여 자리를 교환하는 정렬

버블 정렬

연산

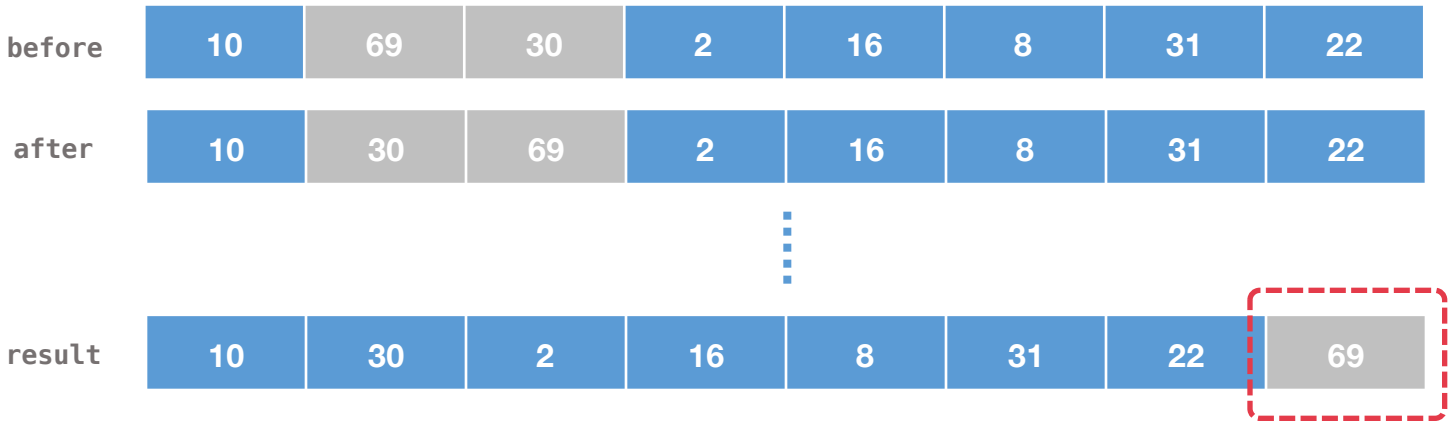
1. 인접한 두 인덱스를 비교해서 정렬이 되어있지 않을 경우 정렬



버블 정렬

연산

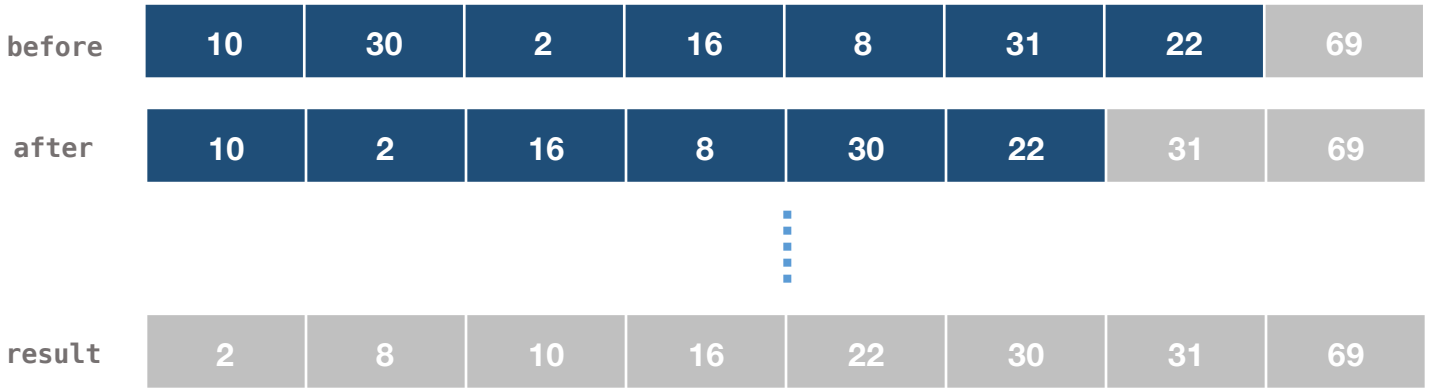
2. 1을 반복하여 가장 큰 원소를 가장 마지막으로 정렬



버블 정렬

연산

3. 마지막에 정렬한 요소를 제외한 나머지 리스트로 위 과정을 반복



버블 정렬

코드 분석

```
public class Bubble_sort<E> {

    public static <E> void bubbleSort(E[] unsorted) {

        for(int iter =1; iter< unsorted.length; iter++){

            for(int inner = 0; inner < unsorted.length - iter; inner
++){


if((((Comparable)(unsorted[inner])).compareTo(unsorted[inner+1])) >
0){

                    E tmp = unsorted[inner];
                    unsorted[inner] = unsorted[inner + 1];
                    unsorted[inner + 1] = tmp;

                }
            }
        }
    }
}
```

- 리스트의 첫 원소부터 바로 인접한 다음 원소와 비교
- 두 원소를 비교하여 더 큰 원소를 temp 에 저장하고 둘 중 뒤에 배열
- 같은 방법으로 가장 큰 수가 리스트의 가장 마지막에 정렬될 때까지 반복 실행

- 정렬 된 리스트를 제외한 나머지를 대상으로 반복 실행

버블 정렬

성능 분석

메모리 사용 공간

O (n) : **n** 개의 원소에 대하여 **n** 개의 메모리 사용

연산 시간

1. 최선의 경우 : 자료가 이미 정렬되어 있음
   전체 비교 횟수는 **n (n – 1) / 2** 이며, 자리 교환 횟수는 없다

2. 최악의 경우 : 자료가 역순으로 정렬되어 있음
   전체 비교 횟수는 **n (n – 1) / 2** 이며, 자리 교환 횟수 또한 **n (n – 1) / 2** 이다

평균 시간 복잡도

O (n^2)

병합 정렬

개념 · 코드 분석 · 성능 분석

병합 정렬

개념

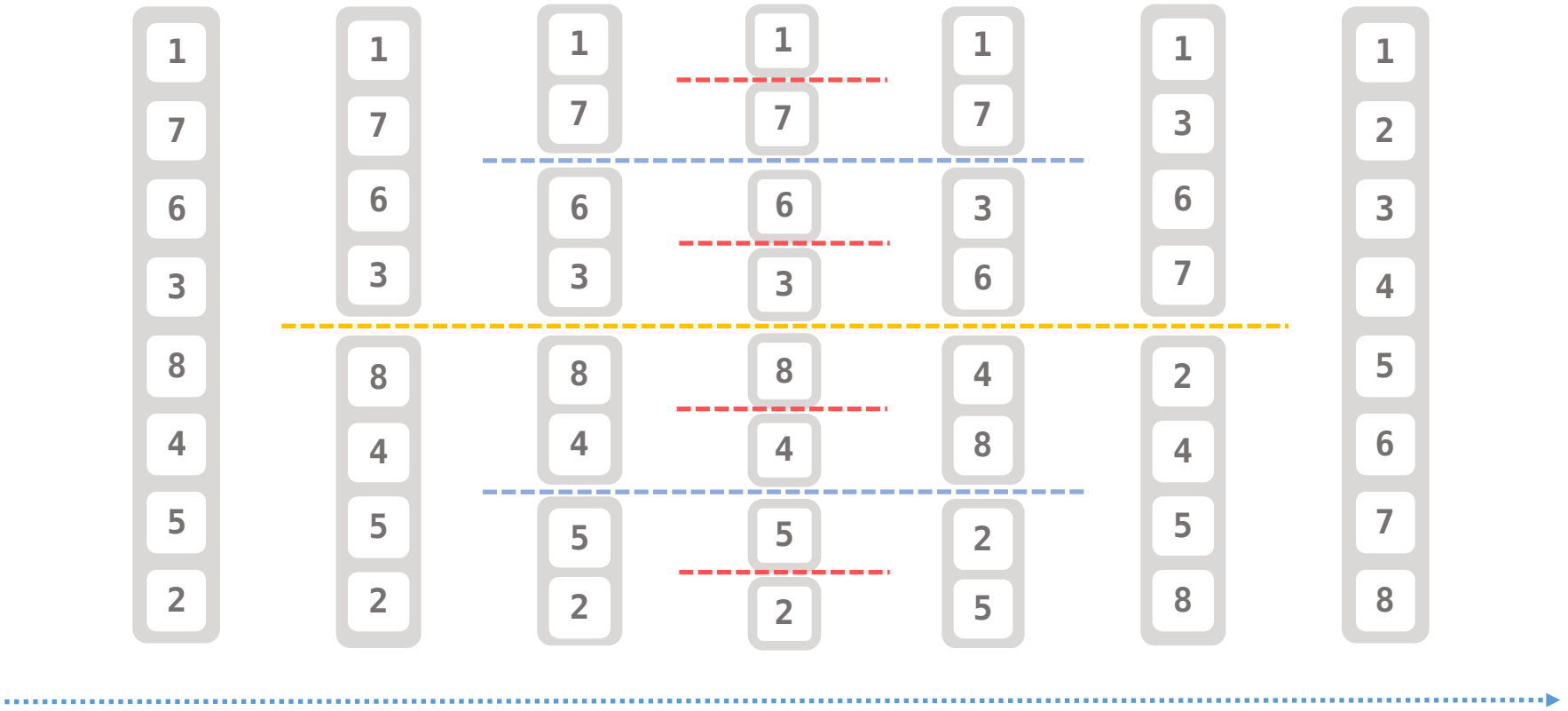
- **여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 정렬 방식이다**

- **분할 정복 기법을 사용한다**

  집합을 분할하고 각 부분 집합에 대해 정렬 작업을 완성한 후 다시 병합하는 과정 반복

병합 정렬

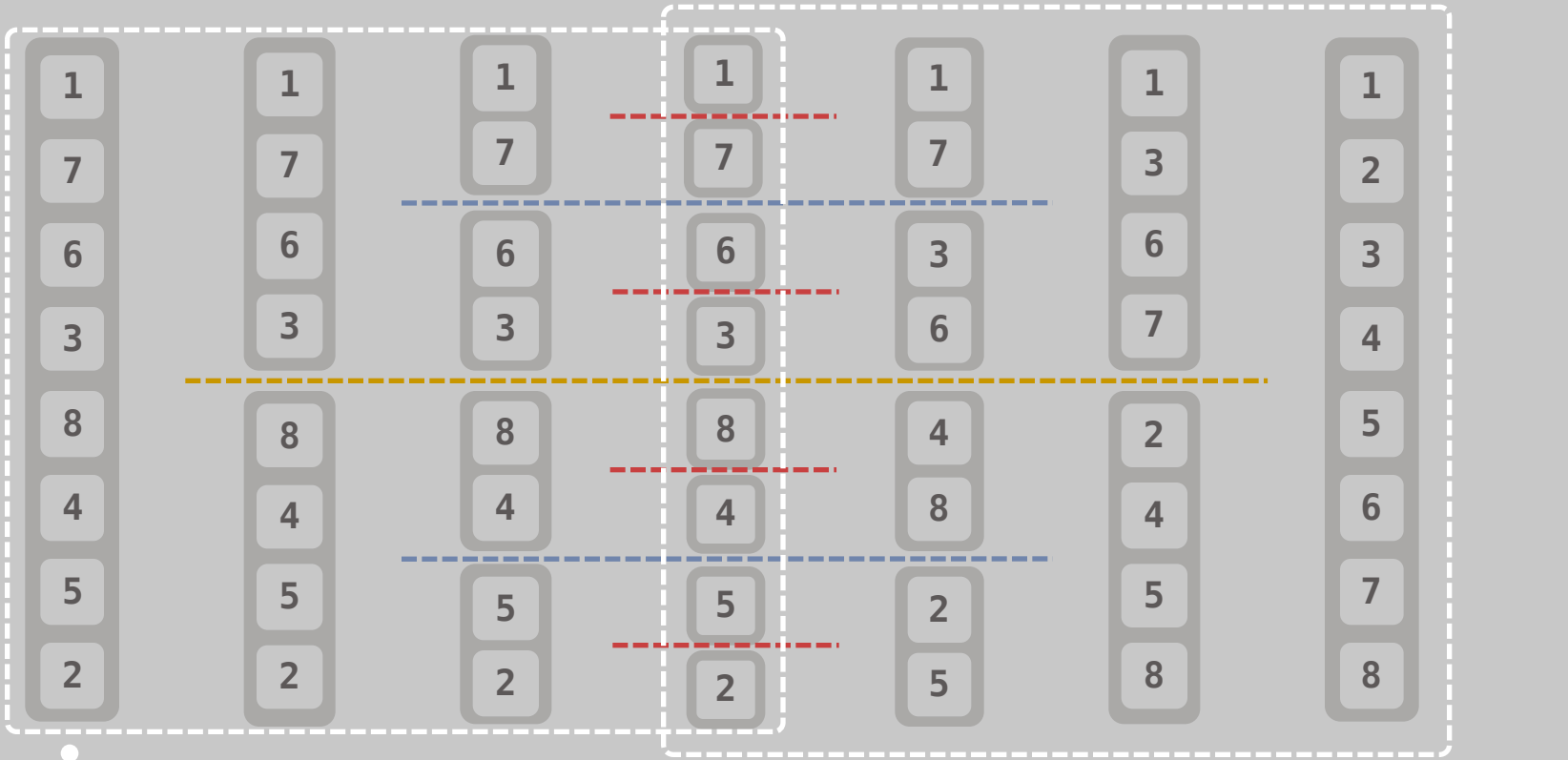
수행 과정


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

병합 정렬

수행 과정

2. 병합 : 2개의 부분집합을 정렬하면서 하나의 집합으로 병합하며, 모든 요소를 병합할 때까지 반복



1. 분할 : 전체 자료의 집합에 대해 최소 원소의 부분집합이 될 때까지 작업 반복

병합 정렬

코드 분석

```
public class MergeSort<T> {

    public <T extends Comparable<T>> void MergeSort(T[] a) {
        MergeSort(a, 0, a.length - 1);
    }

    public <T extends Comparable<T>> void MergeSort(T[] a,
int left, int right){
        if (right - left < 1) return;
        int mid = (left + right) / 2;

        MergeSort(a, left, mid);
        MergeSort(a, mid + 1, right);

        merge(a, left, mid, right);
    }
```

- 최소 원소를 가질 때 까지 두 부분으로 분할

병합 정렬

코드 분석

```
public <T extends Comparable<T>> void merge(T[] a, int left,
int mid, int right) {

        Object[] tmp = new Object[right - left + 1];
        int L = left;
        int R = mid + 1;
        int S = 0;

        while (L <= mid && R <= right) {
            if (a[L].compareTo(a[R]) <= 0)
                tmp[S] = a[L++];
            else
                tmp[S] = a[R++];
            S++;
        }

        if (L <= mid && R > right) {
            while (L <= mid)
                tmp[S++] = a[L++];
        } else {
            while (R <= right)
                tmp[S++] = a[R++];
        }

        for (S = 0; S < tmp.length; S++) {
            a[S + left] = (T) (tmp[S]);
        }

    }
}
```

- 왼쪽, 오른쪽 파트 중 한 쪽이 다 정렬 될 때까지 부분 집합의 원소를 비교하며 병합
- 왼쪽파트가 먼저 정렬이 끝난 경우
- 오른쪽파트가 먼저 정렬이 끝난 경우
- 데이터 이동

병합 정렬

성능 분석

메모리 사용 공간

O (n) : **n** 개의 원소에 대하여 **n** 개의 메모리 사용, 경우에 따라 **log (n)** 개의 추가 메모리 필요

연산 시간

1. 분할 : **n** 개의 원소에 대해 평균적으로 **log (n)** 번 분할 수행
2. 병합 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 **n** 번의 비교연산 수행

평균 시간 복잡도

O (n log n)

퀵 정렬

개념 · 코드 분석 · 성능 분석

퀵 정렬

개념

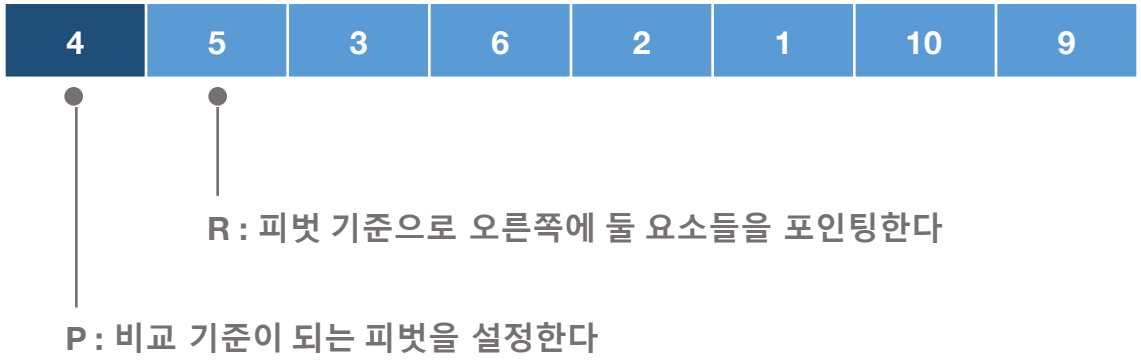
- 평균적으로 매우 빠른 수행 속도를 자랑하는 정렬 방식이다
- 분할 정복 기법을 사용한다
- 피봇 (= 기준 값)을 중심으로 왼쪽 · 오른쪽 집합으로 분할하고 정렬한다
- 왼쪽 부분 집합에는 피봇보다 작은 값을, 오른쪽 부분 집합에는 피봇보다 큰 값을 넣는다
- 피봇은 일반적으로 가운데에 위치한 원소를 선택한다

퀵 정렬

수행 과정

퀵 정렬

수행 과정

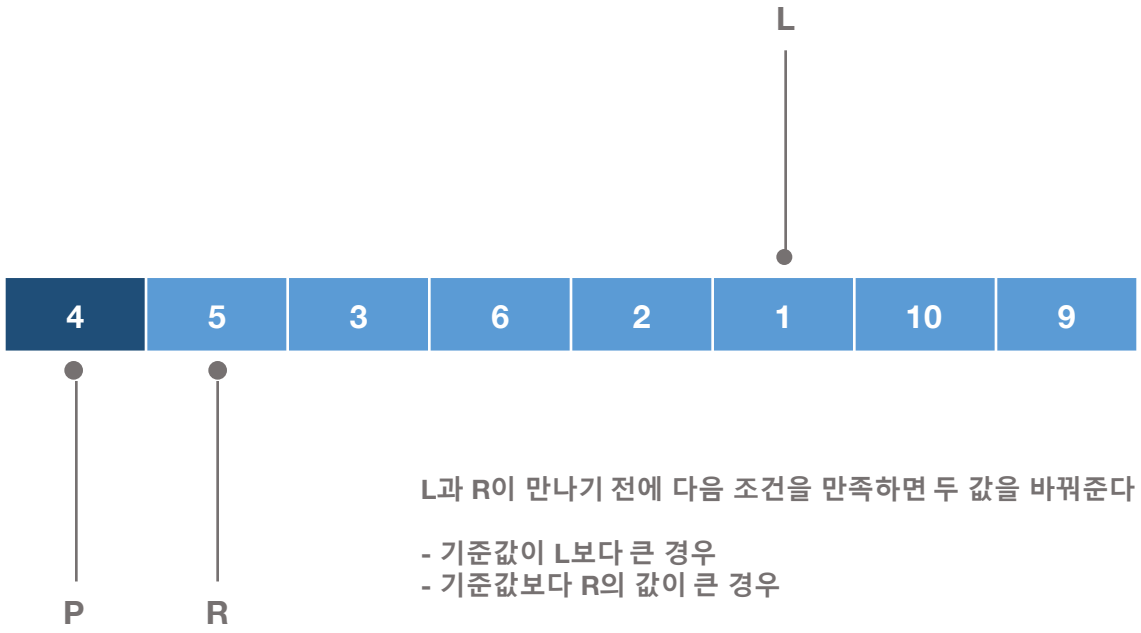

L : 피벗 기준으로 왼쪽에 둘 요소들을 포인팅한다
4
5
3
6
2
1
10
9
R : 피벗 기준으로 오른쪽에 둘 요소들을 포인팅한다
P : 비교 기준이 되는 피벗을 설정한다

퀵 정렬

수행 과정

퀵 정렬

수행 과정

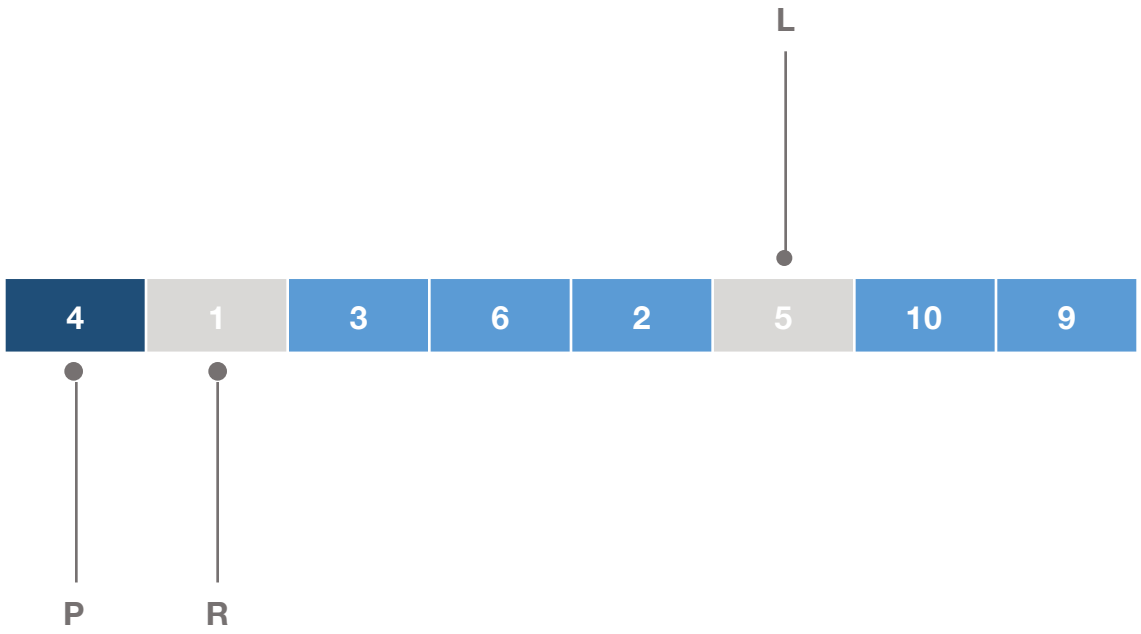


L과 R이 만나기 전에 다음 조건을 만족하면 두 값을 바꿔준다

- 기준값이 L보다 큰 경우
- 기준값보다 R의 값이 큰 경우

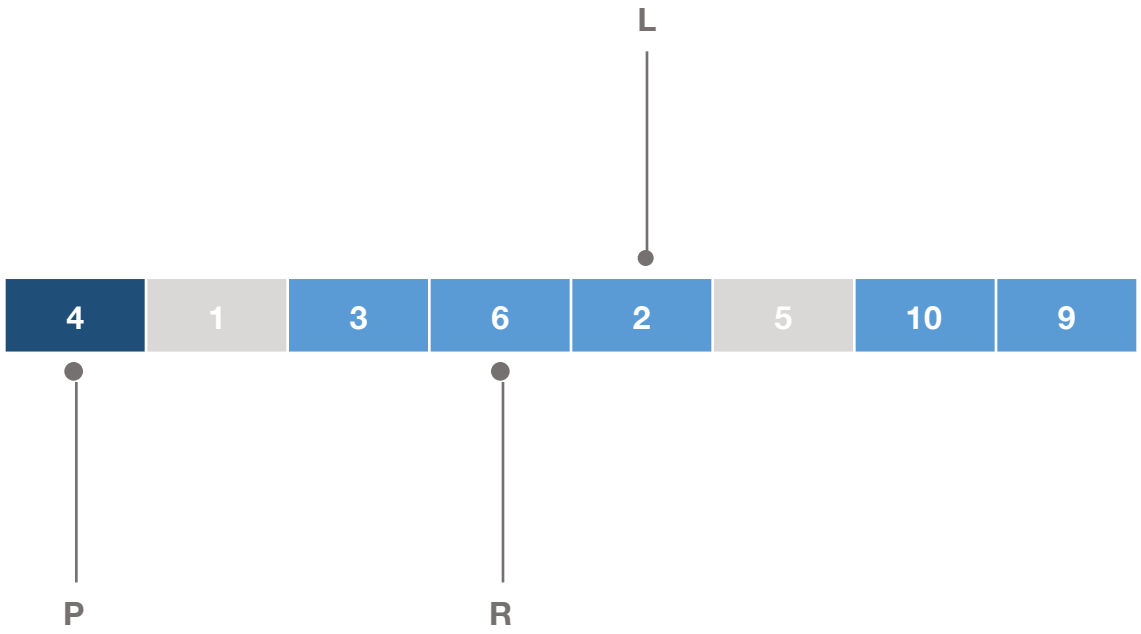
퀵 정렬

수행 과정



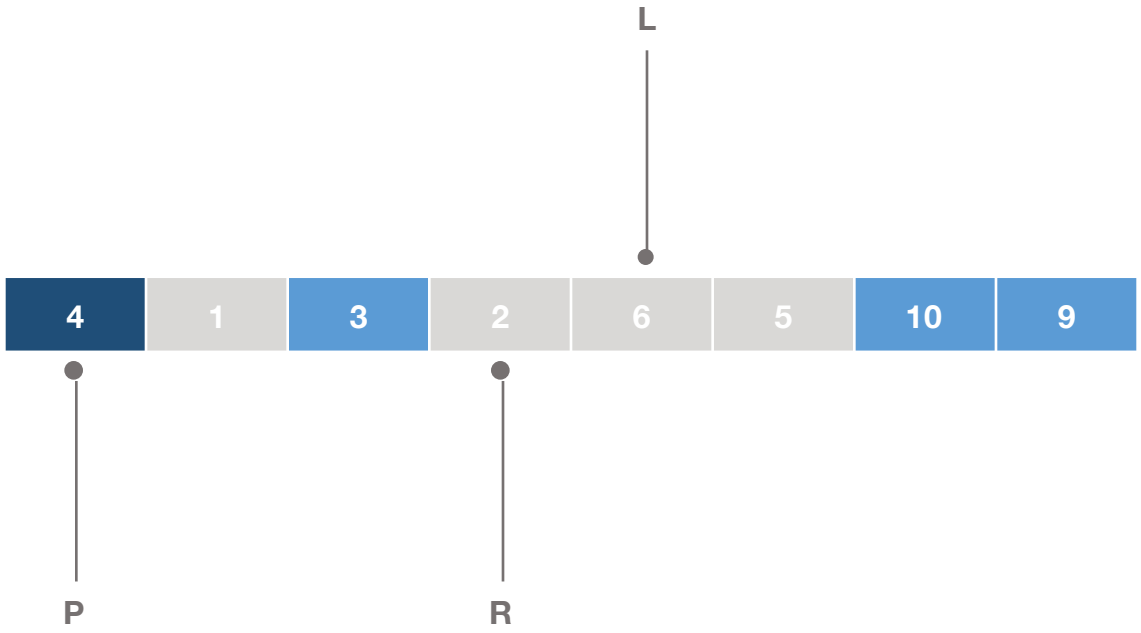
퀵 정렬

수행 과정



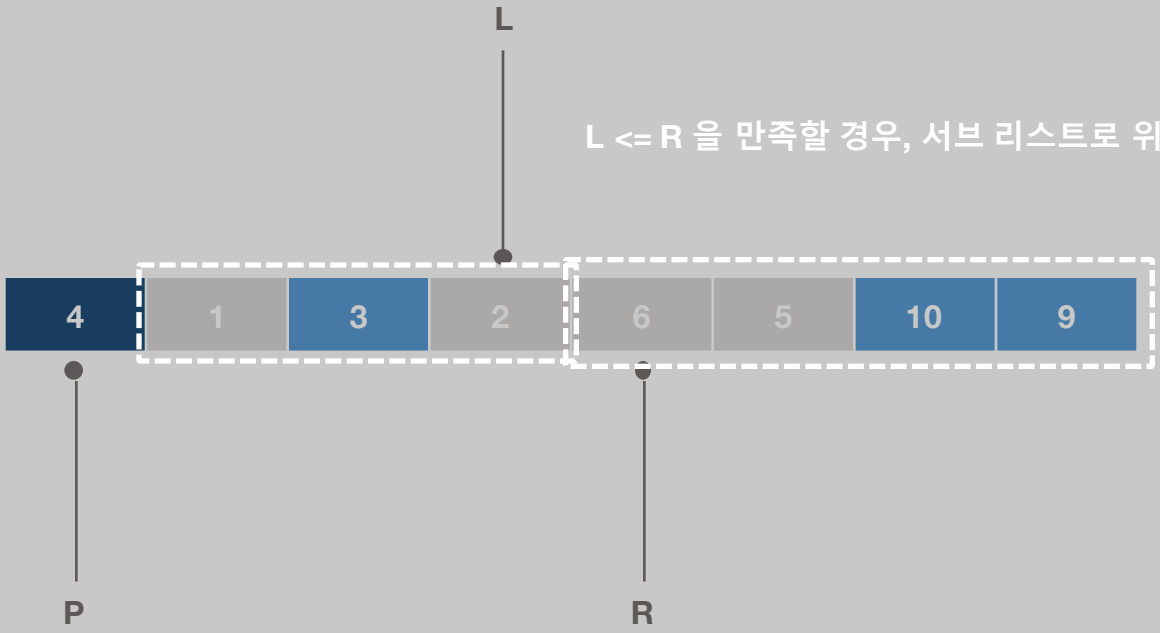
퀵 정렬

수행 과정



퀵 정렬

수행 과정



퀵 정렬

코드 분석

```
public class QuickSort<T> {

    public  <T extends Comparable<T>> void QuickSort(T[] array) {
        quick_sort(array, 0, array.length - 1);
    }

    public static <T extends Comparable<T>> void quick_sort(T[]
arr, int left, int right) {

        if (left < right) {
            int L = left, R = right;
            T pivot = arr[(L + R) / 2];

            do {
                while (arr[L].compareTo(pivot) < 0) L++;

                while (pivot.compareTo(arr[R]) < 0) R--;

                if (L <= R) {
                    T tmp = arr[L];
                    arr[L] = arr[R];
                    arr[R] = tmp;
                    L++;
                    R--;
                }

            } while (L <= R);

            quick_sort(arr, left, R);
            quick_sort(arr, L, right);
        }
    }
}
```

- Pivot 값을 임의 지정
- L은 오른쪽 이동하며 Pivot보다 크거나 같은 값 찾기
- R은 왼쪽 이동하며 Pivot 보다 작은 값 찾기
- L과 R이 가르키는 값 바꾸기
- 서브 리스트로 위 과정을 반복

퀵 정렬

성능 분석

메모리 사용 공간

O (n) : **n** 개의 원소에 대하여 **n** 개의 메모리 사용

연산 시간

1. 최선의 경우 : 피봇에 대해 원소들이 정확히 n / 2 크기의 2개의 집합으로 나눠짐

   전체 비교 횟수는 **n – 1** 이다

2. 최악의 경우 : 자료가 역순으로 정렬되어 있음

   전체 비교 횟수는 **n (n – 1) / 2** 이다

평균 시간 복잡도

O (n log n) : 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해 자리 교환 횟수를 줄여서 더 성능이 좋음

기수 정렬

개념 · 코드 분석 · 사용 사례 · 성능 분석

기수 정렬

개념

- 원소의 키 값을 나타내는 기수를 이용한 정렬 방법이다
- Bucket을 사용하여 원소를 정렬하며 주로 큐를 사용한다
- 정렬할 원소의 키 값에 해당하는 Bucket에 원소를 분배하고 Bucket의 순서대로 꺼내 정렬한다

기수 정렬

특징

장점

- 정렬 방식의 이론적 하한선 - O (n log n)의 성능을 능가하는 효율적인 정렬
- 안정적인 정렬이다 - 키 값이 같은 원소의 순서에 대해 정렬 후에도 순서가 유지된다

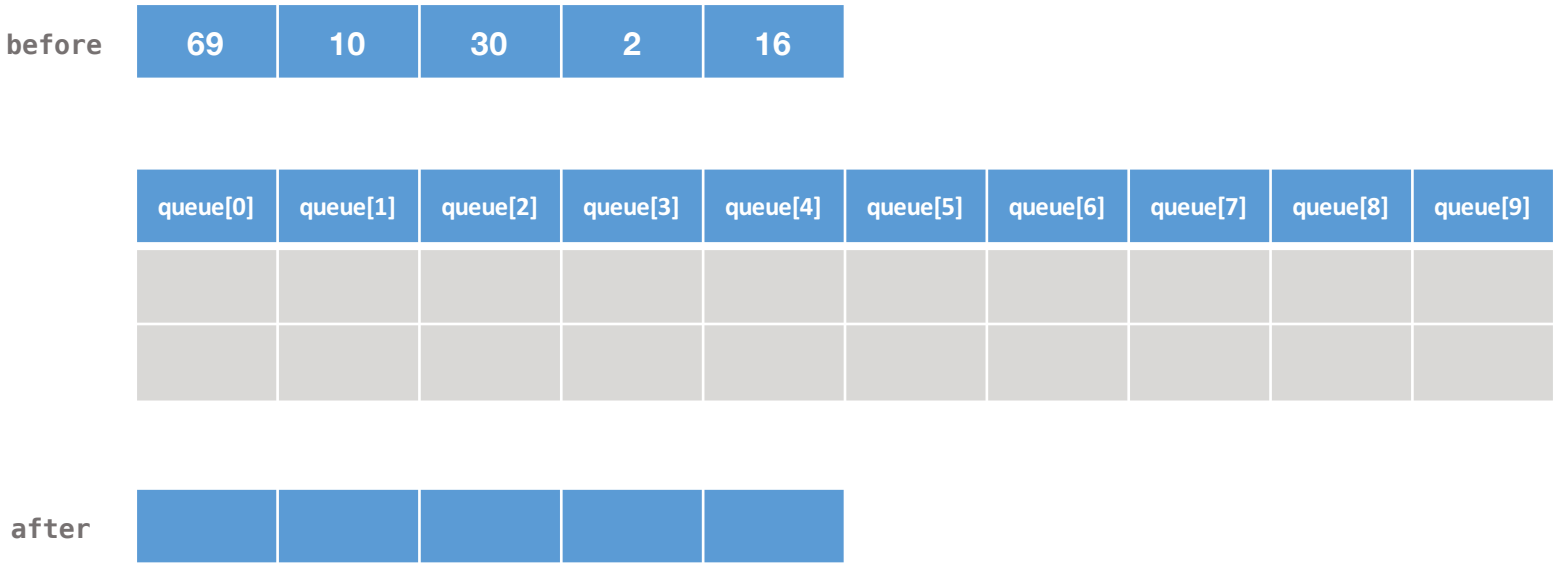
단점

- 버켓을 구현하기 위한 별도의 메모리를 필요로 한다
- 정렬할 수 있는 데이터의 타입이 한정되어 있다

기수 정렬

수행 과정

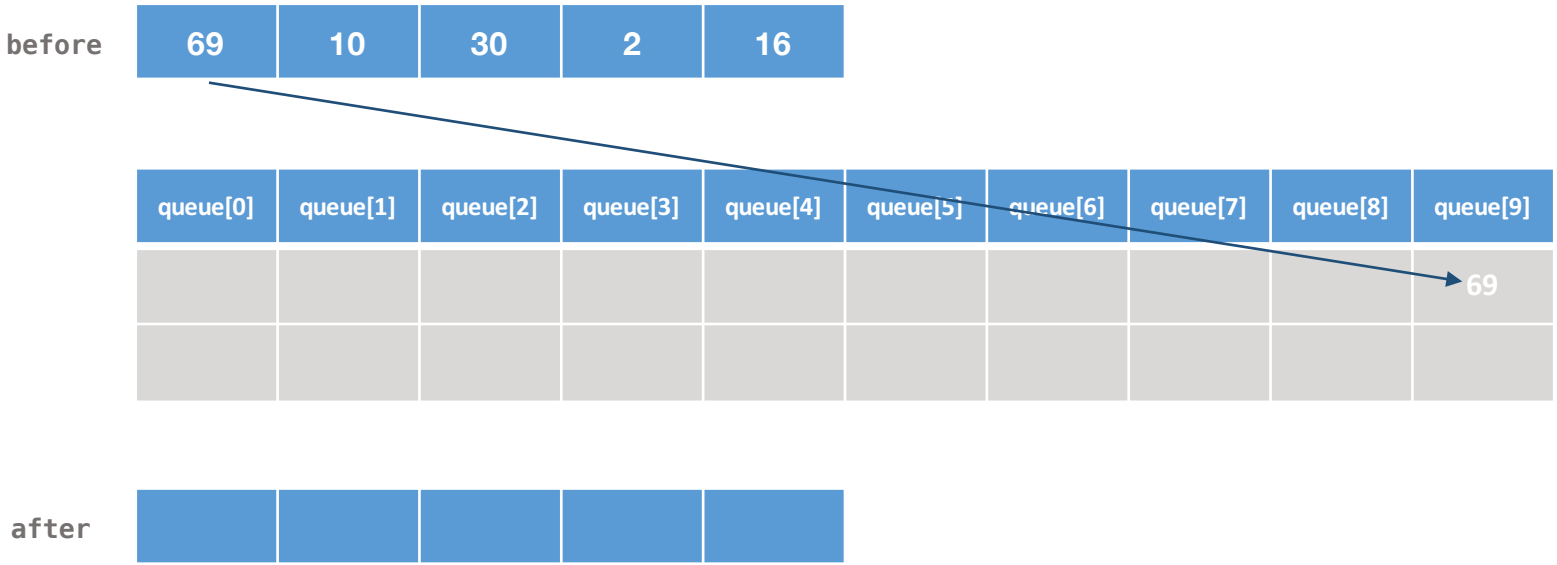
1. 원소의 키 값에 해당하는 큐에 원소를 분배



기수 정렬

수행 과정

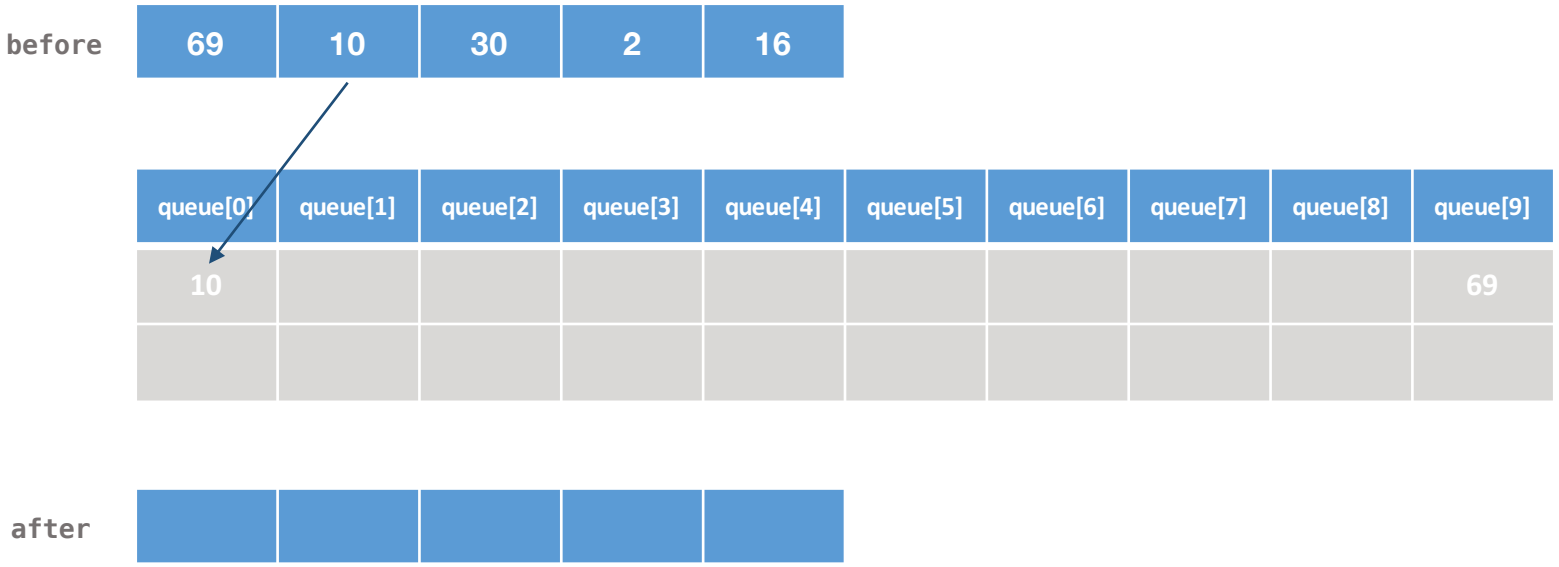
1. 원소의 키 값에 해당하는 큐에 원소를 분배



기수 정렬

수행 과정

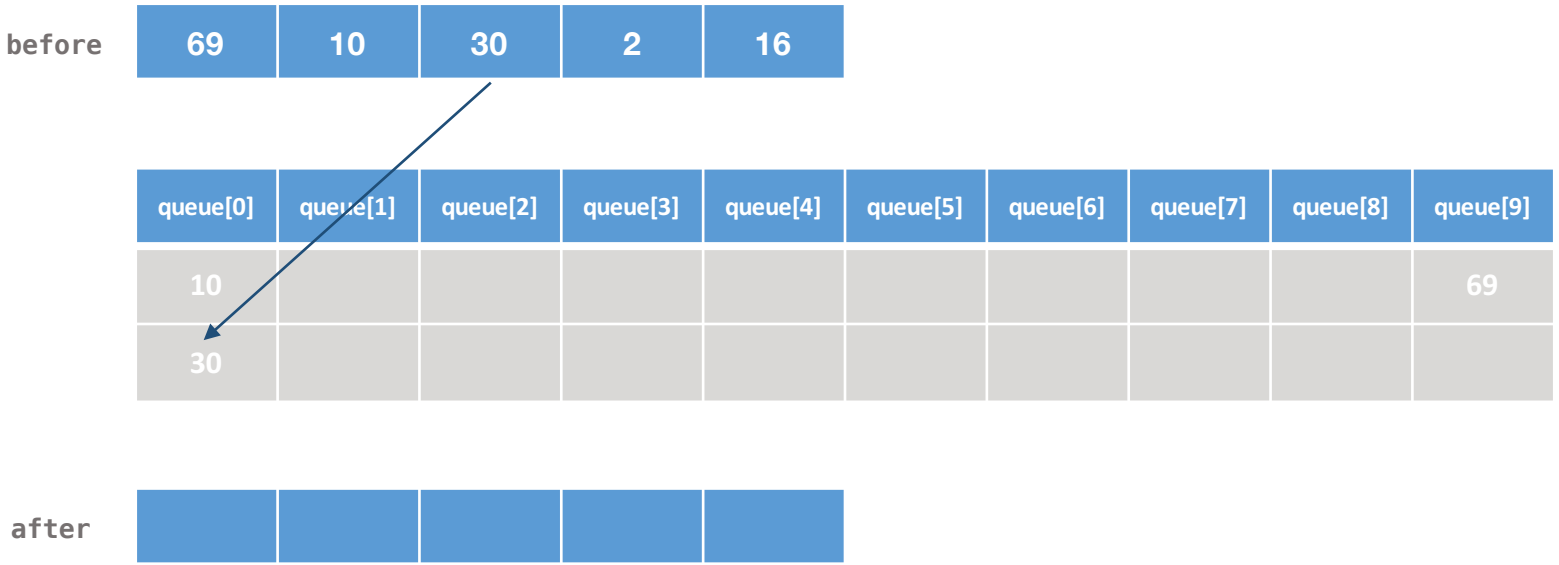
1. 원소의 키 값에 해당하는 큐에 원소를 분배



기수 정렬

수행 과정

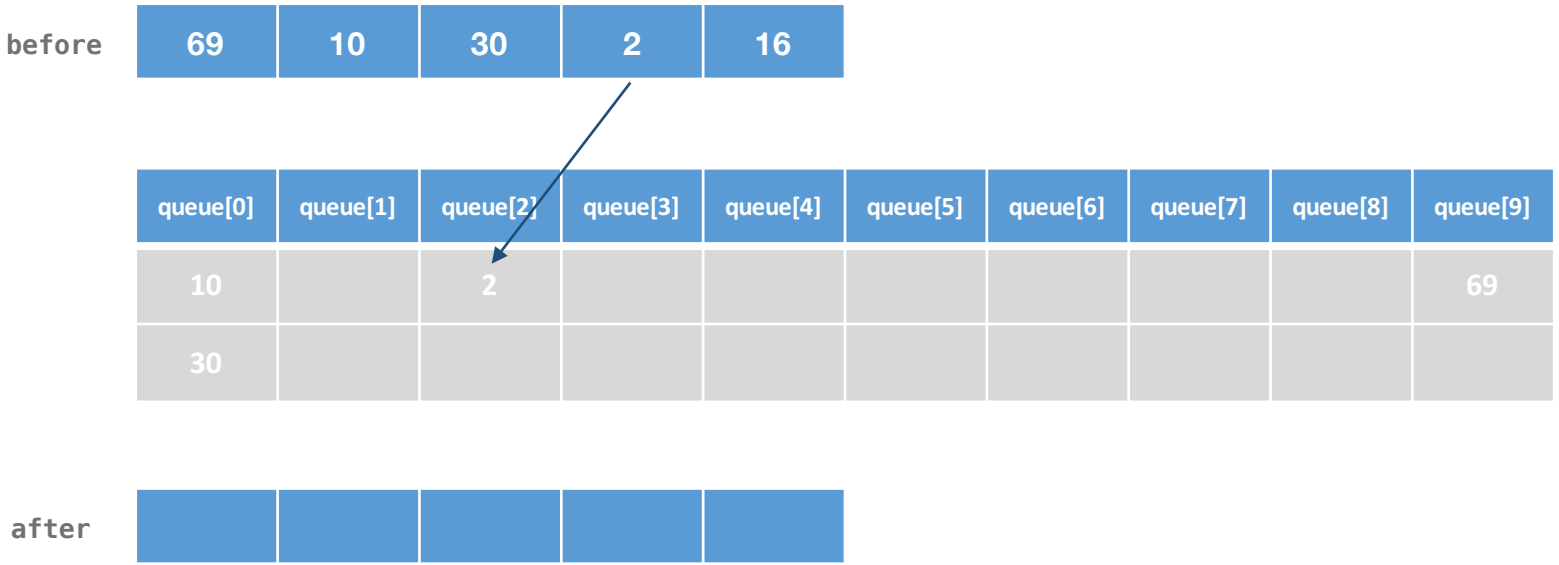
1. 원소의 키 값에 해당하는 큐에 원소를 분배



기수 정렬

수행 과정

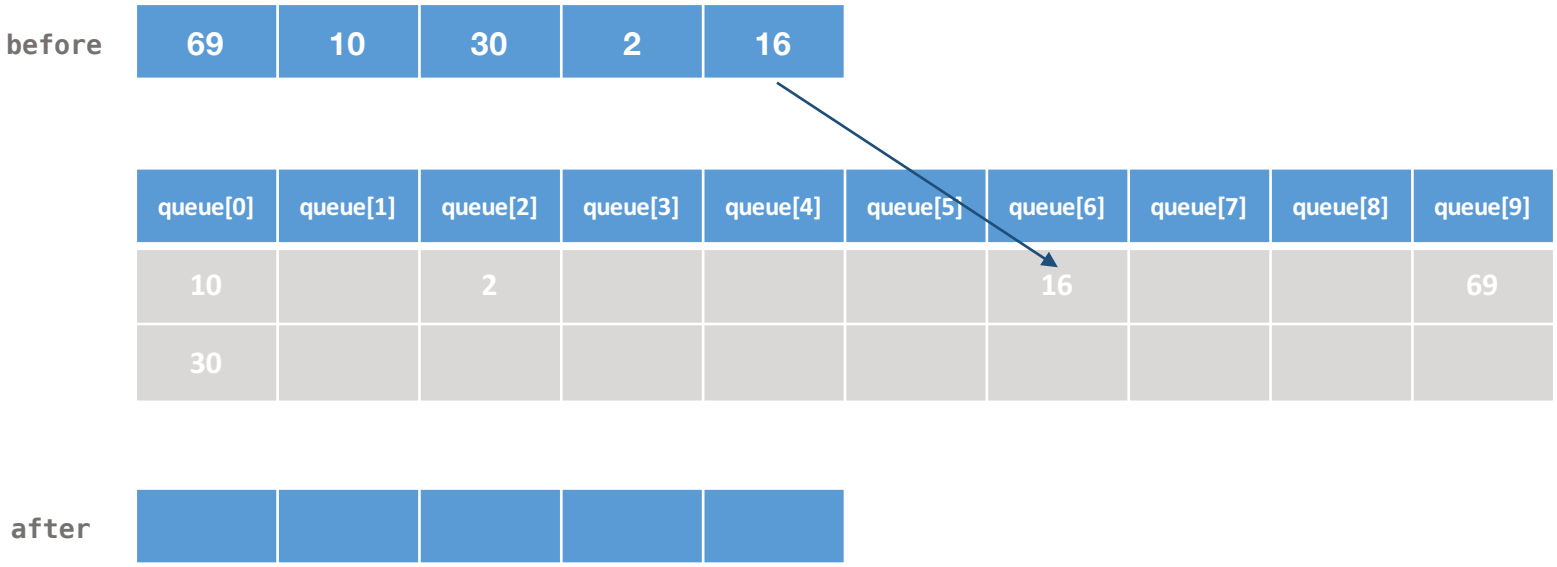
1. 원소의 키 값에 해당하는 큐에 원소를 분배



기수 정렬

수행 과정

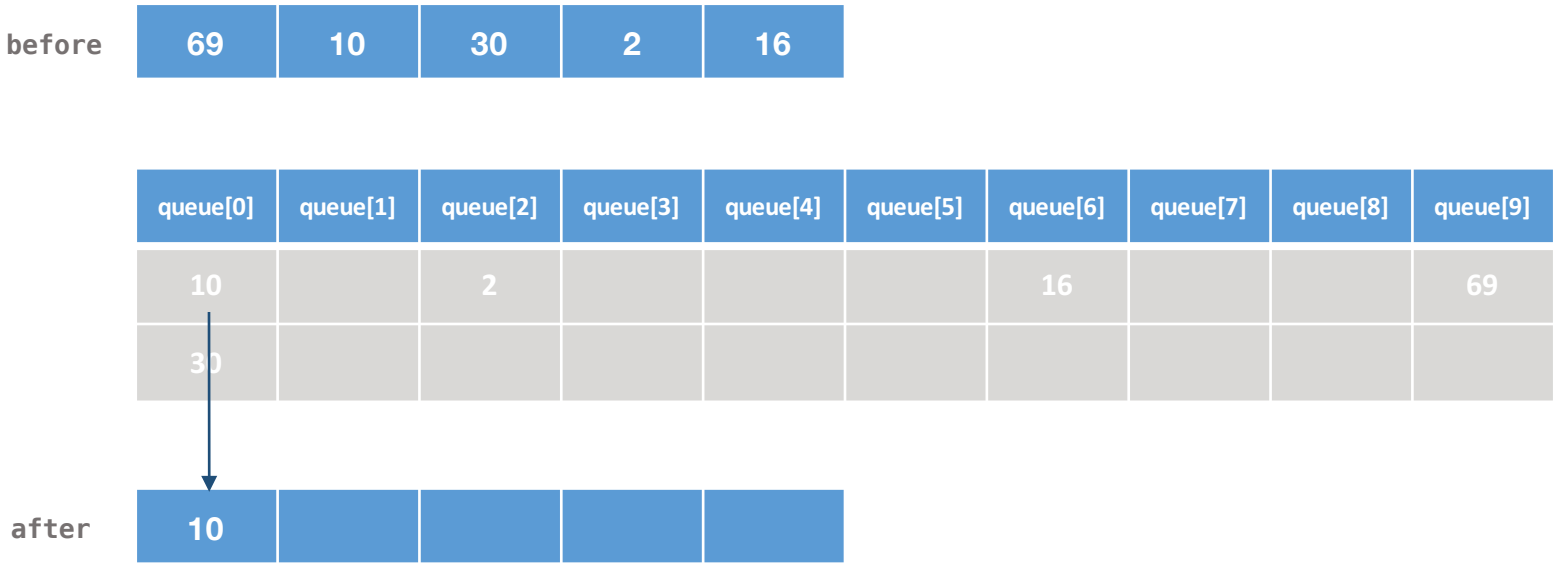
1. 원소의 키 값에 해당하는 큐에 원소를 분배



기수 정렬

수행 과정

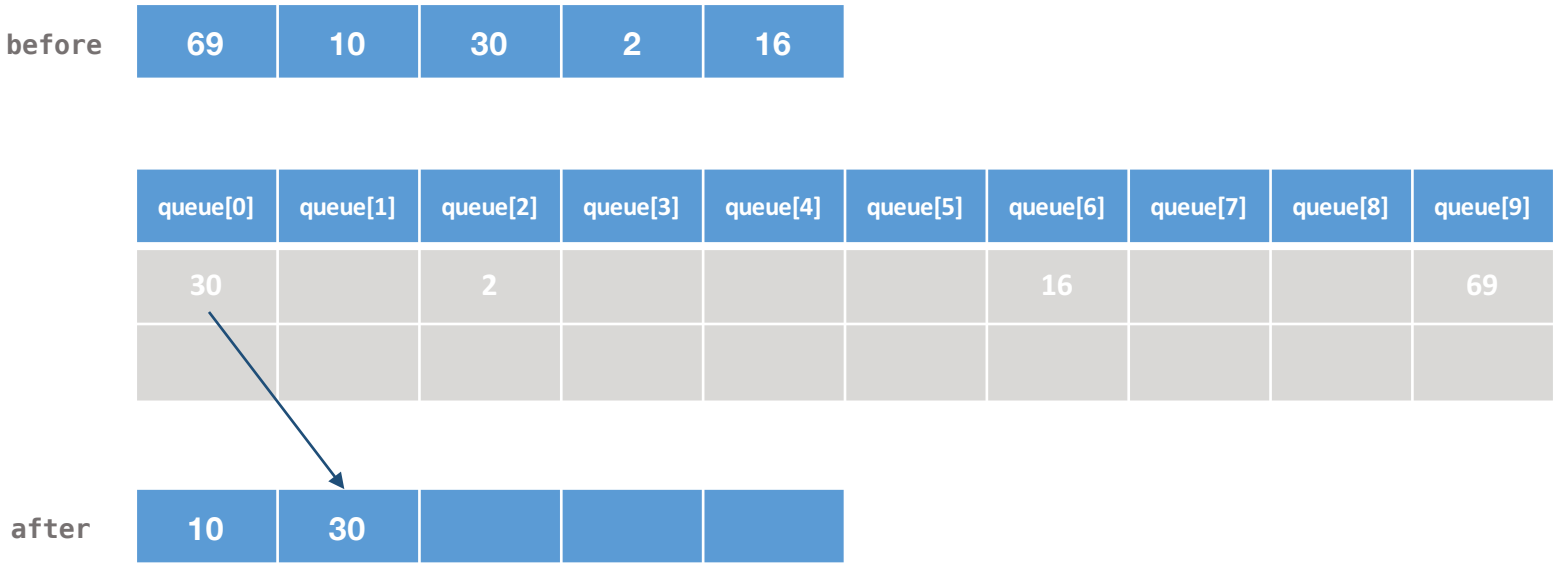
2. 버켓의 순서대로 원소를 꺼내 재분배



기수 정렬

수행 과정

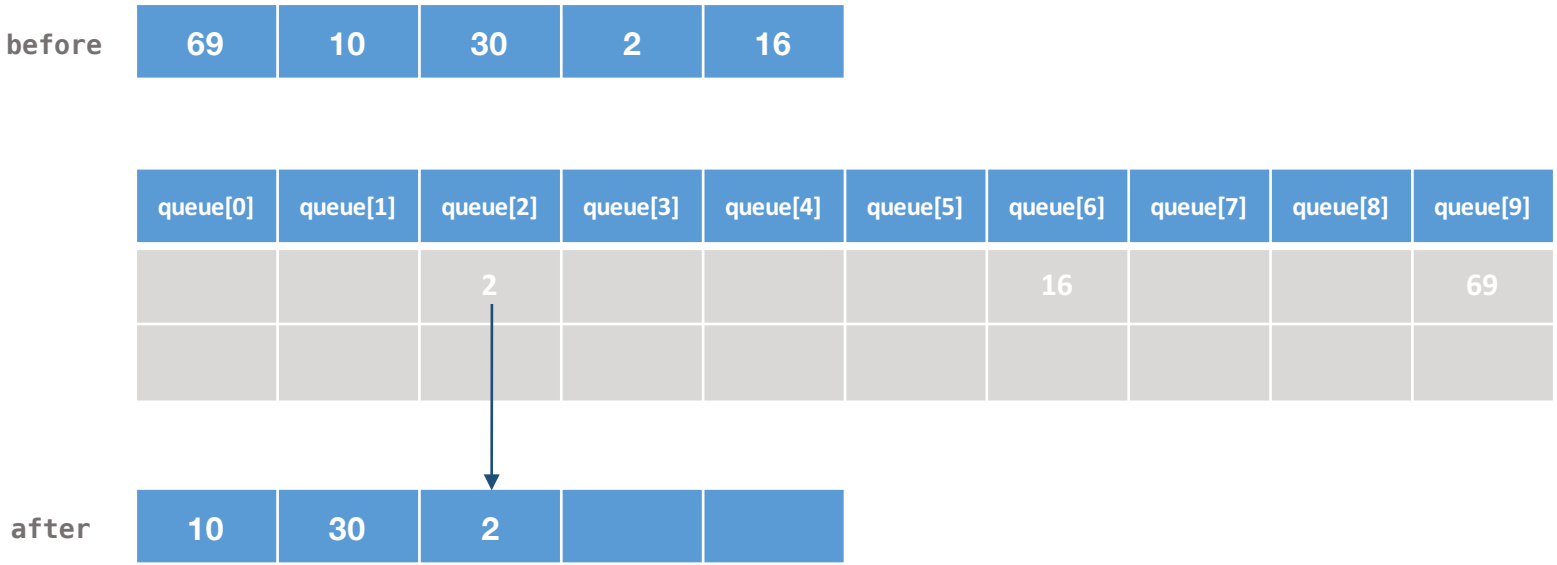
2. 버켓의 순서대로 원소를 꺼내 재분배



기수 정렬

수행 과정

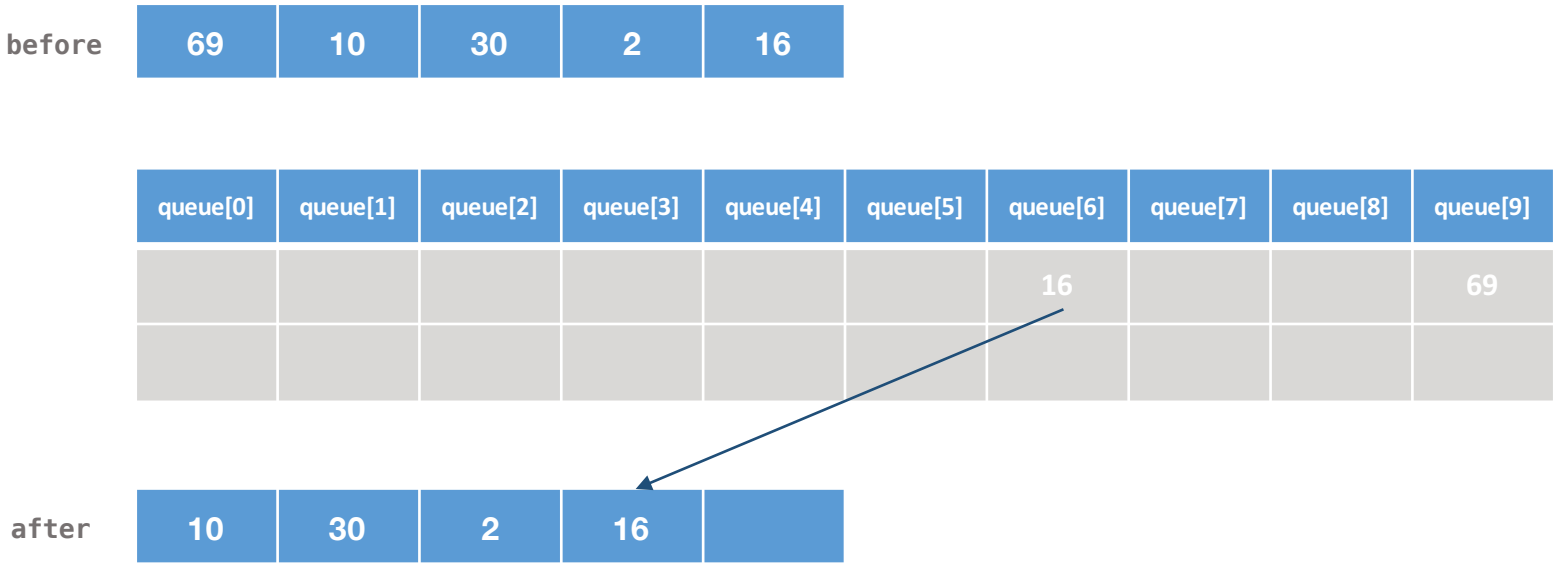
2. 버켓의 순서대로 원소를 꺼내 재분배



기수 정렬

수행 과정

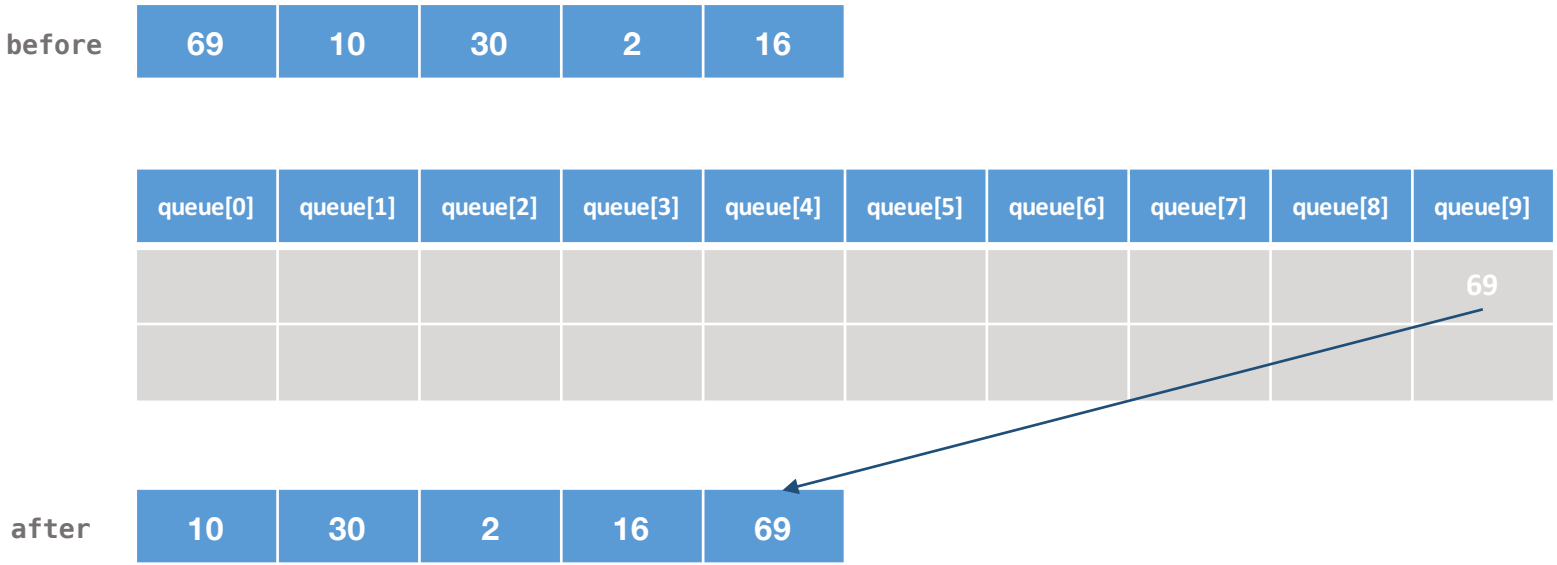
2. 버켓의 순서대로 원소를 꺼내 재분배



기수 정렬

수행 과정

2. 버켓의 순서대로 원소를 꺼내 재분배



기수 정렬

수행 과정

2. 버켓의 순서대로 원소를 꺼내 재분배

before

69	10	30	2	16

queue[0]	queue[1]	queue[2]	queue[3]	queue[4]	queue[5]	queue[6]	queue[7]	queue[8]	queue[9]

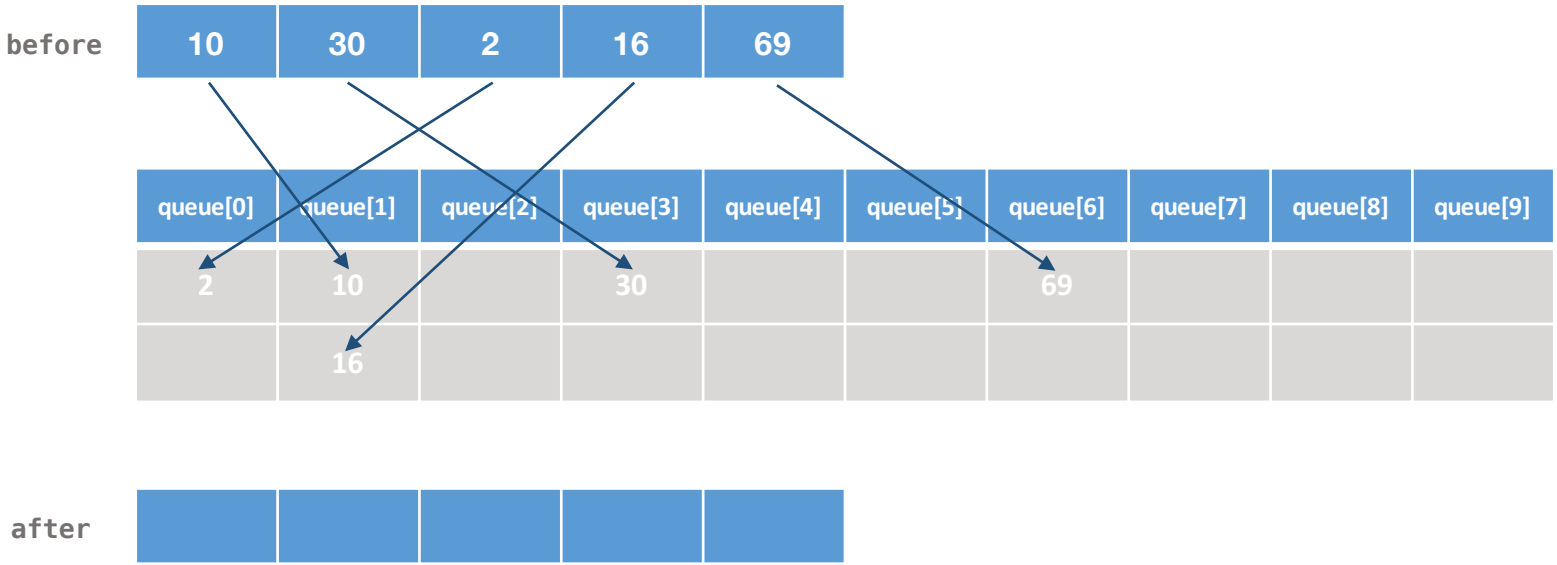
after

10	30	2	16	69

기수 정렬

수행 과정

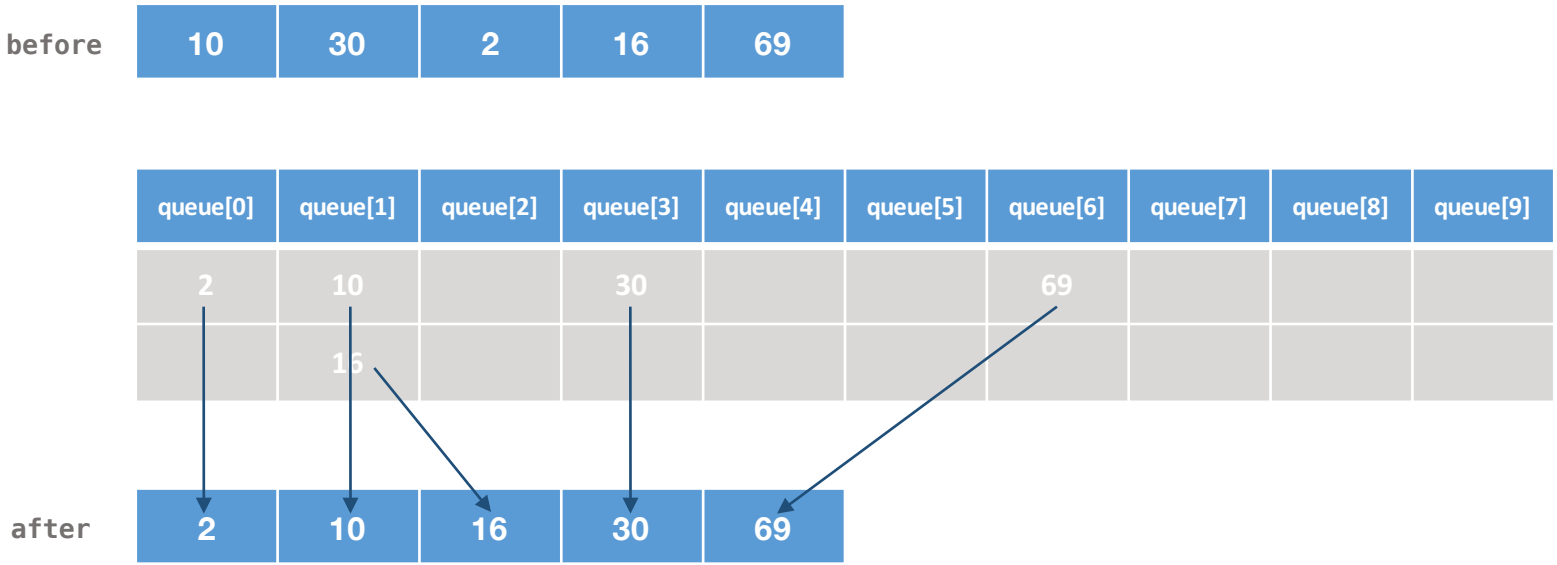
3. 키 값의 자릿수만큼 위의 과정을 반복



기수 정렬

수행 과정

3. 키 값의 자릿수만큼 위의 과정을 반복



기수 정렬

```
public void radixsort(T[] arr){
    int max_digit = 0;
    //최대 자릿수를 구해줌
    for(int i=0; i< arr.length; i++) {
        if(arr[i].toString().length() > max_digit)
            max_digit =arr[i].toString().length();
    }


    Queue<T>[] queues = new Queue[52];
    for(int i=0; i<queues.length;i++){
        queues[i] = new Queue<T>();
    }

    for(int i=0;i<max_digit;i++) {

        for (int j = 0; j < arr.length; j++) {
            Node<T> node = new Node(arr[j]);
            int digit = arr[j].toString().length() -
(i+1);
            if(digit<0){
                queues[0].enqueue(node);
            }else {
                String idx =
arr[j].toString().substring(digit, digit + 1);
                char[] chars = idx.toCharArray();
                if(chars[0]>=48 && chars[0]<=57) { //
숫자일 경우

queues[Integer.parseInt(idx)].enqueue(node);
                }else{  //문자일 경우

queues[alphaToInt(chars[0])].enqueue(node);
                }
            }
        }
```

- 원소들의 최대 자릿수를 구해줌
- 큐 생성
- 큐에 넣어주는 과정
- 비교할 자릿수를 구해줌(digit)
- 숫자일 경우와 문자일 경우로 나누어 enqueue 해줌

기수 정렬

코드 분석

```
int newArrayIndex=0;
        for (int j = 0; j < queues.length; j++) {
            while (!queues[j].isEmpty()) {
                arr[newArrayIndex++] =
queues[j].dequeue().t;
            }
        }
    }


}
private int alphaToInt(char alphabet){
    if(alphabet>=65 && alphabet<=90) return (alphabet-
65); // 대문자
    else return (alphabet-97+26); //소문자
}
```

- 큐에서 차례대로 꺼내어 정렬

기수 정렬

적용 사례

전화번호 정렬

날짜 정렬

주민번호 정렬

기수 정렬

성능 분석

메모리 사용 공간

O (n) : **n** 개의 원소에 대하여 버킷을 포함한 **2n** 개의 메모리 사용

연산 시간

1. 실제 : **kn**

   최대 자릿수를 **k**라 할 때, **k**번 동안 **n**개의 원소가 큐에 대입되고 꺼내어져 정렬

2. 이론 : **n**

   대부분의 컴퓨터가 다룰 수 있는 자릿수가 n에 비해 현저하게 작으므로, 고려되지 않음

평균 시간 복잡도

O (n)

계수 정렬

개념 · 원리 · 특징 · 수행 과정 · 코드 분석 · 성능 분석

계수 정렬

개념

각 원소가 몇 개씩 있는지 세는 과정을 통해 원소를 정렬하는 방식이다

계수 정렬

원리

자기 자신보다 작은 원소의 갯수를 알면 해당 원소의 위치가 결정된다

계수 정렬

특징

장점

- 속도가 비교적 빠른 편이다
- 안정적인 정렬이다 - 키 값이 같은 원소의 순서에 대해 정렬 후에도 순서가 유지된다

단점

- 기수 정렬과 마찬가지로 버켓을 구현하기 위한 별도의 메모리를 필요로 한다
- 버켓을 사용하므로, 정렬할 수 있는 데이터의 타입이 한정되어 있다

계수 정렬

수행 과정

1. 해당 원소의 값을 인덱스로 하는 count 배열을 사용하여 각 원소의 개수를 셈

before

9	1	7	2	1	8	1	2

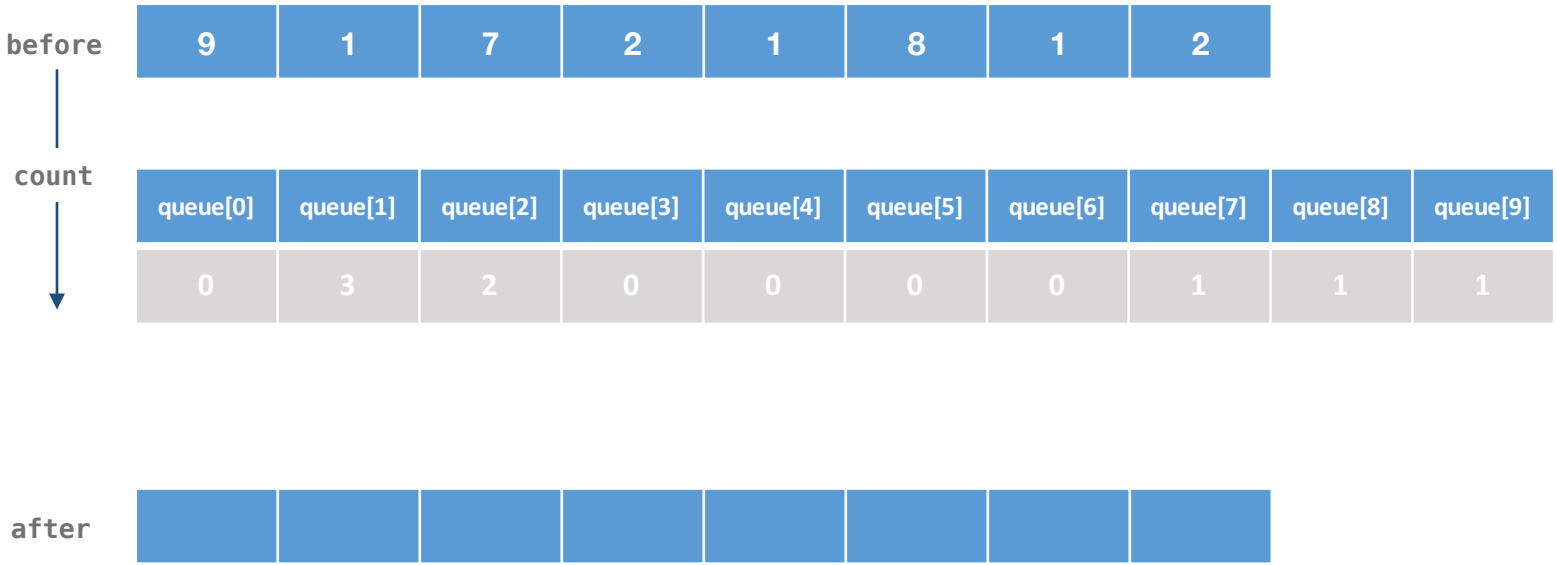
queue[0]	queue[1]	queue[2]	queue[3]	queue[4]	queue[5]	queue[6]	queue[7]	queue[8]	queue[9]
0	0	0	0	0	0	0	0	0	0

after

계수 정렬

수행 과정

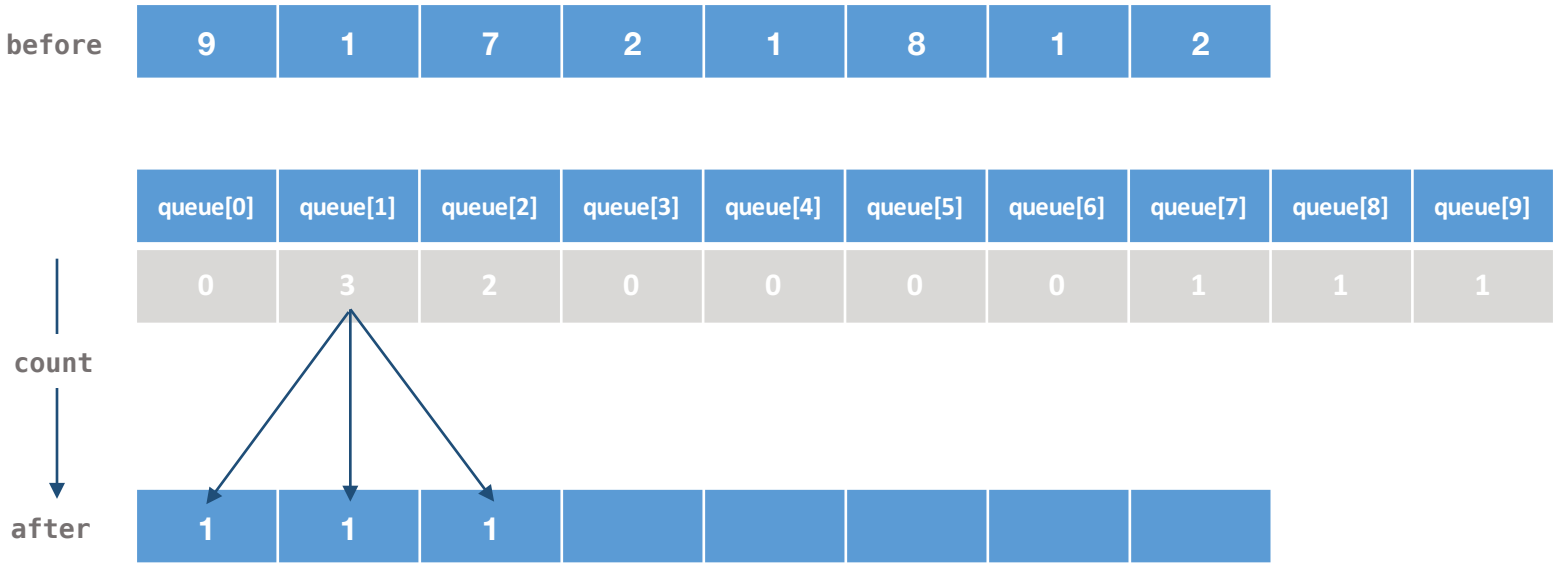
1. 해당 원소의 값을 인덱스로 하는 count 배열을 사용하여 각 원소의 개수를 셈



계수 정렬

수행 과정

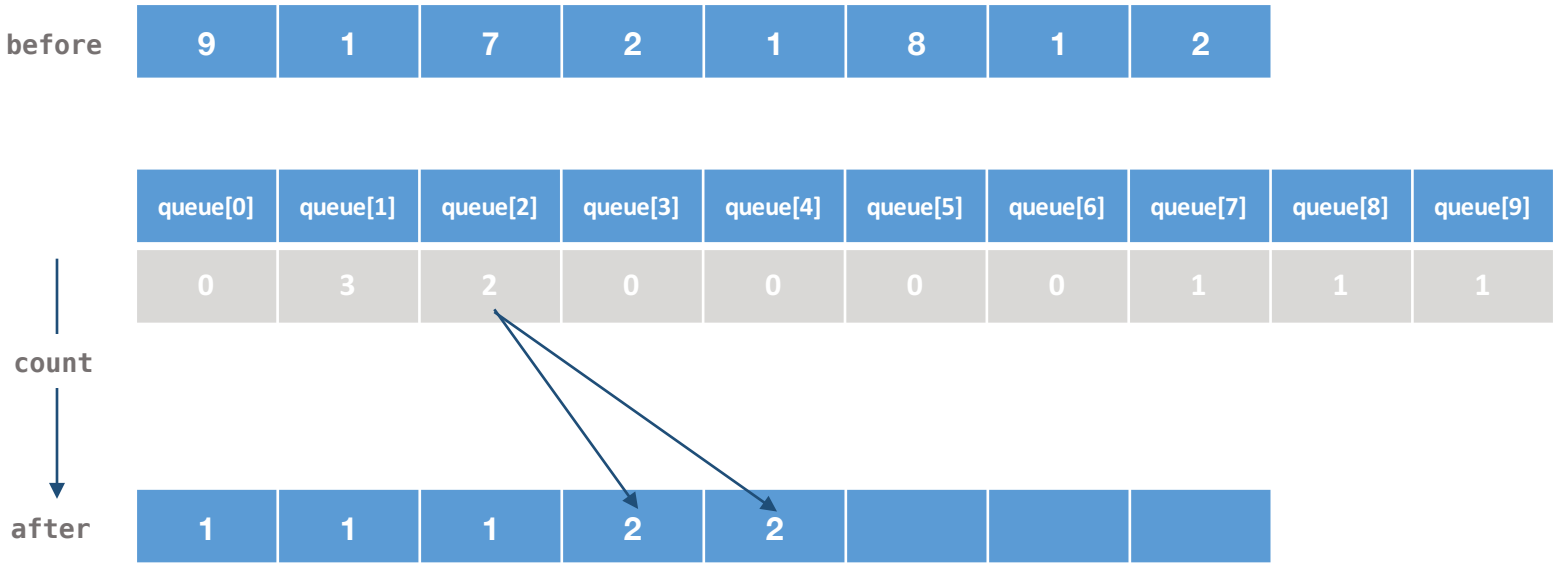
2. count 배열의 첫 원소부터 차례대로 갯수를 세어 재배치



계수 정렬

수행 과정

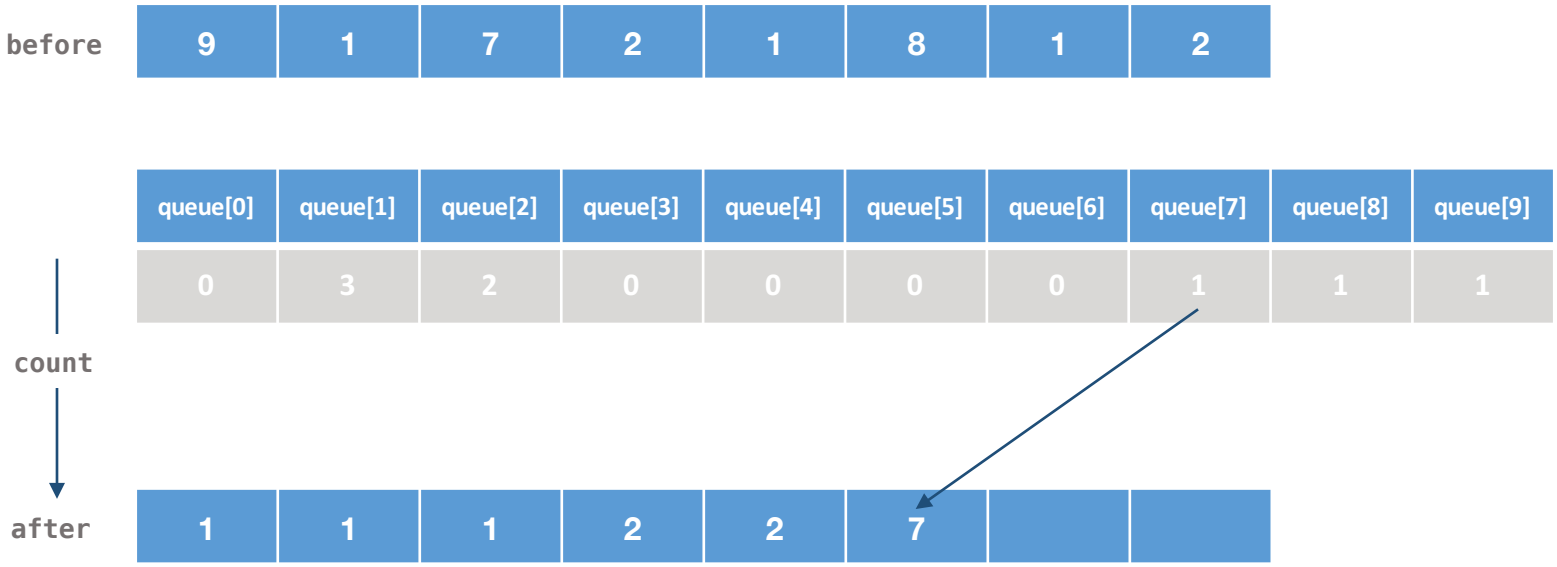
2. count 배열의 첫 원소부터 차례대로 갯수를 세어 재배치



계수 정렬

수행 과정

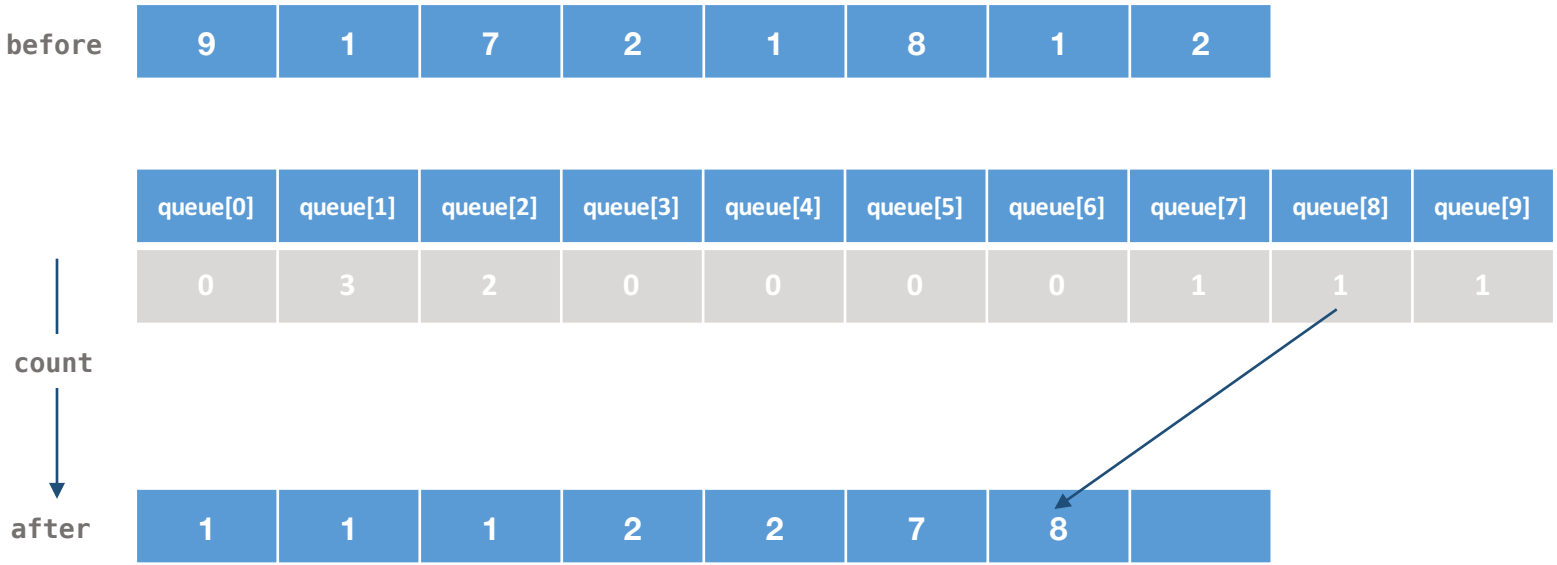
2. count 배열의 첫 원소부터 차례대로 갯수를 세어 재배치



계수 정렬

수행 과정

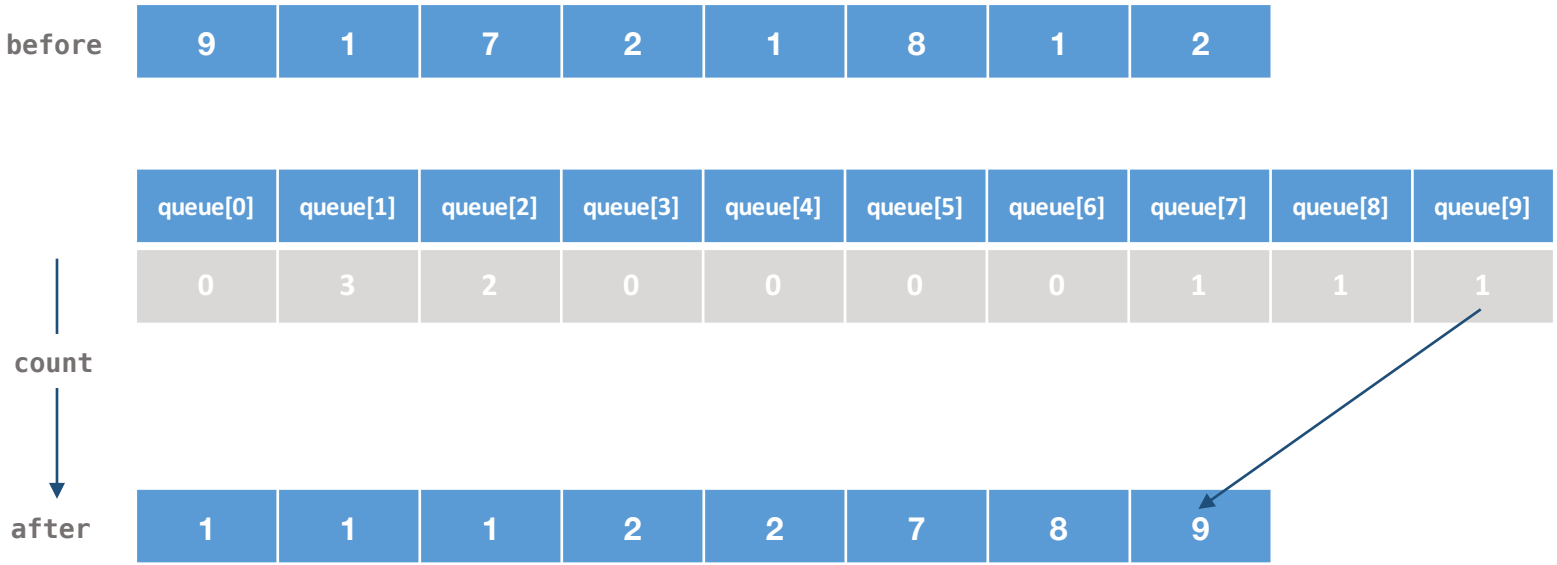
2. count 배열의 첫 원소부터 차례대로 갯수를 세어 재배치



계수 정렬

수행 과정

2. count 배열의 첫 원소부터 차례대로 갯수를 세어 재배치



계수 정렬

코드 분석

```
public static void sort(ArrayList<ICompare>
source) {

    HashMap<Integer, ArrayList<ICompare>> map
            = new HashMap<>();

    for (ICompare item : source) {
        if (map.containsKey(item.identifier()))
{

map.get(item.identifier()).add(item);
        } else {
            ArrayList<ICompare> line = new
ArrayList<>();
            line.add(item);
            map.put(item.identifier(), line);
        }
    }

    for (int key : map.keySet())
        for (ICompare item : map.get(key))
            System.out.println(item);
}
```

- HashMap 생성
- Map에 있으면 추가, 없으면 만들어서 추가
- 정렬된 item 순서대로 출력

계수 정렬

코드 분석

```
public class Data implements ICompare<Data> {

    private static Random random = new Random();

    public int id;
    public String name;
    public String association;

    public Data(String name, String association) {
        this.name = name;
        this.association = association;

        this.id = random.nextInt(10);
    }

    @Override
    public int identifier() { return id; }
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Data))   return false;
        return obj == this || this.id == ((Data)
obj).id;
    }
    @Override
    public String toString() {
        return "id : " + id + " / " + name + " / "
+ association;
    }
}
public interface
ICompare<T> {
    int identifier();
}
```

- ICompare를 implements한 Data클래스
- 생성자
- Override한 메소드

계수 정렬

성능 분석

메모리 사용 공간

O (n + k) : **n** 개의 원소에 대하여 **n** 개 + 원소의 범위에 해당하는 **k** 개의 메모리 사용

연산 시간

전체 원소의 범위를 체크하고 갯수를 측정하는 횟수는 n 번이며

위에서 측정한 내용을 기반으로 정렬 결과를 구성하는 연산은 k 번 실행된다

평균 시간 복잡도

O (n) : 원래 시간 복잡도는 **O (n + k)** 이나, **k**가 **O(n)** 이하의 작은 범위에서 사용되므로 **O(n)**에 근사한다