# [읽을거리] 7. 정렬

## Sortable Interface

정렬알고리즘들을 직접 구현해 봅시다.

입력으로 List 와 Comparator 를 제공하고, 원본은 유지하고 정렬된 새로운 List를 반환하는 형태로 개발하겠습니다.

그래서 Sort기능을 추상화 하여 이러한 인터페이스를 만들고 각 구현체는 이 인터페이스를 구현하도록 하겠습니다.

```
import java.util.Comparator;
import java.util.List;

public interface Sortable {
    <T> List<T> sort(List<T> list, Comparator<T> comparator);
}
```

제너릭 타입 T를 사용하고 있는데, 특별히 T 가 Comparable을 구현하고 있을 때 Comparator를 제공할 필요는 없죠. 그래서 default 메소드도 추가하겠습니다.

```
import java.util.Comparator;
import java.util.List;

public interface Sortable {
    <T> List<T> sort(List<T> list, Comparator<T> comparator);

    default <T extends Comparable> List<T> sort(List<T> list) {
        return sort(list, Comparable::compareTo);
    }
}
```

이제 알고리즘 별로 구현체를 만들어 보겠습니다.

### 1️⃣ BubbleSort

```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

public class BubbleSort implements Sortable {
    @Override
    public <T> List<T> sort(List<T> list, Comparator<T> comparator) {
        List<T> copy = new LinkedList<>(list);
        int size = copy.size();
        for (int i = 0; i < size; i++) {
            for (int j = i + 1; j < size; j++) {
                T d1 = copy.get(i);
                T d2 = copy.get(j);
                if (comparator.compare(d1, d2) > 0) {
                    copy.set(i, d2);
                    copy.set(j, d1);
                }
            }
```

```
        }
        return copy;
    }
}
```

인자로 전달되는 `list` 로부터 복사본 리스트인 `copy` 를 만들어서 순서를 교체해서 정렬을 수행하고 있습니다.

여기서 사용되는 제너릭 타입 T는 레퍼런스 타입입니다. 복사본을 만들다는 것은 참조값들 만들 복사하고, 참조값들의 배치를 재정렬하고 있습니다.

실제 객체 자체가 교체되거나 하지는 않는다는 것을 기억하세요.

## 2️⃣ InsertSort

```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

public class InsertSort implements Sortable {
    @Override
    public <T> List<T> sort(List<T> list, Comparator<T> comparator) {
        List<T> copy = new LinkedList<>(list);
        int size = copy.size();

        for (int i = 1; i < size; i++) {
            T d1 = copy.get(i);
            for (int j = i - 1; j >= 0; j--) {
                T d2 = copy.get(j);
                if (comparator.compare(d1, d2) > 0) break;
                copy.remove(j);
                copy.add(j + 1, d2);
            }
        }
        return copy;
    }
}
```

데이터 삽입 부분에서 기존 아이템을 제거(`remove`) 해서 특정 위치에 삽입(`add`) 하고 있습니다.

## 3️⃣ SelectionSort

```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

public class SelectionSort implements Sortable {
    @Override
    public <T> List<T> sort(List<T> list, Comparator<T> comparator) {
        List<T> copy = new LinkedList<>(list);
        int size = copy.size();

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

            int minIndex = i;
            T min = copy.get(i);

            for (int j = i+1; j < size; j++) {
                T d = copy.get(j);
```

```
                if (min == null || comparator.compare(min, d) > 0) {
                    minIndex = j;
                    min = d;
                }
            }

            copy.remove(minIndex);
            copy.add(i, min);
        }
        return copy;
    }
}
```

내부 for-loop 에서 최소값을 찾아내는 과정에서, 반복 범위 중 첫번째 요소 min으로 먼저 지정하고 나서 나머지 요소 중에서 min을 업데이트 하면서 찾아내는 방식으로 구현되어 있습니다.

## 4️⃣ QuickSort

```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class QuickSort implements Sortable {
    @Override
    public <T> List<T> sort(List<T> list, Comparator<T> comparator) {
        List<T> copy = new LinkedList<>(list);
        return quickSort(copy, comparator);
    }

    private <T> List<T> quickSort(List<T> list, Comparator<T> comparator) {
        if (list.size() <= 1) return list;

        T pivot = list.remove(list.size() / 2);
        List<T> lesser = quickSort(sublist(list, (d) -> comparator.compare(pivot, d) > 0), comparator);
        List<T> greater = quickSort(sublist(list, (d) -> comparator.compare(pivot, d) <= 0), comparator);

        return new LinkedList<T>() {{
            addAll(lesser);
            add(pivot);
            addAll(greater);
        }};
    }

    private <T> List<T> sublist(List<T> list, Predicate<T> filter) {
        return list.stream().filter(filter).collect(Collectors.toList());
    }
}
```

재귀를 위해 내부에 `quickSort` 메소드와 특정 조건에 맞는 값으로만 구성된 부분리스트를 만드는 `sublist` 메소드를 별도로 만들어 활용하고 있습니다.

`quickSort` 에서 pivot 값을 리스트의 중간위치 값으로 설정하고 있습니다. 그리고 merge된 리스트를 반환하기 위해서 double brace initialization을 사용하고 있습니다.

### 🤷‍♀️ pivot 값을 어디서 선택해야 하는가?

이상적인 경우 퀵정렬의 시간 복잡도는 O(nlogn) 입니다. 하지만 그룹의 개수가 비대칭으로 특히 한쪽으로 쏠리게 되는 최악의 경우 시간 복잡도는 $O(n^2)$가 됩니다. pivot값을 기준으로 큰값들과 작은값들을 구분하기

때문에 pivot을 어떻게 정하느냐에 따라서 비대칭성에 영향을 줄 수 있습니다. 다시말해 pivot값을 어떻게 정하느냐에 따라 시간복잡도에 영향을 미치게 됩니다.

어떤 pivot값을 선택하는 것이 유지할지 지식이 있다면 좋겠지만, 보통 원본 데이터의 구성이 어떻게 되어있는지는 알 수 없죠. 그렇기 때문에 어떤 pivot 값을 선택할 수 있는 기준이 없습니다. 그래서 아무 값이나 선택해도 평균적으로 동일한 결과를 수 있다고 기대할 수 있습니다.

결국 특정 리스트 - 어떤 배치상태를 가지는지 알고 있는 리스트 - 가 아닌 경우에는 pivot값을 어떤것을 선택해도 됩니다.

## 5️⃣ MergeSort

```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

public class MergeSort implements Sortable {
    @Override
    public <T> List<T> sort(List<T> list, Comparator<T> comparator) {
        List<T> copy = new LinkedList<>(list);
        return mergeSort(copy, comparator);
    }

    private <T> List<T> mergeSort(List<T> list, Comparator<T> comparator) {
        if (list.size() <= 1) return list;

        int mid = list.size() / 2;
        List<T> left = sort(list.subList(0, mid), comparator);
        List<T> right = sort(list.subList(mid, list.size()), comparator);

        List<T> merged = new LinkedList<>();
        while (!left.isEmpty() || !right.isEmpty()) {
            if (left.isEmpty()) {
                merged.addAll(right);
                break;
            }

            if (right.isEmpty()) {
                merged.addAll(left);
                break;
            }

            T leftFirst = left.get(0);
            T rightFirst = right.get(0);
            if (comparator.compare(leftFirst, rightFirst) < 0) {
                merged.add(left.remove(0));
            } else {
                merged.add(right.remove(0));
            }
        }

        return merged;
    }
}
```

`list.subList` 를 통해서 index를 에 따른 sublist 를 사용하고 있습니다. 하지만, 이렇게 얻은 sublist 는 fixed-size list 이기 때문에 `sort` 를 통해서 재귀시킴으로써 copy가 만들어지도록 하고 있습니다.

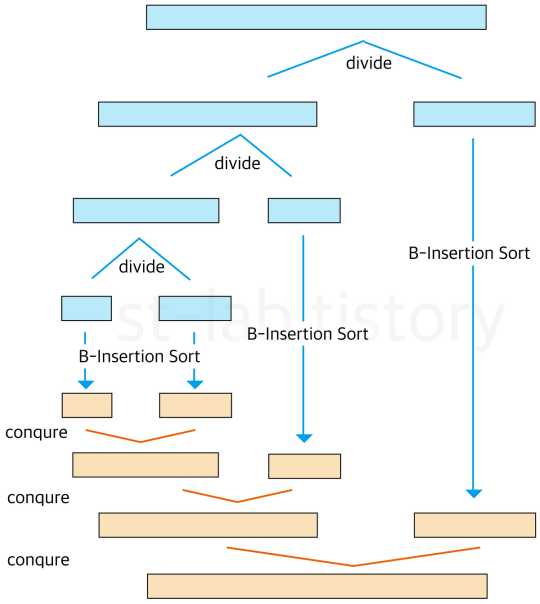
### 🤷‍♂️ Java 는 어떤 sort 알고리즘을 사용하고 있을까?

Java Collections 에서는 `List.sort` 와 `Arrays.sort` 에서 정렬기능을 제공해주고 있습니다. 이 메소드들은 변형된 merge sort 로 구현되어 있었는데, Java 8 버전 이후부터는 모두 **Tim Sort** 알고리즘으로 변경되었습니다.

### Tim Sort

2002년 Tim Peters 가 Insertion sort와 Merge sort를 결합하여 만든 새로운 정렬 알고리즘 입니다. 전체를 작은 덩어리로 잘르고 각각의 덩어리를 Insertion sort로 정렬한 뒤 병합하는 방식으로 동작하는 방식입니다.

현재는 많은 언어 및 라이브러리에서 표준 정렬 알고리즘으로 사용되고 있습니다.




**Timsort**

| | |
|---|---|
| **Class** | Sorting algorithm |
| **Data structure** | Array |
| **Worst-case performance** | $O(n \log n)$[1][2] |
| **Best-case performance** | $O(n)$[3] |
| **Average performance** | $O(n \log n)$ |
| **Worst-case space complexity** | $O(n)$ |

## Unit Test

Junit 을 사용한 테스트 코드 입니다. 각 구현체 별로 @Test 를 구분해서 만들고 검증하고 있습니다.

```
import org.junit.jupiter.api.Test;

import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

class SortTest {

    private List<Integer> getProblemList() {
        return Arrays.asList(10, 9, 8, 7, 6, 5, 4, 3, 2, 1);
    }

    private List<Integer> getAnswerList() {
        return Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    }

    @Test
    void bubbleSort() {
        List<Integer> list = getProblemList();
        List<Integer> sorted = new BubbleSort().sort(list);
        assertEquals(getAnswerList(), sorted);
    }

    @Test
```

```
    void insertSort() {
        List<Integer> list = getProblemList();
        List<Integer> sorted = new InsertSort().sort(list);
        assertEquals(getAnswerList(), sorted);
    }

    @Test
    void selectionSort() {
        List<Integer> list = getProblemList();
        List<Integer> sorted = new SelectionSort().sort(list);
        assertEquals(getAnswerList(), sorted);
    }

    @Test
    void quickSort() {
        List<Integer> list = getProblemList();
        List<Integer> sorted = new QuickSort().sort(list);
        assertEquals(getAnswerList(), sorted);
    }

    @Test
    void mergeSort() {
        List<Integer> list = getProblemList();
        List<Integer> sorted = new MergeSort().sort(list);
        assertEquals(getAnswerList(), sorted);
    }
}
```