알고리즘 과제 보고서

- AL10, 정렬 알고리즘 검증 과제

충남대학교 컴퓨터공학과
분반: 알고리즘 04 분반
학번: 201701975
이름: 구건모

과제의 주요이론

이번 과제는 정렬 알고리즘인 heapSort, insertionSort, quickSort를 구현해보는 것이었습니다.
정렬 알고리즘인 quickSort는 pivot을 설정하고 해당 pivot을 기준으로 작은 값과 큰 값들을 각각 partition을 나눈 후 나눈 partition에서 다시 recursive 하게 Sorting을 진행하는 방식의 알고리즘 입니다. 특히 이번 과제에서는 quickSort의 경우 pivot을 다양하게 설정하여 정렬을 수행하는 과정도 구현하게 됩니다.
HeapSort는 정렬할 원소들을 통해 MaxHeap을 구성하고 해당 heap에서 MAX 값을 제거한 후 다시 MaxHeap으로 구성하는 방식을 반복하여 MAX 값이 제거되면서 공백힙이 될 때까지 정렬되는 방식입니다.
insertionSort는 각 요소들을 앞에서부터 비교하면서 자신의 위치를 찾아가도록 하는 방식의 알고리즘 입니다.

AL10_S1 코드

주요 구현 내용

```java
package sort;

public abstract class Sort<E extends Comparable<E>> {
    private boolean _nonDecreasingOrder;

    public boolean nonDecreasingOrder() {
        return this._nonDecreasingOrder;
    }

    public void setNonDecreasingOrder(boolean newNonDecreasingOrder) {
        this._nonDecreasingOrder = newNonDecreasingOrder;
    }

    protected void swap(E[] aList, int i, int j) {
        E tempElement = aList[i];
        aList[i] = aList[j];
        aList[j] = tempElement;
    }

    protected int compare(E anElement, E theOtherElement) {
        if (this.nonDecreasingOrder()) {
            return anElement.compareTo(theOtherElement);

        } else {
            return -anElement.compareTo(theOtherElement);

        }
    }

    public Sort(boolean givenSortingOrder) {
        this.setNonDecreasingOrder(givenSortingOrder);
    }

    public abstract boolean sort(E[] aList);
}

```

Class Sort

Sort 클래스에는 Sorting 과정에서 필요한 swap, compare 등의 메서드들이 정의되어 있습니다. sort method 상속한 클래스에서 구현하여 사용하므로 abstract으로 정의되어 있습니다.

주요 구현 내용

```java
package sort;

public class InsertionSort<E extends Comparable<E>> extends Sort<E> {

    public InsertionSort(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    @Override
    public boolean sort(E[] aList) {
        // TODO Auto-generated method stub
        if (aList.length <= 1) {
            return false;

        }
        int minLoc = 0;
        for (int i = 0; i < aList.length; i++) {
            if (this.compare(aList[minLoc], aList[i]) > 0) {
                minLoc = i;
            }

        }
        this.swap(aList, minLoc, 0);

        for (int i = 2; i < aList.length; i++) {
            E insertedElement = aList[i];
            int j = i - 1;
            while (this.compare(aList[j], insertedElement) > 0) {
                aList[j + 1] = aList[j];
                j--;
            }
            aList[j + 1] = insertedElement;
        }
        return true;

    }

}
```

class insertionSort

삽입정렬의 Sort method는 기본적으로 정렬할 데이터의 사이즈가 1개인 경우 정렬의 의미가 없기 때문에 2개 이상의 데이터에 대해서만 정렬을 수행합니다. 처음의 minLoc을 0으로 설정하고 for문과 compare 메소드를 이용해 minLoc을 찾아냅니다.

이후에 각 원소를 이전의 위치 값과 비교후에 해당 원소가 이전 원소값보다 작으면 위치를 서로 교환하는 방식으로 정렬이 이루어집니다.

주요 구현 내용

AL10_S1_201701975_구건모/src/sort/InsertionSort.java

```java
package sort;

public class HeapSort<E extends Comparable<E>> extends Sort<E> {

    private static final int HEAP_ROOT = 1;

    public HeapSort(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    private E removeMAX(E[] aList, int heapSize) {
        E removedElement = aList[HeapSort.HEAP_ROOT];
        aList[HeapSort.HEAP_ROOT] = aList[heapSize];
        this.adjust(aList, HeapSort.HEAP_ROOT, heapSize - 1);
        return removedElement;
    }

    private void adjust(E[] aList, int root, int endOfHeap) {
        int parent = root;
        E rootElement = aList[root];
        while ((parent * 2) <= endOfHeap) {
            int biggerChild = parent * 2;
            int rightChild = biggerChild + 1;
            if ((rightChild <= endOfHeap) && (this.compare(aList[biggerChild], aList[rightChild]) < 0)) {
                biggerChild = rightChild;

            }
            if (this.compare(rootElement, aList[biggerChild]) >= 0) {
                break;

            }
            aList[parent] = aList[biggerChild];
            parent = biggerChild;

        }
        aList[parent] = rootElement;
    }

    private void makeInitHeap(E[] aList) {
        for (int rootOfSubtree = (aList.length - 1) / 2; rootOfSubtree >= 1; rootOfSubtree--) {
            this.adjust(aList, rootOfSubtree, aList.length - 1);
        }
    }

```

class heapSort

정렬 알고리즘 중 heapsort를 수행하는 클래스로 특정한 노드의 자식 중 더 큰 자식과 자신의 위치를 바꾸는 알고리즘인 adjust 메서드를 통해 heap의 특성을 유지하도록 합니다.

주요 구현 내용

```
    @Override
    public boolean sort(E[] aList) {
        // TODO Auto-generated method stub
        if (aList.length <= 1) {
            return false;

        }
        int minLoc = 0;
        for (int i = 1; i < aList.length; i++) {
            if (this.compare(aList[minLoc], aList[i]) > 0) {
                minLoc = i;
            }
        }
        this.swap(aList, minLoc, 0);
        this.makeInitHeap(aList);
        for (int heapSize = aList.length - 1; heapSize > 1; heapSize--) {
            E maxElement = this.removeMAX(aList, heapSize);
            aList[heapSize] = maxElement;
        }
        return true;

    }

}
```

Class heapSort의 Sort() method

removeMAX 메서드를 통해 하나씩 aList[heapSize] 에 넣어주고 adjust 메서드로 다시 힙구조를 유지해주는 과정을 반복하면서 정렬이 이루어지게 됩니다.

주요 구현 내용

```java
package sort;

public class QuickSort<E extends Comparable<E>> extends Sort<E> {

    public QuickSort(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    protected int pivot(E[] aList, int left, int right) {
        return left;

    }

    protected int partition(E[] aList, int left, int right) {
        int pivot = this.pivot(aList, left, right);
        this.swap(aList, left, pivot);
        E pivotElement = aList[left];
        int toRight = left;
        int toLeft = right + 1;
        do {
            do {
                toRight++;
            } while (this.compare(aList[toRight], pivotElement) < 0);
            do {
                toLeft--;
            } while (this.compare(aList[toLeft], pivotElement) > 0);
            if (toRight < toLeft) {
                this.swap(aList, toRight, toLeft);
            }
        } while (toRight < toLeft);
        this.swap(aList, left, toLeft);
        return toLeft;
    }

    protected void quickSortRecursively(E[] aList, int left, int right) {
        if (left < right) {
            int mid = partition(aList, left, right);
            quickSortRecursively(aList, left, mid - 1);
            quickSortRecursively(aList, mid + 1, right);

        }
    }
```

*코드의 일부입니다.

Class QuickSort

QuickSort는 pivot 이라는 기준값을 가지고 피벗 앞에는 피벗값보다 작은 값들을, pivot 이후에는 pivot 보다 큰 값들로 분할한 뒤 분할된 두 구간에 대하여 Recursive 하게 정렬을 반복합니다. partition method 를 통해 partition을 나누고 해당 메서드로 얻어진 값을 통해 다시 Recursive하게 구간을 나누어 quicksort를 수행합니다 .

AL10_S2 코드

주요 구현 내용

```java
package sort;

public abstract class QuickSort<E extends Comparable<E>> extends Sort<E> {

    public QuickSort(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    protected abstract int pivot(E[] aList, int left, int right);


    protected int partition(E[] aList, int left, int right) {
        int pivot = this.pivot(aList, left, right);
        this.swap(aList, left, pivot);
        E pivotElement = aList[left];
        int toRight = left;
        int toLeft = right + 1;
        do {
            do {
                toRight++;
            } while (this.compare(aList[toRight], pivotElement) < 0);
            do {
                toLeft--;
            } while (this.compare(aList[toLeft], pivotElement) > 0);
            if (toRight < toLeft) {
                this.swap(aList, toRight, toLeft);
            }
        } while (toRight < toLeft);
        this.swap(aList, left, toLeft);
        return toLeft;
    }

    protected void quickSortRecursively(E[] aList, int left, int right) {
        if (left < right) {
            int mid = partition(aList, left, right);
            quickSortRecursively(aList, left, mid - 1);
            quickSortRecursively(aList, mid + 1, right);

        }
    }
```

Class QuickSort

이번과제의 첫번째 프로젝트 (S1) 과 동일한 구현이지만, 프로젝트(S2) 에서는 pivot 을 여러가지로 설정하여 구현할 것이기 때문에 pivot을 설정하는 부분만 abstract으로 선언하여 상속받은 클래스에서 구현되게 됩니다.

주요 구현 내용

```java
package sort;

public class QuickSortByPivotLeft <E extends Comparable<E>> extends QuickSort<E> {

    public QuickSortByPivotLeft(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected int pivot(E[] aList, int left, int right) {
        // TODO Auto-generated method stub
        return left;

    }


}
```

Class QuickSortByPivotLeft

QuickSortByPivotLeft는 pivot을 left 값으로 설정하여 QuickSort를 수행하기 위해 구현된 클래스로 QuickSort와 sorting 과정은 같고 pivot만 Left 값으로 설정하는 것이기 때문에 pivot() method 만 override 하여 구현됩니다.

주요 구현 내용

```java
package sort;

public class QuickSortByPivotMedian<E extends Comparable<E>> extends QuickSort<E> {

    public QuickSortByPivotMedian(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected int pivot(E[] aList, int left, int right) {
        // TODO Auto-generated method stub
        if ((right - left) < 3) {
            return left;
        }
        int mid = (left + right) / 2;
        if (this.compare(aList[left], aList[mid]) < 0) {
            if (this.compare(aList[mid], aList[right]) < 0) {
                return mid;
            } else {
                if (this.compare(aList[left], aList[right]) < 0) {
                    return right;
                } else {
                    return left;
                }

            }
        } else {
            if (this.compare(aList[mid], aList[right]) < 0) {
                return left;
            } else {
                if (this.compare(aList[mid], aList[right]) < 0) {
                    return right;
                } else {
                    return mid;
                }

            }
        }

    }

}
```

Class QuickSortByPivotMedian

QuickSortByPivotMedian 클래스는 기존의 quickSort를 진행할 때 pivot을 설정하는 과정에서 left,mid,right 세개의 값을 비교하여 median 값을 pivot으로 선택합니다. pivot() 메서드를 통해서 3개의 값을 비교 후 median 값을 return 하게 됩니다.

주요 구현 내용

```java
package sort;

public class QuickSortByPivotMid<E extends Comparable<E>> extends QuickSort<E> {

    public QuickSortByPivotMid(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected int pivot(E[] aList, int left, int right) {
        // TODO Auto-generated method stub
        return ((left + right) / 2);
    }

}
```

Class QuickSortByPivotMid

QuickSortByPivotMid 클래스는 중앙값을 pivot으로 설정하여 QuickSort를 수행하기 위해 구현된 클래스로 QuickSort를 extend 하며 모든 부분이 같고 pivot만 중앙값으로 설정하는 것이기 때문에 pivot() method 만 override 하여 구현되었습니다.

주요 구현 내용

```java
package sort;

import java.util.Random;

public class QuickSortByPivotRandom<E extends Comparable<E>> extends QuickSort<E> {

    private Random _random;

    public QuickSortByPivotRandom(boolean givenSortingOrder) {
        // TODO Auto-generated constructor stub
        super(givenSortingOrder);
        this.setRandom(new Random());
    }

    private Random random() {
        return this._random;
    }

    private void setRandom(Random newRandom) {
        this._random = newRandom;
    }

    @Override
    protected int pivot(E[] aList, int left, int right) {
        // TODO Auto-generated method stub
        int randomLocationFromLeft = this.random().nextInt(right - left + 1);
        int pivot = left + randomLocationFromLeft;
        return pivot;
    }

}
```

Class QuickSortByPivotRandom

QuickSortByPivotRandom 클래스는 pivot 값을 random 하게 설정하여 QuickSort를 수행하기 위해 구현된 클래스이므로, 난수 발생을 위한 Random 멤버변수와 이에 대한 getter/setter 가 구현되어 있고, QuickSort에서 pivot() method를 random하게 값을 가지도록 override 하여 구현되었습니다.

주요 구현 내용

```java
package sort;

public class QuickSortWithInsertionSort<E extends Comparable<E>> extends QuickSortByPivotRandom<E> {
    private static final int MAX_SIZE_FOR_INSERTION_SORT = 20;

    public QuickSortWithInsertionSort(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    private boolean insertionSort(E[] aList, int left, int right) {
        for (int i = (right - 1); i >= left; i--) {
            E insertedElement = aList[i];
            int j = i + 1;
            while (this.compare(aList[j], insertedElement) < 0) {
                aList[j - 1] = aList[j];

                j++;
            }
            aList[j - 1] = insertedElement;
        }

        return true;
    }

    @Override
    protected void quickSortRecursively(E[] aList, int left, int right) {
        int currentSize = right - left;
        if (currentSize > 0) {
            if (currentSize <= QuickSortWithInsertionSort.MAX_SIZE_FOR_INSERTION_SORT) {
                this.insertionSort(aList, left, right);

            } else {
                int mid = partition(aList, left, right);
                quickSortRecursively(aList, left, mid - 1);
                quickSortRecursively(aList, mid + 1, right);
            }
        }
    }
}
```

Class QuickSortWithInsertionSort

QuickSortWithInsertionSort는 Recursive하게 Sorting을 진행 할 때, size가 20개 보다 작은 경우 InsertionSort로 진행하도록 하는 방식으로 기존에 구현했던 insertionSort를 quickSortRecursively에 조건을 주어 20개 이하의 정렬을 수행할 때에는 insertionSort를 하도록 하는 방식입니다.

주요 구현 내용

```java
package sort;

public abstract class Sort<E extends Comparable<E>> {
    private boolean _nonDecreasingOrder;

    public boolean nonDecreasingOrder() {
        return this._nonDecreasingOrder;
    }

    public void setNonDecreasingOrder(boolean newNonDecreasingOrder) {
        this._nonDecreasingOrder = newNonDecreasingOrder;
    }

    protected void swap(E[] aList, int i, int j) {
        E tempElement = aList[i];
        aList[i] = aList[j];
        aList[j] = tempElement;
    }

    protected int compare(E anElement, E theOtherElement) {
        if (this.nonDecreasingOrder()) {
            return anElement.compareTo(theOtherElement);

        } else {
            return -anElement.compareTo(theOtherElement);

        }
    }

    public Sort(boolean givenSortingOrder) {
        this.setNonDecreasingOrder(givenSortingOrder);
    }

    public abstract boolean sort(E[] aList);
}
```

Class Sort

Sort 클래스에는 Sorting 과정에서 필요한 swap, compare 등의 메서드들이 정의되어 있습니다. sort method 상속한 클래스에서 구현하여 사용하므로 abstract으로 정의되어 있습니다.

결과화면 1

```
<terminated> _Main_AL10_S1_201701975_구건모 [Java Application] C:₩Users₩gmku1₩.p2₩pool₩plugin
<<< 정렬 알고리즘들을 검증하는 프로그램을 시작합니다 >>>

> 오름차순 정렬 프로그램의 검증:
- [무작위 리스트]에 대한 [오름차순]의 [InsertionSort] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [오름차순]의 [QuickSort] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [오름차순]의 [HeapSort] 결과는 올바릅니다.

- [오름차순 리스트]에 대한 [오름차순]의 [InsertionSort] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [오름차순]의 [QuickSort] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [오름차순]의 [HeapSort] 결과는 올바릅니다.

- [내림차순 리스트]에 대한 [오름차순]의 [InsertionSort] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [오름차순]의 [QuickSort] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [오름차순]의 [HeapSort] 결과는 올바릅니다.

> 내림차순 정렬 프로그램의 검증:
- [무작위 리스트]에 대한 [내림차순]의 [InsertionSort] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [내림차순]의 [QuickSort] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [내림차순]의 [HeapSort] 결과는 올바릅니다.

- [오름차순 리스트]에 대한 [내림차순]의 [InsertionSort] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [내림차순]의 [QuickSort] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [내림차순]의 [HeapSort] 결과는 올바릅니다.

- [내림차순 리스트]에 대한 [내림차순]의 [InsertionSort] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [내림차순]의 [QuickSort] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [내림차순]의 [HeapSort] 결과는 올바릅니다.

<<< 정렬 알고리즘들을 검증하는 프로그램을 종료합니다 >>>
```

AL10_S1 에 대한 결과입니다.

결과화면 2

```
<terminated> _Main_AL10_S2_201701975_구건모 (1) [Java Application] C:₩Users₩gmku1₩.p2₩pool₩plugins₩org.eclipse.justj.c
<<< 정렬 알고리즘들을 검증하는 프로그램을 시작합니다 >>>

> 오름차순 정렬 프로그램의 검증:
- [무작위 리스트]에 대한 [오름차순]의 [QuickSortByPivotLeft] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [오름차순]의 [QuickSortByPivotMid] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [오름차순]의 [QuickSortByPivotMedian] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [오름차순]의 [QuickSortByPivotRandom] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [오름차순]의 [QuickSortWithInsertionSort] 결과는 올바릅니다.

- [오름차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotLeft] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotMid] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotMedian] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotRandom] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [오름차순]의 [QuickSortWithInsertionSort] 결과는 올바릅니다.

- [내림차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotLeft] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotMid] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotMedian] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [오름차순]의 [QuickSortByPivotRandom] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [오름차순]의 [QuickSortWithInsertionSort] 결과는 올바릅니다.

> 내림차순 정렬 프로그램의 검증:
- [무작위 리스트]에 대한 [내림차순]의 [QuickSortByPivotLeft] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [내림차순]의 [QuickSortByPivotMid] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [내림차순]의 [QuickSortByPivotMedian] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [내림차순]의 [QuickSortByPivotRandom] 결과는 올바릅니다.
- [무작위 리스트]에 대한 [내림차순]의 [QuickSortWithInsertionSort] 결과는 올바릅니다.

- [오름차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotLeft] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotMid] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotMedian] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotRandom] 결과는 올바릅니다.
- [오름차순 리스트]에 대한 [내림차순]의 [QuickSortWithInsertionSort] 결과는 올바릅니다.

- [내림차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotLeft] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotMid] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotMedian] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [내림차순]의 [QuickSortByPivotRandom] 결과는 올바릅니다.
- [내림차순 리스트]에 대한 [내림차순]의 [QuickSortWithInsertionSort] 결과는 올바릅니다.

<<< 정렬 알고리즘들을 검증하는 프로그램을 종료합니다 >>>
```

AL10_S2 에 대한 결과입니다.

생각할 점

> 퀵 정렬의 피봇을 선택하는 방법 중 어떤 방법이 좋다고 생각하는지?

퀵정렬의 피봇 선택 과정에서 임의의 값을 피봇으로 선택했을 때보다 Median 값을 피봇으로 선정하는 것이 더 좋다고 생각합니다. 피봇값을 Median 값으로 선택할 경우 정렬된 배열에 대해서 발생할 수 있는 Worst Case를 피할 수 있도록 해주기 때문에 Pivot을 Median으로 선택하는 방법이 좋다고 생각하였습니다.

> 퀵 정렬의 과정 중 작은 구간에 대해서 삽입정렬을 이용하는 이유는?

이번 과제에서 구현된 QuickSortWithInsertionSort 에서 퀵정렬 중에 작은 구간에 대해서는 insertionSort를 수행하도록 구현하였었는데 그 이유는 작은 양의 데이터를 처리할 때에는 삽입정렬이 유리하기 때문입니다. 따라서 구간이 작으면 삽입정렬로 대체하고 이외의 경우 재귀적으로 Quicksort를 수행합니다.