자료구조 HW4 Report – Sorting Algorithm

공과대학 컴퓨터공학부 2021-18641 이하동

1. 정렬 알고리즘의 동작 방식

정렬 알고리즘이 올바르게 동작하는지 확인하기 위해, SortingExperiment.java 파일로 복사본을 만들어 스켈레톤 코드를 일부 수정하였다.

```
if(numsize <= 15){
    for (int i = 0; i < value.length; i++) {
        value[i] = rand.nextInt( bound: rmaximum - rminimum + 1) + rminimum;
        System.out.print(value[i] + " ");
        // 각각의 배열에 난수를 생성하여 대입
    }
    System.out.println();
} else {
```

```
if(newvalue.length <= 15) { //작은 배열의 경우 정렬 결과 출력
    for (int i = 0; i < newvalue.length; i++) {
        System.out.print(newvalue[i] + " ");
    }
    System.out.println();
}
else {
    System.out.println((System.currentTimeMillis() - t) + " ms");
}
```

<사진 1. SortingExperiment.java에서 수정된 부분>

랜덤으로 생성된 배열에 대해서 배열 크기가 15이하면 출력하도록 수정함. 아래는 그 결과이다.

```
r 10
-16 33 45 -73 -71 66 -14 -30 -24 -93 -87 -60 -3 32 -58
B
-93 -87 -73 -71 -60 -58 -30 -24 -16 -14 -3 32 33 45 66
I
-93 -87 -73 -71 -60 -58 -30 -24 -16 -14 -3 32 33 45 66
H
-93 -87 -73 -71 -60 -58 -30 -24 -16 -14 -3 32 33 45 66
M
-93 -87 -73 -71 -60 -58 -30 -24 -16 -14 -3 32 33 45 66
Q
-93 -87 -73 -71 -60 -58 -30 -24 -16 -14 -3 32 33 45 66
R
-93 -87 -73 -71 -60 -58 -30 -24 -16 -14 -3 32 33 45 66
```

<사진 2. 정렬 결과 print>

버블 정렬: i(1~n)번째 반복에 대해 0~n-i-1번째 요소까지 바로 옆의 요소와 비교하며 위치를 교환한다. i번째 반복에 대해 n-i번째 요소에 가장 큰 값이 들어가게 됨.

삽입 정렬: 크기가 1개인 배열이 이미 정렬되었다고 보고, 1번째~n-1번째 요소를 적절한 위치에 넣는다. i번째 요소를 삽입해야 되는 상황에서 0~i-1번째까지는 정렬되어 있다.

힙 정렬: 주어진 배열을 힙으로 만든다. 배열의 끝 인덱스부터 i를 1씩 감소시켜가면서 힙의 최댓값을 삭제하고, 삭제된 값을 i번째에 넣는다. 이 과정에서 가장 큰 값이 i번째에 들어가고 새로운 최댓값이 힙의 최상단에 위치하게 됨. 이를 반복하여 정렬된 배열을 얻는다.

병합 정렬: 주어진 배열을 최소 단위로 쪼갠다.(1개짜리 배열은 정렬된 배열이다.) 정렬된 두 배열의 최우선 요소들끼리 비교하는 것을 반복하여 merge한다. (개선된 부분은 discussion에서 논의)

퀵 정렬: pivot의 위치를 찾는 partition에서 적절한 pivot의 위치를 기준으로 나머지 요소들을 양 옆으로 분류한다. Pivot의 위치는 최적이므로 양 옆의 subArray에 대해 퀵 정렬을 재귀호출한다.

기수 정렬: k개의 자릿수를 갖는 countable한 element에 대해, 우선순위가 낮은 자리부터 높은 순으로 기준을 잡아 정렬을 반복한다. 이때 Stability가 보장되는 O(n) 정렬을 이용한다. 높은 우선순위에서 같은 두 요소가 낮은 우선순위를 기준으로 재정렬이 보장되므로 전체적인 정렬이 보장된다. 본 과제에서는 Stable한 정렬로 카운팅 정렬을 이용하였다.

2. 정렬 알고리즘의 동작 시간

통제할 변수를 배열의 크기와, 숫자의 범위 두 가지로 나눈다. 배열의 크기는 시간복잡도를, 숫자의 범위는 중복변수에 대한 성능차이를 실

험하고자 함. 실험횟수/최대값/최소값/평균/표준편차를 추출하여 표로 정리하고, 평균값에 대한 그래프를 그려본다.

배열 크기에 대한 실험: n을 1000, 5000, 10000, 50000, 100000, 500000, 1000000으로, 중복 변수가 거의 없도록(범위를 매우 크게, -1억~1억으로) 설정 후 난수를 생성하여 각 케이스별로 100개의 txt파일을 만든 뒤, 평균값을 구한다. (버블, 삽입 정렬의 경우 시간문제로 5만까지만 시행함.)

중복 변수에 대한 실험: n을 100000으로 고정 후 숫자의 범위를 (-1000,1000), (-10000,10000), (-100000,100000), (-1000000,1000000), (-10000000,10000000)로 설정하여 난수를 생성, 각 케이스별로 100개의 txt파일을 만든 뒤, 평균값을 구한다. 이를 위해 스켈레톤 코드를 일부 수정한 SortingFileTest.java와 난수를 생성하는 MakeRandomValue.java를 만들었다. 이들의 코드는 다소 길어서, 보고서의 끝에 부록으로 첨부하겠음.

MakeRandomValue.java에서는 각기 다른 시드로 만든 난수 배열을 txt파일로 저장하였다. Code 폴더 내부에 range, size폴더를 만들고 그 안에 통제할 변수의 케이스별로 폴더를 만들어 txt파일을 생성하였다.

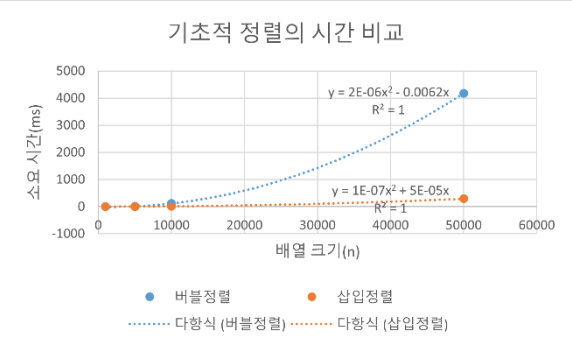
SortingFileTest.java에서는 txt파일을 읽어와 100회에 걸친 실행시간을 구하고(컴파일러 이슈로, 맨앞의 4개는 삭제함.), 이에 대한 평균, 표준편차, 최댓값, 최솟값을 구하였다. (단위는 ms, n은 배열의 크기, k는 범위 크기)

n=1000	평균	표준편차	최댓값	최솟값
버블정렬	0.98	0.271	2	0
삽입정렬	0.08	0.277	1	0
병합정렬	0.12	0.332	1	0
퀵정렬	0.09	0.293	1	0
힙정렬	0.13	0.343	1	0
기수정렬	0.19	0.400	1	0
n=5000	평균	표준편차	최댓값	최솟값
버블정렬	20.81	1.584	26	19
삽입정렬	3.36	0.697	7	2
병합정렬	0.88	0.478	2	0
퀵정렬	0.64	0.480	1	0
힙정렬	0.70	0.456	1	0
기수정렬	0.78	0.565	3	0
n=10000	평균	표준편차	최댓값	최솟값
버블정렬	112.82	9.321	140	93
삽입정렬	11.92	0.861	16	11
병합정렬	1.45	0.541	3	1
퀵정렬	1.21	0.463	2	0
힙정렬	1.34	0.499	2	0
기수정렬	1.41	0.536	3	1
n=50000	평균	표준편차	최댓값	최솟값
버블정렬	4178	735.535	6930	3230
삽입정렬	290.70	16.768	349	258
병합정렬	10.01	3.143	20	6
퀵정렬	4.48	0.648	6	3
힙정렬	7.40	0.642	9	6
기수정렬	5.35	1.628	11	3
n=100000	평균	표준편차	최댓값	최솟값
버블정렬	-	-	-	-
삽입정렬	-	-	-	-
병합정렬	20.08	5.702	35	12
퀵정렬	8.95	0.905	12	7
힙정렬	16.73	2.017	25	14
기수정렬	8.93	2.081	15	6
n=500000	평균	표준편차	최댓값	최솟값
버블정렬	-	-	-	-
삽입정렬	-	-	-	-
병합정렬	118.19	69.063	416	56
퀵정렬	60.75	13.258	116	44
힙정렬	100.95	21.499	211	77
기수정렬	38.62	6.473	58	27

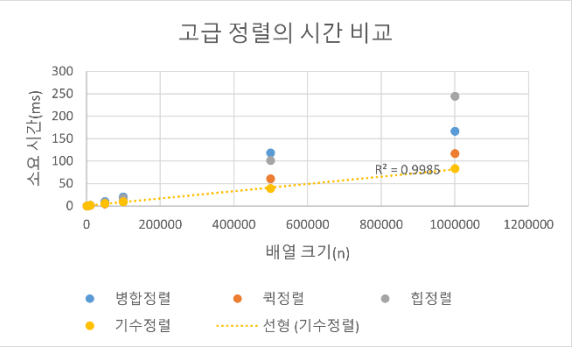
n=1000000	평균	표준편차	최댓값	최솟값
버블정렬	-	-	-	-
삽입정렬	-	-	-	-
병합정렬	166.83	28.503	277	127
퀵정렬	116.70	21.093	187	93
힙정렬	244.54	67.866	577	169
기수정렬	82.97	19.753	157	60

<표 1. 배열 크기에 따른 정렬 시간>

n값과 평균 시간에 대한 그래프를 그려 비교하였다.



<사진 3. 기초적인 정렬의 소요 시간 비교>



<사진 4. 고급 정렬들의 소요 시간 비교>

기초적인 정렬은 모두 2차다항식에 대해 R^2 값이 1로 시간 복잡도가 $O(n^2)$임을 확인하였으며, 성능은 삽입정렬이 더 뛰어났다.

고급 정렬들을 비교한 결과, 기수정렬의 성능이 가장 뛰어났고 그 다음이 퀵 소트, 병합정렬 순으로 성능이 뛰어났다. 특히 기수정렬은 직선에 대한 R^2 값이 1로 시간 복잡도가 $O(n)$임을 확인하였다.

k=1000	평균	표준편차	최댓값	최솟값
버블정렬	3742.50	395.223	5202	3153
삽입정렬	295.59	41.593	519	245
병합정렬	10.28	0.842	15	9
퀵정렬	9.26	0.668	11	8
힙정렬	12.52	1.239	16	11
기수정렬	2.16	0.401	4	2
k=10000	평균	표준편차	최댓값	최솟값
버블정렬	3700.06	453.886	5414	2938
삽입정렬	418.44	178.790	968	245
병합정렬	11.03	1.410	17	9
퀵정렬	8.91	0.790	12	8
힙정렬	13.57	1.763	21	11
기수정렬	2.97	0.354	4	2
k=100000	평균	표준편차	최댓값	최솟값
버블정렬	3465.63	387.835	5699	2925
삽입정렬	289.15	41.250	570	240
병합정렬	10.31	0.987	14	8
퀵정렬	9.89	0.656	12	8
힙정렬	12.65	0.831	16	11
기수정렬	3.76	0.644	5	2
k=1000000	평균	표준편차	최댓값	최솟값
버블정렬	3583.08	1056.382	7746	2941
삽입정렬	347.64	52.179	533	269
병합정렬	10.28	0.790	14	9
퀵정렬	9.97	0.807	14	9
힙정렬	12.68	0.932	16	10
기수정렬	4.47	0.695	7	3
k=10000000	평균	표준편차	최댓값	최솟값
버블정렬	4483.71	851.991	6681	3444
삽입정렬	325.36	22.865	412	298
병합정렬	10.19	0.748	13	9
퀵정렬	10.08	0.626	11	9
힙정렬	12.16	0.829	16	11
기수정렬	6.23	1.176	9	4

<표 2. 숫자 범위에 따른 정렬 시간>

k값과 평균 시간에 대한 표를 그려 비교하였다.

k	버블정렬	삽입정렬	병합정렬	퀵정렬	힙정렬	기수정렬
1000	3742.50	295.59	10.28	9.26	12.52	2.16
10000	3700.06	418.44	11.03	8.91	13.57	2.97
100000	3465.63	289.15	10.31	9.89	12.65	3.76
1000000	3583.08	347.64	10.28	9.97	12.68	4.47
10000000	4483.71	325.36	10.19	10.08	12.16	6.23

<표 3. 정렬 종류에 따른 범위 크기와 평균 소요시간>

기수 정렬의 경우 평균 시간이 log(k)에 비례하는 것을 확인하였으나, 그 외에는 큰 경향성을 보이지 않았다.

3. Discussion

3-1. 범위 실험결과의 해석

기수 정렬의 평균 시간이 log(k)에 비례한 점은, 기수 정렬의 시간복잡도가 자릿수에 대한 일차식임을 실험적으로 뒷받침한다.

이론적으로 퀵 소트는 중복된 요소가 있을 때 비효율적인 동작을 보인다. 그러나 표3에서는 이 점이 드러나지 않아, k=100를 포함하여 퀵 소트의 성능을 실험하였다.

```
퀵 정렬에 대한 범위 100의 정렬 시간 평균, 표준편차, 최대, 최소
[16, 22, 17, 19, 24, 17, 35, 23, 21, 22, 19, 21, 18, 24, 18,
17.708333333333332 2.6073445522283962 35 15
==================================
퀵 정렬에 대한 범위 1000의 정렬 시간 평균, 표준편차, 최대, 최소
[6, 6, 6, 8, 6, 6, 6, 6, 5, 6, 6, 7, 6, 6, 6, 6, 7, 6, 7, 6,
6.333333333333333 0.7350856257559273 8 5
==================================
퀵 정렬에 대한 범위 10000의 정렬 시간 평균, 표준편차, 최대, 최소
[7, 5, 8, 7, 6, 6, 7, 6, 7, 6, 6, 6, 7, 6, 6, 6, 5, 7, 7, 7,
6.270833333333333 0.6879284948561786 8 5
==================================
퀵 정렬에 대한 범위 100000의 정렬 시간 평균, 표준편차, 최대, 최소
[7, 7, 7, 6, 8, 7, 9, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7.104166666666667 0.6237267732904348 9 6
==================================
퀵 정렬에 대한 범위 1000000의 정렬 시간 평균, 표준편차, 최대, 최소
[8, 8, 6, 7, 8, 8, 8, 6, 7, 7, 6, 6, 7, 7, 7, 8, 7, 7, 6, 8,
7.052083333333333 0.7014239902049891 9 6
==================================
퀵 정렬에 대한 범위 10000000의 정렬 시간 평균, 표준편차, 최대, 최소
[6, 6, 7, 7, 7, 7, 8, 7, 8, 7, 7, 7, 7, 7, 6, 7, 7, 7, 7, 7,
7.625 1.6874147151938175 18 6
==================================
```

k=100에서 퀵 소트의 성능이 급감하는 것을 확인할 수 있다.

3-2. 개선된 병합 정렬과 기존 방식의 비교

tmp배열을 매번 새로 만들어 다시 value에 되쓰는 과정을 거치기보단, 처음부터 원본 배열을 복사한 두 번째 배열을 이용한다. 비교 결과를 대입하는 대상으로 두 배열을 번갈아가며 택하는 방식으로 Merge를 수행한다. (출처: 2021 기출문제)

```
static void betterMergeSort(int[] value, int[] tmp, int s, int e) { //
    if (s < e) { //재귀 호출을 반복해가며 크기가 0또는 1인 부분 배열에서 되돌아가도록 한다.
        int m = (s + e) / 2;
        betterMergeSort(tmp, value, s, m);
        betterMergeSort(tmp, value, m + 1, e);
        // 위의 두 부분배열은 base 조건이나, merge() 를 통해 정렬된 두 배열이 된다.
        betterMerge(tmp, value, s, m, e);
    } else return;
}
```

```
static void betterMerge(int[] tmp, int[] value, int s, int m, int e) {
    int i = s;
    int j = m + 1;
    int t = s;
    while (i <= m && j <= e) {
        if (tmp[i] <= tmp[j])
            value[t++] = tmp[i++];
        else
            value[t++] = tmp[j++];
    }
    while (i <= m)
        value[t++] = tmp[i++];
    while (j <= e)
        value[t++] = tmp[j++];

}
```

<사진 5. 개선된 병합 정렬의 코드>

```
개선된 병합 정렬에 대한 크기 50000의 정렬 시간 평균, 표준편차, 최대, 최소
4.052083333333333 0.6218781682454788 8 3
==================================
개선된 병합 정렬에 대한 크기 100000의 정렬 시간 평균, 표준편차, 최대, 최소
8.458333333333334 0.7527726527090811 11 7
==================================
개선된 병합 정렬에 대한 크기 500000의 정렬 시간 평균, 표준편차, 최대, 최소
88.03125 113.58024783107501 1103 45
==================================
개선된 병합 정렬에 대한 크기 1000000의 정렬 시간 평균, 표준편차, 최대, 최소
131.14583333333334 24.73181593207919 265 103
```

```
병합 정렬에 대한 크기 50000의 정렬 시간 평균, 표준편차, 최대, 최소
6.677083333333333 0.7745258878031087 9 5
==================================
병합 정렬에 대한 크기 100000의 정렬 시간 평균, 표준편차, 최대, 최소
12.760416666666666 1.4563457633598353 16 10
==================================
병합 정렬에 대한 크기 500000의 정렬 시간 평균, 표준편차, 최대, 최소
82.40625 43.59009499391975 475 59
==================================
병합 정렬에 대한 크기 1000000의 정렬 시간 평균, 표준편차, 최대, 최소
179.88541666666666 24.042449210408822 263 138
```

<사진 6. 소요 시간의 비교>

성능이 개선된 것을 확인할 수 있다.

3. 중복 요소를 고려한 퀵소트 개선

퀵 소트가 중복된 요소들이 많은 경우 비효율적인 이유는, partition 결과가 한 쪽으로 몰리기 때문이다. 따라서 pivot==element일 때 0.5의 확률을 적용해주면 중복 요소들에 대한 결과가 개선될 것으로 예측된다.

```
if (value[startOfAreaThird] < pivot
    || (value[startOfAreaThird] == pivot && Math.random() < 0.5)  ) {
```

<사진 7. partition 메소드의 수정>

```
퀵 정렬에 대한 범위 100의 정렬 시간 평균, 표준편차, 최대, 최소
17.46875 3.866055040858493 7 50 14
```

```
개선된 퀵 정렬에 대한 범위 100의 정렬 시간 평균, 표준편차, 최대, 최소
16.96875 1.2769463987431726 25 15
```

<사진 8. 개선된 퀵 정렬과의 비교>

평균값과 표준편차가 모두 감소하였으며, 이로부터 성능 개선이 일어났음을 확인할 수 있다.

4. 참고문헌

쉽게 배우는 자료구조(문병로 저)

강의노트

5. 부록 – SortingFileTest.java, MakeRandomValue.java의 코드

```
static void makeSizeData() {
    try {
        String dir = "code/size/";
        int minimum = -100000000;
        int maximum = 100000000;
        Integer[] cases = {1000, 5000, 10000, 50000, 100000, 500000, 1000000};

        for (int Case: cases) {
            System.out.println(Case);
            for(int i = 1; i<=100; i++) {
                Random rand = new Random(i);    // 난수 인스턴스를 생성한다. 시드값 고정
                FileWriter fileWriter = new FileWriter( fileName: dir + Case + "/" + i + ".txt");
                for (int j = 0; j<Case-1; j++) {
                    fileWriter.write( str: rand.nextInt( bound: maximum - minimum + 1) + minimum + "\n");
                }
                fileWriter.write( str: rand.nextInt( bound: maximum - minimum + 1) + minimum+"");
                fileWriter.close();
            }
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```
static void makeRangeData() {
    try {
        String dir = "code/range/";
        int Size = 100000;
        Integer[] cases = {100, 1000, 10000, 100000, 1000000, 10000000};

        for (int Case: cases) {
            System.out.println(Case);
            int minimum = -1 * Case/2;
            int maximum = Case/2;
            for(int i = 1; i<=100; i++) {
                Random rand = new Random(i);    // 난수 인스턴스를 생성한다. 시드값 고정
                FileWriter fileWriter = new FileWriter( fileName: dir + Case + "/" + i + ".txt");
                for (int j = 0; j<Size-1; j++) {
                    fileWriter.write( str: rand.nextInt( bound: maximum - minimum + 1) + minimum + "\n");
                }
                fileWriter.write( str: rand.nextInt( bound: maximum - minimum + 1) + minimum+"");
                fileWriter.close();
            }
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

<사진 9. MakeRandomValue.java의 두 메소드 코드>

```java
static long returnBubbleTime(int[] value) {
    long t = System.currentTimeMillis();
    int[] newValue = (int[]) value.clone();
    newValue = DoBubbleSort(newValue);
    return (System.currentTimeMillis() - t);
}
```

<사진 10. SortingFileTest.java의 정렬 한 번에 걸린 시간을 메소드화한 코드>

```java
static void BubbleSizeTest() {
    try {
        int[] value;
        List<Long> times;
        String dir = "code/size/";
        String resultDir = "code/result/size/BubbleSizeTest.txt";
        FileWriter resultWriter = new FileWriter(resultDir);
        Integer[] cases = {1000, 5000, 10000, 50000};//, 100000};//, 500000, 1000000};
        for (int Case : cases) {
            File path = new File( pathname: dir + Case);
            File[] fileList = path.listFiles();
            times = new ArrayList<>();
            for (File file : fileList) {
                value = new int[Case];
                Scanner scanner = new Scanner(file);
                for (int i = 0; i < Case; i++) {
                    value[i] = scanner.nextInt();
                }
                times.add(returnBubbleTime(value));
            }
            System.out.println("버블 정렬에 대한 크기 " + Case + "의 정렬 시간 평균, 표준편차, 최대, 최소");
            times.remove( index: 0);
            times.remove( index: 0);
            times.remove( index: 0);
            times.remove( index: 0); //앞부분 비정상적인 값 제거.
            System.out.println(times);
            double mean = returnMeanOfList(times);
            long max = Collections.max(times);
            long min = Collections.min(times);
            double stdev = returnStdevOfList(times, mean);
            System.out.println(mean + " " + stdev + " " + max + " " + min);
            System.out.println("================================");
            String result = "버블 정렬에 대한 크기 " + Case + "의 정렬 시간 평균, 표준편차, 최대, 최소\n" + mean +
                    "\n================================\n";
            resultWriter.write(result);
        }
        resultWriter.close();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

<사진 11. SortingFileTest.java의 Size에 대한 테스트 코드, 다른 정렬에 대해서도 같은 로직 적용>

```java
static void BubbleRangeTest() {
    try {
        int[] value;
        List<Long> times;
        String dir = "code/range/";
        String resultDir = "code/result/range/BubbleRangeTest.txt";
        FileWriter resultWriter = new FileWriter(resultDir);
        int Size = 50000;
        Integer[] cases = {100, 1000, 10000, 100000, 1000000, 10000000};
        for (int Case : cases) {
            File path = new File( pathname: dir + Case);
            File[] fileList = path.listFiles();
            times = new ArrayList<>();
            for (File file : fileList) {
                value = new int[Size];
                Scanner scanner = new Scanner(file);
                for (int i = 0; i < Size; i++) {
                    value[i] = scanner.nextInt();
                }
                times.add(returnBubbleTime(value));
            }
```

```java
            System.out.println("버블 정렬에 대한 범위 " + Case + "의 정렬 시간 평균, 표준편차, 최대, 최소");
            times.remove( index: 0);
            times.remove( index: 0);
            times.remove( index: 0);
            times.remove( index: 0); //앞부분 비정상적인 값 제거.
            System.out.println(times);
            double mean = returnMeanOfList(times);
            long max = Collections.max(times);
            long min = Collections.min(times);
            double stdev = returnStdevOfList(times, mean);
            System.out.println(mean + " " + stdev + " " + max + " " + min);
            System.out.println("==================================");
            String result = "버블 정렬에 대한 크기 " + Case + "의 정렬 시간 평균, 표준편차, 최대, 최소\n" + mean + " "
                    "\n==================================\n";
            resultWriter.write(result);
        }
        resultWriter.close();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

<사진 12. SortingFileTest.java의 range에 대한 테스트 코드, 다른 정렬에 대해서도 같은 로직 적용>