Sorting Algorithm의 성능 비교

27기 한준희

1. 서론

1.1 연구 목적

Sorting algorithm, 즉 어떠한 데이터셋을 특정 기준에 맞춰 정렬하는 알고리즘은 종류가 매우 다양하다. 이중 실수 데이터셋을 각 데이터 값의 크기에 따라 오름차순 및 내림차순으로 정렬하는 알고리즘은 Bubble sort, Insertion Sort, Selection Sort, Merge sort, Quick sort 등이 존재한다. 이렇게 다양한 정렬 알고리즘이 개발된 이유는 바로 각 알고리즘 간의 성능 차이 때문이다. 정렬 알고리즘의 성능은 크게 두 가지로 평가할 수 있는데, 첫 번째로 알고리즘의 속도, 그리고 두 번째로 알고리즘의 메모리 소비량이다. 알고리즘의 속도는 당연하게도 빠를 수록 성능을 높게 평가하고, 메모리의 경우 사용량이 적을수록 성능을 높게 평가하게 된다. 알고리즘의 속도를 평가할 때에는 시간 복잡도를 사용하는데, 최악의 경우를 나타내는 Big-O 표기법을 사용하면 Bubble sort, Insertion sort 및 Selection sort는 $O(n^2)$, Merge sort와 Heap Sort는 $O(nlogn)$의 시간 복잡도를 가진다. Quick sort의 경우 예외 없이 pivot 값에 따라 배열을 절반으로 나누는 Merge sort 와는 달리 pivot 값에 따라 유동적으로 배열이 분할되므로 평균적으로 $O(nlogn)$의 시간 복잡도를 가지지만, 배열이 거의 정렬 완료된 상태인 경우 $O(n^2)$까지 높아진다. 하지만 시간 복잡도 만을 사용하여 정렬 알고리즘의 속도를 비교하게 되면 단편적으로 보았을 때 메모리 사용량은 같지만 $O(nlogn)$인 Heap sort 대신 Insertion sort, Selection sort과 같이 $O(n^2)$의 시간 복잡도를 가지는 알고리즘은 사용할 이유가 없으며, $O(nlogn)$ 이 고정적인 Merge sort에 비해 $O(n^2)$ 까지 올라가는 Quick sort를 사용할 이유 또한 없다고 판단할 수 있다. 이에 몇 가지 변인을 설정하고 통제하여 시간 복잡도로는 나타낼 수 없는 각 정렬 알고리즘의 성능을 분석하고 비교하고자 한다.

2. 본론

2.1 실험 설정

2.1.1 실험변수 설정

성능 비교 실험을 진행하기 앞서 결과값에 대한 오차를 최소화하기 위해 두 가지 변수를 설정하였다. 첫 번째는 배열의 상태이다. 우선 배열의 길이에 의한 실험 결과의 차이를 확인하기 위해 100, 100,000, 1,000,000, 그리고 10,000,000일 때 각각 실험을 진행했다. 또한, 각 배열의 길이마다 랜덤 상태, 이미 정렬 완료된 상태, 그리고 정 반대로 정렬된 상태의 세 가지 배열 상태를 사용하여 실험을 진행했다. 배열을 랜덤하게 생성하는 과정에서는 Random의 seed를 1로 고정한 상태로 -1,000,000,000이상 1,000,000,000이하의 정수를 각 인덱스 값에 부여하였다. Random 함수로 생성한 정수의 범위를 여러 가지로 실험하지 않고 -10,000,000이상 10,000,000이하의 정수로 고정한 이유는 정렬 알고리즘은 값의 대소 비교만 하면 되기 때문에 절대적인 값이 실험 결과에 영향을 주지 않을 것이라고 판단했기 때문이다. 또한, 실험한 배열의 크기 중 가장 큰 것이 10,000,000이었기에 2,000,000,000 개의 정수 범위라면 생성한 수가 겹칠 확률이 매우 낮다고 판단했다. 아래 이미지는 크기 100,000의 랜덤 배열 생성 과정에서 사용된 코드이다.

```
int size =  100000;
int[] arr = new int[size];
Random rand = new Random(seed:1);

for (int i = 0; i < size; i++) {
    arr[i] = rand.nextInt(bound:2000000001) - 1000000000;
}
```

두 번째 변수는 컴퓨터의 성능 차이이다. 코드를 실행하는 컴퓨터의 성능이 실험 결과에 영향을 미칠 수 있기 때문에 최대한 실행 환경을 통일하고자 했다. 하지만 컴퓨터의 성능은 완벽하게 통제할 수 없으므로 매 실험마다 10번씩 코드를 실행하고 나온 수치의 평균값을 결과에 사용하여 오차를 줄이고자 하였다.

2.1.2 실험 방법 설정

본 실험에서는 Selection sort, Insertion sort, Quick sort, Merge sort, Heap sort으로 이루어진 5가지의 정렬 알고리즘들의 성능을 비교하기 위해 실제 실행 시간을 측정하는 방법을 사용했다. 아래의 코드처럼 자바의 System 클래스에 있는 currentTimeMillis() 함수를 사용하여 정렬 알고리즘을 실행하기 전과 정렬이 완료되고 난 이후 각각 시간을 측정하여 그 차를 구하는 방식으로 소요 시간을 계산했다.

```
long startTime = System.currentTimeMillis();
// 정렬 알고리즘 실행
long endTime = System.currentTimeMillis();
long elapsedTime = endTime - startTime;
System.out.println(elapsedTime + "밀리초");
```

2.2 실험 결과

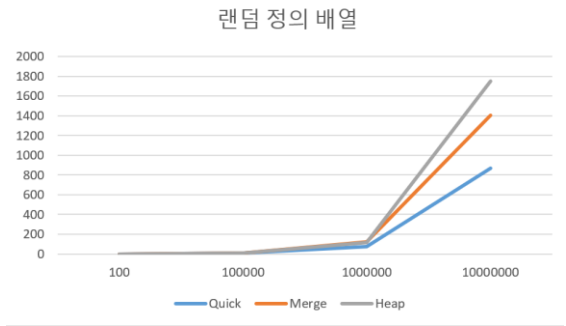
2.2.1 랜덤하게 정의된 배열

배열을 랜덤하게 정의할 때 배열의 길이를 2.1.1에 서술한 것처럼 증가시켰고, 이때 발생하는 정렬 알고리즘 간의 성능 차이를 기록하였다. 또한, $O(nlogn)$의 시간 복잡도를 가지는 알고리즘들의 성능을 보다 직관적으로 비교하기 위해 Quick sort, Merge sort, 그리고 Heap sort의 소요 시간을 그래프로 나타내어 비교했다.

랜덤 상태의 배열에 따른 결과

(millisecond)	100	100,000	1,000,000	10,000,000
Selection	0	1344	130752.2	5분 이상

Insertion	0	2839.8	290019.1	5분 이상
Quick	0	7.1	77.2	870
Merge	0.1	11.7	124.1	1408.7
Heap	0.1	9.8	119.4	1751.3



우선 Selection sort와 Insertion sort는 배열 길이가 작을 때 0 밀리초를 기록하여, $O(nlogn)$ 을 가지는 Merge sort 및 Heap sort보다 평균적으로 아주 약간 빠른 속도를 보였다. 이는 크기가 작은 배열에서는 재귀 및 분할 방식으로 정렬을 하는 것보다 단순히 순회하는 것이 더 빠르기 때문이라고 사료된다. 하지만 10,000,000길이 이상부터는 Selection 및 Insertion sort 둘 다 시간이 과도하게 오래 걸려 측정에 반영하지 않았다. Selection sort는 Insertion sort의 절반 가량 빨랐으며, Quick sort는 다른 $O(nlogn)$ 알고리즘에 비해 모든 배열의 길이에서 압도적으로 빨랐다. Merge sort는 상대적으로 작은 길이의 배열에서는 Heap sort보다 약간 느렸지만, 길이가 10,000,000일 때 Heap sort보다 좋은 성능을 보였다.

2.2.2 이미 정렬된 배열

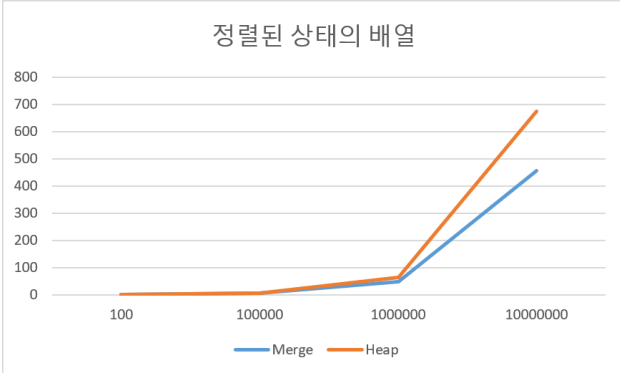
이미 정렬된 배열의 상태를 만들기 위해 배열의 각 인덱스에 그 인덱스 값을 사용했다. 위에서 말한 바와 같이 절대적인 값은 중요하지 않기 때문에 아래와 같은 코드를 사용하여 배열을 정의했다.

```
for (int a = 0; a < size; a++) {
    arr[a] = a;
}
```

각 정렬 알고리즘에 대해 시행한 결과 아래와 같은 결과를 얻을 수 있었다.

정렬된 상태의 배열에 따른 결과

(millisecond)	100	100,000	1,000,000	10,000,000
Selection	0	723.3	71825	5분 이상
Insertion	0	0.3	1.8	3.7
Quick	0.1	X	X	X
Merge	0	5.8	46.9	456.2
Heap	0.1	6.1	64.9	675.1



이미 정렬된 상태의 배열에서는 랜덤하게 정의된 배열이랑 확연히 다른 결과가 나왔다.

Selection sort는 두번째 for 문에서 순회해야 하는 인덱스의 개수가 현저히 줄어들어 랜덤 상태의 배열을 정렬할 때에 비해 걸린 시간이 절반 정도 줄었다. Insertion sort가 가장 차이가 명확했는데, 10,000,000길이의 배열도 3.7 밀리초 만에 정렬하면서 Merge sort 및 Quick sort 보다도 훨씬 빠른 속도를 보였다. 이는 이미 정렬된 상태이기에 n번째 인덱스의 값을 찾으려고 순회하는 for 문이 한 번 밖에 실행되지 않아 거의 $O(n)$ 의 시간 복잡도로 정렬을 했기 때문이라고 판단된다. Merge sort의 경우 랜덤한 상태의 배열일 때와는 달리 항상 Heap sort 보다 속도가 빨랐으며, Quick sort는 배열의 길이가 100,000 이상이 되자 Maximum recursion depth exceeded, 즉 재귀 호출의 최대 개수 제한 초과를 하여 측정이 불가했다. 실험에서 사용한 코드는 Pivot을 가장 왼쪽의 값으로 정했는데, 이로 인해 오름차순으로 정렬했을 때 배열의 길이만큼 Quick sort 함수가 재귀 호출되어 에러를 발생시켰다.

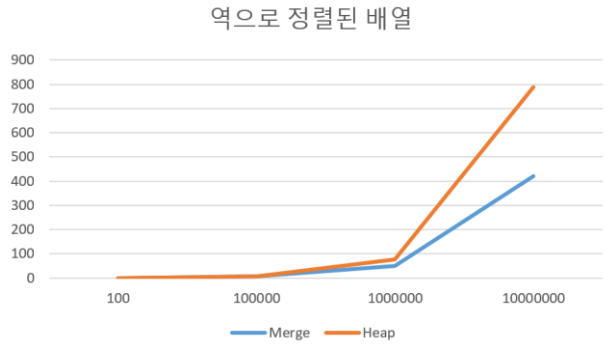
2.2.3 역으로 정렬된 배열

역으로 정렬된 배열의 경우 2.2.2 와 반대로 아래와 같이 배열을 생성했고, 2.2.1, 2.2.2와 동일하게 측정을 진행했다.

```
for (int a = 0; a < size; a++) {
    arr[a] = size - a;
}
```

역으로 정렬된 상태의 배열에 따른 결과

(millisecond)	100	100,000	1,000,000	10,000,000
Selection	0.1	1408.9	179890	5분 이상
Insertion	0	5714.2	5분 이상	5분 이상
Quick	0.1	X	X	X
Merge	0.2	7	48.8	420.2
Heap	0.1	7.8	76.4	787.9



역으로 정렬된 배열에서도 마찬가지로 큰 차이를 발견할 수 있었다. Selection sort 및 Insertion sort는 랜덤하게 정의된 배열로 실험했을 때와 비슷한 양상을 보였는데, Insertion sort의 경우 2배 정도 느려졌으며, 1,000,000부터 5분 이상의 시간이 소요되면서 3 가지의 배열의 상태 중 역으로 정렬되었을 때 최악의 성능을 보였다. 이는 n번째 값이 항상 맨 앞으로 보내져야 하므로 두 번째 for 문이 매번 최대로 실행되어 발생한 결과라고 볼 수 있다. Quick sort는 이미 정렬된 배열의 경우와 마찬가지로 100,000 부터 Maximum recursion depth exceeded 에러가 발생했으며, Merge sort는 Heap sort보다 모든 배열의 길이에서 더 좋은 성능을 보였다.

3. 결론

3.1 실험 결론

위와 같은 실험을 통해 5가지 정렬 알고리즘의 성능 평가 및 비교를 진행하고, 도출된 결과로 다양한 결론을 내릴 수 있었다. $O(n^2)$의 시간 복잡도를 가지는 Selection sort 및 Insertion sort는 대부분의 경우 Selection sort가 더 좋은 성능을 보였다. 이는 Insertion sort의 알고리즘 특성상, 길이가 긴 배열에서 "sorted" 된 부분의 개수가 점차 증가하므로 두 번째 for 문의 실행 횟수도 증가하는 반면 Selection sort는 swap을 진행할 인덱스가 항상 정해져 있기에 발생한 결과이다. 하지만 정렬된 상태의 배열에서는 상술한 이유로 인해 Insertion sort가 압도적으로 빠른 성능을 보였다. $O(nlogn)$의 시간 복잡도를 가지는 Quick sort, Merge sort, 그리고 Heap sort는 랜덤 상태의 배열에서 Quick sort, Heap sort, Merge sort의 순서대로 빨랐는데, Quick sort는 이미 정렬된 배열 및 역으로 정렬된 배열에서 일정 크기 이상의 배열은 recursion limit 초과로 정렬을 할 수 없었다. 또한, Heap sort는 정렬 및 역으로 정렬된 배열에서 Merge sort보다 나은 성능을 보였는데, 이는 Heap 자료구조에 값을 추가할 때 upheap 과정에서 아무런 swap이 발생하지 않았기 때문이라고 판단했다.

각 알고리즘의 배열에 따른 성능 비교를 하면, Selection sort 및 Insertion sort는 역으로 정렬된 배열에서 최악의 성능, 그리고 정렬된 배열에서 최대의 성능을 보였다. Merge sort 및 Heap sort는 랜덤하게 정의된 배열에서 최악, 정렬된 배열에서 최대의 성능을 보였다는 차이가 있다. 또한 Quick sort의 경우 배열의 상태에 따라 정렬을 하지 못할 수도 있다는 변수가 있으며, 만약 하더라도 시간 복잡도가 일정하지 않다는 문제점이 있다는 것을 파악할 수 있었다.

3.2 문제점 및 보완 가능성

위 실험은 타당성 및 정확도를 높이기 위해 몇 가지 보완 가능한 부분이 있다. 우선 첫 번째로 Quick sort의 코드 보완이다. 실험에서 사용한 Quick sort 알고리즘은 pivot으로 항상 가장 왼쪽에 있는 원소를 사용하는데, 이는 정렬 및 역으로 정렬된 배열에서 모든 인덱스마다 재귀 호출을 하는 악영향을 주었다. 만약 Quick sort 알고리즘의 pivot을 배열의 중간지점, 혹은 랜덤한 인덱스로 하였다면 이를 방지하여 정렬 및 역으로 정렬된 배열에서도 시간을 비교할 수 있었을 것이다. 두번째는 배열의 길이가 100인 경우에 시간 측정의 문제점이다. 배열의 길이가 100일 때 모든 정렬 알고리즘은 0 밀리초, 혹은 0.1 밀리초를 기록했다. 하지만 이는 완전히 정확하다고 보기 어려우며, 단지 컴퓨터상의 변화로 발생했을 가능성도 존재한다. 따라서 다른 측정 방법을 통해 이를 비교했다면 더욱 타당한 결과를 얻을 수 있었을 것이다. 마지막으로 Heap sort 알고리즘의 정렬 방식이다. 실험에서 사용한 Heap sort 알고리즘은 Heap 자료구조 클래스를 사용하여 for 문을 사용해 insert 및 remove를 하는 방식이다. 이 때문에 정렬된 배열을 구하기 위해서는 새로운 배열 하나를 정의해야 하며, 다른 정렬 알고리즘들과는 달리 argument로 넘겨준 배열 자체가 바뀌는 것이 아닌 정렬된 값을 가지는 새로운 배열을 리턴 하는 형식이다. 이로 인해 발생된 오차를 해결하였다면 실험의 정확도를 높일 수 있었을 것이다.

4. 부록

1. 알고리즘 속도 비교에 사용된 코드

```
import java.util.Random;

public class CompareSort {
    public static void main(String[] args) {
        int size = 100000;
        int[] arr = new int[size];
        Random rand = new Random(1);

        double sum = 0;
        for (int i = 0; i < 10; i++) {

            // for (int a = 0; a < size; a++) {
            //        arr[a] = rand.nextInt(2000000001) -
1000000000;
            // }

            // for (int a = 0; a < size; a++) {
            //     arr[a] = a;
            // }

            for (int a = 0; a < size; a++) {
                arr[a] = size - a;
            }

            long startTime = System.currentTimeMillis();
```

```
            // AllSorts.Selection(arr);
            // AllSorts.Insertion(arr);
            QuickSort.Quick(arr, 'a');
            // MergeSort.Merge(arr, 'a');
            // int [] heap = HeapSort.Heap(arr);

            long endTime = System.currentTimeMillis();

            long elapsedTime = endTime - startTime;
            sum + = elapsedTime;
            System.out.println(i + 1 + "번째: " +
elapsedTime + " 밀리초");
        }
        System.out.println();
        System.out.println("평균: " + sum / 10 + " 밀리초
");
    }
}
```