Date Structure HW4

2021-15738 양석훈

1. Introduction

이번 과제에서는 다양한 정렬 알고리즘을 직접 JAVA로 구현해보고, 이 들의 성능을 직접 비교해 봄으로써 각 정렬 알고리즘의 특성과 중요성을 알아보았다. Bubble Sort, Insertion Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort의 6가지 정렬을 구현해보았으며, 다양한 경우에 대해서 이 알고리즘들이 정렬을 수행하는데 걸린 시간을 비교해보았다. 성능 비교를 위한 코드의 실행은 i7 core의 Window 10 환경에서 Intellij IDE를 사용해서 진행되었다.

2. Theoretical Background

2.1 Bubble Sort

Bubble Sort는 정렬에 O(n^2)의 시간 복잡도를 가지는 정렬 알고리즘이다. Bubble Sort의 핵심은 인접한 두 원소의 크기를 비교하여 앞의 원소가 더 크다면, 둘의 자리를 바꾸는 것이다. 이 작업을 첫 원소부터 마지막 원소까지 연쇄적으로 계속 비교해 나간다면, 마지막 원소의 크기가 제일 커지게 된다. 이후, 다시 첫 원소부터 비교를 시작해서, 마지막 바로 앞 원소까지 정렬을 하면 마지막 두 개의 원소가 결정되는 것이다. 이 작업을 계속 반복해서 진행함으로써 모든 원소가 정렬되도록 하는 것이 Bubble Sort이다. 이는 결국 Best case, Average case, Worst case 모두 O(n^2)의 시간 복잡도를 가지게 된다.

2.2 Insertion Sort

Insertion Sort는 각 원소를 자신의 알맞은 자리에 삽입함으로써 정렬하는 알고리즘이다. 자세히 설명하자면, 입력 받은 배열의 앞에서부터 차례대로 정렬을 해 나갈 것인데, 이미 정렬된 배열과 아직 정렬되지 않은 배열로 나뉜다고 해보자. 이때, 아직 정렬되지 않은 배열의 첫 원소를 이미 정렬된 배열의 원소들과 차례대로 비교하여서 올바른 위치에 삽입하는 것이다. 이 과정을 재귀적으로 반복하면 정렬된 배열을 얻을 수 있다. Insertion Sort는 Best case는 모든 원소가 이미 정렬되어 있을 경우 n번의 작업으로 정렬이 끝나므로 O(n), Average case, Worst case에서는 모두 O(n^2)의 시간 복잡도를 가지는 것을 쉽게 확인할 수 있다.

2.3 Heap Sort

Heap Sort는 Heap 자료구조를 가지고, 정렬을 수행하는 알고리즘이다. 이를 위해서, 먼저 배열을 Heap의 형태로 변경시켜주는 Build Heap 작업을 먼저 수행한다. 코드를 구현할 때는, 이를 아래에 있는 노드부터 percolateDown 알고리즘을 실행하며 쭉 올라오는 방식으로 구현하였다. 이후, 만들어진 Heap에서 최댓값을 뽑아 정렬되지 않은 배열의 마지막에 삽입하여

percolateDown 알고리즘을 실행하여 Heap 특성을 유지한다. 이 과정에서 Heap Sort는 Best case, Average case, Worst case 모두 O(nlogn)의 시간 복잡도를 가지는 것을 쉽게 확인할 수 있다.

2.4 Merge Sort

Merge Sort는 대표적인 재귀 알고리즘으로, Divide & Conquer의 한 예시이다. 먼저, 배열의 크기가 1인 경우는 정렬이 된 것과 다름없기에 배열 자체를 돌려주면 된다. 또, 배열의 크기가 n이라면, n/2의 크기의 배열 두 개로 나누어 각각에 대해 Merge Sort를 시행한 이후, 각 배열의 앞 원소부터 크기를 비교해가며 작은 원소를 넣는 방식으로 병합을 하는 방식이다. 이렇게 한다면, 재귀적으로 크기 n짜리의 배열이 정렬된 배열임을 알 수 있다. 이 과정에서 Merge Sort는 Best case, Average case, Worst case 모두 O(nlogn)의 시간 복잡도를 가지는 것을 확인할 수 있다.

2.5 Quick Sort

Quick Sort 역시도 재귀적인 알고리즘이다. Pivot을 기준으로 잡아서 Pivot 보다 작은 원소들은 Pivot의 왼쪽으로, Pivot보다 큰 원소들은 Pivot의 오른쪽으로 보낸 이후, Pivot을 기준으로 배열의 좌, 우를 분할한다. 이후, 각각의 분할된 배열에서 같은 작업을 반복하다가 배열의 크기가 1인 경우, 배열 자체를 리턴 해주면 전체 배열 역시도 정렬된 배열이 되는 것이다. 구현할 때는 partition 함수에서 배열의 첫 번째 원소를 pivot으로 잡고 구현을 시작하도록 하였다. Quick Sort의 경우, Best case, Average case에는 O(nlogn)의 시간 복잡도를 가지지만, 모든 원소들이 이미 정렬되거나 동일 원소가 많은 경우에는 partition이 한쪽으로 쏠리게 되므로 Worst case는 O(n^2)의 시간 복잡도를 가진다.

2.6 Radix Sort

Radix Sort는 일의 자리수부터 비교를 시작해서 최대 자리 수까지 비교해가며 매 순간 정렬을 하는 알고리즘이다. 단, 이미 정렬된 자리 수에 대해서 새롭게 정렬이 시행되더라도, 데이터들 간의 상대적인 배치 순서는 변하지 않아야 한다. 즉, 안정성을 가진 정렬을 통해서 진행하여야 한다. 본 구현에서는 Counting Sort를 사용해서 구현하였다. Radix Sort는 모든 경우에 시간 복잡도 O(n)을 가지는 구현이다. 즉, 정수형 데이터들의 정렬 속도가 매우 빠르다. 물론, 실수 데이터들의 정렬에서는 사용할 수 없지만, 정수와 같은 특수 케이스에서는 정렬 속도의 이점으로 인해 유용한 알고리즘이라고 할 수 있다.

2.7 Theoretical Sorting Time

위에 소개된 6가지 정렬 알고리즘들의 수행 시간을 표로 정리한다면 아래 표와 같다. 아래 표는 입력 숫자의 개수를 n이라고 하였을 때, 이들을 Big-theta notation으로 나타낸 결과를 표시한 것이다.

Table 1. Theoretical Sorting Time

Sorting	Best case	Worst case	Average case
Bubble sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Heap sort	n log n	n log n	n log n
Merge sort	n log n	n log n	n log n
Quick sort	n log n	n^2	n log n
Radix sort	n	n	n

3. Result

3.1 Operation Time Analysis of Data Size

입력에 사용되는 데이터의 개수에 따른 실행시간을 분석해보았다. 먼저, 배열의 크기를 변경시켜가며 실행시간을 비교해본 결과이다. 이때, 난수를 발생시킨 결과를 이용하였으며, 숫자를 난수로 뽑았기에 각 경우에 대해 5번의 반복을 한 후 평균을 기록하였다. 동일 데이터에 대해 여러 번 정렬을 시행하면 속도가 느려지는 이유가 있기에 모든 실험에서는 실행시킨 후 처음 정렬해서 나온 시간의 결과를 이용하였다. 또, 난수의 범위는 Radix Sort가 자릿수에 의해 영향을 크게 받지 않도록 -1000~1000으로 설정하였다. 이제, 정렬 알고리즘을 실험해본 결과는 아래와 같다.

Table 2. Operation Time Analysis of Data Size

Data Size	Bubble	Insertion	Heap	Merge	Quick	Radix
100	0ms	0ms	0ms	0ms	0ms	0ms
1000	2.4ms	0.4ms	0.2ms	1.2ms	1ms	0.8ms
10000	117.2ms	30.4ms	2.4ms	4.6ms	6.2ms	2.6ms
50000	3572ms	402.6ms	11.2ms	51.8ms	28.4ms	14.2ms
100000	14889.8ms	2104.4ms	24.6ms	89.6ms	35.8ms	22ms
500000	X	X	87.2ms	122ms	126.4ms	42.4ms
1000000	X	X	135.6ms	187ms	345.6ms	64.8ms

실행결과 위 표와 같은 결과를 얻을 수 있었다. 먼저, Bubble Sort와 Insertion Sort는 평균 수행시간이 O(n^2)인만큼 데이터의 개수가 증가하자 수행시간이 급증하였다. 그래서 n=500000, 1000000인 경우는 Bubble Sort와 Insertion Sort를 실행하는 시간이 너무 길어져서 시간을 측정하지 못하였다. 다른 정렬들은 n=10000까지는 상수에 의해서 시간이 변동되다가 이후에는 전체적으로, Bubble Sort < Insertion Sort < Quick Sort < Merge Sort < Heap Sort < Radix Sort의 순으로 성능이 좋은 것을 확인할 수 있었다. 이는 Table 1을 보면 이들의 평균적

인 시간 복잡도와 같은 순서임을 알 수 있다. 또, 배열의 크기가 증가할 때 시간 복잡도의 경향성 또한 따라감을 확인해볼 수 있다. Quick Sort의 경우 난수의 범위가 좁아서 n=50000부터는 중복원소가 많이 뽑히는데, 단순히 처음 원소를 Pivot으로 정해서 성능이 느려진 것이라고 추측된다. 이는 중간고사 문제와 같이 랜덤으로 partition을 나눠준다면 해결될 것이라 생각한다.

3.2 Operation Time Analysis of Sorted Data

이번에는 특수한 경우에 대한 정렬 시간의 분석을 해보았다. 특히, 경우에 따라 실행시간이 변화하는 Insertion Sort와 Quick Sort의 경우를 확인해보기 위하여 이미 정렬된 배열과, 역으로 정렬된 배열의 경우를 확인해 보았다. 이를 구하기 위해서 제출한 소스코드에서 약간의 수정을 가해 난수가 아닌 입력해주는 배열에 대해서도 시간을 구할 수 있도록 하였다. 이때, 난수를 뽑아서 Radix Sort를 통해 정렬된 배열로 만든 뒤, 이를 입력 값으로 사용하였다. 이 경우 역시 같은 조건에서 5번씩 실험한 값을 평균내서 구하였다. 다른 조건들은 3.1과 동일하다.

Table 3. Operation Time Analysis of Sorted Data

Data Size	Bubble	Insertion	Heap	Merge	Quick	Radix
10000	31ms	1ms	2.2ms	2.8ms	4.6ms	2.4ms
50000	321.2ms	4.2ms	11ms	38.6ms	43ms	12.6ms
100000	1365ms	9ms	26.8ms	78ms	51.2ms	18.6ms

Table 4. Operation Time Analysis of Reversed Sorted Data

Data Size	Bubble	Insertion	Heap	Merge	Quick	Radix
10000	42.6ms	44ms	1.6ms	2.4ms	5.2ms	4.2ms
50000	694ms	552.6ms	9.8ms	56ms	41.2ms	12ms
100000	2812.2ms	2567ms	20.2ms	74.8ms	56.8ms	22.8ms

다른 정렬들은 위의 정렬 결과와 큰 차이가 나지 않으며, Quick Sort와 Insertion Sort의 경우 큰 차이가 나타나는 것을 볼 수 있다. 또한, 이것이 Table 1에서 제시한 이론적인 결과의 경향성(O(n) or O(n^2))을 보여준다는 사실 역시도 확인해볼 수 있다.

4. Conclusion

여러 정렬 알고리즘을 직접 구현하고, 여러 번의 실험을 통해 이들의 시간 복잡도가 이론적인 값을 잘 따라가는지 확인해볼 수 있었다. Radix Sort의 경우 정수 데이터에 대해서 빠른 속도를 보이기에 알맞은 정렬이며, Quick Sort와 Insertion Sort는 사용하는 데이터의 특징에 따라 정렬 시간에 큰 차이가 나기에 주의해서 사용해야 한다는 점을 알 수 있었다. 다른 정렬 역시도 특징을 확인해볼 수 있었다. 특징을 고려하여 정렬 방법을 정한다면 유용할 것이라고 생각된다.