자료구조 Homework 4 Report

서어서문학과 2017-16849 이윤경

1. 정렬 알고리즘의 동작 방식

먼저 출력에서 overflow를 방지하기 위해 뼈대 코드의 println부분을 StringBuilder로 수정하였다. 각 정렬에서 swap의 기능이 사용되어 swap 함수를 클래스 내부에 정의하여 코드를 간결하게 하였다. 또한 명령을 실행할 때, Merge Sort와 Quick Sort의 경우 배열의 시작과 끝 index인 p, r을 매개변수로 받기 때문에 main 함수에 이를 수정하였다. 각 정렬 알고리즘은 쉽게 배우는 자료구조 교재를 참고하여 구현하였고, 일부를 수정하였다. 각 정렬의 동작 방식을 설명하면 다음과 같다.

Bubble Sort의 경우 swapped를 flag로 추가하여 이미 정렬된 배열의 경우 루프를 벗어나도록 하였다. 버블 정렬은 배열의 마지막 index부터 1까지 for loop를 돌며 index가 0인 원소부터 last까지 크기를 비교하여 가장 큰 원소를 그 루프의 마지막 원소로 지정하고, 마지막 원소는 대상에서 제외한다.

Insertion Sort의 경우 1부터 n까지 for loop를 돌며 i번째 원소를 newItem으로 지정하고, 0부터 i-1번째까지의 원소를 newItem과 비교하여 크기 순으로 정렬하였을 때 newItem의 위치에 그 원소를 삽입한다. i가 1부터 n까지 증가하면서 루프를 돌 때, 그 루프에서 i-1번째까지의 원소는 이미 정렬된 상태이다.

Heap Sort의 경우 정렬을 돕기 위해 percolateDown, buildHeap, deleteMax 함수를 정의하였다. percolateDown 함수는 i번째 원소의 두 child 중 더 큰 원소를 child로 지정하여 i번째 원소가 child보다 작을 경우 두 원소를 swap하고, child를 기준으로 재귀적으로 percolateDown 함수를 실행한다. buildHeap은 배열이 Heap 성질을 충족하게 만드는 함수로, child를 가지는 최초의 internal node부터 root node까지 percolateDown을 수행한다. deleteMax 함수는 가장 큰 원소를 제거하는 과정이다. Heap에서 가장 큰 값은 root값이므로 이 값을 max로 지정한 후, 배열의 마지막 값을 root 노드로 설정한다. 이후 max 값을 정렬 대상에서 제외하기 위하여 numItems를 1 감소시킨 후, percolateDown을 수행하여 정렬한다.

Heap sort는 위 함수들을 이용하여 정렬을 수행한다. 먼저 buildHeap을 통해 주어진 배열이 Heap의 성질을 충족하도록 만든다. 이후 결과값을 담을 result[]를 하나 생성하고, 기존 value[]의 길이에서 1까지 감소시키며 max 값을 배열에서 하나씩 제외하며 result 배열에 max 값을 추가한다. 전체 배열의 크기를 하나씩 감소하는 것 대신, max 값을 대상에서 하나씩 제외하기 위해 numItems 변수를 i+1로 설정하여 배열의 길이를 하나씩 감소하도록 하였다. result[]은 value[]와 동일한 크기로, max 값을 삭제하기 전 배열의 마지막 index가 result에서 max값의 index가 된다. 루프가 index가 1일 때 종료되면 value배열에는 가장 작은 값이 남아있으므로 이 값을 result 배열에 추가하여 result 배열을 return한다.

Merge Sort는 tmp[]을 매번 새로 생성하여 공간이 낭비되는 것을 방지하기 위해 value[]과 동일한 크기의 B[]을 만들어 이용하였다. merge Sort는 배열의 가운데 index를 기준으로 배열을 분할하여 재귀적으로 정렬을 진행한 후 두 배열을 합치는 merge 과정으로 이루어져 있다. 배열을 합치는 merge 과정은 두 배열의 index를 증가시키며 값을 비교하여 B[]에 작은 값부터 담아 이를 다시 value[]로

복사하는 과정이다.

Quick Sort는 배열의 마지막 값을 기준으로 partition을 진행한 후 분할된 두 개의 배열에서 재귀적으로 같은 과정을 반복하여 정렬하는 알고리즘이다. Quick Sort는 동일한 원소가 존재할 경우 효율성이 악화되기 때문에 이를 개선하기 위하여 random 함수를 불러와 이 값이 0.5보다 작을 경우에만 partition의 기준 값(x)보다 작은 쪽의 배열에 삽입하는 코드를 추가하였다. partition의 알고리즘은 배열의 마지막 값(x)을 기준으로 배열의 초기값(p)부터 r-1까지 루프를 돌며 x보다 작거나 같은 값은 왼쪽에, 큰 값은 오른쪽에 정렬한 후, 기준 값인 x를 x보다 작거나 같은 값의 오른쪽에 삽입하는 과정이다.

Radix Sort는 정수의 자릿수가 제한되어 있을 때 효율적으로 사용할 수 있는 정렬로, 책의 코드에 음의 정수가 포함되었을 때도 수행할 수 있도록 일부를 수정하였다. 먼저 정수의 부호를 판단하기 위해 배열의 길이만큼 루프를 돌며 음수의 개수를 세어, 음수와 양수의 개수만큼 positive와 negative배열을 새로 생성하였다. 그리고 다시 루프를 돌며 각 배열에 음수와 정수를 담았다. 이 과정에서 음수는 계산을 수월하게 하기 위하여 양수로 바꾸어서 저장하였다. 각 배열은 rSort함수를 통해 Radix Sort를 수행하고 negative 배열에서 가장 큰 값(음수로 바꾸었을 때 가장 작은 값)부터 순서대로 음수로 바꾸어 value 배열에 저장하고, negative 배열이 끝나면 positive 배열에서 가장 작은 값부터 value 배열에 저장하였다.

rSort 함수는 counting sort를 활용하였다. 0부터 9까지의 개수를 셀 cnt[]과 이를 누적하여 담을 start[], 정렬된 값을 담을 B[]을 생성하였다. 먼저 배열의 길이만큼 루프를 돌며 가장 큰 값을 찾아 max로 지정하였다. 그리고 이 max값의 자릿수를 찾기 위하여 Math의 log10함수를 이용하였다. 이제 1의 자리부터 최대 자릿수까지 돌며 정렬을 진행한다. 먼저 cnt배열을 0으로 초기화한 후 각 자릿수의 값(code: value[i]/Math.pow(10, digit-1))%10)에 따라 해당하는 index의 값을 1씩 증가한다. 그리고 start[0] 값은 0으로 지정하고, start 배열은 start[d]에 start[d-1]+cnt[d-1]을 담아 0부터 누적 개수를 저장하며 동일 원소의 개수 처리를 하였다. 다시 value[]의 길이(n)만큼 돌며 그 원소의 해당 자릿수의 값의 start 값이 정렬된 배열에서 그 값의 위치를 의미하기 때문에 B[]에 해당 값을 index로 하여 값을 담고, 동일 원소 처리를 위해 start[]의 해당 값을 1 증가해주었다. 모든 자릿수에서 정렬이 끝나면 value[]에 B[] 원소를 담아 return한다.

2. 동작 시간 분석

동작 시간 분석을 위해, 먼저 시간 복잡도가 $O(n^2)$인 BubbleSort와 InsertionSort를 비교하고, 시간복잡도가 O(nlogn)인 나머지 정렬과 Radix Sort을 비교하였다. $O(n^2)$의 경우 n이 커질수록 속도가 느려져 n이 200,000까지의 시간을 분석하였고, 나머지의 경우 n이 2,000,000까지의 시간을 분석하였다. 각 데이터 크기마다 20번씩 test를 진행하여 최소값, 최대값, 평균과 표준편차를 구하였고 엑셀을 통해 평균을 그래프로 나타내어 추세선과 수식, R^2을 구하였다.

(1) Bubble Sort, Insertion Sort

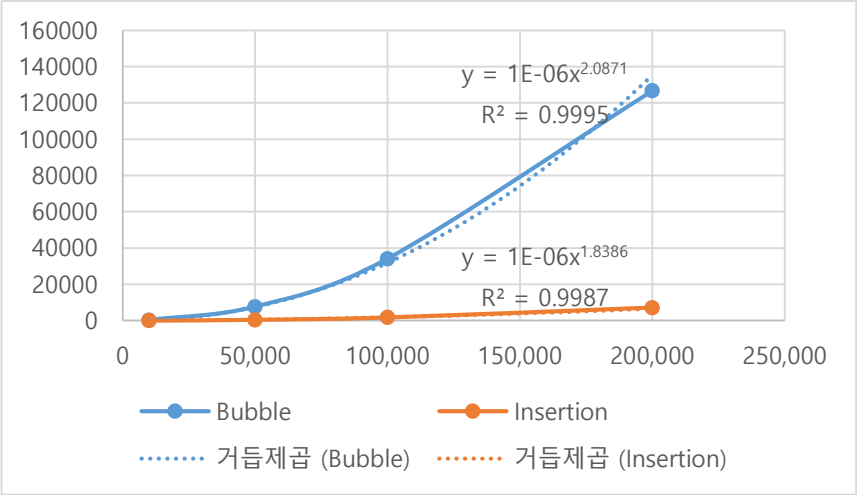
BubbleSort와 InsertionSort는 이중 for 루프 구조로 이루어져 있어 시간 복잡도가 $O(n^2)$이다

<Table 1> Bubble Sort

n	10,000	50,000	100,000	200,000
최소값	218	6447	31907	122903
최대값	313	9043	37431	132144
평균	**253.6**	**7683.1**	**34111.33**	**126692.7**
표준편차	47.5	709.205	2389.089	3951.36

<Table 2> Insertion Sort

10,000	50,000	100,000	200,000
15	375	1532	6481
47	562	2085	7910
28.222	**467.2**	**1762.1**	**7179.89**
9.235	68.401	176.717	428.85



이를 그래프로 그렸을 때 두 알고리즘 모두 $O(n^2)$에 가까운 것으로 나타났다. 이는 R^2 값이 1 에 가깝고, 추세선 수식의 지수가 2 에 가까운 것을 통해 확인할 수 있다. 다만 Bubble Sort 는 Insertion Sort 보다 swap 이 많이 이루어져 소요 시간이 더 큰 것으로 나타났다.

(2) Heap, Merge, Quick, Radix Sort

Heap, Merge, Quick sort 는 모두 O(nlogn)의 시간 복잡도를 갖는다. Heap 을 만드는 과정이 O(n), 이를 정렬하는 과정의 시간 복잡도가 O(nlogn)이 된다. Merge 와 Quick Sort 모두 특정 원소를 기준으로 삼아 양쪽으로 분할하여 정렬하는 과정을 재귀적으로 수행하여 시간 복잡도는 O(nlogn)이다. Radix Sort 는 최대 자릿수만큼 for loop 를 돌고 그 내부에서 value[]의 길이만큼 for loop 를 도는 과정과 cnt[]의 길이만큼 for loop 를 도는 과정이 포함되어 있어 O(d(n+c))이다. 이때 cnt[]는 0~9 까지 10 개이므로 상수 시간이기 때문에 O(dn)으로 표현 가능하다. 마찬가지로 이를 표와 그래프로 나타낸 것은 다음과 같다. 표에서 n 의 단위는 10,000 이다.

<Table 3> Heap Sort

n	5	10	50	100	200
최소값	19	21	180	365	885
최대값	46	35	431	1104	1659

<Table 4> Merge Sort

5	10	50	100	200
22	34	151	264	547
151	148	326	622	1265

평균	**31.761**	**135**	**267.381**	**528.761**	**1225.524**
표준편차	8.579	68.571	59.228	187.631	238.623

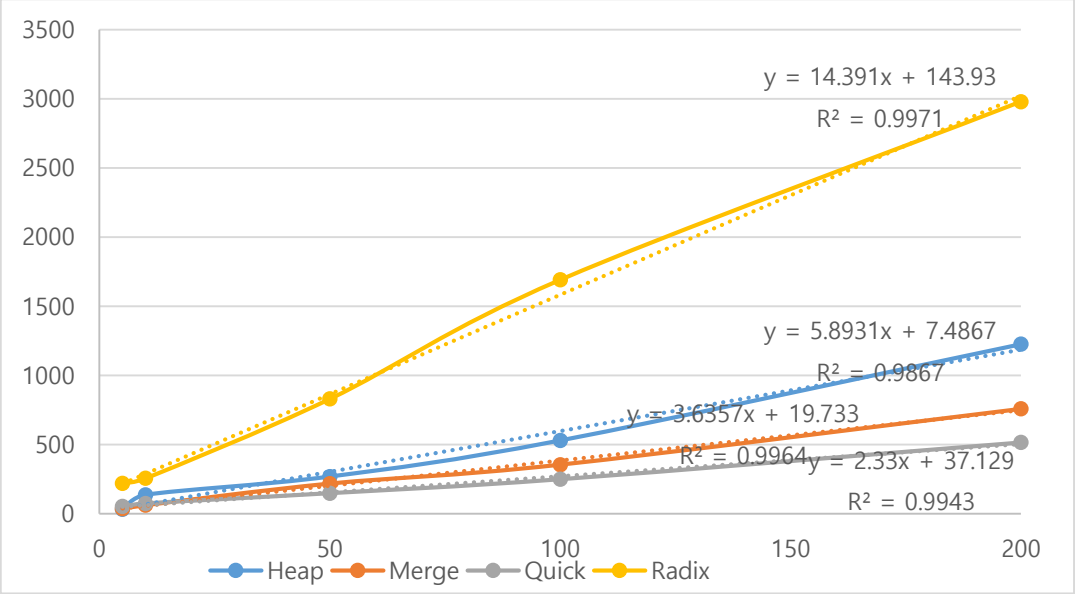
<Table 5> Quick Sort

35.5	**61.318**	**217.409**	**354.227**	**757.227**
26.133	30.292	56.882	94.020	202.717

<Table 6> Radix Sort

n	5	10	50	100	200
최소값	25	47	94	182	350
최대값	112	146	228	620	863
평균	**51.181**	**73.909**	**147.727**	**248.681**	**514.590**
표준편차	22.476	24.712	33.075	105.063	125.313

5	10	50	100	200
124	147	666	1209	2532
338	375	1103	2236	3379
218.809	**255.190**	**829.571**	**1690.286**	**2978.619**
50.396	65.185	124.148	328.701	219.144



네 개의 알고리즘을 비교하였을 때, O(n)인 Radix Sort가 가장 크고, O(nlogn) 정렬의 경우 Heap Sort가 가장 느리고, Quick Sort가 가장 빨랐다. 이론상 n이 충분히 클 때 O(n)인 Radix Sort가 가장 빨라야 하지만, O(n)에 포함된 상수 overhead가 크기 때문에 O(nlogn)의 정렬보다 오래 걸리는 것으로 추정할 수 있다. 또한 네 개의 알고리즘 모두 선형의 추세선에서 R^2값이 1에 가까워 O(n)과 O(nlogn)을 충족한다. 특히 O(nlogn)의 정렬에서 선형처럼 보이는 이유는 log n 값이 충분히 크지 않아 상수의 기울기를 갖는 것과 큰 차이가 없기 때문이다. 미세하지만 n이 커질수록 추세선의 기울기보다 데이터 값의 기울기가 더 커지고 있음을 고려할 때, 이는 기울기가 단순 상수 값이 아니라 log n에 가까운 값이라는 것을 추정할 수 있을 것이다.

3. 결론

직접 정렬 알고리즘을 구현하고, 그에 따른 실행 시간을 측정하며 정렬의 시간 복잡도를 실험해 보았다. $O(n^2)$의 정렬이 다른 정렬에 비해 많은 시간이 소요되었고, 그 중에서도 Bubble Sort가 가장 오래 걸렸다. Radix Sort는 O(n)의 시간을 보장하지만, 실험에 사용된 데이터의 수가 overhead에 비해 크지 않았기 때문에 O(nlogn)의 정렬 알고리즘보다 많은 시간이 소요되었다. O(nlogn)의 알고리즘의 경우 Quick Sort가 가장 빠르고, Merge Sort, Heap Sort 순이었다.