Sorting Report

2015-11323 박성민

1. Implementation

1.1. Bubble Sort

Bubble sort 는 0 to i 의 인덱스를 가지는 j 에 대해서 j 번째 값이 j+1 번째 값보다 큰 경우 swap 해주는 방식을 i 가 (배열 전체 크기 – 1) to 1 까지 반복해 i 가 1 줄어들 때마다 배열의 오른쪽에서부터 정렬되는 방식이다.

비교횟수는 case 에 상관없이 $\sum_1^{n-1} i = \frac{n*(n-1)}{2} \cong O(n^2)$ 이고, assign 횟수는 best case 에서 0, worst case 에서 $3 * \frac{n*(n-1)}{2} \cong O(n^2)$이다. average case 는 마찬가지로 $O(n^2)$이다.

1.2. Insertion Sort

Insertion sort 는 왼쪽부터 정렬된 배열을 늘려나가는 방식으로, i 가 1 to (배열 전체 크기)일 때 정렬된 배열의 가장 오른쪽 값과 그 다음 값을 비교해 클 경우 한 칸씩 옆으로 이동시키는 방식을 통해 구현하였다. 이때 작거나 같은 경우 while 문을 나가게 해 stable sorting 성질을 만족하도록 구현하였다.

비교횟수는 best case 에서 n-1, worst case 에서 bubble sort 와 같은 $\frac{n*(n-1)}{2} \cong O(n^2)$이다. assign 횟수는 best case 에서 2*(n-1)번, worst case 에서 $2 * (\mathrm{n} - 1) + \frac{n*(n-1)}{2} \cong O(n^2)$이다.

1.3. Heap Sort

Heap sort 는 먼저 building heap 과정을 통해 child 의 값보다 parent 의 값이 큰 binary tree 형태로 만든 뒤, 가장 큰 값을 순차적으로 맨 뒤로 빼주는 방식을 통해 정렬되는 방식이다. Percolate down 을 통해 두 child 중 큰 값을 찾고, parent 값과 비교하여 parent 가 작은 경우 swap 한 이후 재귀적으로 반복한다.

비교횟수는 n 이 바뀌면 best case 나 worst case 도 바뀌게 되어 average case 에서 $O(\mathrm{nlog}\, n)$이고, assign 횟수도 마찬가지로 $O(\mathrm{nlog}\, n)$을 만족한다.

1.4. Merge Sort

Merge sort 는 절반으로 나눈 뒤 각각에 대해서 재귀적으로 Merge sort 를 하고, Domerge 메서드를 통해 나눠진 두 배열을 정렬된 순서로 합친다.

Merge sort 는 배열이 binary tree 형태처럼 나뉘는데 높이가 nlogn 으로 나뉘고, 어떤 형태의 배열인 경우에도 assign 과 비교 횟수가 큰 차이가 없어 $O(\mathrm{nlog}\, n)$을 만족하므로 시간복잡도에서 굉장히 좋은 효율을 보이지만, in-place sorting 이 불가능해 DoMerge 내에서 배열의 크기만큼을 추가로 선언해주기 때문에 메모리 오버헤드가 생기는 단점이 있다.

1.5. Quick Sort

Quick sort 는 pivot 을 기준으로 큰 값과 작은 값을 나눈다. 분류가 끝나면 finger1 이 pivot 값을 가리키고, finger1 을 기준으로 그 앞부분의 배열과 그 뒷부분의 배열을 다시 재귀적으로 Quick sort 를 실행한다.

Quick sort 는 평균적으로 Merge sort 와 같은 $O(\mathrm{nlog}\, n)$의 성능을 보장하면서 in-place sorting 이 가능하다는 장점이 있지만, pivot 이 정렬을 효과적으로 하지 못하는 worst case 의 경우 $O(n^2)$의 성능을 나타내며, Java 의 경우 Call by Value 기 때문에 worst case 에서 recursive 하게 메서드를 call 하는 경우 10 만개의 input 에 대해서도 JVM 의 default Heap 이 터져 OutOfMemoryError 가

생기는 경우가 발생한다. 따라서 Java 에서 재귀적으로 구현하는데 있어서 Quick sort 는 상황을 잘 판단해야 한다. 이는 Test 에서 확인할 수 있다.

1.6. Radix Sort

Radix sort 는 전체 값의 비교가 아닌 각 자릿수의 비교를 통해 정렬하는데, int 값이 input 으로 들어오고, 음수와 양수 또한 비교해야 하므로 32bit 에 대해서 비교하는 방식으로 구현하였다. LSB 부터 0 인경우 앞으로 가져오고 1 인경우 뒤로 빼는 방식으로 31 번째 bit 까지 비교한 뒤 sign bit 에 대해서는 거꾸로 1 인경우 앞으로 가져오고 0 인경우 뒤로 빼는 방식으로 구현하였다.

32 번 반복 * 각각 n 회 비교, n*32 회 assign 이 일어나므로 $O(n)$을 만족하여 구현한 정렬 알고리즘 중 average case 에서 가장 좋은 성능을 보인다. 하지만 in-place sorting 이 아니라는 단점이 있다. 0 과 1 로 표현되므로 크기가 n 인 배열 두 개를 선언하여 3n+α 의 공간이 필요하다.

1.7. Table

Algorithm	In-Place	Stable	Time Complexity	Space Complexity
Bubble	O	O	$O(n^2)$	$O(n)$
Insertion	O	O	$O(n)$ / $O(n^2)$	$O(n)$
Heap	X(알고리즘상 O)	X	$O(n\log n)$	$O(n)$
Merge	X	O	$O(n\log n)$	$O(n)$
Quick	X(알고리즘상 O)	X	$O(n\log n)$ / $O(n^2)$	$O(n)$ / $O(n^2)$
Radix	X	O	$O(n)$	$O(n)$

In-place 의 경우 원래는 Heap, Quick 도 O 여야 하지만 Java 에서 Recursive 하게 구현할 경우 call by value 로 인해 새 메모리 공간이 필요하게 되어 X 이다. In-place 는 Space Complexity 와도 연관되는데, In-place 가 가능한 정렬은 $O(n)$의 space complexity 를 가지고, Heap, Merge, Radix 의 경우 배열의 크기만큼의 공간의 상수 배 만큼이 추가적으로 필요하다. Quick 정렬의 average case 도 마찬가지인데, worst case 에서는 함수가 recursive 하게 n-1 번 불릴 수 있고, 각각 불릴 때마다 n-1 to 1 만큼의 메모리가 call 되기 때문에 $O(n^2)$의 공간복잡도를 갖게 될 수 있다. 이는 Test 에서 확인할 수 있다.

Stable 은 Heap 과 Quick 을 제외하고는 정렬 과정에서 순서가 바뀔 일이 없지만 Heap 은 percolate down 과정에서 기존 index 순서가 아닌 binary tree 에서 높이에 따라 바뀌므로 Stable 이 깨질 수 있고, Quick 은 finger1 의 값과 그 다음 값이 같은 값인 경우 finger2 에서 pivot 값보다 작은 값이 나오게 되면 finger1 과 finger2 를 swap 하는 과정에서 기존 Stable 이 깨지게 된다.

Time complexity 는 특수한 성질(모두 정수 int)을 만족해 Radix 가 평균적으로 가장 좋은 성능을 낼 것으로 기대되며, Heap 과 Merge 가 그 다음, Quick 은 pivot 이 잘 선택될 경우 Heap 과 Merge 에 비등비등한 성능을 낼 것으로 예상된다. Bubble sort 와 Insertion sort 는 n 이 커질수록 다른 정렬 알고리즘의 정렬 속도와 차이가 큰 폭으로 커질 것으로 예상된다.

2. Test

난수가 아닌 경우도 수행시간을 출력하게 코드를 고쳐 Test 를 진행했으며, 총 6 가지 case 에 대하여 같은 정렬 방식 네 번을 실행하고 종료하는 방식을 10 회 반복했다. Table 8 의 경우 같은 정렬 방식 네 번을 실행하고 종료하는 방식을 3 회 반복했다. $O(n\log n)$ 이하의 수행시간을 갖는

시행들의 첫 번째 정렬 시간이 이후 세 개의 정렬 수행 시간과 차이가 커 모든 data 에 대해 2, 3, 4 번째 수행에 대하여 최솟값, 최댓값, 평균, 표준편차를 구하였다. 모든 단위는 ms 이다.

r 0.01m	max	min	average	stdev	r 0.1m	max	min	average	stdev
Bubble	132	109	117.10	5.62	Bubble	28270	13803	16805.27	4004.61
Insertion	16	8	10.57	2.97	Insertion	1100	867	894.20	43.25
Heap	4	1	1.57	0.68	Heap	15	10	11.17	1.42
Merge	3	1	1.93	0.37	Merge	19	14	16.30	1.06
Quick	3	1	2.07	0.69	Quick	36	12	15.67	5.35
Radix	3	1	1.87	0.43	Radix	14	10	10.37	1.03

Table 1 Random 0.01m, 0.1m input

sorted 0.1m	max	min	average	stdev	r-sorted 0.1m	max	min	average	stdev
Bubble	3289	1197	2549.13	441.66	Bubble	6664	3437	5010.33	817.64
Insertion	14	0	1.57	3.42	Insertion	2497	1985	2082.97	103.19
Heap	34	11	18.60	7.96	Heap	19	7	10.67	2.97
Merge	29	14	17.30	3.31	Merge	16	10	12.30	2.00
Quick	39	7	11.67	6.78	Quick	16	5	8.47	3.05
Radix	11	3	5.73	2.24	Radix	10	3	5.07	1.80

Table 2 Sorted 0.1m, Reverse-sorted 0.1m input

same 0.1m	max	min	average	stdev	same 0.01m	max	min	average	stdev
Bubble	3221	2118	2750.33	297.62	Bubble	96	20	46.33	20.55
Insertion	13	0	1.57	3.14	Insertion	12	0	1.17	2.61
Heap	30	0	6.00	7.16	Heap	17	0	2.47	3.79
Merge	34	14	17.10	3.66	Merge	61	1	9.87	13.15
Quick	OOME	OOME	OOME	OOME	Quick	638	177	331.13	117.61
Radix	11	3	6.03	2.53	Radix	12	1	6.27	4.39

Table 3 Same 0.1m, 0.01m input

Random 1m	max	min	average	stdev	0.1m summary	max	min	average	stdev
Bubble	2629732	1553040	1949869.89	320531.09	Bubble	28270	1197	8121.58	6682.42
Insertion	125391	122248	123320.00	1061.11	Insertion	2497	0	992.91	859.74
Heap	197	135	156.78	21.90	Heap	34	7	13.48	6.12
Merge	210	139	151.56	22.55	Merge	29	10	15.30	3.16
Quick	193	121	143.22	23.03	Quick	39	5	11.93	6.01
Radix	158	119	143.44	17.65	Radix	14	3	7.06	2.94

Table 4 Random 1m input, 0.1m summary

Table 1 은 r 10000 -300000 300000, r 100000 -300000 300000 명령어를 통해 난수를 생성한 후 각 정렬알고리즘을 시행한 결과이다. 비교적 적은 만 개의 data 에 대해서 Bubble sort 도 0.1 초 정도로 빠르다고 느낄 수 있지만 다른 정렬 알고리즘에 비해 굉장히 느리다. Insertion sort 는 같은 $O(n^2)$의 시간복잡도를 갖는 Bubble sort 에 비해 10 배정도 빠른 정렬속도를 보이는데 이는 Insertion sort 의 worst case 가 Bubble sort 와 같은 수준의 비교 횟수를 가지고, bubble sort 내 assign 횟수보다 Insertion sort 에서의 assign 횟수가 더 적기 때문에 average case 인 random input 에 대해서 10 배 정도의 차이가 난다. 이외에 $O(n\log n)$ 이하의 시간복잡도를 갖는 나머지 정렬 알고리즘의 경우 평균적으로 2ms 정도 시간이 걸렸다.

10 만 개의 결과를 만 개의 결과와 비교해보면, input 의 개수가 만 개에서 십만 개로 열 배 늘었을 때 Bubble sort 는 약 150 배, Insertion sort 는 약 90 배 정도 수행시간이 늘어난 반면 나머지 정렬들의 경우 약 8 배정도 늘어났다. 10 만개의 input 에 대하여 Bubble sort 로 정렬할 경우 한번 정렬하는데 약 16 초가 걸리므로 굉장히 비효율적이다.

Table 2 의 결과는 정렬된 10 만개의 input 과 내림차순으로 정렬된 10 만개의 input 에 대하여 정렬 알고리즘을 수행한 결과이다. 정렬된 배열에 대하여 가장 좋은 효율을 나타내는 알고리즘은 Insertion Sort 로 알려져 있는데, 이는 이미 정렬된 배열의 가장 오른쪽과 그 다음 값을 비교하는 방식으로 정렬된 배열의 크기를 늘려가는 방식이기 때문에 이미 정렬이 되어있다면 n-1 번의 비교만으로 끝나게 된다. 따라서 평균 1.57ms 의 굉장히 빠른 정렬속도를 보여준다. 내림차순으로 정렬된 10 만 개의 결과는 Insertion sort 를 사용하지 않는 이유를 보여주는데, 만약 내림차순으로 정렬되어 있는 경우 Insertion sort 는 모든 값에 대해 비교 및 밀어내기가 일어나기 때문에 $O(n^2)$의 시간복잡도를 갖게 되어 10 만개의 input 에 대하여 평균 2 초의 시간이 걸린다.

Table 3 의 결과는 Quick sort 의 단점을 보여준다. 10 만개의 input 이 모두 숫자 2 인 경우와 만개의 input 이 모두 숫자 2 인 경우의 테스트 결과인데, Quick sort 의 알고리즘에서 pivot 보다 작은 것은 finger1 을 증가시키지 않고, 크거나 같은 것은 finger1 을 증가시키기 때문에 partition 이 극단적으로 나뉘게 되고(0:n-1) 이는 $O(n^2)$ 번의 method call 이 생겨 JAVA 에서는 $O(n^2)$의 공간복잡도를 갖게 된다. 10 만개의 input 의 경우 JVM 의 기본 Heap Size 를 초과하여 OutOfMemoryError 가 떴고, 이는 설정을 변경해 Heap size 를 늘리더라도 input 의 크기가 커진다면 언제든 다시 발생할 수 있음을 나타낸다. Insertion sort 는 Stable 한 속성 때문에 Table 3 의 결과에서도 가장 빠른 정렬 속도를 나타낸다. 입력의 크기를 만 개로 줄여 Quick sort 의 worst case 를 다른 정렬 알고리즘과 비교해 볼 때, 재귀적 호출 및 극단적인 partition 등이 겹쳐 bubble sort 보다 느린 정렬속도를 보였다. 따라서 Quicksort 를 java 에서 사용하려면 굉장히 많은 중복이 가능한 input 의 경우 적절하지 않으며 그렇지 않은 경우 private 로 멤버 변수를 선언한 뒤 이를 멤버 메서드에서 구간을 받아 this.array 를 직접 바꿔주는 방식으로 해야 할 것이다.

Table 4 의 좌측은 백만 개의 random input 을 정렬하였는데, Table 1 에서 input 이 10 배 늘어나는 것과 비슷하게 수행시간이 늘어났다. Bubble sort 와 Insertion sort 의 경우 약 100 배 가량 증가하여 평균 32 분, 20 분이 걸렸으며, 이외의 경우 0.1-2 초로 100 만개의 input 에서 확연하게 그 차이를 느낄 수 있었다.

Table 4 의 우측은 0.1m 의 input 인 Table 1 의 우측, Table 2 를 종합한 결과이다. $O(n)$의 시간복잡도를 가진 Radix sort 가 가장 평균이 낮고, 가장 표준편차가 낮은데 이는 input 에 관계없이 같은 방식으로 수행하기 때문이다. Merge sort 또한 input 의 크기가 같으면 binary tree 처럼 생각하였을 때 tree 의 높이가 같기 때문에 낮은 표준편차를 가진다. Heap 과 Quick 은 비슷한 결과를 보여 메모리 오버로드나 과도한 중복에 대한 문제가 해결된다면 무엇을 사용해도 상관없고 만약 해결되지 않는다면 Heap sort 를 사용하는 편이 더 좋다. 이미 정렬된 알고리즘 또는 처음부터 input 을 하나씩 받으면서 정렬한다면 Insertion sort 를 사용해도 빠른 실행시간을 확보할 수 있다.

3. Test case 및 code

https://github.com/ulgal/DS-sorting