과제4 보고서 2019-14184 이건하

1. 정렬 알고리즘의 원리 분석

1)Bubble sort: 버블 정렬은 이웃한 두 수를 비교하여 오른쪽이 크면 swap하는 방식으로 제일 큰 수를 오른쪽으로 보내는 정렬방식이다. 이러한 방식을 0부터 ith 원소까지 이런 식으로 swap시켜 ith 원소에 가장 큰 원소가 위치하게 하고, i를 n-1부터 1씩 감소시켜 큰 수가 뒤에서부터 차례대로 위치시켜 배열을 정렬한다.

2)Insertion sort: 삽입정렬은 i-1th 원소까지의 배열을 정렬하고 ith 원소의 위치를 비교하여 i-1th 배열에 삽입하는 정렬이다. 그렇게 원소를 삽입하면 정렬된 i크기의 배열이 되고, i를 1씩 증가시켜 마지막 원소까지 삽입하면 전체 배열이 정렬이 된다.

3)Heap sort: 힙 정렬은 가장 큰 원소가 root에 들어가는 heap의 성질을 이용하여 root와 ith 원소를 swap하고, i-1th 원소까지를 heap 성질을 만족시키도록 buildheap 알고리즘을 통해 만들어준다. 이때 i를 n부터 1씩 감소시킴으로써 전체배열을 정렬한다.

4)Merge sort: 병합정렬은 배열을 두 개로의 나누어서 재귀적으로 정렬하는 분할정복 방법이다. 재귀적으로 배열을 쪼갬으로써 1개만 남을 때까지 쪼갤 수 있는데 1개에서부터 배열들을 순서대로 merge함으로써 전체 배열을 정렬할 수 있다.

5)Quick sort: 퀵정렬은 pivot 원소(과제에 사용한 알고리즘에서는 맨 마지막원소)을 기준으로작은 원소는 앞으로 큰 원소는 뒤로 보내 partition을 나눠준다. 나눈 partition들에 대해서 재귀적으로 퀵정렬을 수행하여 전체 배열에 대하여 정렬을 수행한다.

6)Radix sort: 기수정렬은 배열을 원소들의 자리수가 상수 개일 때 적용할 수 있는 정렬 알고리즘이다. 기수정렬은 낮은 자리수부터 [0-9]까지의 배열에 각각의 자릿수에 해당하는 숫자의 개수를 저장하고, 누적합을 구한다. 그 누적합을 사용하여 자릿수별로 숫자를 정렬한다. 과제에 사용한 알고리즘은 음수까지 정렬해야 했으므로 길이가 10인 minus와 plus 배열을 따로 놓아서 자릿수에 해당하는 숫자를 counting하고, minus배열은 역순으로 plus배열을 순방향으로 누적합을 구하여 자릿수를 배열하였다.

2. 동작 시간 분석

각 알고리즘의 동작시간 분석은 Insertion sort와 Bubble sort는 data개수가 5000, 10000, 50000개일 때 비교를 진행했고, 나머지 정렬들은 100000,500000,1000000에 대해서도 추가로 비교를 진행하였다. 실험방법으로는 (-n,n)범위에서 random하게 생성된 n개의 배열을 정렬하는 방식을 택했다. 모든 case에 대하여 100번 반복하여 최대, 최소, 평균, 표준편차를 구하는 방법으로 진행하였고, 단위는 ms이다.

1) Bubble Sort

Bubble	5000	10000	50000
Min	23	105	3225
Max	28	133	7584
Avg	26.5	124	3794.75
Sd	1.55	4.07	570.98

버블 정렬의 경우 다른 정렬과 비교하여 가장 많은 시간이 소요되었다. data의 개수가 5,000개일 때만 보면 다른 정렬이 1-3초 걸릴 때 Bubble 정렬이 26.5초 20배정도 길게 소요되었다. 또한 3개의 case밖에 실험하지 못했지만 average case만 보았을 때 실행횟수가 2배 증가했을 때는 실행

시간이 4배, 5배 증가했을 때는 약 30배 증가함으로 Bubble sort의 시간 복잡도가 O(n^2)임을 알 수 있었다.

2)Insertion sort

Insertion	5000	10000	50000
Min	1	6	189
Max	9	16	250
avg	2.48	9.23	237.89
sd	0.905	1.54	8.08

Insertion sort는 Bubble sort와 같은 O(n^2)임에도 실행시간이 현격하게 줄어들었다. 항상 실행시간이 n^2에 비례하는 Bubble sort와는 달리 정렬된 배열에 대해서는 n에 비례하여 실행시간이 들기 때문이라고 생각할 수 있다. 그러나 실행시간이 5배 증가했을 때는 25.7배 실행시간이 2배 증가했을 때는 3.7배 증가한 사실로 미루어보아 Insertion sort의 실행시간이 O(n^2)임을 알 수 있었다.

3) Heap sort

Heap	5000	10000	50000	100000	500000	1000000	5000000
Min	0	0	4	10	63	138	985
Max	2	2	9	15	101	166	1097
avg	0.47	0.86	4.93	10.78	68.24	144.55	1030.56
sd	0.14	0.173	0.48	0.51	6.47	4.36	15.01

Heap sort는 앞서 실험한 두 정렬방법에 비해 훨씬 좋은 정렬 성능을 보여주었다. 시간복잡도를 대략적으로 계산하면, data의 수가 2배 증가하였을 때, 실행 시간이 2배가 약간 넘은 것으로 보아 O(nlogn)임을 알 수 있었다.

4) Merge sort

Merge	5000	10000	50000	100000	500000	1000000	5000000
Min	0	0	3	9	51	106	609
Max	1	2	10	15	64	119	668
avg	0.45	0.8	4.47	9.84	53.42	109.55	626.31
sd	0.01	0.17	0.65	0.67	1.56	1.81	8.61

Merge sort는 Heap sort보다 많은 데이터개수를 처리하는 배열에서 실행시간이 훨씬 빨랐다. 반면에 데이터 개수가 적을 때는 실행시간이 비슷했는데 이는 임시배열로 되써주는 과정에서 시간이 걸리기 때문인 것으로 생각되며 정렬알고리즘 자체의 성능은 merge sort가 Heap sort 보다는 더 좋다는 사실을 알 수 있었다. 또한 다른 정렬들에 비해 표준편차가 작아 어느 상황에서나 일정한 성능을 유지하는 Merge sort의 특징 또한 알 수 있었다. 시간복잡도도 마찬가지로 데이터개수가 2배 늘어날 때 평균 실행시간이 2배를 조금 넘은 것으로 보아 O(nlogn)에 비례한다는 사실을 알 수 있었다.

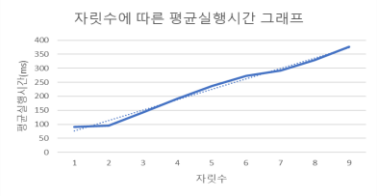
5) Quick sort

Quick	5000	10000	50000	100000	500000	1000000	5000000
Min	0	0	3	6	38	78	462
Max	1	2	9	15	55	86	496
avg	0.29	0.55	3.48	6.94	40.47	81.84	477.07
sd	0.01	0.14	0.57	0.84	1.77	1.46	6.36

Quick sort는 다른 정렬들보다 정렬시간이 훨씬 빨랐다. 시간복잡도도 마찬가지로 데이터개수가 2배 늘어날 때 평균 실행시간이 2배를 조금 넘은 것으로 보아 O(nlogn)에 비례한다는 사실을 알 수 있었다.

6) Radix sort

Radix	5000	10000	50000	100000	500000	1000000	5000000
Min	0	0	8	14	108	227	1381
Max	8	4	16	44	162	329	1857
avg	0.8	1.73	9.41	21.69	118.23	272.16	1434.69
sd	2.19	0.42	0.74	4.07	6.51	25.01	58.24



Radix sort는 Quick sort나 Merge sort에 비해 느렸는데 이는 radix sort의 구현과정에서 계수정렬의 방식으로 임시배열에 써주고 그 임시배열을 다시 원래배열에 배열하는 방식으로 배열을 했기 때문에 연산이 증가하기 때문이라고 해석할 수 있다. 실제로 100000개의 배열을 (-n,n)범위에서 생성하는 것이 아니라 자릿수가 1인 (-9,9)범위에서 생성했을 때는 Radix sort가 약 평균 4ms로 그 다음으로 빠른 Merge sort의 6.41보다 빨랐다. 시간 복잡도는 정확히 n에 비례하지는 않았는데 이는 우리가 n을 증가시킬 때 배열 원소의 범위를 마찬가지로 늘려주었기 때문에 자릿수 k도 증가하기 때문인 것으로 생각된다. 그래프는 자릿수에 따른 Radix sort의 평균 실행시간의 그래프이다. 점선으로 추세선을 그려주었는데 추세선과 거의 일치하는 것을 보아 실행시간이 자릿수에 비례한다는 사실을 알 수 있었다.

3. Search의 동작 방식

Search의 구현방식은 총 3개의 step으로 이루어진다. 첫번째는 입력된 배열의 원소의 범위를 확인하고, 범위의 자릿수에 따라 배열을 분류한다. 이 경우에 배열의 자릿수를 k라고 한다면(단, 배열의 자릿수는 음수를 포함 ex) -9~9이면 자릿수는 2이다.) k가 3보다 작으면 Radix sort를 사용하는 것이 가장 빠르고, 그렇지 않은 경우에는 다른 정렬을 사용하는 것이 더 빨랐다. 두번째는 hash table을 이용해서 충돌횟수를 중복빈도로 근사하는 방식이다. 구현방식은 배열의 범위만큼의 table 배열을 생성하고, table의 자리 값과 일치하는 배열의 원소를 그 자리에 삽입한다. 만약 이미 자리에 있는 원소가 있는 경우에는 충돌횟수를 근사시킨다. 이 과정을 모든 배열에 반복하면 충돌횟수가 구해지고 이는 중복된 원소의 총 개수와 거의 같고, 총 계산시간은 n에 비례한다. 이 충돌횟수를 table의 크기로 나누어 평균을 구하면 이는 각 가능한 원소들에 대한 중복의 평균이 된다. 이 값이 65보다 크면 Merge sort를 사용한다. (실험결과 Heap sort보다 Merge sort가 성능이 더 좋았다.) 마지막으로 count를 0이라고 두고, 배열의 ith 원소와 i+1th 원소를 비교하여 i+1th 원소가 크거나 같으면 count를 증가시키는 방식으로 바로 옆 원소와 정렬되어 있는 횟수를

구할 수 있는데 이 횟수를 배열의 크기로 나누면 배열이 얼마나 정렬되어 있는지 근사적인 수치를 구할 수 있다. 위에서 확인한 것과 같이 정렬되어 있는 배열을 정렬할 때 Insertion sort가 가장 유리하다. 따라서 ratio가 0.5보다 크면 Insertion sort를 그렇지 않으면 Quick sort를 사용한다. 각 step에 대해서 파라미터 tuning을 위한 실험은 기준이 되는 sorting과 나머지 중 가장 빠른 sorting의 정렬시간을 파라미터의 크기 변화에 따라 비교했다. (100번 반복 평균으로 실행시간 표기) 첫번째 실험결과로 데이터가 50000개일 때에 대해서 자릿수가 2이하일 때는 Radix sort가 그렇지 않을 때는 나머지 sorting중 가장 빠른 sorting이 더 좋았다. 두번째 실험으로는 데이터 개수가 50000개이고, 난수 범위를 조절하여 배열을 생성했을 때(난수 범위가 작을수록 충돌이 증가) hash table로 근사한 횟수가 60회 이하이면 Quick sort가 가장 빨랐고, 60회 이상일 때는 merge sort가 빨랐다. 이는 pivot 원소를 맨 뒤 원소로 잡았을 때 중복원소가 많아지면 불균형하기 partition되기 때문에 Quicksort는 느려진다. 이에 비해 Mergesort는 배열에 상태와 상관없이 일정한 성능을 유지한다. 세번째 실험으로는 sorted ratio의 변화에 따른 각 sorting의 실행시간을 측정하였는데 실험을 실행한 배열을 A[i-1]을 기준으로 k만큼의 확률로 A[i]를 A[i-1]보다 크게 1-k의 확률로 A[i]를 A[i-1]보다 작게 배열을 생성해 sorted_ratio의 값이 k에 근접하게 만든다. 이 경우 sorted ratio 값은 k로 유지되면서 전체 배열을 랜덤하게 만들 수 있다. 이러한 방법으로 배열을 생성하고, 실험하면 ratio가 0.54보다 클 때는 Insertion sort가 유리하고, 0.51보다 작고 0.49보다 클 때는 quick sort가 유리했다. 나머지 경우에는 일반적으로 merge sort가 유리했다. 수업시간에 배운대로 Insertionsort는 정렬된 배열을 정렬하는데 유리하고(역정렬은 불리), quicksort는 random하게 분포되어있는 배열을 정렬하는데 유리하며 Mergesort와 Heapsort는 배열의 상태가 관계없이 일정한 성능을 냄을 알 수 있었다.

자릿수	1	2	3
Radix	3.73	4.52	10.76
Rest	7.1	8.69	9.8

평균충돌	500	100	60	50	40
Quick	13.92	5.25	3.91	3.71	3.44
Merge	3.56	4.01	3.99	3.95	4.03
Heap	4.14	4.8	4.86	4.77	4.81

sort_ratio	0.7	0.6	0.55	0.54	0.53	0.51	0.5	0.49	0.4
Insert	0.56	1.07	2.24	2.57	3.98	29.28	256.02	427	477
Quick	overflow	303	162.34	127.57	79.26	14.54	2.54	9.6	124.52
Merge	2.86	3.37	2.98	3.14	3.19	3.49	3.52	3.43	3.01
Heap	3.59	3.61	3.78	3.83	3.87	4.2	4.31	4.28	3.77

4. Search의 동작 시간 분석(100회 수행 평균표기)

	5000	10000	50000
everysort	28.78	130.37	3907.29
search	0.41	0.84	4.2

데이터개수가 각각 5000개, 10000개, 50000개일 때 동작시간을 분석한 결과이다. everysort는 배열에 대해서 6개의 배열을 모두 실행한 시간의 합을 의미하고, search는 최적의 알고리즘을 찾고 이를 실행한 시간을 의미한다. 표를 보면 Search 메서드의 수행시간이 everysort보다 훨씬 적음을 알 수 있다. 또한 Search 시간을 보면 2배일 때 2배로 증가하고, 5배일 때 25배로 증가함을 알 수 있다. 이를 통해 search 알고리즘의 복잡도는 O(n)임을 알 수 있다.