HW4 Report

서울대학교 컴퓨터공학부 2020-12907 이현우

1. 알고리즘에 대한 설명

(1) Bubble Sort

Bubble Sort는 가장 큰 원소를 배열의 맨 뒤로 옮기면서(마치 거품이 위로 떠오르는 것처럼) 배열을 정렬하는 알고리즘으로, O(n^2)의 시간 복잡도를 가진다. 한 번의 cycle에서 배열을 앞에서 원소부터 순차적으로 탐색하면서 한 칸 뒤 원소와 비교하고, 앞에 있는 원소의 값이 더 크다면 두 원소의 위치를 변경한다. 이 cycle에서 결국 제일 큰 원소가 배열의 제일 뒤에 위치하게 된다. 다음 cycle에서는 제일 뒤로 고정된 원소를 제외하고 앞의 과정을 반복하게 되고, 결국 두 번의 중첩된 반복문을 통해 배열이 정렬된다.

(2) Insertion Sort

Insertion Sort는 Bubble sort와는 다르게 정렬된 부분이 배열의 앞쪽에 위치하게 되고, 원소들이 정렬된 부분 중 알맞은 위치에 삽입되는 알고리즘으로, O(n^2)의 시간 복잡도를 가진다. 배열을 정렬된 부분 A, 정렬되지 않은 부분 B로 나눈다면, 한 번의 cycle에서 B의 제일 처음 원소(key)에 대해 A를 탐색하게 된다. A의 제일 뒤 원소부터 탐색하여, key가 현재 탐색 중인 원소보다 작다면 두 원소의 위치를 변경한다. 이 작업은 A의 index 0까지 도달하거나 key가 현재 탐색 중인 원소보다 같거나 크면 중단한다. 위 과정을 배열의 전체 길이만큼 반복하게 되고, 이 또한 두 번의 중첩된 반복문을 통해 배열이 정렬된다. 하지만, Insertion Sort는 대부분의 key가 A의 원소들보다 클 때, 즉 거의 정렬된 배열에 대해서 O(n)에 가까운 효율적인 시간이 걸린다.

(3) Heap Sort

Heap Sort는 자료 구조 Max heap을 이용하여, max heap에서 root node의 값을 얻어 역순으로 배열을 정렬해나감과 동시에 heap을 수선하여 다시 max heap을 만드는 과정을 반복하는 알고리즘으로, O(n logn)의 시간 복잡도를 가진다. 가장 먼저, buildHeap을 통해 최대 힙을 생성하고, 최대 값에 해당하는 root node의 값을 배열의 제일 뒷 부분으로 이동시킨다. 그 후, 정렬된 부분을 제외한 부분에 대해 percolateDown을 통해 다시 힙을 수선하는 과정을 거쳐 배열의 최대값이 다시 root node로 이동하도록 한다. 이후부터는 위 과정을 거쳐 배열의 최대부터 최소까지 정렬하게 된다. 결국 이 알고리즘은 초기의 buildheap 과정(O(n)), 이후의 n개의 최대값 원소 제거에 대해 percolateDown 과정(O(logn))을 거치므로 총 O(n logn)의 시간 복잡도를 가진다.

(4) Merge Sort

Merge Sort는 배열들을 재귀적으로 반으로 나누는 과정을 거쳐 가장 작은 단위로 divide한 후, 작은 단위의 배열부터 정렬(conquer)하고 정렬된 배열들을 합치는 과정을 반복하는 알고리즘으로, 총 O(n logn)의 시간 복잡도를 가진다. 가장 먼저, 배열을 반으로 나눈 후, 반으로 나눈 배열에 대해 재귀적으로 merge sort를 시행한다. 이 과정은 나누어진 배열의 크기가 1이 될 때까지 시행하고, 배열의 크기가 1이 되면 배열들을 merge하는 과정을 거친다.

merge함수는 두 배열의 크기가 1이거나(이 경우 이미 정렬된 배열이다) 이미 정렬된 배열에 대해 수행하므로, 두 배열의 앞 부분부터 비교를 해 다른 배열에 작은 값부터 채워넣어 정렬을 진행한다. 이 과정은 모든 배열이 맨 처음 길이로 merge될 때까지 진행하고, 이 과정이 끝나면 정렬은 끝이 난다. 필자는 추가적으로 정렬한 값을 넣는 보조 배열을 매번 할당하는 대신, 정렬하고자 하는 배열의 복사본을 생성하여 merge sort 시 주 배열과 보조 배열의 역할을 바꿔가며 할 수 있도록 switching merge sort를 구현하였다. 해당 merge sort는 정렬된 배열을 merge하는 O(n) 과정이 logn번 반복되므로, O(n logn)의 시간 복잡도를 가진다고 할 수 있다.

(5) Quick Sort

Quick Sort는 임의의 값(pivot)을 기준으로, pivot보다 작은 값과 크거나 같은 값을 pviot의 왼쪽과 오른쪽으로 분리하여 재귀적으로 정렬하는 알고리즘으로, 총 O(n log n)의 시간복잡도를 가진다. 여기에 필자는, 중복된 원소가 많은 배열에 대해 처리속도를 높이기 위해 3-way partition을 구현하였다. 가장 먼저, 배열의 왼쪽 끝의 값을 pivot 값으로 설정한 뒤 pivot보다 작은 값들은 pivot보다 왼쪽인 제 1구역, 큰 값들은 pivot보다 오른쪽 구역인 제 2구역, 같은 값들은 pivot과 동일한 위치인 제 3구역에 위치하도록 하였다.(partition 과정). 이렇게 나누어진 세 부분 중 1구역과 2구역에 대하여, 재귀적으로 partition을 시행하고, 각 부분의 왼쪽 끝이 pivot으로 설정되어 앞과 같은 과정을 반복하게 된다. 이 과정은 partition된 부분들의 크기가 모두 1이 될 때까지 반복하여 정렬이 이루어지게 된다. partition된 구조를 보면 점근적으로 높이가 logn인 tree라고 할 수 있고, 각 partition을 하는 과정에서 O(n)의 시간 복잡도를 가지므로 전체 알고리즘은 O(n logn)의 시간 복잡도를 가진다고 할 수 있다. 하지만, 배열이 거의 정렬되어 있거나, 역정렬되어 있는 경우에 대해서는 skewed tree처럼 재귀가 호출되기 때문에 시간복잡도가 O(n^2)에 근사될 수 있다.

(6) Radix Sort

Radix Sort는 배열이 k 이하의 자릿수를 가진 특수한 경우에 대해, 일의 자리부터 차례대로 그 자리에 대해 stable한 정렬(같은 값을 가진 원소들은 정렬 후에도 원래의 순서가 유지되도록 하는 정렬)을 하는 알고리즘으로 자릿수 k번 만큼 O(n)의 counting sort를 진행하므로 총 O(n)의 시간복잡도를 가진다. 일반적인 radix sort는 음이 아닌 정수에 대해서만 정렬이 가능하므로, 값이 음수인 부분과 음수가 아닌 부분으로 배열을 나누고, 음수 배열에는 offset(최솟값의 절대값)을 더한 후 각각에 대해 radix sort를 진행하였다. 각 자릿수별로 정렬을 진행할 때에는 counting sort를 사용하였는데, count 배열을 생성하여 자릿수 값에 해당하는 인덱스 증가 및 누적합을 계산하여 원소의 개수를 통해 배열을 정렬하는 방식이다. 이러한 counting sort를 모든 자릿수에 대해 진행하였고(1의 자리부터 커지는 방식으로), 원소의 값이 같으면 이전 순서가 유지되도록 stable한 정렬을 진행하였다. 이후 offset을 다시 더한 음수배열과 양수배열을 이어붙여 정렬을 마무리한다. 이는 자릿수 k가 크기 n보다 충분히 작으면 O(n)의 시간복잡도를 가지므로 효율적인 알고리즘이라고 할 수 있다.

2. 동작 시간 분석

먼저 일반적은 랜덤한 배열에 대한 시간 측정을 위해, -100,000,000 ~ 100,000,000 범위에 대해 10^3, 10^4, 10^5, 10^6개의 데이터에 대해 각 정렬 알고리즘을 수행하였다.(실험은

macbook air m2 8GB에서 진행) 각 수행은 20번씩 반복하였고, 실험 결과는 아래와 같다.

Bubble Sort	배열의 크기			
	10^3	10^4	10^5	10^6
최대	3	88	10165	N/A
최소	1	63	9987	N/A
평균	1.55	73.55	10054.75	N/A
표준편차	0.589491	7.200521	52.60596	N/A

* 각 실험은 20번씩 진행(단위 : ms)

Insertion Sort	배열의 크기			
	10^3	10^4	10^5	10^6
최대	3	18	655	N/A
최소	0	14	621	N/A
평균	1.35	17.1	639.75	N/A
표준편차	0.852936	1.135782	9.637816	N/A

* 각 실험은 20번씩 진행(단위 : ms)

Heap Sort	배열의 크기			
	10^3	10^4	10^5	10^6
최대	1	6	26	133
최소	0	1	18	111
평균	0.2	2.85	21.75	127.05
표준편차	0.4	1.235921	1.920286	5.463287

* 각 실험은 20번씩 진행(단위 : ms)

Merge Sort	배열의 크기			
	10^3	10^4	10^5	10^6
최대	1	5	19	127
최소	0	1	16	107
평균	0.25	2.3	16.8	119.25
표준편차	0.433013	0.9	0.87178	4.656984

* 각 실험은 20번씩 진행(단위 : ms)

Quick Sort	배열의 크기			
	10^3	10^4	10^5	10^6
최대	2	4	22	119
최소	0	1	20	115
평균	0.4	2.2	20.85	116.95
표준편차	0.583095	0.678233	0.653835	1.28355

* 각 실험은 20번씩 진행(단위 : ms)

Radix Sort	배열의 크기			
	10^3	10^4	10^5	10^6
최대	3	6	26	98
최소	0	3	19	86
평균	1.4	4.8	21.2	90.95
표준편차	0.734847	0.87178	1.661325	2.355313

* 각 실험은 20번씩 진행(단위 : ms)

[표 1] 각 정렬 알고리즘과 크기에 대해 20번씩 실험을 진행한 결과

위 표를 보면, 대체적으로 n logn의 시간복잡도를 가지는 heap sort, merge sort, quick sort가 n의 값이 커질 때 좋은 성능을 내는 것을 확인할 수 있다. 특히, merge sort는 heap sort와 quick sort와 비교하여 모든 n 크기에 대해서 준수한 성능을 내는 것을 확인할 수 있고, quick sort는 n이 10^6 정도로 큰 값을 가질 때 merge sort와 heap sort보다 뛰어난 성능을 가지는 것을 확인할 수 있다. 또한, n = 10^6에서의 radix sort를 보면, 모든 알고리즘과 비교해서 제일 좋은 성능을 가지는 것을 확인할 수 있다. 이는 자릿수 k값에 비교하여 n값이 유의미하게 커져서, log n이 k보다 매우 커지기 때문에 radix sort의 수행시간인 k*n보다 커지는 것을 원인이라고 할 수 있다.

3. Search 구현

최적의 정렬 알고리즘을 찾는 search method를 구현하기 위해, 표1에서 살펴본 normal case에 더해 특이 케이스들을 더 살펴보겠다. 먼저, 배열의 최대 자릿수 k(=1, 2, 3, 4 ..)에 대하여 n에 따른 radix sort와 타 정렬 알고리즘을 비교해보았다.

	k = 1				
	10^3	5 * 10^3	10^4	5 * 10^4	10^5
Heap	1.4	1.65	2.45	3.75	6.25
Merge	0.9	1.25	1.75	3.35	6.15
Quick	0.7	1.15	1.30	2.30	4.30
Radix	0.8	1.15	1.35	2.45	4.35

	k = 2				
	10^3	5 * 10^3	10^4	5 * 10^4	10^5
Heap	1.2	1.80	2.75	5.9	6.8
Merge	1.10	1.55	2.45	4.6	6.75
Quick	0.9	1.15	1.85	2.10	3.90
Radix	0.85	1.15	1.90	2.45	4.10

	k = 3				
	10^3	5 * 10^3	10^4	5 * 10^4	10^5
Heap	1.20	1.75	2.15	4.9	9.23
Merge	1.10	1.45	1.95	3.6	5.75
Quick	0.8	1.15	1.85	2.10	3.90
Radix	1.15	1.55	1.90	2.25	4.2

[표 2] 자릿수 k = 1,2,3에 대한 실험 결과 (60번씩 진행)

Hash map을 이용하여 배열내에 얼마나 많은 중복 원소들이 있는지 확인하였고, 중복률(충돌발생 / 배열크기)에 따른 각 알고리즘의 수행 시간을 측정해보았다. 또한, 정렬도(i와 i+1이 오름차순인 비율)에 따라 알고리즘의 수행시간을 측정해보았다. 충돌률과 정렬도 실험 배열은 파이썬을 이용해 최대한 다양한 범위에서 배열을 생성하였다. 그 결과는 아래와 같다.

	충돌률				
	0.9995	0.999	0.9985	0.998	0.9975
Heap	63.25	65.35	63.65	67.65	68.45
Merge	62.85	63.55	64.25	67.95	66.15
Quick	26.55	30.55	38.15	42.15	43.25
Radix	31.55	32.75	32.15	31.95	32.75

* 각 실험은 20번씩 진행, 배열의 크기 50만(단위 : ms)

	정렬도				
	1.0	0.999	0.998	0.997	0.996
Heap	3.4	3.55	3.45	3.65	3.15
Insertion	0.9	8.15	8.55	9.65	9.75
Quick	1.75	1.65	1.75	1.65	1.75
Radix	1.95	2.15	2.15	2.15	2.15

* 각 실험은 20번씩 진행, 배열의 크기 5만(단위 : ms)

[표 3] 충돌률과 정렬도에 따른 실험 결과

위 표를 보면, radix sort는 k=3에서, n = 10^4일 때 quick sort를 제외한 다른 정렬 알고리즘들보다 빠르게 작동하는 것을 확인할 수 있다.(quick sort가 radix sort보다 빠른 이유는 높은 충돌률을 원인이라고 할 수 있다.) 즉, k가 a*log2(n)보다 작을 때, radix sort가 가장 좋은 성능을 내는 알고리즘이라고 할 수 있다. k=3부터 k=5까지 radix sort가 가장 빠른 성능을 내는 n값의 분기점을 찾은 결과는 (k=3, n=5590), (k=4, n=99160), (k=5, n=176200) 임을 확인할 수 있었다.(n을 10씩 변화하면서, 수행시간 오차범위 내의 n값의 평균을 냄) 따라서, radix sort가 빠른 값을 내는 파라미터 값을 0.241*log2(n)으로 정할 수 있었다. 그리고 정렬도에 대해서는, 배열이 정렬된 배열일 때, insertion sort가 가장 빠른 것을 확인할 수 있었고 그 외의 정렬도에서는 표준편차가 커 insertion sort로 확정지기는 힘들 것으로 보인다. 그리고, radix sort와 quick sort를 비교하여 충돌률이 약 0.9986정도 되어야 radix sort가 quick sort보다 빠를 수 있음을 확인할 수 있었다.

따라서, Search 알고리즘을 다음과 같은 flow로 작성하였다.

(1) insertion : sorted rate가 1.0일 때
(2) radix : 자릿수가 0.241 * log2(n)보다 작으면서, collision rate가 0.9986보다 작을 때
(3) quick sort : 이외의 경우

4. Search 동작 시간 분석

search가 최적의 알고리즘을 찾아내고, Search 동작 시간 + 해당 알고리즘 수행 시간이 모든 알고리즘을 수행하는 것보다 유의미하게 빠른지 확인해보았다. 이를 위해, 기존 Search 코드를 수정해 최적 알고리즘을 return하기 전 해당 알고리즘을 Do***()를 통해 수행하도록 하였다. 여러 배열에 대해 동작 시간을 분석한 결과는 아래와 같다.

	Search 동작 시간 분석							
	10^3		10^4		5 * 10^4		정렬된 배열(5 * 10^4)	
	Search	모두 동작	Search	모두 동작	Search	모두 동작	Search	모두 동작
최대	25	37	32	144	41	3145	37	3111
최소	7	25	20	128	31	2987	24	2876
평균	14.962962	30.592592	24.814814	135.44444	33.851851	3095.2592	29.666666	3010.5555
표준편차	3.5222202	2.4230262	2.5245026	5.6262172	2.9901530	46.113059	3.5276684	65.664316
	* 실험은 30번씩 진행, 랜덤 배열은 -10000000 ~ 10000000 에서 생성							

[표 4] search 동작 시간 분석

분석을 진행한 결과, n의 값이 커질수록 search 수행 + 해당 정렬 알고리즘 수행을 하는 시간이 모든 정렬 알고리즘을 일일이 수행하는 것보다 매우 빨라지는 것을 확인할 수 있었다. 특히, 정렬된 배열에 대해 seach를 수행하는 경우 search의 로직 앞 부분에서 'I' 리턴 및 알고리즘을 수행하므로, 동일 크기 배열에 비해 seach 및 수행시간이 유의미하게 빠른 것을 확인할 수 있다. 배열 크기 5 * 10^4 기준으로 모든 알고리즘을 동작시키는 시간의 약 90%는 Bubble sort의 수행 시간인데, 이를 제외하더라도 search의 수행시간이 모든 알고리즘을 동작시키는 시간의 1% 정도이므로, search를 통해 최적 알고리즘 탐색 및 수행이 매우 효율적이라고 할 수 있다.