HW4 Report

자료구조 (001)
2020-12852
최진호

1. 정렬 알고리즘의 동작 방식

1) Bubble Sort

배열의 원소 n 개가 있을 때, 정렬되지 않은 모든 원소에 대해 바로 옆에 있는 원소와 비교하여 더 큰 수를 오른쪽으로 보내는 동작을 n 회 반복하여 정렬하는 알고리즘이다.

2) Insertion Sort

각 원소에 대해 알맞은 자리에 삽입하는 정렬 알고리즘이다. 삽입 할 때는 배열을 뒤로 한 칸 밀어야 하므로 뒤에 있는 원소부터 자리를 옮긴다.

3) Heap Sort

배열을 힙으로 만든 뒤에 deleteMax() 를 이용하여 가장 큰 수를 계속해서 찾아내는 알고리즘이다. 찾아낸 가장 큰 수는 배열의 맨 뒤에서부터 채우며 힙의 크기를 줄여나간다.

4) Merge Sort

배열을 재귀적으로 잘게 쪼개서 정렬하는 알고리즘이다. 원래는 쪼갤 때 마다 메모리 공간을 사용하게 되는데, 수업에서 설명해주신 Switching-Merge Sort 를 구현하여 원래 배열의 딱 두배의 공간만 사용하도록 구현하였다.

5) Quick Sort

Pivot 을 설정하고 해당 pivot 을 기준으로 좌우를 작게 쪼개서 재귀적으로 정렬하는 알고리즘이다. 원래 수업과 책에서는 맨 뒤 원소를 Pivot 으로 잡는다. 그러나 이 방식은 이미 정렬된 배열은 시간이 오래 걸리고 Stack Overflow 가 일어날 수 있으므로 이 과제에서는 가운데 원소를 pivot 으로 설정하였다. 가운데 원소를 pivot 으로 설정하는 것 또한 최악의 경우를 만날 수 있지만, 일반적으로 정렬된 배열이 들어올 확률이 랜덤한 배열이 들어올 확률 보다 크다고 판단되어 이 알고리즘을 적용하였다. 또한 중복 원소에 대한 해결을 위해 수업시간에 설명해주신 짝수 번째 인덱스는 넘기고 홀수 번째 인덱스는 넘기지 않는 알고리즘을 적용하였다.

6) Radix Sort

각 자리 수에 대해 O(n)의 시간복잡도로 안정 정렬하여 결국 O(n)의 시간복잡도로 정렬할 수 있는 알고리즘이다. 따라서 각 자리 수에 대해 Counting Sort 로 정렬하는 알고리즘을 구현하였다.

2. 동작 시간 분석

각 정렬 알고리즘의 동작 시간 분석을 위해, 스켈레톤 코드에서 제공한 r number_conut min_number max_number 을 이용하여 동작 시간을 체크하였다.

각 테스트는 3~5 회진행하여 정렬 시간의 평균으로 분석했다.

1) data 의 개수에 따른 분석

먼저, r 10000 0 500000 부터 분석하였는데 100 개, 1000 개와 같이 데이터 셋이 비교적 작을 경우 각 정렬 시간이 0~1ms 로 큰 차이를 나타내지 않았기 때문에 10000 개부터 분석하였다.

k=1,000, m=1,000,000 / (단위: ms)

(count min max)	Bubble	Insertion	Heap	Merge	Quick	Radix
10k 0 500k	91	19	3	1	1	2
100k -500k 500k	18766	821	17	17	13	14
1m -5m 5m	-	85184	166	146	112	110
10m -5m 5m	-	-	1333	927	712	693

분석 결과, $O(n^2)$인 Bubble Sort 와 Insertion Sort 는 데이터의 개수가 많아지면서 빠르게 느려지는 모습을 확인할 수 있다. 하지만 Insertion Sort 의 경우에는 정렬된 경우 그 원소에 대한 정렬을 마치고 넘어가므로 Bubble Sort 보다는 크게 빠름을 확인할 수 있다. 하지만 일반적인 경우 Bubble Sort 와 Insertion Sort 는 적합하지 않음을 확인할 수 있다. 따라서 시간복잡도가 $O(nlogn)$ 인 Heap, Merge, Quick Sort 또는 $O(n)$인 Radix Sort 를 대부분의 경우에서 채택해야 한다고 생각할 수 있다.

2) data 최대 자리수에 따른 분석

위 실험에서, 데이터 천만 개(10m datas) 정도에서 차이가 눈에 보이므로, 이번엔 데이터를 백만개로 고정하고 최소와 최대를 조정하며 하이퍼파라미터 k 를 구하였다.

	Heap	Merge	Quick	Radix
10m -9999999 9999999	1335	958	727	621
10m -99999999 99999999	1417	947	716	709
10m -999999999 999999999	1441	948	713	786

	Quick	Radix
1m -999999 999999	95	77
1m -9999999 9999999	90	94
100k -9999 9999	18	16
100k -99999 99999	12	14

이 경우에서, 다른 정렬 방식은 데이터의 최대 자리수에 큰 영향을 받지 않았지만 기수정렬은 그 특성에 의해 큰 영향을 받는 것을 확인할 수 있다. 천만개의 data 에서, 7 자리 까지는 Radix Sort 가 빠르지만, 8 자리는 비슷하고 9 자리는 현저히 느려짐을 확인할 수 있다. 이후 Quick Sort 와 Radix Sort 둘 만을 비교한 결과, 데이터의 개수의 따라 k 가 달라져야 함을 확인할 수 있었다. 따라서 k 는 데이터가 백만개가 넘을 때, 데이터의 개수와 같은 자리수로 설정하였다. 또 Radix Sort 를 채택하지 않은 대부분의 경우의 수에서 Quick Sort 가 빠름을 확인할 수 있었다.

3) data 중복에 따른 분석

이번에는 스켈레톤 코드의 기능을 조금 수정하여 모두 같은 원소의 배열로 실험하였다. 모두 같은 원소의 배열로 배열의 크기만 달리해가며 실험한다.

i. 모두 같은 배열

Num count	Insertion	Heap	Merge	Quick	Radix
100k	4	10	8	10	7
1m	8	15	24	23	16
10m	9	29	113	195	72

위 결과를 통해 모두 같은 배열에서는 Insertion Sort 가 최선의 경우에서 기대했던 $O(n)$이므로 가장 빠른 것을 확인 할 수 있다. 따라서 배열이 어느 정도 중복되었을 경우에 Insertion Sort 가 빠를 것이라고 예상할 수 있다. 또한 배열이 중복되었을 때 Heap 을 구성하는 것도 빠르므로 Heap Sort 도 빠를 것이다. 따라서 하이퍼파라미터 k 와 적절한 정렬 방법을 구하기 위해 몇 차례 반복한다.

ii. 99.9% 중복된 배열

Num count	Insertion	Heap	Merge	Quick	Radix
10k	1	3	1	2	4
100k	7	6	5	7	13
1m	148	8	23	46	83

iii. 99 % 중복된 배열

Num count	Insertion	Heap	Merge	Quick	Radix
10k	3	1	1	1	3
100k	29	7	5	8	13
1m	1020	10	26	22	76

iv. 95% 중복된 배열

Num count	Heap	Merge	Quick	Radix
10m	168	313	216	708
100m	1922	3619	2426	7137

v. 90% 중복된 배열

Num count	Heap	Merge	Quick	Radix
10m	270	351	244	711
100m	3285	3960	2659	6952

위 결과를 통해 데이터셋이 커지면 거의 모든 원소가 중복되어야만 Insertion 가 빠름을 확인할 수 있다. 따라서 Insertion Sort 를 사용하는 경우는 제외하고, Heap Sort 가 Quick Sort 보다 빠르도록 하는 값의 범위를 구한다. 약 95% 정도 중복된 배열에서는 Heap Sort 가 더 빠름을 확인할 수 있다. Hash Table 의 크기를 13000019 로 설정하였으므로, 이 크기보다 큰 배열이 들어왔을 경우에는 충돌이 98% 일어났을 때, 아닌 경우에는 95% 일어났을 때 Heap Sort 를 반환하도록 하였다.

3. Search 동작 방식

위의 결과를 토대로, 가장 긴 숫자의 자리수가 데이터셋의 자리수보다 작고 데이터셋이 백만개 이상이면 Radix Sort, 이후 Hash Table 에 중복을 토대로 Heap Sort 가 적절하다고 판단되면 Heap Sort, 둘 다 아닌 일반적인 경우에는 Quick Sort 를 반환하도록 코드를 구성하였다.

4. 동작 시간 분석

3 에서 구현한 Search 에서, 평균 시간복잡도가 $O(n^2)$인 두 정렬을 제외한 4 가지 정렬을 모두 수행하는 시간과 만든 Search 에서 걸리는 시간을 확인하였다.

	Heap	Merge	Quick	Radix	Search
10k -99999 99999	4	3	3	5	25 (Q)
50k -99999 99999	15	9	10	11	27 (Q)
100k -99999 99999	24	14	16	20	28 (Q)
100k -999999 999999	18	14	15	21	32 (Q)
1m -999999 999999	173	144	119	115	10 (R)
1m -9999999 9999999	171	140	115	128	38 (Q)
10m -9999999 9999999	2147	1608	1277	1145	35 (R)
10m -99999999 99999999	2150	1637	1253	1208	85 (Q)

약 5 만 개 데이터부터는 Search 가 4 종류의 정렬을 모두 해서 비교하는 것 보다는 빠르지만, 백만 개 이하의 데이터에서는 그냥 Quick Sort 로 정렬하는 것이 Search 동작보다 빠르므로 실질적으로 사용할 때는 그냥 Quick Sort 로 정렬하는 것이 적절할 것임을 확인할 수 있다. 또한 데이터셋에 따라 조금의 편차가 있어 Search 가 찾는 정렬 방식보다 다른 정렬이 빠를 때도 있는데, 이는 동작시간의 10% 도 되지않는 차이이고, 이 차이를 잡아내려면 Search 동작의 소요시간이 길어질 것이므로 현재의 알고리즘이 적절한 타협지점을 찾아내었다고 생각 할 수 있다.