여러 가지 정렬 알고리즘의 비교

바이오시스템·소재학부

2019-13674 양현서

1. 정렬 알고리즘의 동작 방식

1-1. Bubble sort

가장 단순한 정렬 방법이다. 배열을 끝까지 하나하나 읽으며, 각 원소를 다음 원소와 비교하고, 더 큰 원소가 오른쪽으로 가게 서로 바꾼다. 이 과정에서 가장 큰 원소가 배열의 맨 끝에 위치하게 된다. 다시 배열을 순환할 때는 원래 배열 크기보다 1 작은 횟수로 위의 과정을 반복한다. 결국에는 배열이 정렬되어 있다.

1-2. Insertion sort

모든 요소에 대해, 끝에서부터 자신보다 큰 요소들을 뒤로 한 칸씩 옮겨가며 자신보다 큰 요소들을 모두 뒤로 밀었을 시 그 생겨나는 공간에 자신을 삽입하는 방법으로 정렬한다.

1-3. Heap sort

배열을 힙으로 바꾼 후, 힙에서 원소를 하나씩 빼서 결과 배열에 넣는다. 이렇게 하면 정렬된 배열이 된다. 배열을 힙으로 만들기 위해서는 자식 노드가 존재하는 인덱스 n/2 이하의 노드들에 대해 heapify를 호출한다. Heapify는 힙의 성질을 만족하게 하기 위해 자식 노드 중 큰 것과 교환한다. 그리고 변화된 노드에 대해 heapify를 호출하게 한다. 이렇게 하면 해당 배열이 힙이 된다. 힙에서 원소를 제거할 때는 힙의 마지막 원소와 루트 원소를 교환하고 heapify를 통하여 힙을 수선하게 하면 된다. 이러한 과정이 끝나면 정렬된 배열이 남는다.

1-4. Merge sort

배열을 반으로 나누어 앞의 부분을 merge sort, 뒤의 부분을 merge sort한 후, merge 함수를 호출하여 두 배열을 합친다. merge 함수에서는 두 배열을 앞에서부터 읽어가며 두 배열의 앞 원소 중 작은 것부터 선택하여 대상 배열의 앞에 집어넣는다. 이렇게 하면 정렬된 두 배열이 정렬된 상태로 합쳐지므로 전체 배열도 정렬된다.

1-5. Quick sort

Pivot을 기준으로 partition한 후, 앞부분을 quick sort, 뒷부분을 quick sort한다. partition을 할 때는 pivot을 정한 뒤 그 pivot보다 큰 원소는 pivot의 뒤로, pivot보다 작은 원소는 pivot의 앞으로 옮기는 것이다. 그 구체적인 방법은 0~lastPart1은 pivot보다 작은 영역, lastPart1~i는 pivot보다 큰 영역, i~끝은 아직 읽지 않은 영역으로 나누는 것이다. 어떤 영역에 원소를 추가할 때마다 해당 마커 변수들을 1 증가시키면 된다.

1-6. Radix sort

가장 작은 자리의 수부터 가장 큰 자리 수 기준까지 차례대로 보존적 정렬을 수행한다. 이번 구현에서는 counting sort를 이용하였다. Radix sort를 진행할 때는 대상 원소들이 차지하는 자리수를 미리 구해서 정렬을 하는데 필요한 반복 회수 등을 구해야 한다. 또 각 자리수 대상으로 정렬을 하기 때문에 10으로 나누거나 10으로 나눈 나머지를 반복적으로 구하는 듯 부가적인 연산이 더 많이 들어간다. 어차피 10으로 나누는 행위가 사람들이 10진법을 사용해서이므로 raidx sort를 할 때 꼭 10진법이 아닌 다른 진법을 사용해도 된다고 생각했다. 그래서 컴퓨터에 좀 더 친화적인 계산 진법인 2진법 계열을 사용하기로 결정하였다. 그런데 2진법을 사용하면 32비트 기준 32번이나 정렬을 수행해야 하기 때문에 좋지 않다고 생각을 하였고, 1바이트를 한 자리로 하는 256진법이 좋을 것이라고 생각하였다. Radix sort 함수에서 먼저 음수 처리를 위해, radix sort에서는 최상위비트도 그냥 크기를 나타내는 비트일 뿐이게 처리하는 것이 일관성이 있어 편리하므로, 0x80000000과 XOR하여 음수의 최상위 비트가 0이되고 양수의 최상위 비트가 1이 되게 변환한다. 음수의 경우 최상위비트를 0으로 바꿔주어도 음수끼리의 대소관계는 변하지 않으므로 잘 처리되는 것을 볼 수 있었다. 이렇게 하위 바이트부터 1바이트 단위로 count sort를 실행한다. 먼저 각 숫자들의 빈도를 세고, 그 빈도를 누적빈도로 변환한다. 이렇게 하면 해당 데이터들의 마지막 인덱스를 구하는 효과가 있다. 따라서 해당 인덱스 부분에 해당 데이터를 삽입하고, 그 인덱스를 1 감소시켜 감으로써 적절한 위치에 숫자들을 넣을 수 있는 것이다.

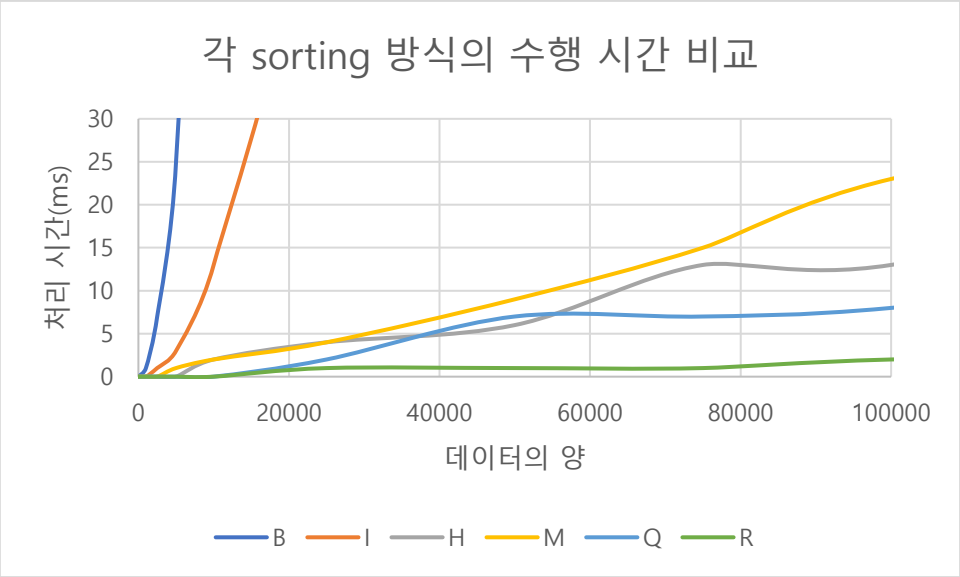
2. 동작 시간 분석

2-1. 이론적 Big O notation

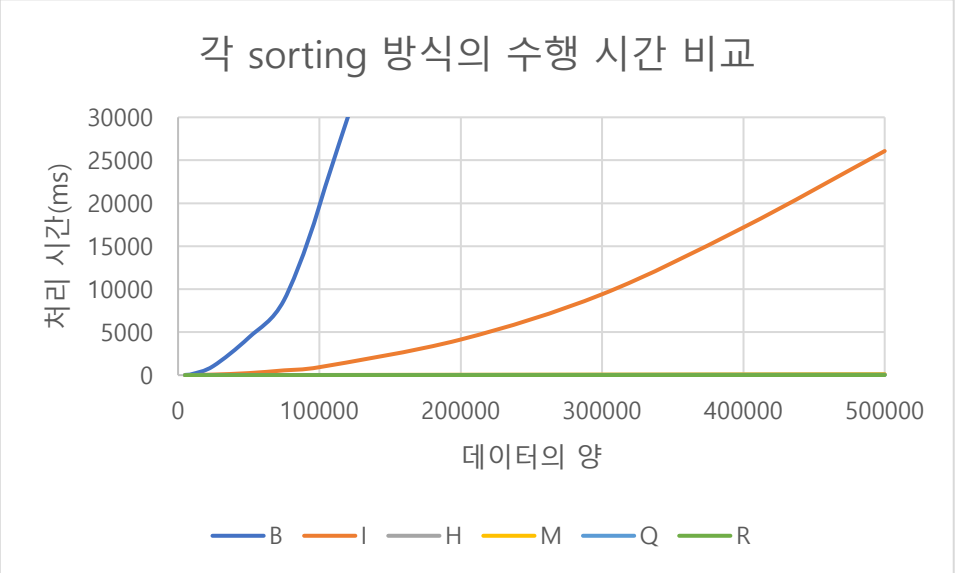
이름	Best case	Worst case	Average case
Bubble sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Heap sort	n log n	n log n	n log n
Merge sort	n log n	n log n	n log n
Quick sort	n log n	n^2	n log n
Radix sort	n	n	n

2-2. 실험 결과

데이터의 양 n=0, 1000, 2500, 5000, 10000, 25000, 50000, 750000, 100000, 200000, 300000, 400000, 500000에 대해서 각 sorting algorithm별 평균 소요 시간을 그래프로 나타냈다. B, I, H, M, Q, R은 각 sorting algorithm들의 첫 글자를 딴 것이다. 데이터의 개수가 1000개 이하일 때는 수행시간이 너무 짧아 별다른 차이를 볼 수 없었지만, 데이터가 20000개만 되어도 bubble sort와 insertion sort의 수행시간이 빠르게 증가하는 것을 볼 수 있다. 데이터가 60000개 이하일 때는 상수 요소에 의해 나머지 4개의 알고리즘의 수행시간의 순서가 바뀌는 것을 볼 수 있다. 그 이상의 데이터에서는 radix sort > quick sort > heap sort > merge sort 순서로 빠른 것을 볼 수 있다.



n이 매우 커짐에 따라 bubble sort와 insertion sort의 수행시간이 매우 빠르게 증가하는 것을 볼 수 있었다.



3. 결론

단순한 정수 정렬에서는 radix sort가 눈에 띄게 가장 빠르다. Merge sort와 Heap sort, Quick sort는 모두 n log n이지만 상수 factor의 차이에 의해 처리 성능은 quick sort, heap sort, merge sort 순서대로 좋은 것 같다. 대신에, 자료들이 매우 적은 경우에는 quick sort보다 heap sort나 merge sort가 성능이 더 좋은 경우도 존재했다.

JIT 컴파일에 의해 첫 번 수행보다 이후에 다시 수행되는 경우의 수행시간이 현저하게 줄어들었는데, bubble sort나 insertion sort와 같이 알고리즘 성능이 JIT 컴파일 시간을 압도하는 경우에는 별 차이가 없었지만, radix sort의 경우, 최초 수행 시 많은 시간이 걸리고, 다시 수행할 경우 수행시간이 50배정도 줄어드는 것을 볼 수 있었다. 그만큼 radix sort가 강력한 알고리즘이라는 것을 의미한다.

이번 실험에서 구현은 매우 단순하지만 시간복잡도가 n^2인 bubble sort 알고리즘, n~n^2인 insertion sort 알고리즘과 나머지 알고리즘들의 수행시간 차이에서 확인할 수 있듯이, 알고리즘을 잘 선택하는 것이 수행 시간을 적게는 수십 배에서 많게는 수백만 배 단축시킬 수 있다는 것을 알 수 있었다.