HW4 Report

2016-17097 유준열

1. 알고리즘 설명

1) Bubble Sort

서로 인접한 두 레코드를 비교하고 정렬한다. 1st iteration을 수행하면 가장 큰 자료가 맨 뒤로 이동하므로, 2nd iteration에서는 맨 끝에 있는 자료는 정렬에서 제외한다. 이에 따라 정렬을 수행할 때마다 정렬에 포함되는 데이터가 1개씩 감소한다.

- 시간 복잡도 : $\sum_{i=0}^{n-1}(n-i-1) = \frac{n(n-1)}{2} \sim O(n^2)$
- 최적화 : flag를 사용해 한 iteration에서 swap이 일어나지 않았다면, for loop를 break 해서 무의미한 iteration이 일어나는 것을 방지할 수 있다.

2) Insertion Sort

한 레코드를 그 앞의 레코드들과 비교하여 삽입할 위치를 결정하고 그 위치에 자료를 삽입하기 위해 자료를 한 칸 씩 뒤로 이동시킨다. 첫 key값은 두 번째 자료부터 시작하므로, 각 iteration에서는 대상 레코드의 위치를 이미 앞에서부터 정렬된 배열 내에서 찾게 된다.

- 시간 복잡도 : Best case $\sim O(n)$, Average/Worst case $\sim O(n^2)$

3) Heap Sort

Complete binary tree와 Max heap을 특성으로 하는 heap을 구현했다. Unsorted list의 n개 element에 대해, parent가 될 수 있는 (int) $\frac{n+1}{2} - 1$ index부터 작은 heap을 구성해 나간다. 새로운 element가 추가될 때 $0 \cdot \frac{n}{2^1} + 1 \cdot \frac{n}{2^2} + 2 \cdot \frac{n}{2^3} + \cdots = \frac{n}{4} \cdot \left(1 + 2 \cdot \frac{1}{2} + 3 \cdot \frac{1}{4} + \cdots\right) = \frac{n}{4} \cdot c = O(n)$이 소모된다. Leaf node에 해당하는 레벨을 d라 했을 때 레벨이 d인 node는 $\frac{n}{2}$개가 된다. 그러므로 last node가 root에 올라오기 위해서는 tree의 높이인 $h = \log_2 n$만큼 자리 이동을 해야 하므로 $O(\log n)$의 시간이 든다. 따라서, 삽입해야 할 element 수가 n개이라면 최종 max heap 구성에는 $O(n \log n)$의 시간이 소모된다. 이후 root를 last node와 swap하고 last node index를 1개씩 줄이는 과정은 $O(1)$이 n번 반복되므로 max heap을 정렬된 ascending list로 전환하는 것은 $O(n)$이 든다.

- 시간 복잡도 : $O(n) + O(n \log n) \sim O(n \log n)$

4) Merge Sort

Divide 와 merge가 recursive하게 일어나는 알고리즘을 구현했다. List의 길이가 1이면 이미 정렬된 것으로 보고, unsorted list를 받아 이를 recursive하게 비슷한 크기의 2개 부분 list로 divide한다. 모든 divide가 끝나면 각 부분 list를 다시 merge해 나감으로써 다시 하

나의 정렬된 list로 합병한다. Merge는 추가적인 temporary list를 필요로 하기 때문에 in-place sorting이 아니다. Merge 과정에서는 두 부분 list의 값들을 처음부터 한 개씩 비교해 더 작은 값을 temporary list에 넣어주는 작업을 한 부분 list가 끝날 때까지 진행한다. 이후 잔여 부분 list의 값들을 temporary list에 추가하고, 이를 원래의 list에 copy한다. 레코드의 개수 $n = 2^k$이라고 가정하면, merge가 일어나야 하는 recursive depth $k = \log_2 n$가 되므로 recursion의 시간 복잡도는 $O(\log n)$이다. 또한, 2^{k-1}개의 두 부분 list를 merge하는데 최대 2^{k-1}번의 비교가 필요하므로 n = 2^k일 때 최대 $2^{k-1} \cdot 2 = n$번의 비교가 필요하다.

- 시간 복잡도 : recursion depth($\log_2 n$) * 비교 연산 수(n) ~ $O(n \log n)$
- 최적화 : Linked list로 구현한다면 link index만 바뀌고 데이터 이동은 무시할 수 있을 정도로 작아지므로 크기가 큰 레코드를 정렬할 경우 매우 효율적인 알고리즘이 된다.

5) Quick Sort

Unsorted list 내 임의의 레코드를 pivot으로 설정하고, 이를 기준으로 list를 <p, >p인 두 부분 list로 분할한다. 구현한 코드에서 pivot은 center 값을 부여했으나, first/last/random 값을 부여해도 큰 차이는 없다. Partition은 부분 list의 크기가 1이 되기 전까지 recursive하게 적용되고, 내부에서는 pivot을 기준으로 왼쪽에서 pivot보다 큰 원소를 찾고 오른쪽에서 pivot보다 작은 원소를 찾아 이를 swap해준 후 left+1의 index를 mid값으로 넘겨 다시금 이전 pivot 보다 작은 값들의 list 내에서 partition이 일어나도록 한다. Recursive algorithm인 merge sort와 유사하게, 데이터 개수가 $n = 2^k$라면 recursive depth $k = \log_2 n$이 되고, 각 recursion 내에서는 전체 list의 대부분의 record를 비교해야 하므로 최대 n번의 비교가 필요하다.

- 시간 복잡도 : recursive depth($\log_2 n$) * 각 recursion의 비교 수(n) ~ $O(n \log n)$
- 최적화 : 어느정도 정렬된 list의 경우 quick sort의 불균형 분할에 의해 수행 시간이 오히려 많이 걸릴 수 있으므로, pivot을 first, last, center 레코드 중 median으로 선택한다면 더욱 list를 균등하게 분할하여 성능을 개선할 수 있다.

6) Radix Sort

Unsorted list의 절대값이 가장 큰 element를 찾아, 그 최대 자릿수까지 count sort를 적용했다. Count sort 내에서는 -값들을 0, +값들과 함께 판별하기 위해 element / digit에 %10을 한 값에 9를 더함으로써 -9~9까지 나오던 값을 0~18로 조정하여 count list의 해당 index의 값을 +1 해주는 방식으로 element의 개수를 셌다. 이후 count list를 모든 count가 누적되는 형태로 변환한 후, 여기에서 한 개를 extract해 output list에 넣을 때마다 값을 -1 해주어 output list에 들어갈 때 이전 digit에서의 결과가 고려되도록 구현했다.

- 시간 복잡도 : max digit을 k라 할 때 $O(kn) \sim O(n)$

2. 동작 시간 분석

차트 1은 [r element_size -10^7 10^7]을 input으로 주고, element_size를 변화시켰을 때 각 알고리즘 별 time complexity를 비교한 것이다. 한 input 당 10번의 시행을 반복하여 이를 평균 낸 값을 사용함으로써 통계적 유의성을 확보했다. 실험결과에 대한 보편적 해석은 다음과 같다.

1) $O(n^2)$: Bubble sort, Insertion sort

Element_size가 10^3개 이상으로 증가하는 log scale의 경우 Bubble sort와 Insertion sort는 다른 알고리즘에 비해 현격한 성능 저하를 보였다. 상대적으로 작은 data에 대해 작성한 차트2를 보면 Bubble sort는 작은 data의 경우에도 현저하게 많은 정렬시간을 소모하므로 가장 효율이 떨어지는 정렬방식이라고 정의할 수 있다. Insertion sort는 $O(n^2)$ 알고리즘 중에는 비교적 우수한 성능을 보이나 data의 크기가 10^3에 가까워질수록 $O(n\log n)$, $O(n)$ 알고리즘과는 큰 차이를 보이며 성능이 악화되므로 $O(n^2)$ 알고리즘은 비교대상 알고리즘 중 가장 성능이 떨어짐을 알 수 있다. **∴ 속도 : Insertion > Bubble**

2) $O(n\log n)$: Heap sort, Merge sort, Quick sort

차트2의 small data의 경우 알고리즘 별 성능 차이를 비교하기 어려우나, big data인 element_size가 $2\cdot10^5$ 이상으로 증가하는 차트3를 통해 $O(n\log n)$ 알고리즘 중에선 Quick sort가 가장 빠른 성능을 가짐을 확인할 수 있다. Heap sort는 $[2\cdot10^5, 8\cdot10^5]$ 범위에선 Merge sort에 대해 상대적으로 우수한 성능을 보이나 $(8\cdot10^5, 10^6]$ 범위부터는 Merge sort가 상대적 성능 우위를 갖는다. 이를 통해 10^6 이상의 data에서는 Merge sort가 Heap sort에 비해 효율적인 알고리즘일 것임을 추론할 수 있다. **∴ 속도 : Quick > Heap, Merge**

3) $O(n)$: Radix sort

Radix sort는 유일하게 $O(n)$을 따르는 알고리즘이지만, $\sim O(kn)$ 즉 maximum value element의 자릿수 k에 의해 영향을 받는다. 이에 따라, 차트3에서 Radix sort가 가장 우수한 성능을 가진 것으로 나오지 않은 것에는 기본적으로 10진법을 사용한 것의 영향이 크다. 만약 더 큰 진법을 사용했다면 메모리 사용량은 늘어났겠지만 성능의 개선 가능성이 있을 것이다. 이에 대해 10진법 Radix sort와 256진법 Radix sort 및 Quick sort의 성능 차이를 추가적으로 비교했다. 차트4를 보면 $[2\cdot10^5, 10^6]$ 범위에서 256진법 Radix sort는 10진법 Radix sort에 비해 개선된 성능을 보일 뿐 아니라 Quick sort보다도 시간 효율성이 뛰어난 것을 확인할 수 있다. 이에 따라, Radix sort가 가장 우수한 성능을 가진 알고리즘이라는 사실을 증명할 수 있다. 다만, Radix sort의 $O(n)$ time complexity를 최적으로 활용하기 위해서는, 진법의 크기를 증가시켜 최대 자릿수 k를 감소시키는 것이 중요하다. 동시에, 진법의 크기를 증가시키는 것은 메모리 사용량의 증가를 유발시키기 때문에 이에 대한 추가적인 고려가 이루어진다면 최적의 정렬 알고리즘을 구현할 수 있을 것으로 예상된다.

3. 차트

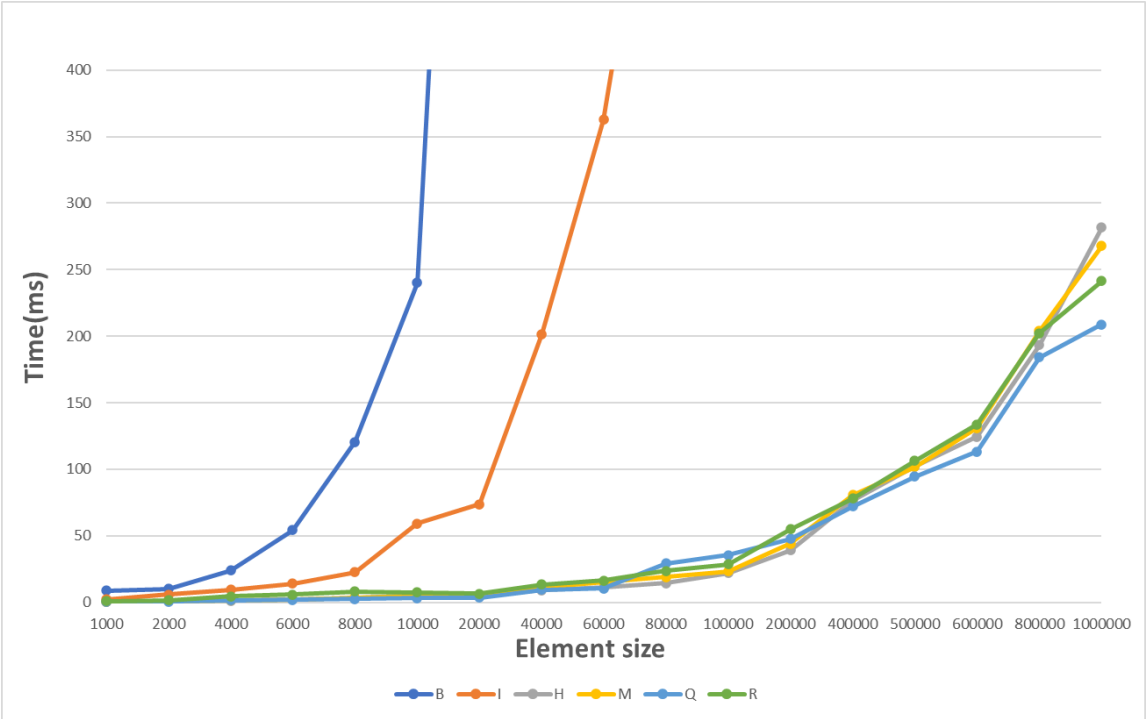

차트1 알고리즘 별 소요시간

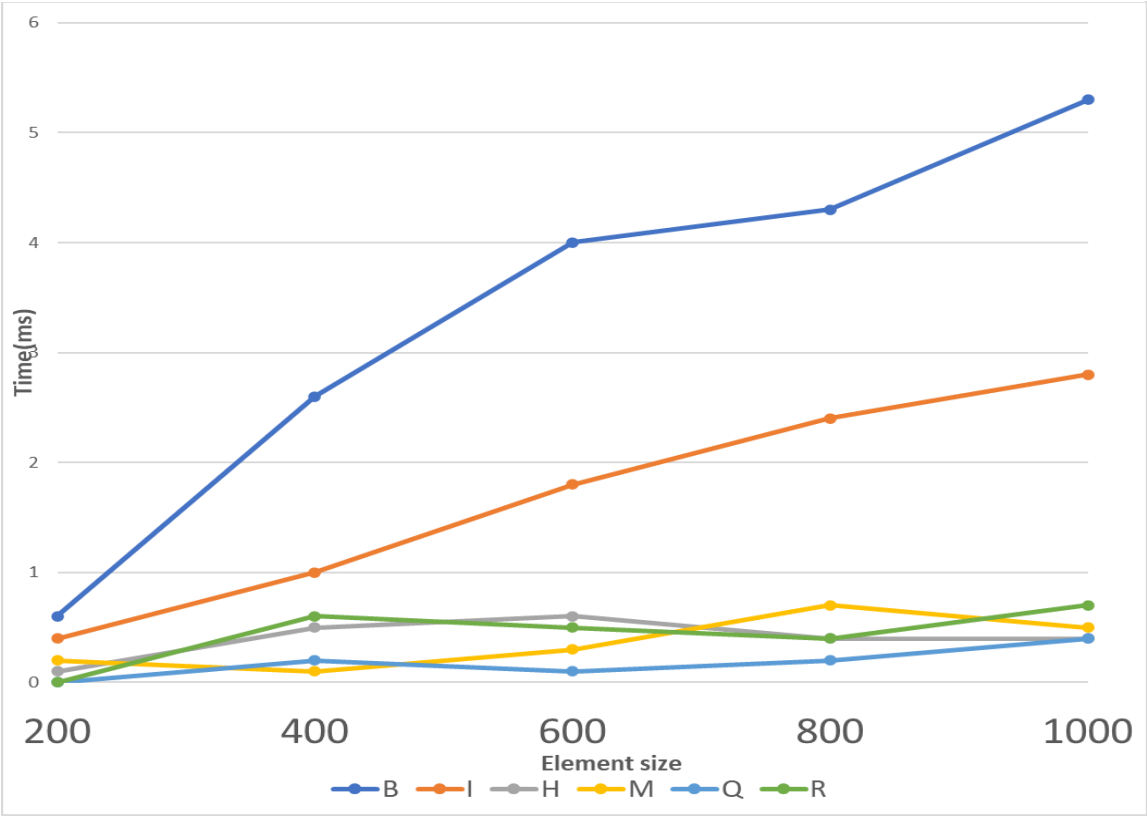

차트2 알고리즘 별 Small data 처리시간

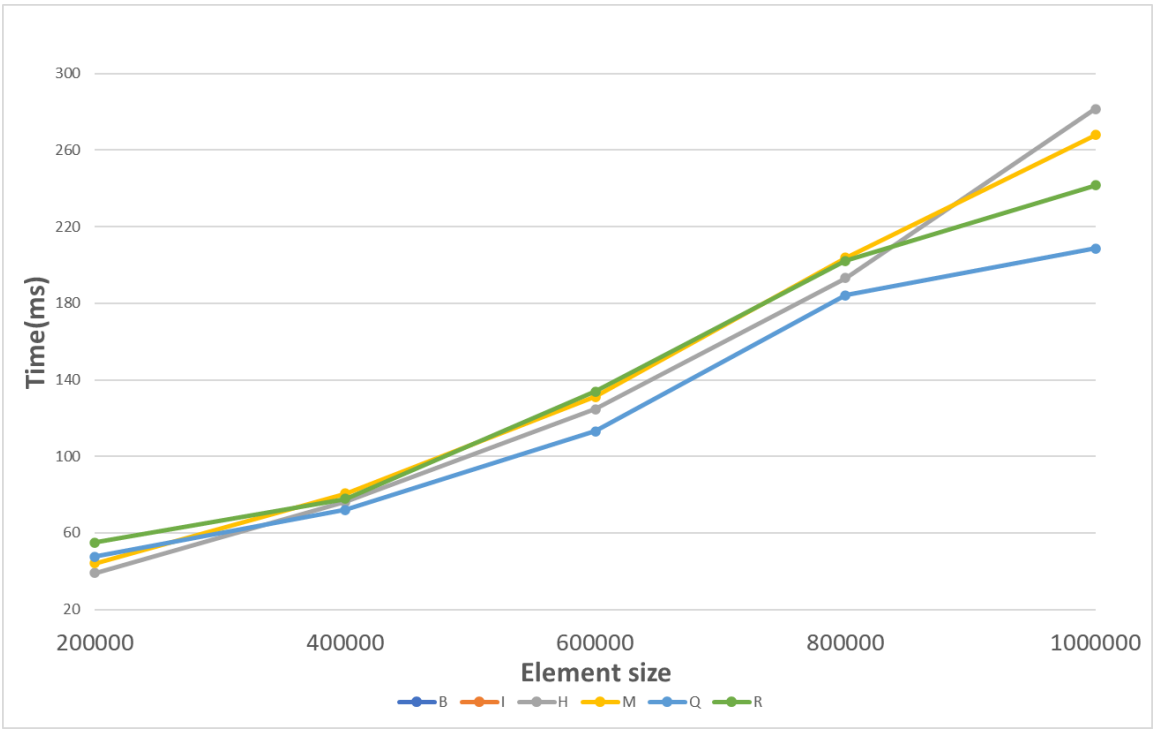

차트3 알고리즘 별 Big data 처리시간

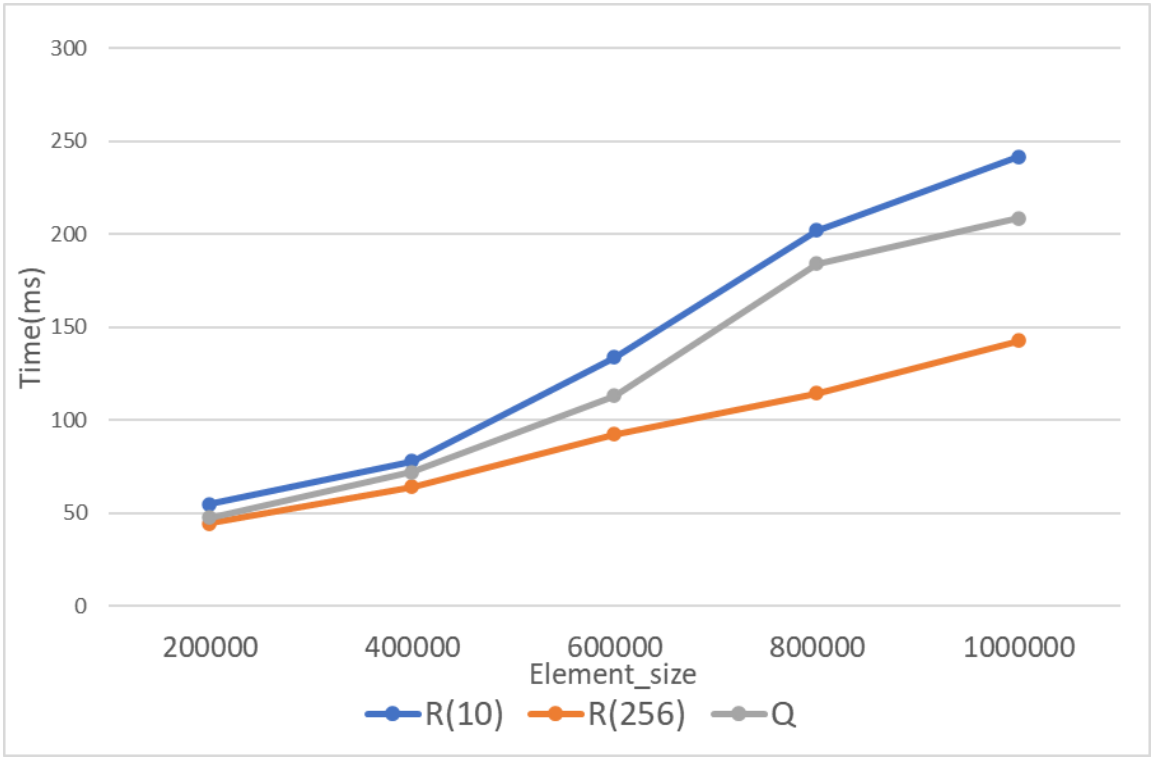

차트4 Radix sort(10진법), Radix sort(256진법), Quick sort 별 Big data 처리시간