# 자료구조 HW4 – Sorting Algorithm 비교

기계항공공학부 2015-17337 이동령

## 1. 정렬 알고리즘 동작 방식 설명

1) Bubble Sort

Bubble Sort에서는 더 큰 값이 배열의 제일 마지막으로 거품처럼 밀려간다. 배열의 시작에서부터 시작하여 k번째값을 k+1값과 비교하여 k번째 값이 더 크면 k+1번째값과 맞바꾼다. 이를 모든 원소에 대해 한 번 진행하면 제일 큰 값이 배열의 마지막에 위치하게 된다. 그 후, 배열의 제일 마지막 값을 제외하고 나머지 원소에 대해 같은 과정을 반복하면 그 다음 최댓값이 마지막에 위치한다. 이를 총 n-1번 반복하면 (n은 원소의 개수) 배열이 정렬된다.

2) Insertion Sort

Insertion Sort는 배열의 처음부터 시작을 하여 k번째 원소를 카피한 뒤 배열의 처음에서 k-1번째까지 카피한 k번째 원소와 비교하여 k번째 원소가 들어갈 인덱스를 찾는다. 인덱스를 찾은 뒤 삽입이 일어날 인덱스에서부터 아까 카피했던 k번째의 위치까지 원소들을 한 칸씩 밀어서 공간을 만든 뒤 카피해둔 k번째 원소를 삽입한다. 이 과정을 2번째 원소부터 n번째 원소까지 반복하면 배열이 정렬된다.

3) Merge Sort

Merge Sort는 재귀적인 방법으로 일어나게 된다. 적당한 키를 골라야 하는데 배열의 중간의 값을 키로 정한다. 그리고 키를 기준으로 왼쪽을 s1, 오른쪽을 s2라는 작은 배열로 나눈 뒤 s1과 s2에 대해서 똑같이 Merge Sort를 호출한다. 이러한 과정이 반복되면 배열은 최소 단위인 1개의 원소를 갖는 단위 배열로 나눠지고 단위 배열은 이미 정렬된 상태이다. 따라서, 단위배열부터 Merge를 해내가면 전체 배열이 정렬이 되는데 이 때 Merge는 두 정렬된 배열의 처음부터 시작하여 값을 서로 비교하여 더 작은 값을 결과가 되는 배열에 먼저 집어넣는다. 계속 작은 배열들의 1번째 원소들만 비교하여 넣고 어느 한쪽 배열이 다 소진되었으면 그대로 나머지 배열의 남은 원소들을 결과 배열에 붙히면 된다. 한 가지 단점은 Merge를 위해 추가적인 메모리가 필요하면 따라서 in-place sorting이 아니다.

4) Quick Sort

Quick Sort 또한 재귀적으로 호출된다. 적당한 키를 골라야 하는데 배열 내에 원소가 랜덤하게 들어갔다는 가정하에 마지막 원소의 값을 키로 정한다. 그리고 키 값을 기준으로 더 작은 값들을 L이라는 작은 배열로 더 큰 값은 R이라는 작은 배열로 나눈 뒤 L과 R에 대해서 똑같이 Quick Sort를 호출한 뒤 그 결과물을 L + 키 + R로 합치면 된다. 이것도

Merge Sort와 마찬가지로 재귀적으로 메소드를 호출하면서 배열이 쪼개지다가 단위 배열까지 쪼개지면 각각의 단위 배열은 이미 정렬이 된 것으로 볼 수 있다. 따라서 쪼갠 배열들을 L + 키 + R의 형태로 합쳐 나가면 정렬된 상태로 쪼개진 배열들이 합쳐지게 된다. Quick Sort는 Merge Sort와 다르게 추가적인 메모리 공간이 필요 없으므로 in-place sorting이다.

5) Heap Sort

Heap Sort는 heap이라는 자료 구조를 이용한 정렬 방법이다. Heap은 완전이진트리(complete binary tree)이며 i번째 노드의 자식 노드가 2i와 2i+1이라는 인덱스를 갖고 있다. 이때 heap은 배열의 형태로 구현될 수 있으며 첫 번째 노드를 루트로 삼아 배열의 인덱스를 완전이진트리의 노드의 인덱스로 생각할 수 있다. Heap Sort는 우선 heap을 만들어야 하는데 이 과정을 percolateDown이라는 메소드로 진행한다. PercolatedDown은 주어진 i번째 원소를 자식노드들의 원소와 비교하여 더 큰 값을 아랫 세대에 남겨둘려고 하는 메소드이다. 따라서 만약 부모노드가 더 큰 값을 갖는다면 가장 작은 값을 갖는 자식노드와 위치를 맞바꾼뒤 바뀐 (같은 값이나 이제는 자식 세대가 된) 노드에 대해 다시 percolateDown을 재귀적으로 진행시킨다. 이런 과정을 n/2번 반복하면 주어진 배열이 heap의 형태를 띈다. 이 때 루트를 가장 마지막 원소와 맞바꾼뒤 바뀐 루트에 대해서 다시 percolateDown을 진행시키고 같은 프로세스를 n번 반복하면 배열이 정렬되어있다. 루트를 마지막 원소와 맞바꾸는 행위는 추상적으로는 루트를 제거한 상태이며 루트가 제거됐을 때 가장 마지막 원소를 루트로 바꾼 뒤 percoteDown을 이용하여 새로운 heap의 형태를 갖추는 heap의 원소 제거 방법을 응용한 정렬방법이다.

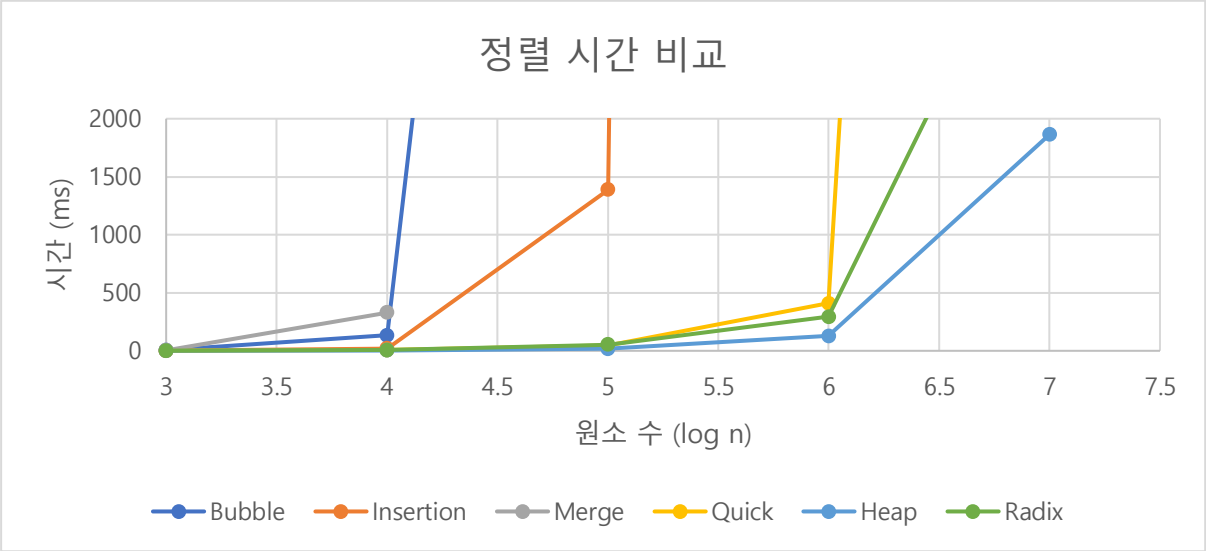
6) Radix Sort

Radix Sort는 같은 자릿수의 숫자들에게만 사용될 수 있기 때문에 배열 원소 중 가장 큰 자릿수에 맞춰서 진행된다. Radix Sort에서는 숫자들이 가장 낮은 자릿수부터 정리된 뒤 자릿수를 하나씩 올려가며 같은 과정을 반복하면 정렬되는 방식이다. 따라서, 정렬 전 bucket이라는 -9에서 9까지 모든 자릿수를 담을 수 있는 bucket이라는 linked list를 원소로 하는 배열을 만들어야한다. 그런 뒤, 주어진 배열의 마지막 자릿수를 분리하여 자릿수에 맞춰 원소들을 bucket안에 집어넣는다. Bucket의 원소는 linked list기 때문에 마지막 자릿수가 겹쳐도 다 담을 수 있다. 그 다음 -9 (제일 작은 자릿수)부터 존재하는 원소들을 R 꺼내서 배열에 담으면 어느 정도 정렬이 되어있을 것이다. 이 과정을 가장 큰 자릿수만큼 반복하면 모든 원소들이 정렬되게 된다.

## 2. 각 정렬방식의 동작 시간 비교

Range (-1000~1000)

| n | 시간 (ms) | | | | | |
|---|---|---|---|---|---|---|
| | Bubble | Insertion | Merge | Quick | Heap | Radix |
| $10^3$ | 6 | 2 | 4 | 1 | 1 | 1 |
| $10^4$ | 132 | 21 | 328 | 7 | 4 | 6 |
| $10^5$ | 16408 | 1389 | Out of mem | 45 | 16 | 52 |
| $10^6$ | 1583018 | 138179 | Out of mem | 408 | 128 | 293 |
| $10^7$ | Too long | Too long | Out of mem | 31220 | 1864 | 4158 |




Bubble Sort와 Insertion Sort 모두 $O(n^2)$여서 n을 10배씩 증가시킬 때 마다 실행시간이 두 자릿수 증가함을 관찰할 수 있으며 $n = 10^7$ 인 경우는 너무 오래 걸리기 때문에 test를 하지 않았다. Merge Sort는 배열의 각 원소를 모두 단위배열로 분리하여 처리해야하기 때문에 $10^5$개 이상의 데이터에 대해서 메모리 부족으로 작동하지 않았다. 이를 통해 Out-place 정렬의 문제점이 극한의 상황에서 생기는 문제에 대해 알 수 있다. Quicksort는 $\theta(n \log n)$, $O(n^2)$이고 Heapsort는 $O(n \log n)$이므로 Quicksort가 Heapsort보다 증가하는 폭이 더 크다. 마지막으로 Radixsort는 $O(n)$이기 때문에 모든 n이 커질수록 모든 정렬방식 중에서 제일 빠르다. 위 그래프에서 x축은 현재 로그 스케일이므로 linear하게 증가하는 Radixsort는 지수함수의 형태를 띈다.

| Range ($n = 10^4$) | 시간 (ms) | | | | | |
|---|---|---|---|---|---|---|
| | Bubble | Insertion | Merge | Quick | Heap | Radix |
| -10 ~ 10 | 126 | 21 | 337 | 22 | 2 | 4 |
| $-10^2$ ~ $10^2$ | 125 | 22 | 330 | 6 | 2 | 6 |
| $-10^3$ ~ $10^3$ | 133 | 22 | 326 | 4 | 3 | 6 |
| $-10^4$ ~ $10^4$ | 134 | 22 | 335 | 4 | 2 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| $-10^5 \sim 10^5$ | 123 | 21 | 309 | 4 | 2 | 8 |

예상과는 다르게 Quick Sort의 경우 원소의 범위가 작을 때 더 오래 걸렸다. 원소의 범위가 적고 10000개의 원소를 생성하다 보면 중복되는 원소가 많아지는데 중복되는 케이스를 처리하는데 더 시간이 오래 걸리기 때문이라고 생각된다. 특히, Quick Sort의 경우 마지막 원소를 pivot으로 삼아 작동하는데 범위가 작을수록 pivot이 원소의 범위의 중앙이 아닌 양 극단으로 치우칠 가능성이 높고 pivot이 한 쪽으로 치우칠수록 효율이 떨어지므로 이러한 현상이 일어난 것으로 보인다. 따라서, 범위를 늘릴수록 원소가 중복될 확률도 떨어지고 극단으로 치우칠 확률도 낮아져서 특히 Quick Sort의 경우 원소수가 같다면 원소의 범위가 넓을수록 효율성이 올라가는 것으로 보인다. 이러한 성질은 Radix Sort를 제외한 다른 정렬방식에서도 보이나 그 양상이 미미해 보인다. 이는 다른 정렬방식들은 pivot을 사용하지 않거나 pivot이 사용되는 Merge Sort의 경우 결국 원소 하나하나로 단위 배열까지 쪼개므로 pivot의 치우침의 영향을 크게 받지 않기 때문으로 보인다. 반면, Radix Sort는 자릿수만큼 과정을 반복하므로 원소의 범위를 10배로 늘릴 때 마다 자릿수가 증가하여 걸리는 시간이 linear하게 증가하는 양상을 보인다.