Sorting - Report

경영학과 2016-14877 송재윤

1. 정렬 알고리즘의 동작 방식

- **Bubble Sort**
  - 바깥 for loop = 가장 큰 원소를 끝자리로 옮기고 정렬대상 줄이는 작업 반복
  - 안쪽 for loop = 가장 큰 원소를 끝자리로 옮김. (왼쪽부터 이웃한 수를 비교하면서 하나씩 바꿔나감)
- **Insertion Sort**
  - Bubble Sort와는 반대로 정렬이 안 된 대상을 하나씩 줄여나감. 즉 정렬된 부분을 하나씩 늘여 나감.
  - 안쪽 for loop에서 `newItem`보다 큰 애들은 `value[loc+1] = value[loc]`로 한 칸씩 뒤로 가게 된다.
  - loop를 빠져 나왔을 때 `loc` 값은 `newItem`보다 같거나 작은 첫번째 요소의 index이므로, 바로 뒤에 `newItem`을 위치시킨다.
- **Heap Sort**
  - 1단계: MaxHeap을 만들어준다. 이때 `n/2`는 최초로 heap 수선이 필요할 수 있는 부모이다.
  - 2단계: root에서부터 하나씩 삭제해나가며 배열의 맨 뒤로 보낸다. 삭제하고난 뒤에는 다시 `percolateDown`으로 heap 수선을 해준다.
    - 현재 노드의 2개의 child 중 큰 것을 찾고, 그 index를 `child`에 담는다
    - `child`의 key 값과 현재 노드의 key 값을 비교하여 `child`가 더 클 경우 `swap` 하고, recursive하게 leaf 노드까지 쭉 내려간다.
- **Merge Sort**
  - 중앙점 `int q = (p+r)/2` 를 pivot으로 잡아준다.
  - pivot의 양쪽을 recursive하게 mergeSort한다.
  - 후처리로 양쪽을 merge해준다.
    - 임시적으로 합치기 위한 공간 `tmp`를 생성한다.
    - 양쪽을 차례대로 비교해나가며 작은 것부터 `tmp`에 넣어 차곡차곡 정리한다.
    - 비교가 끝나고 한 쪽에 원소가 남았는지를 체크하여 남은 경우 그대로 넣어준다.
- **Quick Sort**
  - `partition` 메소드에서 pivot을 중심으로 더 작거나 같은 원소는 좌측으로, 더 큰 원소는 우측으로 재배치한다. 재배치 이후 pivot의 index를 `q`로 리턴받는다.
    - 마지막 원소를 pivot으로 잡아주었다.
    - for loop을 돌면서 pivot을 둘러싼 3가지 구역이 생긴다고 볼 수 있다.

      1) pivot보다 작거나 같은 원소 - if문에서 `swap`을 통해 i, j를 증가시켜 한 칸 넓어진다.

2) pivot보다 큰 원소 - loop를 돌며 j가 증가하여 자동으로 한 칸씩 넓어진다.

3) 아직 비교되지 않은 원소 - loop를 돌며 줄어든다.

- `q`의 양쪽을 recursive하게 quickSort한다.

- **Radix Sort**
  - 주어진 수들의 절대값 중 최대값 `max`를 찾는다.
  - 최대값의 자리수 즉 10의 거듭제곱 형태인 `maxDigits`를 구한다.
  - 1의 자리수부터 `maxDigits`의 자리수까지 loop를 돌며, 한 번의 loop에서는 해당 자리수에 대해 stable sort를 수행한다. 핵심적인 내용은 다음과 같다.
    - 두 개의 for loop를 사용해서 -9 ~ 9의 19개 공간을 `counter`로 준비해놓고, 각각의 수를 해당 자릿수로 가진 원소를 세어준다.
    - `counter[i] += counter[i-1];` 를 이용하여 개수를 index+1 값으로 만든다.
    - `value`와 같은 크기의 임시적인 공간인 `tmp`에서 이 index 값에 대응되는 공간에 `value`의 원소들을 이동시켜준다.
    - `tmp`를 `value`로 복사해 넣는다. 이를 `maxDigits`에 이를 때까지 반복한다.

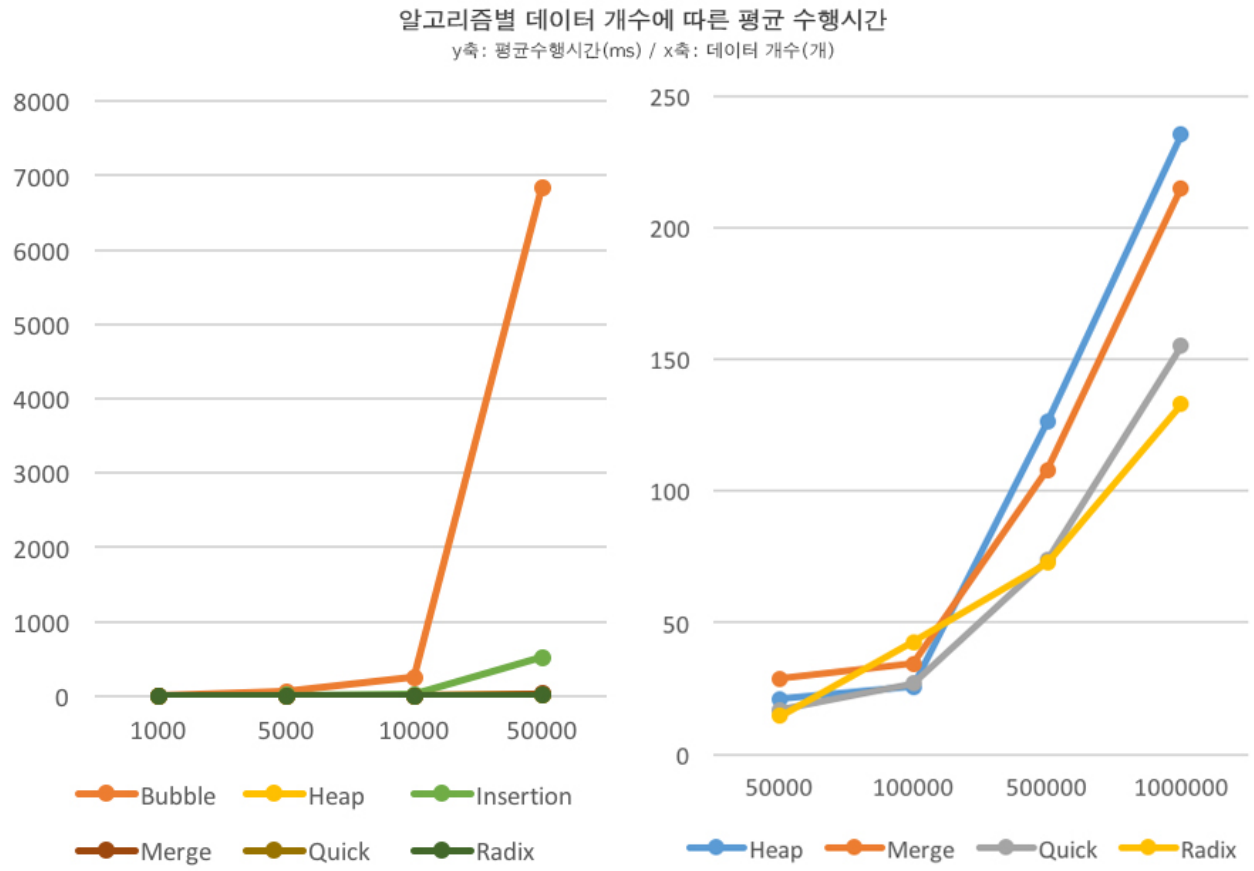
2. 동작 시간 분석

- **원소 개수** 1000, 5000, 10000, 50000, 100000, 500000, 1000000에 따른 수행시간을 비교 분석하였다. BubbleSort와 InsertionSort는 시간상 50000개까지만 시행하였다.
- **원소의 범위**는 -100,000~100,000의 정수형 난수로 설정하였다.
- **측정 횟수**는 각각의 알고리즘을 동일한 개수의 서로 다른 난수에 대하여 100회씩 측정하였다.

> **정렬 알고리즘별 data 개수에 따른 수행시간 그래프**

좌측은 6개의 알고리즘의 수행 시간을 1000, 5000, 10000, 50000개의 데이터에 대하여 측정한 그래프이다. 그래프 양상으로부터 점근적(asymptotic) 측면에서 BubbleSort와 InsertionSort의 효율성이 취약함을 쉽게 알 수 있다.

다음으로 우측은 지나치게 오래 걸리는 BubbleSort와 InsertionSort를 제외하고 나머지 알고리즘의 수행시간을 50000, 100000, 500000, 1000000개의 데이터에 대해 측정한 그래프이다. 점근적인 측면에서 average 및 worst case efficiency가 `O(n)`으로 알려진 Radix Sort는 가장 효율적으로 나타나며, `O(n*logn)`으로 알려진 Heap, Merge, Quick Sort 비해 추세선이 일차형에 가깝다.



> **Data가 10000개일 때 정렬 알고리즘에 따른 수행시간 통계 수치**

10000개의 데이터는 점근적이라고 하기에 매우 작은 수치임에도 불구하고 Bubble Sort와 Insertion Sort의 비효율이 명백히 드러난다.

또한 Quick Sort의 경우 Average Case 효율성은 Quck, Merge Sort와 마찬가지로 n*logn에 비례하여 유사한 값을 가지지만, Worst Case 효율성 즉 아래 표에서 최대값 수치를 보면 Quick, Merge에 비하여 비효율성을 내포하고 있음을 유추할 수 있다.

Radix Sort는 점근적으로 가장 효율적이지만, 10000개의 적은 표본 때문에 아래 표에서는 Heap, Quick, Merge Sort와 유사하게 나타나고 있다.

단위:ms	Bubble	Insertion	Heap	Merge	Quick	Radix
최대값	286	225	7	14	21	10
최소값	210	17	3	4	3	4
중간값	249	23	4	5	4	5
평균	249.73	25.5	4.52	5.16	4.66	5.29
표준편차	13.18	20.99	0.83	1.44	1.91	1.12