2014년 5월

Data Structure HW4 Report

- **정렬 알고리즘의 동작방식**
  - Bubble Sort

입력된 array의 첫번째 원소부터, index i와 i+1, 두개씩 원소를 짝지어 검사한다. 두 원소를 짝지어 검사할 때마다 큰 수가 오른쪽으로 오도록 swap 하고 오른쪽으로 이동한다. 이렇게 마지막 숫자까지 검사하는 1번의 routine을 'pass' 라고 하는데, 반복문을 통하여 큰 수를 오른쪽에 배치하므로, pass #1가 끝나면 array의 맨 오른쪽에는 가장 큰 원소가 위치하게 된다.

pass #1이 끝나면 정렬이 완료된 맨 오른쪽의 원소를 제외한 나머지 원소에 대해서 pass 를 시행한다. 이렇게 총 n번의 pass를 거치면 숫자의 오름차순 정렬이 완료된다.

  - Insertion Sort

입력된 array의 첫번째 원소부터 하나씩 오른쪽으로 이동하며 검사하면서, 검사하는 원소를 적절한 위치로 insertion 해주는 것이 기본 알고리즘이다.

index i의 원소(a라 하자)를 Copy 해놓고, 만약 a가 앞의 원소들보다 작으면, a 앞에 있으면서, a 보다 큰 원소들을 오른쪽으로 한자리씩 shift 해준다. 그리고 마지막으로 shift 한 원소의 원래 자리에 a를 넣어주면 index i 까지의 정렬이 완료된다.

입력된 array에서 i = 0 to n-1 (n은 원소의 개수)에 대해서 위의 과정을 반복해주면 숫자의 오름차순 정렬이 완료된다.

  - Heap Sort

자료구조인 Heap 을 이용하는 정렬 알고리즘이다. 여기서는 입력된 array를 'Max Heap'으로 만들어준 다음에 sort 해준다. Max Heap 은 부모 노드가 자식 노드보다 큰 complete tree 구조를 의미한다.

Max Heap 생성을 위해서, 입력된 array의 index 0 부터 index (n/2 - 1) 까지의 원소들을, (n/2 - 1) to 0 의 순서로 'Percolate Down' 한다. Percolate Down 은 특정 노드를 2개의 Child node 중 큰 node의 값과 비교하면서 swap, recursive 하게 leaf 노드까지 쭉 내려가는 과정을 말한다. Percolate Down을 통해서 heap property를 만족하도록 tree 를 수선해줄 수 있다. (index (n/2) 부터 index (n-1) 까지의 원소들은 이미 heap의 leaf node로, 그 자체로 heap을 형성하고 있기 때문에 percolate Down 해줄 필요가 없다.)

Max Heap을 생성하고 나면, root 노드엔 가장 큰 원소가 위치하게 된다. 이 원소를 빼내서 array의 맨 마지막 원소와 swap 하면, 가장 큰 원소가 array의 맨 뒤에 위치하게 되고, root 노드엔, 기존의 맨 마지막 원소가 위치하게 된다. 제 자리를 찾아간 가장 큰 원소를 제외한 원소들에 대해서, root 노드의 원소를 Percolate Down 해주면 다시 heap 구조가 만들어진다. 이런 식으로 root 노드의 원소들을 하나씩 빼내고, Percolate Down 하는 과정을 반복하면 숫자의 오름차순 정렬이 완료된다.

  - Merge Sort

Recursive 알고리즘으로, 주어진 array를 절반으로 나누어 각각의 sub array를 Merge Sort 한 후 둘을 Merge하는 방식이다.

여기선 정렬이 완료된 두개의 sub array를 병합해주는 Merge 알고리즘이 핵심이다. 각각의 sub array를, subArray1과 subArray2라고 하고, 역시 각각 sub array 원소들을 검사하는 index를 i1, i2 라고 하자. i1, i2 가 가르키는 원소 중 작은 원소를 임시 저장장소인 temp array에 앞에서부터 넣어준다. 예를 들어 i1의 원소가 i2 의 원소보다 작으면 i1의 원소를 temp array에 넣어주고, 그 다음엔 i1+1 의 원소와 i2 의 원소를 비교해준다.

temp array에 sub array의 원소들을 넣어주는 작업이 완료되면, temp array에는 원소들이 오름차순 정렬되어, merge가 완료된다.

recursive 하게 이를 반복하면, 주어진 array의 숫자들의 오름차순 정렬이 완료된다.

- Quick Sort

Partition을 이용하는 정렬 알고리즘이다. Partition 이란, 주어진 data 들에 대해서, pivot item, p를 잡고, (p보다 작은 원소들, p, p보다 큰 원소들)로 배치해주는 것을 의미한다.
여기선 pivot index를 random 하게 잡아서, Quick sort의 worst case를 피할 수 있도록 했다.
random 하게 잡은 pivot item는, precondition 방식으로 구현하기 위해, 일단 array의 맨 앞에 위치 시켰다. 그리고 array의 원소들을 하나씩 검사하면서 pivot item 보다 작은 원소들은 S1 집합에, 큰 원소들은 가장 마지막 S1 원소 바로 뒤부터 시작되는 S2 집합에 넣어준다. precondition 방식은, p보다 작은 원소가 나타나면, S2의 첫번째 원소와 swap 해준다. 이렇게 하면, S1의 크기가 하나씩 늘어나며, p보다 작은 원소와 p보다 큰 원소가 분리 된다.
S1, S2가 분리되고 나면, p를 올바른 위치, 즉 S1, S2 사이에 넣어준다. S1의 마지막 원소와 맨 앞에 있던 p를 swap 해주면 이 작업이 완료된다.
Partition 작업을 마치면, S1과 S2 각각에 대해서 다시 Partition을 recursive 하게 진행한다. 원소 한개에 대한 partition, 즉 base case로부터 역으로 거슬러 올라오면 주어진 array의 숫자들의 오름차순 정렬이 완료된다.

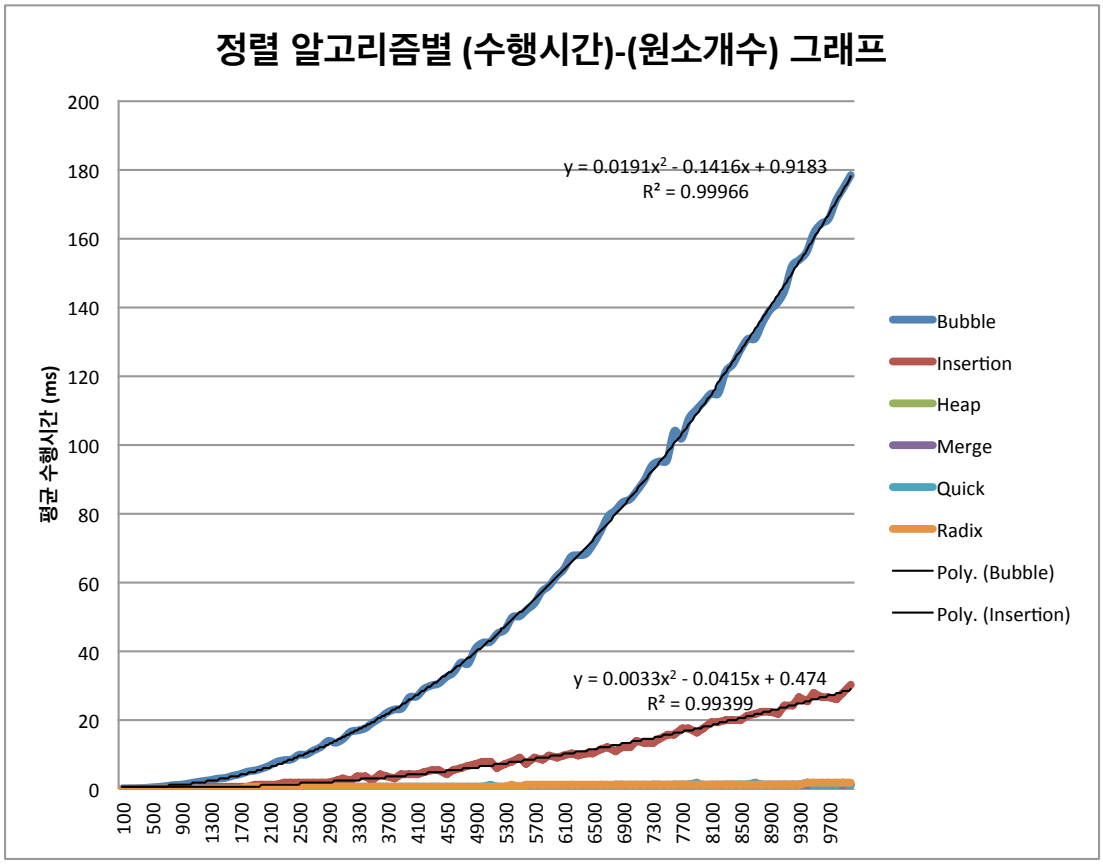
- Radix Sort

Radix Sort를 시행하기 위해 입력된 array에서 제일 큰 자리수를 가진 원소를 먼저 찾아준다. 이 원소의 자리수를 d라고 하자.
1의자릿수에 대해서 원소들을 stable sort 해준다. d 자리수까지 stable sort를 반복해주면 원소의 오름차순 정렬이 완료된다.
여기서 stable sort는 k번째 자릿수에 대해서(1의 자릿수를 k=0, 10의 자릿수를 k=1… 이렇게 정의해준다고 하자), 자릿수 숫자가 같은 원소들을 그룹으로 묶어서 오름차순으로 정렬해준다. (k번째 자릿수가 0인 그룹, 1인 그룹, 2인 그룹 … , 9인 그룹) 이 때 한 그룹 내에서 원소들의 순서는, k번째 자릿수에 대해서 stable sort 하기 이전의 순서를 따른다.
stable sort는 다음과 같이 구현하였다. 먼저 k번째 자릿수가 0, 1, 2, …, 9 인 원소들이 각각 몇개인지 세준다. (각 그룹에 소속되는 원소들이 몇개인지 세준다.)
이렇게 하면, k번 째 자릿수가 0 인 그룹이 array 내에서 몇번째부터 몇번째까진지, 1 인 그룹은 몇번째부터 몇번째까진지, 9인 그룹은 몇번째부터 몇번째 까진지 전부 파악해줄 수 있다.
그리고 입력된 array의 index i = n-1 to 0 까지 (n은 전체 원소의 개수) 각 원소의 k번째 자릿수를 검사하며 (k번째 자릿수가 a라고 가정하면) a 그룹의 맨 마지막 자리에서 부터 숫자를 채워나가기 시작한다. 역으로 숫자를 채워나가는 이유는 이 편이 앞에서부터 숫자를 채워나가는 것보다 코드가 간결해지기 때문이다.
한편, 음수에 대해서 radix sort가 제대로 작동하기 위해서는 자릿수 그룹을 묶을 때 -9 부터 9 까지 총 19개 그룹에 대해서 위의 과정을 진행하면 된다.

• **동작 시간 분석**

각 정렬 알고리즘의 원소개수에 따른 수행시간을 비교해보았다. -1000000 ~ 1000000 의 정수형 난수에 대해서 동일한 원소개수에 대해서 100번씩 수행시간을 측정했다.

실제로 시간을 측정해보니 Bubble sort와 Insert sort는 원소개수에 따라 수행시간이 기하급수적으로 증가해서 원소 개수가 많을 때는 실험을 진행하기가 어려웠다. 따라서 일단 그 추세를 보기 위해서 최대 10000개 원소에 대해서 그래프를 그려보았다. 원소 개수는 한번에 100개씩 증가시켰다. 결과는 아래와 같다.



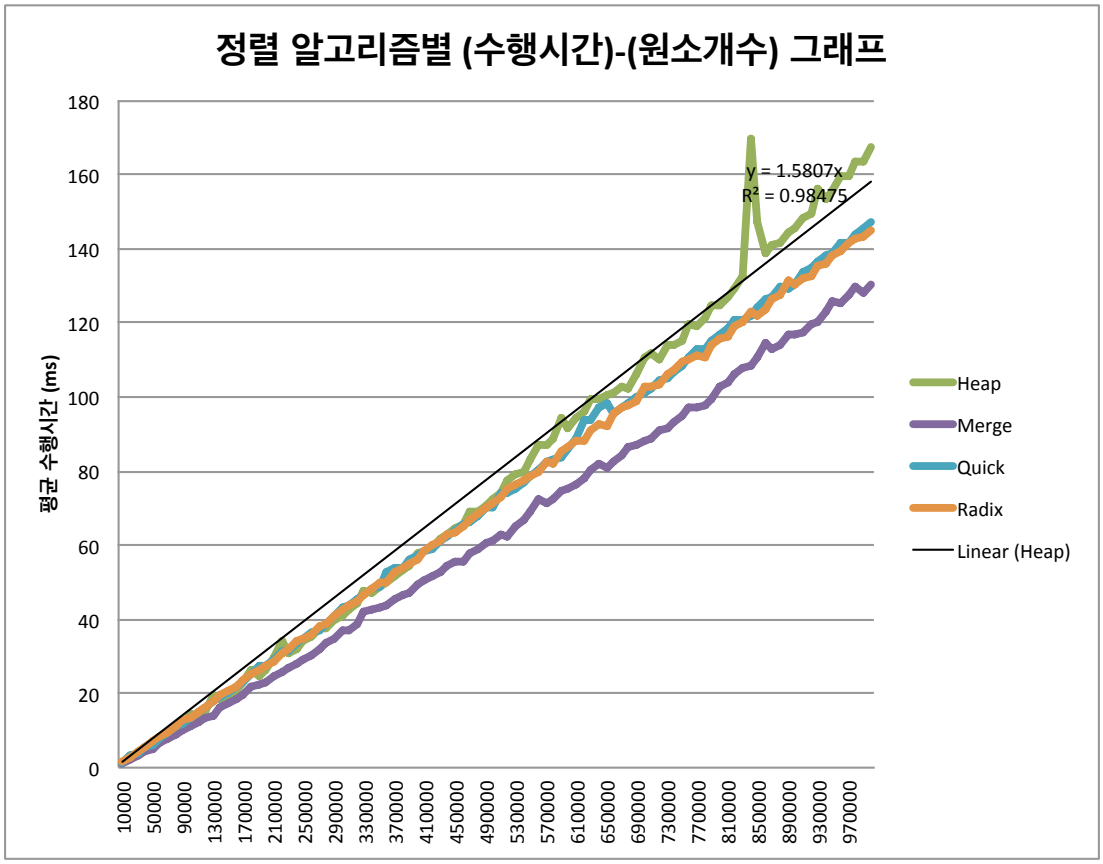
그래프에서 다른 정렬에 대해서는 분석을 일단 유보하자. Bubble sort의 수행시간이 압도적으로 크고, Insertion sort 역시 다른 정렬 알고리즘에 비해 수행시간이 매우 컸다. 10000개 데이터에 대해서 각 알고리즘의 수행시간을 표로 만들어보면 아래와 같다. (서로 다른 난수에 대해 100번 수행)

한편 Bubble sort와 Insertion sort 의 측정 그래프에 추세선을 그려보았다. (그래프 참조) 두 경우에 추세선이 높은 R제곱값으로, 2차 다항식 함수로 표현되는 것을 확인할 수 있었다. 이를 통해 Bubble sort와 Insertion sort가 $O(n^2)$ 의 수행시간을 가진다는 것을 확인 할 수 있었다.

정렬 알고리즘에 따른 수행시간 평균, 표준편차 (데이터 10000개)

	Bubble	Insertion	Heap	Merge	Quick	Radix
Average	178.3	30.5	1.13	1.39	1.37	1.64
Stdev	11.60129303	4.6292548	0.364828727	0.487749936	0.626976874	0.48

Bubble sort와 Insertion sort를 제외한 다른 알고리즘에 대해서 최대 1,000,000개 데이터에 대해서 수행시간을 측정했다. 역시 동일한 원소 개수에 대해서 100번씩 측정하여 평균을 내었으며, 원소 개수는 10000 개씩 증가시켰다. (다음 페이지 그래프)



일반적으로 Heap, Merge, Quick sort 의 수행시간은 O(n log n) 이고, Radix sort의 수행시간은 O(n) 인 것으로 알려져 있다. 그러나 1,000,000 개 데이터를 log 취해보면 6으로 매우 작다. 이로 인해서 여기서 Heap, Merge, Quick sort의 수행시간이 사실상 일차형에 가깝게 나타나는 것으로 보인다.

그러나 추세선을 그려보면 데이터 개수가 많아질수록 점점 데이터 곡선이 추세선을 상회하는 것을 확인 할 수 있다. 이는 점점 기울기가 증가하는 것을 의미한다. 추측하건데 n이 충분히 크면 n log n 함수의 형태를 띌 것으로 생각된다.

1,000,000개 데이터에 대해서 각 알고리즘의 수행시간을 표로 만들어보면 아래와 같다. (서로 다른 난수에 대해 100번 수행)

정렬 알고리즘에 따른 수행시간 평균, 표준편차 (데이터 1000000개)

	Heap	Merge	Quick	Radix
Average	167.36	130.47	146.96	144.71
Stdev	5.650699072	4.095009157	21.08787329	9.801321339

앞에서 언급한 대로 Heap, Merge, Quick sort 의 수행시간은 O(n log n) 이고, Radix sort의 수행시간은 O(n) 이므로 n이 충분히 큰 경우에 대해서는 Radix의 수행시간이 가장 짧아야 한다. 그러나 백만개의 데이터 개수가 수행시간 함수의 상수 계수를 무시할 수 있을 만큼의 큰 값이 아니므로, Radix sort 보다 오히려 Merge sort의 수행시간이 짧은 것으로 나타났다.

- Reference

J.Prichard & F.Carrano, **Data Abstraction & Problem Solving with JAVA**, 3rd Edition PEARSON, 2011.