

1. 정렬 알고리즘에 따른 동작 방식

Bubble Sort

index j 와 index $j+1$ 의 원소를 비교하여 j 가 더 크면 두 원소를 바꾼다. $i = \text{value.length}-1 \sim 1, j = 0 \sim i-1$ 로 수행해보면 index j 의 원소는 index $0 \sim j$ 까지의 원소 중에서 가장 큰 원소가 되므로 $j = i+1$ 까지 수행 시 index i 의 원소는 index $0 \sim i$ 까지의 원소 중에서 가장 큰 원소가 온다. 이를 i 를 1씩 줄여나가며 반복하면 정렬이 완료된다.

Insertion Sort

$i = 1 \sim \text{value.length}-1$ 까지 index i 의 원소를 index $0 \sim i$ 중 적절한 자리에 넣어 index $0 \sim i$ 까지 정렬이 되게 만든다. 이때 $i = 1$ 부터 시작하므로 index $0 \sim i-1$ 까지는 정렬이 되어있다. 따라서 index i 의 원소만 적절한 자리에 넣으면 index $0 \sim i$ 까지의 원소는 정렬된다. 이를 $i = \text{value.length}-1$ 까지 하면 정렬이 된다.

Heap Sort

Heap구조를 이용하여 정렬을 수행한다. Heap구조의 특징 중 하나는 root에 있는 원소(index 0의 원소)가 가장 큰 원소라는 것이기 때문에 index 0과 index $\text{value.length}-1$ 의 원소를 교환하고 index $0 \sim \text{value.length}-1$ 까지를 다시 heap구조로 만들어 위 행위를 반복하면 정렬이 완료된다.

Merge Sort

정렬된 두 배열을 정렬된 한 배열로 합치는 Merge함수를 이용해서 정렬한다. 배열을 절반으로 나누어 각각을 Merge Sort하고 그 둘을 Merge하는 재귀적인 형태로 알고리즘을 구현하였다. Merge Sort가 재귀적으로 호출되다 보면 $\text{length}=1$ 인 배열이 될 텐데 이는 정렬된 배열이므로 이를 이용해 여기서부터 Merge해나가면 정렬을 완료할 수 있다.

Quick Sort

주어진 배열의 마지막 원소를 pivot으로 하여 pivot보다 작거나 같은 원소들은 pivot 왼쪽에 pivot보다 큰 원소들은 pivot의 오른쪽에 배치하는 함수 partition을 이용하여 정렬한다. partition을 사용한 뒤 pivot 왼쪽의 배열과 pivot 오른쪽의 배열이 각각 정렬된다면 전체적으로 정렬이 완료된 것이므로 Quick Sort에서는 partition 후에 재귀적으로 왼쪽 배열과 오른쪽 배열을 Quick Sort한다. 이때 Quick Sort하는 배열의 크기가 2 이하라면 partition 후에 바로 정렬이 완료되고 Quick Sort를 재귀적으로 호출하다 보면 배열의 크기가 줄어들기 때문에 결국 크기가 2이하가 되어 정렬이 완료됨을 알 수 있다.

Radix Sort

기본적으로 Radix Sort는 0이상의 정수들의 배열에 대하여 이용할 수 있는 알고리즘이므로 주어진 배열에서 최솟값을 찾아 모든 원소에서 배열의 최솟값을 빼서 0이상의 정수들의 배열로 수정한다. 그 후 수정한 배열을 이용해 Radix Sort를 진행한 후 다시 최솟값을 더해서 원래 배열의 값으로 수정한다.

Radix Sort는 먼저 일의 자리 숫자를 기준으로 정렬을 하고 그 후 십의 자리, 그 후 백의 자리, 이렇게 최댓값의 자릿수까지 기준으로 정렬을 한다. 이때 이전 정렬의 순서는 유지하며 정렬하는 stable sort의 특성을 가지는 정렬 알고리즘을 구현해야 정렬이 된다.

2. 정렬 알고리즘에 따른 동작 시간 분석

| | Input | Bubble Sort | Insertion Sort | Heap Sort | Merge Sort | Quick Sort | Radix Sort |
|---|-----------------------------------|-------------|----------------|-----------|------------|------------|------------|
| 1 | r 10000 -50000 50000 | 45ms | 32ms | 2ms | 2ms | 2ms | 2ms |
| 2 | r 100000 -500000 500000 | 9300ms | 1556ms | 14ms | 37ms | 13ms | 19ms |
| 3 | r 1000000 -5000000 5000000 | 측정불가 | 측정불가 | 114ms | 138ms | 83ms | 79ms |
| 4 | r 100000 -500 500 | 9332ms | 1531ms | 13ms | 33ms | 23ms | 15ms |
| 5 | r 1000000 -500 500 | 측정불가 | 측정불가 | 113ms | 99ms | 402ms | 65ms |
| 6 | r 1000000 -500000000 500000000 | 측정불가 | 측정불가 | 113ms | 168ms | 86ms | 89ms |
| 7 | r 1000000 0 0 | 측정불가 | 4ms | 8ms | 54ms | 측정불가 | 37ms |

실험 1, 2, 3을 비교해 보면 데이터의 수가 늘어날수록 Bubble Sort, Insertion Sort가 다른 Sort들에 비해 훨씬 느려진다는 것을 알 수 있다. 일반적으로 Radix, Quick, Heap, Merge, Insertion, Bubble순으로 빠른 것을 볼 수 있다.

하지만 실험 2, 4, 5를 보면 **중복되는** 데이터의 수가 많아지면 Quick Sort가 매우 느려짐을 알 수 있다.

또한 실험 3, 5, 6을 보면 데이터의 최댓값과 최솟값의 차이의 **자릿수**가 커지면 Radix Sort가 느려짐을 알 수 있다.

마지막으로 실험 7을 보면 모든 데이터가 동일한 경우 즉 중복이 매우 많고 정렬도가 매우 높은 경우 Insertion Sort와 Heap Sort가 매우 빠름을 알 수 있다.

3. Search의 구현

위 2. 정렬 알고리즘에 따른 동작 시간 분석을 보면 일반적인 경우 Quick Sort가 가장 빠르나 중복되는 데이터가 많아지는 경우 Heap Sort나 Radix Sort가 더 빨라진다. 이때 데이터의 개수에 비해 데이터의 자릿수가 작으면 Radix Sort가 더 빠르고 반대의 경우 Heap Sort가 더 빨라진다.

4. Search의 동작시간 분석