기계항공공학부 2015-15284 문주혁

Swap은 많이 쓰는 함수기 때문에 별도의 메소드로 구현을 해 놓았다.

```
private static void swap(int [] value, int i, int j)
```

value array의 i번째 원소와, j번째 원소를 교환한다.

모든 경우에 n은 배열의 크기를 의미한다.

- [B] Bubble Sort

맨 앞에부터 인접한 두 원소를 비교하여 더 큰 원소가 뒤로 가도록 한다. 자연스럽게 가장 큰 원소가 맨 뒤로 가도록 되어있다. 가장 큰 원소가 맨 뒤로 가면, 그를 제외한 배열에서 같은 과정을 반복한다. 비교 회수는 $(n-1)+(n-2)\dots 2+1=\frac{n(n-1)}{2}$이다. 교환 회수는 이보다 적다 따라서 시간 복잡도는 항상 $\Theta(n^2)$이다.

- [I] Insertion Sort

배열의 앞쪽이 부분적으로 정렬되도록 만든다. 정렬된 부분이 전체가 된다면 정렬이 완료된다. 앞쪽의 k개의 원소가 정렬되어 있다고 한다면, k+1번째 원소를 적당한 위치에 끼워 넣어 k+1개의 원소가 정렬 된 것으로 만든다 비교 회수는 최악의 경우 $1+2+\cdots+(n-2)+(n-1)=\frac{n(n-1)}{2}$이다. 교환 회수는 이보다 적다. 따라서 최악의 경우 시간 복잡도는 $\Theta(n^2)$이다. 평균의 경우 이의 절반으로 마찬가지로 $\Theta(n^2)$이다. 최선의 경우는 비교를 n-1번만 하므로, $\Theta(n)$이다

- [H] Heap Sort

Root node에 최대의 원소가 오는 Heap을 만들고, root node의 원소를 뒤로 보내 정렬하는 방식이다. 이를 위해 밑의 함수를 메소드화 시켰다.

```
private static void percolateDown(int[] value, int i, int n)
```

percolateDown 함수는 value를 크기 n인 heap으로 보고, index i에 해당하는 원소를 적당한 곳으로 보낸다. Child node 중 큰 것이 자신보다 크다면, 그것과 자신을 교환하는 것을 반복한다. Child node가 모두 자신보다 작다면 끝난다.

첫번째, heap을 만들어야 한다. 배열을 complete binary tree로 만들고 밑에서부터 percolate down을 하면 heap이 될 것이다. 이 때 leaf노드의 경우 percolate down을 해줄 필요가 없다.

두번째, 최대원소, 즉 root node의 원소를 맨 뒤(이미 i번 뒤로 보냈다면 n-i번째)로 보내야한다. 이를 위해 맨 뒤의 원소와 root node를 교환하고, 더 이상 최대가 아니게 된 root node에 대해 percolate down을 해서 다시 heap성질을 만족시킨다. 시간 복잡도는 $\log n+\log(n-1)+\cdots+\log 2=n\log n$이다.

- [M] Merge Sort

주어진 배열을 반으로 나누어 각각을 Merge Sort로 recursive하게 정렬시킨다. 이 과정은 Merge Sort 할 배열의 크기가 1이 되면 끝난다. 그렇게 정렬된 두 배열을 작은 수부터 다른 배열에 담아 리턴한다. 그렇기에 in place sorting이 아니다. 정렬된 두 배열을 작은 수부터 다른 배열에 담아 리턴하는 과정은 Merge 함수에 구현되어 있다.

```
private static int[] Merge(int[] value1, int[] value2)
```

정렬된 두 배열 value1, value2를 받아 양 쪽에서 작은 수부터 차례대로 담아 리턴한다.

시간 복잡도는 worst case, average case 모두 nlogn이다.

- [Q] Quick Sort

배열의 한 원소(이 경우 맨 왼쪽)를 pivot으로 하여 왼쪽엔 더 작은 수, 오른 쪽엔 더 큰 수가 오도록 하는 과정을 모든 원소가 정렬될 때까지 반복한다. 이는 partition에 구현되어있다.

```
private static int partition(int[] value, int start, int end)
```

value배열의 start부터 end번째 원소까지를 start원소를 pivot으로 하여 pivot의 왼쪽엔 pivot보다 작은 원소, pivot의 오른쪽에는 pivot보다 큰 원소가 오도록 한다. 그 후 pivot의 index를 리턴한다.

시간 복잡도는 worst case의 경우 n^2, average case 모두 nlogn이다.

- [R] Radix Sort

1의 자리부터 최고자리수까지 stable sorting을 반복한다. Stable sorting으론 CountSort를 사용했다.

```
private static int[] CountSort(int[] value, int n, int k)
```

배열 value를 받아 n진법으로 표현했을 때 k번째 자리수를 기준으로 정렬하여 리턴하는 함수이다.

n은 2의 거듭제곱수인 16을 썼다.

k번째 자리수는 –(n-1)~n-1까지의 범위를 가질텐데(음수를 나누는 경우 고려), 각 경우의 수를 세어 원소를 새로운 배열에 stable하게 넣어 리턴한다.

시간 복잡도는 worst case의 경우 n^2, average case 모두 n이다.

Table 1 Comparison of Sorting Efficiency in Θ()

	Worst case	Average case
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Heapsort	nlogn	nlogn
Mergesort	nlogn	nlogn
Quicksort	n^2	nlogn

Radix sort	n	n

Number of records마다 5회 실행하여 running time의 평균과 표준편차를 기록하였다.

n개를 정렬한다고 하면 r n -2n 2n명령어로 -2n~2n 범위의 수를 정렬하였다.

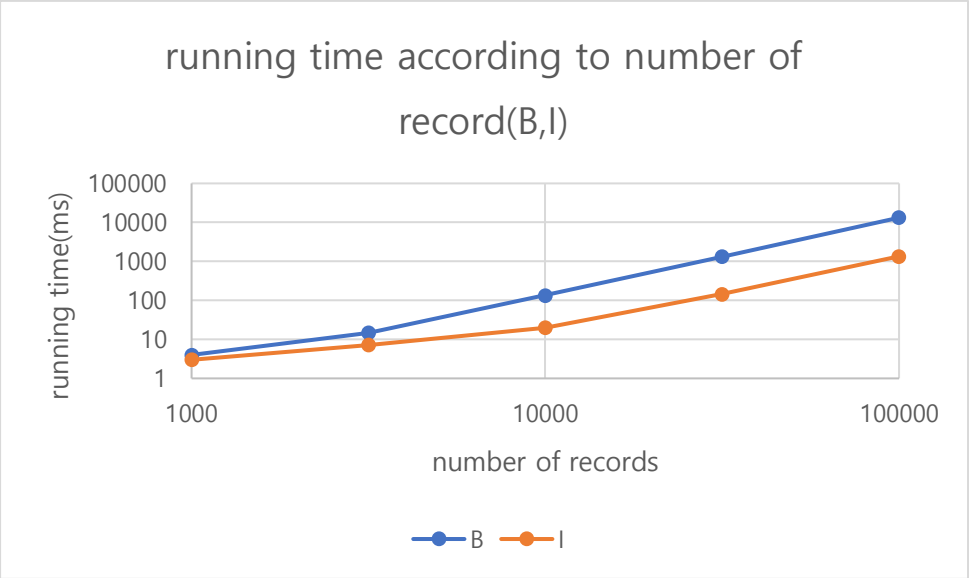
Table 2 mean(std) running time according to number of records

Number of records	running time(ms)					
	Bubble	Insertion	Heap	Merge	Quick	Radix
10^3	4.00*10^0 (0.00*10^0)	3.00*10^0 (7.07*10^-1)	0.20*10^0 (4.47*10^-1)	1.00*10^0 (0.00*10^0)	0.60*10^0 (5.48*10^0)	0.40*10^0 (5.48*10^0)
10^3.5	1.48*10^0 (4.47*10^0)	7.20*10^0 (1.30*10^0)	4.00*10^-1 (5.48*10^-1)	1.20*10^0 (4.47*10^-1)	6.00*10^0 (5.48*10^-1)	1.20*10^0 (4.47*10^-1)
10^4	1.34*10^2 (1.58*10^0)	2.00*10^1 (1.41*10^0)	1.40*10^0 (5.48*10^-1)	2.40*10^0 (5.48*10^-1)	1.60*10^0 (5.48*10^-1)	3.00*10^0 (0.00*10^0)
10^4.5	1.33*10^3 (4.88*10^0)	1.44*10^2 (3.51*10^0)	5.20*10^0 (4.47*10^0)	6.80*10^0 (4.47*10^-1)	5.40*10^0 (2.07*10^0)	7.00*10^0 (0.00*10^0)
10^5	1.32*10^4 (8.13*10^1)	1.35*10^3 (8.73*10^0)	1.26*10^1 (5.48*10^-1)	1.98*10^1 (8.37*10^-1)	1.34*10^1 (1.52*10^0)	1.60*10^1 (0.00*10^0)
10^5.5			3.74*10^1 (1.14*10^0)	5.04*10^1 (1.14*10^0)	2.92*10^1 (1.30*10^0)	5.54*10^1 (5.48*10^-1)
10^6			1.26*10^2 (2.39*10^0)	1.65*10^2 (1.22*10^0)	9.14*10^1 (3.44*10^0)	1.56*10^2 (1.79*10^0)
10^6.5			4.64*10^2 (9.32*10^0)	4.52*10^2 (1.07*10^1)	2.99*10^2 (6.73*10^0)	5.32*10^2 (5.52*10^0)
10^7			1.96*10^3 (1.30*10^1)	1.48*10^3 (1.81*10^1)	9.80*10^2 (9.76*10^0)	1.66*10^3 (3.91*10^1)
10^7.5			6.08*10^3 (3.39*10^3)	4.03*10^3 (2.25*10^3)	2.63*10^3 (1.46*10^3)	4.78*10^3 (2.67*10^3)
10^8			2.96*10^4 (3.31*10^2)	1.69*10^4 (2.84*10^2)	1.12*10^4 (8.01*10^1)	4.60*10^4 (6.67*10^4)

이론에 따르면 Bubble sort 와 Insertion sort의 경우 average case에 time complexity가 n^2이다.

실험에서 얻어진 데이터를 바탕으로 로그스케일 차트를 그려보니, 실제로 정렬할 수가 10배가 되면 걸리는 시간은 100배정도 되는 경향을 보였다.

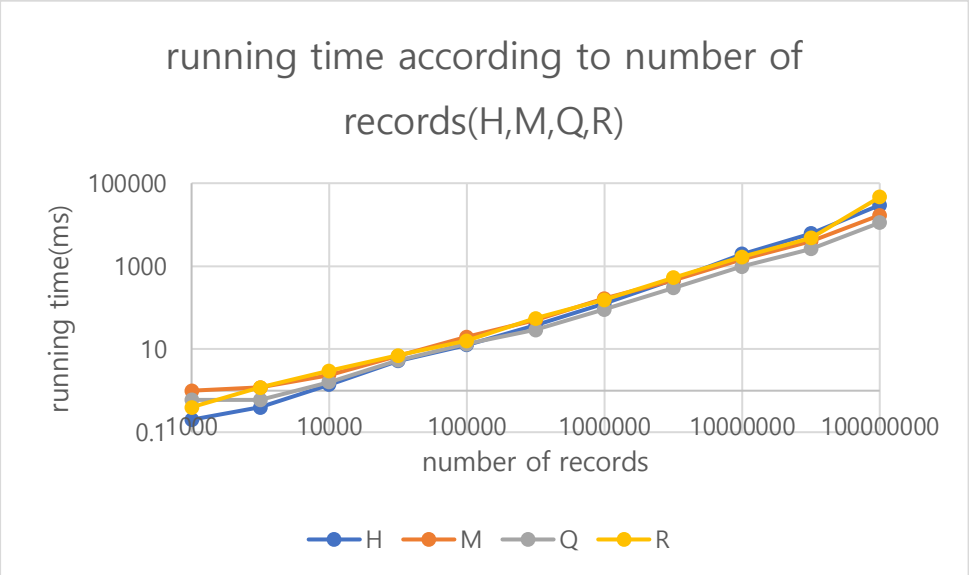
정확한 수치를 위해 n이 10000에서 100000의 범위에 있을 때(n이 너무 작으면 asymptotic 하지 않기 때문에), 가로축을 logn, 세로축을 log(time)으로 하여 선형회귀를 시켜보니 Bubble sort, Insertion sort의 경우 각각 기울기가 1.9941, 1.8289였다.



이론에 따르면 Heap sort, Merge Sort, Quick sort의 경우 average case에 time complexity가 nlogn 이다. Radix sort는 n이다.

실험에서 얻어진 데이터를 바탕으로 로그스케일 차트를 그려보니, 네 경우 모두 정렬할 수가 10 배가 되면 걸리는 시간도 10배정도가 되었다. 이는 time complexity가 n임을 의미하는데, logn의 경우 증가속도가 n에 비해 작아 nlogn이 n처럼 보이는 것이다.

정확한 수치를 위해 가로축을 logn, 세로축을 log(time)으로 하여 선형회귀를 시켜보니 네 경우 각각 기울기가 1.0661, 0.9471, 0.9392. 1.0087였다.



컴퓨터 성능의 한계로 더 큰 n에 대해선 실험해보지 못했으나, n이 충분히 커진다면 running time 이 이론값에 수렴할 것이다.