#define N 전체_원소_개수

#define d 전체_자릿수

#define T 실행_시간

#define s 표본표준편차

정렬 알고리즘의 동작 방식

Bubble sort. 전체 array부터 시작하여 (i, 0 ~ N-1), 관심 대상 array의 size를 뒤부터 하나씩 줄인다. 각각의 관심 대상에 대해서는 처음부터 끝까지 진행하며 (j, 0 ~ i- 1) 만약 큰 값이 작은 값보다 앞에 존재한다면 서로 순서를 바꾸었다.

Insertion sort. 두 번째 item부터 시작하여(i, 1 ~ N-1), 자신이 들어가야 할 자리를 찾았다. 자리를 찾는 알고리즘은 자신의 왼쪽부터 한 칸씩 이동하며(j, i-1 ~ 0) 자신보다 작거나 같은 원소 전에 멈추고 그 자리에 자신을 insert하는 방식이다. 이때, 이동하는 과정에서 마주치는 모든 원소들은 한 칸씩 right shift되었다.

Heap sort. 먼저 heap building을 했다. 전체 array를 maxheap으로 보고, (N/2) - 1번 원소(첫 번째로 자식이 존재하는 node)부터 시작하여, 0번 원소까지 percolate down했다.

이후 building된 maxheap에서 첫 번째 원소부터 차례로 관심의 대상 안에 존재하는 마지막 원소와 swap하고, 관심의 대상인 heap에 대하여 swap된 원소를 percolate down 해서 in-place sorting을 구현했다.

Percolate down은 정해진 parent node에 대하여, 관심의 대상 안에 존재하는 child node들 중에서 더 큰 child와 parent node를 비교하고, 만약 child node가 더 크다면 swap한 뒤 swap한 node에 대해 percolate down하는 방식으로 진행하였다(recursive call).

Merge sort. 입력된 array의 길이가 1보다 큰 경우에 대하여, array를 반으로 나누어 각각 mergesort한다(recursive call). 이후, return된 두 array를 적절히 merge하고, 그 결과를 return한다. 만약 길이가 1보다 작거나 같을 경우에는 입력된 array를 그대로 돌려준다(base case).

Merge는 두 array에 대하여, 각 array의 처음부터 끝까지 각 원소들을 비교하며 더 작은 원소부터 새로운 array에 원소를 copy하다 (동일할 경우 둘 모두 차례로 copy) 최종적으로 둘 중 한 array가 먼저 소모될 경우 나머지 array의 원소를 모두 새로운 array에 이어 붙였다(element-wise).

Quicksort. 우선 Pivot item을 주어진 시작과 끝의 중간 지점으로 잡고(거의 정렬된 array의 경우를 위하여), 전체를 비교하여 작은 item은 왼쪽으로, 동일한 item은 가운데로, 큰 item은 오른쪽으로 보내는 식으로 partition했다. 이후, 더 작은 집단과 더 큰 집단에 대하여, 각 집단의 크기가 1보다 작거나 같아질 때까지 각각 다시 partition하였다 (recursive call).

Radix sort. 음수 범위를 허용하므로, -9부터 +9까지 총 19개의 digit이 가능하다(나머지를 응용하였으므로). 이때, 각 자릿수에 대하여, 절대값이 가장 큰 원소의 최대 자릿수까지 차례대로 1의 자리부터 stable sort를 하였다. 이때, 실제로는 int의 |MIN_VALUE|가 |MAX_VALUE|보다 1 더 크므로, 모두 음수로 바꾸어 최소의 원소를 구하고, 마지막으로 그 원소를 양수로 만든 뒤 자릿수를 String으로 parsing하여 그 길이로 결정하였다.

Stable sort를 구현하기 위해서는 counting sort를 응용하였다. 이때, 마지막에 원본 array로 돌려놓는 과정에서는 순서를 보존하기 위해 뒤에서부터 차례로 temporary array에 원소들을 이동하였다가, 원본 array의 reference를 temporary array로 변경하였다.

동작 시간 분석

Settings. 현재 인턴을 진행하고 있는 연구실 서버에서 작업하였으며, 혼자 사용하는 것이 아니라 상황에 따라 시간이 변동하는 것을 확인할 수 있었다. 대신, 상대적으로 부하가 적은 시간에 시도하여 오차를 줄였다.

Testcases. 모든 상황에 대한 trial은 각 5회씩이었으며, 그로부터 평균과 표본표준편차를 계산하였다. 이때 random number는 $-(2^{32} - 1) \sim (2^{32} - 1)$ 사이에서 매 trial마다 무작위로 만들어 사용하였다. O (N^2) 정렬로 알려진 두 정렬, bubble sort와 insertion sort에 대해서

는 상대적으로 작은 n에 대해 test하였고, O (N log N) 정렬로 알려진 세 정렬, heap sort, merge sort, quick-sort는 비교적 큰 N에 대해서도 test하였다 (Table 1).

Radix sort는 독특한 편이므로, 자릿수를 int max로 고정하고 N의 변화에 대하여 test하고, 이후 충분히 큰 N(1×10^7)에 대해 자릿수가 1자리일 때부터 10자리일 때까지(int max) test하였다. 또한, quicksort와 자리수를 고정하고 N의 변화에 대해, N을 1×10^7으로 고정하고 자리수의 변화에 대해 각각 비교하였다. (Table 1)

Table 1. Tested N for Various Sorting Algorithms[a]

O[b]	N[c]									
N^2	1^2	2^2	3^2	4^2	5^2	6^2	7^2	8^2	9^2	10^2
N log N	1^3	2^3	3^3	4^3	5^3	6^3	7^3	8^3	9^3	10^3
dN	1^3	2^3	3^3	4^3	5^3	6^3	7^3	8^3	9^3	10^3

[a]Bubble, insertion, heap, merge, quick-, radix sort. [b]Big-O notation으로 나타낸 시간복잡도. [c]N × 10^{-3} (O(N^2)), N × 10^{-5} (O(N log N) && O(dN)).

O (N^2). Figure 1-2와 각각 같았다. 예상대로, insertion sort가 bubble sort에 비해 훨씬 빠르다는 사실을 확인할 수 있었다 (slope가 약 10배 작았으며, y-절편도 세 배 가량 차이가 났다). 또한, 두 sorting algorithm 모두 √T가 N에 비례하는 모습을 보였으므로, O (N^2) algoriThm임을 재확인할 수 있었다.

O (N log N). Figure 3-5와 각각 같았다. 세 algorithm 모두 N이 충분히 클 경우 (>10^7) T/N = A log(BN) 꼴의 추세선에 잘 들어맞는 모습을 보여, O (N log N) algorithm임을 재확인할 수 있었다. 이때, quicksort algorithm은 (이름대로) 셋 중에서 가장 빠른 모습을 보였으며 (A = 2.0 × 10^{-5}), 나머지 두 algorithm 중에는 heap sort가 merge sort보다 근소하게 빠른 결과를 보였다 (A = 1.2 × 10^{-4} vs 1.4 × 10^{-4}).

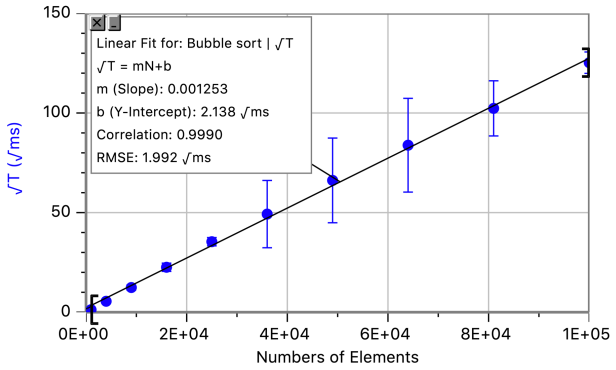


Figure 1. Bubble sort의 N에 따른 T. 가로축은 N, 세로축은 √T ($ms^{-1/2}$)이다. 추세선은 T = mN + b 꼴이며, error bar는 ±√s ($ms^{-1/2}$)을 나타낸 것이다.

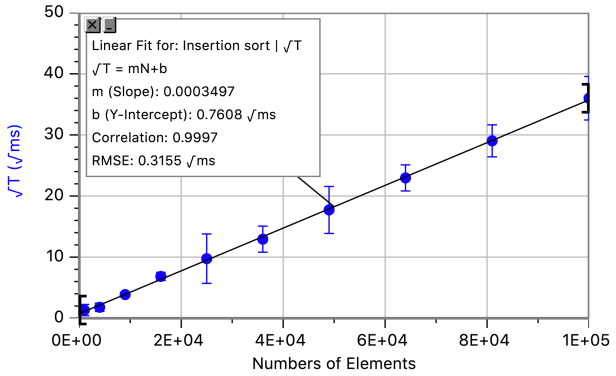


Figure 2. Insertion sort의 N에 따른 T. 가로축은 N, 세로축은 √T ($ms^{-1/2}$)이다. 추세선은 T = mN + b 꼴이며, error bar는 ±√s ($ms^{-1/2}$)을 나타낸 것이다.

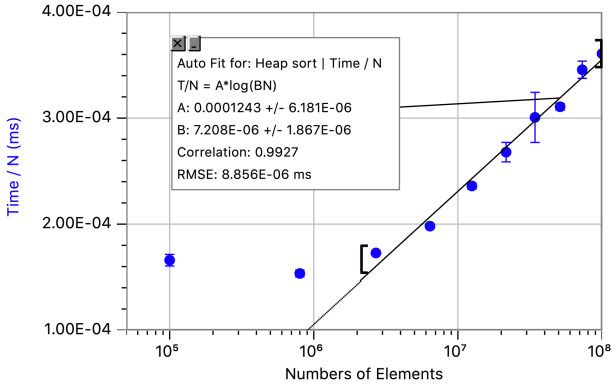


Figure 3. Heap sort의 N에 따른 T. 가로축은 N(로그축), 세로축은 T/N (ms)이다. 추세선은 T = a + log(bN) 꼴이며, error bar는 ±s/N (ms)을 나타낸 것이다.

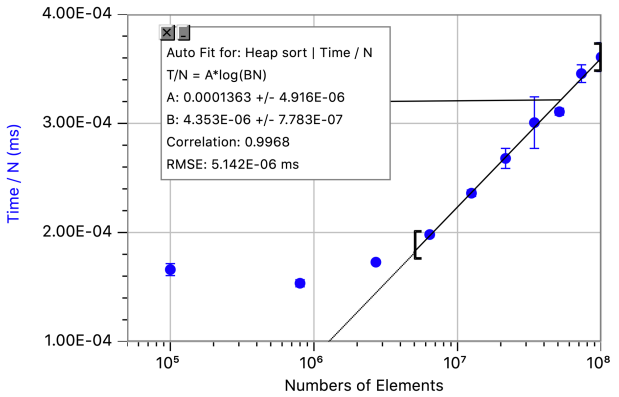


Figure 4. Merge sort의 N에 따른 T. 가로축은 N(로그축), 세로축은 T/N (ms)이다. 추세선은 T = a + log(bN) 꼴이며, error bar는 ±s/N (ms)을 나타낸 것이다.

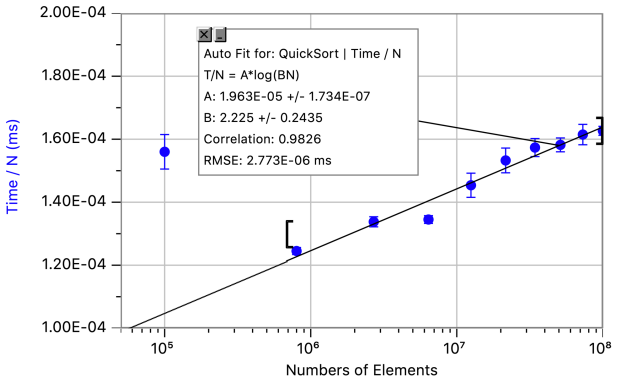


Figure 5. Quicksort의 N에 따른 T. 가로축은 N(로그축), 세로축은 T/N (ms)이다. 추세선은 T = a + log(bN) 꼴이며, error bar는 ±s/N (ms)을 나타낸 것이다.

Radix Sort. 유일하게 non-comparison sorting algorithm이므로, 크게 두 가지로 나누어 살펴보았다. 첫 번째는 다른 algorithm과 마찬가지로 N에 따른 T의 변화를 살펴보았는데, 예상대로 T = mN + b 꼴의 선형 추세선에 잘 들어맞는 결과를 나타내었다 (Figure 6).

또한, radix sort는 자릿수에도 선형으로 의존하는 것으로 알려져 있으므로, N을 10^7으로 고정하고 d를 변화시키면서 T의 변화를 측정하였다 (Figure 7). 그 결과, N과 유사하게 T = md + b 꼴의 선형 추세선을 잘 만족하는 그래프를 얻을 수 있었다. 반면, quicksort의 경우에는 자릿수가 증가함에 따라 같이 T가 증가하다 일정 수준에서 saturation이 일어났는데 (Figure 8), 이는 자릿수가 적을 경우 중복되는 element가 많아 나타난 현상으로 보인다.

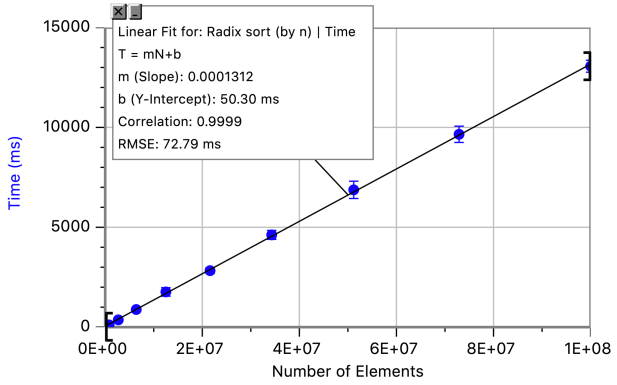


Figure 6. Radix sort의 N에 따른 T. 가로축은 N, 세로축은 T (ms)이다. 추세선은 T = mN + b꼴이며, error bar는 ±s (ms)을 나타낸 것이다.

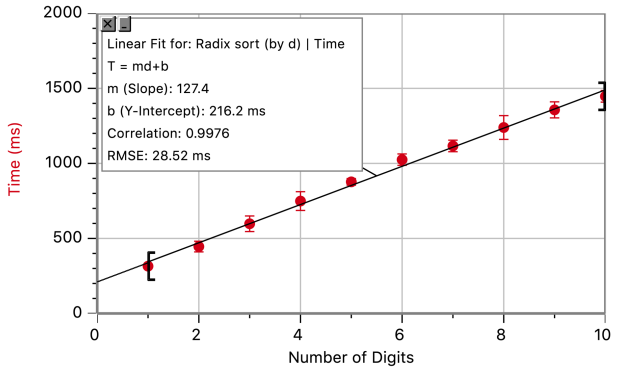


Figure 7. Radix sort의 d에 따른 T. 가로축은 d, 세로축은 T (ms)이다. 추세선은 T = md + b꼴이며, error bar는 ±s (ms)을 나타낸 것이다.

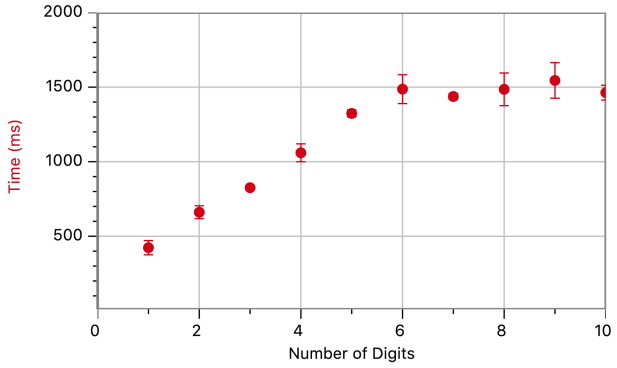
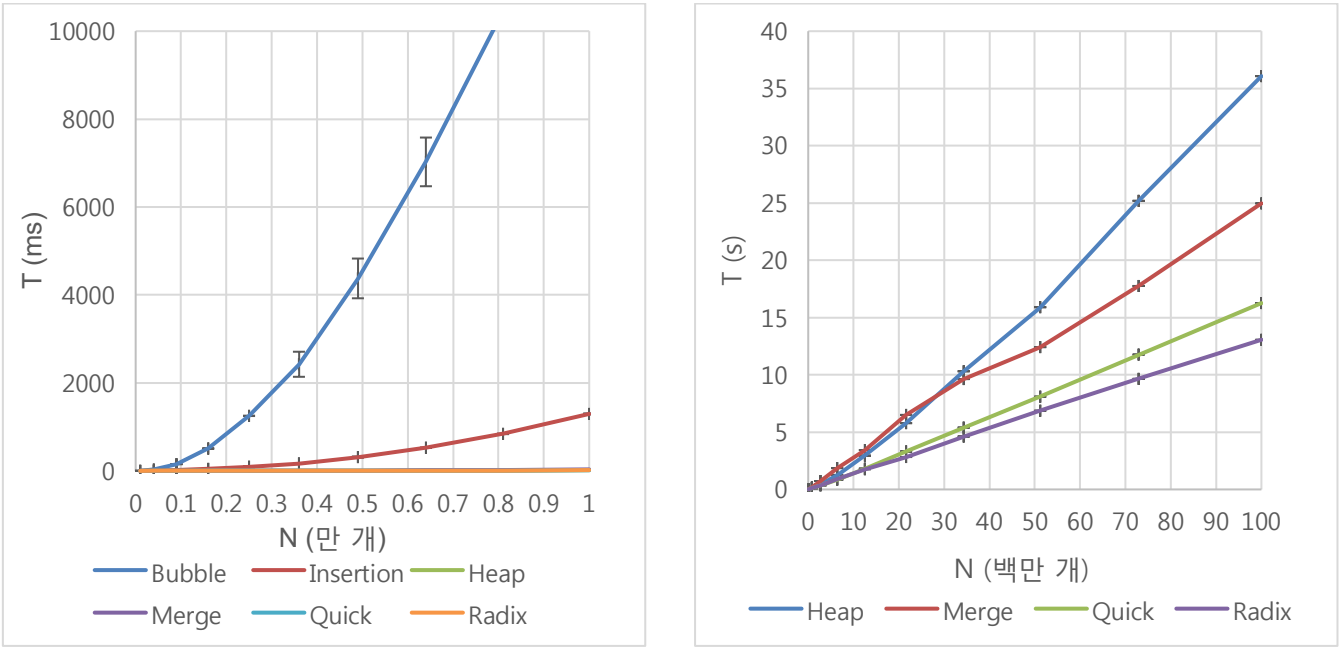


Figure 8. Quicksort의 d에 따른 T. 가로축은 d, 세로축은 T (ms)이다. Error bar는 ±s (ms)을 나타낸 것이다.

All algorithms together. 마지막으로, 모든 알고리즘을 비교하여 그래프로 나타내었다 (Figure 9). 전체적인 비교를 위해 모두 log축으로 그린 그래프도 추가하였다 (Figure 10).



(a) **(b)**

Figure 9. N = **(a)** 0 ~ 1×10^5의 모든 알고리즘에 대한 비교 그래프. **(b)** 1×10^5 ~ 1×10^8의 O(N log N) 또는 O(dN) 알고리즘에 대한 비교 그래프. 추가된 O(N log N) 또는 O(dN) 알고리즘에 대한 데이터는 50회씩 5번 반복하여 각각 평균 낸 결과이다. 오차 막대는 ±s를 나타낸다.

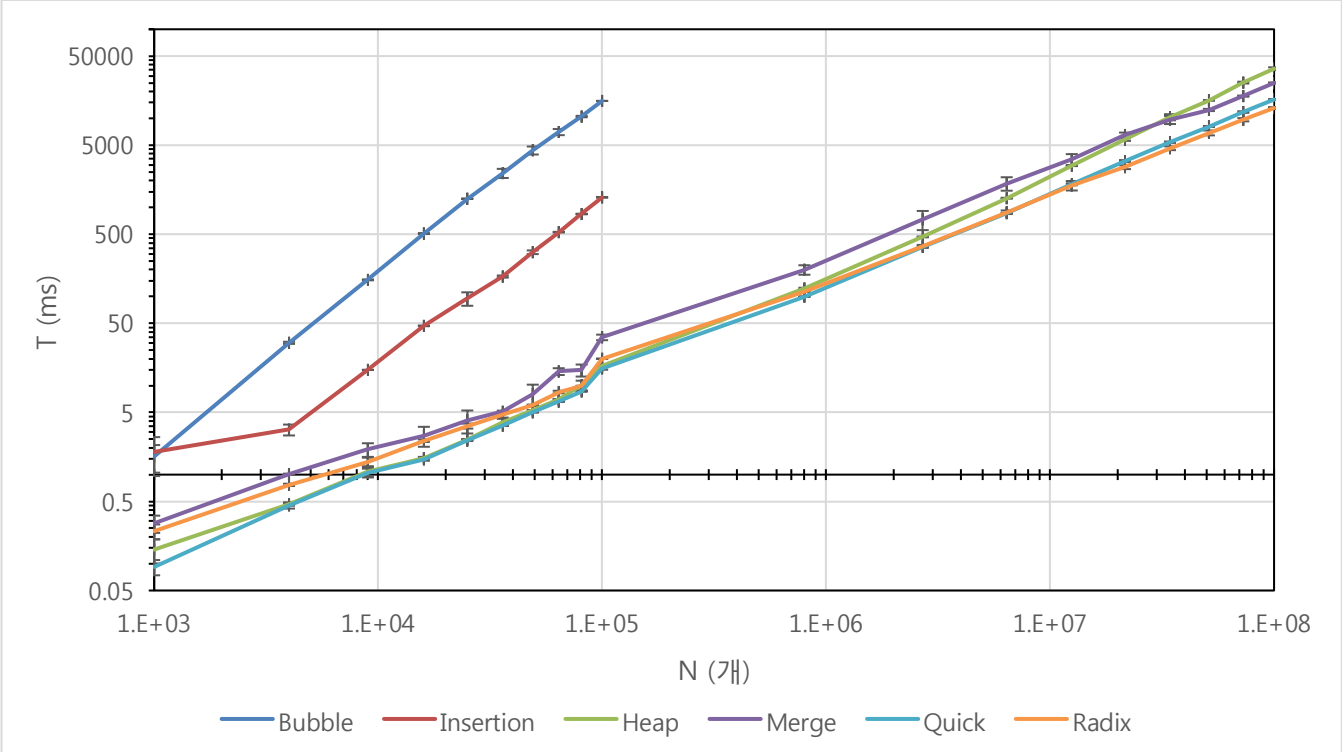


Figure 10. 모두 log축으로 그린 N-T 그래프. 오차 막대는 ±s를 표시한다. Radix sort가 초반에는 heap sort나 quick sort보다 느리지만, 나중에는 더 빨라지는 것을 확인해볼 수 있다. 본문에 없지만 여기에는 추가된 O(N log N) 또는 O(dN) 알고리즘에 대한 데이터는 50회씩 5번 반복하여 각각 평균 낸 결과이다.