

자료구조 HW4 Report

Mechanical Engineering

2019-15838 주기영

1. 정렬 알고리즘 설명

1.1 Bubble sort(stable, in-place)

가장 기초적인 정렬 알고리즘으로 항상 $\theta(n^2)$ 의 시간복잡도를 갖는 정렬이다. 배열에서 가장 큰 원소를 맨 뒤쪽으로 보낸 후에 나머지 배열에서 다시 가장 큰 원소를 맨 뒤쪽으로 보내는 것을 반복하여 정렬을 진행한다. n 개의 원소를 갖는 배열이 주어질 때, 첫 원소부터 끝까지 인접 원소와 비교하여 크다면 두 원소의 위치를 바꿔서 배열에서 가장 큰 원소를 n 번째에 위치시킨 후에 $n-1$ 개의 원소를 갖는 배열로 다시 이 과정을 반복하는 방법을 사용했다.

1.2 Insertion sort(stable, in-place)

기초적인 정렬 알고리즘에 속하여 최악의 경우 Bubble sort와 마찬가지로 $\theta(n^2)$ 의 시간복잡도를 갖는다. 배열이 주어질 때 Insertion sort에서 두 번째 원소부터 시작하여 자신의 자리를 찾아가는 과정을 시작한다. 앞에서부터 n 개의 원소가 정렬되었을 때, $n+1$ 번째 원소는 자신보다 큰 원소를 만나면 순차적으로 위치를 바꾸면서 자신보다 작거나 같은 원소를 만났을 때, 자신의 자리에 정착하여 $n+1$ 개의 정렬된 원소쌍을 만든다.

이 과정을 반복하여 정렬을 진행하는 알고리즘이다. 하지만 최선의 경우 Insertion sort $\theta(n)$ 의 시간복잡도를 갖는다. 이는 모든 원소가 평균적으로 상수만큼의 시간으로 자신의 자리를 찾아가 수 있는 거의 정렬된 배열에 대해서 성립한다. 즉, 일반적으로 비효율적인 정렬 알고리즘에 속하나 Data의 특성에 따라서 매우 효율적인 정렬 알고리즘이 될 수 있고, 유용하게 사용될 수 있다.

1.3 Heap sort(stable, in-place)

Heap sort는 Heap이라고 불리는 우선순위 큐 자료구조의 성질을 이용한 정렬 알고리즘이다. 모든 배열은 Heap으로 구성할 수 있다. Heap class를 두어 n 개의 원소를 갖는 배열이 들어왔을 때 "buildMaxHeap" 메소드를 이용하여 배열을 MaxHeap으로 구성하고, 가장 높은 원소를 배열의 마지막 원소와 위치를 바꾼 후에 "percolateDown"를 이용하여 다시 heap으로 구성한다. $n-1$ 개의 남은 원소를 이용하여 이 과정

을 반복하여 최종적으로 Heap에 하나의 원소가 남을 때까지 진행하면 정렬이 이루어진다. Heap sort에서 가장 중요한 메소드는 "percolateDown"으로, Heap build와 sort과정에서 사용된다. 이는 heap의 더 작은 값의 원소가 큰 값의 부모일 때 원래 자신의 자리를 찾아가는 메소드로, 자신보다 큰 자식이 없을 때까지 재귀적으로 두 자식 중 큰 값을 갖는 자식과 위치를 바꾸는 메소드이다.

Heap sort는 $O(n \log n)$ 의 시간복잡도를 가지며, 좋은 입력이 들어와 sort를 진행하는 동안 모든 원소에 대한 평균적인 "percolateDown"의 사용이 상수라면 $\theta(n)$ 의 시간복잡도를 갖는 이론적으로 좋은 정렬 알고리즘이다. 또한 추가적인 공간을 갖지 않는 in-place 정렬이라는 사실 또한 Heap sort가 갖는 이점 중 하나이다.

1.4 Merge sort(unstable, not in-place)

Merge sort는 대표적인 분할 정복을 이용하는 정렬 알고리즘이다. Merge sort는 배열 집합을 두 개로 분리한 후 하나로 다시 합치는 과정에서 두 개의 배열에서 원소를 차례대로 꺼내어 하나의 정렬된 배열 집합을 얻는다. 하나의 집합을 쪼개어 두 집합을 얻으면 두 집합은 각각 크기가 다르고 구조가 같은 상황에 직면하므로, 재귀적으로 구성할 수 있다. Bubble sort와 다르게 이 때, n 개의 배열에 대해서 함수가 $\log n$ 번 호출되므로, stack overflow가 쉽게 발생하지 않는다. 즉, 배열을 최소단위로 모두 분리한 후 차례대로, 합쳐 원래 배열의 크기를 갖는 정렬된 배열을 얻을 수 있다. Merge sort는 어떤 배열을 정렬하더라도 항상 시간복잡도가 $\theta(n \log n)$ 를 갖는 특징이 있어 이론적으로 좋은 정렬 알고리즘이다. 하지만 Merge sort의 단점은 in-place로 구현할 수 없다는 점이다. 두 개의 집합을 순서대로 합치는 과정에서 다른 배열의 도움이 필요하고, 따라서 정렬을 진행할 때 추가적인 공간의 사용이 발생한다.

이에 더하여 Merge sort의 배열을 합치는 과정에서 다른 배열에 합친 후 다시 원래 배열에 정보를 덮어쓰는 과정이 있는데, 이 과정이 모이면 시간복잡도가 증가한다. 이러한 방법을 해결하기 위해 향상된 Merge sort가 고안되었고, 이 방법을 사용하여 시간복잡

도를 감소시켰다. 이는 Merge sort 과정에서 두 개의 배열을 번갈아가면서 사용하여 배열의 정보를 덮어쓰는 과정이 필요없어지므로, sort에 걸리는 시간이 단축된다. 실제로 두 가지 방법을 사용하여 시간을 측정해본 결과 20%가량의 시간 단축을 확인할 수 있었다.

1.5 Quick sort(unstable, in-place)

Quick sort는 대표적인 분할정복을 이용하는 정렬 알고리즘이다. "partition" 메소드는 Quick sort에서 가장 중요한 메소드로, 배열이 주어졌을 때 분할의 기준이 되는 index를 반환하는 메소드이다. 주어진 배열의 가장 마지막 원소를 pivot value로 설정하고, 이보다 작은 원소는 모두 앞에서부터 채우면 pivot value의 정확한 자리를 찾으면서 배열이 두 개의 section으로 분할된다. 이 과정을 pivot의 위치의 왼쪽과 오른쪽 부분 배열에 대해서 수행하도록 재귀적으로 알고리즘을 구현하였다.

하지만 같은 분할정복 알고리즘인 Merge sort와 달리 Quick sort는 항상 $\theta(n \log n)$ 의 시간복잡도를 보장하지 못한다. 이는 Merge sort는 항상 배열을 균등하게 나눌 수 있는 반면에 Quick sort는 pivot value의 대소에 따라 좌우될 수 있기 때문이다. 따라서 중복된 값이 많은 배열이나 거의 정렬이 이루어진 배열에 대해서는 Quick sort의 최악의 시간복잡도인 $\theta(n^2)$ 으로 작동한다. 또한, 이러한 경우 Quick sort에서 recursion 사용에 따른 stack overflow가 발생하는 것을 확인할 수 있다. 하지만 random하며 겹치는 원소가 지배적이지 않은 배열에서는 Quick sort가 Merge sort, Heap sort에 비해 더 빠르게 정렬을 수행한다. 따라서 일반적인 random 배열의 경우 Quick sort를 사용하는 것이 가장 좋으나 Data의 특성을 꼭 고려하는 것이 필요하다.

1.6 Radix sort(stable, not in-place)

Radix sort는 $O(kn)$ 의 시간복잡도를 갖는 특수 정렬 알고리즘이다. k는 원소의 최대 자리수에 따라 달라지는 상수로, 원소들이 모두 k개 이하의 자릿수를 가진 자연수로 이루어진 배열과 같은 특수한 배열을 대상으로 수행할 수 있다. Radix sort는 일의 자리수부터 모든 자릿수에서 정렬을 진행한다. 중요한 것은 앞선 자리수에서의 정렬 순서를 유지하기 위해서 앞선 자리수에서 정렬된 순서로 계속하여 정렬이 이루어져야 한다. 따라서 Radix sort는 stable sort이다. Radix sort에서 음의 정수도 정렬을 진행하기 위해서 양의 정수 집합과 음의 정수 집합을 분리하여 각각

Radix sort를 진행하고, 다시 원래의 집합에 덮어쓰는 방식으로 알고리즘을 구현하였다.

1.7 stable/unstable, in-place

stable한 정렬은 원래의 배열에서 중복된 원소가 이루고 있는 순서가 정렬 후에도 계속해서 같은 순서를 이루는 정렬을 의미한다. in-place 정렬은 별도의 추가 저장공간 없이 자신의 배열 내부에서 정렬을 할 수 있는 것을 의미한다. Q, M은 대표적인 unstable 정렬, M, R은 대표적인 not in-place 정렬이다.

2. 정렬 데이터 개수와 정렬 시간의 관계

2.1 기본 알고리즘과 고급알고리즘

Fig 1, Fig 2는 각각 데이터 개수에 따른 버블정렬과 병합정렬의 정렬 시간의 관계를 나타낸 그래프이다. 버블정렬은 항상 $\theta(n^2)$, 병합정렬은 항상 $\theta(n \log n)$ 이므로, 이 두가지를 비교해보았고, 입력 데이터는 1부터 n까지 완전 역순의 배열을 이용하였다. 각각의 그래프의 개형을 확인하면 버블정렬은 $y = c_1 x^2$, 병합정렬은 $y = c_2 x \log x$ 의 개형과 상당히 유사함을 관찰할 수 있다.

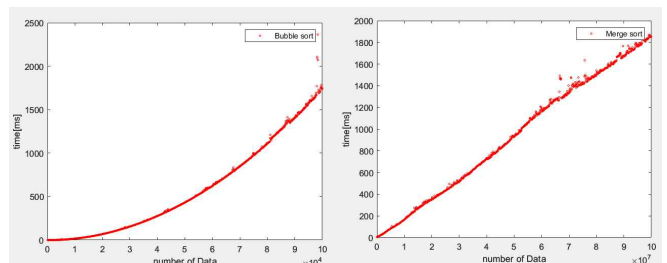


Fig 1. Bubble sort

Fig 2. Merge sort

2.2 정렬 종류에 따른 정렬 시간 비교

각 정렬을 사용함에 따라 정렬에 걸린 시간을 객관적으로 비교하기 위해서 같은 배열을 사용하여 실험을 진행하였고, 또한 충분히 random하면서 같은 수를 포함하지 않는 배열을 사용하였다. random성은 search 메소드를 구현할 때 사용하는 정렬쌍 비율을 이용하여 이 값이 50% 이하인 데이터만을 사용하였고, 10번 반복한 후 평균값을 이용하여 신뢰성을 높였다. Radix sort의 경우에는 최대자리수에 영향을 받는 sort이므로 data가 달라질 때와 최대자리수가 달라질 때의 정렬시간을 각각 조사하였다. Table 1은 10의 거듭제곱의 data 개수에 따른 각 정렬의 정렬 시간을 나타낸 것이다. 충분히 random한 배열에 대해서 항상 Quick sort의 성능이 가장 좋음을 확인할 수 있다. $O(n \log n)$ 의 시간복잡도를 갖는 M과 H는 Q

보다는 시간이 많이 걸리지만 I, B보다 적게 걸리는 것으로 관찰되었고, B는 가장 시간이 많이 걸리는 것으로 관찰되었다. Table 2는 Radix sort의 정렬시간을 나타낸 것으로 data 개수가 달라질 때 최대자리수는 5자리로 고정하였고, 자리수가 달라질 때 데이터 개수는 10^7 개로 고정하고 조사하였다.

data	Q	M	H	I	B
10^3	0.3	0.4	0.8	1.8	3.5
10^4	1.9	2.1	4.3	15.3	40.2
10^5	9.7	10.4	14.3	561.5	8725.1
10^6	63.6	83.4	107.7	55477	N/A
10^7	632.4	917.7	1584.5	N/A	N/A
10^8	7324	10437	33657	N/A	N/A

Table 1. Sorting time of each sorting(단위: ms)

data	10^3	10^4	10^5	10^6	10^7	10^8
R	0.5	4.3	27.4	215	2067	20612
digit	4	5	6	7	8	9
R	1425	2068	2887	3778	4743	6708

Table 2. Sorting time of radix sort(단위: ms)

2.3 sortedPair, redundancy 메소드

sortedPair는 모든 인접 pair의 개수에 대한 정렬이 되어 있는 pair의 개수를 백분율로 나타내는 메소드로 배열의 정렬이 얼마나 이루어져 있는지 판단하는 척도이고, redundancy는 해쉬테이블에 배열의 모든 원소를 삽입한 뒤 원래 배열의 크기에 대한 해쉬 테이블의 크기를 나타내는 메소드로 배열의 중복된 원소를 얼마나 포함하는지 판단하는 척도이다. 즉, redundancy 값이 낮을수록 중복도가 높은 것이다. 이 두 메소드는 Search 메소드를 구현할 때 사용된다.

3. Search method 구현

3.1 최대자리수

Radixsort의 시간복잡도는 $O(kn)$ 로 배열의 최대값의 자리수 k에 비례한다. 따라서 배열의 k값이 작으면 Q보다 좋은 성능을 낼 수 있다. k가 1인 경우에는 배열 크기 5만 이하, k가 2 이상일 때에는 특정한 배열 크기 이하에서 R이 M/Q보다 좋음을 확인할 수 있고, 이를 Table 3에 나타냈다. 각각 최대자리수가 고정되어 있는 배열에 대해서 Fig 3는 배열의 크기에 따른 R, M, Q의 정렬시간을 나타낸 것이다. k=1에서는 M, Q에 비해 더 빠른 것을. k=2에서는 일부에서만 M, Q에 비해 빠른 것을 확인할 수 있다.

k	2	3	4	5	6	7	8	9
n	3300	1800	1200	900	500	400	300	200

Table 3. Hyper parameter(n 이하일 때)

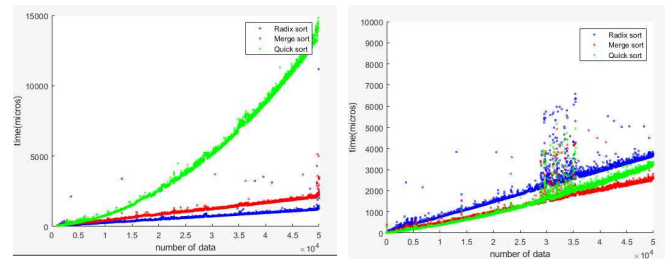


Fig 3. data 개수에 대한 정렬 시간(왼쪽:k=1, 오른쪽:k=2)

3.2 redundancy

Fig 2에서 redundancy 값이 0.5% 이하일 때 Q의 성능이 현저하게 떨어지는데 이는 같은 원소를 모두 pivot의 오른쪽으로 보내는 “partition” 메소드에서 결과적으로 pivot이 배열의 중간이 아닌 가장자리에 쏠리게 되면서 Q의 성능이 안좋아지는 것이다. 이를 해결하기 위해서 Q 대신 M을 사용하였고, 기준이 되는 하이퍼 파라미터를 5천, 만, 3만, 5만을 갖는 배열의 크기에서 각각 구하였다. 배열의 크기와 상관없이 모두 0.5% 근처에서 Q의 정렬시간이 급격하게 증가하여 M이 더 좋음을 Fig 4에서 확인할 수 있다.

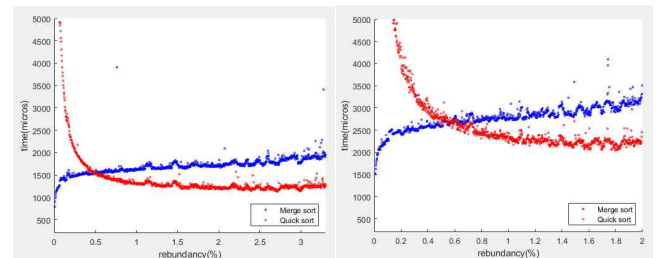


Fig 4. Q와 H의 비교(왼쪽: 3만, 오른쪽: 5만)

3.3 sortedPair

sortedPair값이 높을 때 Q보다 빠른건 M과 I이다. sortedPair값이 90% 이상일 때, I와 M을 비교해보았다. 배열의 크기에 따라서 두 개의 정렬 시간이 같아지는 것을 Table 4에 나타냈다. 3천 이하에서는 90을 HP로 사용하였고, 3천 이하 만 이상, 만 이상 5만 이하에서는 각기 다른 배열 크기 N에 대한 일차함수를 HP로 사용하였다. Fig 5를 보면 sortedPair 값이 약 66% 이상에서 M이 Q보다 정렬시간이 짧은 것을 관찰할 수 있다. 이는 정렬의 크기에 크게 좌우되지 않기에 66.66%를 Q와 M의 HP로 결정하였다. 역순의 배열에서도 Q는 안 좋은 성능을 보이기에 sortedPair 값이 낮을 때에도 Q와 M의 HP가 존재한다. 이 또한 정렬의 크기에 크게 좌우되지 않는 것을 여러 실험을 통하여 확인할 수 있었고, 15%로 정하였다.

	5만	4만	3만	2만	1만	7천	5천	3천
HP(%)	99.7	99.6	99.4	99.2	99.0	97	94	90

Table 4. Hyper parameter(between I, H)

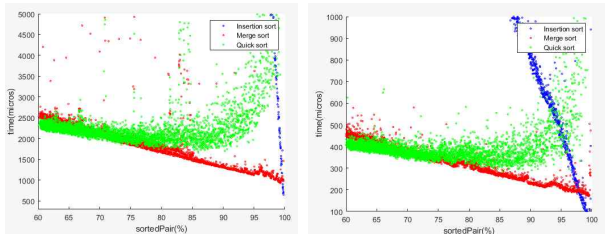


Fig 5. High SortedPair(I, Q, M), 왼쪽 5만 오른쪽 만

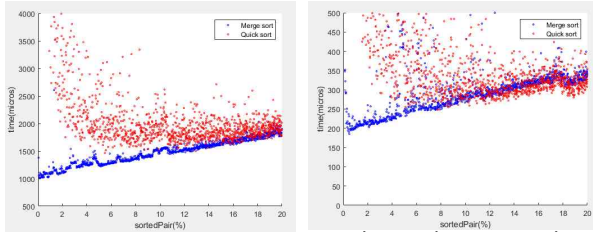


Fig 6. Low SortedPair(Q, M), 왼쪽 5만 오른쪽 만

3.5 Search method

I의 경계 HP가 배열의 크기에 의존하는 이유는 항상 $\theta(n \log n)$ 의 시간복잡도를 갖는 병합정렬과의 비교에서 정렬되지 않는 쌍의 개수를 p 라고 하면 $\theta(pn)$ 에 준하고, 따라서 이를 비교할 때 정렬시간이 같아지는 p 는 $\log n$ 의 함수가 되므로, sortedPair 함수에 이를 반영하면 정렬시간이 같아지는 sortedPair 값이 n 에 대한 함수로 나타나는 것이다. 최대자리수에 관한 HP 또한 이와 같은 이유로 N 에 대한 함수로 나타난다. 다른 HP값은 N 에 큰 영향을 받지 않는 것으로 관찰됐다. 적절한 배열을 만들어 Search 함수가 제대로 동작하는지 관찰해보자.

Table 5는 최대자리 검사에 관하여 Search 함수 동작 시간, Q, M, R 각각의 시간을 나타낸 것이다. N 이 5만, 1만, 5천일 때는 $k=1, 2$ 만을 비교해보았는데 실제로 $k=1$ 일 때는 R이 더 빠르고, $k=2$ 일 때 M, Q가 빠른 것을 관찰할 수 있다. N 이 5만일 때 $k=2$ 에서 S가 M을 반환하는 것은 중복도 계산에 의한 것이다. 5000, 3000일 때에도 각각 $k=3, k=4$ 가 적절한 경계라는 것을 실제로 확인할 수 있다. Table 6는 중복도 검사에 관하여 Search 함수 동작 시간, Q, M 각각의 시간을 나타낸 것이다. 5만, 5천, 3천, 1천에서 redundancy의 HP가 모두 0.5 부근에서 형성되는 것을 알 수 있다. Table 7는 정렬쌍 검사에 관하여 S search 함수 동작 시간, Q, M, I 각각의 시간을 나타낸 것이다. 적절한 sortedPair 값에서 Search가 원하는 정렬을 반환하는 것을 관찰할 수 있다. 또한 sortedPair가 50 근처에서는 충분히 random한 배열이라고 생각할 수 있는데 이 때 Search는 가장 적절한 Q를 반환한다. Search 함수는 최대자리 검사, 중복도 검사, 정렬쌍 검사를 순서대로 진행하는데 이때 중복

도 검사, 정렬쌍 검사, 최대자리 검사 순으로 많은 시간이 걸린다. 충분히 Random한 배열에서 Search가 Q를 반환하는 것에는 Q의 정렬시간과 많거나 비슷한 것을 관찰할 수 있다. 따라서 실제로 Search method는 배열이 충분히 random하다는 것을 알 때 Q를 이용하는 것보다 매우 비효율적일 수 있으며, 이보다는 배열의 특성이 미지인 경우에 재귀에 의한 Stack overflow를 막거나 반환값에 의해서 배열의 특성을 유추해볼 수 있으므로 이러한 경우에 유용하게 사용될 것이라고 생각한다. 더욱이 배열에 대해서 적당한 정보가 제공될 때 Search함수의 boolean 파라미터를 추가하여 필요한 검사만을 진행하는 형태로 Search의 동작시간을 단축시킬 수 있을 것이고, 이는 매우 유용한 방법이라고 생각한다.

N, k	S [μs]	Q [μs]	M [μs]	R [μs]
5만, 1	1109, R	6655	6185	5247
5만, 2	4377, M	5657	5318	8267
1만, 1	263, R	2721	2361	1299
1만, 2	1315, Q	1405	1517	2452
5천, 1	178, R	1884	1784	852
5천, 2	812, Q	947	956	1205
3천, 1	123, R	897	788	411
3천, 2	124, R	797	926	726
3천, 3	815, Q	733	883	1056
1천, 1	70, R	700	727	151
1천, 4	71, R	576	565	476
1천, 5	494, Q	596	646	686

Table 5. Search, Q, M, R 동작시간(최대자리 검사)

N, redundancy	S [μs]	Q [μs]	M [μs]
5만, 0.40	5759, M	5677	5183
5만, 0.60	3208, Q	5350	5602
1만, 0.41	1399, M	1600	1375
1만, 0.61	1441, Q	1336	1353
5천, 0.42	506, M	1014	988
5천, 0.62	620, Q	923	1059
3천, 0.43	116, M	895	717
3천, 0.63	115, Q	707	724
1천, 0.4	61, M	592	574
1천, 0.7	63, Q	555	623

Table 6. Search, Q, M 동작시간(중복도 검사)

N, sortedPair	S [μs]	Q [μs]	M [μs]	I [μs]
5만, 99.81	7567, I	45939	5631	4317
5만, 56.77	8401, Q	5322	5821	227617
5만, 3.84	6093, M	10574	6093	629078
1만, 99.59	2313, I	10481	2349	1866
1만, 51.34	2334, Q	2651	2765	19697
1만, 3.90	2324, M	3097	2443	31496
5천, 98.81	1390, I	2896	1757	820
5천, 51.29	1280, Q	2097	2298	7141
5천, 3.86	1294, M	3356	2464	11561
1천, 96.29	401, I	1198	615	394
1천, 52.95	429, Q	497	661	2036
1천, 3.94	440, M	745	609	2659

Table 7. Search, Q, M, I 동작시간(정렬쌍 검사)