2021-10396 유성현

1. 정렬 알고리즘 동작방식

1.1. Bubble Sort
버블 정렬은 배열의 마지막 원소부터 역순회한다(이하 '1 차 반복문'). 현재 인덱스(이하 last)는 정렬이 완료되지 않은 배열의 마지막 인덱스를 의미한다. 1 차 반복문의 내부에서 첫 번째 원소부터 last 전까지 순회하며(이하 '2 차 반복문'), 현재 원소와 바로 뒤의 원소를 비교하고 현재 원소가 더 클 경우 자리를 바꾼다. 두 원소의 위치를 서로 바꾼 경우에는 boolean 타입의 변수 swapped 를 true 로 설정한다. 2 차 반복문이 끝나고 swapped 가 false 인 경우, last 이전의 배열이 이미 정렬되어 있음을 의미하므로 '1 차 반복문'을 종료한다.

1.2. Insertion Sort
삽입 정렬은 배열의 두 번째 원소부터 마지막 원소까지 순회하며, 배열 전반부부터 정렬해나간다. 현재 인덱스는 정렬해야할 원소의 인덱스(이하 last)이다. 각 반복에서 insert()를 호출하며 last에 해당하는 원소를 올바른 위치에 삽입한다. insert()는 last부터 1까지 순회하며 현재 원소와 바로 앞의 원소를 비교하고, 현재 원소가 더 작은 경우 두 원소 바꾼다. 현재 원소가 바로 앞의 원소보다 크거나 같은 경우, 이미 올바른 위치에 삽입된 것이므로 반복문을 종료한다. 이렇게 정렬하고자 하는 원소를 올바른 위치로 이동시킨다.

1.3. Heap Sort
힙 정렬은 최대 힙(Max-Heap)을 사용하여 배열을 정렬한다. 최대 힙은 완전 이진 트리임과 동시에 부모 노드의 값이 자식 노드의 값보다 항상 크거나 같은 트리이다. 힙 정렬은 최대 힙 구성(이하 buildHeap()), 이른바, '스며내리기'(이하 percolateDown())를 반복하여 전체 배열을 정렬한다.
buildHeap()을 호출하여 주어진 배열을 최대 힙으로 만든다. buildHeap()은 자녀 노드를 가지고 있는 첫 번째 노드($\lfloor (\text{배열의 크기} - 2) \div 2 \rfloor$)부터 뿌리 노드까지 순회하며 percolateDown()을 재귀적으로 호출한다. 이후 배열의 마지막 원소부터 첫 번째 원소까지 순회하며, 최대 힙의 최대값인 첫 번째 원소와 현재 인덱스(이하 i)의 원소를 서로 바꾼다. 원소를 바꾼 후, percolateDown()를 호출하여 0부터 i-1까지의 부분배열에 대해서만 다시 최대 힙을 만든다.

1.4. Merge Sort
정렬해야할 배열의 복사본을 생성한다. 인플레이스(In-Place) 병합 정렬을 수행하기 위함이다. mergeSort()에 시작 인덱스, 끝 인덱스, 원본 배열, 복사본 배열을 인자로 전달한다. 시작 인덱스와 끝 인덱스를 활용해 mid($= start + \lfloor (end - start) \div 2 \rfloor$)값을 구한다. 배열의 시작 인덱스가 끝 인덱스보다 작은 경우, mid를 기준으로 배열을 반으로 나누고 각 부분에 대해 재귀적으로 mergeSort()를 호출한다. 재귀적으로 호출할 때, 원본 배열과 복사본 배열을 '교차'시킨다. 각 부분 배열이 정렬된 후, merge()를 호출하여 정렬된 부분 배열들을 병합한다. merge()는 정렬된 두 부분 배열을 순서대로 비교하며 병합하는 과정을 수행한다. 비교하는 원소 중 더 작은 값을 결과 배열에 추가하고, 해당 원소의 인덱스를 증가시킨다. 이 과정을 두 부분 배열 중 하나의 원소가 모두 결과 배열에 추가되었을 때까지 반복한다. 이후 다른 부분 배열에 남아 있는 원소를 결과 배열에 추가한다.

1.5. Quick Sort
퀵 정렬은 배열의 첫 번째 인덱스를 시작 인덱스(이하 start)로, 마지막 인덱스를 끝 인덱스(이하 end)로 설정하고, quickSort()를 호출함으로써 정렬을 시작한다. start가 end보다 작은 경우, partition()을 호출한다. partition()은 전달받은 start, end를 활용해 중간 인덱스를 계산하고, 'median-of-three' 방법을 활용해 피벗을 설정한다. 이렇게 함으로써 최악의 불균형적 분할을 피할 수 있다. partition()은 배열을 순회하며 2개의 파티션을 갱신한다. 2개의 파티션을 통해 피벗 미만, 피벗 동일, 피벗 초과의 세 배열로 구분한다. 즉, 첫 번째 파티션은 피벗 미만의 배열의 마지막 인덱스 + 1 이고, 두 번째 파티션은 피벗 초과의 배열의 첫 번째 인덱스 - 1이다. partition()은 앞의 파티션 2개를 담은 정수 배열을 반환한다. 파티션을 통해 분할되는 세 개의 부분 배열 중 피벗 미만과 피벗 초과의 배열에 대해서 quickSort()를 재귀 호출한다.

1.6. Radix Sort
기수 정렬은 자릿수를 기준으로 원소들을 정렬하는 방법으로, 10진법을 사용한다. 배열의 최소값과 최대값을 구하고, 정렬할 원소들의 최대 자릿수를 계산한다. 원소들의 일의 자릿수부터 최대 자릿수까지 순차적으로 정렬을 수행한다. -9부터 9까지 한 자리 정수를 모두 처리할 수 있도록 크기가 19인 계수 배열을 초기화한다. 배열을 순회하면서 현재 자릿수를 기준으로 해당 자릿수를 추출하고, 계수 배열을 갱신한다. 계수 배열을 누적 합한다. 배열을 역순으로 순회하면서, 해당 자릿수에 대응되는 인덱스로부터 계수 배열의 원소를 조사한다. 그때 "계수 배열의 원소-1"이 현재 원소가 결과 배열에서의 인덱스에 해당한다.

2. 정렬 알고리즘 동작 시간 분석
도표에서 N은 데이터 크기, A는 평균, D는 표준편차, M은 최대값, m은 최소값을 의미함.
BS는 버블정렬, IS는 삽입정렬, HS는 힙정렬, QS는 퀵정렬, RS는 기수정렬을 의미함.

2.1. 무작위 데이터에 대한 성능(단위 : 밀리초)
원소의 개수를 아래의 4가지 경우로 나누어 실험을 수행했다. 중복값의 존재에 따른 성능 차이를 배제하고자 가능한 원소의 범위를 넓게 잡아 난수를 생성했다. N=10^k일 때, 난수 x는 $-10^k \le x \le 10^k$

의 범위에서 생성되었다.
아래는 정렬 알고리즘을 반복 수행한 결과이다.(버블정렬 30 회,삽입정렬 100 회,기타 1000 회)

정렬	$N=10^4$				$N=10^5$				$N=10^6$			
	A	D	m	M	A	D	m	M	A	D	m	M
BS	70.3	6.52	63	85	11234	136	1113	11610	1173266	3735	1169668	1184738
IS	12.9	0.53	12	15	1208	14.6	1196	1297	126193	462272	120771	586579
HS	0.62	0.49	0	2	7.56	0.70	7	15	93.9	0.74	93	101
MS	0.55	0.50	0	1	6.96	0.54	6	20	83.5	0.91	82	101
QS	0.61	0.49	0	1	7.54	0.54	7	11	87.9	0.82	86	99
RS	0.18	0.38	0	1	2.23	0.62	1	7	25.3	3.87	21	82

무작위로 생성된 데이터에 대하여, 기수정렬이 평균적으로 성능이 가장 좋다.

2.2.1. 정렬된 데이터에 대한 성능(단위 : 밀리초)

실험 설정은 2.1 과 동일하다.

정렬	$N=10^4$				$N=10^5$				$N=10^6$			
	A	D	m	M	A	D	m	M	A	D	m	M
BS	0	0	0	0	0.03	0.18	0	1	0.37	0.48	0	1
IS	0	0	0	0	0.04	0.02	0	1	1.12	0.32	1	2
HS	0.38	0.49	0	1	4.26	0.44	4	7	47.7	2.66	46	85
MS	0.16	0.37	0	1	1.99	0.27	1	3	23.4	0.56	23	27
QS	0.48	0.50	0	1	14.19	1.00	13	34	411	4.14	408	473
RS	0.19	0.39	0	1	2.36	0.51	2	6	27.4	2.13	24	31

정렬된 데이터에 대해서는 평균적으로 삽입정렬의 성능이 가장 좋다.

2.2.2. 거의 정렬된 데이터에 대한 성능(단위 : 나노초)

배열을 순회하며 k 번째 원소와 k+1 번째 원소를 비교하며 후자가 더 큰 경우를 합산하고, 데이터의 크기로 나누어줌으로써 정렬율을 추산할 수 있다. 거의 정렬된 데이터를 생성함으로써 알고리즘 간 성능 차이를 분석했다. 데이터 크기는 5 만개로 설정했다.

[그림 1]

[그림 2]

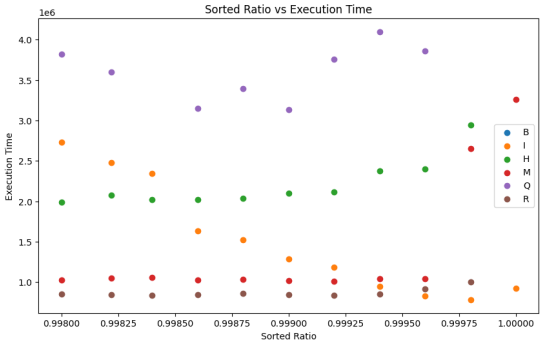


그림 1 에 따르면 정렬율이 0.85 이하일 때, 버블정렬보다 삽입정렬의 성능이 좋고, 삽입정렬보다는 나머지의 성능이 더 좋아보인다. 그러나 그림 2 를 확인하면 정렬율이 0.9995 이상일 때 삽입정렬의 성능이 가장 좋은 것을 확인할 수 있다.

2.3.1. 역순으로 정렬된 데이터에 대한 성능(단위 : 밀리초)

실험 설정은 2.1, 2.2.1 과 동일함.

정렬	$N=10^4$				$N=10^5$				$N=10^6$			
	A	D	m	M	A	D	m	M	A	D	m	M
BS	89.5	0.96	88	91	9036	9.75	9023	9057	911066	1219	908570	914622
IS	23	0.46	23	24	2392	4.93	2385	2404	408457	774622	245120	4951391
HS	0.37	0.48	0	1	4.27	0.45	4	5	46.9	0.51	46	55
MS	0.16	0.36	0	1	1.81	0.39	1	2	20.9	0.51	20	22
QS	0.57	0.49	0	1	15.1	2.54	13	21	454	85	407	680
RS	0.17	0.38	0	1	2.34	0.47	3	3	27	2.23	24	46

역순으로 정렬된 데이터에 대해서는 평균적으로 병합 정렬의 성능이 가장 좋다.

2.3.2. 거의 역순으로 정렬된 데이터에 대한 성능(단위 : 나노초)

정렬율이 아닌 반정렬율을 기준으로 분석한다는 점 이외에 2.2.2 와 동일하다.

[그림 3] [그림 4]

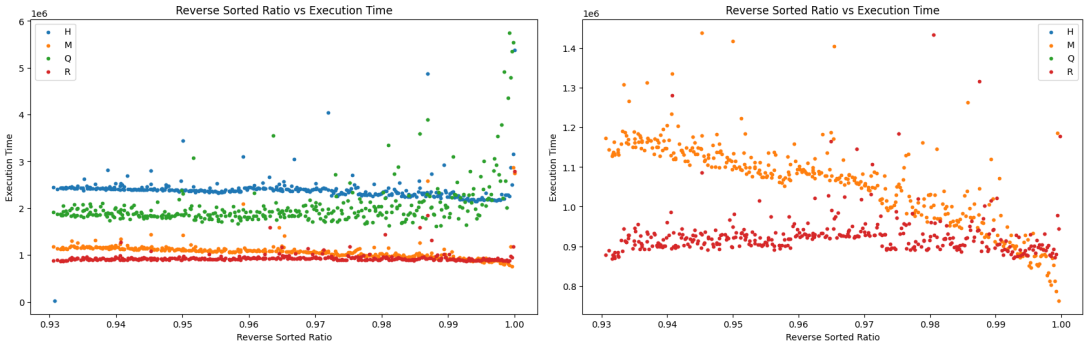


그림 3 에 따르면 반정렬율이 0.98 이하에서는 힙정렬, 퀵정렬, 병합정렬, 기수정렬의 순서로 성능이 좋다. 그러나 그림 4 에 따르면 반정렬율이 약 0.99 이상일 때 병합정렬의 성능이 기수정렬을 능가한다

2.4. 중복값이 많은 데이터에 대한 성능

2.4.1. 모든 값이 동일한 데이터에 대한 성능 비교(단위 : 밀리초)

아래는 모든 정렬 알고리즘에 대하여 1000 번 반복 수행한 결과이다.

정렬	$N=10^4$				$N=10^5$				$N=10^6$			
	A	D	m	M	A	D	m	M	A	D	m	M
BS	0.008	0.1	0	2	0.024	0.15	0	1	0.228	0.42	0	1
IS	0.005	0.07	0	1	0.025	0.16	0	1	0.214	0.41	0	1
HS	0.027	0.16	0	1	0.123	0.33	0	1	1.113	0.33	1	3
MS	0.14	0.35	0	1	1.467	0.5	1	2	17.5	0.55	17	22
QS	0.008	0.09	0	1	0.031	0.17	0	1	0.375	0.48	0	1
RS	0.015	0.12	0	1	0.039	0.19	0	1	0.452	0.52	0	2

모든 값이 동일한 데이터에 대해서는 평균적으로 삽입정렬의 성능이 가장 좋다.

2.4.2. Hash Table 에서 평균충돌률 변화에 따른 분석(단위 : 나노초)

최소값을 0 으로 고정하고, 최대값을 0 부터 1000 까지 변화시켜가며 총 5 만 개의 난수를 생성했다. 항등함수를 해시함수로, 배열의 원소를 해시 테이블에 삽입하며 충돌발생률을 계산했다. 아래는 충돌발생률 변화에 따른 실행결과이다. 충돌발생률은 중복된 원소의 개수를 배열의 크기로 나누어 계산했다.

[그림 5] [그림 6]

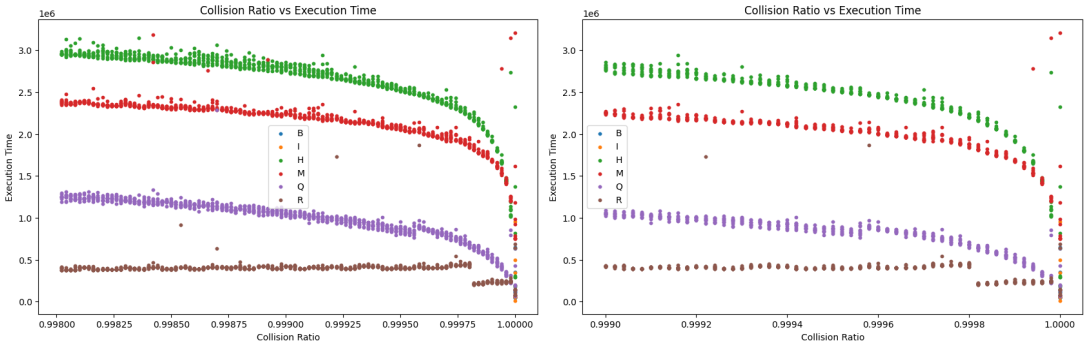


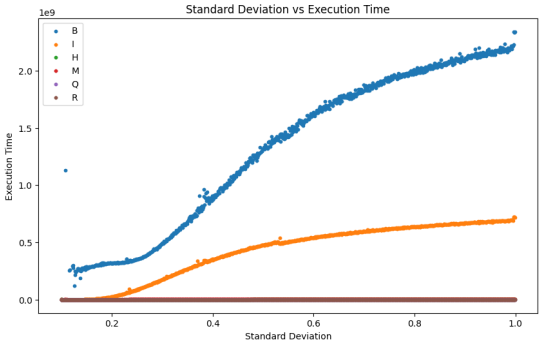
그림 5 를 참고하면, 평균충돌률이 1 부근에서 실행시간이 급격하게 감소함을 확인할 수 있다. 삽입정렬의 경우 2.4.1 에 따르면 모든 원소가 동일할 때, 성능이 가장 좋지만 동일하지 않은 원소가 조금이라도 섞일 때 급격하게 성능이 저하되는 것을 확인했다. 그럼에도 불구하고 확인할 수 있는 사실은 평균적으로 기수정렬의 성능이 가장 좋다는 것이다.

그림 6 은 평균충돌률이 0.999 이상인 범위에 대한 자료이다. 위의 범위에서는 데이터가 거의 동일한 원소로 구성되어 있는 경우이지만, 여전히 기수정렬이 가장 빠르다. 평균충돌률이 1 일때는 동일한 원소로 이루어진 배열이다. 이 경우 삽입정렬이 더 나은 성능을 가짐을 2.4.1 에서 확인했다.

2.4.3. 정규분포에서 표준편차 변화에 따른 분석

특정 숫자에 몰려있는 데이터에 대한 기능의 차이를 확인하기 위해 무작위 확률이 아닌 정규분포에 따라 5 만개의 난수를 생성해 실험을 수행했다. 평균은 0 으로 고정하고 표준편차를 0.001 씩 변화시켜가며 실행한 결과는 아래와 같다.

[그림 7]



[그림 8]

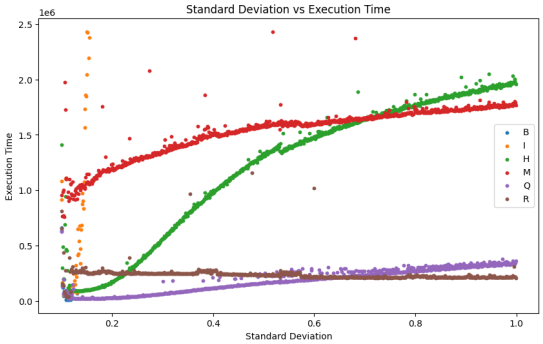


그림 3 과 4 를 참조하면 표준편차가 0 이상 1 이하의 범위에서 정렬 알고리즘 간 성능변동이 확인된다. 특히, 표준편차가 약 0.6 이하의 범위에 대하여 힙 정렬이 병합정렬보다 높은 성능을 보이고, 퀵정렬이 기수정렬보다 높은 성능을 보인다. 종합하면 정규분포를 가정할 때, 표준편차 0.6 이하에서는 퀵정렬이, 0.6 초과에서는 기수정렬이 최고의 성능을 보인다.

2.5. 자릿수에 따른 성능(단위 : 밀리초)

기수정렬은 자릿수에 비례해 복잡도가 증가한다. 따라서 기수 정렬의 성능을 측정하기 위해 최대/최소값의 자리수를 변화시켜가며 100 만개의 난수를 생성했다. 아래는 100 회 반복수행한 결과이다

정렬	$\leq 10^1$	$\leq 10^2$	$\leq 10^3$	$\leq 10^4$	$\leq 10^5$	$\leq 10^6$	$\leq 10^7$	$\leq 10^8$	$\leq 10^9$
HS	56.04	69.65	85.42	94.83	97.16	100.97	94.64	95.96	95.94
MS	46.76	57.93	66.10	75.71	83.02	84.76	84.00	85.52	84.63
QS	18.03	30.17	43.62	58.60	75.36	85.51	86.44	87.34	92.06
RS	8.57	12.15	16.11	19.74	22.96	25.08	26.17	30.21	34.01

int 범위 내에서는 자릿수와 무관하게 기수정렬의 성능이 가장 좋은 것으로 확인된다.

3. Search 동작 방식

첫째, int 의 범위에서는 입력의 최대 자릿수와 무관하게 기수 정렬의 성능이 가장 좋으므로 구현하지 않았다.
둘째, 입력이 이미 정렬이 얼마나 되어 있는지 모든 i 와 i+1 쌍을 통해 근사하고, 정렬율이 0.9995 이상일 때 삽입정렬을 반환한다.
셋째, 두 번째와 같은 방식으로 얼마나 반대로 정렬되어 있는지를 근사하고, 반정렬율이 0.99 이상일 때 병합정렬을 반환한다.
넷째, 배열을 순회하며, 각 원소를 색인으로 하는 Hash table 을 구성하고, 중복이 발생한 총 빈도를 배열의 크기로 나누어 충돌 발생률을 계산한다. 그러나 충돌 발생률이 1 에 근접할 때까지 알고리즘 간 명확한 성능 차이가 보이지 않고, 기수 정렬이 꾸준히 빠른 것으로 확인된다. 충돌발생률이 중복에 따른 성능 차이를 보이는 명확한 지표로 사용되기 어렵다고 판단해 구현하지 않았다.
다섯째, 데이터의 정규분포를 가정하고 표준편차를 계산한다. 표준편차가 0.6 이하일 때, 퀵정렬을 반환한다.
여섯째, 위 알고리즘은 기본값으로 기수정렬을 반환한다.

4. Search 동작 시간 분석

4.1. TestCase 생성 방식

첫째, int 범위에서 난수를 2개 추출하여 난수 배열 생성의 상한과 하한을 설정한다.
둘째, 해당 범위에서 50,000개의 난수를 생성한다.
셋째, 30%의 확률로 올바르게 정렬하고, 30%의 확률로 반대로 정렬하고, 나머지 40%의 확률로 본래 배열을 유지한다.
넷째, 0부터 50,000 사이에서 난수 하나를 추출하여, 그 횟수만큼 무작위로 두 원소를 자리바꿔 배열을 변형한다.

4.2. 분석 결과

위의 방식으로 생성된 난수 배열에 대하여 '(a) DoSearch를 활용해 선택된 정렬을 수행하는 방식'과 '(b) 모든 정렬을 수행하는 방식'을 수행했다. 먼저 수행시간을 측정하고, 다음으로 (a)가 반환하는 최적의 정렬 알고리즘과 (b)가 측정한 가장 빠른 알고리즘이 일치하는지 확인했다.
총 10,000번 반복수행한 결과 (a) 방식의 평균 수행시간은 1.8242 (millisec), (b) 방식의 평균 수행시간은 2653.425 (millisec)으로, DoSearch를 사용한 방법이 **약 1454배 빠른 것으로 확인되었다.** 아울러, DoSearch가 반환하는 최적의 알고리즘은 모든 정렬알고리즘을 수행해 찾은 최적의 알고리즘과 **92.78% 일치함**을 확인했다.