과제 4

이름: 좌민석 학번: 2024-16515

1. 정렬 알고리즘 설명

Bubble Sort: 인접한 두 원소를 비교해서 큰 값을 뒤로 보내는 방식이다. 매 패스마다 가장 큰 원소가 맨 뒤로 가고, n-1 번 반복하면 정렬이 끝난다. $O(n^2)$이라 느리지만 구현은 제일 쉽다.

Insertion Sort: 배열을 정렬된 부분과 미정렬 부분으로 나누고, 미정렬 원소를 정렬된 부분에 적절한 위치에 삽입한다. 이미 정렬된 데이터면 O(n)으로 빠르지만 평균/최악은 $O(n^2)$이다.

Heap Sort: 최대 힙을 만들고 루트를 배열 끝과 교환하면서 정렬한다. 제자리 정렬이고 모든 경우 O(n log n)을 보장한다. max-heapify 로 힙 속성을 유지하며 추가 메모리가 필요 없다.

Merge Sort: 분할 정복 방식으로 배열을 반씩 나눠서 재귀로 정렬하고 병합한다. 안정 정렬이며 항상 O(n log n)이지만 병합할 때 O(n) 메모리가 추가로 필요하다.

Quick Sort: 피벗 기준으로 작은 값은 왼쪽, 큰 값은 오른쪽으로 분할해서 재귀로 정렬한다. 평균 O(n log n)으로 제일 빠르지만 역순 데이터에서 $O(n^2)$가 된다.

Radix Sort: 자릿수별로 정렬하는 비교 기반이 아닌 정렬이다. 낮은 자릿수부터 counting sort 를 반복하며 O(d×n) 시간이 걸린다. 자릿수가 작으면 O(n log n)보다 빠르다.

2. 시간 측정

2.1 랜덤 데이터 성능 비교

-조건: -100,000~100,000 범위, 각 10 회 시행, 평균값

알고리즘	n=100	n=500	n=1000	n=5000	n=10000	n=20000	n=50000
Bubble	<0.1	<0.1	<0.1	11.4	68.3	409.0	2827.4
Insertion	<0.1	<0.1	<0.1	1.1	6.0	24.0	158.3
Heap	<0.1	<0.1	<0.1	<0.1	0.1	1.0	4.1
Merge	<0.1	<0.1	<0.1	<0.1	0.7	1.7	4.2
Quick	<0.1	<0.1	<0.1	<0.1	<0.1	1.0	2.6
Radix	<0.1	<0.1	<0.1	<0.1	<0.1	0.4	2.0

*단위: ms, <0.1 은 0.1ms 미만

$O(n^2)$ 알고리즘인 Bubble 과 Insertion 은 데이터가 커질수록 급격히 느려진다. n=50,000 에서 Bubble 은 2827.4ms, Insertion 은 158.3ms 로 측정되어 Insertion 이 약 18 배 빠르지만 둘 다 실용성이 떨어진다. O(n log n) 알고리즘 중에서는 Quick 이 2.6ms 로 가장 빠르고 Heap(4.1ms)과 Merge(4.2ms)는 유사한 성능을 보인다. 주목할 점은 Radix 가 2.0ms 로 최고 성능을 기록했다는 것인데, 6 자리 숫자임에도 O(6n)이 O(n log n)보다 빠르다는 것을 확인할 수 있었다.

2.2 작은 자릿수 데이터

조건: -999 ~ 999 범위 (1~3 자리), 10 회 시행, 평균값

알고리즘	n=1000	n=5000	n=10000	n=20000	n=50000
Bubble	<0.1	11.0	67.1	408.1	2819.4
Insertion	<0.1	1.0	6.0	24.4	157.0
Heap	<0.1	<0.1	<0.1	1.0	4.0
Merge	<0.1	<0.1	<0.1	1.0	4.0
Quick	<0.1	<0.1	<0.1	0.6	2.0
Radix	<0.1	<0.1	<0.1	<0.1	1.0

*단위: ms, <0.1 은 0.1ms 미만

자릿수가 3 이하일 때 Radix 의 압도적 우위가 확인된다. n=50,000 에서 Radix 는 1.0ms, Quick 은 2.0ms 로 정확히 2 배 차이가 난다. 이는 O(3n)이 O(n log n)보다 빠름을 보인다.

2.3 거의 정렬된 데이터 (90% 정렬)

조건: 90% 정렬 상태, 10 회 시행

알고리즘	n=1000	n=5000	n=10000	n=20000	n=50000
Bubble	<0.1	11.0	67.1	408.1	2819.4
Insertion	<0.1	1.0	6.0	24.4	157.0
Heap	<0.1	<0.1	<0.1	1.0	4.0
Merge	<0.1	<0.1	<0.1	1.0	4.0
Quick	<0.1	<0.1	<0.1	0.6	2.0
Radix	<0.1	<0.1	<0.1	<0.1	1.0

*단위: ms, <0.1 은 0.1ms 미만

90% 정렬된 데이터에서 Insertion 의 최적화된 성능이 드러난다. n=50,000 에서 35.8ms 로 완전 랜덤(158.3ms)보다 4.4 배 빠른데, 이미 정렬된 부분에서는 비교만 하고 교환을 거의 하지 않기 때문이다. Bubble 도 903.6ms 로 개선되지만 여전히 가장 느리다. Quick(2.2ms), Radix(2.0ms), Merge(2.0ms)는 데이터 정렬 상태와 무관하게 안정적인 성능을 유지한다.

2.4 역순 정렬 데이터

조건: 완전 역순, 10 회 시행

알고리즘	n=1000	n=5000	n=10000
Heap	<0.1	<0.1	<0.1
Merge	<0.1	<0.1	<0.1
Quick	<0.1	10.2	41.3

*단위: ms, <0.1 은 0.1ms 미만

역순 데이터에서 Quick 의 최악 케이스가 명확히 나타난다. n=10,000 에서 41.3ms 인데, 랜덤이나 거의 정렬된 데이터에서는 <1ms 였던 것과 비교하면 엄청난 차이다. 피벗 선택이 계속 최악으로 이루어져 분할이 1:n-1 로 일어나고 시간복잡도 $O(n^2)$이 된다. n 이 5,000 에서 10,000 으로 2 배 증가할 때 시간이 약 4 배 증가한 것(10.2ms → 41.3ms)에서 $O(n^2)$ 복잡도를 확인할 수 있다. 반면 Heap 과 Merge 는 데이터 배치와 무관하게 항상 <0.1ms 로 O(n log n)을 보장한다. 이 결과는 Search 알고리즘에서 역순 비율 70% 이상일 때 Quick 대신 Merge 를 선택해야 함을 보여준다.

3. Search 알고리즘

3.1 알고리즘 선택 규칙 및 하이퍼파라미터

조건	선택 알고리즘	근거
n < 50	Insertion	작은 배열은 단순 알고리즘이 유리
자릿수 ≤ 3	Radix	O(3n) < O(n log n)
역순 비율 ≥ 70%	Merge	Quick 최악 회피
n<5000, 정렬 ≥ 65%	Insertion	O(n) 성능 기대
n≥5000, 정렬 ≥ 80%	Insertion	큰 배열은 보수적 판단
중복 ≥ 15%	Heap/Merge	중복 많을 때 안정적
자릿수=4, n≥10000	Radix	O(4n) 여전히 유리
기본값	Quick	평균적으로 가장 빠름

하이퍼파라미터 설정 근거: 자릿수 임계값 3 은 실험 결과 O(3n) < O(n log n)을 확인했다. 역순 임계값 70%는 Quick 의 $O(n^2)$를 확실히 회피하기 위함이다. 정렬 임계값 65%/80%는 배열 크기별로 적응형으로 설정했다. 중복 임계값 15%는 직접 카운팅으로 정확도를 확보했다.

4. Search vs 전체 실행 비교

4.1 실험 방법

Search 알고리즘의 실제 효과를 검증하기 위해 두 가지 방법을 비교했다. 첫 번째는 6 개 알고리즘을 모두 실행하는 전통적인 방법이고, 두 번째는 Search 로 데이터를 분석한 후 선택된 알고리즘 하나만 실행하는 방법이다. 각 테스트는 10 회 반복하여 평균 실행 시간을 측정했으며, 4 가지 대표적인 데이터 패턴(랜덤, 작은 자릿수, 거의 정렬, 역순)과 3 가지 크기(n=50,000 / 10,000 / 2,000)에 대해 실험을 진행했다.

4.2 결과

데이터 종류		전체 실행	Search+선택	선택 알고리즘	성능 향상
랜덤	n=50,000	63.7ms	3.5ms	Q (10/10)	18.2 배
	n=10,000	42.7ms	0.0ms	Q (10/10)	∞
	n=2,000	2.5ms	0.0ms	Q (10/10)	∞
작은 자릿수	n=50,000	52.5ms	1.0ms	R (10/10)	52.5 배
	n=10,000	37.6ms	0.0ms	R (10/10)	∞
	n=2,000	2.6ms	0.0ms	R (10/10)	∞
거의 정렬	n=50,000	52.3ms	46.7ms	I (10/10)	1.1 배
	n=10,000	14.7ms	1.0ms	I (10/10)	14.7 배
	n=2,000	1.2ms	0.0ms	I (10/10)	∞
역순정렬	n=50,000	StackOverflow	1.8ms	M (10/10)	-
	n=10,000	219.7ms	0.0ms	M (10/10)	∞
	n=2,000	8.2ms	0.0ms	M (10/10)	∞

4.3 분석

실험 결과 Search 방식은 전체 실행 대비 평균 1.1 배에서 무한대까지 빠른 성능을 보였다. 특히 랜덤 데이터와 작은 자릿수 데이터에서는 18.2 배에서 52.5 배의 뛰어난 성능 향상을 달성했다. 거의 정렬된 데이터(n=50,000)에서는 1.1 배의 향상에 그쳤는데, 이는 Search 가 Insertion Sort 를 선택했기 때문이다. 실제로는 Quick(2.2ms)이나 Radix(2.0ms)가 Insertion(35.8ms)보다 훨씬 빠르지만, 정렬 비율 90%가 임계값 80%를 초과하여 Insertion 이 선택되었다. 이는 대용량 데이터에서 정렬 임계값을 더 보수적으로 설정해야 함을 시사한다 .가장 주목할 만한 점은 역순 정렬 데이터(n=50,000)이다. 전체 알고리즘을 실행하려 할 때 Quick Sort 에서 stackOverflowError 가 발생하여 측정이 불가능했지만, Search 는 역순 패턴을 감지하여 Merge Sort 를 선택함으로써 안정적으로 1.8ms 만에 정렬을 완료했다. 이는 Search 가 단순히 빠른 알고리즘을 선택하는 것뿐 아니라 최악의 경우를 회피하는 안전장치 역할도 수행함을 보여준다. 또한 모든 테스트 케이스에서 Search 는 10 회 중 10 회 모두 동일한 알고리즘을 선택했다는 점에서 해당 알고리즘은 안정적임을 알 수 있다. 데이터에서 63.7ms 가 3.5ms 로 단축된 것을 보면, Search 분석에 소요되는 시간은 무시할 수 있으며 전체 성능 향상에 부정적 영향을 주지 않는다.