Assignment #2

Student ID: 2021202048

Name: 장대한

Experimental Setup

1. 실험 환경

해당 실험은 정렬 알고리즘의 정확성 검증 및 성능 비교를 위해 다음과 같은 환경에서 수행되었습니다.

- 언어: Kotlin
- 빌드 도구: Gradle
- 테스트 프레임워크: JUnit 5, AssertJ
- 벤치마크 도구: JMH (Java Microbenchmark Harness)

2. 구현 알고리즘

4가지의 기본 정렬 알고리즘과 Counting Sort 알고리즘을 구현했습니다.

- Insertion Sort (삽입 정렬)
- Merge Sort (병합 정렬)
- Quick Sort (퀵 정렬)
- Heap Sort (힙 정렬)
- Counting Sort (계수 정렬)

3. 정확성 검증

JUnit 5의 @ParameterizedTest를 활용하여 다양한 입력 케이스에 대해 각 알고리즘이 올바르게 동작하는지 검증했습니다. 다음은 검증을 위해 사용한 데이터 종류입니다.

- 짝수 개의 숫자로 이루어진 리스트
- 홀수 개의 숫자로 이루어진 리스트
- 빈 리스트
- 중복값이 포함된 리스트
- 무작위 난수 리스트

4. 데이터 분포 시나리오

다양한 데이터 분포 시나리오를 설정하여 알고리즘별 성능 특성을 분석했습니다. 실험한 데이터 분포 시나리오는 다음과 같습니다.

1. RANDOM: 완전히 무작위인 데이터
2. SORTED: 이미 정렬된 데이터
3. REVERSE: 역순으로 정렬된 데이터
4. SMALLRANGE: 작은 값의 범위 내에 분포하는 데이터
5. BIGRANGE: 큰 값의 범위 내에 분포하는 데이터
6. NEARLY_SORTED: 거의 정렬된 데이터

5. 벤치마크 진행 방식

본 실험은 각 테스트 케이스(입력 크기 N, 데이터 분포)에 대해 총 3회의 포크(Fork)를 수행합니다. 첫 번째 포크는 JVM 웜업을 위해 수행되며 결과 측정에는 포함되지 않습니다. 이후 2회의 포크에서 실제 성능을 측정합니다. 각 포크 내에서 웜업 단계는 2초씩 2회 반복 수행하여 충분히 최적화를 수행하도록 합니다. 실제 측정 단계는 2초씩 3회 반복 수행하여 평균 실행 시간을 산출합니다.

입력 크기는 10부터 100, 1000, 10000 까지 늘려서 테스트합니다.

```
@State(Scope.Benchmark)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 2, time = 2, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 3, time = 2, timeUnit = TimeUnit.SECONDS)
@Fork(value = 2, warmups = 1)
@BenchmarkMode(Mode.AverageTime)
open class Benchmark {
```

Results

1. 알고리즘의 정확성

관련한 소스코드 전체는 src/test/kotlin/SortinTest.kt에 있습니다.

알고리즘의 정확성을 확인하기 위해 사용한 데이터에 대한 코드는 아래와 같습니다. 해당 데이터는 성능 테스트에 사용되는 모든 알고리즘에 대해서 적용했습니다.

```kotlin
companion object {
    @JvmStatic 5 Usages
    fun sortingProblem(): List<Arguments> {
        val evenArgument = Arguments.of(
            (0 ≤ .. ≤ 100).shuffled().toMutableList(),
            (0 ≤ .. ≤ 100).toList()
        )

        val oddArgument = Arguments.of(
            (0 ≤ .. ≤ 123).shuffled().toMutableList(),
            (0 ≤ .. ≤ 123).toList()
        )

        val emptyArgument = Arguments.of(
            mutableListOf<Int>(),
            emptyList<Int>()
        )

        val duplicatedList = (0 ≤ .. ≤ 10).shuffled().toMutableList()
        duplicatedList.add(Random.nextInt(until = 10))
        val sortedDuplicatedList = duplicatedList.sorted()
        val duplicatedArgument = Arguments.of(
            duplicatedList,
            sortedDuplicatedList
        )

        return listOf(evenArgument, oddArgument, emptyArgument, duplicatedArgument)
    }
}
```

아래는 실제로 정렬의 동작을 확인하기 위해서 사용한 테스트코드 입니다.

```
@ParameterizedTest
@MethodSource("sortingProblem")
fun `삽입 정렬의 동작 테스트`(rawArray: MutableList<Int>, sortedArray: List<Int>) {
    assertThat(InsertionSort.sort(rawArray))
        .isEqualTo(sortedArray)
}

@ParameterizedTest
@MethodSource("sortingProblem")
fun `합병 정렬의 동작 테스트`(rawArray: MutableList<Int>, sortedArray: List<Int>) {
    assertThat(MergeSort.sort(rawArray))
        .isEqualTo(sortedArray)
}

@ParameterizedTest
@MethodSource("sortingProblem")
fun `퀵 정렬의 동작 테스트`(rawArray: MutableList<Int>, sortedArray: List<Int>) {
    assertThat(QuickSort.sort(rawArray))
        .isEqualTo(sortedArray)
}

@ParameterizedTest
@MethodSource("sortingProblem")
fun `힙 정렬의 동작 테스트`(rawArray: MutableList<Int>, sortedArray: List<Int>) {
    assertThat(HeapSort.sort(rawArray))
        .isEqualTo(sortedArray)
}

@ParameterizedTest
@MethodSource("sortingProblem")
fun `계수 정렬의 동작 테스트`(rawArray: MutableList<Int>, sortedArray: List<Int>) {
    assertThat(CountingSort.sort(rawArray))
        .isEqualTo(sortedArray)
}
```

테스트 결과는 다음과 같습니다. 모든 정렬 알고리즘이 정상적으로 작동함을 확인할 수 있었습니다.

2. 성능 측정 Baseline

정렬 알고리즘의 순수한 실행 시간만을 측정하기 위해 입력 데이터를 가변 리스트로 복사하는 비용을 별도의 Baseline 벤치마크로 측정했습니다. 또한 측정 결과가 사용되지 않을 경우 발생할 수 있는 JVM의 불필요한 코드 제거(Dead Code Elimination) 최적화를 방지하고자 Blackhole을 활용했습니다.

```
@Benchmark
fun baseline(bh: Blackhole) {
    bh.consume(experimentList.toMutableList())
}
```

3. 알고리즘의 성능(처리율)

처리율은 초당 처리 가능한 작업(정렬 수행) 횟수(ops/sec)로 평균 시간의 역수로 계산됩니다. 값이 클수록 성능이 좋습니다. 입력 크기가 N일때의 처리율입니다.

Algorithm	RANDOM	SORTED	REVERSED
Baseline	240.84	176.25	230.17
Counting	14.96	14.96	15.3
Heap	0.54	0.82	0.79
Insertion	0.00002	0.023	0.00001
Merge	0.75	1.28	1.29
Quick	0.83	0.000004	0.000005

Algorithm	NEARLY_SORTED	SMALL_RANGE	BIG_RANGE
Baseline	260.91	265.8	263.29
Counting	15.3	15.54	7.02
Heap	0.77	0.54	0.54
Insertion	0.0008	0.00002	0.00002
Merge	1.16	0.76	0.75
Quick	0.22	0.86	0.78

4. 알고리즘의 성능(평균 시간)

작업 1회당 소요된 평균 시간(ns)입니다. 값이 작을수록 성능이 좋습니다.

Algorithm	RANDOM	SORTED	REVERSED
Baseline	4,152	5,674	4,345
Counting	66,830	66,130	66,842
Heap	1,847,522	1,217,880	1,264,707
Insertion	49,307,534	42,632	104,578,204
Merge	1,332,954	784,100	774,211
Quick	1,207,836	264,314,513	203,155,978

Algorithm	NEARLY_SORTED	SMALL_RANGE	BIG_RANGE
Baseline	3,833	3,762	3,798
Counting	65,377	64,358	142,451
Heap	1,290,782	1,866,467	1,838,912
Insertion	1,306,508	49,137,195	49,233,478
Merge	859,656	1,318,709	1,329,797
Quick	4,463,783	1,169,667	1,289,768

5. 알고리즘의 성능(메모리)

작업 1회당 할당된 평균 메모리 양(Byte/op)입니다. 값이 작을수록 메모리 효율이 좋습니다.

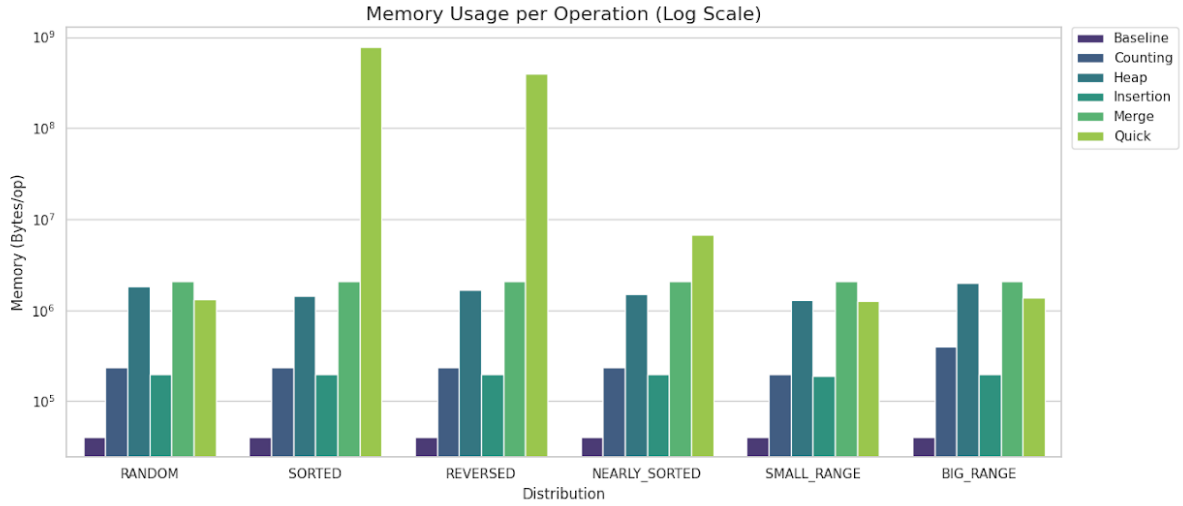
Algorithm	RANDOM	SORTED	REVERSED
Baseline	40,040	40,040	40,040
Counting	238,008	238,008	238,008
Heap	1,811,384	1,436,264	1,665,960
Insertion	197,986	197,992	197,997
Merge	2,081,928	2,081,928	2,081,928
Quick	1,330,984	779,770,139	399,990,017

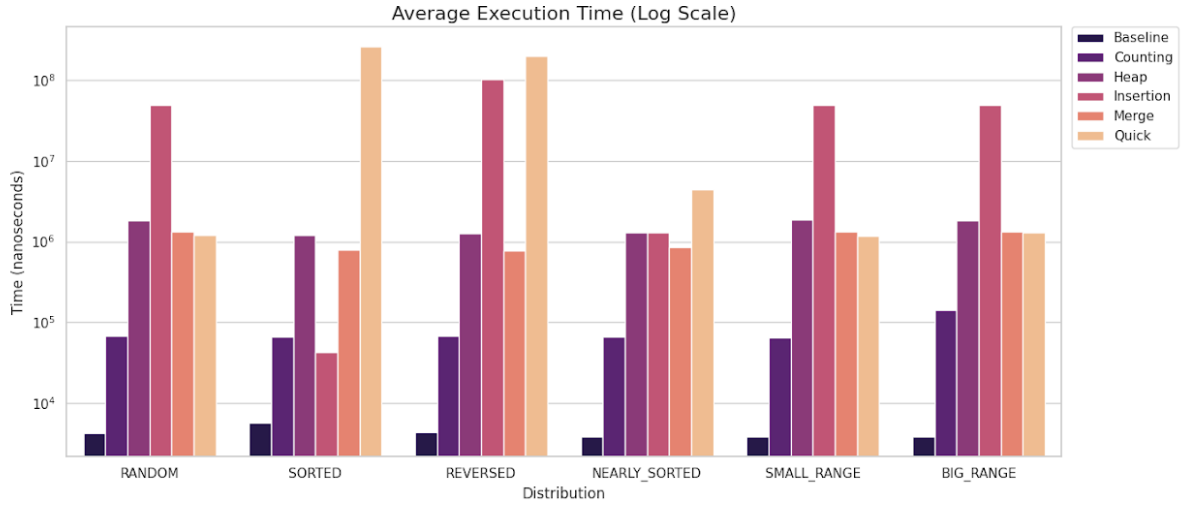
Algorithm	NEARLY_SORTED	SMALL_RANGE	BIG_RANGE
Baseline	40,040	40,040	40,040
Counting	238,008	197,832	399,664
Heap	1,494,440	1,282,152	1,977,112
Insertion	197,992	189,810	199,682
Merge	2,081,928	2,081,928	2,081,928
Quick	6,683,497	1,268,360	1,385,560

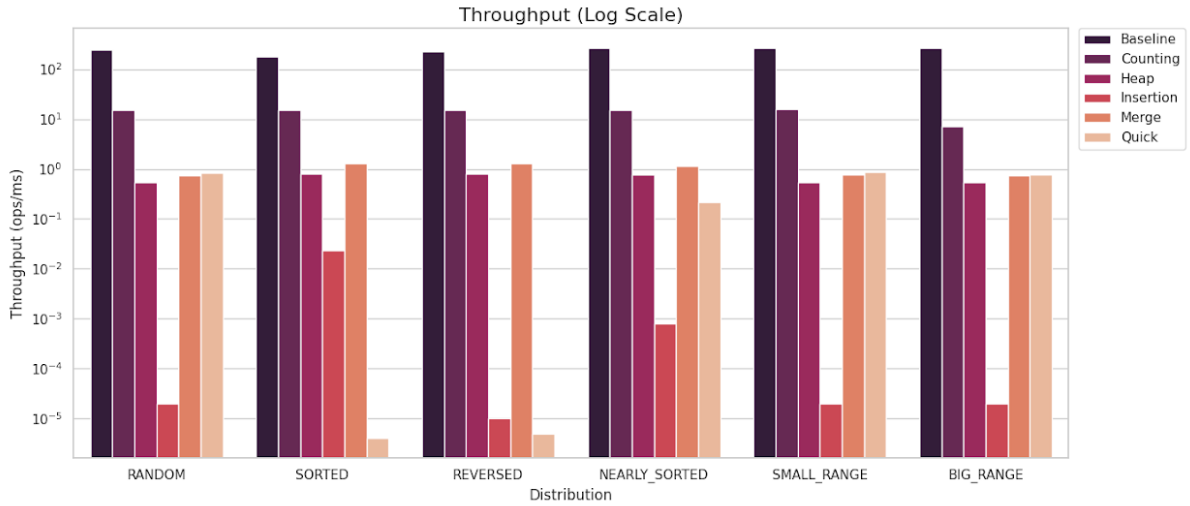
6. 전체 결과

벤치마크에 대한 전체 결과는 build/results/output.json 에서 확인할 수 있습니다.

Visualizations







Analysis

1. 실행 속도 및 처리율 분석

데이터의 정렬 유무와 값의 범위가 큰지가 성능에 중요한 영향을 주었습니다.

- 퀵 정렬(Quick Sort)은 데이터 상태에 따라 성능 차이가 매우 컸습니다. 무작위 데이터에서는 빨랐지만(평균 1.2ms) 이미 정렬되어 있거나 역순인 데이터에서는 약 200배 이상 느려졌습니다(203ms~264ms). 이는 잘못된 피벗 선택으로 인한 것이라고 생각합니다.
- 삽입 정렬(Insertion Sort)은 반대였습니다. 무작위 데이터에서는 가장 느린 편이었지만(약 49ms) 이미 정렬된 데이터에서는 0.042ms 만에 끝나며 가장 빨랐습니다. 거의 정렬된 데이터에서도 퀵 정렬보다 빨랐습니다.
- 계수 정렬(Counting Sort)은 대부분의 상황에서 가장 빨랐습니다(약 15 ops/ms). 하지만 값의 범위가 매우 큰 데이터(Big Range)에서는 속도가 절반 정도로 떨어졌습니다.
- 병합 정렬(Merge Sort)과 힙 정렬(Heap Sort)은 데이터 상태와 상관없이 항상 일정한 속도를 유지했습니다.

2. 메모리 사용량 분석

- 퀵 정렬은 정렬된 데이터에서 재귀 호출이 깊어져 무작위 데이터보다 수백 배 많은 메모리(약 779MB/op)를 사용했습니다.
- 삽입 정렬과 힙 정렬은 추가 메모리를 거의 사용하지 않아 효율적이었습니다.
- 병합 정렬은 알고리즘 특성상 항상 일정한 양(약 2MB/op)의 추가 메모리가 필요했습니다
- 계수 정렬은 데이터 값의 범위가 클수록 메모리를 더 많이 사용했습니다.

3. 결론

모든 상황에서 항상 가장 좋은 정렬 알고리즘은 없었습니다. 데이터가 거의 정렬된 상태라면 삽입 정렬이 안정적인 성능이 필요하다면 힙 정렬이나 병합 정렬을 선택하는 것이 좋을 것 같습니다. 만약 정수 데이터면서 값의 범위가 작다면 계수 정렬이 좋은 선택이 될 것 같다는 생각을 해볼 수 있었습니다.