Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЛАБОРАТОРНА РОБОТА № 3

з дисципліни «Теорія алгоритмів»

на тему «Метод швидкого сортування»

ВИКОНАВ:
студент 4 курсу
групи ІП-72з
Сахнюк Антон Юрійович
Залікова - 6224

ПЕРЕВІРИВ:
Доцент кафедри ОТ
к.т.н., с.н.с.
Антонюк А.І.

Київ - 2021

ЗАВДАННЯ

Мета: реалізація трьох модифікацій алгоритму швидкого сортування (Quick Sort) та порівняння їх швидкодії.
Завдання: Реалізувати три модифікації алгоритму швидкого сортування (Quick Sort) та порівняти їх швидкодію. Швидкість алгоритмів порівнюється на основі підрахунку кількості порівнянь елементів масиву під час роботи алгоритмів.

ПРОГРАМНИЙ КОД

```
package ua.santoni7.l3


import java.util.concurrent.atomic.AtomicLong
import kotlin.math.max
import kotlin.math.min
import kotlin.random.Random


fun readNumber(prompt: String = "Input number: "): Int {
    print(prompt);
    val s = readLine()
    return s?.toInt() ?: throw IllegalStateException("Could not read a number")
}


/**
 * Клас, що інкапсулює обрахування опорного елементу під час процедури [partition] у алгоритмі [quickSort]
 *
 * Всередині наведені досліджувані реалізації обрахування опорного елементу:
 * [Left] - завжди лівий елемент
 * [Right] - завжди правий
 * [Middle] - елемент, індекс якого це середнє арифметичне лівого і правого індексів
 */
sealed class PivotProvider(val name: String) {
    abstract fun getPivot(array: IntArray, leftIndex: Int, rightIndex: Int): Int


    object Left : PivotProvider("LEFT") {
```

```
        override fun getPivot(array: IntArray, leftIndex: Int, rightIndex: Int)
= array[leftIndex]

    }


    object Right : PivotProvider("RIGHT") {

        override fun getPivot(array: IntArray, leftIndex: Int, rightIndex: Int)
= array[rightIndex]

    }


    object Middle : PivotProvider("MIDDLE") {

        override fun getPivot(array: IntArray, leftIndex: Int, rightIndex: Int)
= array[(leftIndex + rightIndex) / 2]

    }

}


/**
 * Генерація випадкового масиву
 */
val random = Random(System.currentTimeMillis())


fun generateArray(size: Int, min: Int, max: Int): IntArray = IntArray(size)
{ random.nextInt(min, max) }


fun IntArray.swapElements(i: Int, j: Int) {

    val tmp = this[j]

    this[j] = this[i]

    this[i] = tmp

}


/**
 * Допоміжний метод для обрахунку кількості порівнянь
 */
inline fun <T> runAndIncrement(counter: AtomicLong, action: () -> T): T {

    counter.incrementAndGet()

    return action.invoke()

}
```

```
fun quickSort(array: IntArray, left: Int, right: Int, pivotProvider:
PivotProvider, comparsionCounter: AtomicLong) {
    if (left >= right) return
    val index = partition(array, left, right, pivotProvider, comparsionCounter)
    if (left < index - 1 && index - 1 != right) {
        quickSort(array, left, index - 1, pivotProvider, comparsionCounter)
    }
    if (index < right && index != left) {
        quickSort(array, index, right, pivotProvider, comparsionCounter)
    }
}


fun partition(array: IntArray, l: Int, r: Int, pivotProvider: PivotProvider,
comparsionCounter: AtomicLong): Int {
    var left = l
    var right = r
    val pivot = pivotProvider.getPivot(array, l, r)
    while (left <= right) {
        while (runAndIncrement(comparsionCounter) { array[left] < pivot }) {
            left++
        }

        while (runAndIncrement(comparsionCounter) { array[right] > pivot }) {
            right--
        }

        if (left <= right) {
            array.swapElements(left, right)
            left++
            right--
        }
    }
    return min(r, max(l, left))
}


/**
 * Допоміжний клас що обраховує кількість порівнянь елементів масиву під час
виконання швидкого сортування у окремому потоці
```

```
 */
class Worker(
    val pivotProvider: PivotProvider,
    val array: IntArray
) : Thread(pivotProvider.name) {
    val counter = AtomicLong(0)

    override fun run() {
        quickSort(array, 0, array.size - 1, pivotProvider, counter)
    }

    fun joinAndGetResults(): Long {
        join()
        return counter.get()
    }
}

/**
 * Точка входу програми
 */
fun main(args: Array<String>) {
    val results = mutableMapOf(PivotProvider.Left to 0L, PivotProvider.Right to
0L, PivotProvider.Middle to 0L)
    val iterationCount = readNumber("Input iteration count: ")
    val arraySize = readNumber("Input array size: ")
    val maxElement = readNumber("Input max element: ")
    for (i in 1..iterationCount) {
        val array = generateArray(arraySize, 0, maxElement)

        val w1 = Worker(PivotProvider.Left, array.copyOf())
        val w2 = Worker(PivotProvider.Middle, array.copyOf())
        val w3 = Worker(PivotProvider.Right, array.copyOf())

        val workers = listOf(w1, w2, w3)
        workers.forEach { it.start() }
        workers.forEach {
```

```
            results[it.pivotProvider] = results[it.pivotProvider]!! +
it.joinAndGetResults()
        }
        println("Iteration $i finished\n---------")
    }

    val resultsString = results.toList().joinToString(separator = "\n") {
        "${it.first.name} -> ${it.second * 1.0 / iterationCount} comparsions on
average"
    }
    println("Results on running $iterationCount iterations with array size of
$arraySize:\n$resultsString")
}
```

РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМИ

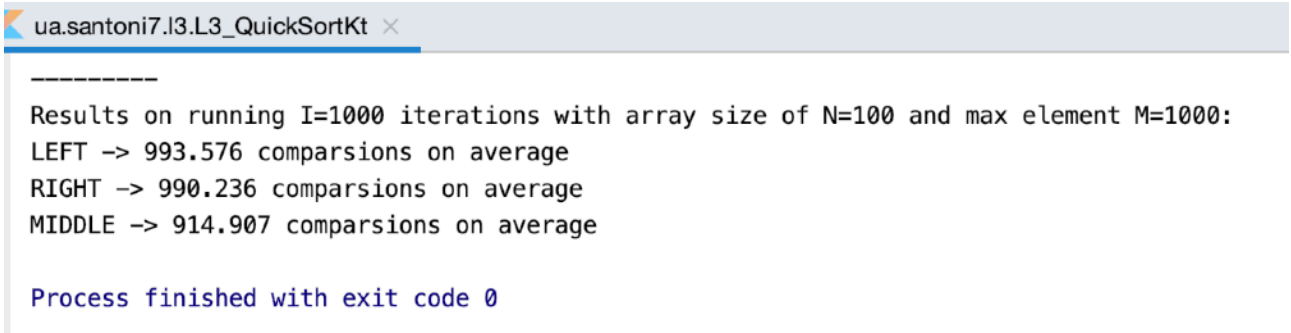
Програма приймає на вхід декілька параметрів: кількість ітерацій I, кількість елементів масиву N, і максимальне можливе значення елементу масиву M. Далі під час виконання кожної ітерації генерується масив N випадкових чисел діапазону [0; M] і створюються 3 його копії для сортування 3ма варіантами швидкого сортування. Ці копії передаються у 3 потоки, кожен з яких сортує масив і підраховує кількість порівнянь. Головний потік чекає завершення виконання усіх 3х потоків і запам'ятовує кількість порівнянь для кожного методу. Процедура повторюється I ітерацій. В кінці ми обраховуємо середнє арифметичне значення кількості порівняннь для кожної варіації сортування і виводимо ці дані на екран.

Було реалізовано 3 варіації вибору опорного елемента (див. клас PivotProvider у коді):

* ***[Left]*** *- завжди лівий елемент*
* ***[Right]*** *- завжди правий*
* ***[Middle]*** *- елемент, індекс якого це середнє арифметичне лівого і правого індексів*

Далі наводяться результати виконання програми при різних I,N,M:

1. Результати при I=1000 ітерацій, розмір масиву N=100, і максимальний елемент масиву M=1000:
LEFT -> 993.576 порівнянь в середньому
RIGHT -> 990.236 порівнянь в середньому
MIDDLE -> 914.907 порівнянь в середньому



2. I=1000, N=1000, M=10000:
LEFT -> 15102.569 порівнянь в середньому
RIGHT -> 15124.166 порівнянь в середньому
MIDDLE -> 13741.802 порівнянь в середньому

```
Results on running I=1000 iterations with array size of N=1000 and max element M=10000:
LEFT -> 15102.569 comparsions on average
RIGHT -> 15124.166 comparsions on average
MIDDLE -> 13741.802 comparsions on average

Process finished with exit code 0
```

3. I=1000, N=10000, M=10000

LEFT -> 201240.828 порівнянь в середньому

RIGHT -> 200939.501 порівнянь в середньому

MIDDLE -> 180847.407 порівнянь в середньому

```
Results on running I=1000 iterations with array size of N=10000 and max element M=10000:
LEFT -> 201240.828 comparsions on average
RIGHT -> 200939.501 comparsions on average
MIDDLE -> 180847.407 comparsions on average

Process finished with exit code 0
```

4. I = 1000, N=50000, M=1000000

LEFT -> 1203019.638 порівнянь в середньому

RIGHT -> 1202433.46 порівнянь в середньому

```
Results on running I=1000 iterations with array size of N=50000 and max element M=1000000:
LEFT -> 1203019.638 comparsions on average
RIGHT -> 1202433.46 comparsions on average
MIDDLE -> 1079854.363 comparsions on average

Process finished with exit code 0
```

MIDDLE -> 1079854.363 порівнянь в середньому

Далі наведемо таблицю отриманих результатів в залежності від кількості елементів масиву

	N=100	N=1000	N=10000	N=50000
LEFT	993	15102	200879	1203019
RIGHT	990	15124	201428	1202433
MIDDLE	914	13741	181016	1079854

ВИСНОВКИ

У даній роботі ми ознайомились з реалізацією алгоритму швидкого сортування, і можливими його варіаціями. Дослідивши можливі варіації алгоритму вибору опорного елемента, ми обрали такі 3 з них - завжди лівий

елемент, завжди правий елемент, чи середній елемент (індекс якого - середнє арифметичне індексів лівого і правого кінців відрізка опрацьовуваного масиву).

Було реалізовано ці 3 варіанти мовою Kotlin з можливістю запустити їх на випадково-згенерованому масиві заданої розмірності. Під час виконання сортування ми підраховуємо кількість порівнянь елементів масиву між собою, що і буде основним показником швидкодії варіації алгоритму. Визначена нами ефективність залежить від вхідних даних, тому програма запускалась у великій кількості ітерацій і потім підраховувалось середнє значення, щоб отримати максимально наближений до правдивого результат

У результаті ми отримали таблицю що показує середню кількість порівнянь для кожної варіації при певному розмірі масиву. З неї ми можемо зробити висновок, що варіації “завжди лівий” і “завжди правий” працюють з приблизно однаковою ефективністю і жоден з них не показує кращий результат перед іншим при усіх значеннях N (розмірність масиву).

Однак, ми можемо побачити, що при усіх досліджених розмірностях масиву, варіація з вибором центрального елементу в якості опорного показала кращий результат ніж дві інші варіації. Наприклад, при N=100 кількість порівнянь алгоритму “middle” склала на 7.9% менше ніж “left”. При N=1000 цей показник складає 9%, при N=10000 — 9,8%, при N=50000 — 10,2%

Як бачимо, покращення ефективності навіть трохи зростає при збільшені розмірності масиву.

Робимо висновок, що серед досліджуваних варіацій найбільш ефективною є вибір центрального елемента в якості опорного.