Lab 1. Алгоритмы поиска в матрице.

1. Язык

Работа была выполнена на языке программирования Java с использованием framework для работы с Excel.

2. Немного кода

Код алгоритмов представлен в Algorithms.txt файле.
2.1. Для создания матриц и тестирования алгоритмов был использован класс Matrix:

```
public class Matrix {

    14 usages
    private final int[][] matrix;
    11 usages
    private final int rows;
    8 usages
    private final int columns;
    11 usages
    private final int key;

    2 usages
    Matrix(int rows, int columns, int key) {
        this.rows = rows;
        this.columns = columns;
        this.key = key;
        matrix = new int[rows][columns];
    }
```

2.2. Генерация матриц:

```
    public void generateInTheFirstWay(){
        for(int i = 0; i < rows; ++i){
            for(int j = 0; j < columns; ++j){
                matrix[i][j] = (columns / rows* i + j) * 2;
            }
        }
    }

    1 usage
    public void generateInTheSecondWay(){
        for(int i = 0; i < rows; ++i){
            for(int j = 0; j < columns; ++j){
                matrix[i][j] = (columns / rows* (i+1) * (j+1)) * 2;
            }
        }
    }
```

2.3. Бинарный поиск:

```java
public boolean binarySearch(){
    for(int i = 0; i < rows; ++i){
        int result = binSearch(i,  right: columns-1);
        if(result != -1){
            return true;
        }
    }
    return false;
}

1 usage
private int binSearch(int row, int right){
    int left = 0;
    while (left <= right){
        int middle = (left+right)/2;
        if(matrix[row][middle] == key){
            return middle;
        }
        if(matrix[row][middle] > key){
            right = middle - 1;
        } else{
            left = middle + 1;
        }
    }
    return -1;
}
```

2.4. Поиск “Лесенкой”:

```java
public boolean ladderSearch() {
    int row = 0, column = columns - 1;
    while (row < rows && column > -1){
        if(matrix[row][column] == key){
            return true;
        }
        else if(matrix[row][column] < key){
            row++;
        } else{
            column--;
        }
    }
    return false;
}
```

2.5. Поиск “Лесенкой” с использованием экспоненциального поиска по столбцу:

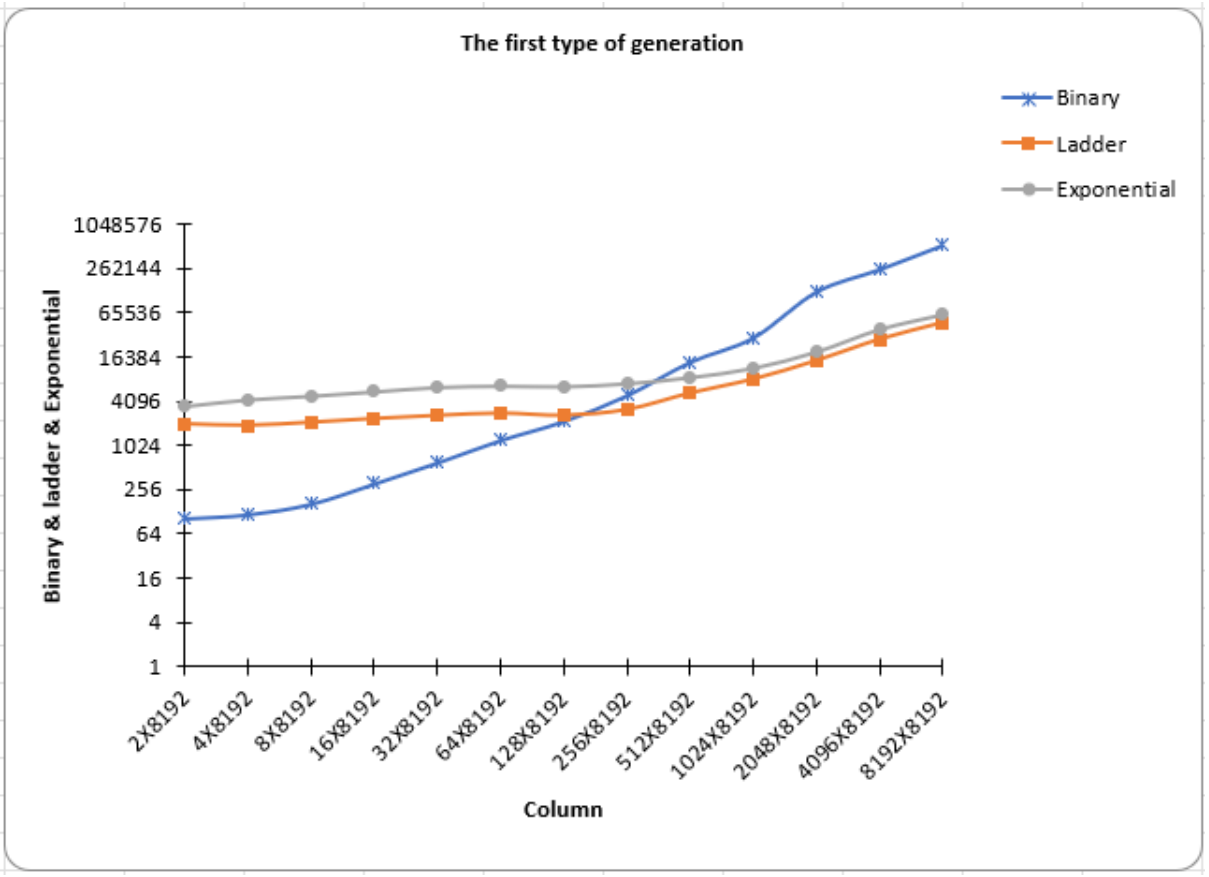
```java
public boolean ladderExpSearch(){
    int line = 0, column = columns - 1;
    while (column > -1){
        if(matrix[line][column] == key){
            return true;
        }
        else if(matrix[line][column] > key){
                column--;
        } else{
            int finish = 1;
            while ((line + finish < rows) && (matrix[line + finish][column] < key)){
                finish = finish * 2;
            }
            int start = line + finish/2;
            if(line + finish >= rows){
                finish = rows - 1;
            } else {
                finish += line;
            }
            if(start == finish){
                return false;
            }
            while (start < finish){
                int middle = (start+finish)/2;
                if(matrix[middle][column] == key){
                    return true;
                }
                if(matrix[middle][column] > key){
                    finish = middle - 1;
                } else{
                    start = middle + 1;
                }
            }
            line = start;
        }
    }
    return false;
}
```

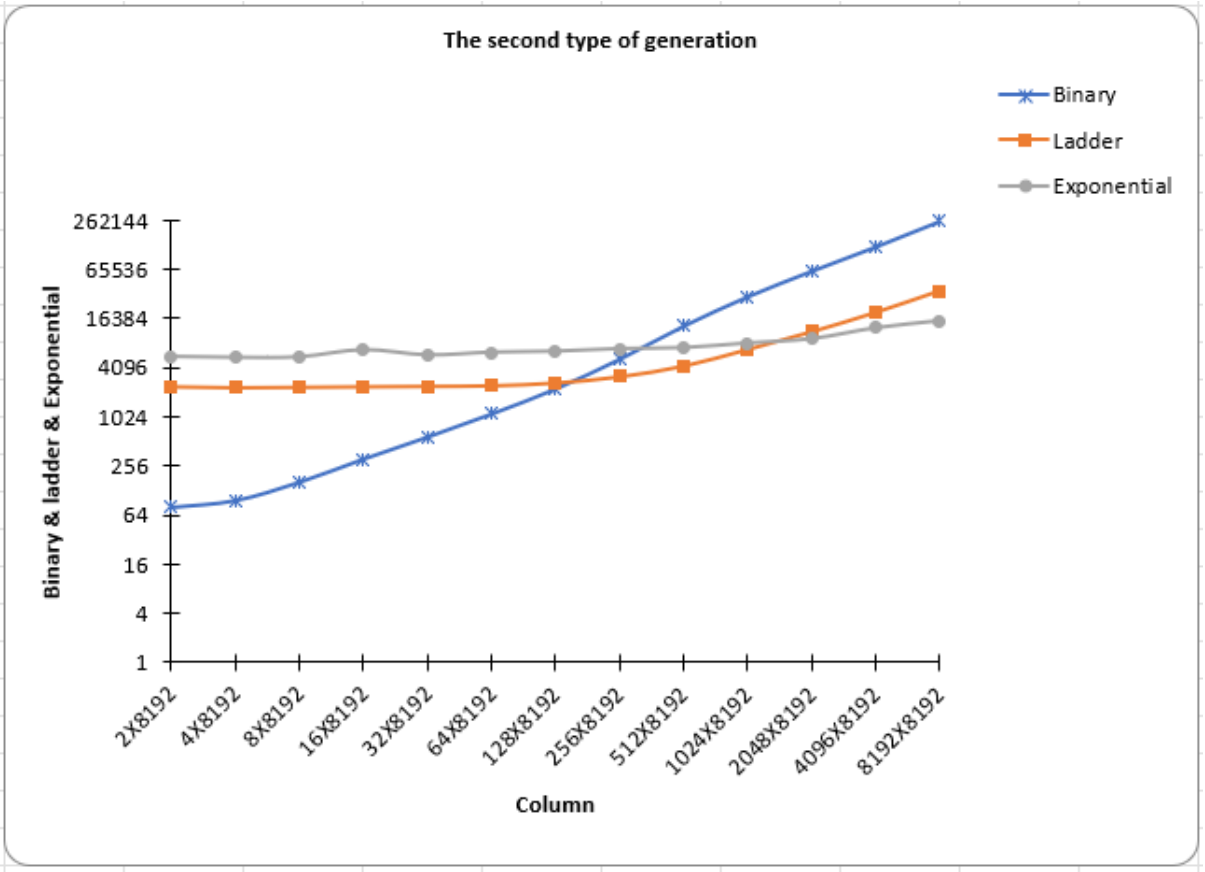
3. Визуализация в Excel

Статистика скорости работы алгоритмов автоматически заполняется в “Comparison of searches.xlsx”:

The first type of generation			
Matrix siz	Binary	Ladder	Exponential
2X8192	105	1996	3495
4X8192	119	1912	4299
8X8192	166	2099	4821
16X8192	312	2356	5524
32X8192	602	2595	6344
64X8192	1214	2788	6608
128X8192	2198	2603	6491
256X8192	4916	3122	7141
512X8192	13962	5222	8653
1024X8192	29901	8049	11597
2048X8192	125517	14339	19382
4096X8192	252189	27870	39279
8192X8192	536612	46732	62442

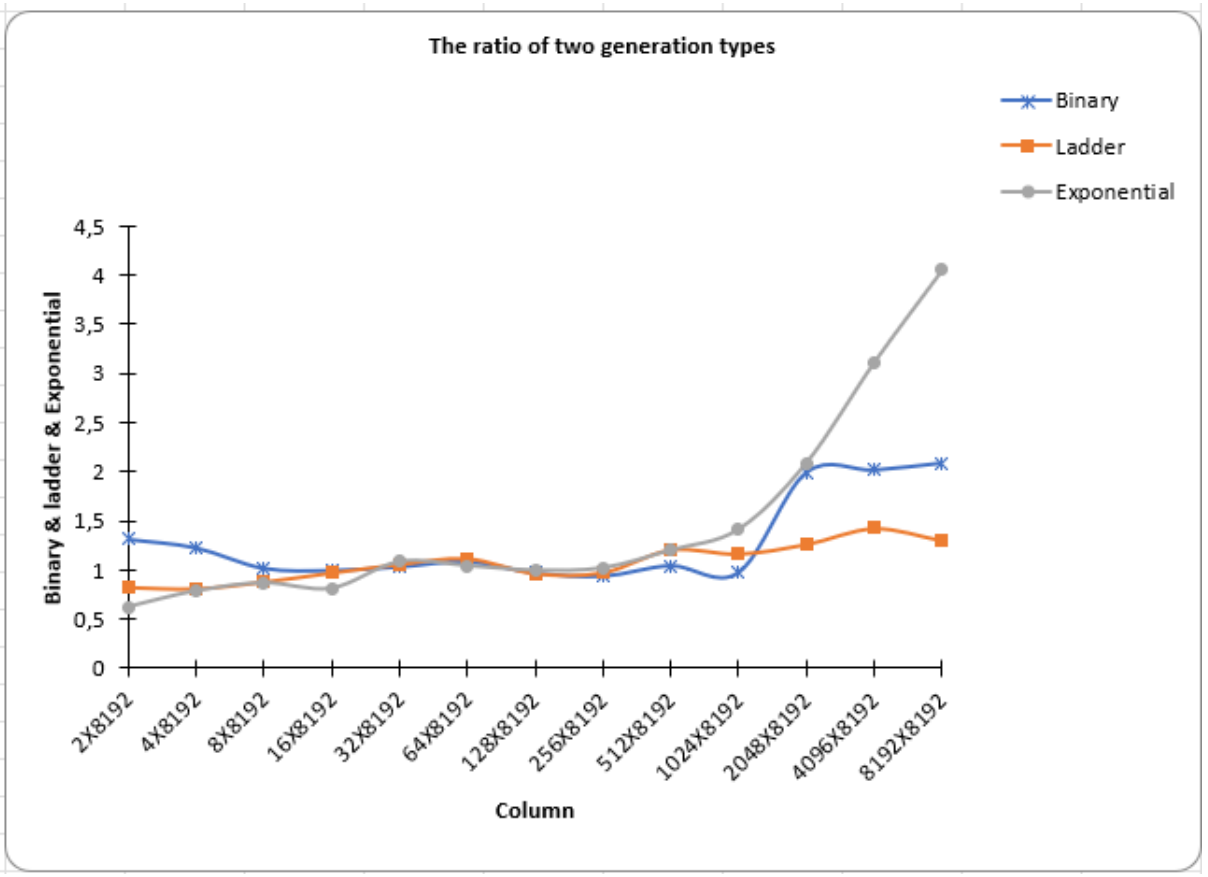
The second type of generation			
Matrix siz	Binary	Ladder	Exponential
2X8192	80	2421	5601
4X8192	97	2358	5464
8X8192	163	2386	5544
16X8192	313	2421	6786
32X8192	583	2449	5848
64X8192	1119	2490	6328
128X8192	2265	2692	6506
256X8192	5240	3201	6990
512X8192	13358	4320	7192
1024X8192	30603	6895	8200
2048X8192	62899	11338	9268
4096X8192	124890	19528	12612
8192X8192	256699	35854	15407





The ratio of two generation types

Matrix siz	Binary	Ladder	Exponential
2X8192	1,3125	0,824453	0,623996
4X8192	1,226804	0,810857	0,786786
8X8192	1,018405	0,879715	0,869589
16X8192	0,996805	0,973152	0,814029
32X8192	1,03259	1,059616	1,084815
64X8192	1,084897	1,119679	1,044248
128X8192	0,970419	0,966939	0,997694
256X8192	0,938168	0,97532	1,021602
512X8192	1,045216	1,208796	1,203142
1024X8192	0,977061	1,167368	1,414268
2048X8192	1,995533	1,264685	2,091282
4096X8192	2,019289	1,427181	3,114415
8192X8192	2,090433	1,303397	4,052833



4. Сравнение алгоритмов

Графики не столь репрезентативны, как числовые результаты тестов, они были использованы скорее как инструмент для визуализации. Для сравнения были использованы данные из трех таблиц, представленных выше.

4.1. Бинарный поиск показывает себя в разы быстрее двух остальных поисков на небольшом количестве строк (до **256**). При размере матрицы **256 x 8192** время работы поиска ‘Лесенкой’ становится равным времени работы бинарного поиска.

4.2. Начиная с размера матрицы **512 x 8192**, разница становится все более заметной, с увеличением строк бинарный поиск работает все хуже и хуже в сравнении с двумя другими алгоритмами.

4.3. При достижении максимального размера матрицы **8192 x 8192** бинарный поиск в разы медленнее поиска ‘Лесенкой’.

4.4. Экспоненциальный поиск проигрывает примерно в два поиску “Лесенкой” на небольшом количестве строк, однако с ростом строк разница между поисками сокращается (при втором типе генерации данных экспоненциальный поиск становится наиболее эффективным)

4.5. Из графика отношения времени работ алгоритмов, можно заметить, экспоненциальный поиск работает примерно в **4 раза** быстрее на втором типе генерации данных на размере матрицы **8192 x 8192,** чем на первом.

5. Вывод

5.1. Я написал три алгоритма на Java, вывел данные измерений в Excel файл и визуализировал их с помощью графиков.

5.2. Сравнил время работы этих алгоритмов на двух типах генерации данных.

5.3. Бинарный поиск стоит использовать при небольшом количестве строк, т.к при таких данных **M*log(N) < M + N** (например: **256*log(8192) < 256 + 8192**). При этом на больших данных эффективен поиск “Лесенкой” (**8192 * log(8912) = 107000** при бинарном поиске, а **8912 + 8192 = 16384** при поиске “Лесенкой”). Экспоненциальный поиск в целом работает чуть хуже поиска “Лесенкой”