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

2.1. Для создания матриц и тестирования алгоритмов был использован класс Matrix:

```java
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:

```java
    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:

```
76      public boolean binarySearch(){
77          for(int i = 0; i < rows; ++i){
78              int result = binSearch(i,  start: 0, finish: columns-1);
79              if(matrix[i][result] == key){
80                  return true;
81              }
82          }
83          return false;
84      }
85
            2 usages   YuriyShorin *
86      private int binSearch(int row, int start, int finish){
87          while (start <= finish){
88              int middle = (start+finish)/2;
89              if(matrix[row][middle] == key){
90                  return middle;
91              }
92              if(matrix[row][middle] > key){
93                  finish = middle - 1;
94              } else{
95                  start = middle + 1;
96              }
97          }
98          return finish;
99      }
```

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 строке:

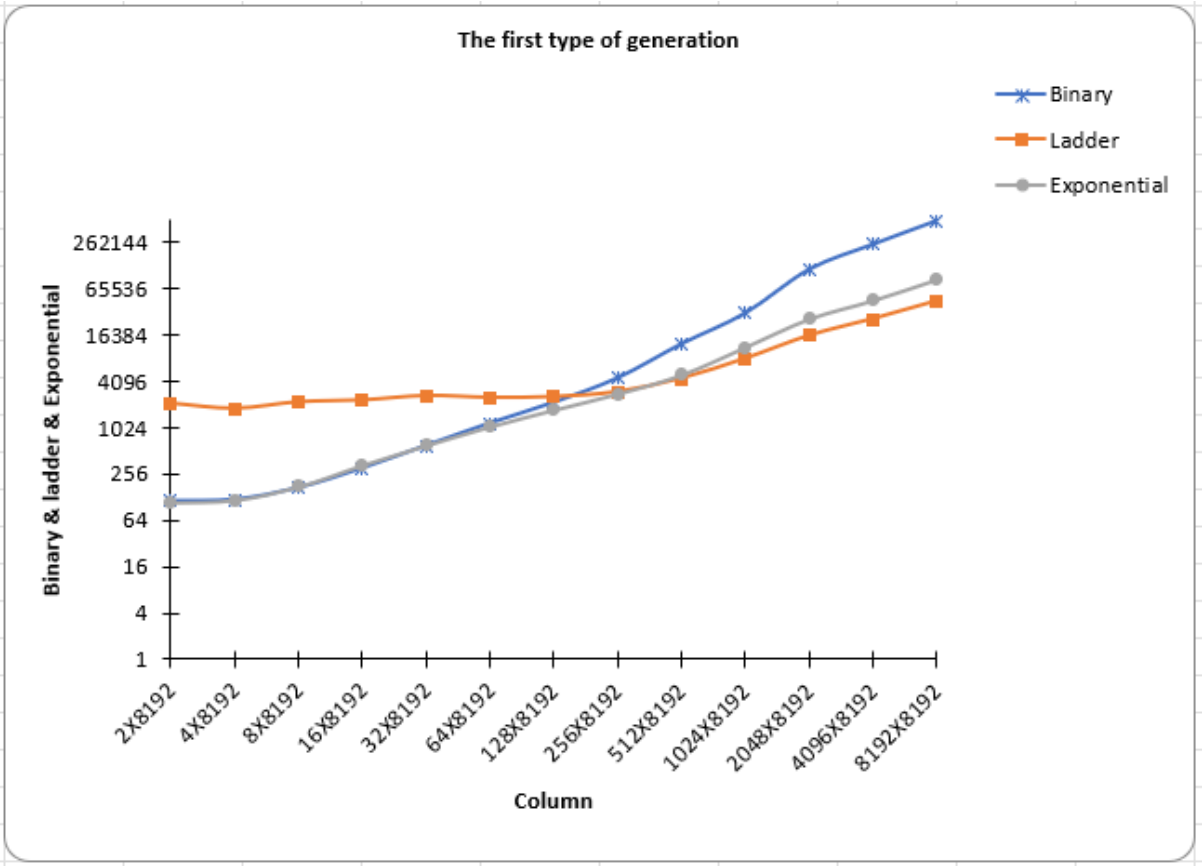
```java
public boolean ladderExpSearch(){
    int row = 0, column = columns - 1;
    while (row < rows){
        if(matrix[row][column] == key){
            return true;
        }
        else if(matrix[row][column] > key){
            int start = 1;
            while ((column - start > -1) && (matrix[row][column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

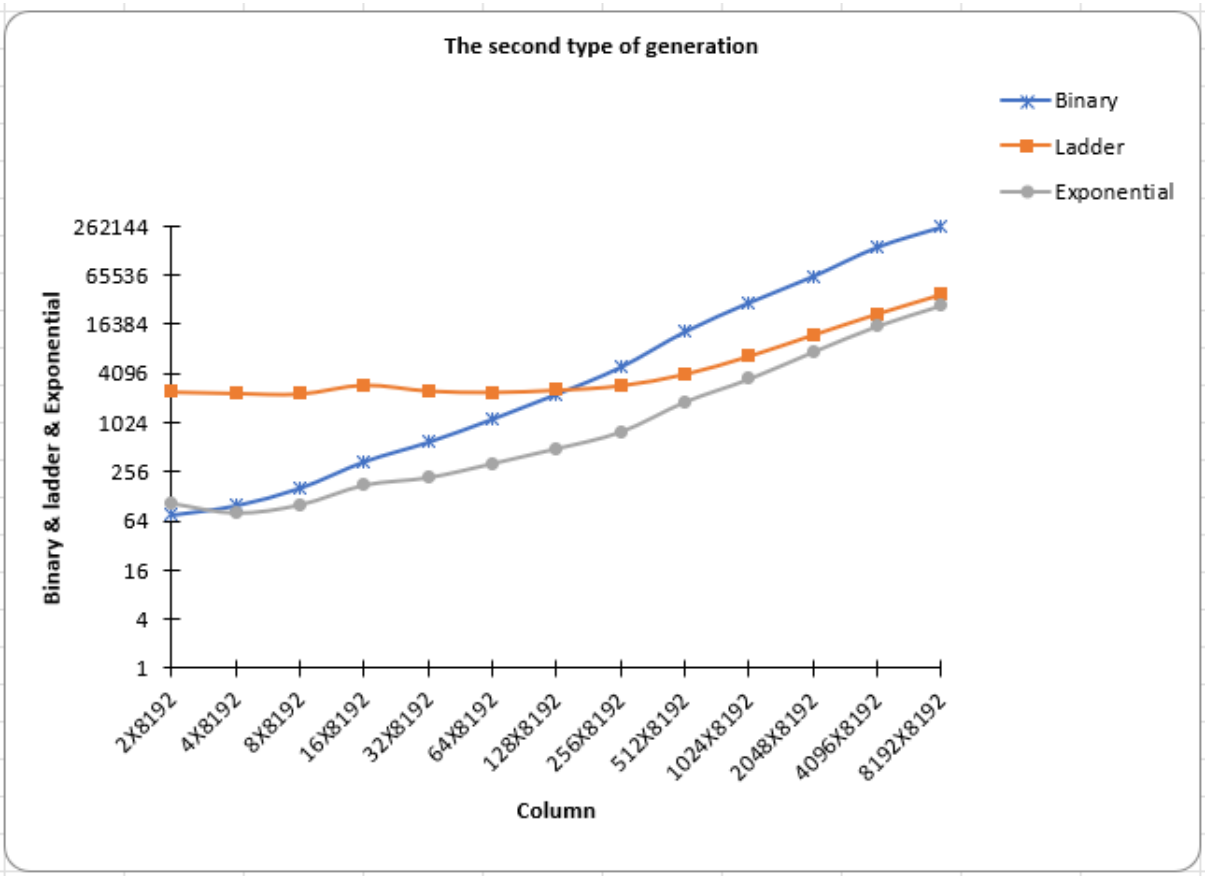
3. Визуализация в Excel

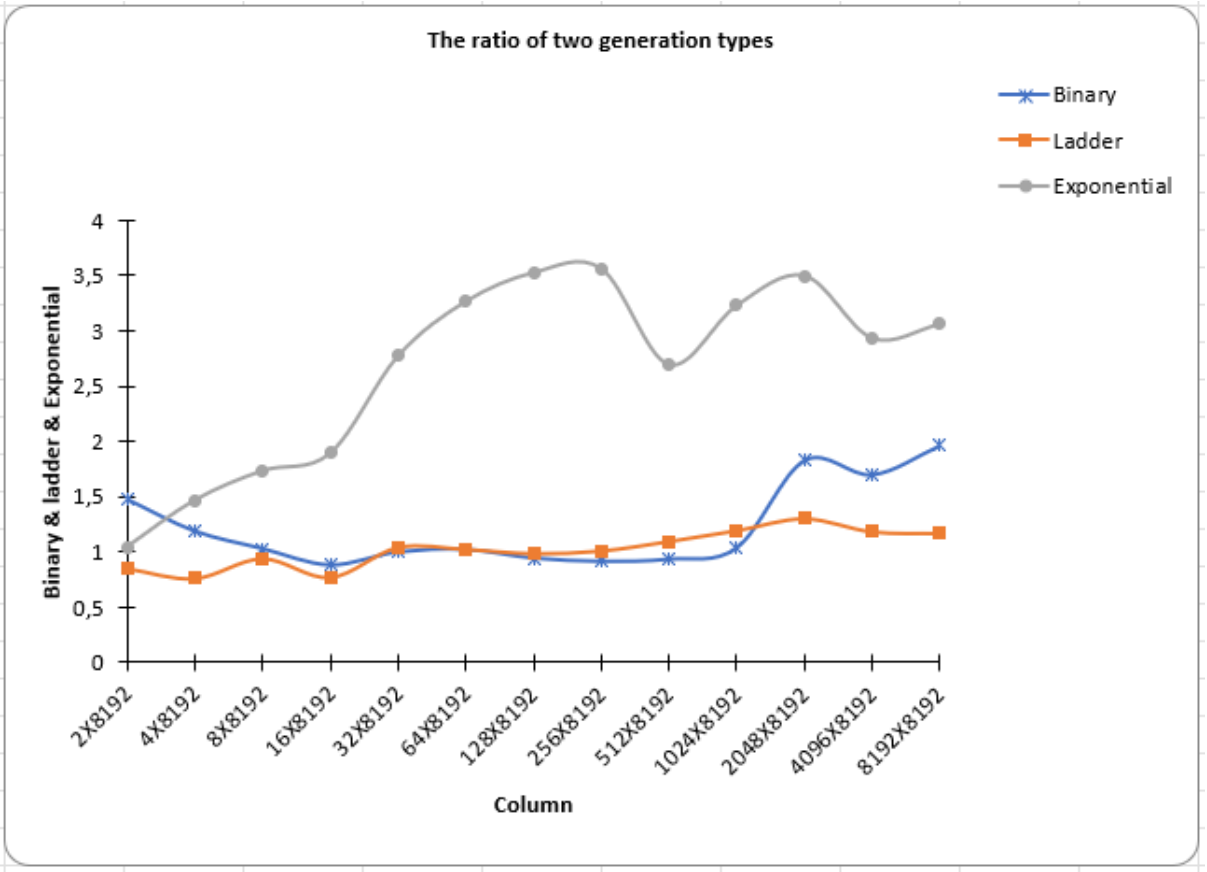
Статистика скорости работы алгоритмов автоматически заполняется в “Comparison of searches.xlsx”:

The first type of generation			
Matrix siz	Binary	Ladder	Exponential
2X8192	114	2102	111
4X8192	119	1809	119
8X8192	169	2196	177
16X8192	305	2319	337
32X8192	599	2647	610
64X8192	1172	2506	1068
128X8192	2182	2568	1760
256X8192	4535	2981	2847
512X8192	12566	4466	5018
1024X8192	31606	8088	11481
2048X8192	117017	16088	26713
4096X8192	247793	26501	46454
8192X8192	503020	45381	86680

The second type of generation			
Matrix siz	Binary	Ladder	Exponential
2X8192	77	2464	106
4X8192	100	2372	81
8X8192	164	2332	102
16X8192	346	3002	178
32X8192	599	2538	220
64X8192	1146	2443	327
128X8192	2312	2600	499
256X8192	4961	2946	799
512X8192	13462	4070	1862
1024X8192	30487	6771	3554
2048X8192	63869	12357	7634
4096X8192	145610	22387	15833
8192X8192	255824	38811	28317




The second type of generation
Binary
Ladder
Exponential
Binary & ladder & Exponential
262144
65536
16384
4096
1024
256
64
16
4
1
2X8192
4X8192
8X8192
16X8192
32X8192
64X8192
128X8192
256X8192
512X8192
1024X8192
2048X8192
4096X8192
8192X8192
Column


The ratio of two generation types
Binary
Ladder
Exponential
Binary & ladder & Exponential
4
3,5
3
2,5
2
1,5
1
0,5
0
2X8192
4X8192
8X8192
16X8192
32X8192
64X8192
128X8192
256X8192
512X8192
1024X8192
2048X8192
4096X8192
8192X8192
Column

4. Сравнение алгоритмов

4.1. На первом типе генерации данных:

Бинарный поиск работает практически одинаково по скорости с Экспоненциальным на небольшом количестве строк (до **256**), опережая в скорости поиск лесенкой.

При размере матрицы **256 x 8192** время работы поиска ‘Лесенкой’ становится равным времени работы бинарного и экспоненциального поисков.

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. Поиск ‘Лесенкой’ остается быстрее экспоненциального.

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

4.2. На втором типе генерации данных:

Бинарный поиск работает медленнее экспоненциального на любом размере матрицы.

При размере матрицы **256 x 8192** время работы поиска ‘Лесенкой’ становится меньше времени работы бинарного и экспоненциального поисков.

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

Экспоненциальный поиск работает быстрее поиска ‘Лесенкой’ на любом размере матрицы.

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 самый эффективный из трех алгоритмов, т.к. на

небольших данных работает примерно одинаково с бинарным поиском, а на больших и поиском ‘Лесенкой’.