Минобрнауки РФ

ФГБОУ ВО Тверской Государственный Технический Университет

Кафедра  “Программное Обеспечение”

Лабораторная работа N1

По дисциплине “Структуры и алгоритмы обработки данных”

Выполнил: студент группы Б.ПИН.РИС-22.06

Иванов Алексей Михайлович

Проверил: Мальков Александр Анатольевич

г. Тверь, 2023

Оглавление

[**Часть 1. Постановка задачи.** 3](#_Toc145406338)

[**Часть 2. Алгоритм решения.** 3](#_Toc145406339)

[**Часть 3. Оценка временной эффективности в терминах О-функций.** 4](#_Toc145406340)

[**Часть 4. Программный код.** 5](#_Toc145406341)

[**Часть 5. Тестирование и результаты работы.** 9](#_Toc145406342)

# **Часть 1. Постановка задачи.**

Работа с симметричными матрицами:

1. Преобразование индексов матрицы в индекс вектора;

2. Формирование вектора и запись в него элементов верхнего треугольника исходной матрицы (операция упаковки);

3. Получение значений элементов матрицы из ее упакованного представления (операция распаковки).

# **Часть 2. Алгоритм решения.**

Один из основных способов хранения разреженных матриц заключается в запоминании ненулевых элементов в одномерном массиве и идентификации каждого элемента массива индексами строки и столбца, как на рис. 3.5, а.

Доступ к элементу массива A с индексами *I* и *j* выполняется выборкой индекса *i*из вектора ROW, индекса *j* из вектора COLUMN и значения элемента из вектора A. Слева указан индекс *k* векторов, наибольшее значение которого определяется количеством нефоновых элементов. Отметим, что элементы массива обязательно запоминаются в порядке возрастания номеров строк.

Более эффективное представление с точки зрения требований к памяти и времени доступа к строкам матрицы, показано на рис. 3.5,б. Вектор ROW уменьшен, количество его элементов соответствует числу строк исходного массива A, содержащих нефоновые элементы. Этот вектор получен из вектора ROW рис. 3.5,а так, что его *i*-й элемент является индексом *k* для первого нефонового элемента *i*-й строки.

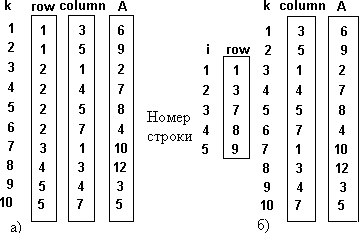


Рис. 3. 5. Последовательное представление разреженных матриц

Представление матрицы А, данное на рис. 3.5, сокращает требования к объему памяти более чем в 2 раза.

# **Часть 3. Оценка временной эффективности в терминах О-функций.**

Составим сводную таблицу эффективности для матриц в обоих типах представлений.

|  |  |  |
| --- | --- | --- |
|  | Классический вид | Последовательное представление |
| Вставка по индексу | O(1) | O |
| Замена по индексу | О(1) | O |
| Удаление по индексу | O(1) | O |
| Получение по индексу | O(1) | O |
| Память | O() | O() |

k – коэффициент наполненности.

Классический вид матрицы предполагает мгновенный доступ ко всем элементам с любой целью при известном индексе. Однако это требует хранения абсолютно всех элементов, даже фоновых.

При последовательном представлении все-по другому.

Для удаления и получения элемента необходимо его найти в векторе строк и столбцов. В векторе строк можно получить его практически мгновенно, за О(1), так как строка известна. Но далее необходимо пройтись по вектору столбцов в поисках необходимого.

Матрица в среднем заполнена равномерно. Пусть матрица имеет размер n\*n и содержит p нефоновых элементов. Тогда k= p / n^2. В среднем на каждой строке содержится p/n элементов. В среднем придется перебрать половину из них, и тогда итоговое время поиска, а значит и удаления, и получения (так как непосредственно сами операции выполняются почти мгновенно, за О(1)), займут O(p/2n) = O(kn/2)=O(kn).

Для вставки по индексу также необходимо найти потенциальное место для нового элемента, но вместе с этим придется сдвигать в среднем половину элементов вектора столбцов и вектора элементов вниз, а также смещать в среднем половину индексов строк в векторе строк. Итого, эта операция займет О(kn) + O(p/2) + O(p/2) + O(n/2) = O(kn^2).

Однако стоит заметить, что замена элемента зависит лишь от времени поиска, так что для нее имеем сложность по времени O(kn).

Таким образом, данная структура хранения матриц эффективна при большом количестве элементов и высокой разреженности их. Например, для матрицы 10000\*10000, заполненной некими данными при коэффициенте заполненности k=0.1 (примем вес одного элемента за 8 бит), выигрыш в памяти при последовательном хранении составит порядка 700 Мбайт.

# **Часть 4. Программный код.**

Итоговая библиотека классов имеет следующий вид:

/// <summary>

/// Класс, реализующий квадратную матрицу с данными определнного типа

/// </summary>

/// <typeparam name="T">Тип данных, хранимых в матрице</typeparam>

class Matrix<T>

{

/// <summary>

/// Ребро матрицы

/// </summary>

private int size = 0;

/// <summary>

/// Двумерный массив, реализующий матрицу

/// </summary>

private T[,] matrix = { };

/// <summary>

/// Конструктор, создающий пустую матрицу данного размера

/// </summary>

/// <param name="size">Размер матрицы</param>

public Matrix (int size)

{

this.size = size;

this.matrix = new T[size, size];

}

/// <summary>

/// Конструктор, создающий экземпляр класса на основе

/// двумерного квадратного массива

/// </summary>

/// <param name="matrix">Двумерный массив, оборажающий матрицу</param>

public Matrix(T[,] matrix)

{

this.size = matrix.GetLength(0);

this.matrix = matrix;

}

/// <summary>

/// Возвращает размер матрицы

/// </summary>

/// <returns>Размер матрицы</returns>

public int getSize()

{

return size;

}

/// <summary>

/// Назначает элемент в данную позицию

/// </summary>

/// <param name="row">Номер ряда вставляемого элемента</param>

/// <param name="col">Номер столбца вставляемого элемента</param>

/// <param name="elem">Вставляемый элемент</param>

public void setElem(int row, int col, T elem)

{

matrix[row, col] = elem;

}

/// <summary>

/// Удаляет элемент по данной позиции

/// </summary>

/// <param name="row">Номер ряда удаляемого элемента</param>

/// <param name="col">Номер столбца удаляемого элемента</param>

public void delete(int row, int col)

{

matrix[row, col] = default(T);

}

/// <summary>

/// Возращает элемент на данной позиции

/// </summary>

/// <param name="row">Номер ряда получаемого элемента</param>

/// <param name="col">Номер столбца получаемого элемента</param>

/// <returns>Элемент на данной позиции</returns>

public T getElem(int row, int col)

{

return matrix[row, col];

}

/// <summary>

/// Возращает верхний треугольник матрицы

/// </summary>

/// <returns>Верхний треугольник матрицы</returns>

public T[][] getUpperTriagle()

{

T[][] upperT = new T[size][];

for (int i = 0; i < size; i++)

{

T[] temp = new T[size - i];

for (int j = i; j < size; j++)

{

temp[j - i] = matrix[i, j];

}

upperT[i] = temp;

}

return upperT;

}

/// <summary>

/// Возвращает вектор из элементов матрицы

/// </summary>

/// <returns>Вектор из элементов матрицы</returns>

public T[] getVector()

{

return Matrix<T>.GetVector(this.getUpperTriagle());

}

/// <summary>

/// Возвращает упакованную матрицу

/// </summary>

/// <returns>Упаковываемая матрицы</returns>

public int[][] box()

{

return Matrix<T>.Box(this.getUpperTriagle());

}

/// <summary>

/// Возвращает вектор от произвольного массива массивов

/// </summary>

/// <param name="matrix">Матрица</param>

/// <returns>Вектор от произвольной матрицы</returns>

public static T[] GetVector(T[][] matrix)

{

List<T> list = new List<T>();

for (int i = 0; i < matrix.GetLength(0); i++)

for (int j = 0; j < matrix[i].GetLength(0); j++)

if (!matrix[i][j].Equals(default(T) ) )

list.Add(matrix[i][j]);

return list.ToArray();

}

/// <summary>

/// Упаковывает произвольный массив массивов

/// </summary>

/// <param name="matrix">Произвольный массив массивов</param>

/// <returns>Упакованный массив массивов</returns>

public static int[][] Box(T[][] matrix)

{

List<int> boxedCols = new List<int>();

List<int> boxedRows = new List<int>();

int rows = 0, cols = 0;

int notNullElems = 0;

for (int i = 0; i < matrix.Length; i++)

{

boxedRows.Add(notNullElems);

for (int j = 0; j < matrix[i].Length; j++)

{

if (!matrix[i][j].Equals(default(T)))

{

boxedCols.Add(j);

notNullElems++;

}

}

}

return new int[][] { boxedRows.ToArray(), boxedCols.ToArray() };

}

}

/// <summary>

/// Класс, отображающий упакованную квадратную симметричную матрицу

/// </summary>

/// <typeparam name="T">Тип элементов в упакованной матрице</typeparam>

class BoxedMatrix<T>

{

/// <summary>

/// Вектор упакованной матрицы

/// </summary>

private T[] vector;

/// <summary>

/// Массивы индексов упанованной матрицы

/// </summary>

private int[][] indexes;

/// <summary>

/// Размер упакованной матрицы

/// </summary>

private int size;

/// <summary>

/// Размер упакованной матрицы

/// </summary>

public int Size { get { return this.size; } }

/// <summary>

/// Конструктор, создающий объект на основе вектора матрицы

/// и массивов индексов

/// </summary>

/// <param name="vector">Вектор матрицы</param>

/// <param name="indexes">Массив индексов</param>

public BoxedMatrix(T[] vector, int[][] indexes) {

this.vector = vector;

this.indexes = indexes;

this.size = indexes[0].Length;

}

/// <summary>

/// Конструктор, создающий объект на основе двумерного массива

/// представляющего собой матрицу

/// </summary>

/// <param name="matrix">Двумерный массив - матрица</param>

public BoxedMatrix(T[,] matrix)

{

Matrix<T> tempMat = new Matrix<T> (matrix);

this.vector = tempMat.getVector();

this.indexes = tempMat.box();

this.size = tempMat.getSize();

}

/// <summary>

/// Конструктор, создающий объект на основе матрицы

/// </summary>

/// <param name="matrix">Матрица</param>

public BoxedMatrix(Matrix<T> matrix)

{

this.vector = matrix.getVector();

this.indexes = matrix.box();

this.size = matrix.getSize();

}

/// <summary>

/// Конструктор, создающий новую пустую матрицу длины 0

/// </summary>

public BoxedMatrix()

{

this.vector = new T[0];

this.indexes = new int[][] { };

this.size = 0;

}

/// <summary>

/// Определяет положение элемента в верхнем треугольнике

/// на основе данного положения в полной матрице

/// </summary>

private (int , int) defineRowColInUpper(int factrow, int factcol)

{

int row, col;

if (factcol >= factrow)

(row, col) = (factrow, factcol);

else

(row, col) = (factcol, factrow);

return (row, col - row);

}

/// <summary>

/// Находит позицию в векторе элемента по

/// его позиции в цельной матрице

/// </summary>

/// <param name="factRow">Строка элемента</param>

/// <param name="factCol">Столбец элемента</param>

/// <returns>Индекс элемента в векторе</returns>

private int findInVector(int factRow, int factCol)

{

int row, col;

(row, col) = defineRowColInUpper(factRow, factCol);

if (row + 1 < indexes[0].Length)

{

for (int i = indexes[0][row]; i < indexes[0][row + 1]; i++)

{

if (indexes[1][i] == col)

return i;

}

} else

{

for (int i = indexes[0][row]; i < indexes[1].Length; i++)

{

if (indexes[1][i] == col)

return i;

}

}

return -1;

}

/// <summary>

/// Возвращает элемент по данной позиции

/// </summary>

/// <param name="factRow">Номер строки элемента</param>

/// <param name="factCol">Номер столбца элемента</param>

/// <returns>Элемент на данной позиции</returns>

public T getElem(int factRow, int factCol)

{

int index = findInVector(factRow, factCol);

if (index == -1)

return default(T);

else

return vector[index];

}

/// <summary>

/// Удаляет элемент по данной позиции

/// </summary>

/// <param name="factRow">Номер строки элемента</param>

/// <param name="factCol">Номер столбца элемента</param>

public void deleteElem(int factRow, int factCol)

{

vector[findInVector(factRow, factCol)] = default(T);

}

/// <summary>

/// Назначает элемент на данную позицию

/// </summary>

/// <param name="factRow">Номер строки элемента</param>

/// <param name="factCol">Номер столбца элемента</param>

/// <param name="elem">Назначаемый элемент</param>

public void setElem(int factRow, int factCol, T elem)

{

int row, col;

(row, col) = defineRowColInUpper(factRow, factCol);

//replacement

int pos = findInVector(row, col);

if ( pos != -1)

{

vector[pos] = elem;

return;

}

//insertion in col

for (int i = indexes[0][row]; i < indexes[0][row + 1]; i++)

{

if (i == indexes[0][row + 1] - 1 || indexes[1][i+1] >= col)

{

List<int> newCols = new List<int>(indexes[1]);

newCols.Insert(i, col);

List<T> newVector = new List<T> {};

foreach(T val in vector)

newVector.Add(val);

newVector.Insert(i, elem);

indexes[1] = newCols.ToArray();

vector = newVector.ToArray();

break;

}

}

for (int i = row; i < size; i++)

indexes[0][i]++;

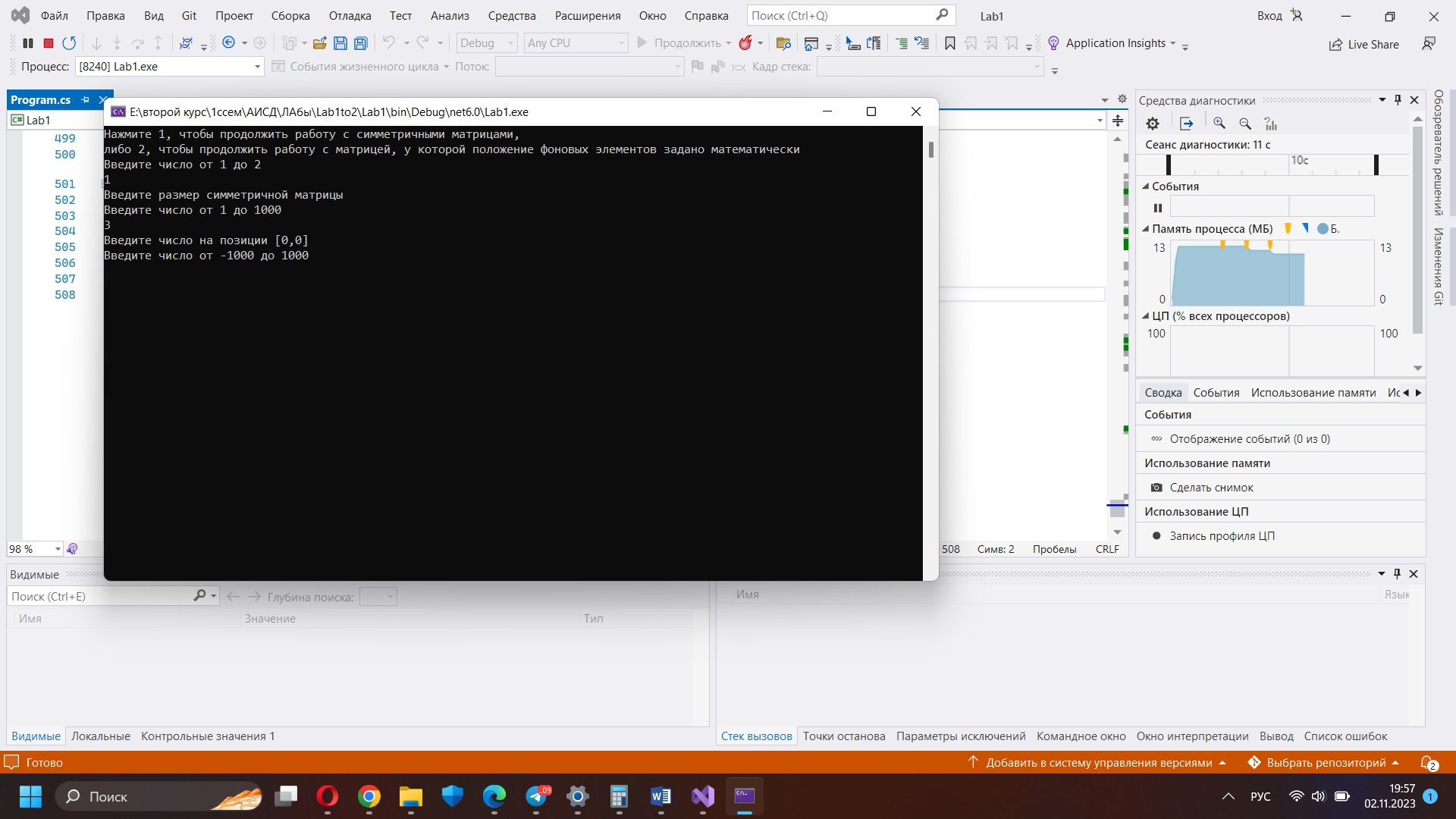
}

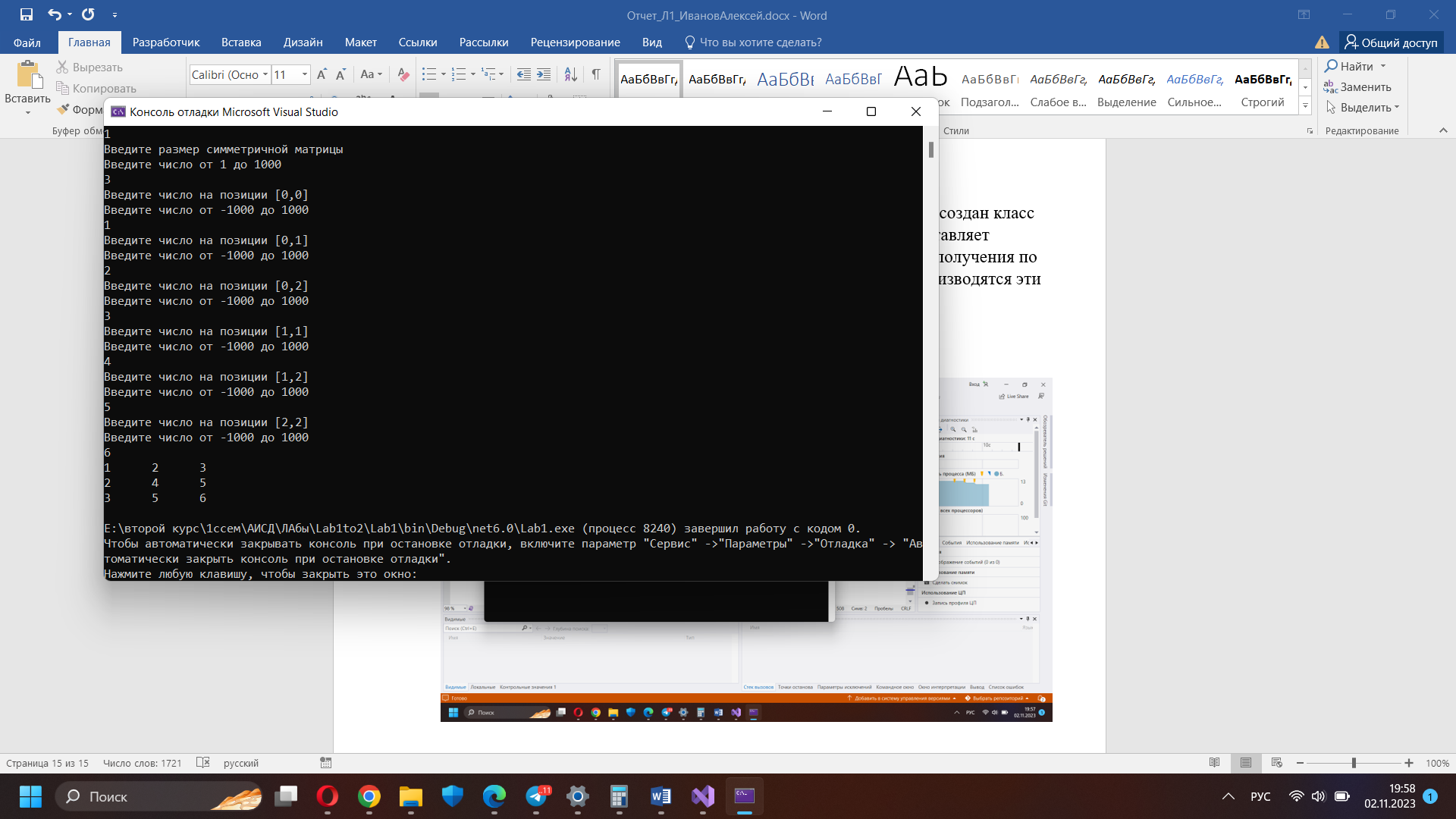
}

Класс Matrix<T> позволяет работать с произвольными квадратыми матрицами, храня данные в классическом виде, предоставляя возможность вставки нового элемента, замены и поиска по индексу. Но он является скорее вспомогательным и создан, чтобы не смешивать работу с обычными матрицами и с сжатыми. Для перехода к сжатой матрице созданы функции, позволяющие получить верхний треугольник, а также вектор индексов элементов матрицы.

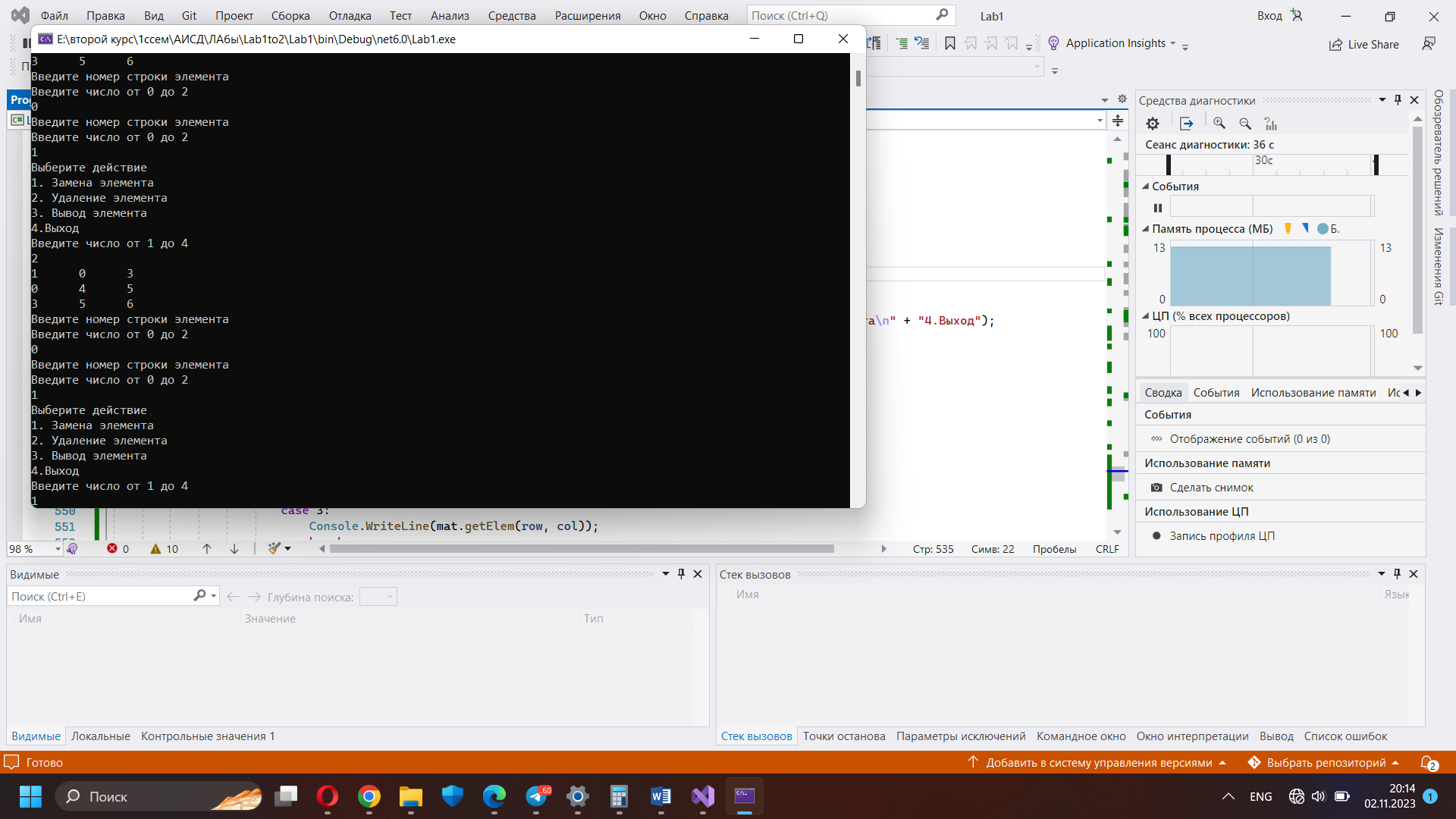
Для работы с матрицами в последовательном представлении был создан класс BoxedMatrix<T>, позволяющий осуществить переход. Он предоставляет функции установки элемента по индексу, удаления по индексу и получения по индексу. Класс скрывает от пользователя способы, которыми производятся эти действия, соответствуя принципу инкапсуляции.

# **Часть 5. Тестирование и результаты работы.**

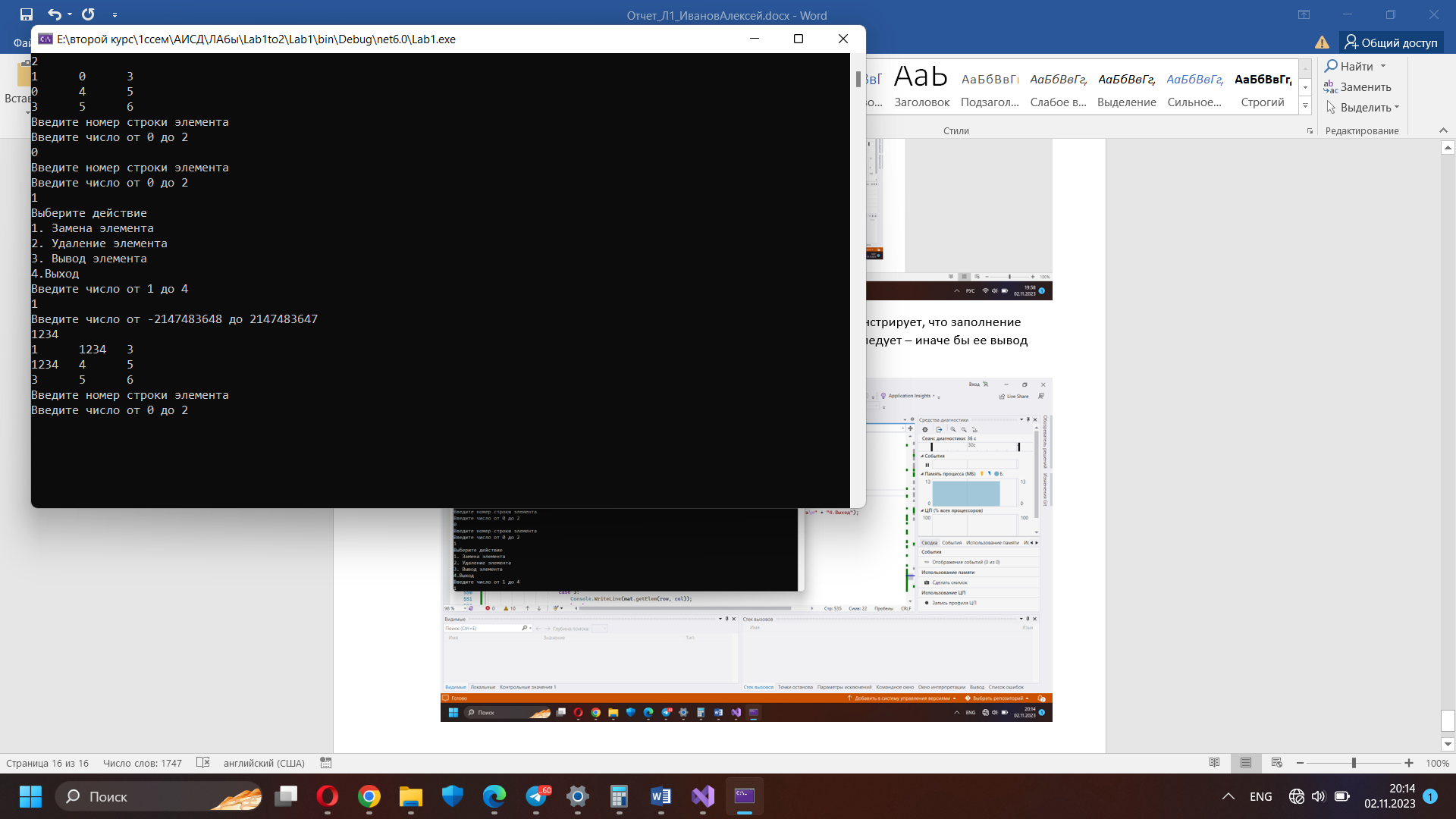




Для тестирования было построено консольное приложение. Оно демонстрирует, что заполнение матрицы идет корректно, а также получение элементов работает как следует – иначе бы ее вывод был невозможен.



Данный пример показывает, что при удалении элемента (1;0) он удаляется в обоих частях матрицы.



Данный пример показывает, что при замене элемента (1;0) на 1234 он меняется в обоих частях матрицы.