Отчет по 3 лабораторной работе

По дисциплине «Типы и структуры данных»

Подготовила Мищенко Маргарита

Группа ИУ7-35Б

Вариант 5

Условие задачи: техническое задание

Разработать программу умножения или сложения разреженных матриц.  
Предусмотреть возможность ввода данных, как с клавиатуры, так и использования  
заранее подготовленных данных. Матрицы хранятся и выводятся в форме трех  
объектов. Для небольших матриц можно дополнительно вывести матрицу в виде  
матрицы. Величина матриц - любая (допустим, 1000 \* 1000). Сравнить эффективность  
(по памяти и по времени выполнения) стандартных алгоритмов обработки матриц с  
алгоритмами обработки разреженных матриц при различной степени разреженности  
матриц и различной размерности матриц .

Задача

Разреженная (содержащая много нулей) матрица хранится в форме 3-х объектов:  
 - вектор A содержит значения ненулевых элементов;  
 - вектор JA содержит номера столбцов для элементов вектора A;  
 - связный список IA, в элементе Nk которого находится номер компонент  
в A и JA, с которых начинается описание строки Nk матрицы A.

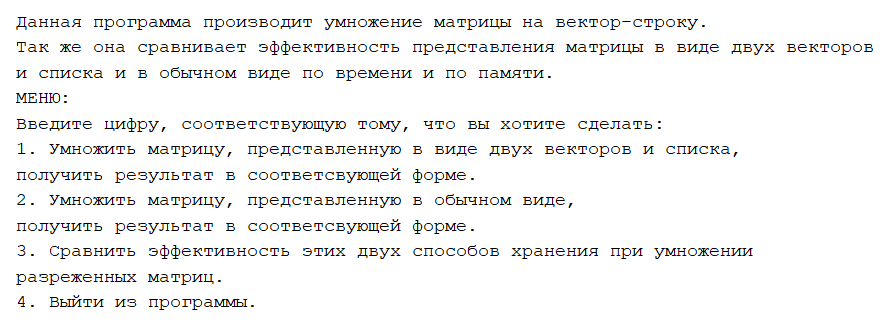
1. Смоделировать операцию умножения вектора-строки и матрицы,  
хранящихся в этой форме, с получением результата в той же форме.

2. Произвести операцию умножения, применяя стандартный алгоритм работы  
с матрицами.

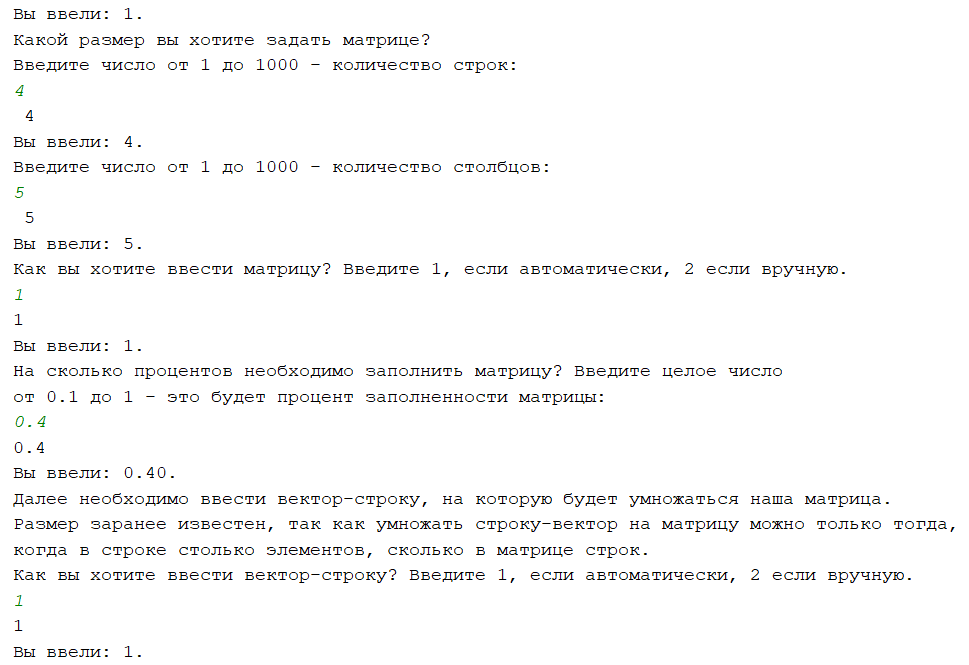
3. Сравнить время выполнения операций и объем памяти при использовании  
этих 2-х алгоритмов при различном проценте заполнения матриц.

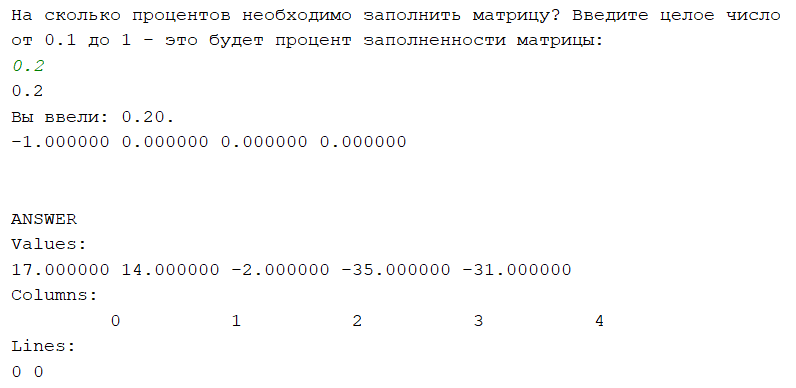
Входные и выходные данные

При запуске программы появляется меню, которое поясняет работу программы и предлагает пользователю выбрать один из 4х вариантов работы:

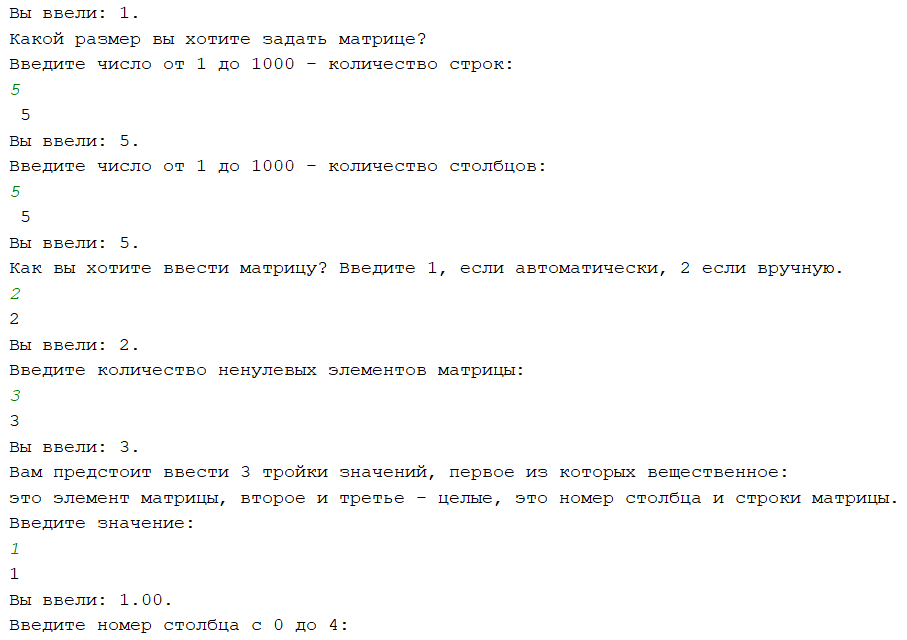
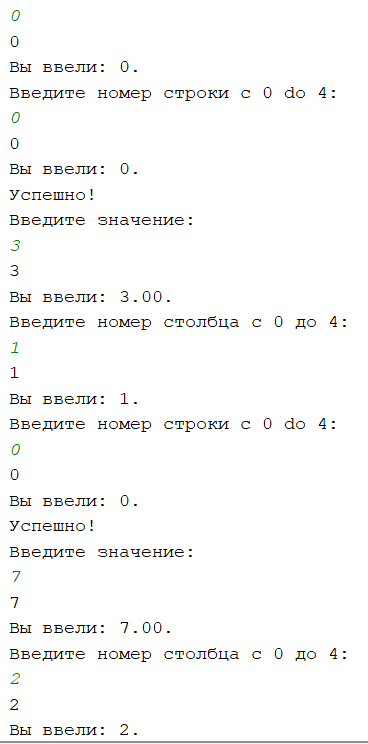


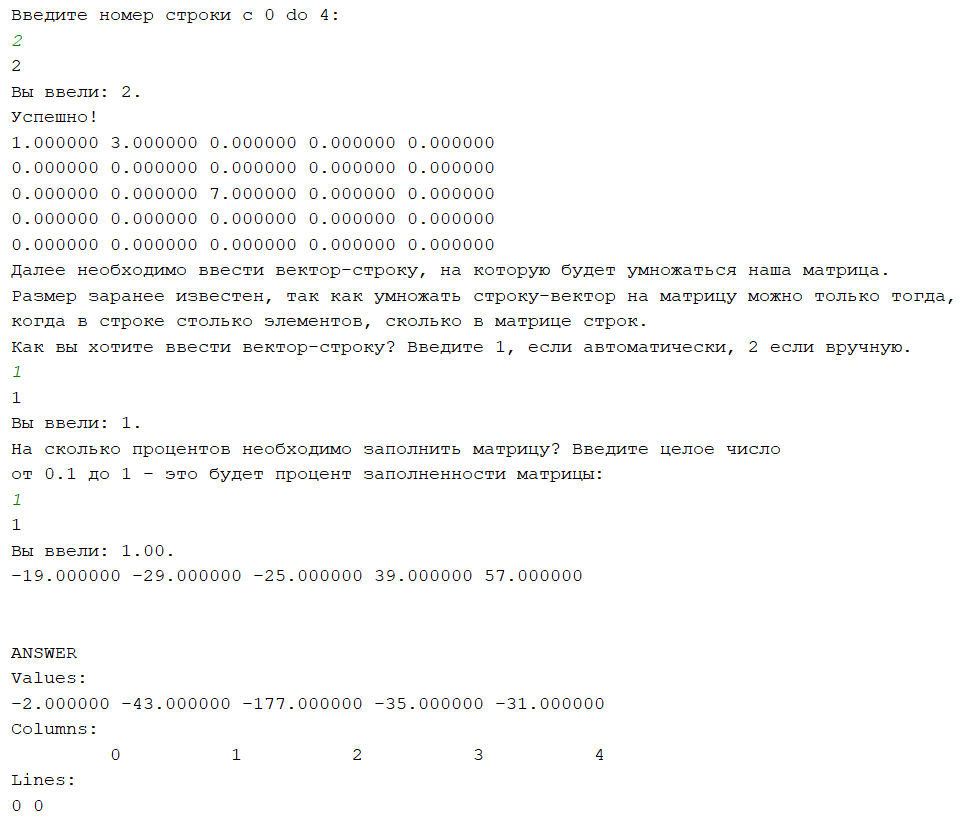
При выборе цифры 1:

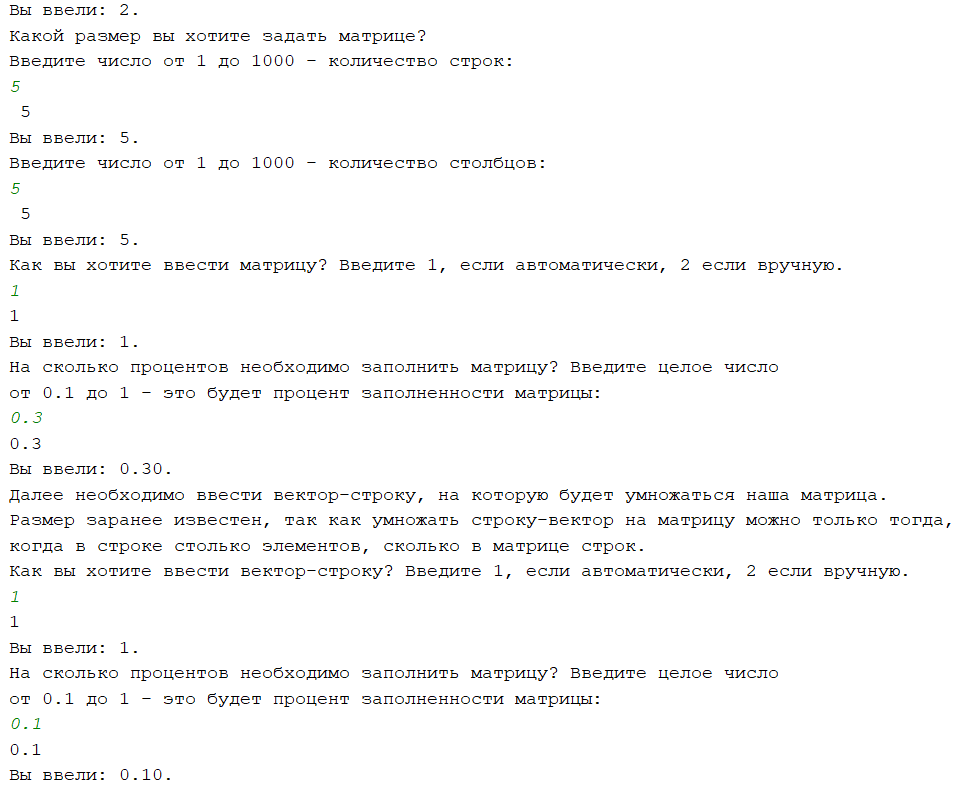


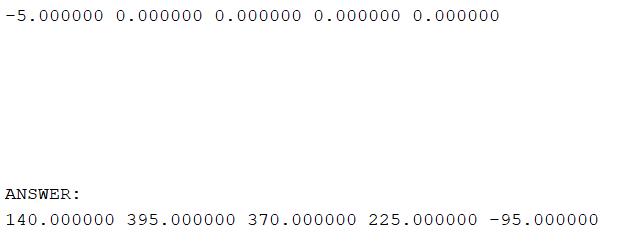


Также пользователь может ввести все сам:

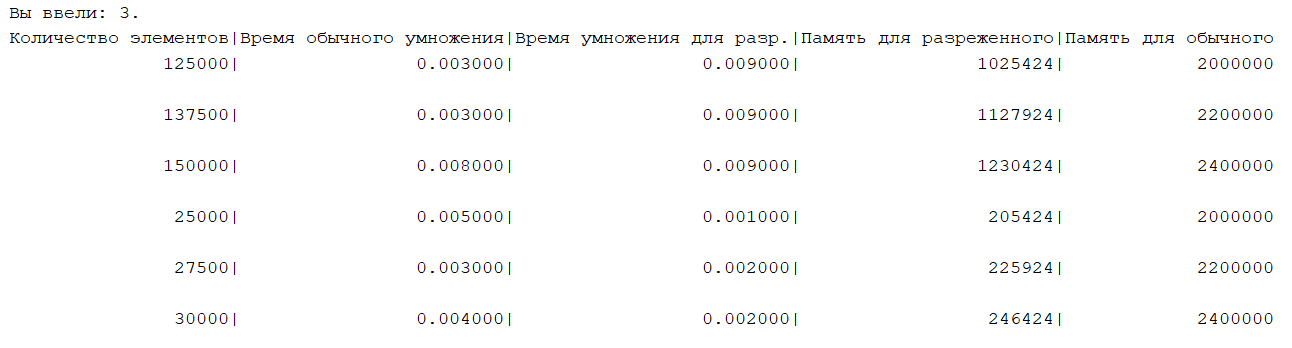
 



При выборе в меню цифры 2 (здесь так же пользователь может ввести все сам): 



При выборе цифры 3:



Функции программы

**float** rounding(**double** num, **long** count);

-данная функция округляет свой первый элемент до количества знаков после запятой, хранящегося во втором аргументе.

**void** scan\_int(**int** \*num);

-данная функция безопасно считывает целое число.

**int** auto\_input(**float** \*values, **int** \*columns, list\_t \*\*lines, **int** line, **int** column, **float** proc, **int** \*not\_null\_lines);

-данная функция производит автоматическое создание матрицы в виде двух векторов и связанного списка.

**void** scan\_float(**float** \*num);

-данная функция безопасно считывает float.

**int** hand\_input\_mtr(**float** \*values, **int** \*columns, list\_t \*\*lines, **int** line, **int** column, **int** \*not\_null\_lines);

-данная функция производит ручное (когда все вбивается пользователем) создание матрицы в виде двух векторов и связанного списка.  
  
**int** hand\_input\_str(**float** \*values, **int** \*columns, list\_t \*\*lines, **int** line, **int** column);

-данная функция производит ручное (когда все вбивается пользователем) создание строки-вектора в виде двух векторов и связанного списка.

**void** print\_from\_tail(**const** list\_t \*list) ;

-данная функция печатает связный список с хвоста.

**void** print\_from\_head(**const** list\_t \*list) ;

-данная функция печатает связный список с головы.

**void** push(list\_t\*\* list, **int** nk, **int** i);

-данная функция добавляет элемент в связный список.

**void** print\_mtr(**float** \*\*mtr, **int** line, **int** column);

-данная функция выводит матрицу.

**void** print\_array\_i(**int** \*array, **int** num);

-данная функция выводит массив целых чисел.

**void** print\_array\_f(**float** \*array, **int** num);

-данная функция выводит массив вещественных чисел.

**int** mult\_sparse\_mtr(list\_t \*lines\_mtr, **const int** \*columns\_mtr, **const float** \*values\_mtr,  
 **const int** \*columns\_str, **const float** \*values\_str,  
 list\_t \*\*lines\_res, **int** \*c\_res, **float** \*res,  
 **int** elems, **int** elems\_str, **int** column, **int** not\_null\_lines);

-данная функция производит умножение вектора-строки на разреженную матрицу, оба объекта представлены в виде двух векторов и односвязного списка.  
  
**float** \*\*making\_ord\_mtr(**float** \*\*mtr, list\_t \*lines\_mtr, **const int** \*columns\_mtr, **const float** \*values\_mtr,  
 **int** elems, **int** line, **int** column, **int** not\_null\_lines);

-данная функция создает из двух векторов и односвязного списка обычную матрицу в виде двумерного массива.

**float** \*making\_ord\_vec(**float** \*vec,**const int** \*columns\_vec, **const float** \*values\_vec, **int** elems);

-данная функция создает из двух векторов и односвязного списка обычную вектор-строку в виде массива.

**float** \*mult\_ord\_mtr(**float** \*\*mtr, **const float** \*str, **int** line, **int** column, **float** \*str\_res);

-данная функция умножает обычную вектор-строку на обычную матрицу.

**void** sparse\_mult();

-данная функция подготавливает входные аргументы для умножения (два вектора и список).

**void** ord\_mult();

-данная функция подготавливает входные аргументы для умножения (обычное представление).

**void** check\_efficiency();

-данная функция проверяет эффективность алгоритмов.

tm check(**int** line, **int** column, **float** proc, **int** \*num\_elems\_res, **int** \*num\_elems, **int** \*num\_elems\_str,  
**int** \*not\_null\_lines)

-данная функция делает замеры времени.

**void** print\_result(tm time1, **int** line, **int** column, **int** num\_elem, **int** not\_null\_lines, **int** num\_str, **int** test1)

-данная функция печатает результат проверки эффективности.

Используемые структуры

**float** \*values = malloc((column \* line + 1) \* **sizeof**(**float**));

-в данном массиве представлены значения элементов матрицы, не равных нулю.

**int** \*columns= malloc((column \* line + 1) \* **sizeof**(**int**;

-в данном массиве представлены соответствующие этим элементам номера столбцов.

list\_t \*lines = **NULL**;

-связный список, хранящий индекс элемента из первых двух массивов, с которого начинается каждая непустая строка.  
  
**float** \*values\_str = malloc((line + 1) \* **sizeof**(**float**));

-в данном массиве представлены значения элементов вектора-строки, не равных нулю.

**int** \*columns\_str = malloc((line + 1) \* **sizeof**(**int**)), num\_elems\_str = 0;

-в данном массиве представлены соответствующие этим элементам номера столбцов.

list\_t \*lines\_str = **NULL**;

-связный список, хранящий индекс элемента из первых двух массивов, с которого начинается каждая непустая строка.

Данная структура представляет односвязный список. В первом ее элементе first\_in\_line хранится индекс элемента, с которого начинается строка под номером num\_line. Третий элемент данной структуры содержит указатель на следующий член односвязного списка.

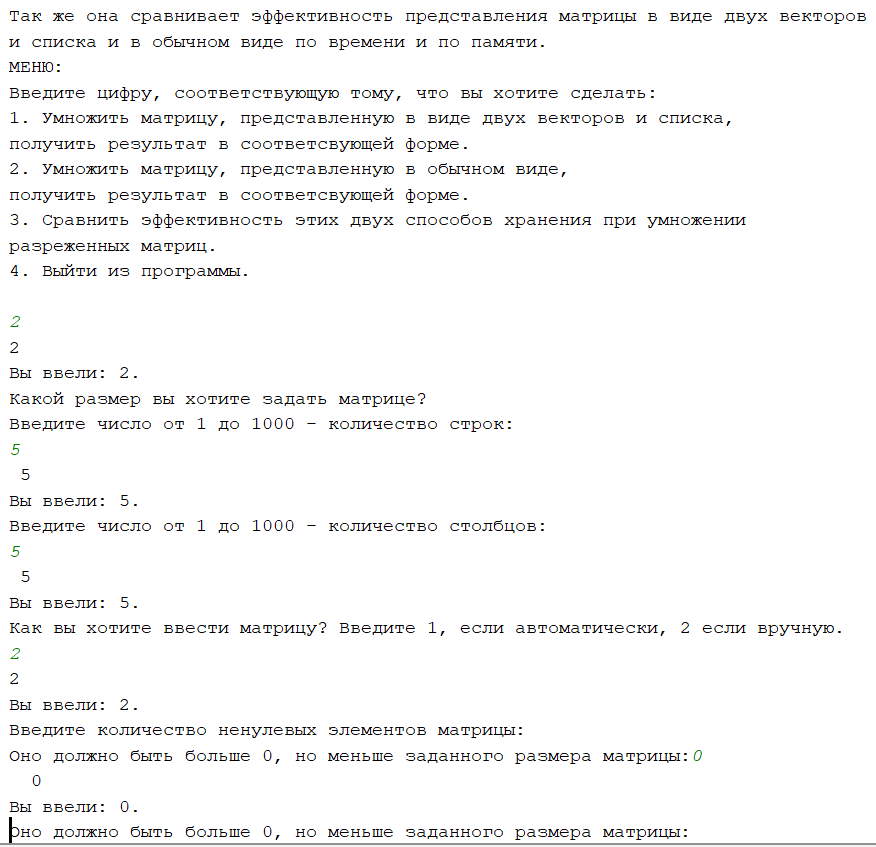
**typedef struct** line  
{  
 **int** first\_in\_line;  
 **int** num\_line;  
 **struct** line \*next;  
}list\_t;

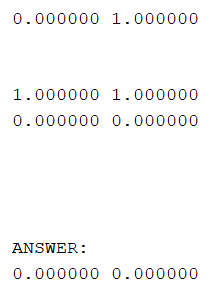
Данная структура нужна для возвращения из функции, делающей замеры времени для исследования эффективности, сразу двух параметров.

Первый параметр – время обработки обычной матрицы, второй параметр – время обработки матрицы, хранящейся в виде двух векторов и списка.  
  
**typedef struct** timee  
{  
 **double** time\_ord;  
 **double** time\_sparse;  
}tm;

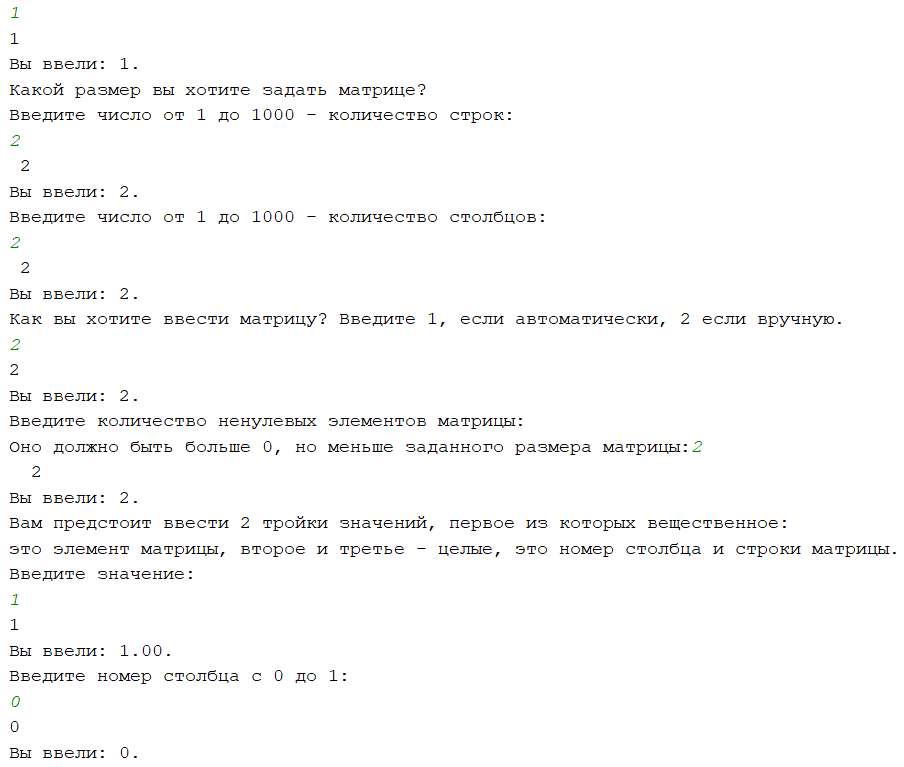
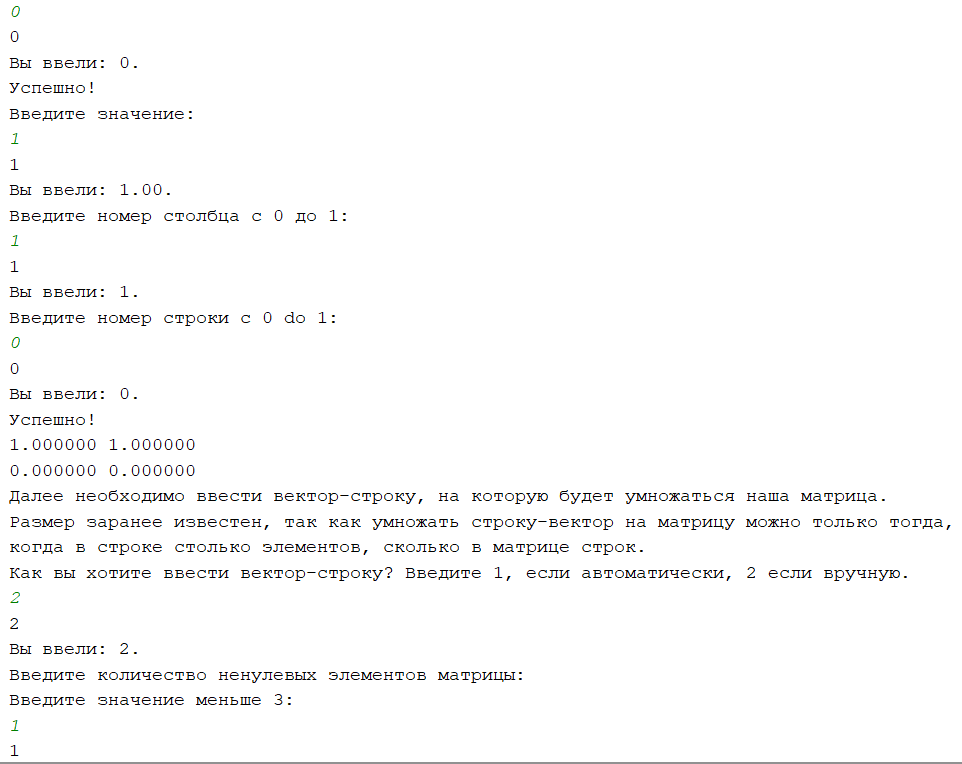
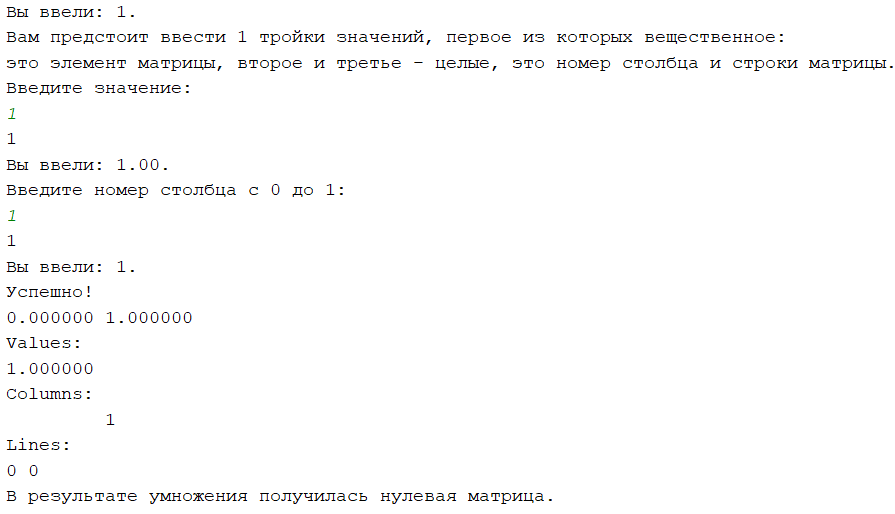
Аварийные ситуации

Программа защищена от ввода нулевого количества элементов матрицы или вектора-строки, в подобном случае она выводит повторное сообщения о требованиях к корректному вводу.



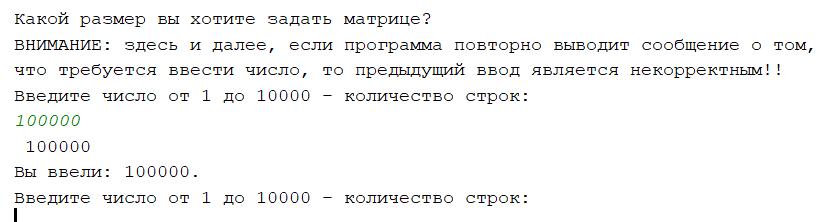
Нулевая матрица может получиться только при умножении стандартным способом: 

Если умножить те же матрицы другим способом:

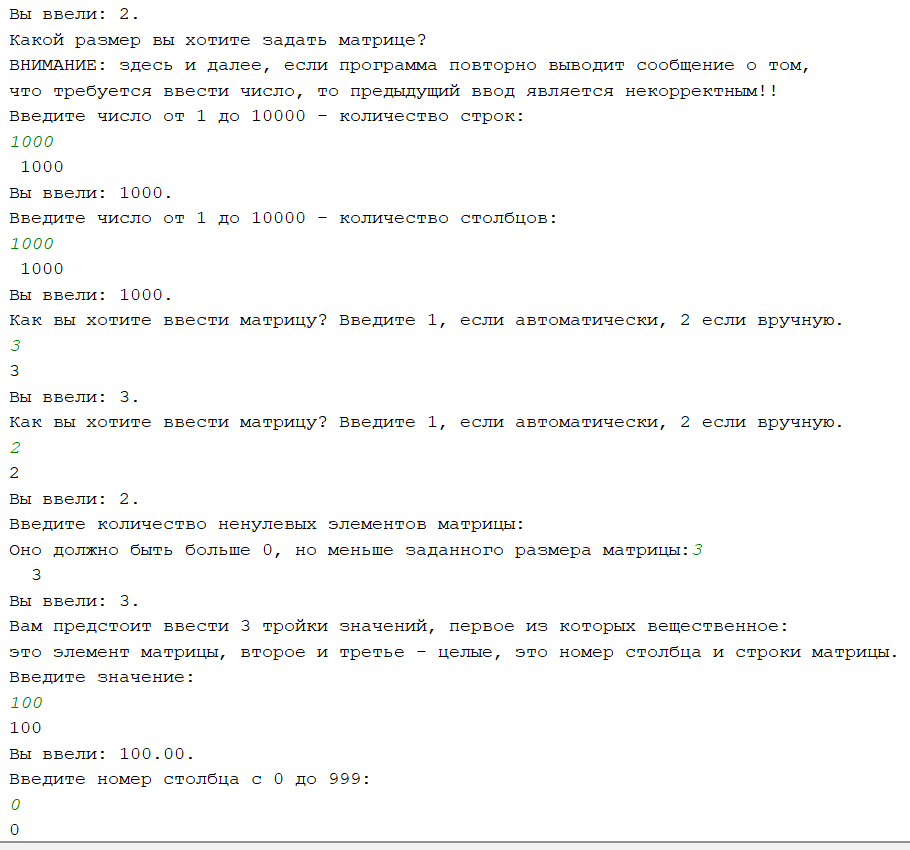
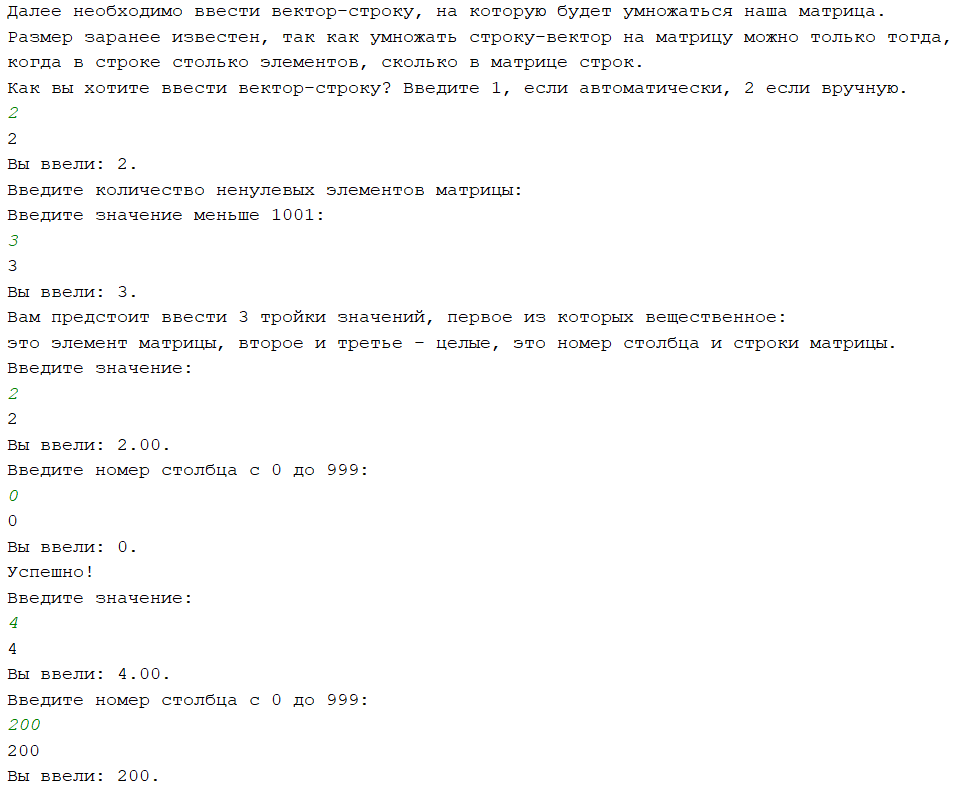
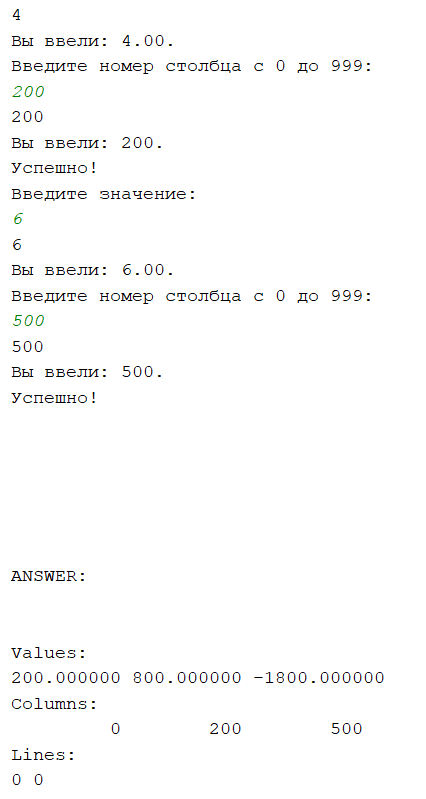
  

То есть выводить нечего, на экране появляется сообщение, что в результате умножения получилась нулевая матрица.

Примеры неверного ввода (программа игнорирует неверный ввод и требует ввести нужное значение еще раз, пока ввод не станет корректным, намекая на то, что должен ввести пользователь):

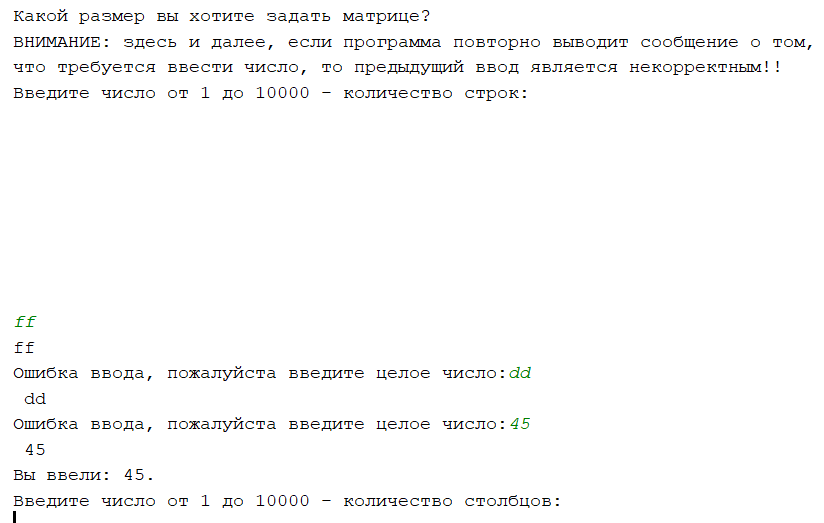


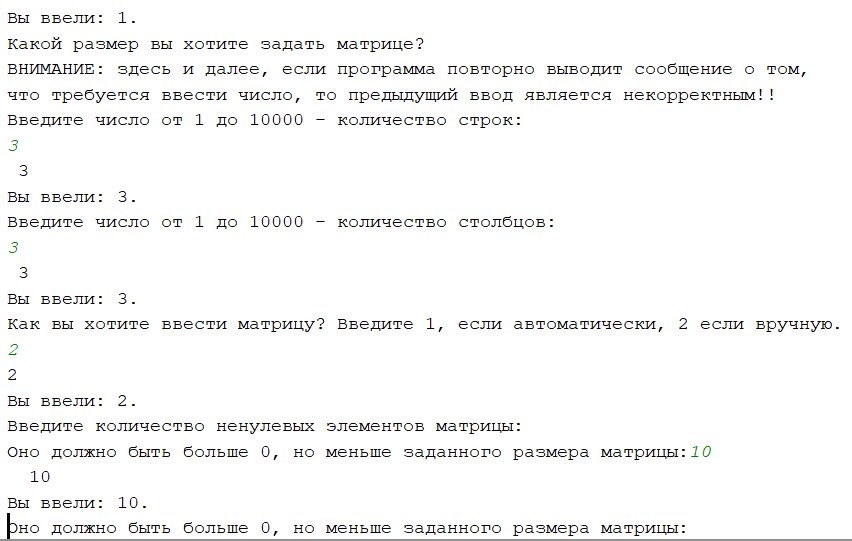
Возможные аварийные ситуации связаны с обработкой больших размеров матриц:

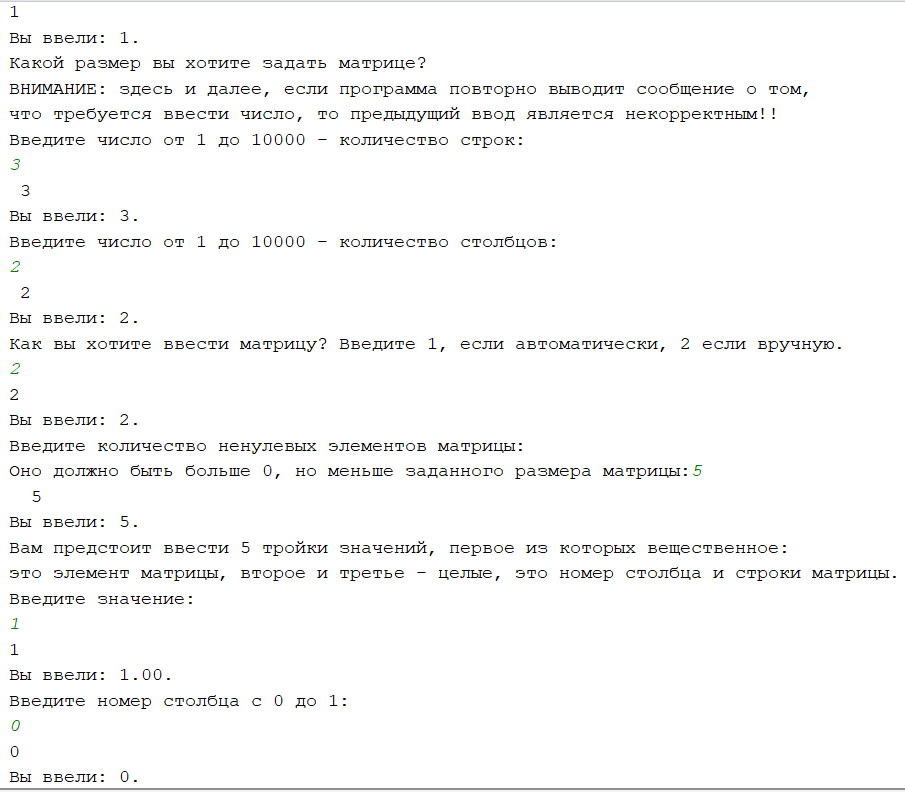
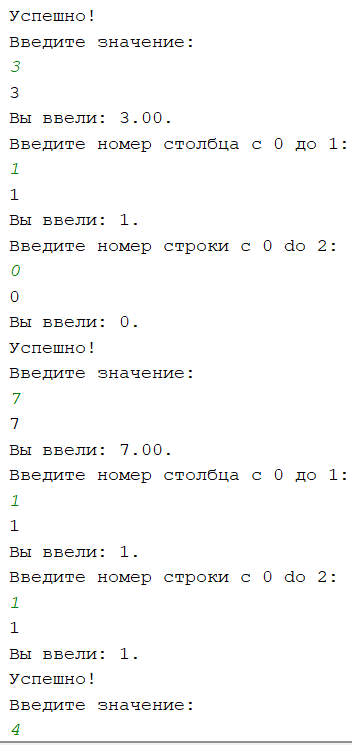
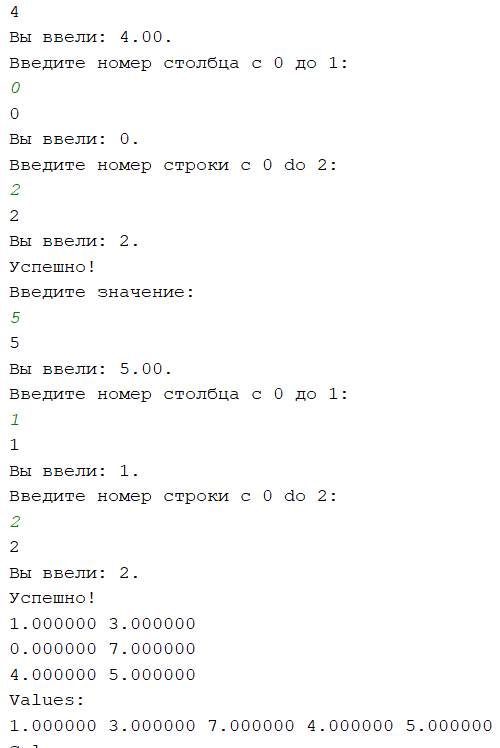
Так же программа предоставляет возможность ввода данных вручную даже при больших размерностях.

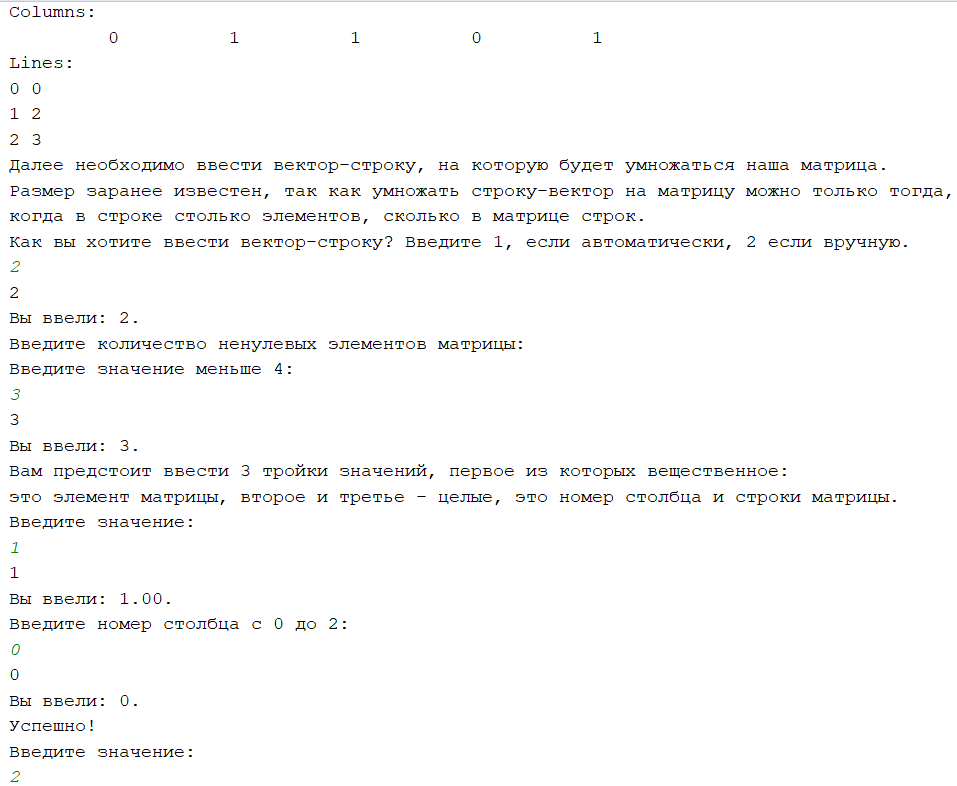
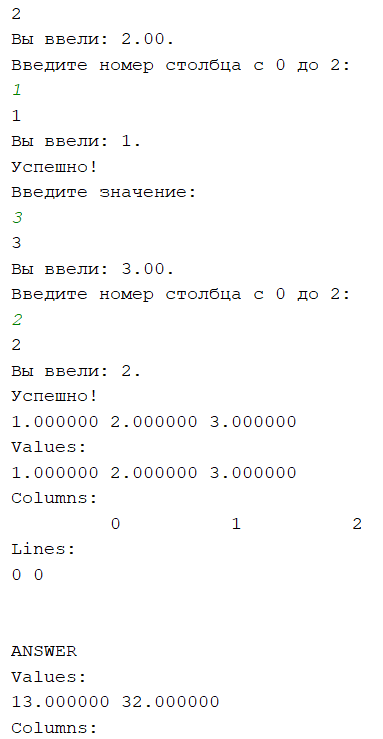
Пустой ввод в программе так же невозможен, как и нулевой ввод. При его попытке (нажатии Enter во время ввода) программа просто проигнорирует это и будет ждать пока вы введете нужное ей значение.

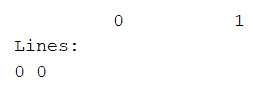


Что будет если ввести большее количество ненулевых элементов, чем может быть всего элементов в матрице: 

Тест на правильность умножения:



То же самое при обычном умножении:

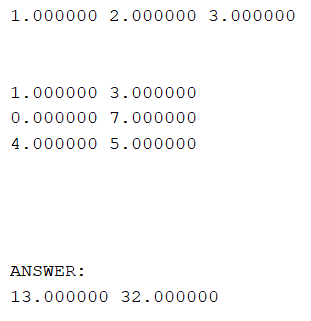


Таблица эффективности

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер матрицы | Процент заполненности | Время обычного умножения | Время умножения для разреженной матрицы | Память при обычном хранении | Память при хранении в структурах |
| 550000 | 100 | 0.004000 | 0.048000 | 4510424 | 2200000 |
| 50 | 0.006000 | 0.034000 | 2255424 |
| 25 | 0.007000 | 0.016000 | 1127924 |
| 5 | 0.004000 | 0.002000 | 225924 |
| 500000 | 100 | 0.006000 | 0.071000 | 4100424 | 2000000 |
| 50 | 0.004000 | 0.025000 | 2050424 |
| 25 | 0.004000 | 0.013000 | 1025424 |
| 5 | 0.008000 | 0.002000 | 205424 |
| 400000 | 100 | 0.004000 | 0.041000 | 3280424 | 1600000 |
| 50 | 0.003000 | 0.015000 | 1640424 |
| 25 | 0.003000 | 0.007000 | 820424 |
| 5 | 0.004000 | 0.002000 | 164424 |

Таким образом, мы видим, что использование векторов и разреженного списка становится выгодно по памяти только при высокой разреженности (меньше или равной 45 %). При этом, чем больше размерность матрицы, тем выгоднее использование трех структур при большом проценте разреженности.

Эффективным по времени способ хранения разреженных матриц становится только при совсем небольшом количестве элементов (примерно 5 %) от общего числа. Если процент заполненности меньше, то такое умножение делается намного медленнее, иногда даже в 2 раза.

Вывод

Можно сделать вывод, что структуры для хранения матриц можно применять только тогда, когда они сильно разрежены – тогда это дает значительный выигрыш и по времени, и по памяти. Иначе лучше применять обычный способ хранения.

Так, если мы имеем матрицу, 15% или менее элементов которой ненулевые, которая имеет большую размерность, то с ней будет целесообразно работать, используя первый алгоритм, представленный в данной лабораторной. Во всех остальных случаях к этому способу прибегать не стоит, так как программа может работать очень медленно из-за сложного обращения к элементам через три структуры, может не давать выигрыша по памяти на больших размерностях.

Вопросы

1. Что такое разреженная матрица, какие схемы хранения таких матриц Вы знаете?

Разреженная – матрица, содержащая достаточно большое количество элементов, из которых лишь малая часть является ненулевыми (n^(1+g) для матрицы размерности n, g<1).

Простейшая схема хранения разреженной матрицы: хранить массив ненулевых элементов (AN), и два массива их «координат» (I, J).

Кнут предложил хранить дополнительно массивы NR (содержит номер из AN следующего ненулевого j элемента, расположенного в матрице по строке) и NC (номера –‘’- по столбцу), а также массивы JR и JC (указатели для входа в строку и столбец). Чанг и Густавсон предложили схему разреженного строчного формата: значения ненулевых элементов хранятся в массиве AN, соответствующие им столбцовые индексы - в массиве JA. Кроме того, используется массив указателей, например IA, отмечающих позиции AN и JA, с которых начинаются описание очередной строки. Дополнительная компонента в IA содержит указатель первой свободной позиции в JA и AN.

2. Каким образом и сколько памяти выделяется под хранение разреженной и обычной матрицы?

Для хранение обычной матрицы: M \* N \* sizeof(list\_t). Память под разреженную матрицу выделяется в зависимости от схемы хранения. Память выделяется по мере наполнения ненулевыми элементами.

3. Каков принцип обработки разреженной матрицы?

Предполагает работу только с ненулевыми элементами.

4. В каком случае для матриц эффективнее применять стандартные алгоритмы обработки матриц? От чего это зависит?

Чем больше ненулевых элементов в матрице, тем менее эффективно использовать разряженные алгоритмы. При достижении определенного процента наполнения наблюдается значительная деградация разряженного алгоритма по времени работы.