

Рекурсия и двумерные массивы





Оглавление

[Приветствие](#)

[Двумерные массивы](#)

[Создание матрицы](#)

[Заполнение матрицы случайными числами](#)

[Закрашивание области](#)

[Рекурсия](#)

[Вычисление факториала](#)

[Вычисление чисел Фибоначчи](#)

[Примеры из жизни](#)

[Тетрис](#)

[Заключение](#)



[00:01:36]

Приветствие

Привет, друзья! Мы продолжаем изучать программирование. Сегодня поговорим о сложных, но интересных темах — двумерных массивах и рекурсии. Рассмотрим несколько алгоритмов, которые иногда просят на собеседованиях. Попишем код — всё, как обычно. Давайте приступать.

[00:02:11]

Двумерные массивы

Первое и главное — как задаются двумерные массивы. По аналогии с одномерными массивами мы указываем тип данных (например, **string**), в квадратных скобках ставим запятую как показатель того, что у нас будет две размерности: первая — строки, вторая — столбцы. Потом даём массиву название (например, **table**). После этого пишем **new string** и указываем, какое количество строчек и столбцов нам нужно.

```
string[, ] table = new string[2, 3]
```

В первом примере задан массив строк (можно сказать, что это таблица из строк). Но можно задать и прямоугольную таблицу чисел, которую в математике обычно называют матрицей. В квадратных скобках также первым числом мы обозначаем количество строк, которые будут в нашем двумерном массиве, а вторым числом указываем количество столбцов.

```
Int[, ] matrix = new int[5, 8]
```

Давайте немного попишем код и посмотрим, что у нас получится.

[00:03:03]

Создание матрицы

Начнём с первого примера — напишем таблицу строк. Тип данных **string** вы уже должны были запомнить. Дальше — **new string**. Указываем, что у нас будет 2 строки и 5 столбцов.

```
string[, ] table = new string[2, 5];
```

Чтобы обратиться к нужному элементу, указываем имя массива и в квадратных скобках пишем индекс строки (1) и индекс столбца (2). После этого можем работать как раньше, как будто это обычная переменная.

```
string[, ] table = new string[2, 5];  
table[1, 2] = "слово";
```

От себя отмечу, что индексы, которые вы будете использовать как для строк, так и для столбцов, меняются от нуля. То есть, если рассматривать весь наш массив, самый первый элемент будет **table[0,0]**. Учитывая, что у нас всего две строки, максимальный индекс строки будет **table[1,0]**.

Столбцов у нас пять, поэтому обращаться к ним мы будем так: **table[0,0]** — первый столбик, **table[0,1]** — второй, **table[0,2]** — третий, **table[0,3]** — четвёртый, **table[0,4]** — пятый. Напоминаю, что для строк инициализация происходит через константу `Empty`.

```
//
string[,] table = new string[2,5];
// String.Empty
// table [0,0] table [0,1] table [0,2] table [0,4]
// table [1,0] table [1,1] table [1,2] table [1,4]

table[1, 2] = "слово";
```

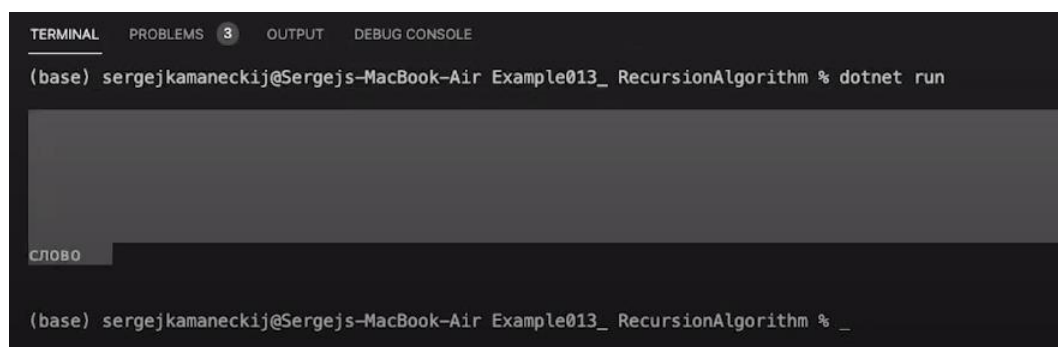
Давайте теперь, используя циклы (а здесь нам потребуется цикл в цикле), распечатаем данный массив. Укажем счётчик `rows`, количество строк не больше 2. Дальше будет внутренний (вложенный) цикл, который обозначает количество столбцов (у нас их 5). Здесь мы уже можем сделать распечатывание элементов массива. Давайте, как и ранее, воспользуемся интерполяцией. Обращаемся к элементам массива через имя массива. Дальше индекс строки и индекс столбца.

```
//
string[,] table = new string[2,5];
// String.Empty
// table [0,0] table [0,1] table [0,2] table [0,4]
// table [1,0] table [1,1] table [1,2] table [1,4]

table[1, 2] = "слово";

for (int rows = 0; rows < 2; rows++)
{
    for (int columns = 0; columns < 5; columns++)
    {
        Console.WriteLine($"{table[rows, columns]}");
    }
}
```

Попробуем запустить код в таком виде и посмотрим, что получится. После компиляции видим что-то непонятное:



```
TERMINAL  PROBLEMS 3  OUTPUT  DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run

слово

(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % _
```

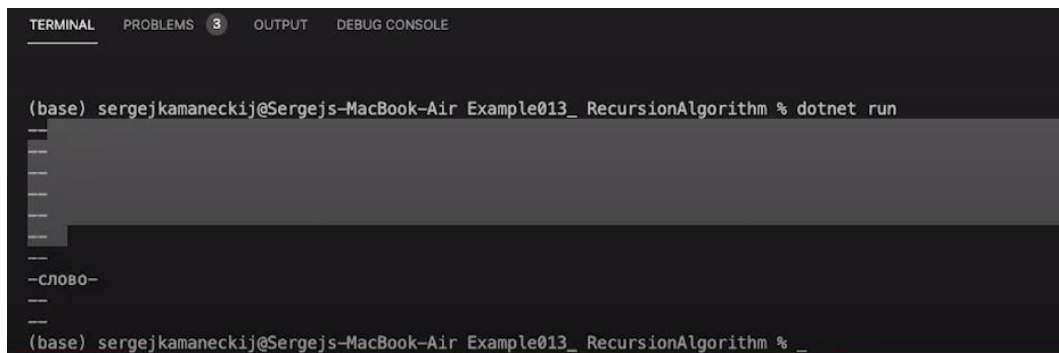
Чтобы убедиться, что у нас действительно вывелись 10 элементов, сделаем обрамление в виде минусов или дефисов (кому как больше нравится) и убедимся, что у нас действительно выводятся строки.

```
//
string[,] table = new string[2,5];
// String.Empty
// table [0,0] table [0,1] table [0,2] table [0,4]
// table [1,0] table [1,1] table [1,2] table [1,4]

table[1, 2] = "СЛОВО";

for (int rows = 0; rows < 2; rows++)
{
    for (int columns = 0; columns < 5; columns++)
    {
        Console.WriteLine($"-{table[rows, columns]}-");
    }
}
}
```

Как я и говорил, по умолчанию строки у нас иницируются пустой строкой (**String.Empty**).



```

TERMINAL  PROBLEMS  3  OUTPUT  DEBUG CONSOLE

(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run
--
--
--
--
--
--
--СЛОВО--
--
--
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % _

```

Теперь давайте попробуем сделать примерно то же с числами. Определим двумерный массив, назовём его **matrix** и укажем для него, например, 3 строки и 4 столбца. Затем поступим аналогично, используя циклы. Вместо **rows** можно написать **i** (внешний цикл, щёлкающий строки), вместо **columns** — **j** (внутренний цикл, щёлкающий столбцы). Выводим на экран. Там, где в прошлом примере был дефис (минус), поставим пробел.

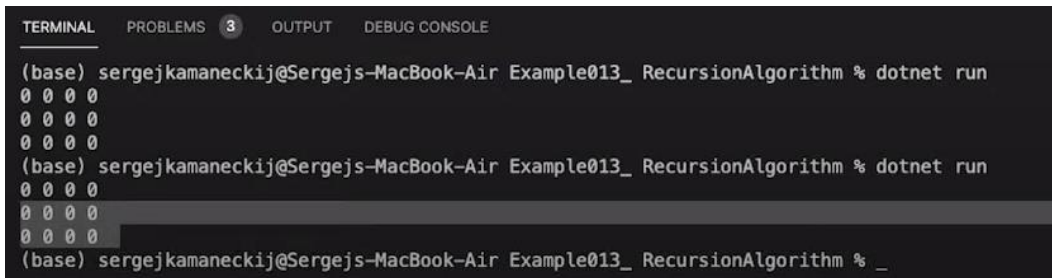
```
int[,] matrix = new int[3, 4];

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 4; j++)
    {
        Console.WriteLine($"{matrix[i, j]} ");
    }
}
}
```

Очистим терминал и посмотрим, что получится.


```
    }  
    Console.WriteLine();  
}
```

Убедимся, что получим тот же результат:



```
TERMINAL  PROBLEMS 3  OUTPUT  DEBUG CONSOLE  
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run  
0 0 0 0  
0 0 0 0  
0 0 0 0  
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run  
0 0 0 0  
0 0 0 0  
0 0 0 0  
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % _
```

Да, увидели всё то же самое.

[00:10:14]

Заполнение матрицы случайными числами

Теперь попробуем воспользоваться знаниями с предыдущих лекций и опишем метод, который будет отдельно печатать двумерную матрицу на экран и заполнять её числами. Итак, поехали.

Учитывая тот факт, что мы плюс-минус знаем, как это делается, можем немножко схитрить. Для текущего кода сделаем обрамление в виде метода. Делаем отступы, чтобы всё было красиво. В качестве аргумента передаём прямоугольную таблицу чисел. Вместо `matrix` будем передавать сокращённое название — `matr`. Метод `PrintArray` в качестве аргумента принимает двумерную таблицу чисел и будет печатать её на экран. В качестве аргумента передаём ту матрицу, которая была определена чуть раньше. Чтобы код был более скомпонованным, инициализацию массива перенесём поближе к вызову печати.

```
void PrintArray(int[,] matr)  
{  
    for (int i = 0; i < matr.GetLength(0); i++)  
    {  
        for (int j = 0; j < matr.GetLength(1); j++)  
        {  
            Console.Write($"{matr[i, j]} ");  
        }  
        Console.WriteLine();  
    }  
}  
  
int[,] matrix = new int[3, 4];  
  
PrintArray(matrix);
```

Всё хорошо:

```
TERMINAL  PROBLEMS  5  OUTPUT  DEBUG CONSOLE

(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % _
```

Теперь опишем дополнительный метод, который будет заполнять нашу матрицу случайными числами. Здесь всё почти так же, как с одномерными массивами. Для *i* указываем **matr.GetLength(0)**, для *j* — **matr.GetLength(1)**. Затем обращаемся к конкретному элементу на позиции «итый-житый» и пишем через использование генератора псевдослучайных чисел. Возьмём полуинтервал от 1 до 10. Напоминаю, из-за круглых скобок может показаться, что это интервал (как в математике), но у нас получается именно полуинтервал.

Проверим работоспособность нашего метода. Сначала инициализируемся, убедимся, что у нас нули. Затем сделаем **FillArray**, в качестве аргумента передадим наш массив и снова распечатаем. А чтобы отделить нули от чисел, перед финальной распечаткой добавим **Console.WriteLine()**.

```
void PrintArray(int[,] matr)
{
    for (int i = 0; i < matr.GetLength(0); i++)
    {
        for (int j = 0; j < matr.GetLength(1); j++)
        {
            Console.Write($"{matr[i, j]} ");
        }
        Console.WriteLine();
    }
}

void FillArray(int[,] matr)
{
    for (int i = 0; i < matr.GetLength(0); i++)
    {
        for (int j = 0; j < matr.GetLength(1); j++)
        {
            matr[i, j] = new Random().Next(1, 10); //[1; 10)
        }
    }
}

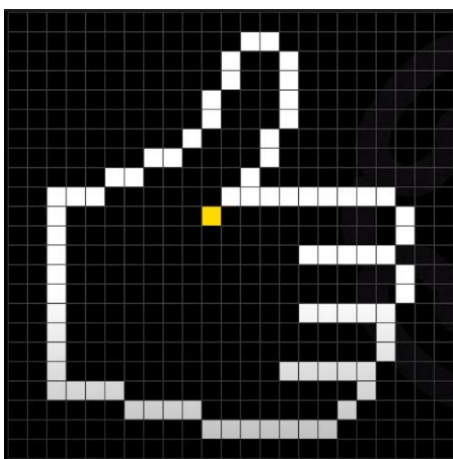
int[,] matrix = new int[3, 4];
PrintArray(matrix);
FillArray(matrix);
Console.WriteLine();
PrintArray(matrix);
```

Очистим консоль и запустим по новой:

закрашивать области, вы использовали данный алгоритм (не совсем его, но для объяснения я упрощу). Покажу работу на простом примере.

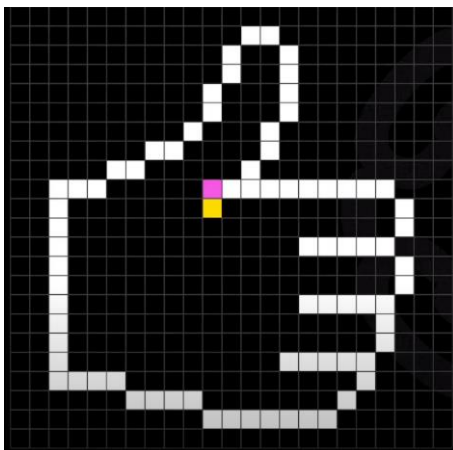
Итак, допустим, мы договорились, что у нас есть прямоугольная таблица чисел. В ней 23 строки и 25 столбцов. 0 — незакрашенный пиксель, 1 — закрашенный. Как же закрасить область?

Для этого определяем какую-то точку, которая находится внутри замкнутого контура.

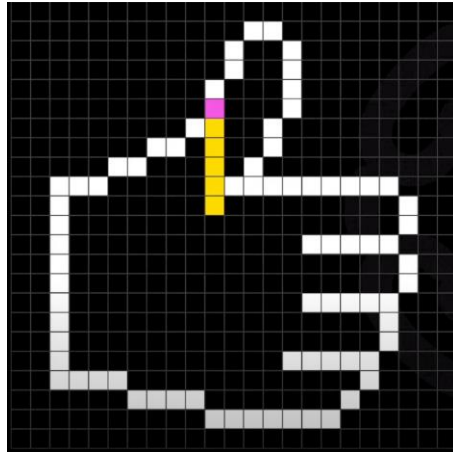


После этого нам надо определиться с тем, как мы будем делать обход внутренних точек. Если мы попали в точку, и эта точка не закрашена, мы её закрашиваем. Далее определяем правило обхода. В моём случае правило такое — сначала идём вверх, потом влево, потом вниз и вправо. Сейчас всё покажу детально.

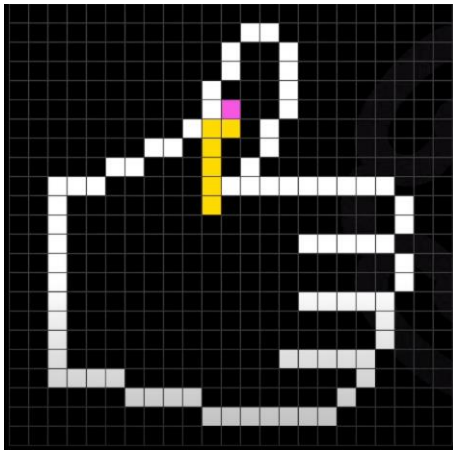
Итак, мы остановились на точке. Дальше смотрим на точку выше.



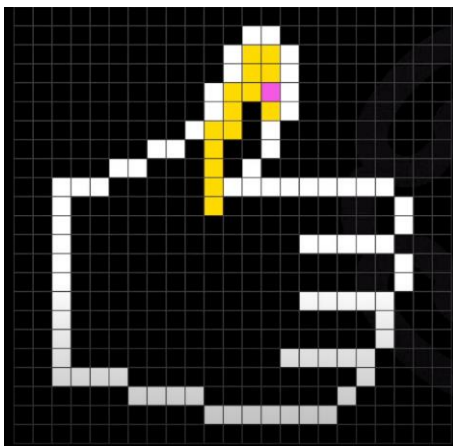
Если она не закрашена, мы её красим. От неё же смотрим на точку выше. Если она не закрашена, красим. И так пока не встретим закрашенную точку.



Дальше по правилу обхода смотрим на точку слева. Если бы её можно было закрасить, мы бы закрасили. Но в нашем случае снова попадаем на контур. Следующая по порядку точка ниже. Она тоже закрашена — не красим. Идём в последнее правое направление. Точка не закрашена — красим.



Затем для текущей точки повторяем все те же действия: идём вверх, если можно, красим, если нельзя — не красим. Двигаемся дальше по правилу.



И так далее. Вся идея закрашивания сводится к простым шагам.

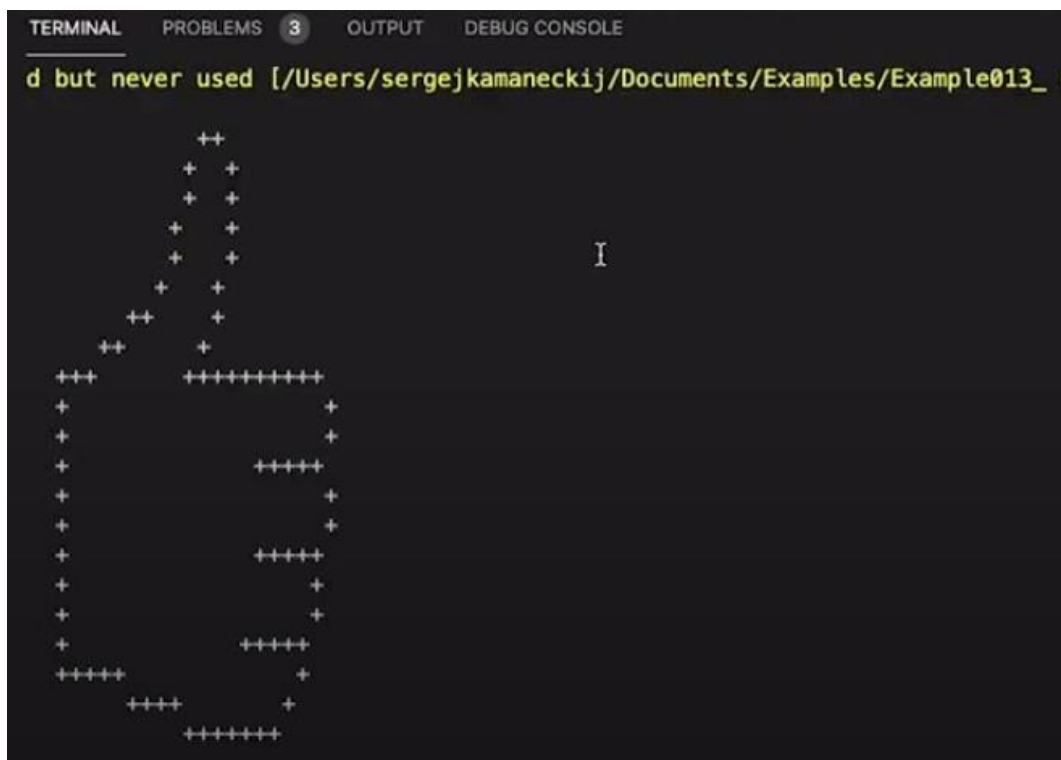
Давайте формально опишем шаги влево, вверх, вправо и вниз. Если мы находимся в текущей точке с координатами x, y (в данном случае x — позиция строки, а y — столбца), движение будет выглядеть таким образом:

будем искусственно печатать пробел. В противном случае мы можем распечатать плюс. **PrintImage** в качестве аргумента передаём массив, который содержит в себе картинку.

```
void PrintImage(int[,] image)
{
    for (int i = 0; i < image.GetLength(0); i++)
    {
        for (int j = 0; j < image.GetLength(1); j++)
        {
            if(image[i,j] == 0) Console.Write($" ");
            else Console.Write($"+");
        }
        Console.WriteLine();
    }
}

PrintImage(pic);
```

Запустим и посмотрим, что получится.



Видим палец вверх. Он не совсем красивый, но ничего страшного — это тонкости шрифтов. Технически в настройках можно указать шрифты с фиксированной шириной для символа. Но нас сейчас это не особо волнует.

Дальше предлагаю описать метод, который будет закрашивать картинку. По аналогии назовём его **FillImage**. А в качестве аргумента я укажу позицию строки и пикселя, с которого мы должны будем начать закрашку. Дальше я проверяю условие: если текущий пиксель (pic) с указанной позицией (row, col) равен нулю (то есть не закрашен), я буду его красить единичкой. А дальше вызову **FillImage**. И здесь мы определяем правило — что за чем идёт. Сначала поднимаемся на строчку выше (row-1, col), потом идём влево (row, col-1), потом вниз (row+1, col), потом вправо (row, col+1).

```

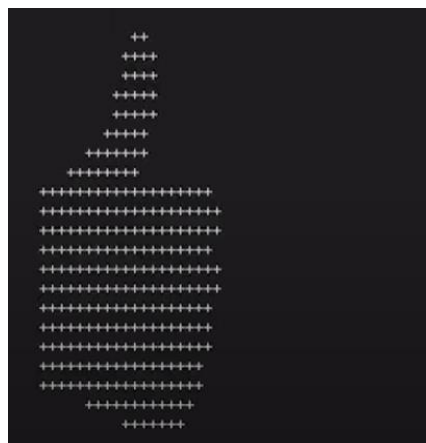
void PrintImage(int[,] image)
{
    for (int i = 0; i < image.GetLength(0); i++)
    {
        for (int j = 0; j < image.GetLength(1); j++)
        {
            if(image[i,j] == 0) Console.Write($" ");
            else Console.Write($"+");
        }
        Console.WriteLine();
    }
}

void FillImage(int row, int col)
{
    if (pic[row, col] == 0)
    {
        pic[row, col] = 1;
        FillImage(row - 1, col);
        FillImage(row, col - 1);
        FillImage(row + 1, col);
        FillImage(row, col + 1);
    }
}

PrintImage(pic);
FillImage(13, 13);
PrintImage(pic);

```

Давайте попробуем посмотреть, к чему нас это приведёт. В качестве случайной точки я указал (13, 13). Очищу терминал и запущу по новой.



Прошу любить и жаловать — один из алгоритмов, позволяющий закрашивать замкнутые области. У него есть тонкости: алгоритму нужно много ресурсов. Если вы захотите красить 5K картинку, вас ждут неприятности. Но на маленьких изображениях всё хорошо — нужно всего 7 строк кода.

[00:24:47]

Рекурсия

Здесь для вас открылась новая сущность — ситуация, при которой метод вызывает сам себя. В математике (программирование — это частный случай математики) есть целая область, которая занимается подобными случаями — рекуррентные соотношения. В программировании это просто называется рекурсией.

Что такое рекурсия? Это функция, которая вызывает сама себя. Есть шутка: чтобы понять рекурсию, нужно понять рекурсию. В ней подчёркивается суть явления. Пример с закраской картинки непростой, поэтому давайте рассмотрим более тривиальный. Начнём с классической задачи математики — вычисления факториала. В программировании решим её с помощью рекурсии.

[00:25:36]

Вычисление факториала

Что такое факториал? В математике под факториалом понимают произведения чисел от 1 до заданного и обозначают его восклицательным знаком — «!». Пример факториала: $5! = 5 * 4 * 3 * 2 * 1$. Если бы нам нужно было вычислить $120!$, мы бы считали произведение чисел от 1 до 120.

Программисты выяснили, что если мы считаем **5 * на произведение чисел от 1 до 4**, по определению факториала, это $4!$. Таким образом, $5!$ мы можем представить как **5 * $4!$** . $4!$ — это $4 * 3!$ и так далее. Таким образом, мы смогли задать функцию через саму себя. Попробуем написать это кодом.

Определим функцию или метод, который будет принимать число, факториал которого нужно вычислить. Как вы понимаете, это снова метод, который принимает какой-то аргумент (в частности, число) и возвращает факториал этого числа.

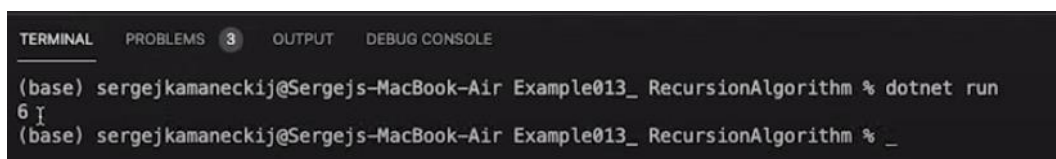
Определим метод как возвращающий `int`, и в качестве аргумента принимающий другое целое число. Далее по определению факториала мы явно укажем: «если мы дошли до единицы ($n = 1$), мы должны вернуть 1». Почему? Это определение факториала: $1! = 1$. Кстати, отметим, что $0!$ — это тоже 1.

Итак, если $n = 1$, возвращаем 1. В противном случае берём текущее значение и умножаем на факториал предыдущего числа ($n - 1$). Не забываем, что мы должны явно возвращать значение. То есть, если 1, возвращаем 1. Если не 1, то **$n * \text{Factorial}(n-1)$** (факториал предыдущего числа).

```
int Factorial(int n)
{
    // 1! = 1
    // 0! = 1
    if(n == 1) return 1;
    else return n * Factorial(n-1);
}

Console.WriteLine(Factorial(3)); // 1 * 2 * 3 = 6
```

Запустим и посмотрим, что получится.

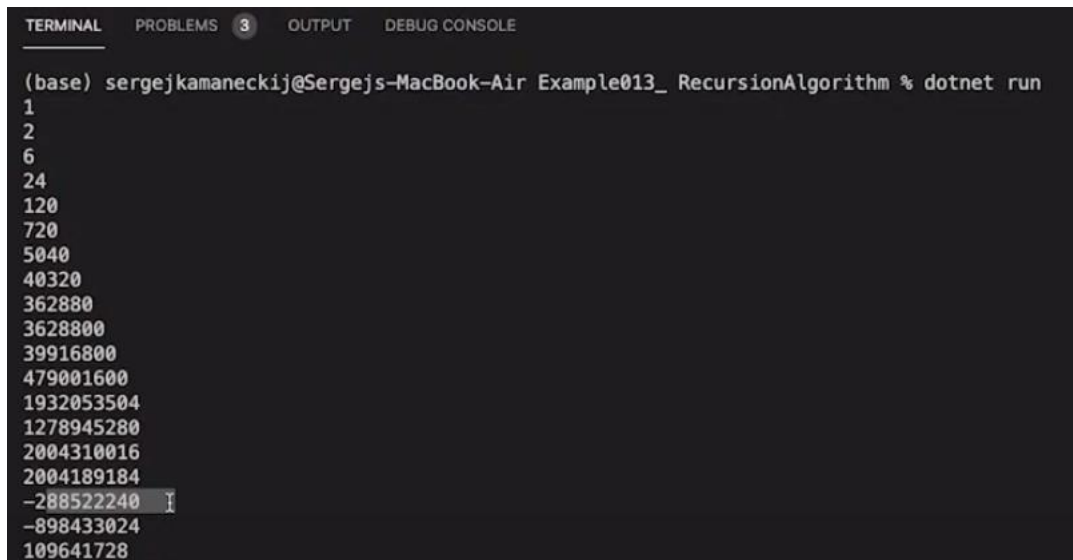


```
TERMINAL  PROBLEMS 3  OUTPUT  DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run
6
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % _
```

Получаем 6. Дальше, если попытаемся вычислить $5!$, получим 120 — всё правильно.

Казалось бы, задачу решили. Но есть проблема: когда мы будем вычислять большие числа (допустим, $40!$), в какой-то момент начнём получать отрицательные числа, чего быть не должно.

```
int Factorial(int n)
{
    // 1! = 1
    // 0! = 1
    if(n == 1) return 1;
    else return n * Factorial(n-1);
}
for (int i = 1; i < 40; i++)
{
    Console.WriteLine(Factorial(i));
}
```



```
TERMINAL  PROBLEMS  3  OUTPUT  DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
1932053504
1278945280
2004310016
2004189184
-288522240
-898433024
109641728
```

Это связано с переполнением типа. Давайте проверим, до какого значения можем посчитать факториал.

```
int Factorial(int n)
{
    // 1! = 1
    // 0! = 1
    if(n == 1) return 1;
    else return n * Factorial(n-1);
}
for (int i = 1; i < 40; i++)
{
    Console.WriteLine($"{i}! = {Factorial(i)}");
}
```

Очистим и запустим терминал. Всё хорошо до $17!$.


```
TERMINAL  PROBLEMS 3  OUTPUT  DEBUG CONSOLE

(base) sergejkamaneckij@Sergejs-MacBook-Air Example013_ RecursionAlgorithm % dotnet run
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
14! = 1278945280
15! = 2004310016
16! = 2004189184
17! = -288522240 I
18! = -898433024
19! = 109641728
20! = -2102132736
21! = -1195114496
22! = -522715136
23! = 862453760
24! = -775946240
25! = 2076180480
26! = -1853882368
27! = 1484783616
28! = -1375731712
29! = -1241513984
30! = 1409286144
31! = 738197504
32! = -2147483648
```

То есть число 17! попросту не вмещается в тип данных **integer**, поэтому появляется первая ваша задача, связанная с переполнением. Как её решать? Разными способами. Есть тип данных, который такие числа ещё способен переваривать, — **double**. Давайте **integer** заменим на **double** для возвращаемого результата. Потому что аргументом мы здесь передаём только число до 40.

```
double Factorial(int n)
{
    // 1! = 1
    // 0! = 1
    if(n == 1) return 1;
    else return n * Factorial(n-1);
}
for (int i = 1; i < 40; i++)
{
    Console.WriteLine($"{i}! = {Factorial(i)}");
}
```

```
TERMINAL  PROBLEMS  3  OUTPUT  DEBUG CONSOLE

10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 1.21645100408832E+17
20! = 2.43290200817664E+18
21! = 5.109094217170944E+19
22! = 1.124000727776077E+21
23! = 2.585201673888498E+22
24! = 6.204484017332394E+23
25! = 1.5511210043330986E+25
26! = 4.0329146112660565E+26
27! = 1.0888869450418352E+28
28! = 3.0488834461171384E+29
29! = 8.841761993739701E+30
30! = 2.6525285981219103E+32
31! = 8.222838654177922E+33
32! = 2.631308369336935E+35
33! = 8.683317618811886E+36
34! = 2.9523279903960412E+38
35! = 1.0333147966386144E+40
36! = 3.719933267899012E+41
37! = 1.3763753091226343E+43
38! = 5.23022617466601E+44
39! = 2.0397882081197442E+46
```

Видим нормальные значения. $E + 29$ означает, что получившееся число нужно умножить на 10^{29} . Это достаточно большие числа. Но тип `double` позволяет их хранить.

[00:31:05]

Вычисление чисел Фибоначчи

Мы посмотрели на простой пример использования рекурсии. Думаю, он понятнее, чем закрашка картинки. Есть ли ещё какие-то знакомые вам рекурсивные математические функции? Да, например, числа Фибоначчи, где каждое следующее число задаётся предыдущим. Давайте попробуем написать кодом вывод этих чисел.

Для начала укажем определение. Первое число можно указать как $f(1)=1$, дальше — $f(2)=1$. Для всех следующих — то есть, $f(n)$ — мы определяем числа Фибоначчи как $f(n-1) + f(n-2)$. Давайте напишем этот код.

У нас есть функция, которая будет что-то возвращать (начнём пока с целых чисел). Определяем её наименование — **Fibonacci**. В качестве аргумента указываем **n**. Если значение $n=1$ или $n=2$, возвращаем 1. Иначе мы хотим вернуть $Fibonacci(n-1) + Fibonacci(n-2)$.

```
// f(1) = 1
// f(2) = 1
// f(n) = f(n-1) + f(n-2)

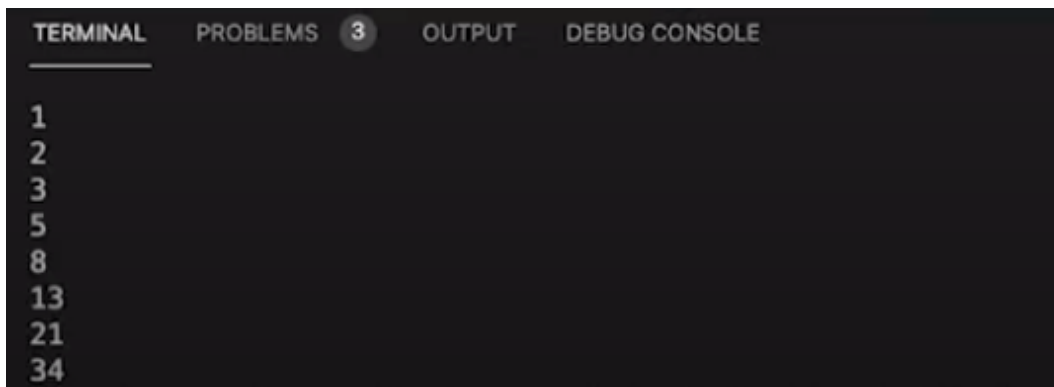
int Fibonacci(int n)
```

```
{  
    if(n == 1 || n == 2) return 1;  
    else Fibonacci(n-1) + Fibonacci(n-2);  
}
```

Технически всё. Если кого-то смущает наименование, вы знаете, что делать — rename. Можно указать, например, f. Так функция будет выглядеть компактнее. Но я оставлю полное наименование.

Дальше посмотрим, как будут считаться некоторые числа Фибоначчи. Покажем первые 10.

```
int Fibonacci(int n)  
{  
    if(n == 1 || n == 2) return 1;  
    else return Fibonacci(n-1) + Fibonacci(n-2);  
}  
  
for (int i = 1; i < 10; i++)  
{  
    Console.WriteLine(Fibonacci(i));  
}
```



TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE

```
1  
2  
3  
5  
8  
13  
21  
34
```

Получаются те самые числа Фибоначчи — каждое равно сумме двух предыдущих.

Есть тонкий момент. По аналогии с факториалом попробуем посчитать первые 50 чисел Фибоначчи так, чтобы возвращалось double-значение.

```
double Fibonacci(int n)  
{  
    if(n == 1 || n == 2) return 1;  
    else return Fibonacci(n-1) + Fibonacci(n-2);  
}  
  
for (int i = 1; i < 50; i++)  
{  
    Console.WriteLine(Fibonacci(i));  
}
```

Первые числа вылетят в консоль достаточно быстро, а дальше она начнёт работать медленнее. Попробуем понять, с какого числа начинается проблема.

```
double Fibonacci(int n)
{
    if(n == 1 || n == 2) return 1;
    else return Fibonacci(n-1) + Fibonacci(n-2);
}

for (int i = 1; i < 50; i++)
{
    Console.WriteLine($"f({i}) = {Fibonacci(i)}");
}
```

Начиная с сорокового числа значения выводятся очень медленно. С чем это связано, обязательно обсудим на семинаре. Но сразу отмечу, что рекурсию можно сделать более шустрой. Можно ли заменить рекурсию другими конструкциями (в частности, циклом) — да, можно. Именно этим мы займёмся на семинаре.

[00:35:04]

Примеры из жизни

Вы посмотрели математические примеры использования рекурсии, но есть и житейские. Например, если нужно сделать обход какой-то папки, обычной директории на вашем компьютере. В этом случае у нас опять всё сводится к рекурсии: чтобы обойти директорию (показать всё, что в этой директории есть), нам последовательно нужно пройти все папки, которые уже есть в этой папке и показать их содержимое. То есть, чтобы показать содержимое папок, нужно показать содержимое папок.

Чтобы сделать это на языке C#, нам придется изучить API для работы с файловой системой. Если будет интересно, это можно сделать на семинарском занятии. Пока помним о том, что рекурсия не только где-то в математике используется, но и в жизни.

Если я ещё не убедил вас в том, что нам нужны двумерные массивы и рекурсивные методы, предлагаю рассмотреть идею использования двумерных массивов и методов на примере игры в тетрис. Мы обязательно детально это разберём на семинарах. Пока что сама идея. Может быть, кто-то из вас захочет попробовать самостоятельно.

[00:36:21]

Тетрис

От себя отмечу, что весь функционал и весь инструментарий, чтобы реализовать эту игру, у вас есть. В чём основная затея? Обычное поле тетриса — это двумерный массив. В нём есть 0 (свободные клетки) и 1 (клетки, на данном этапе занятые какой-то фигурой). Движение фигуры — смещение 0 и 1 относительно друг друга.

Прямоугольная таблица чисел — это матрица. А матрицы мы можем поворачивать. Соответственно, поворот фигуры — это поворот матрицы. Обязательно дождемся семинаров и разбираем это более детально.

[00:37:01]

Заключение

Мы узнали, как задаются двумерные массивы в языке C#. Обязательно запомните запятую, которую нужно не забывать ставить и в дальнейшем указать количество строк и столбцов. Помним о функционале `Get.Length`, который позволяет получить то самое количество строк и столбцов.

Также теперь мы знаем, что массивы на самом деле сплошь и рядом. Из предыдущей лекции мы сами узнали, что есть различные виды методов. Здесь мы ещё дополнили знания тем, что есть методы, которые вызывают сами себя, и они называются рекурсивными. На практике они используются достаточно часто. На старте бывают нужны не часто, но знать о них и немного попрактиковаться в написании я настоятельно рекомендую.

На этом я с вами прощаюсь. Всем счастливо, пока-пока.