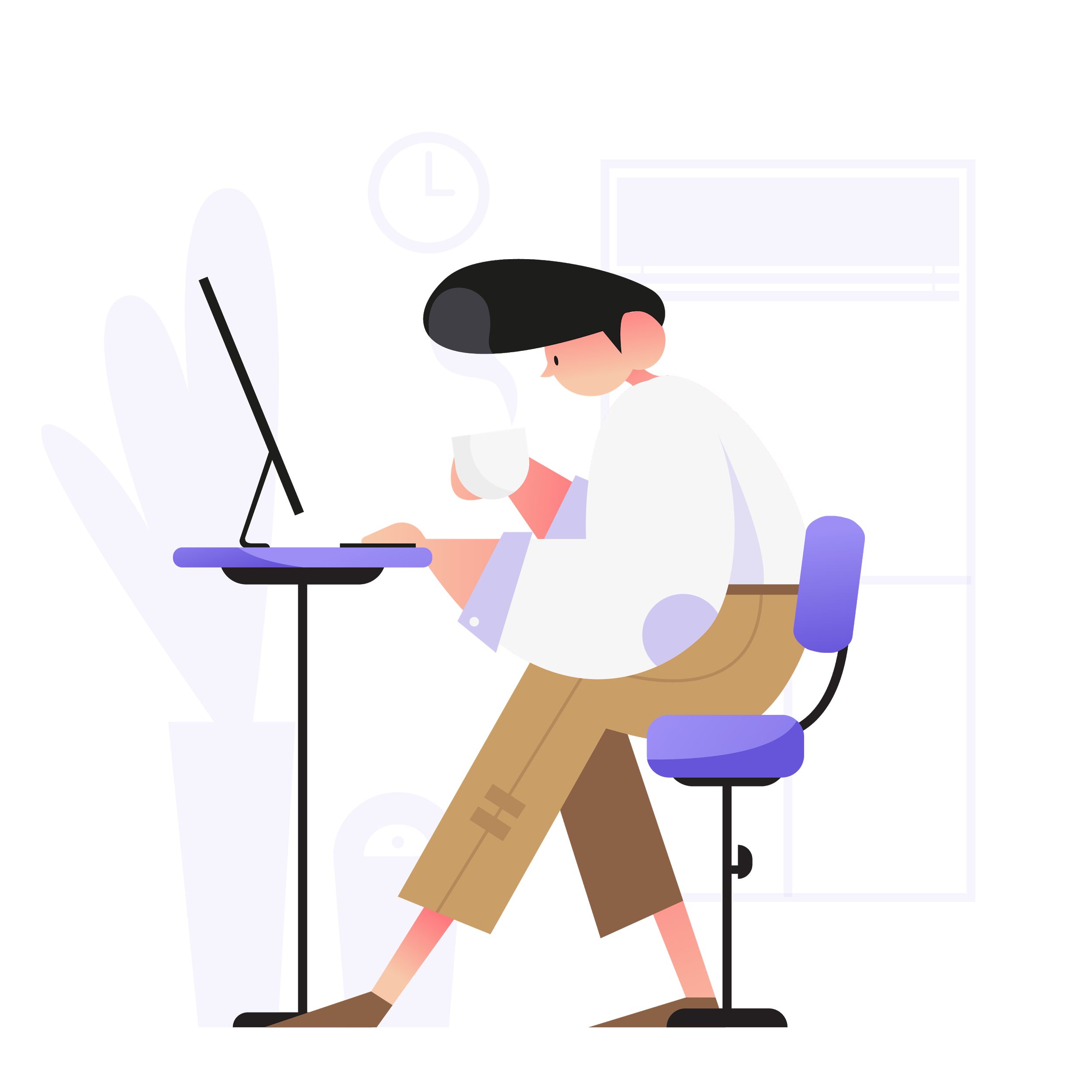
Алгоритмы и структуры данных на С#

Динамическое программирование

Netcore 3.1



# На этом уроке

1. Динамическое программирование.
2. Количество маршрутов с помощью динамического программирования.
3. Количество маршрутов с препятствиями с помощью динамического программирования.
4. Наибольшая общая подпоследовательность с помощью динамического программирования.
5. Поиск с возвратом с помощью динамического программирования.
6. Использование метода поиска с возвратом в задаче о восьми ферзях.

Оглавление

[На этом уроке](#_2et92p0)

[Оглавление](#_3dy6vkm)

[Введение](#_j9i020c67qkk)

[Динамическое программирование](#_zhjfg242q1s9)

[Пример. Задача на количество вариантов](#_3znysh7)

[План решения задач](#_2uafesx1o2vp)

[Количество маршрутов](#_x58w5ldaeomy)

[Количество маршрутов с препятствиями](#_tyjcwt)

[Наибольшая общая подпоследовательность](#_rt3dz7jr4yx3)

[Решение](#_1t3h5sf)

[Поиск с возвратом](#_4d34og8)

[Описание метода](#_2s8eyo1)

[Использование метода](#_17dp8vu)

[Задача о восьми ферзях](#_3rdcrjn)

[Заключение](#_9iuz4swaeaq5)

[Практическое задание](#_vs1tr29tezr1)

[1. Задача](#_c9dtpfeb0upc)

[Дополнительные материалы](#_2jxsxqh)

[Используемая литература](#_7c9jrnh1eqkx)

# 

# 

# Введение

Динамическое программирование — один из путей решения задач в программировании, которое следует знать уважающему себя программисту, так как принципы, которое оно использует, применимы во множестве случаев: они помогают разбивать одну большую задачу на несколько малых.

# Динамическое программирование

**Динамическое программирование** в теориях управления и вычислительных систем — способ решения сложных задач путём разбиения их на более простые подзадачи.

Ключевая идея в динамическом программировании достаточно проста: как правило, чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее. Часто многие из этих подзадач одинаковы. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений. Это особенно полезно в случаях, когда число повторяющихся подзадач крайне велико.

Метод динамического программирования сверху — это простое запоминание результатов решения тех подзадач, которые могут повторно встретиться в дальнейшем. Динамическое программирование снизу включает в себя переформулирование сложной задачи в виде рекурсивной последовательности более простых подзадач.

## Пример. Задача на количество вариантов

У исполнителя «Калькулятор» две команды, которым присвоены номера:

1. Прибавь 1.
2. Умножь на 2.

Сколько есть программ, которые число 1 преобразуют в число 16?

Эта задача решается довольно легко, если составить рекуррентное соотношение:

P(1) = 1;  
P(N) = P(N – 1) + P(N / 2), при N > 1, если N кратно 2;  
P(N) = P(N – 1), при N > 1, если N не кратно 2.

Попробуйте запрограммировать его сами. Здесь можно использовать как рекурсивный, так и нерекурсивный способ.

Ответ для проверки: 36.

### План решения задач

1. Найти решение задачи для маленьких ограничений.
2. Ввести обозначение, что такое Pi — сформулировать, какую величину мы считаем.
3. Получить рекуррентное соотношение.
4. Определить, при каких ограничениях работает эта формула, и выписать начальные значения.
5. Определить порядок вычислений.
6. Определить, где лежит вычисленное значение.

## Количество маршрутов

Пусть за один ход королю разрешается передвинуться на одну клетку вниз или вправо. Необходимо определить, сколько существует различных маршрутов, ведущих из левого верхнего угла в правый нижний. Будем считать, что положение короля задаётся парой чисел (a, b), где a задаёт номер строки, а b — номер столбца. Строки нумеруются сверху вниз от 0 до n − 1, а столбцы — слева направо от 0 до m − 1. Таким образом, первоначальное положение короля — клетка (0, 0), а конечное — клетка (n − 1, m − 1). Пусть W(a, b) — количество маршрутов, ведущих в клетку (a, b) из начальной клетки. Запишем рекуррентное соотношение. В клетку (a, b) можно прийти двумя способами: из клетки (a, b − 1), расположенной слева, и из клетки (a − 1, b), расположенной сверху от данной. Поэтому количество маршрутов, ведущих в клетку (a, b), равно сумме количеств маршрутов, ведущих в клетку слева и сверху от неё. Получили рекуррентное соотношение:

|  |
| --- |
| W(a, b) = W(a, b − 1) + W(a − 1, b) |

Это соотношение верно при a > 0 и b > 0. Зададим начальные значения: если a = 0, то клетка расположена на верхнем краю доски, и прийти в неё можно единственным способом — двигаясь только влево, поэтому W(0, b) = 1 для всех b. Аналогично W(a, 0) = 1 для всех a. Создадим массив W для хранения значений функции, заполним первую строку и первый столбец единицами, а затем заполним все остальные элементы массива. Поскольку каждый элемент равен сумме значений, стоящих слева и сверху, заполнять массив W будем по строкам сверху вниз, а каждую строку — слева направо.

|  |
| --- |
| using System;  namespace Geekbrains {  class Program  {  const int N = 3;  const int M = 3;   static void Print2(int n, int m, int[,] a)  {  int i, j;  for (i = 0; i < n; i++)  {  for (j = 0; j < m; j++)  Console.Write(a[i, j]);  Console.Write("\r\n");  }  }   static void Main(string[] args)  {  int[,] A = new int[N, M];  int i, j;  for (j = 0; j < M; j++)  A[0, j] = 1; *// Первая строка заполнена единицами*  for (i = 1; i < N; i++)  {  A[i, 0] = 1;  for (j = 1; j < M; j++)  A[i, j] = A[i, j - 1] + A[i - 1, j];  }   Print2(N, M, A);  }  } } |

## Количество маршрутов с препятствиями

Пусть некоторые клетки на доске будут «запрещёнными»: король не может ходить на них. Карта запрещённых клеток задана при помощи массива Map[n][m]: нулевое значение элемента массива означает, что эта клетка запрещена, единичное значение означает, что на клетку можно ходить. Массив Map считывается программой после задания значений n и m. Король может ходить только вниз или вправо. Для решения этой задачи придётся изменить рекуррентное соотношение с учётом наличия запрещённых клеток. Для запрещённой клетки количество ведущих в неё маршрутов будем считать равным 0. Получим:

**W(a,b) = W(a – 1, b) + W(a, b – 1), если Map[a][b] = 1,  
W(a,b) = 0, если Map[a][b] = 0.**  
Также надо учесть, что для клеток верхней строки и левого столбца эта формула некорректна, поскольку для них не существует соседней сверху или слева клетки.

Попробуйте решить задачу самостоятельно.

# Наибольшая общая подпоследовательность

Задача нахождения наибольшей общей подпоследовательности (англ. longest common subsequence, LCS) — задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух). Часто задача определяется как поиск всех наибольших подпоследовательностей. Это классическая задача информатики, которая имеет приложения, в частности, в задаче сравнения текстовых файлов (утилита diff), а также в биоинформатике.

Подпоследовательность можно получить из некоторой конечной последовательности, если удалить из последней некоторое множество её элементов (возможно, пустое). Например, BCDB — подпоследовательность последовательности ABCDBAB. Последовательность Z — общая подпоследовательность последовательностей X и Y, если Z будет подпоследовательностью как X, так и Y. Требуется для двух последовательностей X и Y найти общую подпоследовательность наибольшей длины. Заметим, что НОП может быть несколько.

Обратите внимание: подпоследовательность отличается от подстроки. Например, если есть исходная последовательность ABCDEF, то ACE будет подпоследовательностью, но не подстрокой, а ABC будет как подпоследовательностью, так и подстрокой.

## Решение

Вначале найдём длину наибольшей подпоследовательности. Допустим, мы ищем решение для случая (n1, n2), где n1, n2 — длины первой и второй строк. Пусть уже существуют решения для всех подзадач (m1, m2), меньших заданной. Тогда задача (n1, n2) сводится к меньшим подзадачам следующим образом.

Рекурсивное решение:

|  |
| --- |
| static int lcsLength(string a, string b) {  if (a.Length == 0 || b.Length == 0)  return 0;  else if (a[0] == b[0])  return 1 + lcsLength(a.Substring(1), b.Substring(1));  else  return Math.Max(lcsLength(a.Substring(1), b), lcsLength(a, b.Substring(1))); } |

Представленный алгоритм позволяет найти длину максимальной последовательности.

Решение этой задачи очень похоже на решение задачи о поиске длины пути. Также её можно свести к решению с помощью матрицы.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | G | E | E | K | B | R | A | I | N | S |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | **1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| E | 0 | 1 | **2** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| E | 0 | 1 | 2 | **3** | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| K | 0 | 1 | 2 | 3 | **4** | 4 | 4 | 4 | 4 | 4 | 4 |
| M | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| I | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | **5** | 5 | 5 |
| N | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | **6** | 6 |
| D | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 6 | 6 |
| S | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 6 | **7** |

# Поиск с возвратом

Поиск с возвратом, бэктрекинг (англ. backtracking) — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве М. Как правило, он позволяет решать задачи, в которых ставятся вопросы типа: «Перечислите все возможные варианты…», «Сколько существует способов…», «Есть ли способ…», «Существует ли объект…» и т. п.

## Описание метода

Рекурсия используется также для поиска лучшего решения сложных задач. В алгоритмах с возвратом создаются тестовые варианты, которые просчитывают возможность справиться с поставленной задачей. Если программа делает вывод, что предлагаемый способ не приведёт к нужному результату, тестовый вариант отбрасывается и вверху по стеку вызовов ищется другой.

Такой метод очень удобен: он позволяет строить частичное решение и определять, приведёт ли оно к полному. Благодаря такому подходу вы можете прекратить дальнейшее усовершенствование пробного варианта и вернуться на несколько шагов назад, чтобы продолжить работу оттуда.

|  |
| --- |
| *// Исследуем тестовое решение* *// Возвращаем false, если его нельзя развить до полного решения* *// Возвращаем true, если рекурсивный вызов SearchSolution приводит к полному // решению* Boolean: SearchSolution(Solution: test\_solution) *// Если можно сделать вывод, что частичное решение*  *// не приводит к полному, возвращаем false*  If <test\_solution не может решить проблему> Then Return false *// Если это полное решение, возвращаем true*  If <test\_solution - полное решение> Then Return true *// Расширяем частичное решение*  Loop <Проходим по всем возможным расширениям в test\_solution.>  <Расширяем test\_solution.>  *// Рекурсивно проверяем, ведёт ли это к решению*  If (SearchSolution(test\_solution)) Then Return true *// Это расширение не ведёт к решению. Отменяем расширение*  <Отменяем расширение.>  End Loop  *// Если мы дошли до этой строки, частичное решение*  *// не приводит к полному* Return false End SearchSolution |

Алгоритм SearchSolution берёт в качестве параметра любые необходимые данные, чтобы проработать частичное решение, и возвращает true, если оно приводит к полному.

Сначала алгоритм проверяет, допустимо ли частичное решение. Если оно не приводит к полному решению, алгоритм возвращает false и вызывает метод LeadsToSolution, который завершает текущий тест и приступает к новому. Если предлагаемое решение допустимо, алгоритм циклически проходит по всем возможным его расширениям до получения конечного результата. Для каждого расширения алгоритм рекурсивно ссылается на самого себя, чтобы определить, будет ли оно работать. Если рекурсивный вызов возвращает false, расширение не годится: оно отменяется и начинается следующая попытка с новым расширением. Если алгоритм перепробовал все возможные расширения и не нашёл среди них целесообразного, он возвращает false, чтобы вызов SearchSolution закрыл тестовое решение.

## Использование метода

Классический пример использования алгоритма поиска с возвратом — задача о **восьми ферзях**. Её формулировка такова: «Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Сперва на доску ставят одного ферзя, а потом пытаются поставить каждого следующего ферзя так, чтобы его не били уже установленные ферзи. Если на очередном шаге такую установку сделать нельзя, возвращаются на шаг назад и пытаются поставить ранее установленного ферзя на другое место.

## Задача о восьми ферзях

В решении мы используем рекурсивный подход.

Запускаем функцию поиска при N = 1.

Начало функции SearchSolution(1).

1. Если расстановка (CheckBoard) не подходит, не ставим ферзя.
2. При N = 9 ферзи расставлены (т. е. девятого ферзя ставить никуда не нужно). Решение найдено. Заканчиваем поиск. Идем к шагу 7.
3. Ищем решение. Запускаем цикл перебора клеток доски (row и col).
4. Ставим ферзя на доску в позицию [row][col], сообщаем об этом вызовом SearchSolution(N + 1). Идём к шагу 1.
5. Если ферзь не подошёл (SearchSolution вернул 0), убираем его (board[row][col] = 0).
6. Проверяем следующую позицию (идём к шагу 3).
7. Выходим из функции.

Здесь для проверки полезно написать функцию CheсkBoard, которая проверяет расположение ферзей. Самих ферзей можно расставлять в двухмерном массиве 8 × 8, который первоначально нужно заполнить нулями.

|  |
| --- |
| using System;  namespace Geekbrains {  class Program  {  const int N = 8;  const int M = 8;  *// Доска для ферзей* *// 0 - пустая клетка* *// Число - номер ферзя*  static int[,] board = new int[N, M];   static void Main(string[] args)  {  Zero(N, M, board);  SearchSolution(1);  Console.WriteLine(" ");  Print(N, M, board);  }   static bool SearchSolution(int n)  { *// Если проверка доски возвращает 0, то эта расстановка не подходит*  if (CheckBoard() == 0) return false; *// 9 ферзя не ставим. Решение найдено*  if (n == 9) return true;  int row;  int col;  for (row = 0; row < N; row++)  for (col = 0; col < M; col++)  {  if (board[row, col] == 0)  { *// Расширяем test\_solution*  board[row, col] = n; *// Рекурсивно проверяем, ведёт ли это к решению.*  if (SearchSolution(n + 1)) return true; *// Если мы дошли до этой строки, данное частичное решение* *// не приводит к полному*  board[row, col] = 0;  }  }   return false;  }  *// Проверка всей доски*  static int CheckBoard()  {  int i, j;  for (i = 0; i < N; i++)  for (j = 0; j < M; j++)  if (board[i, j] != 0)  if (CheckQueen(i, j) == 0)  return 0;  return 1;  }  *// Проверка определённого ферзя*  static int CheckQueen(int x, int y)  {  for (int i = 0; i < N; i++)  for (int j = 0; j < M; j++) *// Если нашли фигуру*  if (board[i, j] != 0)  if (!(i == x && j == y)) *// Если это не наша фигура*  { *// Лежат на одной вертикали или горизонтали*  if ((i - x) == 0 || (j - y) == 0)  return 0; *// Лежат на одной диагонали*  if (Math.Abs(i - x) == Math.Abs(j - y))  return 0;  }  *// Если мы дошли до этого места, то всё в порядке*  return 1;  }  *// Выводим доску на экран*  static void Print(int n, int m, int[,] a)  {  int i, j;  for (i = 0; i < n; i++)  {  for (j = 0; j < m; j++)  Console.Write(a[i, j]);  Console.Write("\n");  }  }  *// Очищаем доску*  static void Zero(int n, int m, int[,] a)  {  int i, j;  for (i = 0; i < n; i++)  for (j = 0; j < m; j++)  a[i, j] = 0;  }  } } |

# Заключение

Рассмотрев, что такое динамическое программирование, на примере различных задач, можно выделить что основное в динамическом программировании — разделение задачи на более маленькие, запоминание результатов выполнения подзадач, чтобы уменьшить количество вычислений при повторах.

# Практическое задание

## 1. Задача

Для прямоугольного поля размера M на N клеток, подсчитать количество путей из верхней левой клетки в правую нижнюю. Известно что ходить можно только на одну клетку вправо или вниз.

# Дополнительные материалы

1. [Задача о восьми ферзях на Википедии.](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D0%B2%D0%BE%D1%81%D1%8C%D0%BC%D0%B8_%D1%84%D0%B5%D1%80%D0%B7%D1%8F%D1%85)
2. [Д. П. Кириенко. Динамическое программирование](http://ejudge.btty.su/bmstu/2007-2008/docs/dp1.pdf).
3. [Сайт К. Ю. Полякова. Задачи ЕГЭ (рекурсивные алгоритмы)](http://kpolyakov.spb.ru/school/ege.htm).

# Используемая литература

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.