Динамическое программирование (DP) - это метод решения задач, основанный на разбиении большой задачи на более мелкие подзадачи и сохранении результатов вычислений этих подзадач для последующего использования.

Динамическое программирование широко используется для оптимизации производительности алгоритмов и ускорения вычислений во многих областях, включая компьютерную графику, искусственный интеллект, физику, экономику и другие.

Одной из основных причин использования динамического программирования является то, что оно позволяет избежать повторного вычисления одних и тех же результатов в рамках одной задачи. Это значительно ускоряет процесс вычислений и снижает нагрузку на вычислительные ресурсы.

Кроме того, динамическое программирование позволяет эффективно решать задачи с оптимальной подструктурой, то есть задачи, которые можно решить, разбив их на более мелкие подзадачи. Данный подход особенно полезен в задачах оптимизации и нахождения наилучшего решения.

В целом, динамическое программирование является очень мощным инструментом, который позволяет эффективно решать широкий спектр задач в различных областях.

Динамическое программирование можно использовать в задачах, удовлетворяющих двум основным критериям:

1. Наличие оптимальной подструктуры. Это означает, что решение задачи может быть представлено как комбинация решений более мелких подзадач. То есть, если мы можем решить каждую подзадачу и объединить их решения, то мы можем получить оптимальное решение для исходной задачи.
2. Перекрытие подзадач. Это означает, что в задаче могут возникать повторяющиеся подзадачи. Если мы можем сохранить результаты вычислений этих подзадач и использовать их для решения других подзадач, то мы можем существенно ускорить процесс вычислений.

В целом, динамическое программирование может быть использовано в широком спектре задач, таких как нахождение наибольшей общей подпоследовательности, определение оптимального пути в графе, решение задач на расстановку скобок, определение наибольшей возрастающей подпоследовательности и многих других.

Однако, стоит отметить, что динамическое программирование может быть неэффективно, если задача не удовлетворяет критериям оптимальной подструктуры и перекрытия подзадач, или если эти критерии не могут быть эффективно использованы.

Один из примеров задач, которая может быть решена с помощью динамического программирования, - это задача нахождения наибольшей общей подпоследовательности (Longest Common Subsequence, LCS) двух строк.

Например, у нас есть две строки: "ABCDGH" и "AEDFHR". Необходимо найти наибольшую общую подпоследовательность этих двух строк.

Для решения этой задачи мы можем использовать следующий алгоритм на основе динамического программирования:

1. Создаем матрицу размером (m+1) x (n+1), где m и n - длины двух строк.
2. Заполняем первую строку и первый столбец матрицы нулями.
3. Для каждой ячейки матрицы (i,j) рассчитываем ее значение следующим образом:

* Если символы s1[i] и s2[j] равны, то значение ячейки (i,j) равно значению ячейки (i-1,j-1) + 1.
* Иначе значение ячейки (i,j) равно максимуму из значений ячеек (i-1,j) и (i,j-1).

1. Значение в правом нижнем углу матрицы будет являться длиной наибольшей общей подпоследовательности.

Вот как выглядит код на Python, который реализует этот алгоритм:

def lcs(s1, s2):

# Определяем длины строк

m, n = len(s1), len(s2)

# Инициализируем двумерный массив размером (m+1) x (n+1) нулями

dp = [[0] \* (n+1) for \_ in range(m+1)]

# Заполняем матрицу построчно, начиная со второй строки и второго столбца

for i in range(1, m+1):

for j in range(1, n+1):

# Если символы s1[i-1] и s2[j-1] равны, то значение ячейки (i,j) равно значению ячейки (i-1,j-1) + 1.

if s1[i-1] == s2[j-1]:

dp[i][j] = dp[i-1][j-1] + 1

# Иначе значение ячейки (i,j) равно максимуму из значений ячеек (i-1,j) и (i,j-1).

else:

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

# Возвращаем значение в правом нижнем углу матрицы, которое является длиной наибольшей общей подпоследовательности.

return dp[m][n]

Алгоритм решения моей задачи:

1. Считываем данные из входного файла.
2. Создаем двумерный массив boolean dp размером (M+1)x(N+1), где dp[i][j] будет равно true, если игрок, начинающий игру, может выиграть, начиная с клетки (i,j), и false в противном случае.
3. Заполняем dp[M][N] значением false, так как правая нижняя клетка отравлена.
4. Проходим по всем клеткам пирога в порядке слева-направо, снизу-вверх. Для каждой клетки (i,j) рассматриваем все возможные ходы: выбираем клетку (k,l), где k<i и l<j, и проверяем, может ли игрок выиграть, начиная с этой клетки. Если хотя бы один из таких ходов позволяет игроку выиграть, то dp[i][j] равно true, иначе - false.
5. Завершаем алгоритм. Все клетки, для которых dp[i][j] равно true, являются выигрышными ходами.

using System;

using System.IO;

using System.Collections.Generic;

namespace PoisonedCakeGame

{

class Program

{

static void Main(string[] args)

{

int m, n; // размеры поля

List<int> X = new List<int>(); // лист с входными данными

// Чтение входных данных

using (StreamReader sr = new StreamReader("input.txt"))

{

string[] mn = sr.ReadLine().Split();

m = int.Parse(mn[0]);

n = int.Parse(mn[1]);

string[] x = sr.ReadToEnd().Split(new char[] { ' ', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

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

{

X.Add(int.Parse(x[i]));

}

}

// Определение выигрышных ходов

List<Tuple<int, int>> winningMoves = new List<Tuple<int, int>>();

bool[,] winningPositions = new bool[m + 1, n + 1];

for (int i = m - 1; i >= 0; i--)

{

for (int j = n - 1; j >= 0; j--)

{

if (!winningPositions[i, j])

{

winningMoves.Add(new Tuple<int, int>(i + 1, j + 1));

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

{

for (int l = 0; l < j; l++)

{

winningPositions[k, l] |= !winningPositions[i, j];

}

}

}

}

}

winningMoves.Reverse();

// Вывод результатов в файл

using (StreamWriter sw = new StreamWriter("output.txt"))

{

sw.WriteLine("для игры \"отравленный пирог\" " +

"используется прямоугольный пирог, разделенный на M \"строк\" " +

"горизонтальными разрезами и на N столбцов- вертикальными. т" +

"аким образом, пирог должен быть разбит на M на N клеток," +

" правая нижняя из которых \"отравлена\". Играют двое игроков, " +

"ходы делаются по очереди. Каждый ход заключается в том," +

" что игрок выбирает еще одну из несъеденных клеток пирога и съедает все клетки," +

" расположеные левее и выше выбранной (в том числе и выбранную). " +

"проигрывает тот кто съедает отравленную клетку.\r\n " +

"которая по заданной игровой позиции определяет " +

"все возможные выигрышные ходы для начинающего в этой позиции. " +

"\r\n\r\nданные во входном файле расположены в следующем порядке:" +

" M, N (1<=M, N<=9), X1, ..., Xm. З" +

"десь Xi - число оставшихся клеток в i-м снизу горизонтальном ряду." +

" Все числа во входном файле разделяются пробелами и/или символами перевода строки." +

"\r\nВыходные данные: в первую строку необходимо вывести количество " +

"различных выигрышных ходов K, а последующие K строк - сами выигрышные ходы. " +

"\r\nКаждый ход задается парой чисел (i,j), где i-номер (снизу) горизонтального ряда, " +

"а j-номер (справа) вертикального ряда, которому принадлежит выбранная клетка " +

"(1<=i<=m, 1<=j<=n)");

sw.WriteLine(winningMoves.Count);

foreach (var move in winningMoves)

{

sw.WriteLine(move.Item1 + " " + move.Item2);

}

}

}

}

}

источник https://tproger.ru/articles/dynprog-starters/