ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
«ПРИДНЕСТРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Т. Г. ШЕВЧЕНКО»

Физико-технический институт
Факультет информатики и вычислительной техники

Кафедра программного обеспечения вычислительной техники

ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Тирасполь
Издательство Приднестровского Университета
2025

УДК 004.657
ББК 32.81я73

Составитель:
А. В. Кирсанова, доцент кафедры программного обеспечения вычислительной техники ФТИ ГОУ ПГУ им. Т. Г. Шевченко

Рецензенты:
С. В. Помян, доцент кафедры программного обеспечения вычислительной техники ФТИ ГОУ ПГУ им. Т. Г. Шевченко
А. П. Рачук, зам. начальника службы информационных технологий СЗАО «Интерднестрком»

Динамическое программирование: учебное пособие [Электронный ресурс] / ГОУ «Приднестровский государственный университет им. Т. Г. Шевченко», Физико-технический институт; Факультет информатики и вычислительной техники; составитель: А. В. Кирсанова. – Тирасполь: Изд-во Приднестр. ун-та, 2025. – 68 с.

В пособии описан подход к решению сложных задач по отысканию оптимального решения путём разбиения их на более простые подзадачи. Дан теоретический материал, множество примеров, приведены решения большого количества задач.

Пособие предназначено студентам направления «Программная инженерия», «Информационные системы и технологии». Изучается в дисциплинах «Алгоритмы и структуры данных» и «Алгоритмы обработки данных».

УДК 004.657
ББК 32.81я73

Рекомендовано Научно-методическим Советом ПГУ им. Т. Г. Шевченко

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современные задачи информатики часто требуют оптимизации вычислений для достижения высокой производительности при обработке сложных данных. Одним из распространенных подходов к решению таких задач является метод динамического программирования (ДП), который позволяет эффективно решать задачи с перекрывающимися подзадачами и оптимальной подструктурой. Динамическое программирование нашло широкое применение в различных областях, таких как оптимизация процессов, анализ данных, разработка алгоритмов для задач комбинаторики, графов и планирования.

Метод динамического программирования иногда ставят в один ряд с рекурсивными алгоритмами, однако он избегает повторных вычислений за счёт сохранения промежуточных результатов в таблице или массиве, что существенно снижает временную сложность. В отличие от рекурсии, которая может приводить к экспоненциальному росту числа операций, ДП обеспечивает полиномиальную сложность для многих задач, что делает его предпочтительным выбором в ситуациях, где производительность критична.

Задача данного пособия показать применение метода ДП для решения разнообразных задач, включая сравнение с рекурсивными подходами.

Актуальность темы обусловлена необходимостью эффективного решения задач оптимизации и обработки данных в условиях роста их сложности и объёма. Динамическое программирование является фундаментальным инструментом, широко применяемым в разработке программного обеспечения, машинном обучении и анализе больших данных, что подчёркивает важность глубокого изучения и практической реализации данного метода.

Например, в системах управления базами данных, таких как *PostgreSQL* или *Apache Spark*, алгоритмы ДП оптимизируют сложные запросы, включая задачи поиска наибольшей общей подпоследовательности или распределения вычислительных ресурсов.

В практических приложениях ДП часто встречается в специализированных системах. Данный метод применяется для анализа генетических последовательностей, в логистике – для оптимизации маршрутов доставки, а в финансовом моделировании – для оценки

инвестиционных стратегий. Однако существующие программные решения, использующие ДП, нередко ограничены узкой специализацией. Многие из них разрабатываются под конкретные задачи, такие как обработка последовательностей или планирование, и распространяются по платным подпискам, что затрудняет их использование в образовательных или универсальных целях.

Актуальность ДП усиливается благодаря его гибкости и возможности адаптации к современным технологиям, включая параллельные вычисления и многоядерные архитектуры. Метод позволяет решать сложные задачи с высокой производительностью, что особенно важно в системах реального времени, таких как системы рекомендаций или автоматизированное управление. Изучение и практическая реализация ДП остаются важными для подготовки специалистов, способных разрабатывать эффективные решения для текущих и будущих вызовов информационных технологий.

1 ОБЩИЕ СВЕДЕНИЯ О МЕТОДЕ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ И ЕГО ПРИМЕНИМОСТИ

Понятие *динамическое программирование* ввел в конце 50-х годов американский математик Ричард Беллман для характеристики алгоритмов, действующих в зависимости от меняющейся ситуации. Главным моментом в его подходе является использование полиномиальных по времени алгоритмов вместо экспоненциальных.

Динамическое программирование – метод оптимизации, приспособленный к задачам, в которых процесс принятия решений может быть разбит на отдельные этапы (шаги). Такие задачи называют многошаговыми.

Метод состоит в том, что оптимальное решение строится постепенно, шаг за шагом. На каждом шаге оптимизируется решение только этого шага, но решение выбирается с учетом последствий, так как решение, оптимальное для этого шага, может привести к неоптимальному решению всей задачи, т. е. оптимальное решение задачи содержит оптимальные решения ее подзадач.

Метод ДП – один из наиболее мощных и широко известных математических методов современной теории управления, он быстро получил широкое распространение.

Р. Беллман сформулировал принцип оптимальности: «*Каково бы ни было начальное состояние на любом шаге и решение, выбранное на этом шаге, последующие решения должны выбираться оптимальными относительно состояния, к которому придет система в конце данного шага».*

Использование этого принципа гарантирует, что решение, выбранное на любом шаге, является не локально лучшим, а лучшим с точки зрения задачи в целом.

Данный метод усовершенствует выполнение задач, решаемых, например, с помощью рекурсий или перебора вариантов.

Условия применения динамического программирования:

1. Небольшое число подзадач. Уменьшение числа подзадач происходит из-за многократного их повторения (так называемые перекрывающиеся подзадачи).

2. Дискретность (неделимость) величин, рассматриваемых в задаче.

Главная особенность алгоритмов рекурсивных алгоритмов заключается в том, что они делят задачи на *независимые* подзадачи. Когда подзадачи независимы, ситуация усложняется, и в первую очередь потому, что непосредственная рекурсивная реализация даже простейших алгоритмов этого типа может требовать недопустимо больших затрат времени.

Метод динамического программирования предназначен для решения задач, которые можно разделить на перекрывающиеся подзадачи с оптимальной подструктурой. Этот подход широко применяется в задачах оптимизации, комбинаторики, анализа графов и других областях информатики, где требуется минимизация или максимизация целевой функции.

Основная идея ДП заключается в сохранении промежуточных результатов вычислений во избежание повторного решения одних и тех же подзадач, что существенно снижает временную сложность по сравнению с рекурсивным подходом.

Метод ДП применим, если в процессе решения задачи:

1) выполняется свойство оптимальности подзадач;

2) выполняется свойство перекрытия подзадач.

Динамическое программирование основано на принципе оптимальности Беллмана: оптимальное решение задачи можно построить из оптимальных решений её подзадач.

Алгоритм, основанный на методе ДП, состоит из:

а) подзадач;

б) способа пересчета;

в) порядок решения подзадач (обычно порядок очень прост, но не всегда).

Динамическое программирование принципиально исключает все повторные вычисления в любой рекурсивной программе при условии, что мы можем себе позволить сохранять значения функции для аргументов, которые меньше, чем текущий вызов.

Свойство. Динамическое программирование уменьшает время выполнения рекурсивной функции максимум до значения, необходимого для вычисления функции для всех аргументов, меньших или равных данному аргументу, при условии, что стоимость рекурсивного вызова остается постоянной.

Математически задача описывается рекуррентным соотношением, которое определяет, как решение для текущего состояния зависит от решений для предыдущих состояний.

Рекуррентное соотношение – это рекурсивная функция с целочисленными значениями. Значение любой такой функции можно определить, вычисляя все значения этой функции, начиная с наименьшего, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения. Эта технология называется **восходящим динамическим программированием** (*bottom-up dynamic programming*). Она применима к любому рекурсивному вычислению при условии, что мы можем себе позволить хранить все ранее вычисленные значения. Такая технология разработки алгоритмов успешно используется для решения широкого круга задач. Поэтому следует уделить внимание простой технологии, которая может изменить зависимость времени выполнения алгоритма с экспоненциальной на линейную.

Нисходящее динамическое программирование (*top-down dynamic programming*) – еще более простая технология, которая позволяет автоматически выполнять рекурсивные функции при том же (или меньшем) количестве итераций, что и восходящее динамическое программирование. При этом мы вводим в данную рекурсивную программу инструментальные средства, обеспечивающие сохранение каждого вычисленного ею (в качестве заключительного действия) значения и проверку сохраненных значений во избежание повторного вычисления любого из этих значений (в качестве ее первого действия). Иногда нисходящее динамическое программирование называют также мемуаризацией (*memoization*).

При использовании нисходящего динамического программирования известные значения сохраняются; при использовании восходящего динамического программирования они вычисляются заранее. В общем случае предпочтение отдается нисходящему динамическому программированию перед восходящим, поскольку:

– оно представляет собой механическое преобразование естественного решения задачи;

– порядок решения подзадач определяется сам собой;

– решение всех подзадач может не потребоваться.

Приложения, в которых применяется динамическое программирование, различаются по сущности подзадач и объему сохраняемой для них информации.

Критический момент, который мы не можем игнорировать, состоит в том, что динамическое программирование становится неэффективным, когда количество возможных значений функции, которые могут нам потребоваться, столь велико, что мы не можем себе позволить сохранять их (при нисходящем программировании) или вычислять предварительно (при восходящем программировании).

Итак, пусть $S(n)$ – решение задачи для входных данных размера n, а $S(k)$ – решения для подзадач меньшего размера $k < n$. Тогда рекуррентное соотношение выражается как:

$$S(n) = f\left(S(n-1), S(n-2), \ldots, S(n-k)\right),$$

где f – функция, комбинирующая решения подзадач, а базовые случаи $S(0), S(1), \ldots$ задаются явно. В ДП результаты подзадач сохраняются в таблице $dp[n]$ (или многомерной таблице), которая заполняется либо сверху вниз (мемоизация), либо снизу вверх (табуляция).

Рекурсивный подход решает задачу, разбивая её на подзадачи и вызывая себя для каждой из них. При наличии перекрывающихся подзадач это приводит к экспоненциальной сложности из-за многократного вычисления одних и тех же значений. Например, для вычисления *n*-го числа Фибоначчи рекуррентное соотношение:

$$F(n) = F(n-1) + F(n-2)$$

с базовыми случаями

$$F(0) = 0 \ , \ \ F(1) = 1 \ ,$$

в рекурсивной реализации требует $O(2^n)$ операций. В ДП используется таблица ($dp[i] = dp[i-1] + dp[i-2]$, где каждое значение вычисляется ровно один раз, что даёт сложность $O(n)$. Таким образом, ДП преобразует экспоненциальную сложность в линейную.

2 ЗАДАЧИ, ПОДХОДЯЩИЕ ДЛЯ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Задачи, подходящие для динамического программирования, характеризуются перекрывающимися подзадачами, оптимальной подструктурой и конечным числом состояний, которые можно описать таблицей фиксированного размера. Перекрывающиеся подзадачи означают, что решение задачи зависит от решений одних и тех же подзадач, используемых многократно. Оптимальная подструктура подразумевает, что оптимальное решение задачи строится из оптимальных решений подзадач. Примеры таких задач включают вычисление чисел Фибоначчи, задачу о рюкзаке, нахождение наибольшей общей подпоследовательности, поиск кратчайшего пути в графе и разбиение строки на слова.

Ниже приведён список из 35 задач, которые могут быть решены с использованием динамического программирования:

1. Числа Фибоначчи: Найти n-е число Фибоначчи, где каждое число является суммой двух предыдущих.
2. Биномиальные коэффициенты. Вычислить количество способов выбрать k элементов из n.
3. Задача о рюкзаке. Выбрать предметы с максимальной суммарной стоимостью, не превышая вес рюкзака.
4. Наибольшая возрастающая подпоследовательность. Дан массив чисел, найти длину наибольшей строго возрастающей подпоследовательности.
5. Наибольшая общая подпоследовательность. Даны две строки, найти длину их наибольшей общей подпоследовательности.
6. Минимальное количество монет. Дана сумма и номиналы монет, найти минимальное количество монет для набора суммы.
7. Разбиение строки на слова. Дана строка и словарь, проверить, можно ли разбить строку на слова из словаря.
8. Кратчайший путь в графе. Найти кратчайшие пути между всеми парами вершин в графе.
9. Задача о покрытии отрезков. Даны отрезки на прямой, найти минимальное количество точек, покрывающих все отрезки.
10. Максимальная сумма подмассива. Дан массив чисел, найти подмассив с максимальной суммой.

11.Число путей в сетке. Найти количество путей из верхнего левого угла в нижний правый в сетке, двигаясь только вправо или вниз.

12.Задача о скобках. Найти количество правильных скобочных последовательностей заданной длины.

13.Задача о размене монет. Дана сумма и номиналы монет, найти количество способов набрать сумму.

14.Максимальный квадрат. Дана матрица из 0 и 1, найти размер наибольшего квадрата из 1.

15.Максимальный прямоугольник. Дана матрица из 0 и 1, найти площадь наибольшего прямоугольника из 1.

16.Задача о путях с препятствиями. Найти количество путей в сетке с препятствиями, двигаясь только вправо или вниз.

17.Наибольшая возрастающая подматрица. Дана матрица, найти размер наибольшей квадратной подматрицы с возрастающими элементами.

18.Задача о редакционном расстоянии. Даны две строки, найти минимальное количество операций для преобразования одной строки в другую.

19.Задача о разрезе стержня. Дан стержень и цены за куски разной длины, найти максимальную стоимость разрезания.

20.Задача о триангуляции. Дан выпуклый многоугольник, найти минимальную стоимость его триангуляции.

21.Задача о матричной цепочке. Даны матрицы, найти минимальное количество операций для их перемножения.

22.Задача о палиндроме. Найти длину наибольшей палиндромной подстроки в данной строке.

23.Задача о подмножестве с суммой. Дан массив и сумма, проверить, существует ли подмножество с заданной суммой.

24.Задача о лестнице. Найти количество способов подняться на n-ю ступеньку, шагая на 1 или 2 ступеньки.

25.Задача о покраске забора. Найти количество способов покрасить забор из n досок в k цветов, чтобы соседние доски имели разные цвета.

26.Задача о домино. Найти количество способов покрыть доску размером 2 на n домино размером 1 на 2.

27.Задача о раскраске графа. Найти количество способов раскрасить граф в k цветов так, чтобы соседние вершины имели разные цвета.

28.Задача о разбиении числа. Найти количество способов представить число как сумму чисел из заданного множества.

29.Задача о максимальной независимости. Найти размер наибольшего независимого множества в дереве.

30.Задача о минимальном покрытии. Найти минимальное вершинное покрытие в двудольном графе.

31.Задача о разбиении массива. Дан массив, разбить его на k подмассивов с минимальной максимальной суммой.

32.Задача о расписании. Даны задачи с дедлайнами и штрафами, найти минимальный штраф за просрочку.

33.Задача о разрезе графа. Найти минимальный разрез в графе, разделяющий его на две части.

34.Задача о покрытии точками. Даны точки на плоскости, найти минимальное количество кругов заданного радиуса, покрывающих все точки.

35.Задача о максимальной сумме пути. Дана треугольная матрица, найти путь от вершины к основанию с максимальной суммой.

Предлагаем в качестве упражнения проделать работу, аналогичную той, что проведена до списка для задачи о рюкзаке.

3 ОБЩИЕ РЕКОМЕНДАЦИИ К РЕШЕНИЮ ЗАДАЧ МЕТОДОМ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

При использовании метода динамического программирования рекомендуется сохранять промежуточные результаты, чтобы избежать повторных вычислений, причем выбор итеративного подхода (снизу вверх) или мемоизации (сверху вниз) зависит от специфики задачи, выбор должен обеспечить оптимальную производительность.

Для реализации задач определяются основные наборы данных: входные параметры, такие как массивы чисел, строки или матрицы, и таблицы для хранения промежуточных результатов. Например, в задаче о рюкзаке желательно использовать массив весов предметов, массив их стоимостей, вместимость рюкзака и двумерная таблица,

где каждый элемент отражает максимальную стоимость для подмножества предметов и доступного веса. Для других задач, таких как наибольшая возрастающая подпоследовательность, используется одномерный массив, хранящий длины подпоследовательностей для каждого индекса массива.

Алгоритм начинается с инициализации таблицы базовыми случаями, зависящими от условий задачи. Например, в задаче о рюкзаке таблица заполняется нулями для случаев, когда нет предметов или вес равен нулю. Затем алгоритм итеративно заполняет таблицу, вычисляя значения на основе предыдущих результатов. Для каждого предмета и доступного веса определяется, включать ли предмет, чтобы максимизировать стоимость. Процесс учитывает оптимальные решения подзадач, сохранённые в таблице, что исключает необходимость повторных вычислений.

По завершении вычислений алгоритм извлекает итоговое решение из таблицы. В задаче о рюкзаке – это максимальная стоимость, а для восстановления выбранных предметов алгоритм проходит по таблице в обратном порядке, определяя, какие предметы были включены. Аналогичный подход применяется к другим задачам. Таблица заполняется в порядке возрастания параметров, а результат извлекается из последней ячейки или путём анализа сохранённых данных.

4 РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ МЕТОДА ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

4.1. Вычисление *n*-го числа Фибоначчи

Формулировка задачи: вычислить *n*-й член последовательности Фибоначчи. Первый и второй ее члены равны 0 и 1, а каждый следующий равен сумме двух предыдущих. Приведем код решения задачи на языке C# с использованием метода динамического программирования.

```
public string Solve(IReadOnlyDictionary<string, object> pa-
rameters)
   {
```

```
        int sequenceIndex = int.Parse(parame-
ters["n"].ToString() ?? throw new InvalidOperationExcep-
tion("Parameter 'n' is missing"));

        if (sequenceIndex <= 1)
        {
            return sequenceIndex.ToString();
        }

        int[] fibonacciNumbers = new int[sequenceIndex + 1];
        fibonacciNumbers[0] = 0; // Первое число Фибоначчи
        fibonacciNumbers[1] = 1; // Второе число Фибоначчи

        for (int currentIndex = 2; currentIndex <= se-
quenceIndex; currentIndex++)
        {
            fibonacciNumbers[currentIndex] = fibonacci-
Numbers[currentIndex - 1] + fibonacciNumbers[currentIndex -
2]; // Суммируем два предыдущих числа
        }

        return fibonacciNumbers[sequenceIndex].ToString();
    }
```

Как видно, рекуррентное соотношение имеет вид

```
fibonacciNumbers[currentIndex] =
fibonacciNumbers[currentIndex-1]+fibonacciNumbers[cur-
rentIndex-2];
```

и в задаче применены и соблюдены озвученные выше принципы Беллмана.

4.2. Задача о максимальной возрастающей подпоследовательности

Для заданного массива $A[1,...n] = \{4, 5, 7, 2, 1, 13, 18\}$ найти элементы, представляющие возрастающую последовательность максимальной длины. Для нашего примера результат представлен на рисунке.

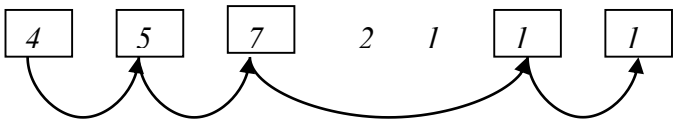

Рисунок 1 – Пример последовательности максимальной длины

Известен простой алгоритм, однако он может быть неэффективным. Процесс решения с помощью метода ДП представляет собой последовательность следующих шагов:

1. Создать массив *D* [1,…*n*], в каждый элемент которого *D[j]* будет записана длина максимальной возрастающей подпоследовательности, заканчивающейся в *A[j]*.

Для нашего примера элементы массива *D* будут иметь такие значения:

A: 4, 5, 7, 2, 1, 13, 18

D: 1, 2, 3, 1, 1, 4, 5

A: 4, 5, 7, 2, 1, 5*, 18, 3

D: 1, 2, 3, 1, 1, 2, 4, 2

2. Определить подзадачу и найти способ пересчета.

Для примера возьмем число 18. Необходимо определить, какова максимальная подпоследовательность, которая заканчивается на этом элементе. Для этого необходимо определиться, какой элемент может быть предпоследним (перед 18) этой последовательности, очевидно, что это числа, стоящие перед 18 и меньшие 18. Рассмотрим, например, абстрактный массив.

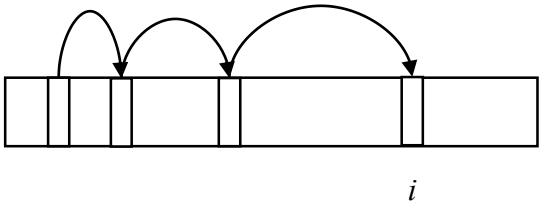

Рисунок 2 – Абстрактный массив

Рассмотрим предыдущий (перед *i*) элемент этой возрастающей подпоследовательности. Неизвестно его положение, но решение основано на том, что предыдущая часть этой подпоследовательности перед предпоследним тоже обязана быть оптимальной (самой длинной). Она обязана быть оптимальной, так как если бы это было не так и существовала другая подпоследовательность, заканчивающаяся в этом элементе, то эта последовательность и была бы взята, и тогда вся последовательность, включая *i*-й лемент, была длиннее.

Таким образом, для данной задачи сформулировано свойство оптимальности подзадач. Если имеется максимальная возрастающая подпоследовательность, заканчивающаяся в *i*-м элементе, то если

отбросить *i*-й элемент, часть все равно останется оптимальной, но длина на 1 меньше. Это и есть то свойство, которое позволяет решить данную задачу с помощью ДП.

Приведем псевдокод решения данной задачи (здесь $D[j]$ – это максимальная возрастающая подпоследовательность заканчивающаяся в $A[j]$).

```
D[1,…n]
D[1] = 1
for i = 2 to n
    D[i] = 1 + max {D[j]: 1≤j≤i-1, A[j]<A[i]}
return max {D[j]: 1≤j≤n}
```

В цикле осуществляется перебор всех возможных кандидатов на то, чтобы быть предыдущим перед *i* элементом в максимальной возрастающей подпоследовательности. Этим предыдущим может быть любой с первого элемента до (*i*-1)-го, который строго меньше *i*-го. Причем эта подпоследовательность обязана быть оптимальной, ее длина $D[j]$. Единицу необходимо прибавлять, так как необходимо учесть сам *i*-й элемент. В программе необходимо написать еще один цикл где *j* будет изменяться от 1 до *i*-1 и изначально переменная максимально будет установлена в нуль, а затем она может и увеличиваться и это запишется в $D[i]$.

Время работы $O(n^2)$, но можно реализовать и за время $\Theta(n^2)$.

Приведем несколько замечаний об этом алгоритме.

1. Для каждого *D[i]* необходимо знать несколько предыдущих *D[j]* – это и есть перекрытие подзадач, а каких именно – зависит от того, что находится в *i*-й ячейке.

Можно написать рекурсивную функцию, но тогда необходимо выполнять один и тот же пересчет много раз. Улучшение заключается в реализации рекурсии с запоминанием (мемоизация). Т. е. в любом случае необходимо хранить массив *D* и рекурсивно его рассчитывать следующим образом: если в *D* хранится, например, (-1), который означает, что еще не выполнен расчет, то выполняем рекурсивный расчет. Иначе – просто выводится элемент. В данном алгоритме такие действия бессмысленны. Ниже будет приведен пример, когда наличие такой рекурсии имеет смысл.

2. Предположим, что алгоритм нашел длину максимальной подпоследовательности, но необходимо решить еще вопрос отыскания самой подпоследовательности. Для этого необходимо создать еще

один массив *prev*, в который записывается индекс предыдущего элемента в оптимальной последовательности, заканчивающейся в *A[j]*. Т. е. в псевдокоде между первой и второй строкой необходимо добавить следующий фрагмент:

```
prev [1…n]
prev[1] = n/a //не определено
```

а в цикле добавить строку:

```
prev[i] (после строки D[i]=1+max{…}).
```

Сама подпоследовательность будет получена с конца, но можно и инвертировать массив.

4.3. Задача о стоимости редактирования (edit distance)

Стоимость редактирования – это число, которое определяется для двух слов и говорит нам насколько два слова похожи, т. е. чем меньше это расстояние – эта стоимость, тем более похожи эти два слова.

Например, при опечатке встроенный фрагмент программы должен подчеркнуть слово с ошибкой и предложить разумную замену, т. е. слово близкое к тому, в котором допущена ошибка. Ошибкой может быть пропуск символа, замена символа, лишний символ. Значит необходимы операции вставки, удаления и замены символов.

Пример 1: вместо слова «палиндром» написано «полиномы».

```
замена      вставка   удаление
   *        *  *       *
П О Л И Н  -  - О М Ы
П А Л И Н Д Р О М
```

Как видно символ О надо заменить на А, символы Д и Р надо вставить, символ Ы надо удалить. Такое изображение одного слова над другим называется *выравниванием*. Стоимость редактирования равна 4.

Пример 2: вместо слова «палиндром» написано «полиномы » (с тремя пробелами).

П О Л И Н О М Ы - - - -
П А Л И Н - - - Д Р О М

Стоимость равна 8. Стоимость выравнивания равна количеству позиций, в которых написаны различные символы. Пропуск сверху означает, что вставляется символ, а снизу – удаляется.

В общем случае сформулируем задачу так: пусть даны два слова $A[1…p]$ и $B[1…q]$. Необходимо найти стоимость редактирования.

1. Определить подзадачи. Создать матрицу D, куда будут записаны стоимость редактирования $A[1…i]$ и $B[1…j]$. Псевдокод приведен ниже.

```
A[1…p]
B[1…q]
D[1…p, 0…q] //D[i, j] – стоимость редактирования
for i = 0 to p // инициализация заполняет нулевой столбец
     D[i, 0] = i
for j = 0 to q // инициализация заполняет нулевую строку
    D[0, j] = j
```

После составления таблицы редактирования буден получена стоимость редактирования.

Таблица 1 – Таблица редактирования
для получения стоимости редактирования

		П	А	Л	И	Н	Д	Р	О	М
	0	1	2	3	4	5	6	7	8	9
П	1									
О	2									
Л	3									
И	4									
Н	5									
О	6									
М	7									
Ы	8									

В отмеченные клетки записываются стоимость выравнивания префиксов. Выполним эти же действия для префиксов «полино» и «палин». Подзадачи будут расти по длине и в итоге в клетку [8, 9] запишется стоимость редактирования слов целиком. Префикс ПОЛИНО выравниваем с ПАЛИН.

Оптимальное выравнивание данных префиксов приведено в таблице.

Таблица 2 – Оптимальное выравнивание префиксов ПОЛИНО и ПАЛИН

П	О	Л	И	Н	О
П	А	Л	И		Н

Идея, как в предыдущем случае. Неизвестна максимальная подпоследовательность, заканчивающаяся в *i*-м элементе, но известно, что если *i*-й элемент последовательности отрезать, то последовательность останется оптимальной. Это свойство оптимальности подзадач. В этом примере так же, как в предыдущем, если в выравнивании последний столбец отрезать, то оставшаяся часть будет оптимальным выравниванием слов без отрезанной части. Что именно отрезалось, тоже неизвестно, но вариантов не так много (см. рис. 3).

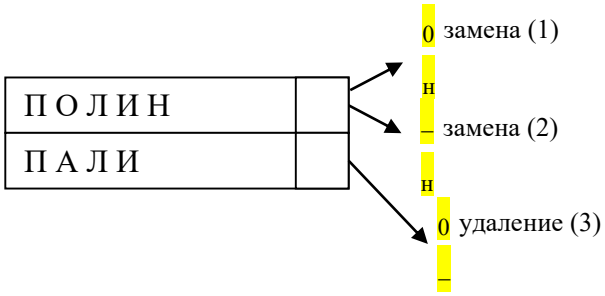

Рисунок 3 – Оптимальное выравнивание

Как видно имеются три способа (ситуации) выравнивания последнего столбца. В любой ситуации известно, что всё, что слева – это оптимальное выравнивание префиксов на один короче. Т. е. оставшееся в этих ситуациях будет:

1. *D[i*-1*, j*-1*] – слева-сверху.*
2. *D[i, j*-1*] – слева.*
3. *D[i*-1*, j] – сверху.*

Таким образом, чтобы найти элемент [6; 5] необходимо обратиться к элементам сверху, слева и слева-сверху ([5; 4], [6; 4], [5;5]). Т. е. в общем случае, чтобы найти элемент [*i; j*], необходимо обратиться к элементам [*i*-1; *j*], [*i*; *j*-1] и [*i*-1; *j*-1]. В тот момент, когда будет найдено значение элемента *[i; j],* необходимо знать значение указанных выше трех элементов. Массив можно заполнять по строкам, по столбцам. Продолжение псевдокода представим ниже.

```
for i=1 to p
for j=1 to q
     D[i,j] = min {D[i-1, j]+1, D[i, j-1]+1, D[i-1, j-
1]+(A[i]= =B[j]?0:1)}
return D[p, q].
```

Результат заполнения массива (таблицы).

Таблица 3 – Оптимальное выравнивание префиксов ПАЛИНДРОМ и ПОЛИНОМ

		П	А	Л	И	Н	Д	Р	О	М
	0	1	2	3	4	5	6	7	8	9
П	1	0	1	2	3	4	5	6	7	8
О	2	1	1	2	3	4	5	6	6	7
Л	3	2	2	1						
И	4									
Н	5									
О	6									
М	7									
Ы	8									

Аналогичные действия выполним и для слов КОТ и СКОТ.

Таблица 4 – Оптимальное выравнивание префиксов КОТ и СКОТ

		К	О	Т
	0	1	2	3
С	1	1	2	3
К	2	1	2	3
О	3	2	1	2
Т	4	3	2	1

Код программы решения данной задачи на языке C# приведен ниже.

```
public string Solve(IReadOnlyDictionary<string, object> pa-
rameters)
     {
       string firstString = parame-
ters["firstString"].ToString() ?? throw new InvalidOpera-
tionException("Parameter 'firstString' is missing");
       string secondString = parameters["sec-
ondString"].ToString() ?? throw new InvalidOperationExcep-
tion("Parameter 'secondString' is missing");

        int firstLength = firstString.Length;
        int secondLength = secondString.Length;

         int[,] editDistances = new int[firstLength + 1,
secondLength + 1];
     // Инициализация первой строки и столбца
         for (int firstIndex = 0; firstIndex <=
firstLength; firstIndex++)
          {editDistances[firstIndex, 0] = firstIndex; //
Удаление символов из первой строки}

         for (int secondIndex = 0; secondIndex <= second-
Length; secondIndex++)
           {editDistances[0, secondIndex] = secondIndex; //
Вставка символов во вторую строку}

         // Вычисление минимального количества операций
          for (int firstIndex = 1; firstIndex <=
firstLength; firstIndex++)
            {for (int secondIndex = 1; secondIndex <= sec-
ondLength; secondIndex++)
               {if (firstString[firstIndex - 1] == sec-
ondString[secondIndex - 1])
                 {editDistances[firstIndex, secondIndex] =
editDistances[firstIndex - 1, secondIndex - 1]; // Символы
совпадают, операция не нужна
                  }
                 else
                   {editDistances[firstIndex, secondIndex]
= Math.Min(Math.Min
     (editDistances[firstIndex - 1, secondIndex] + 1, //
Удаление
      editDistances[firstIndex, secondIndex - 1] + 1 //
Вставка
     ),
```

```
      editDistances[firstIndex - 1, secondIndex - 1] + 1 //
Замена
                        );
                }
            }
        }

     return editDistances[firstLength, second-
Length].ToString();
     }
```

В качестве подзадач можно было рассмотреть $D[i_1, i_2, j_1, j_2]$ = оптимальная стоимость редактирования подстрок. $A[i_1, i_2]$ и $B[j_1, j_2]$. Но матрица D – четырехмерная, и заполнение сложнее и работа дольше $(p^2 \cdot q^2)$; а A и B – двумерные. Время работы $O(p \cdot q)$.

Заметим также, что необязательно хранить всю матрицу, так как, например, при построчном заполнении достаточно хранить текущую и предыдущую строки. Это оправдано при очень длинных строках. Например, задача отыскания расстояния между двумя геномами. Но операция замены в биоинформатике встречается редко (для нуклеотидов).

4.4. Задача о рюкзаке

Рассмотрим задачу о рюкзаке (0/1 *Knapsack Problem*). Дано n предметов, каждый с весом w_i и стоимостью c_i, и рюкзак вместимостью W. Нужно максимизировать суммарную стоимость выбранных предметов, не превышая вес W. Пусть $S(i, w)$ – максимальная стоимость, которую можно получить, рассматривая первые i предметов и вес w. Рекуррентное соотношение:

$$S(i,w) = \max S(i-1,w), c_i + S(i-1, w-w_i), \text{ если } w_i \le w,$$
$$\text{иначе } S(i,w) = S(i-1,w)$$

Базовые случаи: $S(0,w) = 0$, $S(i,0) = 0$. Для решения с помощью ДП создаётся таблица $dp[i][w]$ где $i = 0 \ldots n$, $w = 0 \ldots W$. Заполняем таблицу:

для $i = 0$ или $w = 0$, $dp[i][w] = 0$;

иначе $dp[i][w] = \max dp[i-1][w], vc_i + dp[i-1][w-w_i]$,

если $w_i \le w$, или $dp[i][w] = dp[i-1][w]$.

Итоговое решение: $dp[n][W]$. Временная сложность: $O(n \cdot W)$, что значительно лучше рекурсивной сложности $O(2^n)$.

Пример использования задачи о рюкзаке: входные данные $n = 3$, $w = [10, 20, 30]$, $c = [60, 100, 120]$, $W = 50$. Заполнение таблицы dp : $dp[1][50] = 60$ (берём предмет 1), $dp[2][50] = 160$ (берём предметы 1 и 2), $dp[3][50] = 220$ (берём предметы 2 и 3). Итог: максимальная стоимость = 220 (предметы с весами 20 и 30).

Сформируем псевдокод решения данной задачи. Пусть *C*[] и *W*[] – массивы исходных данных, состоящие из натуральных чисел. Определим подзадачу исходной задачи, пусть *A[w]* – оптимальная стоимость предметов, которую можно поместить в рюкзак объема *W*. Пусть Часть массива известна (рисунок 4).

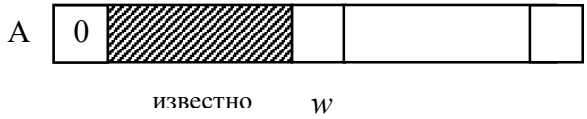

Рисунок 4 – Пример массива для задачи рюкзака

Тогда псевдокод имеет вид:

```
A[0]=0
for w=1 to W
   A[w]= max_{w>w_i}  {A[w-wi]+ci}
return A[w]
```

Опять для поиска максимума необходимо организовать отдельный цикл.

Для лучшего понимания, сделаем всё на примере: имеется рюкзак объемом *W*=10, а предметы имеют характеристики веса *w*: 8, 1, стоимости *c:* 5, 2 и *n* = 11.

	0	1	2	3	4	5	6	7	8	9	10
A	0	2	4	6	8	10	12	14	16		

Рисунок 5 – Пример массива для задачи рюкзака объемом 10

Так для $A[8] = max\{A[0] + 5, A[7] + 2\} = max\{5, 16\} = 16$.

Однако такая стратегия к оптимальному решению может и не привести. Например, если взят дорогой предмет, но его объем может таков, что больше ничего в рюкзак не поместится. Сложность по времени такого решения $O(n \cdot W)$, что неэффективно.

Время работы зависит от длины входа экспоненциально. Пусть $w = 3$. Т.е. на вход приходит n предметов: $11_2 = 3_{10}$. Если на вход подано $n = 111_2 = 7_{10}$, то время работы увеличится вдвое. При $n = 1111_2 = 15_{10}$ время работы увеличивается еще вдвое. Если дописать десять единиц, то размер входа изменится незначительно, а время увеличивается в 1000 раз, а если дописать 100 единиц, то алгоритм будет работать столько, что не дождемся никогда. Но для этой задачи нет полиномиального решения.

Рассмотрим другую разновидность задачи о рюкзаке, когда брать только по одному разу каждый предмет, т.е. необходимо хранить информацию о том, какой предмет уже был взят. Хранить все возможные сочетания из n предметов невозможно, так как их 2^n штук. Переопределим задачу, чтобы легче было найти способ пересчета, тогда рекуррентное соотношение будет проще. Пусть таблица $A[w, i]$ – оптимальная стоимость первых i предметов для рюкзака объемом w. Другими словами раньше был ограничен только размер рюкзака, а теперь постепенно будем наращивать количество предметов, которое можно использовать. Определимся со способом пересчета.

Необходимо заполнить рюкзак объема w первыми i предметами. Но известно только то, что i-й предмет либо входит, либо не входит в рюкзак. Если он входит, то у останется рюкзак меньшего объема, в котором используется только первые (i-1) предметов. Если он не входит, то используются только первые (i–1) предметов, (но объем будет меньше). То есть необходимо найти максимум из двух случаев

A[w, i] = max{A[w-w_i, i-1]+c_i; A[w, i-1]}, где значение A[w-w_i, i-1]+c_i соответствует случае, если предмет входит;
A[w, i-1] соответствует случае, если предмет не входит .

Алгоритм прост, сначала инициализация, затем последовательно наращивается количество предметов и объем.

```
// инициализация
for i=1 to n
    for w=1 to W
        A[w, i]=max{A[w-wi, i-1] +ci; A[w, i-1]}
```

Рассмотрим пример. Пусть дан рюкзак объема W=10. Характеристики предметов даны в таблице 5.

Таблица 5 – Исходные данные для задачи о рюкзаке

	1	2	3	4
Стоимость, *c*	5	2	10	1
Объем, *w*	3	7	4	2

Заполняем таблицу (*i* – номер предмета; *w* - объем).

Таблица 6 – Решение задачи о рюкзаке

w\i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	10	10
5	0	3	3	10	
6	0	3	3	10	
7	0	3	3	10	
8	0	3	3	10	
9	0	3	3		
10	0	3	3		

Если используем только первый предмет, то записываются нули, если рюкзак объема 4, а с $w=5$ уже помещается первый предмет. Когда заполняли ячейку [4, 1] она ссылалась только на ячейку [4, 0]. В свою очередь ячейка A[5, 1] зависит от [5, 0] и [0, 0], а A[4, 1] = A[4, 0] = 0, при этом $A[5, 1] = max\{A[0, 0]+3, A[5, 0]\} = max\{3, 0\} = 3$

Теперь первый предмет помещается в рюкзак, и его либо следует взять либо нет. Если предмет взят, т. е. в рюкзаке объема 5 лежит предмет объема 5, то останется рюкзак объема нуль и больше никакие предметы невозможно положить в рюкзак и необходимо сохранить информацию о том, что этот предмет стоит 3 рубля. Либо можно ничего не класть в рюкзак объема 5.

Затем заполняется столбец 2 таблицы. Итак, разрешено теперь два предмета использовать. Изначально опять до пятерки ничего помещаться не будет. Потом с пятерки записываем 3.

$A[7, 2] = max\{A[0, 1] + 2, A[7, 1]\} = max\{2, 3\} = 3.$

На данный момент остался рюкзак объема 7 и разрешено использовать первый или второй предмет.

Возможны две ситуации. Первая – можно в рюкзак положить второй предмет объема 7, но тогда останется рюкзак объема ноль, так как текущий объем был 7, а положен второй предмет, объем которого 7. Значит, после этого можно использовать только первый предмет, но на самом деле в рюкзак больше ничего не поместится, а стоимость второго предмета равна двум. Вторая ситуация – второй предмет не используем, тогда на данный момент имеется рюкзак объема 7, а кладем только первый предмет.

$A[7, 2] = max\{A[0, 1] + 2, A[7, 1]\} = max\{2, 3\} = 3.$

Затем по столбцу 2 число 3 так и пройдет дальше, так как первый и второй предметы все равно невозможно положить одновременно.

Заполняем столбец 3.

$A[9, 3] = max\{A[5, 2]+10, A[9, 2]\} = max\{13, 3\} = 13.$

Первая ситуация – допустим третий предмет положили в рюкзак, тогда останется рюкзак объема 9 – 4 = 5, в который можно складывать первый и второй предметы, получаем стоимость 10.

Вторая ситуация – если третий предмет не взят, то рюкзак по-прежнему объема 9 и можно в него складывать первые два предмета.

$A[9, 3] = max\{A[5, 2] + 10, A[9, 2]\} = max\{13, 3\} = 13.$

Теперь появился предмет объема 2 (заполняем столбец 4).

Для этой конкретной задачи есть алгоритм приближенного решения, которое будет не более, чем в два раза хуже, чем оптимальное.

4.5. Задача о перемножении последовательности матриц

Пусть имеется последовательность матриц A_1, A_2, A_3, …, которые требуется перемножить, а m_1*m_2 – размер первой матрицы; m_2*m_3 – размер второй матрицы; ….. m_3*m_n – размер последней матрицы.

Умножение матриц ассоциативно:

$$(A_1 \cdot A_2) \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$$

Но умножение не коммутативно:

$$A_1 \cdot A_2 = A_2 \cdot A_1$$

Способы умножения могут отличаться, так как в зависимости от размеров матрицы тратится разное время. То есть для матриц на рисунке 6 время работы будет равно $O(p \cdot q \cdot r)$.

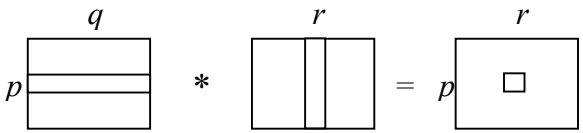

Рисунок 6 – Демонстрация перемножения

Пример: для матриц A_1 размером 2*50, A_2 размером 50*2 и A_3 размером 2*50 при умножении $A_1 \cdot A_2 \cdot A_3$ потребуется $2 \cdot 50 \cdot 2 + 2 \cdot 2 \cdot 50 = 400$ арифметических операций, а при перемножении $A_1 \cdot A_3 \cdot A_2$ потребуется $50 \cdot 2 \cdot 50 + 50 \cdot 2 \cdot 50 = 10000$ арифметических операций.

Порядок матриц при умножении имеет значение. Задача – определить оптимальный порядок перемножения последовательности матриц.

Создадим массив $D[i, j]$ – оптимальная стоимость перемножения матриц $A_i,...,A_j$. Количество способов расстановки скобок при перемножении очень велико. Разобьем n матриц на последовательные тройки $A_1A_3A_2 \cdot A_4A_5A_6 \cdot ... \cdot A_{n-2}A_{n-1}A_n$, в каждой тройке – два способа, итого способов $2^{\frac{n}{3}}$. Это время очень неэффективное. Перебирать все возможные расстановки долго и потому нерационально.

Пусть необходимо перемножить матрицы $A_i \cdot A_{i+1} \cdot ... \cdot A_j$.

Чтобы найти оптимальный способ их перемножить, надо в последовательности $(A_i \cdot A_{i+1} \cdot ... \cdot A_k)(A_{k+1} ... A_j)$ расставить скобки.

У произведения $(A_i \cdot A_{i+1} \cdot ... \cdot A_k)$ размер $m_i * m_{k+1}$, а у $(A_{k+1} ... A_j)$ размер $m_{k+1} * m_{j+1}$.

Заполним матрицу оптимальной стоимостью перемножения *D* по диагонали (см. рисунок 6)

$$D[i, j] = \min_{i \le k < j} \{D[i,k] + D[k+1, j] + m_i \cdot m_{k+1} \cdot m_{j+1}\}.$$

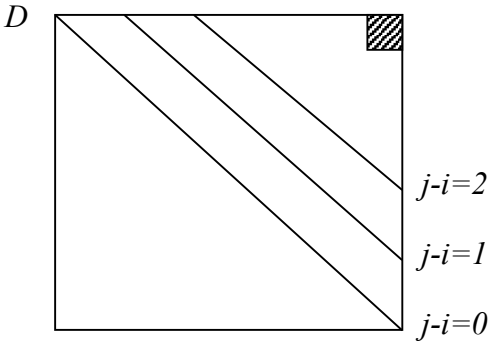

Рисунок 7 – Демонстрация заполнения матрицы максимальной стоимости

Разность номеров столбцов и строк $i - j$ всегда константа. Назовем эту разность размером подзадачи *size*. Тогда псевдокод решения задачи будет следующим:

```
for i=1 to n
    D[i, i]=0 //инициализация
for size=1 to n-1
    for i=1 to n-s
        j=i+size
        D[i, j] = min_{i≤k<j} {D[i,k] + D[k+1, j] + m_i · m_{k+1} · m_{j+1}}
return D[1, n].
```

Время работы n^3, так как необходимо заполнить $n^2/2$ ячеек, а на каждую ячейку требуется *size* времени, то получаем $n^2/2 \cdot n \approx n^3$.

4.6. Задача о сдаче

Пусть имеются монеты 1, 2 и 5 рублей. Необходимо дать сдачу 10 рублей.

Способы, которыми можно дать сдачу перечислим в таблице.

Таблица 7 – Способы, которыми можно дать сдачу

№ способа	Сумма сдачи	Варианты дать сдачу
1 способ	1 рубль	1
2 способа	2 рубля	1+1; 2
3 способа	3 рубля	1+2; 2+1; 1+1+1
5 способов	4 рубля	1+1+1+1; 2+2; 1+1+2; 1+2+1; 2+1+1

Пусть *F(n)* – количество способов возврата сдачи *n* рублей.

Рассмотрим решение на примере $F(10)$. Возврат сдачи начинается с возврата первой монеты. Первая монета может быть: 1, 2 или 5 рублей. Если первая монета – это 1 рубль, то останется вернуть 9 рублей; 2 рубля – останется вернуть 8 рублей; 5 рублей – останется вернуть 5 рублей.

Причем может быть только один из этих трех случаев. Согласно принципу сложения, необходимо сложить количество способов, при которых первая возвращенная монета будет 1 рубль и останется вернуть 9 рублей, с количеством способов вернуть 10 рублей, при котором первая монета будет 2 рубля и останется вернуть 8 рублей, и с количеством способов вернуть 10 рублей, если первая монета 5 рублей и останется вернуть 5 рублей. То есть

$$F(10) = F(9) + F(8) + F(5)$$
$$F(9) = F(8) + F(7) + F(4)$$

И в общем виде:

$$\begin{cases} F(k) = F(k-1) + F(k-2) + F(k-5), & \text{при } k \geq 5 \\ F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5 \end{cases}$$

Продемонстрируем детальное решение этой задачи:

$$F(5) = F(0) + F(3) + F(4) = 1 + 3 + 5 = 9$$
$$F(6) = F(1) + F(4) + F(5) = 1 + 5 + 9 = 15$$
$$F(7) = F(2) + F(5) + F(6) = 2 + 9 + 15 = 26$$
$$F(8) = F(3) + F(6) + F(7) = 3 + 15 + 26 = 44$$
$$F(9) = F(4) + F(7) + F(8) = 5 + 26 + 44 = 75$$
$$F(10) = F(5) + F(8) + F(9) = 9 + 44 + 75 = 128.$$

А теперь алгоритм. Начальные условия даны.

$$F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5$$
$$F(k) = ?$$

Псевдокод:

```
for i=5 to k
    F(i)=F(i-1)+F(i-2)+F(i-5)
// вывод F(k)
```

Время работы алгоритма $\theta(k)$, но если уточнить, то имея на входе данные – сдача k руб. и понимая, что в записи числа k имеется $\log_2 k$ цифр, уточним: $\theta(k) = \theta(2^{\log_2 k})$, т. е. время экспоненциальное.

Программа на языке C# приведена ниже.

```
int k = 10;
int[] coins = { 1, 2, 5 };
int[] dp = new int[k + 1];
dp[0] = 1;
foreach (int coin in coins) // Перебираем все монеты
        {
         for (int i = coin; i <= k; i++)
            {
                dp[i] += dp[i - coin];
            }
        }
Console.WriteLine($"Количество способов дать сдачу
{k} рублей: {dp[k]}");
```

Таким образом, решена классическая задача комбинаторики, однако можно было решить, используя производящие функции.

4.7. Задача об оптимальной сдаче

Данная задача отличается от предыдущей тем, что необходимо найти алгоритм возврата сдачи *k* рублей с помощью минимального количества монет.

Пусть имеются монеты: x_1, x_2, x_3. Необходимо дать сдачу *k* рублей.

Детальная цель – найти минимальное количество монет, с помощью которого можно вернуть сдачу *k* рублей, очевидно, что это задача оптимизации.

Пусть $x_1 < x_2 < x_3$. Сначала формализуем то, что необходимо найти.

Пусть *F(k)* – минимальное количество монет для возврата сдачи размером *k* рублей.

Общая формула для *F(k)* имеет вид:

$$k > x_3 : F(k) = \left[1 + F(k - x_1); \quad 1 + F(k - x_2); \quad 1 + F(k - x_3)\right]$$

Выбор монеты

x_1 даст $1 + F(k - x_1)$ способов;

x_2 даст $1 + F(k - x_2)$ способов;

x_3 даст $1 + F(k - x_3)$ способов.

Однако нужен минимум. Поэтому в общем виде запись будет выглядеть следующим образом:

$$F(k) = \min\left[1 + F(k - x_1);\ 1 + F(k - x_2);\ 1 + F(k - x_2)\right]$$

Самостоятельно напишите формулу для *F(k)* при $k \le x_3$.

Рассмотрим задачу для $k = 9;\ x_1 = 1; x_2 = 4; x_3 = 6$.

Таблица 8 – Данные для задачи об оптимальной сдаче

k	1	2	3	4	5	6	7	8	9
F(k)	1	2	3	1	2	1	2	2	3

По общей формуле

$$F(k) = \min\left[1 + F(k - x_1);\ 1 + F(k - x_2);\ 1 + F(k - x_3)\right]$$

выполним расчеты подробно:

$$F(7) = \min\left[1 + F(7-1);\quad 1 + F(7-4);\quad 1 + F(7-6)\right] =$$
$$= \min\left[\quad 1 + F(6);\quad 1 + F(3);\quad 1 + F(1)\right] =$$
$$= \min\left[\quad 1 + 1;\quad 1 + 3;\quad 1 + 1\right] = 2$$

$$F(8) = \min\left[1 + F(8-1); 1 + F(8-4); 1 + F(8-6)\right] =$$
$$= \min\left[1 + F(7);\ \ 1 + F(4);\quad 1 + F(2)\right] =$$
$$= \min\left[1 + 2;\quad 1 + 1;\quad 1 + 2\right] = 2$$

$$F(9) = \min\left[1 + F(9-1);\ 1 + F(9-4);\ 1 + F(9-6)\right] =$$
$$= \min\left[1 + F(8);\quad 1 + F(5);\quad 1 + F(3)\right] =$$
$$= \min\left[1 + 2;\quad 1 + 2;\quad 1 + 3\right] = 3$$

Таким образом, определено минимальное количество монет, равное 3, а чтобы определить, какие это монеты конкретно, достаточно посмотреть в те точки, где достигался минимум. Для F(9) минимум был достигнут в точке 1, т. е. взяли $x_1 = 1$. Осталось вернуть 8. Для F(8): $x_2 = 4$, осталось вернуть 4. Для F(4): $x_2 = 4$. Значит набор: 1, 4, 4. Бывает более одного оптимального решения.

Программа на языке C# приведена ниже.

```
int[] coins = { 5, 2, 1 }; // Монеты в порядке убывания
int k = 10; // Сумма сдачи

Console.WriteLine($"Нахождение сдачи на сумму {k} рублей с
минимальным количеством монет:");

foreach (int coin in coins)
   {
    int count = k / coin; // к-во монет текущего номинала
взять
     if (count > 0)
        {
         Console.WriteLine($"{count} монет(а) по {coin}
руб.");
         k -= count * coin;
         }
   }
```

Можно преобразовать программу для ввода с клавиатуры номинала монет и сдачи.

4.8. Задача о максимальном призе

Дана таблица *m*n* целых чисел. Человек начинает путешествие в нижней левой клетке таблицы, заканчивает – в верхней правой. Он может ходить вправо, вверх, вверх-вправо. Дополнительное условие: сумма клеток, по которым прошел человек должна быть максимальной.

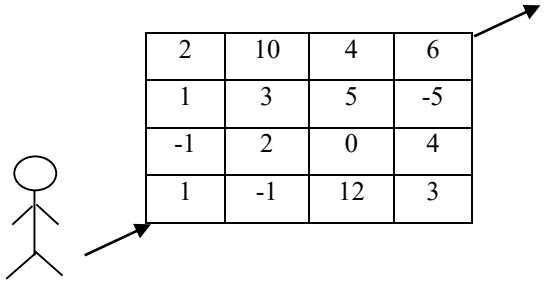

Рисунок 8 – Демонстрация задачи о максимальном призе

Для данного примера, например, вверх-вверх-вверх-вправо-вправо- вправо. То есть 1 + (–1) + 1 + 2 + 10 + 4 + 6 = 23

Цель: найти алгоритм, который получает на вход таблицу *m*n* с призами в каждой клетке и возвращает значение максимального приза, который можно получить, путешествуя по таблице по указанным выше условиям.

Очевидное решение – перебрать все пути, их 2^n . Значит алгоритм экспоненциальный.

Пусть *F(i, j)* – максимальный приз, который получит человек, стартующий в точке *[i, j]*. *P* – таблица входных данных.

Таблица 9 – Исходные данные для задачи о максимальном призе

P =

4				*[n; n]*
3				
2				
1				
			3	

$F(1, 1)$ – ответ на задачу, тогда

$F(n, n) = P[n, n]$

$F(i, n) = P[i, n]+F(i+1, n)$, при $1 \le i \le n$

$F(n, j)=P[n, j]+F(n, j+1)$

Эти формулы позволяют посчитать значения для n-й строки, n-го столбца.

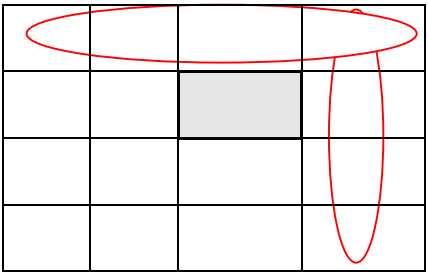

Рисунок 9 – Демонстрация первого действия

Теперь можно найти приз для элемента [*n-1, n-1*] (см. серая клетка на рисунке 9). Тогда:

$$F(n-1,n-1) = P[n-1;n-1]+$$
$$+\max\left[\frac{F(n,n-1)}{вверх};\frac{F(n,n)}{вправо-вверх}+\frac{F(n-1,n)}{вправо}\right]$$

Итак, формула расчета $F(n\text{-}1, n\text{-}1)$ найдена. Аналогично можно посчитать по очереди все элементы влево в n-1 строке, затем в n-1 столбце и т. д. пока не дойдем до [1,1]. В частности:

$F(n, n)= P[n, n]$

$F(i, n)= P[i, n]+F(i+1, n)$

$F(n, i)= P[n, i]+F(n, i+1)$

$F(i, j) = P[i, j]+max[F(i+1,j),F(i+1,j+1),F(i, j+1)]$ для $1 \le i, j < n$.

Кроме формул надо определиться со способом пересчета: считаем последнюю строку слева направо, потом последний столбец

сверху вниз. Затем предпоследнюю строку и предпоследний столбец и т. д. до [1, 1].

Время работы алгоритма $\theta(n^2)$, так как в процессе работы сначала заполняется матрица, затем пересчитываются все элементы. Сам маршрут, который обеспечивает этот приз может быть найден, начиная с $F(1, 1)$, сохраняя какой элемент обеспечивает максимум.

Пример: Дана таблица. Найти максимальный приз.

Таблица 10 – Исходные данные для задачи максимального приза

1	10	2	1
1	-2	5	-3
1	-3	6	8

По формуле считаем последнюю строку и столбец и сразу заполняем.

Таблица 11 – Расчет последней строки и столбца в задаче максимального приза

	1	2	3	4
3	1+13=14	10+3=13	2+1=3	1+(-3)=-2
2				1+(-3)=-2
1				8+(-2)=6

Выполняем дальнейшие расчеты:

Вверх вправо-вверх вправо

$F(2, 3) = 5 + max$ [**3**, 1, -2] = 5 + 3 = 8

$F(2, 2) = -2 + max$[**13**, 3, 8] = -2 + 13 = 11

$F(2, 1) = 1 + max$ [**14**, 13, 11] = 1 + 14 = 15

$F(1, 3) = 6 + max$ [**8**, -2, 6] = 6 + 8 = 14

$F(1, 2) = -3 + max$[11, 8, **14**] = -3 + 14 = 11

$F(1, 1) = 1 + ma\ x$[**15**, 11, 11] = 1 + 15 = 16

Получим таблицу значений.

Таблица 12 – Расчет остальных строк и столбцов в задаче максимального приза

	1	2	3	4
3	1+13=14	10+3=13	2+1=3	1+(-3)=-2
2	15	11	8	1+(-3)=-2
1	16	11	14	8+(-2)=6

Приз равен 16. Найдем сам путь. Начало в F(1, 1). Максимум достигнут, когда движение было вверх, дальше 14 вверх, 13 – вправо, 3 – вправо.

```
// данные могут быть введены с клавиатуры,
//в этом коде – заполнение случайными числами
        Console.Write("Введите количество строк (m): ");
        int m = int.Parse(Console.ReadLine());

        Console.Write("Введите количество столбцов (n): ");
        int n = int.Parse(Console.ReadLine());

        Console.Write("Введите минимальное значение в таб-
лице: ");
        int minVal = int.Parse(Console.ReadLine());

        Console.Write("Введите максимальное значение в таб-
лице: ");
        int maxVal = int.Parse(Console.ReadLine());

        int[,] grid = new int[m, n];
        Random rand = new Random();

        Console.WriteLine("\nСгенерированная таблица:");
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                grid[i, j] = rand.Next(minVal, maxVal + 1);
                Console.Write($"{grid[i, j],4}");
            }
            Console.WriteLine();
        }
```

```
int[,] dp = new int[m, n];
(int, int)[,] prev = new (int, int)[m, n];

for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        dp[i, j] = int.MinValue;

dp[m - 1, 0] = grid[m - 1, 0];
prev[m - 1, 0] = (-1, -1); // старт

// Заполняем DP и массив предыдущих шагов
for (int i = m - 1; i >= 0; i--)
{
    for (int j = 0; j < n; j++)
    {
        int current = dp[i, j];

        if (i - 1 >= 0) // Ход вверх
        {
            int sum = current + grid[i - 1, j];
            if (sum > dp[i - 1, j])
            {
                dp[i - 1, j] = sum;
                prev[i - 1, j] = (i, j);
            }
       }

       if (i - 1 >= 0 && j + 1 < n) // Ход вверх-вправо
        {
            int sum = current + grid[i - 1, j + 1];
            if (sum > dp[i - 1, j + 1])
            {
                dp[i - 1, j + 1] = sum;
                prev[i - 1, j + 1] = (i, j);
            }
        }

         if (j + 1 < n) // Ход вправо
        {
            int sum = current + grid[i, j + 1];
            if (sum > dp[i, j + 1])
            {
                dp[i, j + 1] = sum;
                prev[i, j + 1] = (i, j);
            }
        }
    }
}
```

```
Console.WriteLine($"\nМаксимальная сумма по пути: {dp[0, n -
1]}");
List<(int, int)> path = new List<(int, int)>(); // путь

int x = 0, y = n - 1;

while (x != -1 && y != -1)
     {
      path.Add((x, y));
       (x, y) = prev[x, y];
     }

path.Reverse();

Console.WriteLine("\nПуть (формат: [строка, столбец]):");
foreach (var (i, j) in path)
        {Console.WriteLine($"[{i}, {j}] → значение: {grid[i,
j]}");}
```

В программе реализован и вывод пути.

4.9. Задача о хрустальных шарах

Имеется здание высотой 100 этажей и два хрустальных шара. Существует неизвестный этаж *k*, начиная с которого шары разбиваются, если бросить их. Найти *k* с минимальным количеством бросков. Если шар разбился, его больше использовать нельзя.

Примеры решений:

1) в худшем случае 100 бросков. Начинаем с первого этажа и идем вверх;

2) первый шар бросаем с 50 этажа. Если шар разбился бросаем с 25 этажа, если нет – с 75 этажа, то есть, используя алгоритм двоичного поиска за $\lceil \log n \rceil$, находим количество бросков, т. е. $\lceil \log_2 100 \rceil = 7$;

3) формализуем задачу и напишем рекурсивную формулу, которая позволит решить задачу.

Пусть *F(n, k)* – минимальное количество бросков, которое нужно, чтобы определить этаж, начиная с которого разбиваются шары. Причем в здании *n* этажей и у нас есть *k* шаров. То есть

$F(n, 1) = n$
$F(n, \log n) = \log n$

$$F(n,k) = \underbrace{1}_{\text{одна попытка}} + \min_{1\le i\le n}[\max(\overbrace{F(i,k-1)}^{\text{шар разбился}}; \overbrace{F(n-i+1,k)}^{\text{шар не разбился}})]$$

решение с какого этажа бросить первый шар

$i = 1, 2, 3, \ldots, n.$

Фактически имеются две ситуации:

1) если шар брошен с i-го этажа, и он разбился, то осталось k-1 шаров;

2) если шар не разбился, то шаров осталось k; а неисследованных этажей осталось $(n - i + 1)$.

Берем из найденных чисел наибольшее, так как необходимо найти наихудший случай. Формула для расчета найдена, найдем порядок пересчета:

$F(1, 1), F(2, 1), \ldots, F(100, 1).$
$F(1, 2), F(2, 2), \ldots, F(100, 2).$

Путь имеется 2 шара и 100-этажное здание, а x – оптимальный этаж для одного броска.

Если шар разобьется, то останется 1 шар и $(x - 1)$ этажей. Если шар не разбился, то осталось 2 шара и 100 этажей. Мы ищем решение для худшего количества бросков, поэтому возьмем наибольшее количество бросков:

max ([1 шар, (x – 1) этаж], [2 шара, (100 – x) этажей])

Выражение *[2 шара, (100-x) этажей]* – фактически, это исходная задача, но для меньшего количества этажей. На данном этапе неизвестно значение величины *max*, но известно, что $100 - x \le x - 1$.

Если бы это было не так, т. е.$100 - x > x - 1$, то бросали бы не с этажа x, а с более верхнего этажа, тогда уменьшили бы 100-x и тем самым уменьшили бы максимум. Но ведь x выбрали так, чтобы максимум в (1) был минимален. Из этого следует, что $100 - x \le x - 1$. Дальше по рекурсии.

Первый раз бросили с этажа x. Осталось 2 шара и $(100 - x)$ этажей, но нельзя воспользоваться больше, чем $(x - 1)$ попыткой. Поэтому нельзя бросить шар с этажа, большего чем $(x - 1)$ этаж, т. е. $100 - x \le x - 1$.

Второй раз бросили с этажа $x + (x - 1)$; третий раз $x + (x - 1) + (x - 2)$ и т. д. Получили уравнение

$$x + (x-1) + (x-2) \ldots \ge 100 \quad x_1 = 14; x_2 = 14 + 13 = 27; x_3 = 27 + 12 = 39, \ldots$$

Вывод: достаточно 14 бросков, чтобы вычислить k.

Код программы на языке C# для решения данной задачи представлен ниже.

```
int floors = 100; // Количество этажей
int eggs = 2;     // Количество шаров

int[,] dp = new int[eggs + 1, floors + 1];

// Заполняем таблицу
//dp[e, f] — мин. число бросков при e шарах и f этажах
for (int e = 1; e <= eggs; e++)
    {
     for (int f = 1; f <= floors; f++)
        {
         if (e == 1)
           {
            // Если один шар — линейный поиск
            dp[e, f] = f;
            }
          else
            {
             dp[e, f] = int.MaxValue;

             for (int k = 1; k <= f; k++)
                {
                  int breaks = dp[e - 1, k - 1]; // Шар
разбился
                  int survives = dp[e, f - k];// Шар не раз-
бился
                  int worst = 1 + Math.Max(breaks, sur-
vives);
```

```
                    if (worst < dp[e, f])
                        dp[e, f] = worst;
                    }
                }

        Console.WriteLine($"Минимальное количество бросков в
худшем случае: {dp[eggs, floors]}");
    }
}
```

В коде можно реализовать ввод исходных данных с клавиатуры.

4.10. Задача о множестве независимых вершин

Задача состоит в вычислении количества независимых множеств вершин в дереве. Дан неориентированный граф *G(V, E)*.

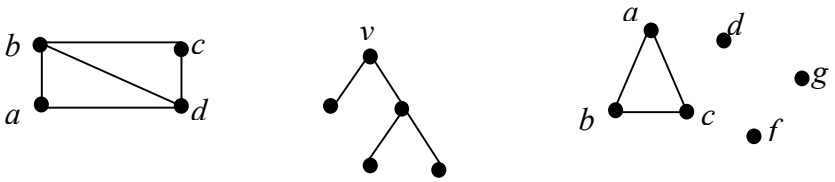

Рисунок 10 – Неориентированные графы

Независимое множество вершин графа (*independent set*) – это такое множество вершин графа, что любые две вершины этого множества не связаны ребром.

Например, на рисунке 10, *а*: {a, c}. На рисунке 10, *в* независимое множество состоит из одной вершины.

В дереве на рисунке 11 есть независимые множества вершин: I группа $\ni V_1$, II группа $\not\ni V_1$.

Утверждение I: если независимое множество $\ni V_1$, то оно состоит из независимых множеств, находящихся в *A, B, C.*

Замечание:

1) так как $V_1 \in$ независимому множеству, то в A нельзя брать V_2, в $B - V_3$, в $C - V_4$;

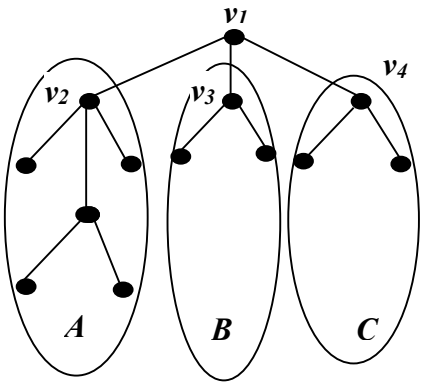

Рисунок 11 – Дерево

2) любое независимое множество из A можно объединить с независимым множеством из B и с независимым множеством из C, так как они не имеют непосредственных связей-ребер.

Утверждение II: независимое множество $\ni V_1$. Тогда берем любое независимое множество из A $\ni V_2$ или $\not\ni V_2$; из B $\ni V_3$ или $\not\ni V_3$; из C $\ni V_4$ или $\not\ni V_4$.

Обозначим F_v^+ – количество независимых множеств, корнями которых является v при условии что $v \in$ НМ. F_v^- – аналогично, но $v \notin$ независимому множеству.

Необходимо найти количество независимых множеств, то есть найти $F_{root}^+ + F_{root}^-$. Осталось написать рекурсивные формулы.

$$F_v^+ = \begin{cases} 1, \ v \in leaf\,(вершина - лист) \\ \prod\limits_{v_i \in child(v)} F^-(V_i), \ v \notin leaf \end{cases}$$

$$F_v^- = \begin{cases} 1, \ v \in leaf \\ \prod\limits_{v_i \in child(v)} (F_{v_i}^+ + F_{v_i}^-), \ v \notin leaf \end{cases}$$

Найдем порядок расчета. Очевидно, для того, чтобы найти F для корня, надо найти F для его потомков и т. д. вверх.

Пример: Дано дерево.

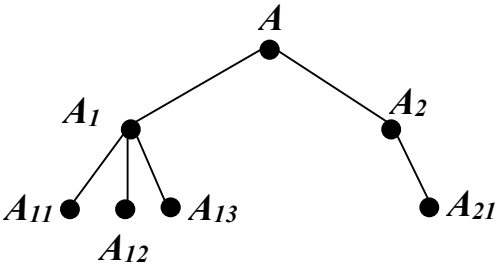

Рисунок 12 – Пример для демонстрации решения

Ответом будет $F_A^+ + F_A^-$.

Таблица 13 – Порядок расчета

	A_{11}	A_{12}	A_{13}	A_{21}	A_1	A_2	A
F^+	1	1	1	1	1	1	16
F^-	1	1	1	1	8	2	

Подробнее:

$$F_{A_1}^+ = F_{A_{11}}^+ \cdot F_{A_{12}}^+ \cdot F_{A_{13}}^+ = 1 \cdot 1 \cdot 1 = 1$$
$$F_{A_1}^- = (F_{A_{11}}^+ + F_{A_{11}}^-)(F_{A_{12}}^+ + F_{A_{12}}^-)(F_{A_{13}}^+ + F_{A_{13}}^-) = 2 \cdot 2 \cdot 2 = 2^3$$
$$F_{A_2}^+ = 1; \qquad F_{A_2}^- = 2$$
$$F_A^+ = F_{A_1}^- \cdot F_{A_2}^- = 8 \cdot 2 = 16$$
$$F_A^- = (1+8)(1+2) = 27$$

Ответ: 16+27=43
Код на языке C# представлен ниже.

```
    class TreeIndependentSet
{
static List<int>[] tree;
static int[,] dp;
```

```
        static bool[] visited;

        static void Main()
         {
                int n = int.Parse(Console.ReadLine()); // кол-
во вершин
                tree = new List<int>[n + 1];
                dp = new int[n + 1, 2];
                visited = new bool[n + 1];

                for (int i = 1; i <= n; i++)
                    tree[i] = new List<int>();

                for (int i = 0; i < n - 1; i++)
                {
                    var parts = Console.ReadLine().Split();
                    int u = int.Parse(parts[0]);
                    int v = int.Parse(parts[1]);

                    tree[u].Add(v);
                    tree[v].Add(u);
                }

                Dfs(1); // корень — вершина 1
                int result = Math.Max(dp[1, 0], dp[1, 1]);
                Console.WriteLine($"Максимальное  независимое
множество: {result}");
            }

            static void Dfs(int v)
            {
                visited[v] = true;
                dp[v, 0] = 0; // v не включён
                dp[v, 1] = 1; // v включён

                foreach (int u in tree[v])
                {
                    if (!visited[u])
                    {
                        Dfs(u);

                        dp[v, 0] += Math.Max(dp[u, 0], dp[u,
1]);
                        dp[v, 1] += dp[u, 0]; // нельзя вклю-
чать u, если v включён
                    }
                }
            }
        }
```

4.11. Задача о непрерывной подпоследовательности элементов массива максимальной суммы

Задана последовательность чисел a_i, $a_i \in R$ $1 \le i \le n$. Найти непрерывную подпоследовательность, сумма элементов которой максимальна.

Т.е. надо найти $\max\limits_{1 \le i,j \le n, i \le j} \sum\limits_{k=i}^{j} a_k$, точнее $a_i, a_{i+1}, a_{i+2}, ..., a_j$, длина которой равна $j+i+1$.

Если бы числа были положительные, то всё просто, то есть берем весь массив: $\max \sum\limits_{k=1}^{m} a_k$.

Рассмотрим для примера последовательность 1, 1, 1, 10, -12, 10.

Здесь общая сумма равна 13. Но первые 4 элемента в сумме тоже равны 13. Значит, нет смысла складывать все элементы.

Сформулируем задачу в общем виде. Пусть дана последовательность $a_1, a_2, ..., a_{n-1}, a_n$. Найдем максимальную сумму непрерывной подпоследовательности.

Согласно принципам ДП, начнем с конца. Найдем максимальную сумму подпоследовательностей, которые могут начинаться с a_n и до конца. Затем с a_{n-1} и до конца и т. д.. пока не получим решение.

Начиная с конца, возможны ситуации: a_n входит или a_n не входит в подпоследовательность.

Пусть F_i^+ – максимальная сумма подпоследовательности, содержащейся в последовательности $a_i, a_{i+1}, ..., a_n$, так что подпоследовательность содержит элемент a_i.

F_i^- – максимальная сумма подпоследовательности, содержащейся в последовательности $a_i, a_{i+1}, ..., a_n$, так что подпоследовательность не содержит a_i.

Тогда решением будет $\max[F_1^+, F_1^-]$.

$$F_i^+ = \begin{cases} a_n, & i = n \\ \max[a_i; a_i + F_{i+1}^+], & i < n \end{cases}$$

$$F_i^- = \begin{cases} 0, & i = n \\ \max[F_{i+1}^+; F_{i+1}^-], & i < n \end{cases}$$

Порядок расчета: $F_n^{\pm} \leftarrow F_{n-1}^{\pm} \leftarrow ... \leftarrow F_1^{\pm}$. Время работы $\theta(n)$

Таблица 14 – Исходные данные для задачи о максимальной сумме подпоследовательности

Номер элемента	1	2	3	4	5	6	7
Элемент	1	2	3	10	-14	8	2

Тогда для этого набора F_1^+, F_1^- будут такими (см. табл. 15).

Таблица 15 – Расчетные данные для задачи о максимальной сумме подпоследовательности

Номер элемента	7	6	5	4	3	2	1
F_i^+	2	10	-4	10	13	15	17
F_i^-	0	2	10	10	10	13	15

Подробный расчет представлен ниже:

$$F_6^+ = \max[a_6,\ a_6 + F_7^+] = \max[8,\ 8+2] = 10$$
$$F_6^- = \max[F_7^+,\ F_7^-] = \max[2,\ 0] = 2$$
$$F_5^+ = \max[a_5,\ a_5 + F_6^+] = \max[-14,\ -14+10] = -4$$
$$F_5^- = \max[F_6^+,\ F_6^-] = \max[10,\ 2] = 10$$
$$F_4^+ = \max[a_4,\ a_4 + F_5^+] = \max[10,\ 10-4] = 10$$
$$F_4^- = \max[F_5^+,\ F_5^-] = \max[10,\ -4] = 10$$
$$F_3^+ = \max[a_3,\ a_3 + F_4^+] = \max[3,\ 3+10] = 13$$
$$F_3^- = \max[F_4^+,\ F_4^-] = 10$$
$$F_2^+ = \max[a_2,\ a_2 + F_3^+] = \max[2,\ 2+13] = 15$$
$$F_2^- = \max[F_3^+,\ F_3^-] = \max[13,\ 10] = 13$$
$$F_1^+ = \max[a_1,\ a_1 + F_2^+] = \max[2,\ 2+15] = 17$$
$$F_1^- = 16$$

Ответ: 16.

Если надо восстановить саму подпоследовательность, то следует запомнить в каких точках достигался максимум.

Код программы на языке C#.

```
static void Main()
    {
        Console.WriteLine("Введите последовательность целых
чисел через пробел:");
        var input = Console.ReadLine().Split();
        int[] a = Array.ConvertAll(input, int.Parse);
        int n = a.Length;

        int[] dp = new int[n];
        dp[0] = a[0];

        int maxSum = dp[0];
        int endIndex = 0;

        for (int i = 1; i < n; i++)
        {
            dp[i] = Math.Max(a[i], dp[i - 1] + a[i]);

            if (dp[i] > maxSum)
            {
                maxSum = dp[i];
                endIndex = i;
            }
        }

        // Восстановим начало подотрезка
        int sum = 0;
        int startIndex = endIndex;
        for (int i = endIndex; i >= 0; i--)
        {
            sum += a[i];
            if (sum == maxSum)
            {
                startIndex = i;
                break;
            }
        }

        Console.WriteLine($"\nМаксимальная сумма: {maxSum}");
        Console.WriteLine("Подпоследовательность:");
        for (int i = startIndex; i <= endIndex; i++)
        {
            Console.Write(a[i] + " ");
        }
        Console.WriteLine();
    }
```

В коде не только реализовано отыскание суммы, но и самого отрезка.

4.12. Задача одномерного ДП. Создание оптимального плана производства предприятия

Рассмотрим одномерную задачу ДП: на предприятии производят стулья, столы и шкафы. Известны данные о количестве изделий каждого вида, прибыли и количестве рабочих, требуемых для этого.

Таблица 16 – Исходные данные для задачи об оптимальном плане производства

	Прибыль	Требуется рабочих
Стулья	1	1
Столы	3	2
Шкафы	5	3

Пусть всего на предприятии 10 рабочих, x – количество произведенных стульев, y – столов, z – шкафов. Тогда целевая функция прибыли рассчитывается по формуле:

$$1 \cdot x + 3 \cdot y + 5 \cdot z \to \max$$

Есть ограничение по количеству рабочих:

$$1 \cdot x + 2 \cdot y + 3 \cdot z \le 10$$

Можно разделить задачу на этапы:

1) можно производить только стулья;
2) стулья и столы;
3) общая задача.

Обозначим:

$F_3(k)$ – максимальную прибыль, которую можно получить, производя только стулья, имея в распоряжении k рабочих, где $k \in [0;\ 10]$.

$F_2(k)$ – максимальная прибыль, которую можно получить, производя только стулья и столы.

$F_1(k)$ – максимальную прибыль, которую можно получить, производя стулья, столы и шкафы.

Цель – найти $F_1(10)$. Сначала найдем просто F_3 во всех k. Затем найдем F_2 в этих же k. Затем $F_1(10)$ по рекурсивной формуле.

$F_3(0) = 0(0)$, в скобках указано значение, максимизирующее прибыль;

$F_3(1) = 1(1)$, т. е. если имеется 1 рабочий, то он сделает 1 стул и прибыль будет равна 1;

$F_3(2) = 2(2)$, 2 стула, прибыль = 2;

$F_3(k) = k(k)$, k стульев, прибыль = k.

Далее найдем F_2 с помощью .

$F_2(0) =$ **0 (0)**

$F_3(1) = 0$ столов $+ F_3(1) = 0 + 1 =$ **0 (1)**, ноль столов один стул.

$F_2(2) = \max[0 + F_3(2);\ 3 + F_3(0)] = \max[2,\ 3] =$ **3 (1)**

Во введенных обозначениях 0 столов соответствует y = 0, а 1 стол – y = 1.

Если произведено 0 столов, то прибыль = 0, а оставшиеся 2 рабочих будут производить стулья, прибыль = 2.

Если произведен 1 стол, то прибыль равна $3 + F_3(0)$.

$F_2(2) = \max[0 + F_3(3),\ 3 + F_3(1)] = \max[0 + 3,\ 3 + 1] =$ **4 (1)**

Если произведено 0 столов, то все 3 рабочих переходят на производство стульев , прибыль равна 3. Если произвели 1 стол, то прибыль равна 3, а 1 рабочий переходит на производство стульев. Значит прибыль 3 (за стол)+1 (за стул) = 4.

Аналогично:

$$F_2(4) = \max[0 + F_3(4),\ 3 + F_3(2),\ 6 + F_3(0)] =$$
$$= \max[0 + 4,\ 3 + 2,\ 6 + 0] = \max[4,\ 5,\ 6] =$$ **6 (2)**

$$F_2(k) = 3 \cdot \left\lfloor \frac{k}{2} \right\rfloor + 1 \cdot \left(остаток\left(\frac{k}{2}\right)\right)$$

$$F_2(k) = 3 \cdot целую\ часть\ \frac{k}{2} + 1 \cdot остаток\left(\frac{k}{2}\right)$$

прибыль за 1 стол * количество столов + прибыль с 1 стула * количество стульев

Считаем F_1.

Таблица 17 – Расчетные данные для задачи об оптимальном плане производства

0	1	2	3	4	5	6	7	8	9	10
0(0)	1(1)	2(2)	3(3)	4(4)	5(5)	6(6)	7(7)	8(8)	9(9)	10(10)
0(0)	1(0)	3(1)	4(1)	6(2)	7(2)	9(3)	10(3)	12(4)	13(4)	15(3)

$$F_1(10) = \max_{z=0,1,2,3}[0 + F_2(10),\ 5 + F_2(7),\ 10 + F_2(4),\ 15 + F_2(1)] =$$

$$= \max[15,\ 15,\ 16,\ 16] = \mathbf{16\ (2,\ 3)}$$

Если произведено 0 шкафов, то оставшиеся 10 рабочих будут производить столы и стулья: $0 + F_2(10)$, если произведен 1 шкаф, то 3 рабочих будут делать шкаф, а 7 – столы и стулья $5 + F_2(7)$; если произведено 2 шкафа: $10 + F_2(4)$; если произведено 3 шкафа: $15 + F_2(1)$.

То есть оптимальный план дает прибыль 16. Эта прибыль достигается в случае, когда шкафов 2, тогда остается 4 рабочих, они уйдут на производство 2 столов, значит на производство стульев никого не останется.

Итак: 2 шкафа + 2 стола; прибыль 16, шкафов = 3. Рассуждайте сами.

Одномерная задача ДП решена. Код ее решения на языке C# представлен ниже.

```
class ProductionPlanDP
{
    class Product
    {
        public string Name;
        public int Workers;
        public int Profit;
        public int MaxCount;

        public Product(string name, int workers, int profit,
int maxCount)
        {
            Name = name;
            Workers = workers;
            Profit = profit;
```

```
            MaxCount = maxCount;
        }
    }

    static void Main()
    {
        // Данные по изделиям
        var products = new Product[]
        {
            new Product("Стул", 3, 20, 10),
            new Product("Стол", 5, 45, 5),
            new Product("Шкаф", 7, 70, 3)
        };

        int maxWorkers = 30;
        int[] dp = new int[maxWorkers + 1];

        // Перебираем изделия
        foreach (var product in products)
        {
            // Перебираем количество изделий (ограничение)
            for (int count = 1; count <= product.MaxCount;
count++)
            {
                // Перебираем число рабочих от максимума
вниз
                for (int w = maxWorkers; w >= product.Work-
ers; w--)
                {
                    dp[w] = Math.Max(dp[w], dp[w - prod-
uct.Workers] + product.Profit);
                }
            }
        }

        Console.WriteLine($"Максимальная прибыль: {dp[max-
Workers]}");
    }
}
```

4.13. Задача двумерного ДП. Создание оптимального плана производства предприятия

Если необходимо учитывать два ограниченных ресурса, то задача становится двумерной. В данной задаче необходимо максимизировать прибыль, не превышая лимиты по рабочим, складу и количеству изделий.

Добавим пример ограничения по складским помещениям: производство требует площади складских помещений в количестве 2 (для стула), 3 (для стола) и 5 (для шкафа) единиц складского помещения. Всего имеется 9 единиц складского помещения.

Тогда система примет вид:

$$\begin{cases} 1x + 3y + 5z \to \max \ - \textit{целевая функция} \\ 1x + 2y + 3z \le 10 \ - \textit{ограничение по рабочим} \\ 2x + 3y + 5z \le 9 \ - \textit{новое ограничение} \end{cases}$$

Обозначим $F_3(k,m)$ – максимальную прибыль при производстве только стульев, имея *k* рабочих и *m* единиц складского помещения.

$F_2(k,m)$ – максимальную прибыль при производстве только столов и стульев, имея *k* рабочих и *m* единиц складского помещения.

$F_1(k,m)$ – максимальную прибыль при производстве стульев, столов и шкафов, имея *k* рабочих и *m* единиц складского помещения.

Ответом будет $F_1(10,9)$.

Вычислим

1) $F_3(k,m),\ 0 \le k \le 10,\ 0 \le m \le 9$.
2) $F_2(k,m)$ с помощью $F_3(k,m)$
3) $F_1(10,9)$.

Чтобы сэкономить на вычислениях, рассчитаем только те F_2, которые нужны для расчета F_1.

$F_3(0,0) = 0(0)$

$F_3(0,m) = 0(0)$

$F_3(k,0) = 0(0)$

$F_3(1,1) = 0(0)$

Необходимы 2 единицы склада, а имеется 1. Продолжим расчеты.

$F_3(k,1) = 0(0)$

$F_3(1,2) = 1(0)$

$F_3(2,2) = 1(1)$

$F_3(3,2) = 1(1)$

……

$F_3(10,2) = 1(1)$

$F_3(1,3) = 1(1)$

……

$F_3(10,3) = 1(1)$

$F_3(2,4) = 2(2)$

В общем виде: $F_3(k,m) = 1 * \min\left(\left\lfloor \frac{k}{1} \right\rfloor, \left\lfloor \frac{m}{2} \right\rfloor\right)$.

$F_3(k,m)$ = прибыль от 1 стула * минимум

$$\left(\frac{\text{количество рабочих всего}}{\text{количество рабочих, производящих 1 стул}}, \frac{\text{общая складская площадь}}{\text{складские единицы на 1 стул}} \right)$$

Вычислим F_2, но не все, сначала напишем формулу F_1 и посмотрим, какие F_2 нужны.

Для $k = 10$, $m = 9$ можно произвести $z = 0$ или $z = 1$ шкаф.

Для $z = 0$ прибыль со шкафов равна нулю, значит, рабочие уйдут на производство столов и шкафов, т. е. $F_2(10,9)$

Для z = 1 прибыль = 5, $k = 7$, $m = 4$, т. е. в $F_2(7,4)$.

Таким образом, находить все почти 100 значений F_2 не надо, достаточно $F_2(10,9)$ и $F_2(7,4)$.

$F_3(10,9) = \max[0 + F_2(10,9),\ 5 + F_2(7,4)]$

$F_2(7,4) = \max[0 + F_3(7,\ 4),\ 3 + F_3(5,\ 1)] = \max[0 + 2,\ 3 + 0] = 3(1)$

7 – количество рабочих, 4 – количество единиц склада, 0 столов, 1 стол

$$F_2(10,9) = \max_{y=0,1,2,3}[0 + F_3(10,9),\ 1 + F_3(8,6),\ 6 + F_3(6,3),\ 9 + F_3(4,0)] =$$

$$= \max[0 + 4,\ 3 + 3,\ 6 + 1,\ 9 + 0] = 9(3)$$

$F_1(10,9) = \max[0 + F_2(10,9),\ 5 + F_2(7,4)] = \max[0+9,\ 5+3] = 9(0)$,
т. е. оптимальное решение – это сделать 0 шкафов, а максимальная прибыль равна 9.

Если мы произведем 0 шкафов, значит, все ресурсы (10, 9) перейдут в F_2

А значение $F_2(10,9)$ оптимального решения (прибыль равна 9) достигается при производстве 3 столов. Поэтому производим 3 стола, а оставшиеся 4 рабочих и 0 склада уйдут на производство стульев.

Оптимальный план: 0 шкафов, 3 стола, 0 стульев.

Замечания:

1) порядок переменных (по возрастанию *x, y, z*) позволил упростить вычисления, сэкономив на вычислении неиспользуемых F_2;

2) в одномерной задаче всегда считали вектор, а в двумерной таблицу и т. д.

3) никакое дополнительное ограничение не может улучшить целевую функцию. Видно в приведенных задачах: прибыль была 16, стала 9.

Таким образом, сложность растет на порядок степени полинома с каждым добавлением ограничения. Это так называемое проклятие ДП, упомянутое Беллманом.

```
class ProductionPlanWithStorage
{
    class Product
    {
        public string Name;
        public int Workers;
        public int Storage;
        public int Profit;
        public int MaxCount;

        public Product(string name, int workers, int stor-
age, int profit, int maxCount)
        {
            Name = name;
            Workers = workers;
            Storage = storage;
            Profit = profit;
            MaxCount = maxCount;
        }
    }
```

```
    static void Main()
    {
        // Данные изделий
        var products = new Product[]
        {
            new Product("Стул", 3, 2, 20, 10),
            new Product("Стол", 5, 3, 45, 5),
            new Product("Шкаф", 7, 5, 70, 3)
        };

        int maxWorkers = 30;
        int maxStorage = 9;

        int[,] dp = new int[maxWorkers + 1, maxStorage + 1];

        foreach (var product in products) // Перебор изделий
        {
         // Для каждого изделия учитываем ограничение по ко-
личеству
            for (int count = 1; count <= product.MaxCount;
count++)
            {
              // Обновление DP: от больших значений к меньшим
             for (int w = maxWorkers; w >= product.Workers;
w--)
              {
              for (int s = maxStorage; s>= product.Storage;
s--)
                  {
                    int prevProfit = dp[w - product.Workers,
s - product.Storage];
                    int  newProfit  =  prevProfit  +  prod-
uct.Profit;
                    if (newProfit > dp[w, s])
                        {
                            dp[w, s] = newProfit;
                        }
                     }
                }
            }
        }

        Console.WriteLine($"Максимальная прибыль: {dp[max-
Workers, maxStorage]}");
    }
```

4.14. Игра с числами

Дана упорядоченная последовательность четного количества целых чисел: $a[0], a[1], ..., a[n-1]$. Два игрока по очереди берут одно число с края (левый или правый). Игра длится, пока все числа не выбраны. Цель каждого – максимизировать сумму своих чисел. Выясним какую сумму может гарантированно набрать первый игрок, если оба играют оптимально.

Пусть дана последовательность: 5, 10, 1, 2, 3, 6. Игроки по очереди берут числа:

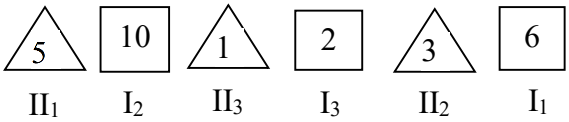

Рисунок 13 – Демонстрация процесса – игроки по очереди берут числа

В итоге: первый игрок набрал 6 + 10 + 2 = 18, а второй 5 + 3 + 1 = 9.

Цель – найти стратегию, максимизирующую выигрыш первого игрока. Возможно применить жадный алгоритм, но он не всегда здесь работает. Поэтому решаем с помощью ДП.

Итак, есть последовательность $a_1, a_2, \ldots, a_{2n}$. Выбор игрока – это первый или последний элементы текущей последовательности. Но в процессе игры элемент может оказаться где угодно. Поэтому анализ будем производить сначала для пар, то есть для подпоследовательностей длины 2.

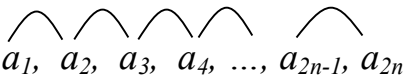

Рисунок 14 – Демонстрация анализа для пар

Если найдем решения для пар, то сможем найти и для подпоследовательностей длины 3.

Рисунок 15 – Демонстрация анализа для длины 3

Затем длины 4, 5 и т. д. до $2 \cdot n$ в конце.

Обозначим через $F(i, j)$ максимальный выигрыш первого игрока при условии, что сейчас его ход и остались числа, заданные индексами от i до j, причем $i < j$: $a_i, a_{i+1}, a_{i+2}, \ldots, a_j$. Ответом будет $F(1, 2 \cdot n)$.

Определим порядок пересчета. Пусть осталось два числа: 5, 1. Игрок возьмет максимум. То есть сначала считаем $F(1, 2)$, $F(2, 3)$, $F(3, 4)$, ..., $F(2n\text{-}1, 2n)$. Затем будем подниматься, пока не найдем $F(1, 2n)$.

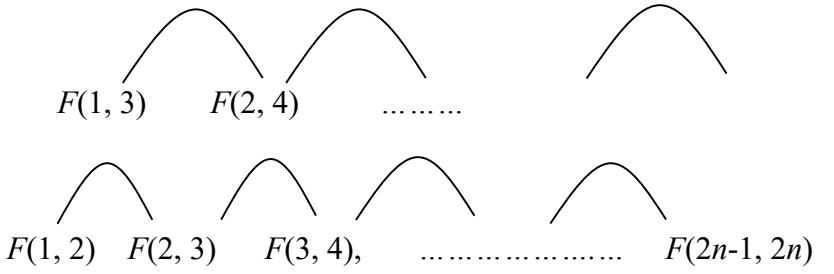

Рисунок 16 – Демонстрация анализа для длины n

Начальные условия таковы: $F(i, i+1) = max[a_i, a_{i+1}]$ *(индекс максимума)*

$$F(i, j) = max\{a_i + min\,[F(i+2, j), F(i+1, j-1)];$$
$$[a_j + min\,[F(i+1, j-1), F(i, j-2)]\}.$$

Если первый игрок выбрал a_i, то второй выберет a_{i+1} или a_j. Но неизвестно, что именно он выберет. Поскольку это игра с нулевой суммой, всё что выиграет один игрок – проиграет второй и наоборот.

Если второй игрок выберет a_{i+1}, то останется $F(i+2, j)$, если же он выберет a_j, то останется $F(i+1, j-1)$, то есть первый игрок останется с выигрышной суммой $F(i+2, j)$ или $F(i+1, j-1)$. Берем худший случай – минимальную сумму.

Аналогично рассуждаем для выбора a_j первым игроком.

Время работы: количество пар $n - 1$, количество троек $n - 2$, четверок $n - 3$, …

Итого $\Theta(n^2/2)$.

Код программы на языке C# представлен ниже.

```
class OptimalGameStrategy
{
    static void Main()
    {
        Console.WriteLine("Введите четное количество чисел
через пробел:");
        var input = Console.ReadLine().Split();
        int[] a = Array.ConvertAll(input, int.Parse);
        int n = a.Length;

        if (n % 2 != 0)
        {
            Console.WriteLine("Ошибка: количество чисел
должно быть четным.");
            return;
        }

        int[,] dp = new int[n, n];
        int[] prefixSum = new int[n + 1]; // для удобного
подсчета суммы на отрезке

        for (int i = 0; i < n; i++)// Префиксная сумма
            prefixSum[i + 1] = prefixSum[i] + a[i];

        // ДП по длине подпоследовательности
        for (int len = 1; len <= n; len++)
        {
            for (int i = 0; i + len - 1 < n; i++)
            {
                int j = i + len - 1;

                if (i == j)
                {
                    dp[i, j] = a[i];
                }
                else
                {
                    int sum = prefixSum[j + 1] - pre-
fixSum[i];
                    dp[i, j] = Math.Max(
                        a[i] + (sum - a[i] - dp[i + 1, j]),
                        a[j] + (sum - a[j] - dp[i, j - 1])
                    );
                }
            }
        }

        int firstPlayerScore = dp[0, n - 1];
        int totalSum = prefixSum[n];
```

```
        int secondPlayerScore = totalSum - firstPlay-
erScore;

        Console.WriteLine($"\nОптимальная сумма первого иг-
рока: {firstPlayerScore}");
        Console.WriteLine($"Оптимальная сумма второго иг-
рока: {secondPlayerScore}");
    }
```

4.15. Стохастическое динамическое программирование. Задача динамического программирования с выбором остановки

Задача динамического программирования с выбором остановки (англ. *optimal stopping problem*) заключается в том, что обычный шестигранный кубик (значения от 1 до 6, равновероятны) разрешено бросить не более трёх раз. После каждого броска можно остановиться и взять значение, которое выпало в этот раз или бросить снова (если броски остались). Найти оптимальную стратегию и максимизировать математическое ожидание выигрыша (то есть ожидаемую прибыль).

Интуитивно понятно. Если при броске вначале выпадает мало очков, то стоит еще кидать. Можно остановиться на имеющейся прибыли.

Необходимо точное решение. Получим его с помощью техники ДП. Пусть производится третий бросок. Тогда выигрыш равен c_3. Потом получим решение для второго и первого броска. Это будет сделать несложно, так как, зная F_3, найдем $F_{2,1}$. Затем найдем F_1, зная F_2 и F_3, но с элементами вероятности.

Пусть $F_i(z)$ – максимальное математическое ожидание выигрыша при условии, что находимся на броске *i* и выпало *z* очков.

Начнем с $F_3(z)$. Учтем, что вероятность попадания на все грани одинакова и равна 1/6. Если на третьем броске выпало *z* очков, то прибыль равна *z*.

Таблица 18 – Расчет F_3

z	1	2	3	4	5	6
$F_3(z)$	1	2	3	4	5	6

Математическое ожидание $F_3(z)$ считаем так: значение $F_3(z)$ * вероятность для всех.

$$EF_i = 1\cdot\frac{1}{6} + 2\cdot\frac{1}{6} + 3\cdot\frac{1}{6} + 4\cdot\frac{1}{6} + 5\cdot\frac{1}{6} + 6\cdot\frac{1}{6} = 3{,}5\ ;\ EF_i = 3{,}5.$$

$$F_2(z) = \max[\underset{\text{продолжить игру}}{3{,}5}\ ;\ \underset{\text{остановиться}}{z}]$$

Если остановились, то прибыль равна *z*, а если бросаем дальше, то переходим в III бросок, но не знаем какую получим прибыль, т.к. это случайная величина, зависящая от результата третьего броска, но знаем математическое ожидание прибыли третьего броска, она равна 3,5.

Т. е. $F_2(z) = \max[3{,}5; z]$

Таблица 19 – Расчет F_3 и F_2

	$F_3(z)$	$F_2(z)$
1	1	3,5 (продолжаем)
2	2	3,5 (продолжаем)
3	3	3,5 (продолжаем)
4	4	4 (остановка)
5	5	5 (остановка)
6	6	6 (остановка)

$$EF_2 = 3{,}5\cdot\frac{1}{6} + 3{,}5\cdot\frac{1}{6} + 3{,}5\cdot\frac{1}{6} + 4\cdot\frac{1}{6} + 5\cdot\frac{1}{6} + 6\cdot\frac{1}{6} = 4{,}24$$

Аналогично найдем $F_1(z)$. $F_1(z) = \max[\underset{\text{продолжим}}{EF_2}\ ;\ \underset{\text{остановка}}{z}]$

Таблица 20 – Расчет F_3, F_2 и F_1

	$F_3(z)$	$F_2(z)$	$F_1(z)$
1	1	3,5 (продолжаем)	4,24 (продолжаем)
2	2	3,5 (продолжаем)	4,24 (продолжаем)
3	3	3,5 (продолжаем)	4,24 (продолжаем)
4	4	4 (остановка)	4,24 (продолжаем)
5	5	5 (остановка)	5 (остановка)
6	6	6 (остановка)	6 (остановка)

$EF_1(z) = 4{,}66$

Заполним таблицу. Найдем ответ на вопрос: каково максимальное математическое ожидание выигрыша. Оно считается с помощью F_1, но неизвестно с чего начнётся игра. Она может начаться с 1, тогда получим в среднем 4,24; с 2 – 4,24, но игра может начаться и с 5, тогда получим 5. Поэтому, так как неизвестно с чего начнется игра, можно найти только математическое ожидание выигрыша и оно равно EF_1. Поэтому максимальное математическое ожидание выигрыша в игре является величина 4,66. Чтобы найти стратегию выигрыша, достаточно посмотреть в таблицу.

Например:

1) первый бросок – выпало 4; смотрим в таблицу – продолжаем; второй бросок – 3, продолжаем; третий бросок 2;

2) первый бросок – 5. Останавливаемся.

Программа на языке C#.

```
class OptimalDiceStrategy
{
    static void Main()
    {
        double[] values = { 1, 2, 3, 4, 5, 6 };

        // Шаг 1: последний бросок — берем что выпало
        double V1 = 3.5; // Среднее значение 6-гранного ку-
бика

        // Шаг 2: после II броска — остановиться, если x >=
V1
        int threshold2 = Threshold(values, V1);

        // Шаг 3: после I броска — сравниваем с ожиданием от
II шага
        double V2 = ExpectedMax(values, V1);
        int threshold3 = Threshold(values, V2);

        // Вывод стратегии
        Console.WriteLine("=== Оптимальная стратегия ===");
        Console.WriteLine($"После 1-го броска: остановиться,
если выпало ≥ {threshold3}");
        Console.WriteLine($"После 2-го броска: остановиться,
если выпало ≥ {threshold2}");
        Console.WriteLine("После 3-го броска: берем то, что
выпало.");
    }
    // Вычисление ожидания от max(x, nextExpectation)
```

```
    static double ExpectedMax(double[] values, double
nextExpected)
    {
        double sum = 0;
        foreach (var x in values)
        {
            sum += Math.Max(x, nextExpected);
        }
        return sum / values.Length;
    }

    // Порог: минимальное значение x, при кот.выгоднее оста-
новиться
    static int Threshold(double[] values, double nextEx-
pected)
    {
        for (int x = 1; x <= values.Length; x++)
        {
            if (x >= nextExpected)
                return x;
        }
        return values.Length + 1;
    }
```

4.16. Стохастическое динамическое программирование. Постоянный платеж

Сформулируем предыдущую задачу с дополнением, добавим стоимость броска – 1 у.е. Это меняет стратегию и расчёт оптимального выигрыша. Итак, есть симметричный кубик с гранями от 1 до 6. Можно бросить до 3 раз. После каждого броска можно остановиться и получить выпавшее значение минус сумма потраченных денег (1 у.е. за каждый выполненный бросок) или бросить снова (если есть ходы). Цель – максимизировать чистый выигрыш (выпавшее значение – расходы на броски).

Пусть $F_n(z)$ – максимальное математическое ожидание прибыли при условии, что сразу после броска *n* на кубике выпало *z*. Платежи делаются перед броском. Ответом будет $EF_1(z)-1$; (1 – плата за бросок).

Найдем F_3, F_2, F_1.

$$EF_3(z)=3{,}5;\ F_2(z)=\max[\underset{stop}{z};\underset{continue}{-1+EF_3}];\ F_2(z)=\max[z;2.5];$$

$$F_1(z)=\max[z;-1+3.8]=2.8$$

Таблица 21 – Расчет F_3, F_2 и F_1

z	$F_3(z)$	$F_2(z)$	$F_1(z)$
1	1	2,5 (*cont*)	2,8 (*cont*)
2	2	2,5 (*cont*)	2,8 (*cont*)
3	3	3 (*stop*)	3 (*stop*)
4	4	4 (*stop*)	4 (*stop*)
5	5	5 (*stop*)	5 (*stop*)
6	6	6 (*stop*)	6 (*stop*)

$EF_1 - 1 = 3,9 - 1 = 2,9$ – это максимальное математическое ожидание прибыли.

Стратегия: если на первом броске выпало 2, то *continue* (продолжить); на втором броске выпало 4, то *stop* (остановиться). Код программы на C# приведен ниже.

```
class OptimalDiceWithCost
{
    static void Main()
    {
        int sides = 6;        // 6-гранный кубик
        int maxRolls = 3;     // 3 броска
        double cost = 1.0;    // Стоимость одного броска

        double[] values = new double[sides];
        for (int i = 0; i < sides; i++)
            values[i] = i + 1; // значения: 1..6

        // Ожидание на каждом шаге
        double[] V = new double[maxRolls + 1];
        V[1] = Average(values) - 3 * cost; // последний бро-
сок, платим за все 3

        for (int k = 2; k <= maxRolls; k++)
        {
            double prevExpected = V[k - 1];
            int throwNumber = maxRolls - k + 1; // сколько
бросков уже было
            V[k] = ExpectedMaxProfit(values, prevExpected,
throwNumber * cost);
        }
```

```
        // при каком значении стоит остановиться на каждом
броске)
        Console.WriteLine("=== Оптимальная стратегия с уче-
том стоимости броска ===");
        for (int k = maxRolls; k >= 2; k--)
        {
            int throwNumber = maxRolls - k + 1;
            double costSpent = throwNumber * cost;
            int threshold = Threshold(values, V[k - 1],
costSpent);
            Console.WriteLine($"• После {throwNumber}-го
броска: остановиться, если выпало ≥ {threshold}");
        }
        Console.WriteLine($"• После {maxRolls}-го броска:
берем, что выпало, но платим за 3 броска.");

        Console.WriteLine($"\nМаксимальное ожидаемое значе-
ние выигрыша: {V[maxRolls]:0.###} у.е.");
    }

    // Среднее значение
    static double Average(double[] arr)
    {
        double sum = 0;
        foreach (var x in arr) sum += x;
        return sum / arr.Length;
    }

    // Ожидаемое значение прибыли с учетом следующего шага
    static double ExpectedMaxProfit(double[] values, double
nextExpected, double costSpent)
    {
        double sum = 0;
        foreach (var x in values)
        {
            double profit = x - costSpent;
            sum += Math.Max(profit, nextExpected);
        }
        return sum / values.Length;
    }

    // минимальное значение, при котором выгодно остано-
виться
```

```
    static int Threshold(double[] values, double nextEx-
pected, double costSpent)
    {
        for (int x = 1; x <= values.Length; x++)
        {
            if ((x - costSpent) >= nextExpected)
                return x;
        }
        return values.Length + 1;
    }
```

Приведенный код не только представляет собой решение задачи о прибыли, но содержит рекомендации пользователю, когда необходимо остановиться.

ЗАКЛЮЧЕНИЕ

Динамическое программирование является методом для решения задач с многошаговой структурой, основанным на разбиении на перекрывающиеся подзадачи и использовании принципа оптимальной подструктуры. ДП – это универсальный подход к решению сложных задач, его главное преимущество – возможность найти оптимальное решение, избегая при этом избыточных вычислений благодаря сохранению промежуточных результатов в таблице.

Данное методическое пособие призвано развить навыки применения динамического программирования для решения практических задач в различных областях, от информатики до экономики, включая планирование и управление, где требуется найти оптимальное решение в многоэтапном процессе.

Освоение метода, представленного в данном пособии, позволит студентам развить навыки, необходимые для построения эффективных алгоритмов и решения реальных задач, и станет инструментом для понимания и практического применения динамического программирования вообще, в том числе в рамках дисциплин «Алгоритмы обработки данных» и «Алгоритмы и структуры данных», что позволит студентам подготовиться к выполнению программной части лабораторных работ.

ЛИТЕРАТУРА

1. Беллман, Р. Динамическое программирование и уравнения в частных производных / Беллман Р., Энджел Э. – Москва: Мир, 1974. – 203 с. – Текст : непосредственный.

2. Венцель, Е. С. Элементы динамического программирования / Е. С. Венцель. – Москва: Наука, 1964. – 174 с. – Текст : непосредственный.

3. Динамическое программирование. Классические задачи. – URL: https://habr.com/ru/articles/113108/ – Текст : электронный.

4. Довгалюк, П.М. Динамическое программирование и все-все-все / П.М. Довгалюк. – Москва: Ленанд, 2024. – 200 с. – Текст : непосредственный.

5. Кабаева, И. И. Задачи динамического программирования / И. И. Кабаева. – Текст : электронный // European research. – 2016. – № 11 (22). – URL: https://cyberleninka.ru/article/n/zadachi-dinamicheskogo-programmirovaniya (дата обращения: 01.10.2025).

6. Калихман, И.Л. Динамическое программирование в примерах и задачах: учебное пособие / И.Л. Калихман, М.А. Войтенко. – Москва: Высш.школа, 1979. – 125 с. – Текст : непосредственный.

7. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – 3-е изд. – Москва: Вильямс, 2013. – 1328 с. – Текст : непосредственный.

8. Лежнёв, А. В. Динамическое программирование в экономических задачах: учебное пособие / А. В. Лежнёв. – Москва: БИНОМ, 2020. – 179 с. – Текст : непосредственный.

9. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл. 2-е изд., доп. – Москва: Техносфера, 2004. – 368 с. – Текст : непосредственный.

10. *C#* 6.0 и платформа *.NET* 4.6 для профессионалов / К. Нейгел, [и др.]. – Москва: Диалектика, 2016. – 960 с. – Текст : непосредственный.

11. Окулов, С. М. Динамическое программирование / С. М. Окулов, О. А. Пестов. – Москва : БИНОМ, 2012. – 296 с. – Текст : непосредственный.

12. Трауб, Дж. Общая теория оптимальных алгоритмов / Дж. Трауб, Х. Вожняковски. – Москва: Мир, 1983. – 384 с. – Текст : непосредственный.

13. Шень, А. Программирование: теоремы и задачи / А. Шень. – [7-е изд., доп.] – Москва: МЦНМО, 2021. – 320 с. – Текст : непосредственный.

Учебное издание

ДИНАМИЧЕСКОЕ ПРОГРАММИОВАНИЕ

Учебное пособие

Составитель:
Алла Владимировна Кирсанова

ИЛ № 06150. Сер. АЮ от 21.02.02.
Подписано в печать ..25. Формат 60 × 90/16.
Усл. печ. л.4,25. Электронное издание. Заказ № .

Изд-во Приднестр. ун-та. 3300, г. Тирасполь, ул. Мира, 18.