Центр дополнительного образования детей
Дистантное обучение

117630, Москва, ул. акад. Челомея, д. 8б, тел./факс 936-3104

Курс «Олимпиадные задачи по программированию».
Преподаватель: Михаил Сергеевич Густокашин

Лекция 8
Динамическое программирование (часть 1)

Оглавление:

msg@list.ru
http://g6prog.narod.ru
http://desc.ru

Введение

Динамическим программированием называется метод, который позволяет решать «переборные» задачи, опираясь на уже решенные подзадачи меньшего размера. При этом необходимо, чтобы все решения можно было запомнить в таблице (т.е. данные, посчитанные однажды, не пересчитываются).

Идеи динамического программирования очень похожи на идеи метода доказательства по индукции, известного из школьной программы математики. Сформулируем необходимые требования:

1) Все решения подзадач должны запоминаться в таблице.
2) Существует известные решения для задачи с малой размерностью (аналогично проверке для минимального параметра в методе математической индукции).
3) Существует способ выразить решение задачи через подзадачи (возможно, отличные от исходной задачи) **меньшей размерности**. Это больше всего напоминает рекуррентное соотношение в математике.

Требование меньшей размерности означает, что значение параметра (или одного из параметров, если их несколько) в подзадаче должно быть меньше, чем значение параметра в задаче и в итоге последовательность подзадач должна сходиться к известным решениям (нулевому параметру, например). Если параметры подзадач «уходят в бесконечность» или возникает круговая зависимость – это означает, что метод динамического программирования неприменим. В некоторых случаях разумно подсчитать сумму всех параметров и требовать, чтобы на каждом шаге эта сумма уменьшалась (однако это не всегда возможно).

Задачи динамического программирования делятся на два типа: оптимизация целевой функции или подсчет количества вариантов решения. В первом случае нам необходимо выбрать лучшее среди всех решений подзадач, во втором – просуммировать все количества решений подзадач.

В случае если ищется решение, иногда бывает необходимо подсчитать не только сам ответ, но и восстановить решение полностью (записать, какие действия выполнялись на каждом шаге).

Оптимизация целевой функции для заданного числа подзадач.

К задачам оптимизации целевой функции для заданного числа подзадач относятся задачи, где решение задачи можно найти исходя из знания решений нескольких предыдущих задач. Будем решать задачи исходя из приведенной выше схемы.

Задача «Покупка билетов» предлагалась на Московской городской олимпиаде в 2003 году:

За билетами на премьеру нового мюзикла выстроилась очередь из N человек, каждый из которых хочет купить 1 билет. На всю очередь работала только одна касса, поэтому продажа билетов шла очень медленно, приводя «постояльцев» очереди в отчаяние. Самые сообразительные быстро заметили, что, как правило, несколько билетов в одни руки кассир продаёт быстрее, чем когда эти же билеты продаются по одному. Поэтому они предложили нескольким подряд стоящим людям отдавать деньги первому из них, чтобы он купил билеты на всех.

Однако для борьбы со спекулянтами кассир продавала не более 3-х билетов в одни руки, поэтому договориться таким образом между собой могли лишь 2 или 3 подряд стоящих человека.

Известно, что на продажу i-му человеку из очереди одного билета кассир тратит A_i секунд, на продажу двух билетов — B_i секунд, трех билетов — C_i секунд. Напишите программу, которая подсчитает минимальное время, за которое могли быть обслужены все покупатели.

Обратите внимание, что билеты на группу объединившихся людей всегда покупает первый из них. Также никто в целях ускорения не покупает лишних билетов (то есть билетов, которые никому не нужны).

Во входном файле записано сначала число N — количество покупателей в очереди ($1 \le N \le 5000$). Далее идет N троек натуральных чисел A_i, B_i, C_i. Каждое из этих чисел не превышает 3600. Люди в очереди нумеруются начиная от кассы.

В выходной файл выведите одно число — минимальное время в секундах, за которое могли быть обслужены все покупатели.

Будем искать решение по приведенной выше схеме.

Создадим одномерный массив (X), длиной 5000, в i-ой ячейке которого будем запоминать время, необходимое для покупки билетов всеми людьми до i-го включительно.

Далее - начальные данные. Создадим в массиве барьер – трех фиктивных людей с номерами 0, -1 и -2; в соответствующие ячейки заполним нулями (время покупки билетов фиктивными людьми – 0 секунд). Заполним соответствующие ячейки массивов A, B и C «бесконечностью», которой в данном случае может быть любое число больше 3600 (фиктивные люди покупают билеты очень медленно, это необходимо, чтобы они никогда не участвовали в решении).

Теперь запишем оптимизирующую функцию. Пусть заполнены все ячейки до K-ой, не включая ее. Нам необходимо выразить X[K] через уже решенные подзадачи. Для каждого человека (если считать, что за ним очереди нет) существует всего 3 варианта приобрести билет: либо все до него покупают билет, и он берет только для себя; либо все, кроме предыдущего покупают себе билета, а предыдущий берет два билета; либо все, кроме двух предыдущих покупают себе билеты, а стоящий на позиции K-2 покупает 3 билета. Т.к. требуется посчитать минимальное время, то формула будет выглядеть следующим образом:

$$X[K] = \min(X[K-1]+A_k,\ X[K-2]+B_{k-1},\ X[K-3]+C_{k-2})$$

Таким образом, для решения нам необходимо инициализировать массив X фиктивными людьми, считать массивы A, B и C, и прогнать цикл по K от 1 до N. Ответ будет содержаться в ячейке X[N].

Сложность такого решения линейна.

Оптимизация целевой функции для всех подзадач.

Рассмотрим задачи, где для получения ответа требуются все уже решенные задачи меньшего размера.

Для примера возьмем классическую задачу динамического программирования: задачу о наибольшей возрастающей подпоследовательности.

Условие ее такого:

Дана последовательность из N чисел. Требуется найти длину наибольшей возрастающей подпоследовательность. Элементы в подпоследовательности не обязательно идут подряд (т.е. часть элементов исходной последовательности можно «выбросить») и каждый элемент подпоследовательности должен быть больше предыдущих.

Будем действовать по уже выработанной схеме.

Создадим одномерный массив (X) длины N, в K-ой ячейке которого будем хранить максимальную длину возрастающей подпоследовательности, последним элементом которой является элемент с номером K.

В данной задаче не будем создавать барьер, а подберем такую оптимизирующую функцию, которая сможет обойтись без него.

Сама оптимизирующая функция будет выглядеть следующим образом: среди всех предыдущих чисел, меньших текущего, выберем то, для которого длина наибольшей возрастающей подпоследовательности максимальна. Если таких чисел не нашлось, то в X[K] запишем 1 (подпоследовательность состоит из одного числа).

$$X[K] = \max_{i=0,k-1}(X[i] \mid X[K] > X[i]) + 1$$

Как уже было описано выше, в случае, если таких чисел не нашлось, функция max должна возвращать 0.

Чтобы получить ответ, необходимо просмотреть все элементы массива X и выбрать среди них максимальный.

Данное решение верно, т.к. мы перебираем все варианты (но не пересчитываем каждый раз решение подзадачи).

Сложность такого решения будет $O(N^2)$.

Восстановление решения задачи оптимизации.

В предыдущих пунктах мы только находили ответ, но не восстанавливали самого решения. Например, в задаче о покупке билетов могло бы требоваться указать, сколько билетов купил какой человек. А в задаче о поиске наибольшей возрастающей подпоследовательности – указать элементы этой подпоследовательности.

Обычно восстановление проводится задом наперед, т.е. от ответа к начальной позиции и, при необходимости, переворачивается. Для восстановления используются либо циклы, которые «перепрыгивают» некоторые элементы, двигаясь от конца к началу или рекурсивные функции.

Так, в задаче о наибольшей возрастающей подпоследовательности для того, чтобы восстановить ее необходимо кроме длины хранить еще и номер предыдущего элемента (а для первого элемента в подпоследовательности, когда длина равна 1, некий специальный признак, например, число -1). Затем, после того, как мы найдем элемент с ответом, получим решение, если будем двигаться, начиная с его номера и переходя на предыдущий до тех пор, пока не достигли -1. Ответ будет получен в обратном порядке, существует два варианта – записать во вспомогательный массив и вывести его с конца, либо использовать рекурсивную функцию перехода на предыдущий и выводить ответ на выходе из функции. Стоит отметить, что рекурсия работает медленнее и занимает больше памяти.

Рассмотрим, в качестве примера, еще одну классическую задачу динамического программирования: дискретную задачу о рюкзаке.

Дан набор, состоящий из N неделимых предметов (т.е. разделять каждый предмет на части нельзя), для каждого из которого известен его вес Mi и стоимость Si. Максимальная грузоподъемность рюкзака – L. Требуется унести в рюкзаке предметы, суммарная стоимость которых максимальна.

Для решения этой задачи создадим массив (X) длины L, K-я ячейка которого представляет собой структуру и хранит:

1) Максимальную суммарную стоимость вещей на данный момент (поле sum). Общий вес вещей K

2) Какая вещь была добавлена последней (поле num).

Инициализируем массив следующим образом: во X[0] (нулевой суммарный вес) запишем фиктивную вещь с номером -1 и суммарной стоимостью 0. Во все остальные ячейки запишем в качестве номера вещи -2 – признак того, что набор такого веса составить невозможно.

Сделаем динамику «по вещам», т.е. будем искать решение задачи для J первых вещей исходя из знания оптимального решения для J-1 первой вещи. Для 0 вещей наше утверждение выполняется. На каждом шаге будем пытаться добавить очередную вещь ко всем уже составленным наборам. Если набор суммарным весом, равным сумме веса очередной вещи и веса дополняющего ее до текущего веса набора, не существовал или его стоимость была меньше, чем новая, то заменяем его.

```
for (j=0; j<N; i++) //цикл по вещам
{
```

```
     for (i=L-M[j]; i>=0; i--) //цикл по весам
          if (X[i].num != -1)
//если набор веса i (дополнительный) существует
               if ((X[i+M[j]].num == -1) || (X[i+M[j]].sum < S[j] +
X[i].sum))
//если набор не существовал или был хуже нового, то заменяем
               {
                    X[i+M[j]].sum = S[j] + X[i].sum;
                    X[i+M[j]].num = j;
               }
    }
```

Особенность этого метода – проход с конца массива к началу. Это необходимо, чтобы одна и та же вещь не вошла в набор два раза. Проиллюстрируем на примере. Пусть у нас имеются вещи со следующими парами вес-стоимость (1, 3), (3, 2), (4, 6), (2, 2) и рюкзак грузоподъемностью 6.

Вес	0		1		2		3		4		5		6	
Шаг 1	-1	0	1	3	-2	0	-2	0	-2	0	-2	0	-2	0
Шаг 2	-1	0	1	3	-2	0	2	2	2	5	-2	0	-2	0
Шаг 3	-1	0	1	3	-2	0	2	2	3	6	3	9	-2	0
Шаг 4	-1	0	1	3	4	2	4	5	3	6	3	9	4	8

Первое число в столбце – номер вещи, второе – суммарный вес. Синим цветом помечена обновленная информация, красным – те ячейки, где проводилось сравнение, но замена не произошла (результат не был улучшен).

Для того, чтобы получить ответ, нам необходимо найти в массиве элемент с максимальным полем `sum`. Восстановление начнем с этого поля, двигаясь назад. А именно – запомним или выведем поле `num` той ячейки, где находимся сейчас. Это означает, что вещь с номером `num` была положена последней и чтобы восстановить предыдущую часть необходимо отнять ее вес и продолжать это до тех пор, пока не дойдем до фиктивной вещи с номером -1.

Еще один пример: задача «Разложение числа» с Московской городской олимпиады по информатике 2005-2006 учебного года.

Учительница математики попросила школьников составить арифметическое выражение, так чтобы его значение было равно данному числу *N*, и записать его в тетради. В выражении могут быть использованы натуральные числа, не превосходящие *K*, операции сложения и умножения, а также скобки. Петя очень не любит писать, и хочет придумать выражение, содержащее как можно меньше символов. Напишите программу, которая поможет ему в этом.

В первой строке входного файла записаны два натуральных числа: *N* ($1 \le N \le 10000$) — значение выражения и *K* ($1 \le K \le 10000$) — наибольшее число, которое разрешается использовать в выражении.

В единственной строке выходного файла выведите выражение с данным значением, записывающееся наименьшим возможным количеством символов.

Если решений несколько, выведите любое из них.

При подсчете длины выражения учитываются все символы: цифры, знаки операций, скобки.

Заметим, что для решения представления числа i существует три варианта: требуемое число может быть записано непосредственно (i ≤ K, требуемое количество символов – количество цифр в числе), число может быть представлено в виде суммы или произведения двух меньших чисел. Будем считать множитель и просто запись числа одним и тем же (в этом случае скобки не требуются). В случае если мы используем произведение двух чисел,

хотя бы одно из которых представляется суммой, то это число должно быть окружено скобками.

Перед тем, как восстанавливать выражение, научимся считать его длину. Для этого нам потребуется два вспомогательных массива, один (m) из которых будет содержать длину выражения этого числа как множителя (непосредственно число или произведение), а второй (s) – длину выражения этого числа через сумму чисел.

Вначале заполним массивы «бесконечностями» (в нашем случае это может быть число n+2, когда число представляется в виде суммы единиц). Затем заполним элементы массива m с индексами от 1 до k длиной числа (для этого есть несколько способов: подсчет длины строки, вычисление десятичного логарифма или просто запоминание).

После этого наступает черед непосредственно динамического программирования. Будем пытаться для каждого числа (i) из диапазона от k+1 до n найти наилучшее представление его в виде произведения и в виде суммы. Для представления в виде суммы нам необходимо перебрать все пары слагаемых j и p, таких, что $j \le p$ и $j+p=i$, и выбрать среди них наилучшую. При этом для каждого из чисел j и p необходимо выбирать минимум из представления в виде суммы и виде произведения. Сами эти числа необходимо запомнить, т.е., например, записать в специальные массивы s1[n] = j и s2[n] = p. Эта информация сильно облегчит восстановление решения.

Кроме того, для числа i нужно найти наилучшее представление в виде произведения. Для этого будем перебирать все числа j, начиная с 2 и пока $j^2 \le i$. Для каждого j будем находить дополнительный множитель p ($p = i/j$), такой, что $j \times p = i$ (т.е. число i делится на j нацело). Естественно, что для чисел j и p также нужно выбирать наилучшее из представлений. Однако в случае произведения возникает небольшое отличие: если сомножитель представляется в виде суммы, необходимо окружить его скобками. Таким образом, к представлению сомножителя (каждого из чисел p и j) в виде суммы, необходимо прибавлять 2 (т.е. сравнивать между собой s[j] + 2 и m[j]). Лучшее разбиение на множители также нужно запоминать (m1[n]=j и m2[n]=p).

После построения таких массивов мы уже знаем длину ответа (min(s[n], m[n])) и нам необходимо восстановить его. Для восстановления воспользуемся рекурсивной функцией, принимающей на вход два параметра: число, которое нужно восстановить и признак того, является ли восстанавливаемое число сомножителем (это необходимо для правильной расстановки скобок). При первом запуске число не будет являться сомножителем (ответ не надо окружать скобками). Сама функция должна рассматривать три основных случая: если восстанавливаемое число не превышает k, то просто выводим это число; если наше число наилучшим образом представимо произведением, то вызываем восстановление для первого сомножителя, печатаем знак «*» и вызываем функцию печати второго сомножителя (в этом случае в функции должны передаваться признаки того, что восстанавливаемое число является сомножителем); в случае же если число наилучшим образом представимо в виде суммы (здесь также два случая: нужны ли скобки или нет; если нужны, то прибавляем к длине 2 и ставим эти скобки вокруг выражения), то вызываем восстановление для первого операнда, ставим между ними знак «+» и восстанавливаем второй операнд (в этом случае признак сомножителя передавать не надо – скобки не нужны).

Существует и другой вариант решения, при котором представление сразу накапливается в виде строки (string), а в конце работы просто выводится.

При решении этой задачи необходимо писать достаточно аккуратную реализацию, т.к. ее асимптотическая сложность $O(N^2)$, что в нашем случае равно 10^8. Поэтому некоторая сложность восстановления оправдана уверенностью в производительности такого метода (по сравнению с операциями над динамически выделяемыми строками).

Подсчет числа ответов.

Задачи подсчета числа ответов практически не отличаются от задач поиска ответа. Однако вместо функции, выбирающей оптимальное решение, здесь используется сложение

или умножение, в зависимости от реализации. Следует отметить, что в большинстве такого рода задач для хранения количества вариантов требуются длинная арифметика, поэтому перед реализацией стоит оценить возможную длину максимального ответа, для того, чтобы получить требуемую длину числа. Операция сложения двух чисел, максимальное из которых равно N, требует O(logN) времени (т.е. пропорционально количеству знаков), а длинное умножение $O(\log^2 N)$.

Рассмотрим классическую математическую задачу о числах Фибоначчи, известную еще в Средневековье. В оригинале это задача о кроликах, которые, как и положено кроликам, плодятся и через некоторое время умирает. Задача состоит в том, чтобы определить количество кроликов через некоторый промежуток времени.

Если переформулировать ее в математических терминах, то получим рекуррентное соотношение $F_i = F_{i-1} + F_{i-2}$ и $F_0 = 1$, $F_1 = 1$. Задача состоит в том, чтобы найти F_N по заданному N.

Для вычисления N-го числа Фибоначчи существует формула, однако она непригодна для компьютерного использования.

Оценим скорость роста чисел Фибоначчи. Заметим, что числа возрастают и $2 \times F_i \geq F_{i+1}$. Таким образом, можно сделать оценку $F_N \leq 2^N$. Отсюда получим, что операция сложения двух длинных чисел Фибоначчи будет занимать O(logN).

Если пользоваться формулой, приведенной в определении (т.е. просто складывать последовательно пары подряд идущих чисел для получения следующего), то общая сложность с учетом расходов на длинную арифметику получится O(NlogN).

Стоит заметить, что при этом вовсе не обязательно хранить N длинных чисел, достаточно только 3: для двух предыдущих чисел и результата сложения.

Т.к. задачи, требующие подсчета чисел Фибоначчи, встречаются достаточно часто, постараемся найти более быстрый способ их подсчета.

Рассмотрим матрицу $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Верхний левый элемент в ней соответствует первому числу Фибоначчи. Возведем эту матрицу в квадрат: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$, теперь в верхнем левом углу матрицы стоит уже второе число Фибоначчи. Аналогично для 3-го числа: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$. Вообще говоря, выполняется следующее соотношение:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_{i-1} & F_{i-2} \\ F_{i-2} & F_{i-3} \end{pmatrix} = \begin{pmatrix} F_{i-1} + F_{i-2} & F_{i-1} \\ F_{i-1} & F_{i-2} \end{pmatrix} = \begin{pmatrix} F_i & F_{i-1} \\ F_{i-1} & F_{i-2} \end{pmatrix}$$

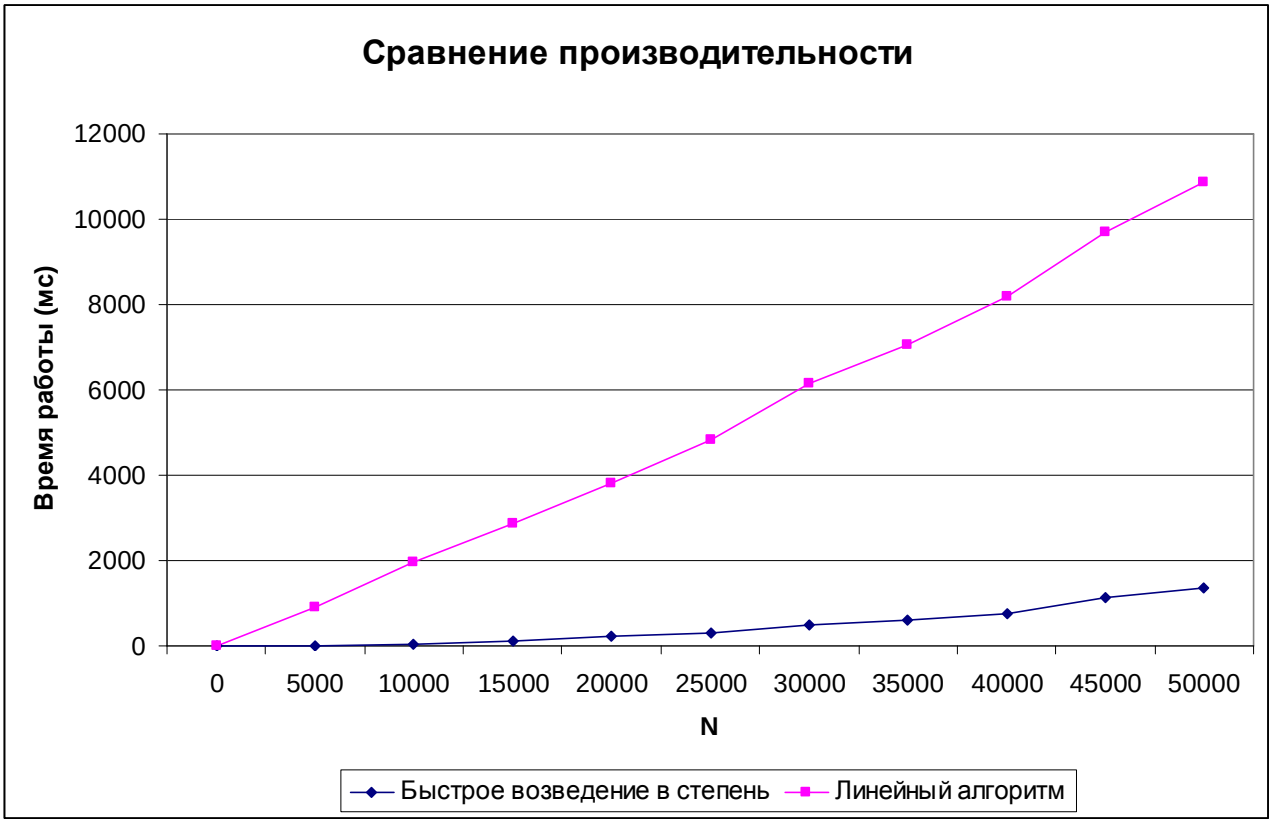
Т.к. за начальную матрицу мы принимаем также матрицу, состоящую из первых чисел Фибоначчи, то можно получить следующее утверждение:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^N = \begin{pmatrix} F_N & F_{N-1} \\ F_{N-1} & F_{N-2} \end{pmatrix}$$

Если просто возводить матрицу в степень, то мы получим сложность $O(N\log^2 N)$. Член $\log^2 N$ возник здесь из-за необходимости длинного умножения.

Однако мы знаем способ быстрого возведения в степень для чисел, который также применим и для матриц. Т.о. мы можем добиться сложности $O(\log^3 N)$.

На практике скорость работы выглядит так (усредненные значения по 5 экспериментам):



Некоторое «отставание» быстрого возведения в степень от оценки объясняется тем, что операция умножения (не используемая в линейном алгоритме) работает медленнее операции сложения.

Ленивая динамика.

В предыдущих пунктах мы вычисляли данные последовательно. Однако такой способ не всегда бывает оптимальным: при этом могут вычисляться лишние решения подзадач, которые не нужны для подсчета ответа. Также проблема может состоять в том, что порядок вычисления не линейный, но эта проблема возникает, в основном, в задачах с несколькими параметрами. Другое название этого метода «рекурсия с меморизацией».

Суть метода заключается в том, что мы пишем рекурсивную процедуру, которая перебирает только необходимые решения подзадач. Отличие от обычного перебора состоит в том, что однажды посчитанное значение не пересчитывается. Общая схема метода ленивой динамики выглядит так:

```
<type> f(<type> n)
{
     if (X[n] == -1) {
//некоторые вычисления, результат которых заносится в переменную
res
          X[n] = res;
     }
     return X[n];
}
```

Т.е. если решение было однажды посчитано, то оно не пересчитывается. Признак того, что решение еще не было посчитано – число -1. Доступ к массиву должен осуществляться только с помощью функции `f`! Прямое обращение к `X` может привести к тому, что мы получим неправильные данные. Этот факт несколько замедляет работу и, хотя ленивая динамика применима ко всем задачам, но в тех случаях, когда критична скорость, лучше использовать обычный обход.

Для примера рассмотрим задачу «Космический мусорщик» со Всероссийской командной олимпиады школьников 2002 года.

В околоземном космическом пространстве накопилось много мусора, поэтому ученые сконструировали специальный аппарат - ловушку для космического мусора. Для того чтобы хорошо собирать мусор, этот аппарат должен двигаться по достаточно сложной траектории, сжигая собранный по пути мусор. Ловушка может передвигаться в пространстве по 6 направлениям: на север (N), на юг (S), на запад (W), на восток (E), вверх (U) и вниз (D). Движением ловушки управляет процессор. Программа движения задается шестью правилами движения, которые соответствуют каждому из указанных направлений. Каждое такое правило представляет собой строку символов из множества {N, S, W, E, U, D}. Команда ловушки есть пара из символа направления и параметра - целого положительного числа M. При исполнении такой команды ловушка в соответствии со своей программой выполняет следующее. Если параметр больше 1, то она перемещается на один метр в направлении, которое указано в команде, а затем последовательно выполняет команды, заданные правилом для данного направления, с параметром меньше на 1. Если же параметр равен 1, то просто перемещается на один метр в указанном направлении. Пусть, например, заданы следующие правила:

Направление	**Правило**
N	N
S	NUSDDUSE
W	UEWWD
E	-
U	U
D	WED

Тогда при выполнении команды S(3) мусорщик выполнит следующие действия:
1) переместится на 1 метр в направлении S
2) выполнит последовательно команды N(2), U(2), S(2), D(2), D(2), U(2), S(2), E(2).
Если далее проанализировать действия мусорщика при выполнении команд из пункта 2, получим, что в целом он совершит следующие перемещения:

SNNUUSNUSDDUSEDWEDDWEDUUSNUSDDUSEE

По заданной команде определите, какое общее количество перемещений на один метр совершит ловушка при выполнении заданной команды. В приведенном примере это количество равно 34.

Первые шесть строк входного файла задают правила для команд с направлением N, S, W, E, U и D соответственно. Каждая строка содержит не более 100 символов (и может быть пустой). Следующая строка содержит команду ловушки: сначала символ из множества {N, S, W, E, U, D}, затем пробел и параметр команды - целое положительное число, не превышающее 100.

Выведите в выходной файл единственное число - количество перемещений, которое совершит ловушка. Гарантируется, что ответ не превышает 10^9.

Можно решать эту задачу с помощью обычного динамического программирования, создав таблицу из 6 строк, соответствующих направлениям, и заполнять их последовательно. Однако при этом возникнет некоторое (возможно, большое) количество «лишних» ячеек, значение которых нам не нужно.

При использовании ленивой динамики нам также понадобится таблица размером 6×100, вначале заполненная признаками того, что данная ячейка еще не обрабатывалась (-1). Первый столбец (для команд с параметром 1) заполним 1. Рекурсивную функцию оформим согласно общей схеме. При этом вычисления будут состоять в следующем: для известной команды и параметра пройдем по всей строке, соответствующей этой команде и просуммируем значения, возвращенные функциями, вызванными для очередной команды из строки с параметром на 1 меньшим. После этого прибавим к результату 1 (сама команда) и запишем его в соответствующую ячейку.

Заметим, что рекурсивное решение этой задачи без запоминания уже однажды вычисленных значений не уложиться в ограничение по времени (очень большое количество вызовов функций с одинаковыми параметрами).

Методы оптимизации задачи подсчета числа решений.

Некоторые задачи на подсчет количества решений методом динамического программирования могут иметь несколько решений (как мы уже видели на примере чисел Фибоначчи). Для примера рассмотрим следующую задачу:

Фишка может двигаться по полю длины N только вперед. Длина хода фишки не более K. По известным значениям N и K найти число различных путей, по которым фишка может пройти поле от начала до конца.

Эта задача очень похожа на числа Фибоначчи, ее отличие состоит в том, что нам надо суммировать не 2, а K предыдущих чисел. Это объясняется тем, что на ступеньку с номером i мы можем попасть только со ступенек номерами i-1, i-2, …, i-K.

Таким образом, как и в задаче о числах Фибоначчи мы можем просто считать очередное число по известным предыдущим и использовать для этого O(N×K) операций длинного сложения (для вычисления каждого элемента нам нужно выполнить K операций сложения). Числа будут расти очень быстро, так что без длинного сложения не обойтись.

Второй вариант также похож на способ вычисления чисел Фибоначчи. Отличие состоит в том, что мы будем использовать квадратную матрицу размером K, левый верхний угол которой заполнен 1. Выглядеть она будет так:

$$\begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & \dots & 1 & 0 \\ & & \dots & & \\ 1 & 1 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{pmatrix}$$

Показать правильность этого метода можно тем же способом, что и для чисел Фибоначчи.

Возводя эту матрицу в степень N, мы получим в верхнем левом ее элементе ответ. Опять же воспользуемся длинным умножением и получим сложность $O(\log N \times K^3)$. При этом будут выполняться как операции длинного сложения, так и более медленные операции длинного умножения.

Также у этой задачи существует решение, которое мы не рассматривали для чисел Фибоначчи. Будем идти с конца, т.е. пытаться выразить F(N, K). Пусть нам уже известны все значения функции F(X, K) для X от N/2-K до N/2. Для вычисления решения нужно аккуратно «склеить» из двух путей, длиной порядка половины N и одного связующего их отрезка длиной от 1 до K. При этом важно не упустить ни одного случая и не посчитать ничего дважды. Это вычисление потребует K умножений. Общая сложность решения получается $O(K^2) \times \log(N/K)$.

В некоторых задачах начальная матрица будет иметь другой вид, но в целом данные методы применимы к значительному числу задач подсчета числа решений. При этом в зависимости от соотношения N и K в конкретной задаче следует выбирать подходящий метод.