Динамическое программирование

Динамическое программирование

Словосочетание динамическое программирование впервые было использовано в 1940-х годах Р. Беллманом для описания процесса нахождения решения задачи, где ответ на одну задачу может быть получен только после решения задачи, «предшествующей» ей. Первоначально эта область была основана, как системный анализ и инжиниринг.

Слово «программирование» в словосочетании «динамическое программирование» в действительности к традиционному программированию почти никакого отношения не имеет и происходит от словосочетания «математическое программирование», которое является синонимом слова «оптимизация». Поэтому слово «программа» в данном контексте скорее означает оптимальную последовательность действий для получения решения задачи. К примеру, определенное расписание событий на выставке иногда называют программой. Программа в данном случае понимается как допустимая последовательность событий.

В общем случае мы можем решить задачу, в которой присутствует оптимальная подструктура, проделывая следующие три шага:

- Разбиение задачи на подзадачи меньшего размера.
- Нахождение оптимального решения подзадач рекурсивно, проделывая такой же трехшаговый алгоритм.
- Использование полученного решения подзадач для конструирования решения исходной задачи.

Подзадачи решаются делением их на подзадачи ещё меньшего размера и т. д., пока не приходят к тривиальному случаю задачи, решаемой за константное время (ответ можно сказать сразу). К примеру, если нам нужно найти

Недостаток рекурсии:

Простой рекурсивный подход будет расходовать время на вычисление решение для задач, которые он уже решал.

Что делать?

Чтобы избежать такого хода событий мы будем сохранять решения подзадач, которые мы уже решали, и когда нам снова потребуется решение подзадачи, мы вместо того, чтобы вычислять его заново, просто достанем его из памяти. Этот подход называется кэшированием.

Пример 1. Числа Фибоначчи

```
//рекурсивный вариант
int fib (int n) {
   if (n <= 1)
      return 1;
   return fib(n - 2) + fib(n - 1);
}

//ДП
int i;
int fib_num[1000] = {1, 1};
for (i = 2; i < 1000; i++)
   fib_num [i] = fib_num [i - 2] + fib_num [i - 1];
```

Динамическим программированием называется способ программирования, при котором

- задача разбивается на *подзадачи*,
- вычисление идет от малых подзадач к большим,
- ответы подзадач запоминаются в таблице и используются при решении больших задач.

Необходимо определить исходные данные задачи – **параметры**.

Например, при нахождении суммы некоторого набора чисел параметрами задачи будут количество чисел и их значения.

Тогда задача может быть формализована в виде некоторой функции, аргументами которой могут являться количество параметров и их значения.

Под **подзадачей** понимается та же постановка задачи, но с меньшим числом параметров или с тем же числом параметров, но при этом хотя бы один из параметров имеет меньшее значение.

Преимущество метода состоит в том, что если подзадача решена, то ее ответ где-то хранится и никогда **не вычисляется** заново.

Сведение решения задачи к решению некоторых подзадач может быть записано в виде **соотношений**, в которых значение функции, соответствующей исходной задаче, выражается через значения функций, соответствующих подзадачам.

Значения аргументов у любой из функций в правой части соотношения меньше значения аргументов функции в левой части соотношения. Если аргументов несколько, то достаточно уменьшения одного из них.

Динамическое программирование может быть применено к задачам оптимизации, когда требуется найти оптимальное решение, при котором значение какого-то параметра будет мин. или макс. в зависимости от постановки задачи.

Обычно требуется описать оптимальное решение, выписать рекуррентные соотношения, связывающие оптимальные значения параметра для подзадач, двигаясь снизу вверх, вычислить оптимальные решения для подзадач и используя их построить оптимальное решение для поставленной задачи.

Отметим, что для динамического программирования характерно, что зачастую решается не заданная задача, а более общая, при этом решение исходной задачи является частным случаем решения более общей задачи.

Пример 2. Найти количество последовательностей длины N из нулей и единиц, не содержащих двух единиц подряд.

Пусть sec[k] – количество последовательностей длины k из нулей и единиц, не содержащих двух единиц подряд.

k = 0 seq [k] = 0

k = 1 seq[k] = 2 1, 0

k = 2 seq[k] = 3 00, 01, 10

Пусть мы знаем решение для всех i < k, тогда посчитаем seq[k].

Чтобы получить последовательность длины k из последовательности длины k – 1, нужно в конец дописать либо 0 либо 1.

Дописываем 0: seq[k] = seq[k – 1]

Дописываем 1: seq[k] = seq[k – 2], т.к. в конце должно быть только …01

seq[k] = seq[k – 1] + seq[k – 2]

Пример 3. Сумма квадратов

На какое минимальное количество квадратов можно разложить число n?

Пусть sq[k] – минимальное количество квадратов, на которые можно разложить число k.

k = 0 sq [k] = 0

k = 1 sq [k] = 1 1×1

k = 2 sq [k] = 2 1×1 + 1×1

Пусть нам известно решение для всех i < k, тогда посчитаем sq[k].

Предположим, что нам нужно узнать, сколько квадратов будет в разложении k, если в этом разложении обязательно есть квадрат j×j.

sq[k] = 1 + sq[k – j · j], если j · j ≤ k

Пример 3. Сумма квадратов, продолжение

```
dp[0] = 0;
for (int i = 1; i <= n; i++) {
   dp[i] = INT_MAX;
   for (int j = 1; j * j <= i; j++) {
    dp[i] = min(dp[i], dp[i - j*j] + 1);
   }
}
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
dp==	0	1	2	3	1	2	3	4	2	1	2	3	3	4	3	4	1	2	2 …

Пример 4. Алгоритм Ахо

Пусть даны две строки S_1 и S_2. Необходимо за минимальное число *допустимых* операций преобразовать строку S_1 в строку S_2. Допустимой операцией являются следующие операции удаления символа из строки и вставки символа в строку:

DEL(*S, i*) – удалить *i*-ый элемент строки *S*;

INS(*S, i, c*) – вставить символ *c* после *i*-го элемента строки *S*.

Обозначим через *S* [*i*..*j*] – подстроку от *i*-го символа до *j*-го.

Пусть *M*(*i*, *j*) – минимальное количество операций, которые требуются, чтобы преобразовать начальные *i* символов строки S_1 в начальные *j* символов строки S_2: $S_1[0..i] \rightarrow S_2[0..j]$.

$S_1[0..0]$ и $S_2[0..0]$ – пустые строки.

Заметим, что для преобразования пустой строки в строку длины j

требуется j операций вставки, т.е. $M(0, j) = j$.

Аналогично для преобразования строки длины i в пустую строку

требуется i операций удаления, т.е. $M(i, 0) = i$.

Пусть мы решили подзадачу с параметрами $i-1$ и $j-1$. Это означает, что из строки $S_1[0..i-1]$ построена строка $S_2[0..j-1]$ за минимальное число допустимых операций $M(i-1, j-1)$.

I) Пусть $S_1[i] = S_2[j]$. Тогда для получения строки $S_2[0..j]$ из строки $S_1[0..i]$ не требуется никаких дополнительных операций.

Следовательно, $M(i, j) = M(i-1, j-1)$.

II) Пусть $S_1[i] \neq S_2[j]$. Возможны два способа получения строки $S_2[0..j]$.

1. Пусть из строки $S_1[0..i–1]$ построена строка $S_2[0..j]$ за минимальное количество операций M ($i–1, j$). Тогда для получения строки $S_2[0..j]$ из строки $S_1[0..i]$ требуется удалить i-ый символ из строки S_1.

2. Пусть из строки $S_1[0..i]$ построена строка $S_2[0..j–1]$ за минимальное количество операций M ($i, j–1$). Тогда для получения строки $S_2[0..j]$ из строки $S_1[0..i]$ потребуется одна операция вставки i-го символа строки S_1 после символа $S_2[j–1]$.

Из 2-х возможностей нужно выбрать лучшую.

Получим следующие рекуррентные соотношения:

$M(0, j) = j;\ \ M(i, 0) = i;$

$M(i, j) = \min(M(i-1, j-1), M(i-1, j) + 1, M(i, j-1) + 1),$

если $S_1[i] = S_2[j]$;

$M(i, j) = \min(M(i-1, j), M(i, j-1)) + 1,$

если $S_1[i] \neq S_2[j]$;

Решением задачи будет значение $M(m,n)$,
где m — длина строки S_1, а n — длина строки S_2.

Пример

S_1 = ”*abc*”, S_2 = ”*aabddc*”

Построим таблицу *M*, нумерация строк и столбцов которой начинается с нуля

и элементами которой будут числа, равные значениям функции, описанной

выше.

-1	0	1	2	3
c	6	5	4	**3**
d	5	4	3	4
d	4	3	2	3
b	3	2	1	2
a	2	1	2	3
a	1	0	1	2
	0	1	2	3
		a	*b*	*c*

Обратный ход

$M[1,3] = 2$, означает, что из строки “*a*” можно получить строку “*aab*”, используя две допустимых операции. В примере за три допустимых операции можно преобразовать строку S_1 в S_2. Для определения операций нужно встать на последний символ строки S_1 и начать движение по таблице от правого верхнего угла. В примере движение начнется с ячейки $M[3,6]$.

Находясь в ячейке $M[i, j]$, будем рассматривать два случая.

1) Если $M[-1, i] = M[j, -1]$, то будем сдвигаться по диагонали влево-вниз, попадая в ячейку $M[i-1, j-1]$. При этом будем перемещаться по строке S_1 на один символ влево, т.е. сделаем текущим в строке символ, находящийся в $i-1$ позиции.

2) Если $M[-1, i] \neq M[j, -1]$, то будем сдвигаться по таблице на одну позицию либо влево, попадая в ячейку $M[i, j-1]$, либо вниз в ячейку $M[i-1, j]$. Этот выбор будет зависеть от того, какой из элементов, находящихся в этих ячейках, меньше. При движении влево будем удалять i-ый символ в строке S_1, перемещась на один символ влево. При движении вниз будем

Последовательность действий для примера

Изначально текущим в строке S_1 является последний символ –

символ *c*. Так как $M[-1, 3] = M[6, -1]$, то осуществляем переход

в ячейку $M[5, 2]$ и текущим в S_1 становится предпослений

символ – *b*. Далее, так как $M[-1, 2] \neq M[5, -1]$, передвигаемся в

ячейку $M[4, 2]$. При этом вставим после текущего символа *b*

символ $S_2[5] = d$ (j=5). Продолжая этот процесс вставим символ

$S_2[4] = d$, затем в строке S_1 сделаем текущим сивол *a*, вставим в

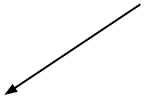
строку S_1 символ *a*. Процесс продолжается до тех пор, пока не

достигнем ячейки $M[0,0]$.

Для нашего примера последовательность операций будет

Отметим, что решений в данной задаче может быть несколько.

Движение по таблице представлено ниже.

	вниз по *i*-му столбцу из *j*-ой строки в *j*–1-ю	INS(S_1, *i*, S_2[*j*])	вставка после *i*-й позиции в S_1 символа S_2[*j*]
	движение влево по *j*-й строке из *i*-го столбца в *i*–1-й	DEL(S_1, *i*)	удаление *i*-го символа в S_1 и передвижение на *i*–1-ю позицию
	движение по диагонали влево вниз		перемещение текущей позиции в S_1 на один символ влево

Задача о телефонном номере (подключена в системе тестирования NSUTS в школьных тренировках)

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MN
7 PRS	8 TUV	9 WXY
	0 OQZ	

Если вы обратили внимание, то клавиатура многих телефонов выглядит как показано –> Использование изображенных на клавишах букв позволяет представить номер телефона в виде легко запоминающихся слов. Многие фирмы пользуются этим и стараются подобрать себе номер телефона так, чтобы он содержал как можно больше букв из имени фирмы.

Напишите программу, которая преобразует исходный цифровой номер телефона в соответствующую последовательность букв и цифр, содержащую как можно больше символов из названия фирмы. При этом буквы из названия фирмы должны быть указаны в полученном номере в той же последовательности, в которой они встречаются в названии фирмы. Например, если фирма называется *IBM*, а исходный номер телефона — **246**, то замена его на ***BIM*** не допустима, тогда как замена его на **2*IM*** или ***B*4*M*** является правильной.

S_1= “IBM”, S_2= “246”. При этом, если в S_1 встречаются буквы, которые соответствуют цифрам номера телефона в нужном порядке, то они останутся без изменения.

Формат входных данных:

Первая строка входного файла содержит название фирмы. Она состоит только из заглавных букв латинского алфавита, количество которых не превышает 80 символов. Вторая прока содержит номер телефона в виде последовательности цифр. Цифр в номере телефона также не более 80.

Формат выходных данных:

В единственной строке выходного файла должно содержаться число букв из измененного номера.

Пример файла входных данных:

IBM

246

Пример файла выходных данных:

2

Пример 5. Задача "Divisibility“ 1999-2000 ACM NEERC

(подключена в системе тестирования NSUTS в школьных тренировках)

Рассмотрим произвольную последовательность целых чисел. Можно поставить знаки операций + или – между целыми в данной последовательности, получая при этом различные арифметические выражения, которые при их вычислении имеют различные значения. Давайте, например, возьмем следующую последовательность: **`17, 5, -21, 15`**. Из нее можно получить восемь различных выражений:

```
17+5+-21+15= 1617-5+-21+15=6
17+5+-21-15=-1417+5--21+15=58
17+5--21-15= 2817-5+-21-15=-24
17-5--21+15= 4817-5--21-15=18
```

Назовем последовательность **делимой** на *K,* если можно так расставить операции + или – между целыми в последовательности, что значение полученного выражения делилось бы нацело на *K*. В приведенном выше примере последовательность делима на **`7 (17+5+-21-15=-14),`** но не делима на 5.

Напишите программу, которая определяет делимость последовательности целых чисел.

Решение

Если число N делится на некоторое число K, то остаток от деления N на K равен 0.

Остаток от деления суммы чисел на некоторое число равен остатку от деления суммы остатков от деления каждого числа на это число.

(a + b) mod c == (a mod c + b mod c) mod c

Пример 6. Задача "Gangsters" (подключена в системе тестирования NSUTS в школьных тренировках)

- N гангстеров идут в ресторан. *i*-ый гангстер заходит в T_i-е время и имеет при себе P_i денег. Дверь ресторана имеет k+1 стадий открытия, выраженных в целых числах от 0 до k. Состояние открытия может измениться на 1 в единицу времени, т.е. либо открыться на 1, либо закрыться на 1, либо остаться прежним. В начальный момент состояние двери закрытое = 0.
- i-тый гангстер может войти в ресторан, если дверь открыта специально для него, т.е. состояние двери совпадает с шириной его плеч S_i. Если в момент времени, когда гангстер подошел к ресторану, состояние открытия двери не совпадает с шириной его плеч, то он уходит и никогда не возвращается. ресторан работает в интервале времени [0, T].
- Цель: собрать в ресторане гангстеров с максимальным количеством денег.

Гангстеры , продолжение

первая строка входного файла содержит значения N, K и T, разделенные пробелами ($1 \le N \le 100$, $1 \le K \le 100$, $0 \le T \le 30000$); вторая строка содержит моменты времени, в которые гангстеры подходят к ресторану T_1, T_2, ... , T_N, разделенные пробелами ($0 \le T_i \le T$ для $i = 1, 2. ..., N$);в третьей строке записаны суммы денег каждого гангстера P_1, P_2, ... , P_N, разделенные пробелами ($0 \le P_i \le 300$, для $i = 1, 2. ..., N$); четвертая строка содержит значения ширины плеч каждого гангстера, разделенные пробелами ($0 \le S_i \le K$ для $i = 1, 2. ..., N$). Все значения целые.

Выходные данные: В выходной файл выдать одно целое число — максимальное значение достатка всех гангстеров, собранных в ресторане. Если ни один гангстер не может попасть в ресторан, выдать 0.

Пример 1

Вход:	Выход:
4 10 20 10 16 8 16 10 11 15 1 10 7 1 8	26

Пример 2

Вход:	Выход:
2 17 100 5 0 50 33 6 1	0

Пример

t = 1 2 3 4 5 6

S = 1 2 3 4 5 1

P = 1 1 1 1 1 100

L – состояние двери

4					**4**	**5**	**5**
3				**3**	**3**	**4**	**5**
2			**2**	**2**	**3**	**3**	**4**
1		**1**	**1**	**2**	**2**	**3**	**103**
0	**0**	**0**	**1**	**1**	**2**	**2**	**3**
0	**0**	**1**	**2**	**3**	**4**	**5**	**6**

← гангстеры

$$m_{i,j} = \max \{ [m_{i-1,j-1}, m_{i-1,j}, m_{i-1,j+1}] + f_i \}$$

где

$$f_i = \begin{cases} p_i, & \text{если } L = s_i \\ 0 \end{cases}$$

Пример 7. Рюкзак 1

Имеется n неделимых предметов, вес i-го предмета есть w_i .

Определить, существует ли набор предметов, суммарный вес которого равен W килограммам.

Если такой набор существует, то определить список предметов в наборе.

Обозначим через $T(n, W)$ функцию, значение которой равно 1, если искомый набор имеется, и равно 0, если набора нет. Аргументами функции будут количество предметов n, по которому можно определить вес каждого предмета, и суммарный вес набора W.

Определим подзадачи, решением которых будут функции $T(i, j)$, где i обозначает количество начальных предметов, j – требуемый суммарный вес, где $0 \le i \le n$, $1 \le j \le W$.

Определим начальные значения функции T:

$T(0, j) = 0$ при $j \ge 1$ (нельзя без предметов набрать массу $j > 0$),

$T(i, 0) = 1$ при $i \ge 1$ (всегда можно набрать вес, равный 0).

Для решения подзадачи, соответствующей функции $T(i, j)$, рассмотрим два случая.

1) i-ый предмет в набор не берется.

Решение задачи с i предметами сводится к решению задачи с i – 1 предметом:

$$T(i,\ j) = T(i - 1,\ j).$$

2) i-ый предмет в набор берется. Вес оставшихся предметов уменьшается на величину w_i:

$$T(i,\ j) = T(i - 1,\ j - w_i).$$

При этом нужно учитывать, что эта ситуация возможна только тогда,

когда масса i-го предмета не больше значения j.

Для оптимального решения из двух возможных вариантов нужно выбрать наилучший. Рекуррентное соотношение при $i \geq 1$ и $j \geq 1$:

$T(i,\ j) = T(i - 1,\ j)$ при $j < w_i$

$T(i,\ j) = max\ (T(i - 1,\ j),\ T(i - 1,\ j - w_i))$ при $j \geq w_i$.

W = 16; $w_1 = 4$; $w_2 = 5$; $w_3 = 3$; $w_4 = 7$; $w_5 = 6$.

W = 16; w_1 = 4; w_2 = 5; w_3 = 3; w_4 = 7; w_5 = 6.

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
3	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
4	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
5	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Решение нашего примера определяется $T[5, 16] = 1$.

$T[5, 16] = T[4, 16]$, то 5-ый предмет можно в набор не включать.

$T[4, 16] \neq T[3, 16]$ –> 4-ый предмет включается. Оставшаяся масса равна $16\text{-}w_4 = 16\text{-}7 = 9$.

$T[3, 9] = T[2, 9]$ –> 3-ый предмет в набор не включаем.

$T[2, 9] \neq T[1, 9]$] –> 2-ой предмет включается. Оставшаяся масса равна $9\text{-}w_2 = 9 - 5 = 4$.

$T[1, 4] \neq T[0, 4]$ –> 1-ой предмет включается, оставшаяся масса равна 0.

Пример 8. Задача о рюкзаке

Задача состоит в том, чтобы определить наиболее ценную выборку из n предметов, подлежащих упаковке в рюкзак, имеющий ограничение по весу, равное W килограмм.

При этом i-ый предмет характеризуется стоимостью c_i и весом

w_i.

Итак, необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины

W, а суммарная стоимость была максимальна.

Если перебирать всевозможные подмножества данного набора из n предметов, то получится решение сложности не менее чем

$O(2^n)$.

В настоящее время неизвестен алгоритм решения этой задачи, сложность которого является полиномиальной. Мы рассмотрим алгоритм решения данной задачи для случая, когда все

Решение

Обозначим через $T(n, W)$ функцию, значение которой соответствует
решению нашей задачи. Аргументами функции является количество
предметов n, по которому можно определить стоимость и массу
каждого предмета, и ограничение по весу W.

Определим подзадачи, решением которых будут функции $T(i, j)$,
а именно, определим максимальную стоимость предметов, которые
можно уложить в рюкзак с ограничением по весу j килограмм, если
можно использовать только первые i предметов из заданных,
где $0 \le i \le n$, $0 \le j \le W$.
Определим начальные значения функции T : $T(0, 0) = 0$,
$T(0, j) = 0$ при $j \ge 1$ (нет предметов, максимальная стоимость равна 0),
$T(i, 0) = 0$ при $i \ge 1$ (можно брать любые из первых i предметов, но

Для решения подзадачи, соответствующей функции $T(i, j)$, рассмотрим два случая.

1) i-ый предмет не упаковывается в рюкзак. Решение задачи с i предметами сводится к решению задачи с $i - 1$ предметом:

$$T(i, j) = T(i - 1, j).$$

2) i-ый предмет упаковывается в рюкзак. Масса оставшихся предметов уменьшается на величину w_i, а при добавлении i-го предмета стоимость выборки увеличивается на c_i:

$$T(i, j) = T(i - 1, j - w_i) + c_i.$$

При этом нужно учитывать, что эта ситуация возможна только тогда, когда масса i-го предмета не больше значения j.

Для оптимального решения из двух возможных вариантов

упаковки рюкзака нужно выбрать наилучший.

Рекуррентное соотношение при $i \geq 1$ и $j \geq 1$:

$T(i, j) = T(i-1, j)$ при $j < w_i$

$T(i, j) = max\,(T(i-1, j),\ T(i-1, j-w_i) + c_i)$ при $j \geq w_i$.

W = 16,

$c1 = 5$,

$w1 = 4$;

$c2 = 7$,

$w2 = 5$;

$c3 = 4$,

$w3 = 3$;

$c4 = 9$,

$w4 = 7$;

$c5 = 8$,

$w5 = 6$.

Пример

W = 16,
*c*1 = 5,
w1 = 4;
*c*2 = 7,
w2 = 5;
c3 = 4,
w3 = 3;
*c*4 = 9,
w4 = 7;
*c*5 = 8,
w5 = 6.

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5
2	0	0	0	0	5	7	7	7	7	12	12	12	12	12	12	12	12
3	0	0	0	4	5	7	7	9	11	12	12	12	16	16	16	16	16
4	0	0	0	4	5	7	7	9	11	12	13	14	16	16	18	20	21
5	0	0	0	4	5	7	8	9	11	12	13	15	16	17	19	20	21

Решение примера определяется $T[5, 16] = 21$.
В примере суммарная масса предметов, подлежащих упаковке в рюкзак, совпадает с *W*, в общем-же случае она не должна превосходить величину *W*.

Обратный ход

Требуется определить набор предметов, которые подлежат упаковке в рюкзак.

Сравним значение $T[n, W]$ со значением $T[n\text{-}1, W]$.

1) Если $T[n, W] \neq T[n-1, W]$, то предмет с номером n обязательно упаковывается в рюкзак, после чего переходим к сравнению элементов $T[n-1, W-w_n]$ и $T[n-2, W-w_n]$ и т.д.

2) Если $T[n-1, W] = T[n-1, W]$, то n-ый предмет можно не упаковывать в рюкзак. В этом случае следует перейти к рассмотрению элементов $T[n-1, W]$ и $T[n-2, W]$.

Пример

В примере $T[5, 16] = T[4, 16]$, поэтому 5-ый предмет в рюкзак не упаковывается. Переходим к сравнению элементов таблицы $T[4, 16]$ и $T[3, 16]$. Их значения не равны, следовательно, четвертый предмет должен быть включен в искомый набор, а ограничение на вес становится равным $16 – w_4 = 16 – 7 = 9$. Далее сравним элементы $T[3, 9]$ и $T[2, 9]$, они равны, поэтому третий предмет в рюкзак не упаковывается и сравниваем $T[2, 9]$ и $T[1, 9]$, они не совпадают, следовательно, второй предмет должен быть взят в рюкзак, а ограничение на вес становится равным $9 - w_2 = 9 – 5 = 4$.

И наконец сравниваем элементы $T[1, 4]$ и $T[0, 4]$, они не равны, поэтому второй предмет включатся в искомый набор, при этом,

```
void Print_item(int i, int j)
{
  if (T[i][j]==0)  //максимальный рюкзак для параметров
      return;      //имеет нулевую ценность
  else if (T[i-1][j] == T[i][j])
          Print_item (i-1,j); //можно составить
           // рюкзак без i-го предмета
       else {
          Print_item(i-1,j-w[i]); //Предмет i
           //упаковывается в рюкзак
          Printf("%d ", i);  // печать i-го предмета
       }
}
```

В программе нужно вызвать функцию **Print_item** с параметрами ***(n,W)*.**

Заметим, что рассуждения были приведены для случая, когда все предметы различны. Самостоятельно рассмотрите, какие изменения будут внесены в таблицу в противном случае.

Пример 9. Задача о расстановке скобок

Рассмотрим вычисление произведения *n* матриц

$$M = M_1 \times M_2 \times \ldots \times M_n. \qquad (1)$$

Порядок, в котором матрицы перемножаются, может существенно сказаться на общем числе операций, требуемых для вычисления матрицы *M,* независимо от алгоритма, применяемого для умножения матриц.

Умножение матрицы размера $[p \times q]$ на матрицу размера $[q \times r]$ требует ***pqr*** операций.

Пример

Рассмотрим произведение матриц:

$$M = M_1 \times M_2 \times M_3 \times M_4$$
$$[10\times20]\ [20\times50]\ [50\times1]\ [1\times 100]$$

Если вычислять матрицу M в порядке: $M1 \times (M2 \times (M3 \times M4,))$, то

потребуется 125 000 операций.

(50*1*100) → [50 ×100], 5000;

(20*50*100) → [20 ×100], 100000;

(10*20*100) → [10 ×100], 20000.

Вычисление же в порядке: $(M1 \times (M2 \times M3))\times M4$ требует лишь 2 200 операций.

(20*50*1) → [20 ×1], 1000;

(10*20*1) → [10 ×1], 200;

(10*1*100) → [10 ×100], 1000.

Перебор с целью минимизировать число операций имеет экспоненциальную сложность.

На первом этапе определим за какое минимальное количество операций можно получить матрицу *M* из равенства (1).

Будем считать подзадачами вычисление минимального количества операций при перемножении меньшего, чем *n,* количества матриц. В качестве параметров рассматриваемой задачи возьмем индексы *i* и *j* (1≤ *i* ≤ *j* ≤ *n),* обозначающие номера первой и последней матриц в цепочке

$$M_i \times M_{i+1} \times \ldots \times M_j .$$

Сначала решим подзадачи, когда *j =i+*1, т.е. когда перемножаются

две рядом стоящие матрицы.

Решения – количество затраченных операций, запишем в ячейке таблицы *T* с номерами (*i. j*).

T_{ij} – число, равное количеству операций при умножении цепочки матриц $M_i \times \ldots \times M_j$, где 1≤ *i* ≤ *j* ≤ n.

Обозначим через t_{ij} минимальную сложность вычисления цепочки матриц $M_i \times M_{i+1} \times \ldots \times M_j$, где $1 \le i \le j \le n$.

Ее можно получить следующим образом:

$$t_{ij} = \begin{cases} 0, & если\ i = j \\ \min\limits_{i \le k < j} (t_{ik} + t_{k+1.j} + r_{ikj}), & если\ j > i \end{cases}$$

Здесь t_{ik} — минимальная сложность вычисления цепочки

$$M' = M_i \times M_{i+1} \times \ldots \times M_k ,$$

а $t_{k+1,j}$ — минимальная сложность вычисления цепочки

$$M'' = M_{k+1} \times M_{k+2} \times \ldots \times M_j .$$

Третье слагаемое r_{ikj} равно сложности умножения M' на M''.

Утверждается, что t_{ij} ($j > i$) — наименьшая из сумм этих трех членов по всем возможным значениям k, лежащим между i и $j - 1$.

Для примера из четырех матриц в таблице будут определены

следующие эпементы T: $t_{1,2}$, $t_{2,3}$ и $t_{3,4}$.

0	10000		
	0	1000	
		0	5000
			0

$M_1 \times M_2 \times M_3 \times M_4$

[10×20] [20×50] [50×1] [1× 100]

$$t_{ij} = \begin{cases} 0, & если\ i = j \\ \min\limits_{i \le k < j} (t_{ik} + t_{k+1.j} + r_{i\,k\,j}), & если\ j > i \end{cases}$$

Далее перейдем к решению подзадач с параметрами $j=i+2$.

Рассмотрим, например, цепочку матриц $M_1 \times M_2 \times M_3$.

Решением этой подзадачи будет минимальное количество операций,

выбранное из двух возможных порядков перемножения матриц: $(M_1 \times M_2) \times M_3$ и $M_1 \times (M_2 \times M_3)$. При этом для выражений в скобках ответы уже записаны в таблице T. Результат запишем в ячейку $T_{1,3}$.

Затем перейдем к решению подзадач с параметрами $j=i+3$ и т.д.

Итак, t_{ij} вычисляются в порядке возрастания разностей нижних

индексов. Процесс начинается с вычисления t_{ii} для всех i, затем

$t_{i,i+1}$ для всех i, потом $t_{i,i+2}$ и т. д. При этом t_{ik} и $t_{k+1,j}$ будут уже вычислены, когда мы приступим к вычислению t_{ij}.

Оценка сложности данного алгоритма есть $O(n^3)$.

В результате работы алгоритма для примера из четырех матриц

будет построена следующая таблица T:

Порядок, в котором можно произвести эти умножения, легко определить,

приписав каждой клетке то значение k, на котором достигается минимум.

0	10000	1200	2200
	0	1000	3000
		0	5000
			0

Алгоритм

```
for (i=0; i<n; i++)    m_{i,i} = 0;
for (l=1; l<n; l++)
  for (i=0; i<n; i++) {
   j = i + l;
   for (k = 0; k < j; k++)
      m_{ij} = min(m_{i,k}+ m_{k+1,j} + r_{i-1}*r_k* r_j)
  }
```

r_{i-1} – количество строк в M'

r_k – количество столбцов в M'

r_j – количество столбцов в M''

Упражнение

Задана строка, состоящая из вещественных чисел, разделенных
арифметическими операциями.

Требуется расставить в строке скобки таким образом, чтобы
значение полученного выражения было максимальнвым.