Основы программирования

Лекция №7

Динамическое программирование

Часть 1

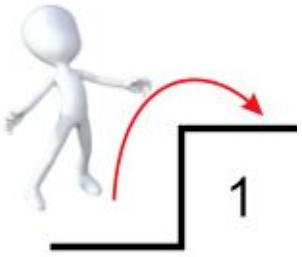
Задача 1:

У нас имеется лесенка из ***n*** ступенек. Перед лестницей стоит человек который за один шаг умеет подниматься либо на следующую ступеньку, либо перепрыгивать через ступеньку.

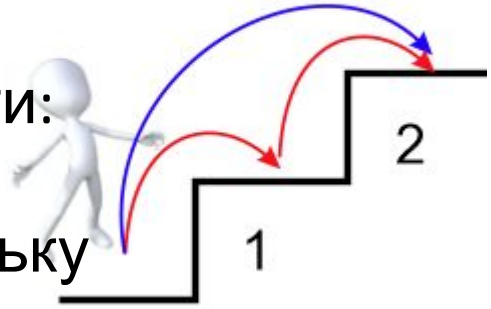
Спрашивается ***сколькими различными самыми оптимальными (за меньшее число ходов) способами он может попасть на последнюю ступеньку.***

Чтобы разобраться в условии задачи и может быть заметить какие-то закономерности начнем с маленьких лестниц.

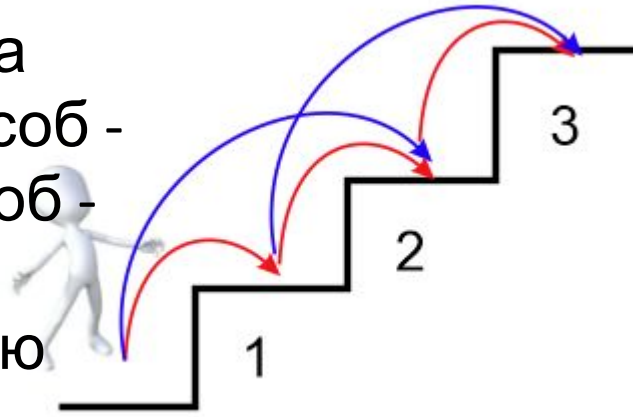
Если лестница состоит всего из одной ступеньки то количество способов (обозначим его a_1) равно единице: можно просто встать на эту ступеньку. $a_1 = 1$.



Если нам нужно попасть на последнюю ступеньку, а их всего 2 то у нас есть два пути:
либо пройти по каждой ступеньке
либо сразу перепрыгнуть на вторую ступеньку
$a_2 = 2$



Ну и наконец, давайте посмотрим еще на лестницу из трех ступенек. **Первый** способ - пойти по всем ступенькам. **Второй** способ - не наступать на первую ступеньку.
И **третий** способ - не наступать на вторую ступень. Итого имеем три способа. $a_3 = 3$



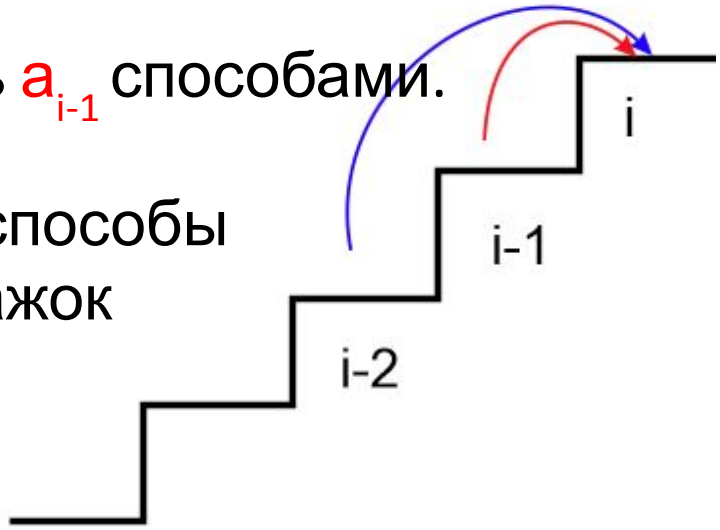
Давайте теперь посмотрим с другой стороны. Рассмотрим какую-то лестницу из ***i***- ступенек.

Как мы могли попасть на i-ю последнюю ступеньку?

Мы могли либо попасть с (i-1) ступеньки.

Это - **первый способ**.

А на (i-1) ступеньку мы могли попасть a_{i-1} способами.



Теперь если мы возьмем все эти a_{i-1} способы и к каждому из них добавим такой шажок в конце с (i-1) на i, то мы получим столько же способов попасть на i-ю ступеньку. То есть, <u>из каждого способа попасть на (i-1) ступеньку получается один способ попасть на i-ю</u> *(как Вы помните из условий задачи, у нас человек умеет ходить только на **одну** или **две** ступеньки!).*

Но кроме этого, есть еще **пути второго вида**. <u>Это путь, ведущий с (i-2) ступеньки напрямую на эту i-ю ступеньку.</u> Таких путей a_{i-2} и также если каждому из этих путей добавить один шаг, то мы снова получим путь на i-ю ступеньку.

Давайте посмотрим, как мы можем попасть на четвертую ступеньку, раскладывая наши пути на две группы.

У нас нарисованы все пути до второй ступеньки, их две штуки, и к каждому из них, мы можем добавить **один большой шаг** (вариант с двумя маленькими шагами мы не рассматриваем, т.к. он не оптимален – два шага больше чем один!) и получаем тоже **два пути** до четвёртой ступеньки.

Ещё, у нас есть три пути попасть на третью ступеньку. И, если мы к каждому из них добавим еще один маленький шажок, то мы получим, соответственно, ещё **три способа** попасть на четвертую ступеньку.

Таким образом, общее количество способов попасть на i-ю ступеньку получается равным $a_i = a_{i-1} + a_{i-2}$.

Очень похоже, что это формула для чисел Фибоначчи. Но это еще пока не совсем.

Для того чтобы это формула задавала числа Фибоначчи нужны соответствующие начальные значения. Потому что, если задать другие начальные значения, то у нас получится другая последовательность.

Давайте посмотрим какие начальные значения нам здесь нужны. У нас есть a_1, это единичка. Мы можем сказать что a_2 равно двойка

А можем посчитать ещё a_0 Формально говоря, это количество способов с 0-й ступеньки попасть на нулевую, то есть количество способов остаться на месте. Таких способов не 0 как может показаться, такой способ 1(один) – это ***просто ничего не делать***. То есть, путь состоит из нуля шагов, но такой путь один – это просто стоять на месте. Потому что, когда мы говорим что у нас **ноль** способов, что-то сделать, это означает что мы не можем это сделать. Здесь же мы **можем** остаться на месте, просто никуда не надо идти. Поэтому a_0 , здесь равно единице и можно, в принципе, начинать эту последовательность с $a_0=1$.

Таким образом, задача вышла, по сути комбинаторная, ***про количество способов*** сделать что-то. Она свелась к вычислению некоторой ***рекуррентной последовательности***. Как ее вычислять – поговорим далее.

Задача 2:

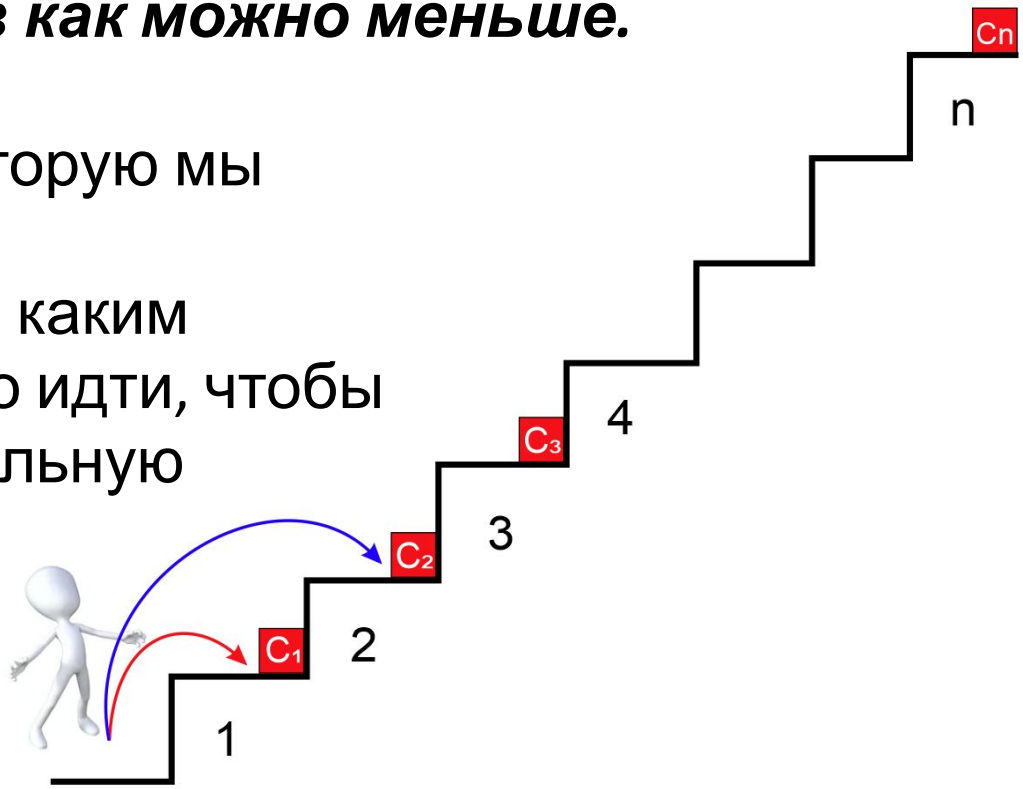
У нас снова имеется лесенка из ***n*** ступенек. Перед лестницей стоит человек который за один шаг умеет подниматься либо на следующую ступеньку, либо перепрыгивать через ступеньку.

На каждой ступеньке стоит устройство, которое берёт плату за проход по данной ступеньке. На каждой ступеньке плата фиксированная, но разная! Обозначим эту плату - C_i.

Нам нужно найти наилучший способ подняться на n-ю ступеньку, заплатив как можно меньше.

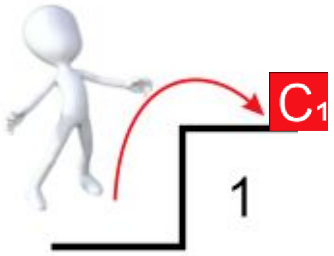
Т.е. нужно найти:

- Минимальную сумму, которую мы можем заплатить
- И построить маршрут, по каким ступенькам лучше всего идти, чтобы оставить такую минимальную сумму



Чтобы разобраться в условии задачи и, может быть, заметить какие-то закономерности снова начнём с маленьких лестниц.

Пусть сумма платы **за весь** подъём до i-й ступеньки будет S_i.

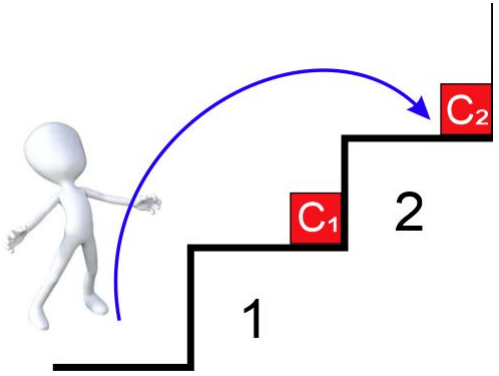


Если лестница состоит всего из одной ступеньки, то это сумма равна C_1, т.к. мы платим только на первом аппарате.

$S_1 = C_1$

Если нам нужно на вторую ступеньку, то нет смысла наступать на первую (чтобы не платить лишнего).

Тогда $S_2 = C_2$



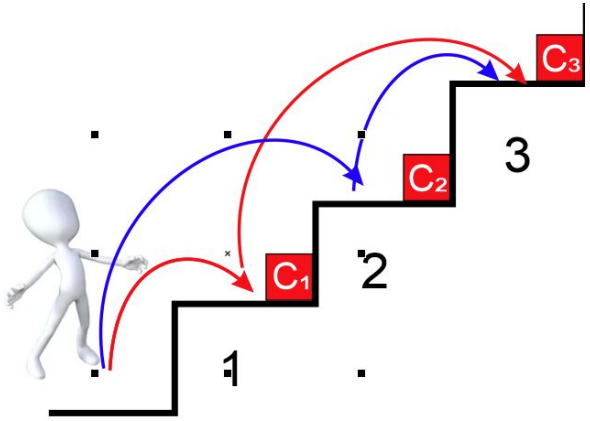
Ну и наконец, давайте посмотрим еще на лестницу из трех ступенек.

Первый способ (как и в прошлой задаче) - пойти по всем ступенькам. Но нам это не выгодно!

<u>Второй способ</u> и <u>*третий способ*</u> - не наступать <u>на первую</u> или <u>*вторую ступеньку*</u>.

На третьей ступеньке мы всё равно заплатим, а дальше есть варианты – платим на первой или на второй. Разумеется мы заплатим там, где цена ниже. Тогда сумма оплаты запишется так:

$S_3 = C_3 + \min(C_1, C_2)$



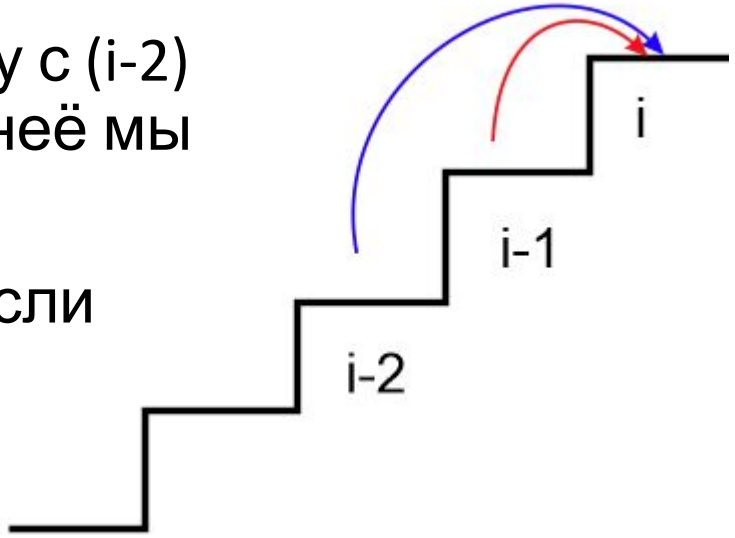
Давайте теперь рассмотрим лестницу из *i* ступенек. Будем искать S_i.

Ясно, что в минимальную стоимость войдёт C_i.

Давайте теперь посмотрим как мы могли попасть на i-ю ступеньку.

1) Мы могли попасть с (i-1) ступеньки. Тогда, как минимум, мы должны были заплатить S_{i-1}.

2) Если мы пришли на i-ю ступеньку с (i-2) ступеньки, то чтобы добраться до неё мы должны заплатить минимум S_{i-2}.



Таким образом, уже понятно, что если выбрать минимальное из S_{i-1} и S_{i-2}, то это и будет минимальная цена, которая и определит до какой ступеньки (i-1 или i-2) нам лучше было сначала добраться.

$S_i = C_i + \min(S_{i-1}, S_{i-2})$

В итоге: мы знаем $S_1 = C_1$, $S_2 = C_2$, формулу для S_i, можем даже посчитать $S_0 = 0$, т.к. если стоим – никому платить не нужно.

Мы с вами рассмотрели две задачи. В одной нужно было посчитать количество способов сделать что-то. В другой задаче нужно из всех этих способов выбрать наилучший. Такие задачи можно решать перебором, то есть можно перебрать все варианты.

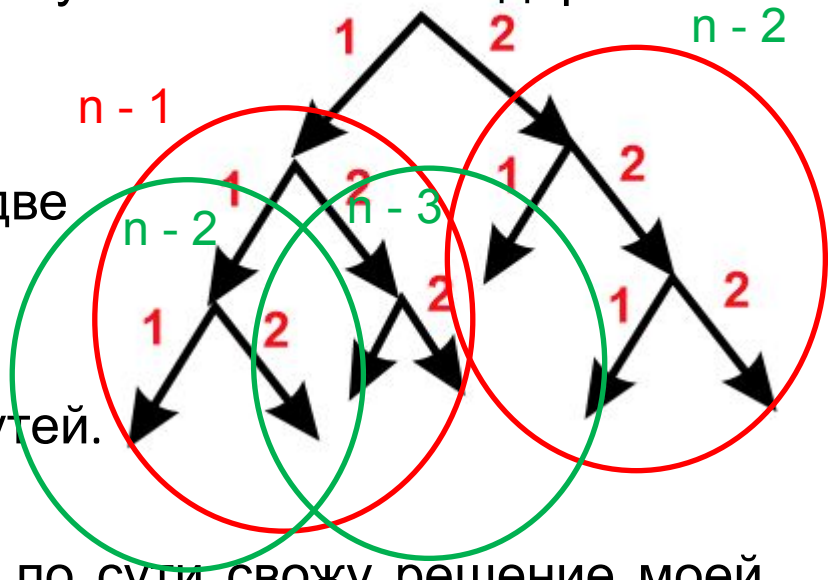
Но чтобы найти наилучший придётся перебрать все варианты. Для этого нам нужно количество операций, как минимум равное количеству вариантов. А количество вариантов здесь растёт экспоненциально, поскольку на каждом шаге у нас есть выбор из двух путей на одну ступеньку или на две. Поэтому, даже если мы ходим на две ступеньки, у нас будет минимальный путь длиной $n/2$. И все равно у нас время будет порядка 2 в какой-то степени (например, $2^{n/2}$), всё равно растёт очень быстро.

А нам хочется считать для каких-то очень больших n, поэтому мы хотим этот перебор как то сократить.

Это можно сделать используя ***Динамическое программирование***.

Это как раз и есть прием, который позволяет нам в подобных задачах сократить перебор и сделать это очень и очень эффективно.

Если мы перебираем варианты, то у нас получается вот такое дерево.



На первом шаге я могу походить на одну ступеньку или на две, на втором шаге - могу опять сходить на одну или на две и так далее. В этом дереве получается экспоненциальное количество конечных вершин, то есть собственно, различных путей. Допустим у нас n ступенек.

Когда я хожу на одну ступеньку, то я по сути свожу решение моей задачи, для n ступенек, к решению задачи для n - 1 ступенек. И к решению задачи, когда я хожу на две ступеньки, для n -2 ступенек. То есть, если мы решим две вот таких задачи для n-1 и для n – 2, то мы решим задачу для n, просто складывая количество путей или выбирая лучший путь там и там, и из них выбирая лучше.

При этом, когда я решаю задачу для n – 1, то я ее по сути свожу тоже к двум задачам: задача для n - 2 ступенек и еще одна задача для минус n-3 ступенек.

Но, в этом случае, задачу для n - 2 ступенек мне приходится решать два раза! А она большая, она решается за большое время. А задача для n-2 соответственно, тоже сводится к задаче для n-3 и на другой ветви задачи для n-2 сводится к задаче для n-3

Поэтому, задачу для n минус трех ступенек мы решаем уже три раза и так далее. Соответственно мы много раз решаем, при переборе, одни и те же задачи.

Что же делает динамическое программирование? Мы один раз решаем какую-нибудь задачу, например задачу для i равного 3 и запоминаем ответ, и после этого, этот ответ мы используем уже в готовом виде и когда решаем задачу для четырех ступенек и когда решаем задачу для пяти ступенек. То есть мы каждую задачу решаем один раз, а поскольку решить задачу зная решение предыдущих задач это довольно быстро, то получается очень быстрый алгоритм, который позволяет нам решить общую задачу.

Таким образом динамическое программирование предлагает нам следующий подход к переборным задачам:

- во-первых, давайте сведём задачу к подзадачам, которые выглядят так же, но имеют размер поменьше. То есть лестницу из n ступенек, сведем к лестнице из n-1 и n-2 ступенек.
- во вторых, заметим что эти подзадачи пересекаются. То есть, у них есть общие части у этих подзадач. Мы будем решать каждую подзадачу только один раз и запоминать её ответ. То есть, сводим к подзадачам, эти подзадачи – пересекающиеся, то есть у них есть какие-то общие части, и мы вычисляем ответ для каждой подзадачи один раз и используемого столько раз сколько нам нужно

Это и дает нам существенный прирост в скорости. То есть вместо 2^n , мы, в данном случае, решаем задачу за время порядка длины лестницы то есть за n.

Это одномерное динамическое программирование, так как мы, по сути, заполняем одномерный массив ($S_i = C_i + \min(S_{i-1}, S_{i-2})$). У нас один цикл который по очереди считает все S_i .

В более сложных задачах нам может понадобиться двумерный массив или трёхмерный массив, 2-3 вложенных цикла и так далее.

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

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 n!, то тривиальной задачей будет 1! = 1 (или 0! =

Недостаток рекурсии:

Простой рекурсивный подход будет расходовать время на вычисление решения 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 (или мемоизацией).

Пример 1. Числа Фибоначчи

```
//рекурсивный вариант
int fib (int n) {
    if (n <= 1) return 1;
    return fib(n - 2) + fib(n - 1);
}
//рекурсия с мемоизацией (мы используем прежнюю функцию)
int fib_num[1000] = {1, 1};
int fib (int n) {
   if ( !fib_num[n] )
    return fib_num[n] = fib(n - 2) + fib(n - 1);
   return fib_num[n];
}
//Динамическое программирование (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 (ДП) 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.

Заполнение таблицы в ДП называется ***прямым ходом***

Исходные данные подзадач называются ***параметрами***. Их Необходимо определить.

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

Под **подзадачей,** понимается та же постановка основной 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 минимальным или максимальным в зависимости от постановки задачи.

Процедура восстановления оптимального решения называется ***обратным ходом***

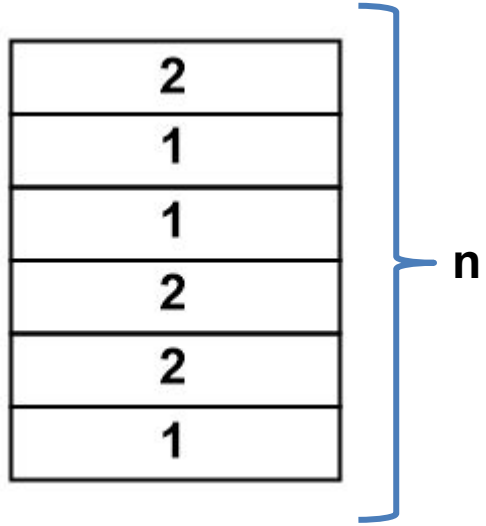
Обычно требуется описать оптимальное решение, выписать рекуррентные соотношения, связывающие оптимальные значения параметра для подзадач, двигаясь снизу вверх, вычислить оптимальные решения для подзадач и, используя их, построить оптимальное решение для поставленной задачи.

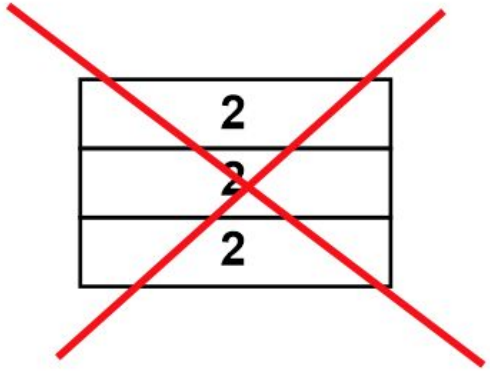
Отметим, что для динамического программирования характерно, что зачастую решается не заданная задача, а более общая, при этом решение исходной задачи является частным случаем решения более общей задачи.

План решения задачи методом ДП

Будем решать задачу и формулировать план.

Задача: Есть контейнеры двух видов, и мы составляем из них стопки, как-то их перемешивая при этом. Нам запрещается класть подряд друг на друга три контейнера 2-го вида. Будем считать, что это опасно. При этом, контейнеры 2-го вида могут как то чередоваться с контейнерами первого вида. И тогда это безопасная стопка. Спрашивается сколько всего разных стопок можно сделать из **n** каких-нибудь контейнеров обоих видов. Самих контейнеров 1 и 2-го вида у нас бесконечный запас, и мы можем составлять из них стопку любой высоты .

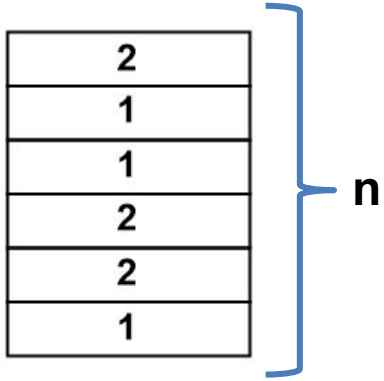


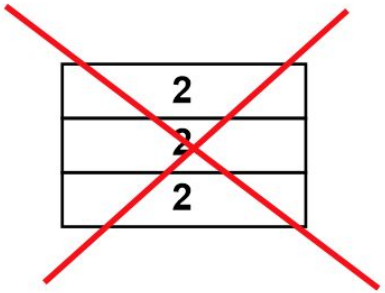


План решения задачи методом ДП

Первый пункт нашего плана - ***Решение задачи для маленьких ограничений***

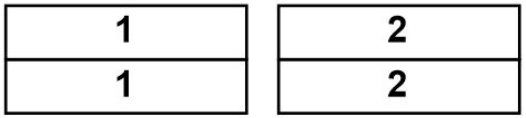
- Таким способом мы поймём условия задачи
- второй смысл этого пункта, эти маленькие значения нам все равно понадобится, когда мы напишем рекуррентной формулу и будем подбирать для нее нужны начальные значения



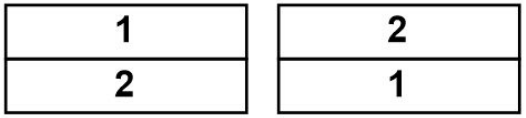


Введём обозначение количества вариантов стопок: a_i.

Очевидно, что стопка из одного контейнера может быть двух видов: 1 или 2-го. Т.е. $a_1 = 2$.



Если у нас стопка высоты два контейнера то она тоже может состоять из любых



План решения задачи методом ДП

Если у нас стопка высоты три контейнера, то вариантов у нас может быть 2^3 (вспомнили комбинаторику?!), но один вариант запрещён условиями задачи (тот, когда стопка из трёх контейнеров 2-го вида), т.е. $a_3 = 2^3-1 = 7$.

Обратите внимание: деление на подзадачи происходит до тех пор, пока не получатся тривиальные задачи, решаемые за константное время

Второй пункт нашего плана – ***Написать что такое*** a_i.

Нужно точно сформулировать - какую же величину мы ищем. В нашем случае, для a_i, - это количество стопок высоты i.

Третий пункт нашего плана – ***Получить рекуррентную формулу*.***

(****рекуррентная формула*** *- это формула, сводящая вычисление n-го члена какой-либо последовательности (чаще всего числовой) к вычислению нескольких предыдущих её членов. Обычно эти члены находятся в рассматриваемой последовательности «недалеко» от её n-го члена, число их от n не зависит, а n-й член выражается через них*

План решения задачи методом ДП

Итак. **Третий** пункт нашего плана – ***Получить рекуррентную формулу.***

Она фактически должна получиться «сама-собой» из некоторых рассуждений про задачу.

В нашем случае, поговорим про стопку высоты i.

Посмотрим на верхней контейнер.

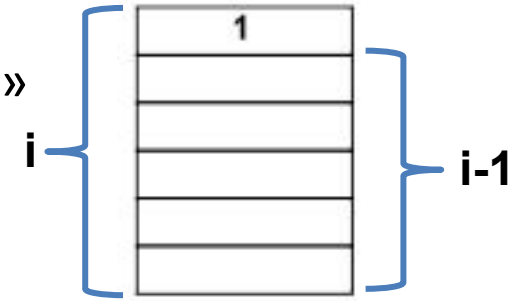


Рис.1

- Если этот верхний контейнер имеет тип 1, то под одним может быть любая стопка высоты i -1 (рис. 1). То есть, выходит a_{i-1} вариантов.
- Если этот верхний контейнер имеет тип 2, то:

– под ним **может быть контейнер 1-го типа**, а под этим контейнером - любая стопка высоты i -2 (рис. 2). Таких вариантов a_{i-2}.

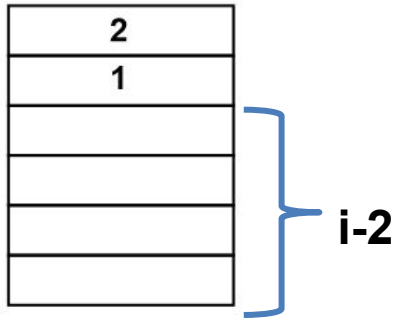


Рис.2

– под ним **может быть контейнер 2-го типа**. Но тогда, ещё ниже, ОБЯЗАТЕЛЬНО находится контейнер типа 1 (*по условиям задачи, мы не можем сложить стопку из трёх контейнеров 2-го вида!*), а под этим контейнером 1-го вида - любая стопка высоты i -3 (рис. 3). Таких стопок a_{i-3}.

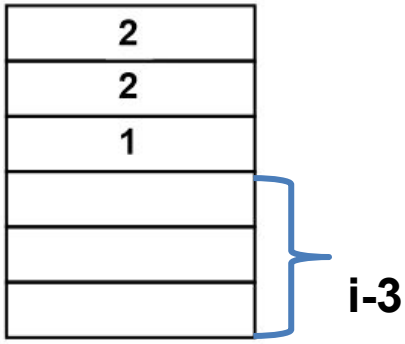


Рис.3

Таким образом, у нас получается, три варианта:

План решения задачи методом ДП

Никаких других вариантов расстановки у нас нет: *либо контейнеров 2-го типа сверху нет, либо один, либо два.* **Таким образом, мы рассмотрели все варианты, они все разные. По этому, мы можем сложить количество** a_{i-1}, a_{i-2} и a_{i-3} и получить рекуррентную формулу:

$$a_i = a_{i-1} + a_{i-2} + a_{i-3}.$$

Обратите внимание, что мы думали именно вот в таком направлении:

мы пытались получить a_i , как-то пытались и вот оно какое-то получилось из предыдущих.

Это не то, что мы взяли там данные a_{i-1} и a_{i-2} и пытались как-то скомбинировать, чтобы получить a_i . Нет, мы пытались наоборот a_i разложить в решение задач меньшего размера, так удобнее искать общую формулу.

Четвёртый пункт нашего плана – ***Это – ограничения и начальные значения.***

План решения задачи методом ДП

Итак, **Четвёртый** пункт нашего плана – ***Это – ограничения и начальные значения.***

Обратите внимание, что рекуррентная формула действует не для всех i ! Например, если мы подставим i=1 единицу, то у нас получится индекс 1-3, т.е. ничего не получится.

Выпишем ограничения при которых формула работает и запишем начальные значения (которых нам, пока, не хватает), при которых данная формула однозначно определяет последовательность.

Как пишутся ограничения. Надо посмотреть на те индексы которые входят в эту последовательность (в формулу для последовательности) и уже видно, что если здесь встречается **i-3**, значит i должно быть, хотя бы, *тройкой*. Да, это если мы допускаем существование a_0 . Если не допускаем, то i должно быть хотя бы четверкой. Итого: **i ≥3**.

Кстати, a_0 здесь тоже можно легко посчитать. Если у нас нет контейнеров, то такая стопка, *одна*. Это - **пустая стопка**. У нас пустое множество контейнеров, оно одно такое. $a_0 = 1$.

План решения задачи методом ДП

Соответственно если $i \geq 3$, это ограничение для рекуррентной формулы, то для всех остальных i надо выписать руками a_0, a_1 и a_2 Мы это уже сделали в начале: $a_1 = 2$, $a_2 = 4$, ну и, на прошлом слайде, $a_0 = 1$.

Тогда , a_3 , можно посчитать по рекуррентной формуле:

$$a_3 = a_0 + a_1 + a_2 = 1 + 2 + 4 = 7 .$$

Зачем четвёртый пункт плана нужен еще.

Когда Вы будете писать программу, Вы будете писать цикл вычисления по рекуррентной формуле и очень важно, чтобы у вас цикл начинался не с нуля или единицы, а именно с $i \geq 3$. Если Вы в этом месте ошибетесь на единичку в ту или иную, сторону то у вас **программа будет работать неверно**. Поэтому, этот пункт имеет двойной смысл:

- Во-первых вы подбираете нужное начальные значения которые задаются перед вычислениям в цикле по этой рекуррентной формуле
- Во-вторых вы пишете ограничение для этого цикла ясно, что он будет работать до n, а вот от какого начального значения – тут нужно явно написать.

План решения задачи методом ДП

Есть еще один пункт, который пока был очевиден, но в дальнейшем, когда мы будем говорить про двумерное динамическое программирование, или если у Вас возникнут какие-то более хитрые одномерные задачи, с ним могут быть проблемы.

Пятый пункт нашего плана – ***Это порядок вычислений.***

Мы вычисляем много значений. В данном случае, a_0 , a_1 , a_2 и так далее. В других задачах, это может быть целая таблица значений если у нас a_{ij} где j прибегает , например, от 0 до n. Может быть там даже трехмерная такая табличка, если это a_{ijk} и возникает вопрос такой: ***«А в каком же порядке вычислять эти значения?»***

Когда у нас массив одномерный, то мы шли слева направо и по очереди вычисляли, но и здесь это может оказаться не так, например, в каких-то задачах нам нужно сначала будет вычислить чётное значение a , потом вычислить нечетные. Ну а если мы заполняем таблицу, то могут быть тоже разные порядки ее заполнения: в какой-то задаче она может заполняться по стрелочкам, в какой-то по столбцам и т.д. Будут даже задачи где нужно ее заполнять например по

План решения задачи методом ДП

Поэтому мы должны определиться как же нам писать наши циклы. То есть, в каком порядке нам нужно заполнять этот массив.

Можно дописать еще и шестой пункт. Его обычно не проговаривают в учебниках, а только упоминают:

Шестой пункт нашего плана – ***Где в таблице лежит ответ.***

Пока (в рассмотренных задачах) он у нас лежал в последнем элементе одномерного массива, но если это будет таблица то он может, например, лежать в последнем столбце последней строки, а может лежать - в первом столбце последний строки. А может быть ответ - это сумма всех элементов в последнем столбце.

С такими задачами Вы тоже будете сталкиваться,

По этому, если Вы, перед тем как писать программу, аккуратно на листочке выполните все эти пункты плана, то написание программы будет чисто автоматическим процессом. При этом ошибиться в ней будет почти невозможно. Если вы попытаетесь писать программу «на глазок», приморно формулу напишите, приморно напишите

План решения задачи методом ДП

1) ***Решение задачи для маленьких ограничений***
2) ***Написать (определить) искомую величину («что такое a_i»)***
3) ***Получить рекуррентную формулу.***
4) ***Выписать ограничения для формулы и её начальные значения.***
5) ***Определить порядок вычислений.***
6) ***Определить где лежит ответ.***

Пользуйтесь планом и у вас не будет проблем с решением задач динамического программирования.

При этом, обратите внимание, что практически все пункты этого плана делаются автоматически. Думать, по большому счету, надо только в **третьем** пункте, когда Вы из задачи получаете рекуррентную формулу. Всё остальное более-менее делается чисто технически, но это надо сделать чтобы быстрее и правильнее безошибочно написать код.

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

При n < 32 полный перебор потребует нескольких секунд, а при n = 64 полный перебор не осуществим в принципе. Для решения задачи методом динамического программирования - сведем исходную задачу к подзадачам.

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

n = 0 seq [n] = 0

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

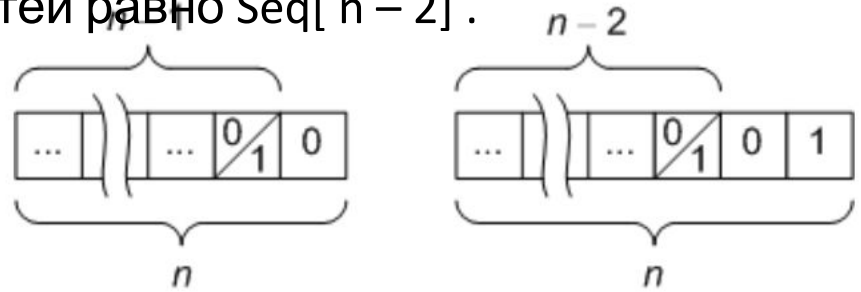
n = 2 seq[n] = 3 00, 01, 10 11

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

Допустим, что мы уже нашли Seq[n–1], Seq[n–2] — число таких последовательностей длины n – 1 и n – 2.

Посмотрим, какой может быть последовательность длины n. Если последний ее символ равен 0, то первые n – 1 — любая правильная последовательность длины n – 1 (не важно, заканчивается она нулем или единицей — следом идет 0). Таких последовательностей Seq[n – 1].

Если последний символ равен 1, то предпоследний символ обязательно должен быть равен 0 (иначе будет две единицы подряд), а первые n – 2 символа — любая правильная последовательность длины n – 2, число таких последовательностей равно Seq[n – 2] .



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

Если дописываем 0: seq[n] = seq[n – 1]

Если дописываем 1: seq[n] = seq[n – 2], т.к. в конце должно быть только …01

Таким образом, Seq[1] = 2, Seq[2] = 3, а

seq[n] = seq[n – 1] + seq[n – 2] , при n > 2.

То есть данная задача фактически сводится к нахождению чисел Фибоначчи.

Пример 3. Сумма квадратов

На какое минимальное количество квадратов можно разложить число *n*?

Любое число всегда можно представить как сумму квадратов других чисел. Обратите внимание, что 1 - это квадрат, и мы всегда можем разбить число на (1*1 + 1*1 + 1*1 + ...).

Учитывая число n, достаточно найти минимальное количество квадратов, сумма которых равна X.

Идея проста, мы начинаем с 1 и переходим к числу, квадрат которого меньше или равен n. Для каждого числа x мы повторяем для n-x.

Пример 1: n = 100; Результат: 1

Объяснение: 100 можно записать как 10^2. Обратите внимание, что 100 также может быть записано как $5^2 + 5^2 + 5^2 + 5^2$, но для этого представления требуется 4 квадрата, а нам нужно минимальное их количество.

Пример 2: n = 6; Результат: 3

Объяснение: 6 может быть записано как $2^2 + 1^2 + 1^2$, т.е. для этого представления требуется 3 квадрата.

Пример 3. Сумма квадратов

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 известны ответы для всех чисел k-1, которые хранятся в каком-нибудь массиве sq, и нам бы хотелось найти sq[k].

Возьмем это число k и проанализируем, какие могут быть ситуации:

1. k является полным квадратом. В этом случае sq[k] = 1.

2. Возможно, предыдущее число k-1 было полным квадратом. Тогда sq[k] = sq[k-1] + 1.

Вообще, вариант прибавления единицы к предыдущему кажется не таким уж плохим.

Теперь предположим, что нам нужно узнать, сколько квадратов будет в разложении числа 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);
   }
}
```

```
Числа: 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 …]
```

(*j* – это размер квадрата)

Пример 3. Сумма квадратов, вариант 2 (рекурсия)

```
// A naive recursive C++ program to find minimum
// number of squares whose sum is equal to a given number
#include <bits/stdc++.h>
using namespace std;

// Returns count of minimum squares that sum to n
int getMinSquares(unsigned int n)
{
   // base cases if n is perfect square then
   // Minimum squares required is 1 (144 = 12^2)
   if (sqrt(n) - floor(sqrt(n)) == 0)
    return 1;
   if (n <= 3)
    return n;

   // getMinSquares rest of the table using recursive
```

Пример 3. Сумма квадратов, вариант 2 (рекурсия)

```
    // formula Maximum squares required is n (1*1 + 1*1 + ..)
    int res = n;
    // Go through all smaller numbers to recursively find
    minimum
    for (int x = 1; x <= n; x++)
    {
     int temp = x * x;
     if (temp > n)
         break;
     else
         res = min(res, 1 + getMinSquares(n - temp));
    }
    return res;
}
// Driver code
int main()
{
    cout << getMinSquares(6);
    return 0;
}
```

Пример 3. Сумма квадратов, вариант 3 (ДП)

```
#include <bits/stdc++.h>
using namespace std;
// Returns count of minimum squares that sum to n
int getMinSquares(int n)
{
   // We need to check base case for n i.e. 0,1,2
   // the below array creation will go OutOfBounds.
   if(n<=3)
   return n;

   // Create a dynamic programming table to store sq
   int* dp = new int[n + 1];

   // getMinSquares table for base case entries
   dp[0] = 0;
   dp[1] = 1;
   dp[2] = 2;
   dp[3] = 3;
```

Пример 3. Сумма квадратов, вариант 3 (ДП)

```
    // getMinSquares rest of the table using recursive formula
    for (int i = 4; i <= n; i++)
    {
// max value is i as i can always be represented as 1*1+1*1+...
     dp[i] = i;

// Go through all smaller numbers to to recursively find minimum
     for (int x = 1; x <= ceil(sqrt(i)); x++)
     {
         int temp = x * x;
         if (temp > i)
             break;
         else
             dp[i] = min(dp[i], 1 + dp[i - temp]);
     }
    }
    // Store result and free dp[]
    int res = dp[n];
    delete[] dp;
    return res;
}

// Driver code
int main()
{
    cout << getMinSquares(6);
    return 0;
}
```

Пример 3. Сумма квадратов, вариант 4 (ДП с запоминанием)

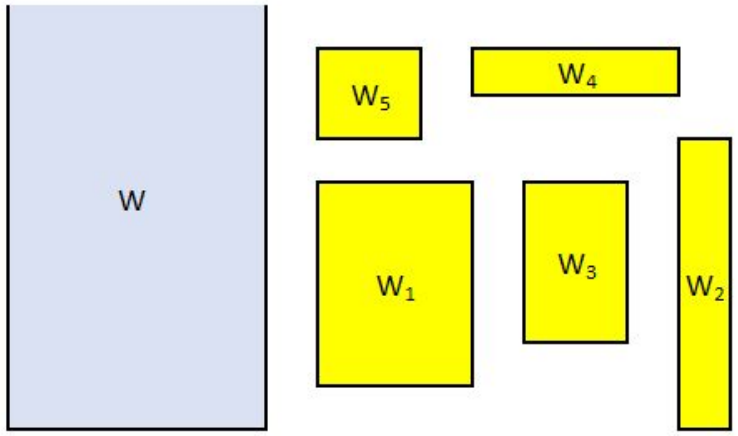
```
#include <bits/stdc++.h>
using namespace std;

int minCount(int n)
{
   int* minSquaresRequired = new int[n + 1];
   minSquaresRequired[0] = 0;
   minSquaresRequired[1] = 1;
   for (int i = 2; i <= n; ++i)
   {
    minSquaresRequired[i] = INT_MAX;
    for (int j = 1; i - (j * j) >= 0; ++j)
    {
        minSquaresRequired[i]
            = min(minSquaresRequired[i],
                minSquaresRequired[i - (j * j)]);
    }

    minSquaresRequired[i] += 1;
   }
   int result = minSquaresRequired[n];
   delete[] minSquaresRequired;
   return result;
}
int main()
{
   cout << minCount(6);
   return 0;
}
```

Пример 4. Рюкзак 1

Имеется *n* неделимых предметов, вес *i*-го предмета равен w_i .
Определить, существует ли набор предметов, суммарный вес которого равен *W* килограммам.
Если такой набор существует, то определить список предметов в наборе.



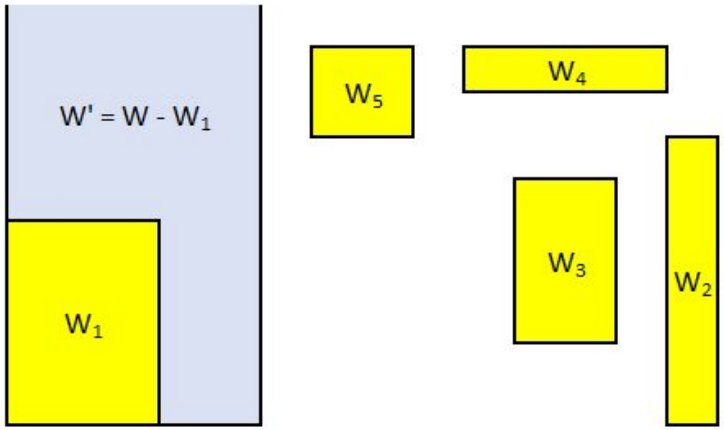
Давайте рассуждать. Для простоты заменим вес на площадь. Пусть рюкзак – это прямоугольник голубого цвета площадью W. Предметы – жёлтые прямоугольники с площадями W_1, W_2 и так далее, до W_5.

Поместим первый предмет в «рюкзак». В этом случае, площадь в «рюкзаке» станет меньше. $W' = W - W_1$. А предметов снаружи останется четыре (рис. 1).

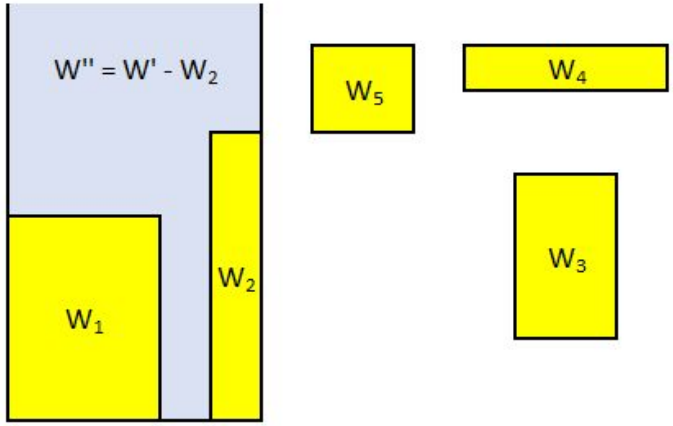
Пусть мы взяли второй предмет. Его площадь W_2. Свободного места останется $W'' = W' - W_2$. Теперь предметов снаружи остаётся три (рис. 2).

То есть, на каждом новом шаге, мы решаем одну и ту же задачу о размещении в «рюкзаке» (каждый раз) меньшей площади меньшего набора предметов.

1)



2)



Продолжая эти шаги мы придём к задаче размещения набора из одной вещи в «рюкзаке» некоторого начального размера.

Давайте так и сделаем. Разобьём большую задачу на маленькие:

1) Будем решать задачу для «рюкзаков» разного размера от 0 до W
2) Будем добавлять в рюкзак вещи, постепенно добавляя в набор предмет. Сначала в набор войдёт предмет №1, потом набор из 1-го и 2-го предметов, потом 1-й и/или 2-й и/или 3-й и так далее.

Получим двумерную таблицу. По одной координате которой отложим размеры рюкзаков, а по другой – наборы предметов которыми будем пытаться **набирать вес равный весу для данного конкретного рюкзака**.

Обозначим через $T(n, W)$ функцию, значение которой будет равно **1**, если искомый набор имеется (то есть, данный набор предметов **позволяет набрать вес равный весу для рюкзака**), и равно **0**, если данный набор предметов **не позволяет** набрать суммарный вес равный весу для рюкзака.

Аргументами этой функции у нас будут:

- n - номер вещи, которую мы будем добавлять к текущему набору и, тем самым, увеличивать количество возможных комбинаций, которыми мы можем попытаться набрать нужный вес
- размер «рюкзака» W на i-м шаге.

Определим подзадачи – вычисление функции $T(i, j)$, где:

i – количество начальных предметов в наборе,

j – требуемый суммарный вес («маленький рюкзак»), где $0 \le i \le n, 1 \le j \le W$.

Ещё раз, параметры функции - количество предметов и требуемый суммарный вес

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:

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

Тогда решение задачи с i предметами сводится к решению задачи на предыдущем шаге, то есть с (i – 1) предметом

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 :

- Масса оставшихся предметов уменьшается на величину w_i
- 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 $j \geq 1$:

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

Начальные значения:

$$T(0, j) = 0 \text{ при } j \geq 1,$$
$$T(i, 0) = 1 \text{ при } i \geq 1.$$

Начальные значения:

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

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

$T(i,\ j) = T(i-1, j)$ при $j < w_i$

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

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

$W_{макс} = 16$	**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
$W_1 = 4$	**1**	1				1												
$W_2 = 5$	**2**	1				1	1				1							
$W_3 = 3$	**3**	1			1	1	1		1	1	1			1				
$W_4 = 7$	**4**	1			1	1	1		1	1	1	1	1	1		1	1	1
$W_5 = 6$	**5**	1			1	1	1	1	1	1	1	1	1	1	1	1	1	1

Результат прямого хода

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

Обратный ход:

- Решение нашего примера определяется T[5, 16] = 1.
- *T*[5, 16] = *T*[4, 16], то получается, что 5-ый предмет в набор не включаем.
- *T*[4, 16] ≠ *T*[3, 16] –> **4-ый предмет включается**. Оставшаяся масса равна 16-w_4 = 16-7 = 9.
- *T*[3, 9] =*T*[2, 9] –> 3-ый предмет в набор не включаем.
- *T*[2, 9] ≠ *T*[1, 9]] –> **2-ой предмет включается**. Оставшаяся масса равна 9-w_2 = 9 - 5 = 4.
- *T*[1, 4] ≠ *T*[0, 4] –> **1-ой предмет включается**, оставшаяся масса равна 4-w_1 = 4 - 4 = 0.

Пример 5. 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 (Все веса – НАТУРАЛЬНЫЕ ЦЕЛЫЕ ЧИСЛА!).

Вообще эта задача для разных условий решается в прикладной математике, криптографии, логистике, генетике, химии, статистике и др. науках

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 входные данные – **целые числа.** Его быстродействие $O(nW)$.

Пример 5. Задача о рюкзаке

Сравнительный анализ методов

Метод	Тип алгоритма	Сложность	Плюсы	Минусы
Полный перебор	точный	$O(2^N)$	точное решение	входные данные не велики временная сложность
Метод ветвей и границ	точный	$O(2^N)$	возможно значительное сокращение времени	работает как полный перебор
Жадный алгоритм	приближенный	$O(N * log\ N)$	высокая скорость может работать с большими значениями N	решение неточное
Динамическое программирование	точный	$O(W*N)$	независимость от вида исходных данных точное решение	большой объём вычислительной работы

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. Для этого будем рассматривать маленькие рюкзаки размера j кг такие, что $0 \le j \le W$ и наборы вещей i, такие, что $0 \le i \le n$, которые будем пытаться туда уложить.

Тогда подзадачи – это вычисление значений функции T(i, j) = max 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 ограничение по весу равно 0).

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. Вес, который мы теперь можем положить в «рюкзачок» размером j, уменьшается на величину w_i, но и при добавлении *i*-го предмета стоимость вещей в «рюкзачке» 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. Если предмет тяжелее грузоподъёмности «рюкзачка», то мы его туда не положим в любом случае.

Для оптимального решения из двух возможных

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 $j \geq w_i$.

Начальные условия:

$T(0,j) = 0$ при $j \geq 0$, (нет предметов, максимальная стоимость равна 0)

$T(i,0) = 0$ при $i \geq 1$. (можно брать любые из первых i предметов, но ограничение по весу равно 0)

W = 16	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
c_1 = 5, w_1 = 4	1	0				5	5	5	5	5	5	5	5	5	5	5	5	5
c_2 = 7, w_2 = 5	2	0				5	7	7	7	7	12	12	12	12	12	12	12	12
c_3 = 4, w_3 = 3	3	0			4	5	7	7	9	11	12	12	12	16	16	16	16	16
c_4 = 9, w_4 = 7	4	0			4	5	7	8	9	11	12	13	14	16	16	18	20	21
c_5 = 8, w_5 = 6	5	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,

Пример

W = 16,
*c*1 = 5,
w1 = 4;
*c*2 = 7,
w2 = 5;
*c*3 = 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 просто 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

Переходим к сравнению элементов таблицы

- В таблице $T[5, 16]$ и $T[4, 16]$ **равны**, поэтому **5-ый предмет** в рюкзак **не упаковывается**.
- $T[4, 16]$ 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**
- 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 **первый предмет включатся** в искомый набор, при этом, ограничение по весу становится равным **0**.

Итак, для нашего примера, в рюкзак упакуются предметы с номерами 1, 2, 4.

Пример

W = 16,

$c_1 = 5$,
$w_1 = 4$;

$c_2 = 7$,
$w_2 = 5$;

$c_3 = 4$,
$w_3 = 3$;

$c_4 = 9$,
$w_4 = 7$;

$c_5 = 8$,
$w_5 = 6$.

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 $T[5,16] = 21$. Это максимальная стоимость набора предметов, которые можно упаковать в рюкзак W кг.

- В примере суммарная масса предметов, подлежащих упаковке в рюкзак, совпадает с *W*, в общем-же случае она просто не должна

```
void Print_item(int i, int j)
{
  if (T[i][j]==0)  //набор предметов построен. 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.

Пятиминутка

Дан набор из 5 вещей:

Номер вещи	1	2	3	4	5
Стоимость	2	3	10	4	8
Вес	1	6	3	4	2

Построить таблицу для определения оптимальной выборки из **5** вещей при ограничении **10 кг**

№\кг	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											
5											

По таблице определить:

1) стоимость оптимальной выборки_____ 2) набор вещей, которые ее составляют_____