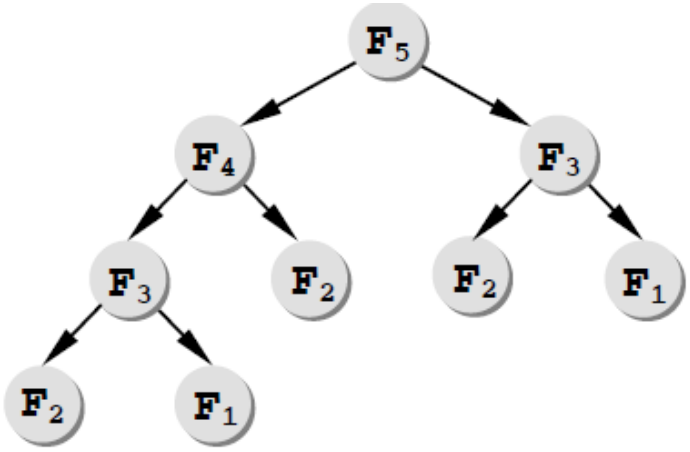
Динамическое программирование

Мы уже сталкивались с последовательностью чисел Фибоначчи:

$$F_1 = F_2 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ при } n > 2$$



Для их вычисления можно использовать рекурсивную функцию:

```
def Fib ( n ):
  if n < 3: return 1
  return Fib(n-1) + Fib(n-2)
```

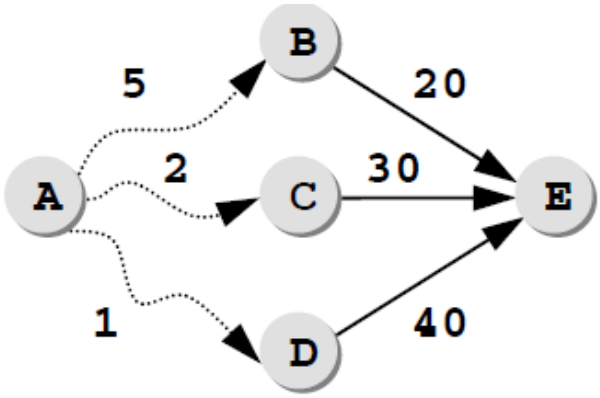
Каждое из этих чисел связано с предыдущими, вычисление F_5 приводит к рекурсивным вызовам, которые показаны на рисунке справа. Таким образом, мы 2 раза вычислили F_3, три раза F_2 и два раза F_1. Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

Напрашивается такое решение – для того, чтобы быстрее найти F_N, будем хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется **F**, сначала заполним его единицами:

```
for i in range(3,N+1):
  F[i] = F[i-1] + F[i-2]
```

Динамическое программирование – это способ решения сложных задач путем сведения их к более простым задачам того же типа

Такой подход впервые систематически применил американский математик Р. Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке. Например, пусть нужно перейти из пункта A в пункт E через один из пунктов B, C или D (числами обозначена «стоимость» маршрута):



Пусть уже известны оптимальные маршруты из пунктов B, C и D в пункт E (они обозначены сплошными линиями) и их «стоимость». Тогда для нахождения оптимального маршрута из A в E нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут A–B–E, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», то есть решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счёт этого удается ускорить выполнение программы. Например, на одном и том же компьютере вычисление $\mathbf{F}_{35}$ в программе на Python с помощью рекурсивной функции требует около 58 секунд, а с использованием массива – менее 0,001 с.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

```
f1 = 1
f2 = 1
for i in range(3,N+1):
  f2,f1 = f1 + f2, f2
```

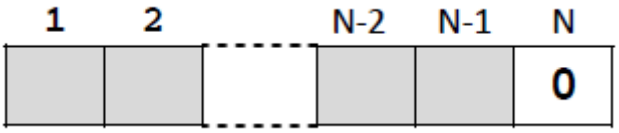
Ответ всегда будет находиться в переменной **f2**.

Задача 1. Найти количество K_N цепочек, состоящих из **N** нулей и единиц, в которых нет двух стоящих подряд единиц.

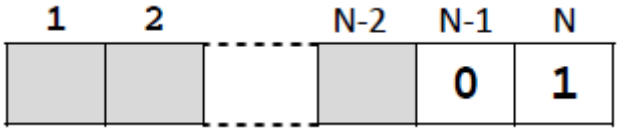
При больших N решение задачи методом перебора потребует огромного времени вычисления. Для того, чтобы использовать метод динамического программирования, нужно

1. выразить K_N через предыдущие значения последовательности K_1, K_2 , …, K_{N-1} ;
2. выделить массив для хранения всех предыдущих значений K_i $(i = 1, \dots N-1)$.

Самое главное – вывести рекуррентную формулу, выражающую N K через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из N бит, последний элемент которой – 0.



Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длиной **N** с нулем в конце существует столько, сколько подходящих последовательностей длины **N−1**, то есть K_{N-1}. Если же последний символ – 1, то вторым обязательно должен быть 0, а начальная цепочка из **N−2** битов должна быть «правильной». Поэтому подходящих последовательностей длиной **N** с единицей в конце существует столько, сколько подходящих последовательностей длины **N−2**, то есть K_{N-2}.



В результате получаем $K_N=K_{N-1}+K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две последовательности длиной 1 (0 и 1), то есть $K_1=2$. Далее, есть 3 подходящих последовательности длины 2 (00, 01 и 10), поэтому $K_2=3$. Легко понять, что решение нашей задачи – число Фибоначчи:

$K_N=F_{N+2}$.

Поиск оптимального решения

Задача 2. *В цистерне N литров молока. Есть бидоны объемом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все используемые бидоны были заполнены и их количество было минимальным.*

Человек, скорее всего, будет решать задачу перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введенного числа **N**.

Самый простой подход – заполнять сначала бидоны самого большого размера (6 л), затем –меньшие и т.д. Это так называемый «жадный» алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для N = 10 «жадный» алгоритм даёт решение 6+1+1+1+1 –всего 5 бидонов, в то время как можно обойтись двумя (5+5).

Как и в любом решении, использующем динамическое программирование, главная проблема – составить рекуррентную формулу. Сначала определим оптимальное число бидонов K_N, а потом подумаем, как определить какие именно бидоны нужно использовать.

Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объем 1 л, в этом случае $K_N=1+K_{N-1}$. Если последний бидон имеет объём 5 л, то $K_N=1+K_{N-5}$, а если 6 л – $K_N=1+K_{N-6}$. Так как нам нужно выбрать минимальное значение, то

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6}).$$

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его нужно сохранить в отдельном массиве **P**. Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начальных значений берем $K_0=0$ и $P_0=0$.

Полученная формула применима при $N \geq 6$. Для меньших N используются только те данные, которые есть в таблице. Например,

$$K_3 = 1 + K_2 = 3, \quad K_5 = 1 + \min(K_4, K_0) = 1.$$

N	0	1	2	3	4	5	6	7	8	9	10
K	0	1	2	3	4	1	1	2	3	4	2
P	0	1	1	1	1	5	6	1	1	1	5

Как по массиву **P** определить оптимальный состав бидонов? Пусть, для примера $N = 10$ Из массива **P** находим, что последний добавленный бидон имеет объем 5 л. Остается 10 – 5= 5 л, в элементе **P[5]** тоже записано значение 5, поэтому второй бидон тоже имеет объём 5 л. Остаток 0л означает, что мы полностью определили набор бидонов.

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда-Уоршелла по сути используется метод динамического программирования.

Задача 3 (Задача о куче). *Из камней весом p_i (i=1, …, N) набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W – целые числа*.

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как (1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из **N** бит. При этом количество вариантов равно $\mathbf{2^N}$, и при больших **N** полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения всех более простых задач этого типа (при меньшем количестве камней и меньшем весе *W*).

Построим матрицу **T**, где элемент **T[i][w]** – это оптимальный вес, полученный при попытке собрать кучу весом **w** из **i** первых по счёту камней. Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берём).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца p_1 – значения p_1 (взяли единственный камень). Это простые варианты задачи, решения для которых легко подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц:

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0								
5	0								
7	0								

Теперь предположим, что строки с 1-ой по **(i-1)**-ую уже заполнены. Перейдем к **i-ой** строке, то есть добавим в набор **i-ый** камень. Он может быть взят или не взят в кучу. Если мы не добавляем его в кучу, то **T[i][w] = T[i-1][w]**, то есть решение не меняется от добавления в набор нового камня. Если камень с весом p_i добавлен в кучу, то остается «добрать» остаток $w-p_i$ оптимальным образом (используя только предыдущие камни), то есть **T[i][w] = T[i-1][w-pi]+pi**.

Как же выбрать, «брать или не брать»? Проверить, в каком случае полученный вес будет больше (ближе к **w**). Таким образом, получается рекуррентная формула для заполнения таблицы:

при $w < p_i$: $T[i][w] = T[i-1][w]$

при $w \geq p_i$: $T[i][w] = \max(T[i-1][w], T[i-1][w-p_i] + p_i)$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке – слева направо:

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0	0	2	2	4	4	6	6	6
5	0	0	2	2	4	5	6	7	7
7	0	0	2	2	4	5	6	7	7

Видим, что сумму 8 набрать невозможно, ближайшее значение – 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом p_i не включен в набор, то **T[i][w] =**

`T[i-1][w]`, то есть число в таблице не меняется при переходе на строку вверх. Начинаем с левого нижнего угла таблицы, идем вверх, пока значения в столбце равны 7. Последнее такое значение – для камня свесом 5, поэтому он и выбран. Вычитая его вес из суммы, получаем 7 – 5 = 2, переходим во второй столбец на одну строку вверх, и снова идем вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взят первый камень с весом 2.

Как мы уже отмечали, количество вариантов в задаче для N камней равно 2^N, то есть алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, то есть сложность нашего алгоритма – *O(N ·W)*.

Однако нельзя сказать, что он имеет линейную сложность, так как есть еще сильная зависимость от заданного веса *W*. Такие алгоритмы называют *псевдополиномиальными*, то есть «как бы полиномиальными». В них ускорение вычислений достигается за счёт использования дополнительной памяти для хранения промежуточных результатов.

Количество решений

***Задача 4**. У исполнителя Утроитель две команды, которым присвоены номера:*

```
1. прибавь 1
2. умножь на 3
```

Первая из них увеличивает число на экране на 1, вторая – утраивает его. Программа для Утроителя – это последовательность команд. Сколько есть программ, которые число 1 преобразуют в число 20?

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для числа 1существует только одна программа – пустая, не содержащая ни одной команды. Для числа 2 есть тоже только одна программа, состоящая из команды сложения. Если через K_N обозначить количество разных программ для получения числа *N* из 1, то $K_1=K_2=1$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности $K_1, K_2, \ldots, K_{N-1}$, то есть с решениями таких же задач для меньших *N*.

Если число *N* не делится на 3, то оно могло быть получено только последней операцией сложения, поэтому $K_N=K_{N-1}$. Если *N* делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить K_{N-1} (количество программ с последней командой сложения) и $K_{N/3}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-1}, & \text{если } N \text{ не делится на 3} \\ K_{N-1} + K_{N/3}, & \text{если } N \text{ делится на 3} \end{cases}$$

Остается заполнить таблицу для всех значений от 1 до заданного N = 20. Для небольших значений N эту задачу легко решить вручную:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_N	1	1	2	2	2	3	3	3	5	5	5	7	7	7	9	9	9	12	12	12

Заметим, что количество вариантов меняется только в тех столбцах, где *N* делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и первый):

N	1	3	6	9	12	15	18	21
K_N	1	2	3	5	7	9	12	15

Заданное число 20 попадает в последний интервал (от 18 до 20), поэтому ответ в данной задаче – 12.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив K, индексы которого изменяются от 0 до N , и заполнить его по приведённым выше формулам:

```
K = [0]*(N+1)
K[1] = 1
for i in range(2,N+1):
  K[i] = K[i-1]
  if i % 3 == 0:
    K[i] += K[i//3]
```

Ответом будет значение **K[N]**

Задача 5 (Размен монет). *Сколькими различными способами можно выдать сдачу размером W рублей, если есть монеты достоинством pi=(i=1,2,…,N)? Для того, чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль (p_1=1).*

Это задача, так же, как и задача о куче, решается полным перебором вариантов, число которых при больших N очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших значений N и W).

В матрице **T** значение **T[i][w]** будет обозначать количество вариантов сдачи размером **w** рублей (**w** изменяется от 0 до W) при использовании первых **i** монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет (напомним, что p_1=1) есть тоже только один вариант. Поэтому нулевой столбец и первую строку таблицы можно заполнить сразу единицами. Для примера мы будем рассматривать задачу для W =10 и набора монет достоинством 1, 2, 5 и 10 рублей:

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1										
5	1										
10	1										

Таким образом, мы определили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. Для вычисления **T[i][w]** предположим, что мы добавляем в набор монету достоинством **pi**. Если сумма w меньше, чем pi, то количество вариантов не увеличивается, и **T[i][w] = T[i-1][w]**. Если сумма больше **pi**, то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством **pi** использована, то нужно учесть все варианты «разложения» остатка **w-pi** на все доступные монеты, то есть **T[i][w] = T[i-1][w]+T[i][w-pi]**.В итоге получается рекуррентная формула

при **w < p_i**: **T[i][w] = T[i-1][w]**

при **w ≥ p_i**: **T[i][w] = T[i-1][w]+ T[i][w-pi]**

которая используется для заполнения таблицы:

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	3	3	4	4	5	5	6
5	1	1	2	2	3	4	5	6	7	8	10
10	1	1	2	2	3	4	5	6	7	8	11

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счёт хранения промежуточных результатов, однако требует дополнительного расхода памяти