Одномерная динамика

Сегодня начнем изучать **динамическое программирование** (или *динамику*, как сокращенно называют это понятие программисты).

Мы уже несколько раз подбирались к понятию динамического программирования когда обсуждали рекурсию и жадные алгоритмы. Вернемся на несколько занятий назад и вспомним решение задачи о поиске n-ого числа ряда Фибоначчи. Рекурсивный алгоритм работал очень долго, так как значения функции приходилось вычислять многократно.

Для ускорения работы мы подключили модуль `lru_cache` который сохраняет значения вычисленных ранее функций. Сегодня мы рассмотрим альтернативный вариант решения этой задачи – сохраним все промежуточные значения в списке. Обозначим его `dp` и будем хранить в `dp[i]` i-ое число ряда Фибоначчи.

Если вспомнить логику построения решения с помощью рекурсии, то чтобы узнать n-ое число, нам нужно было узнать $n-1$ и $n-2$ число ряда, для них нужно было $n-3$ и $n-4$ и так далее, пока мы не доходили до нулевого или первого числа ряда. После того, как рекурсия получала значения для 0 и 1, значения возвращались обратно. Получается мы проходили двойной путь: сначала делаем запросы от n до 0, а затем возвращаем ответы обратно с 0 до n.

Так или иначе, для вычисления n-ого число нам нужно получить все значения элементов ряда от 0 до n. Давайте собирать ответы сразу, начиная с нуля: `dp[0] = 1`, `dp[1] = 1`, `dp[2] = dp[0] + dp[1]` и так далее. В общем виде можно вычислить `dp[i]` по формуле `dp[i] = dp[i-1] + dp[i-2]`:

```
dp[0] = 1
dp[1] = 1
for i in range(2, n + 1):
    dp[i] = dp[i-1] + dp[i-2]
print(dp[n])
```

Данное решение будет работать за линейное время относительно n и каждое значение считать только один раз.

Таким образом, мы можем дать одно из определений динамического программирования: **динамика – это рекурсия с сохранением.** У нас есть точка выхода из "рекур-

сии"(0 или 1), есть формула вычисления нового значения по уже известным значениям этой последовательности – `dp[i] = dp[i-1] + dp[i-2]`. Такую формулу часто называют **рекуррентной формулой**.

При этом заполнение списка идет от меньших значений к большим, в то время как в рекурсии наоборот – рекурсивная функция вызывается для больших значений, а затем вызывает сама себя для меньших значений. Всё как в рекурсии, но промежуточные значения сохранены в ячейках списка, а не прячутся магическим образом в оперативной памяти с помощью `lru_cache`.

На прошлом занятии мы не смогли жадностью решить задачу о наполнении рюкзака. Давайте рассмотрим один из вариантов этой задачи и найдем решение полным перебором и с помощью динамического программирования.

Задача

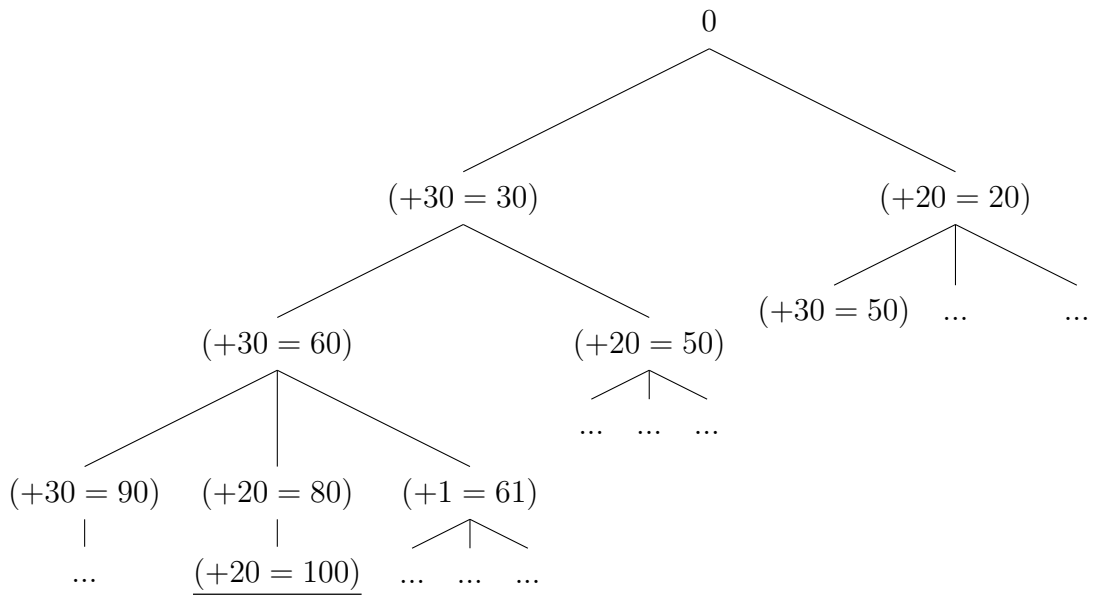
Имеется неограниченное количество купюр стоимостью 1, 20 и 30 единиц. Найдите способ выдать сумму в 100 единиц как можно меньшим количеством купюр.

Жадное решение этой задачи дает неверный ответ, так как оно будет «жадно» набирать сумму купюрами по 30, и в конце придется использовать 10 единиц, которые наберутся 10 купюрами:

$$100 = 30 + 30 + 30 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$$

Жадный алгоритм нашел способ, использующий 13 купюр, однако такое способ далеко не оптимален.

Мы могли бы сделать полный перебор всех вариантов, с помощью которых можно набрать сумму 100, и затем выбрать из этих вариантов тот, который содержит меньше чисел. Такой перебор можно сделать с помощью дерева вариантов, в узлах которого расположим числа-купюры.



Дерево вариантов получения необходимой суммы

Давайте построем такое дерево с помощью кода. С помощью рекурсивного перебора будем строить ветки дерева, пока сумма узлов меньше 100. Такой способ позволит нам рассмотреть все варианты и выбрать лучший:

```
def f(a, n):
    global Min
    if n < 100:
        f(a + [1], n+1)
        f(a + [20], n+20)
        f(a + [30], n+30)
    elif n == 100:
        if len(a) < Min:
            Min = len(a)
Min = 101
f([], 0)
print(Min)
```

Добавив счётчик для количества веток, получим, что такой алгоритм построил 846 575 веток-вариантов!

Попробуем решить эту задачу эффективнее, используя динамическое программирование. Мы знаем как оптимально набрать сумму 0 – нужно 0 купюр. Пусть `dp[i]` – минимальное количество купюр необходимое, чтобы набрать сумму `i`, тогда `dp[0] = 0`.

Теперь допустим, мы только что набрали 100 единиц. Рассмотрим все варианты нашего последней взятой нами купюры:

- сумма была 99 + взяли купюру в 1 единицу

- сумма была 80 + взяли купюру в 20 единиц
- сумма была 70 + взяли купюра в 30 единиц

Так как у нас всего три вида купюр, это все возможные варианты нашего последнего шага.

Представим, что мы уже каким-то образом узнали самый лучший способ получить суммы 99, 80 и 70 (то есть уже узнали `dp[99]`, `dp[80]` и `dp[70]`). Тогда наш ответ для 100 это лучший из способов для 99, 80 и 70 + 1 купюра:

```
dp[100] = min(dp[99], dp[80], dp[70]) + 1
```

Это утверждение будет верным, только если все предыдущие значения будут вычислены лучшим образом, то есть соблюдается правило оптимальности для подзадач.

А как же нам вычислить лучший ответ для 99, 80 и 70 Точно так же, как для 100 – для каждой из этих сумм рассмотрим все варианты получения и выберем оптимальный (например, `dp[80] = min(dp[79], dp[60], dp[50]) + 1`). Продолжая двигаться назад таким образом, мы встретим в самом конце самое лучшее значение для суммы 0 – 0 купюр. Получается, что на каждом шаге мы делали лучший выбор, используя лучшие выборов предыдущих шагов. Значит и итоговое значение для 100 мы получим самым оптимальным способом. Более строго доказать корректность этого алгоритма можно используя принцип математической индукции.

За какое время такой алгоритм найдет ответ? Мы вычислили ответы для всех сумм от 1 до n и на каждом шаге делали выбор из 3-х элементов. Сложность такого решения по времени и по памяти равна $O(n)$.

При написании кода решения этой задачи мы создадим список из 101 элемента, в котором сохраним ответы для всех сумм от 0 до 100. Значение в 0 – ноль. При написании рекурсивной формулы, мы предполагаем, что ответ для текущей суммы всегда можно получить добавлением купюры достоинством 1 к предыдущей сумме. А потом, в зависимости от суммы, пытаемся улучшить ответ с помощью купюр 20 или 30:

```
dp = [0] * 101
a, b, c = 1, 20, 30

for i in range(1, 101):
    dp[i] = dp[i-1] + 1
    if i >= 20:
        dp[i] = min(dp[i-1], dp[i-20]) + 1
    if i >= 30:
        dp[i] = min(dp[i-1], dp[i-20], dp[i-30]) + 1

print(dp[100])
```

```
4
```

Ответ задачи находится в `dp[100]` и равен четырем. Сам список динамики сохранил абсолютно все промежуточные значения, а значит, в отличии от жадности (где мы для получения ответа «прыгаем» жадно только по лучшим промежуточным ответам), динамика вычисляет лучший ответ для каждого возможного варианта.

Итак, сегодня мы познакомились с отличной альтернативой рекурсивным переборам и жадным алгоритмам – динамическим программированием. На следующем занятии мы обсудим общий план решения задач с помощью динамики.