Динамическое программирование

Лекция

Задача о рюкзаке

Knapsack problem

Сложить в ограниченный объём
наиболее ценные предметы

Задача о рюкзаке

- Представьте что вы воришка пробравшийся в музыкальный магазин и у вас есть рюкзак способный унести 4 килограмма.
- На выбор есть 3 предмета:
    - Гитара (1 кг, 1500$)
    - Труба (3 кг, 2000$)
    - Аккордеон (3,5 кг, 3000$)
- Какие предметы украсть выгоднее?

Задача о рюкзаке

простое решение:

- Перебираем все возможные множества товаров и выбираете множество с максимальной стоимостью.

Пусто 0$ / 0 кг	Гитара 1500$ / 1 кг	Труба 2000$ / 3 кг	Аккордеон 3000$ / 3,5 кг
Гитара + Аккордеон не помещается	Гитара + Труба 3500$ / 4 кг	Аккордеон + Труба не помещается	Гитара + Аккордеон + Труба не помещается

- Для 4-х предметов 16 множеств
- Для 5-и предметов 32 множества и т. д.
- Для любого значительного количества предметов это неприемлемо медленно.

Задача о рюкзаке

жадный алгоритм:

- Пока есть место берём самый дорогой товар

Гитара 1500$ / 1 кг	Труба 2000$ / 3 кг	Аккордеон 3000$ / 3,5 кг

- На каждом шаге мы локально выбираем самый выгодный вариант.
- Это пример быстрого и “достаточно” хорошего решения.

- Жадные алгоритмы не заглядывают вперёд. Они повторяют локально оптимальные по какому-либо критерию шаги и надеются, что решение будет глобально оптимальным.

Задача о рюкзаке

динамическое программирование:

- Задача решается путём решения меньшей подзадачи

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1500$)				
Аккордеон 3,5 кг / 3000 $				
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- Строка гитара. Пытаемся уложить гитару в рюкзак. Если ёмкость позволяет кладём гитару.

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1500$)	**1500$ Г**			
Аккордеон 3,5 кг / 3000 $				
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- Посмотрим на следующую ячейку. На этот раз емкость рюкзака составляет 2 кг. Понятно , что гитара здесь поместится!

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	**1500$ Г**		
Аккордеон 3,5 кг / 3000 $				
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- Процедура повторяется для остальных ячеек строки “Гитара”. Наша текущая оценка того что стоит украсть: **Гитара за 1500$**

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $				
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- Теперь укладываем Аккордеон. Появляется выбор между аккордеоном и гитарой.
- Аккордеон не влез, поэтому берём самое выгодное предыдущее решение.

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $	**1500$ Г**			
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- Аккордеон не влез, поэтому берём самое выгодное предыдущее решение.

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $	1500$ Г	**1500$ Г**	**1500$ Г**	
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- И наконец в рюкзаке на 4 кг мы получаем новое выгодное решение, взять Аккордеон.

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $	1500$ Г	1500$ Г	1500$ Г	**3500$** **А**
Труба 3 кг / 2000$				

Задача о рюкзаке

динамическое программирование:

- Теперь проделаем тот же алгоритм для трубы

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $	1500$ Г	1500$ Г	1500$ Г	3500$ А
Труба 3 кг / 2000$	**1500$ Г**	**1500$ Г**		

Задача о рюкзаке

динамическое программирование:

- На третьем шаге новое выгодное решение взять трубу

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $	1500$ Г	1500$ Г	1500$ Г	3500$ А
Труба 3 кг / 2000$	1500$ Г	1500$ Г	**2000$ Т**	

Задача о рюкзаке

динамическое программирование:

- На четвёртом шаге происходит выбор между старым решением на 4 кг взять Аккордеон или (Трубу + самое выгодное решение на 1 кг)

	1 кг	2 кг	3 кг	4 кг
Гитара (1кг / 1000$)	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 3,5 кг / 3000 $	1500$ Г	1500$ Г	1500$ Г	3000$ А
Труба 3 кг / 2000$	1500$ Г	1500$ Г	2000$ Т	**3500$ Т + Г**

Задача о рюкзаке

динамическое программирование:

Итоговая формула:
cell[i,j] = max(предыдущий максимум **cell[i-1, j],**
или **стоимость текущего элемента +**
стоимость оставшегося пространства **cell[i-1,j - вес предмета])**
где i - строки, j - столбцы

	1 кг	2 кг	3 кг	4 кг
Гитара 1кг / 1000$	1500$ Г	1500$ Г	1500$ Г	1500$ Г
Аккордеон 4кг / 3000 $	1500$ Г	1500$ Г	1500$ Г	3000$ А
Труба 3кг / 2000$	1500$ Г	1500$ Г	2000$ Т	**3500$** **Т + Г**

Числа Фибоначчи

Leonardo Fibonacci (1170 - 1242)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, …

Вычисление

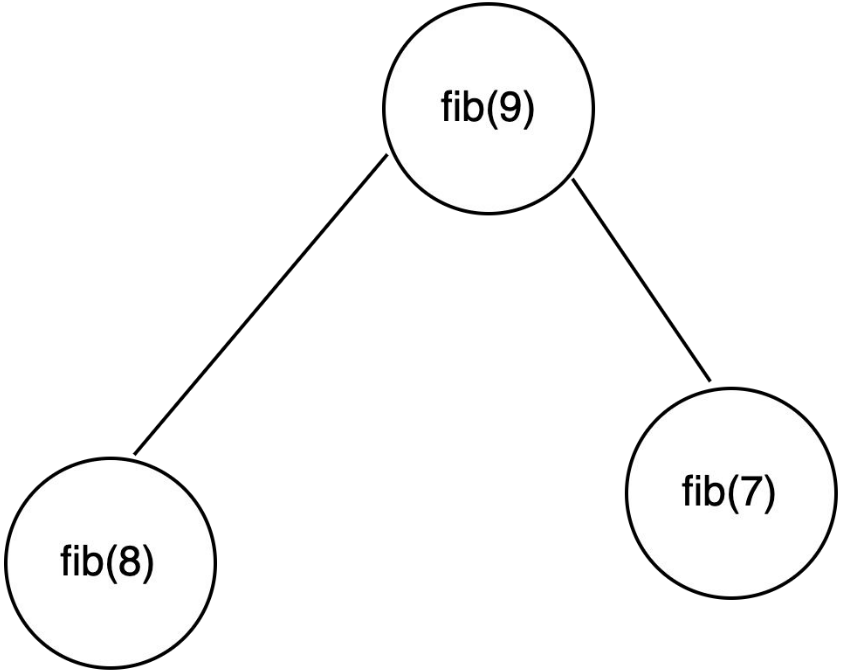
- 0, 1, 1, 2, 3, 5, 8 ……
- F1 = 1; F2 = 1 …….
- Каждое последующее число равно сумме двух предыдущих
- Из определения получаем рекуррентное соотношение
- F(n) = F(n-1) + F(n-2)

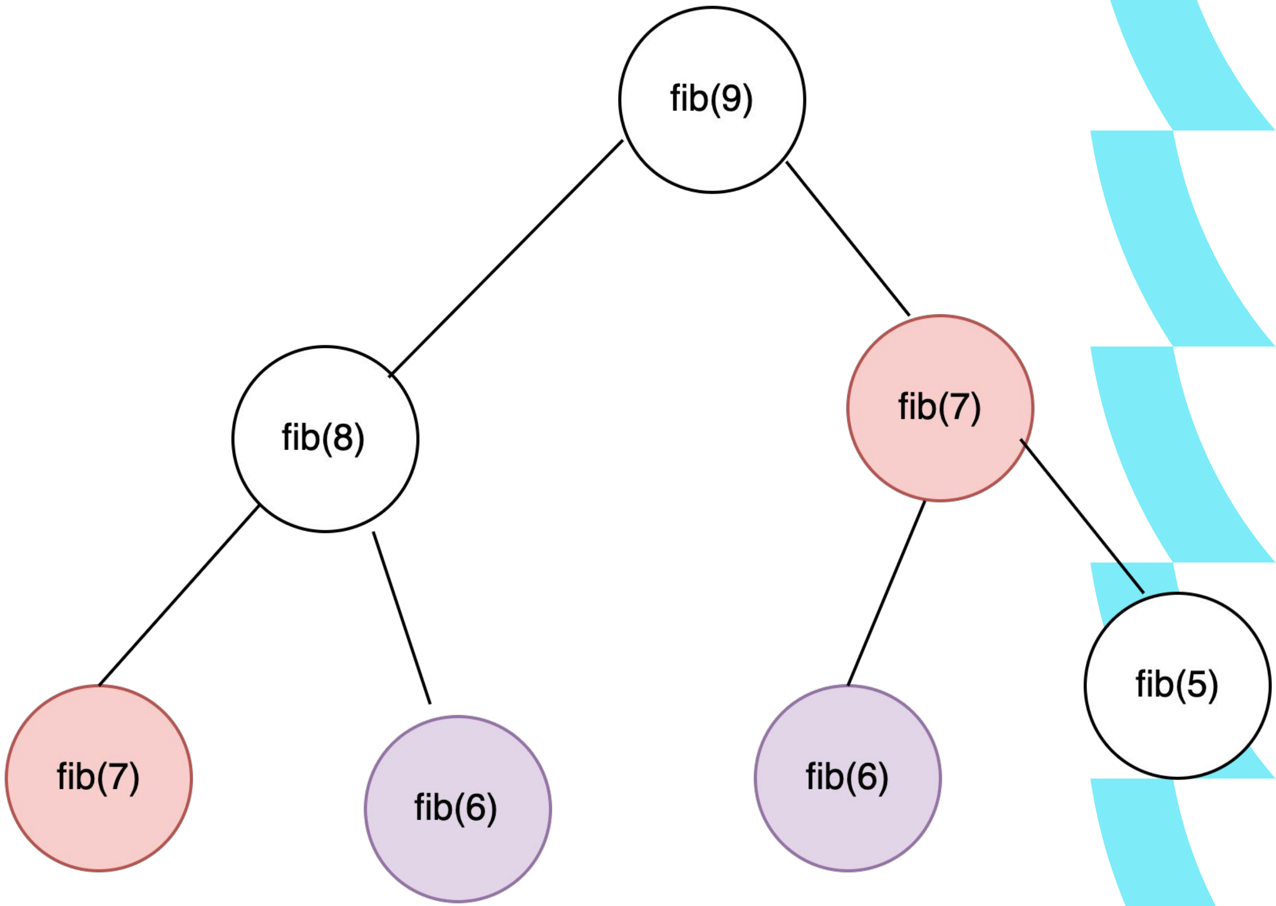
Наивная реализация

- Рекурсивный подход
- Более понятная реализация
- Больше потребления ресурсов
- Относительно небольшое значение можем долго считать

```
function fib(n int) {
    if (n == 0) || (n == 1) || (n == 2) {
        return n
    }
    return fib(n - 1) + fib(n - 2)
}
```

Как мы считаем



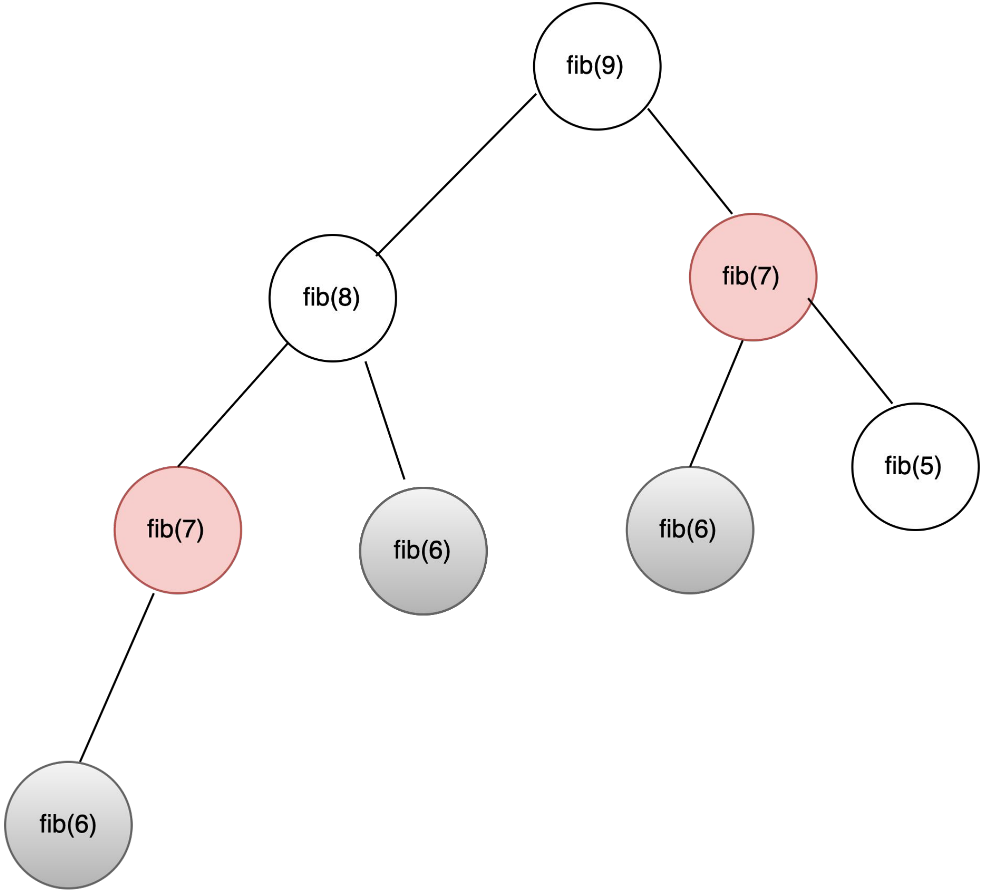

fib(9)
fib(8)
fib(7)
fib(7)
fib(6)
fib(6)
fib(5)

Недостатки

- В коде, приведенном выше, есть два рекурсивных вызова

  (return fib(n-1) + fib(n-2))

- Некоторые значения мы будем высчитывать по многу раз
- Каждый вызов fib ничего не знает о предыдущих вызовах
- На каждом вызове все значения вычисляются заново
- Несмотря на лаконичность рекурсивного подхода, происходят большие затраты на вычисления.



Временная сложность

- Попробуем рассчитать время T(n) для расчета n-ого элемента
- Количество сложений, необходимых для вычисления F(n-1) и F(n-2), тогда будет равно T(n-1) и T(n-2) соответственно
- T(n) = T(n-1) + T(n-2) + 1 (единица это константное действие в нашем коде)
- Рост сложности по времени происходит по экспоненте O(2^n)
- Если быть точнее - рост происходит с коэффициентом Ф (фи)

Более оптимальный подход

- Будем запоминать все, что мы вычисляем
- Для этого заведем массив, в котором будем хранить все промежуточные значения
- Это позволит нам вычислять значение лишь один раз
- Подобного рода подход, когда мы храним предыдущие состояния, называется **мемоизация**


МЕМЫ

Попробуем запоминать результат

- Введем новую переменную **dp**, в которой будем хранить массив
- При попытке рассчитать новое значение мы будем проверять dp, на возможное наличие значения в нем
- И только если его не найдем там, будем считать.

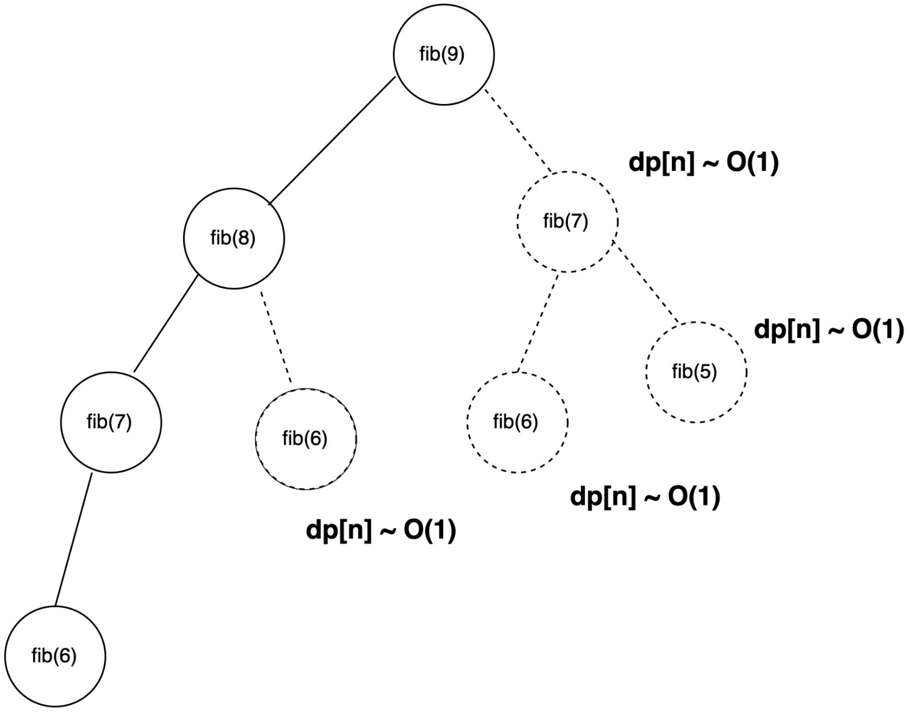
```
function fib(n int) {
    if (n <= 2) {
        return n
    }
    if exist dp[n] {
        return dp[n]
    } else {
        dp[n] = fin(n-1) + fib(n-2)
    }
    return dp[n]
}
```

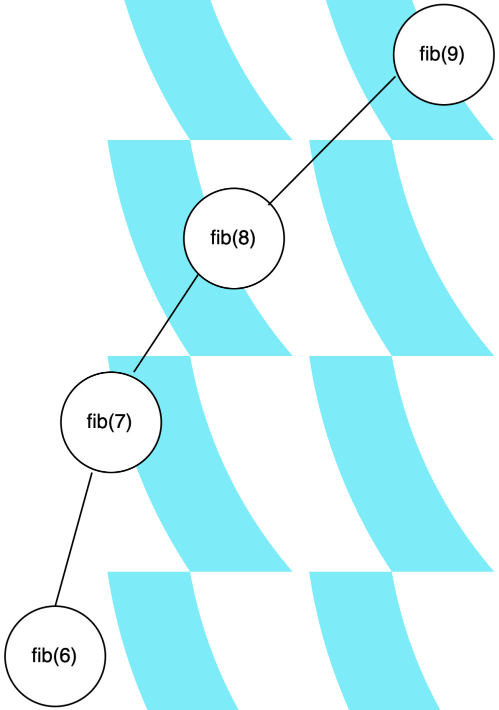
Временная сложность

T(n) ~ O(n)

```
function fib(n int) {
    # сложность O(1)
    if (n <= 2) {
        return n
    }
    # сложность O(1)
    if exist dp[n] {
        return dp[n]
    } else {
        # в эту ветку мы попадем не больше
        # одного раза для каждого n
        dp[n] = fin(n-1) + fib(n-2)
    }
    return dp[n]
}
```

Цепочка вызовов





Итерационный подход

- Определимся с начальными данными
- dp[1] = 1 dp[2] =   1
- Как выглядит наша рекуррентная формула, иными словами, что мы будем считать на каждой итерации?
- dp[i] = dp[i-1] + dp[i-2]

```
function fib(n) {
    dp[1] = 1
    dp[2] = n
    for i = 3; i<=n; i++ {
        dp[i] = dp[i-1] + dp[i-2]
    }

    return dp[n]
}
```

Резюме

- Рекуррентная формула большие значения через меньшие
- Каждое значение мы вычисляем один раз и запоминаем его в массиве или хеш-таблице.
- Когда мы говорили о простой рекурсии без сохранения значений сложность стремилась к экспоненте (~1.5^n)
- Можем вычислять большие значения, например, для 50 значения мы сделаем 50 сложений
- Сложность нашего подхода стремится к O(n)

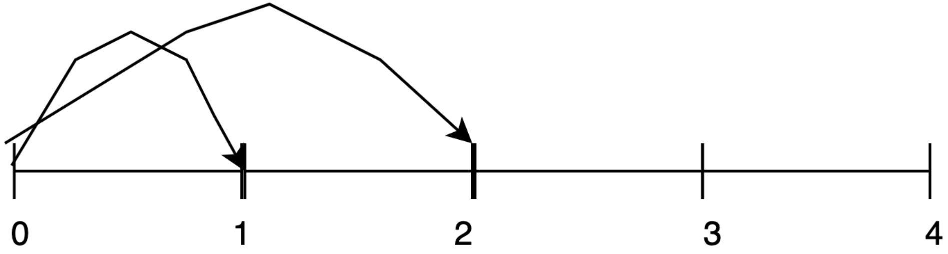
Задача про кузнечика

Классическая задача, относящаяся к так называемому одномерному программированию

Дана прямая, поделенные на отрезки. В самом начале этой прямой сидит кузнечик, который может прыгать только на один или два отрезка вперед. Необходимо понять, сколько есть способов допрыгать до n-го отрезка

Варианты решения

- Сколько способов для n = 3?
- **1+1+1   1 + 2   2 + 1**
- В n мы попадем либо с единицы, либо с двойки. То есть либо с n-1 либо с n-2
- Нам надо сложить количество путей, которые есть, чтобы добраться до 2-ого и до 1-ого отрезка

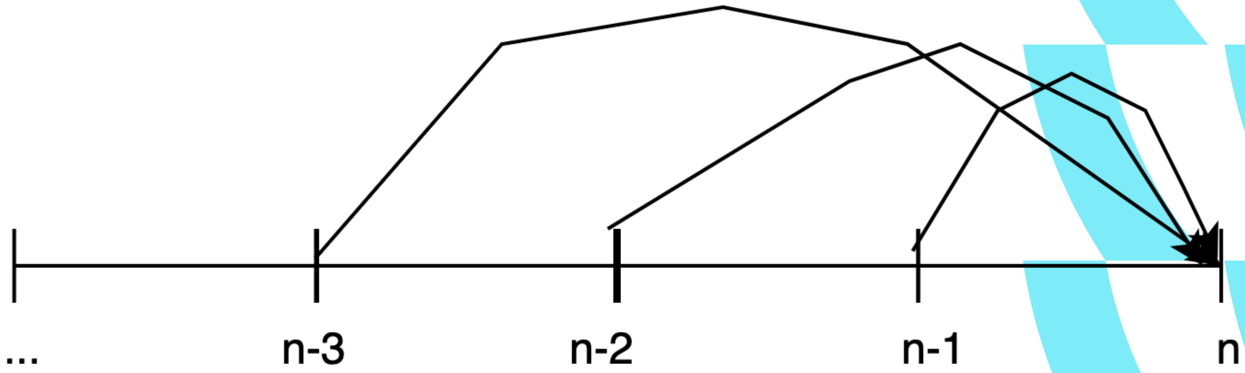


- Получаем известное нам рекуррентное соотношение
- F(n) = F(n-1) + F(n-2) то есть мы начинаем считать с конца.
- dp[i] = dp[i-1] + dp[i-2]

```
# кол-во вариантов добраться до
# нулевой точки
# вырожденный случай
dp[0] = 1
dp[1] = 1
for i = 3; i<=n; i++ {
    dp[i] = dp[i-1] + dp[i-2]
}
```

- Как будет выглядеть путь, например, до 4 отрезка
- *1->1->1->1*; *1->1->2*; *1->2->1*; *2->1->1*; *2->2*
- Можно ли их объединить?
- *1->1->1->1* прыжок с n-1
- *1->1->2* прыжок с n-2
- *1->2->1* прыжок с n-1
- *2->1->1* прыжок с n-1
- 2->2 прыжок с n-2
- Все эти пути объединяет то, что какой длинны не был бы отрезок, в итоговую точку мы придем либо n-1 либо с n-2 отрезка

- Усложним задачу: кузнечик может прыгать на три отрезка
- Это значит, что в конечную точку он может попасть уже не из двух, а из трех мест
- Рекуррентное соотношение теперь тоже изменится: в начальных условиях появляется новое слагаемое
- На нашем отрезке появляется третий с конца отрезок
- В ручную надо посчитать 3 начальных значения



```
dp[0] = 1
dp[1] = 1
dp[2] = 2
for i = 3; i<=n; i++ {
    dp[i] = dp[i-1] + dp[i-2] + dp[i-3]
}
```

Пришло время определиться с обобщением решения для подобного рода задач

Этапы решения задач на ДП

1. Какие значения мы вычисляем и что мы считаем dp[n] кол-во способов допрыгнуть до n-ого столбика
2. Определяемся с рекуррентным соотношением. В нашем случае это “Трибоначчи” так как ряд строится не на двух, а на трёх значениях.
3. Определяемся в базовыми кейсами. В нашем случае для нулевого, первого и второго значения. То есть для тех ситуаций, когда рекуррентная формула может не сработать

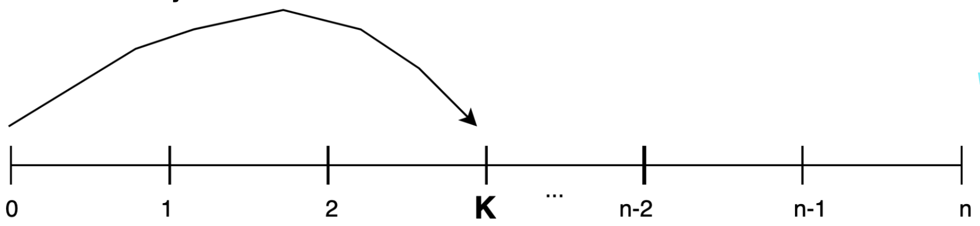
Этапы решения задач на ДП

4. Определиться с порядком вычисления значений. В случае с рекурсией мы начали с конца, но как мы рассматривали ранее, этот подход не самый удачный. Мы вычисляем с начала и при этом запоминаем все результаты вычислений. Надо понимать, что рекуррентная формула не всегда будет такой простой.
5. Сформулировать требования к ответу. В нашем случае это **dp[n]**

Усложним условия

- Все предыдущие формулировки имели недостаток - известно точное кол-во шагов на старте
- Допустим кузнечик может прыгать на K шагов
- Теперь на последний столбик кузнечик может прыгнуть с n-k столбика
- dp[n] = dp[n-1] + dp[n-2] + … + dp[n-k]

$$dp[n] = \sum_{j = 1}^{\min(K,\ i)} dp[i - j]$$



Решение

$$dp[n] = \sum_{j=1}^{\min(K,\, i)} dp[i - j]$$

При условии, что кузнечик может прыгать на K шагов возникают проблемы с рекуррентным соотношением и начальными условиями. Теперь вложенный цикл нам заменяет условия вида

$dp[n] = dp[n-1] + dp[n-2] + \ldots + dp[n-k]$

```
for i=1; i<=n; i++ {
    dp[0] = 1
    // считаем min(k, i)
    r = k
    if (i<r) {
        r = i
    }
    dp[i] = 0
    for j=1; j<=r; j++ {
        dp[i] = dp[i] + dp[i-j]
    }
}
```

Решение на python

```
print(jump(4, 2))
5
print(jump(3, 2))
3
```

```python
def jump(n, k):
    a = [0] * (n + 1)
    a[0] = 1
    for i in range(1, n + 1):
        r = k
        if i < r:
            r = i
        a[i] = 0
        for j in range(1, r + 1):
            a[i] += a[i - j]

    return a[n]
```

Кузнечик и монетки

В условия добавляются монетки. На каждой кочке кузнечик может взять монетки, а может потерять.
Теперь ему необходимо добраться до последней кочки собрав максимальное количество монеток

Как посчитать монетки?

```
def max_coins(n, k, coins):
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        for j in range(1, min(k, i) + 1):
            dp[i] = max(dp[i], dp[i - j] + coins[i - 1])

    return dp[n]
```

- Какое рекуррентное соотношение будет для монеток?
- dp[i] = max(dp[i], dp[i - j] + coins[i - 1])
- Для каждого dp[i] (сумма монеток на конкретной кочке) мы должны теперь рассчитывать максимальное значение.

```
n = 6
k = 2
coins = [0, 4, 15, 9, -7, 0]
result = max_coins(n, k, coins)
print("Максимальное количество монет:", result) #28
```

А как запомнить откуда мы пришли?

- Заводим новый массив jumps, куда мы будем писать наши прыжки
- На каждой итерации, в цикле прыжков, мы должны проверять количество монеток на данном столбике и если оно равно ранее рассчитанному значению - записываем в массив прыжков
- dp[i] == dp[i - j] + coins[i - 1]
- На каждой итерации по i ищем максимальную сумму

```
jumps = []
i = n
while i > 0:
    for j in range(min(k, i), 0, -1):
        if dp[i] == dp[i - j] + coins[i - 1]:
            jumps.append(i)
            i -= j
            break

jumps.reverse()
```

Код целиком

n = 6

k = 2

coins = [0, 4, 15, 9, -7, 0]

result = max_coins(n, k, coins)

print("Максимальное количество монет:", result[0])

print("Количество прыжков:", result[1])

print("Последовательность прыжков:", result[2])

```
Максимальное количество монет: 28
Число прыжков: 4
Последовательность прыжков: [2, 3, 4, 6]
```

```python
def max_coins(n, k, coins):
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        for j in range(1, min(k, i) + 1):
            dp[i] = max(dp[i], dp[i - j] + coins[i - 1])

    max_coins_collected = dp[n]

    jumps = []
    i = n
    while i > 0:
        for j in range(min(k, i), 0, -1):
            if dp[i] == dp[i - j] + coins[i - 1]:
                jumps.append(i)
                i -= j
                break

    jumps.reverse()

    return max_coins_collected, len(jumps), jumps
```

Промежуточные итоги

- Всегда ищем рекуррентное соотношение
- Пытаемся установить начальные значения
- Смотрим, можно ли разбить на одинаковые подзадачи

Черепашка и монетки

Собираем монетки в двумерном пространстве

Черепашка и монетки

- Теперь будем собирать монетки в двухмерном пространстве
- Наша черепашка может двигать только вправо и вниз. Нужно собрать как можно больше монет

0	1	1	1	1	1
0	0	0	0	0	1
0	40	70	0	0	1
100	0	0	0	0	1

- Жадный алгоритм соберёт только 8 монет.

Черепашка и монетки

- Легко посчитать движение по первым горизонтали и вертикали
  dp[0][k] = dp[0][k - 1] + COINS[0][k]
  dp[k][0] = dp[k - 1][0] + COINS[k][0]

0	1	1	1	1	1
0	0	0	0	0	1
0	40	70	0	0	1
100	0	0	0	0	1

Черепашка и монетки

dp[0][k] = dp[0][k - 1] + COINS[0][k]

dp[k][0] = dp[k - 1][0] + COINS[k][0]

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + COINS[i][j]

0	1	1	1	1	1
0	0	0	0	0	1
0	40	70	0	0	1
100	0	0	0	0	1

max max max max max

```
dp = [[None] * M for i in range(N)]

for i in range(N):
    for j in range(M):
        if i == 0 and j == 0:
            dp[0][0] = COINS[0][0]
        elif i == 0:
            dp[0][j] = dp[0][j - 1] + COINS[0][j]
        elif j == 0:
            dp[i][0] = dp[i - 1][0] + COINS[i][0]
        else:
            dp[i][j] = max(dp[i - 1][j],
                           dp[i][j - 1]) +
COINS[i][j]
    print(dp[i])
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 6]
[0, 41, 111, 111, 111, 112]
[100, 100, 111, 111, 111, 112]
```

Черепашка и монетки

восстановление маршрута

- Первый способ восстановления маршрута хранить маршрут в дополнительном массиве prev

0	1	2	3	4	5
0	1	2	3	4	6
0	41	111	111	111	112
100	100	111	111	111	113

```
dp = [[None] * M for i in range(N)]
prev = [[None] * M for i in range(N)]

for i in range(N):
    for j in range(M):
        if i == 0 and j == 0:
            dp[0][0] = COINS[0][0]
            prev[0][0] = -1 # предыдущей клетки нет
        elif i == 0:
            dp[0][j] = dp[0][j - 1] + COINS[0][j]
            prev[0][j] = 0 # слева пришли
        elif j == 0:
            dp[i][0] = dp[i - 1][0] + COINS[i][0]
            prev[i][0] = 1 # сверху пришли
        else:
            dp[i][j] = max(dp[i - 1][j],
                           dp[i][j - 1]) + COINS[i][j]
            if dp[i - 1][j] > dp[i][j - 1]:
                prev[i][j] = 1
            else:
                prev[i][j] = 0
    print(prev[i])
[-1, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1]
[1, 1, 0, 0, 0, 0]
[1, 0, 1, 0, 0, 1]
```

Черепашка и монетки

восстановление маршрута

- Второй способ можно просто догадаться. Ведь она пришла из клетки с максимальным числом монет

0	1	2	3	4	5
0	1	2	3	4	6
0	41	111	111	111	112
100	100	111	111	111	113

```
i, j = N - 1, M - 1
answer = []
answer_directions = []
while i > 0 or j > 0:
   if i != 0 and (j == 0 or dp[i - 1][j] > dp[i][j - 1]):
       i -= 1
       answer_directions.append('DOWN')
   else:
       j -= 1
       answer_directions.append('RIGHT')
   answer.append((i, j))
print answer[::-1] # reverse
print answer_directions[::-1] # reverse
```

```
[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5)]
['RIGHT', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']
```

Спасибо!