МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт математики, механики и компьютерных наук им. И. И. Воровича
Кафедра вычислительной математики и математической физики

# *Курс «Основы алгоритмизации и программирования»*

***Андрей Петрович Мелехов***

# Лекция 2. Рекурсия

**Ростов-на-Дону**

**2020**

## Рекурсивные функции

**Функция, вызывающая саму себя, называется рекурсивной. Такую функцию называют еще явно рекурсивной. Существует и косвенная рекурсия, когда одна функция P вызывает функцию Q, а Q в свою очередь вызывает P. Этот цикл вызовов может состоять и из большего количества функций.**

**Пример 1. Вычисление факториала n!**

```
def fact(k):
    if k == 0 or k == 1:
        return 1  # конечная ветвь
    else: # рекурсивная ветвь k! = k * (k-1)!
        return k * fact(k - 1)

print(fact(5)) # начальный вызов функции
```

**Результат работы программы – значение**

$$120 = 5! = 1 * 2 * 3 * 4 * 5.$$

**У этой функции в рекурсивной ветви происходит сведение полной задачи вычисления факториала k! к подзадаче, вычислению факториала от меньшего числа (k – 1)! и его умножение на k: $k! = k * (k - 1)!$**

**Функция вызывает себя, но с другим аргументом.**

# Стек

**Для хранения локальных переменных и формальных параметров функций у каждой программы отводится специальная область памяти – стек. Это обычная оперативная память компьютера, только к ней применяются специальные правила доступа – по принципу стека (анг. stack – стопка) или по принципу LIFO (Last-In-First-Out, последним пришел – первым ушел). В старых книгах ее еще называли магазинной памятью (магазин автомата). Сейчас чаще всего сравнивают со стопкой: сверху кладут и сверху забирают, то есть элемент, который положили сверху последним, заберут первым.**

**Пример 2.** Косвенная рекурсия

```
def flip(n):
    print('Flip', n, end = ' ')
    if n > 0: flop(n - 1) # вызов функции
                          # flop (определена ниже)
def flop(n):
    print('Flop', n, end = ' ')
    if n > 0: flip(n - 1) # вызов функции flip

flip(10) # первый вызов функции
```

**Результат:**

**Flip 10 Flop 9 Flip 8 Flop 7 Flip 6 Flop 5 Flip 4 Flop 3 Flip 2 Flop 1 Flip 0**

**Замечание.** **В Python можно вызывать функцию определенную ниже (как в этом примере). Нужно только, чтобы в момент первого вызова функции (начала ее работы), она уже была описана выше.**

**Рекурсия является одним из фундаментальных алгоритмических приемов. Существуют известные алгоритмы (например, быстрая сортировка), которые именно в рекурсивной форме имеют естественную и красивую реализацию. Существуют данные, определенные рекурсивно (списки, деревья), для которых естественна рекурсия. Есть известный алгоритм полного перебора (алгоритм с возвратом). В теории**

**компиляторов один из методов построения компилятора – метод рекурсивного спуска.**

**Сначала мы рассмотрим простые задачи, для которых лучше подходят итерационные алгоритмы (с помощью цикла). На этих простых примерах легче понять рекурсивные алгоритмы.**

**А уже потом рассмотрим несколько более сложных примеров, которые решаются лучше всего с помощью рекурсии.**

**Пример 3. (Числа Фибоначчи).** **Вычислить значение n-го числа Фибоначчи, заданного рекуррентной формулой:**

$$F_0 = 1,$$

$$F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2}, \text{ для } n \geq 2.$$

**То есть это последовательность чисел, начинающаяся с двух единиц, и в которой, начиная со второго номера, каждый следующий элемент равен сумме двух предыдущих:**

**1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …**

**Можно написать рекурсивную функцию, воспользовавшись рекуррентной формулой, задающей последовательность.**

```
def fibonacci(n):
    if n == 0 or n == 1: # конечная ветвь
      return 1
    else:                # рекурсивная ветвь
      return fibonacci(n-1) + fibonacci(n-2)

# вызов функции от 7:
print(fibonacci(7))
```

**Результат:**

**21**

**<u>Замечание.</u> Вычислять числа Фибоначчи эффективней с помощью цикла, а не рекурсии!**

**Задачи, которые могут быть решены с помощью рекурсии, имеют следующее свойство: полное решение задачи можно свести с помощью одного или нескольких шагов к такой же задаче, но меньшей (в каком-то смысле) размерности.**

**Например, нужно сделать какие-то действия со всеми n элементами списка.**

**Эту задачу можно разбить на две:**

1. **Выполнить действие для одного элемента.**
2. **Выполнить действие с n – 1 оставшимся элементом (рекурсивный переход).**

**И полное решение есть объединение этих двух решений.**

**Типичная простейшая структура рекурсивного алгоритма:**

**if достигли конечного шага then**

**этот шаг выполняется (конечная ветвь, завершающая рекурсию)**

**else**

**задача сводится с помощью рекурсии к задаче меньшей размерности (рекурсивная ветвь)**

**Конечных и рекурсивных ветвей в подпрограмме может быть несколько. Рекурсивные ветви сводят общую задачу к более простым задачам того же типа, что и вся задача так, чтобы далее можно было продолжить процесс упрощения задачи. Конечные ветви прекращают рекурсивные вызовы, когда задача сведена к простейшим случаям – без них рекурсия получилась бы бесконечной.**

**Пример 4.** **Вычисление элемента последовательности по заданному номеру** $n$**.**

$$a_n = 2a_{n-1} - 4, \qquad a_1 = 3.$$

```
def a(n):
    if n == 1:
        return 3  # конечная ветвь
    else: # рекурсивная ветвь:
        return 2 * a(n - 1) - 4
# вызовы функции при разных n:
for i in range(1,11):
    print('a%d = %d' % (i, a(i)), end =', ')
```

**Результат:**

**a1 = 3, a2 = 2, a3 = 0, a4 = –4, a5 = –12, a6 = –28, a7 = –60, a8 = –124, a9 = –252, a10 = –508,**

**Пример 5.** **Проверка, является ли строка палиндромом (рекурсивная функция с логическим значением)**

```
def palindrom(s):
    if len(s) < 2:
        return True
    else:
        return s[0]==s[-1] and palindrom(s[1:-1])
# Вызов функции:
print(palindrom('12321'))
```

**Дерево рекурсивных вызовов:**

**1) palindrom('12321') проверка '1'=='1' and вызов ↔**

**2) palindrom('232') проверка '2'=='2' and вызов ↔**

**3) palindrom('3') проверка длины 1 < 2, возвращает True**

**В каждом вызове palindrom( ) своя локальная строка s.**

**Пример 6.** Рекурсивный подсчет суммы чисел в списке

```
def sum_list(L, n):
    if n < len(L): # Рекурсивная ветвь
        return L[n] + sum_list(L, n + 1)
    else: return 0 # Конечная ветвь

L = [1, 5, 10, -12]
print(sum_list(L, 0))
```

**Результат: 4**

**1) Вызов: sum_list(L, 0) = L[0] + вызов ↔**

**2) sum_list(L, 1) = L[1] + вызов ↔**

**3) sum_list(L, 2) = L[2] + вызов ↔**

**4) sum_list(L, 3) = L[3] + вызов ↔**

**5) sum_list(L, 4) = 0 и возврат в 4).**

**Пример 7.** Рекурсивный подсчет суммы чисел в многомерном списке произвольной вложенности

```
def sum_list2(L):
    s = 0
    for x in L:
        if not isinstance(x, list):
            s += x
        else: # Рекурсивная ветвь
            s += sum_list2(x)
    return s

L = [1, [[[5, 10], -12], [1, [5]]]]
print(sum_list2(L))
```

**Результат: 10**

**Начальный вызов 1: L = [1, [[[5, 10], –12], [1, [5]]]]**

**sum_list2(L) = 1 + вызов 2**

↓↑

**sum_list2( [[[5, 10], –12], [1, [5]]]) = вызов 3 + вызов 5**

↓↑ ↓↑

**sum_list2( [[5, 10], –12])=вызов 4 + (–12)** **sum_list2([1, [5]]) =1+ выз-6**

↓↑ ↓↑

**sum_list2( [5, 10])= 5 + 10** **sum_list2([5]) =5**

Рассмотрим более сложные задачи. Для этих задач рекурсивные алгоритмы будут давать естественные решения, которые без применения рекурсии решить гораздо сложнее.

**Пример 1. Перестановки.** Вывести на экран все перестановки из n различных объектов. Например, рассмотрим количество объектов **n = 3**, и пусть объектами будут три натуральных числа: **1, 2, 3.**

Все их перестановки:

| | |
|---|---|
| **№ 1** | **1, 2, 3** |
| **№ 2** | **1, 3, 2** |
| **№ 3** | **2, 1, 3** |
| **№ 4** | **2, 3, 1** |
| **№ 5** | **3, 1, 2** |
| **№ 6** | **3, 2, 1** |

Всего 6 перестановок, в общем случае для **n** объектов получится **n!** перестановок.

Эту задачу можно решить с помощью рекурсии, понижая размерность задачи от количества **n** к количеству **n – 1**.

Будем элементы для перестановки хранить в списке **A**. Переставлять будем натуральные числа, как и в примере выше. В начальный момент времени они будут находиться подряд в **A**.

## Алгоритм рекурсивного перехода

1) Если переставить надо один объект (**n = 1**), то решение элементарно – перестановка одна. Это будет конечная ветвь рекурсии, в этот момент будем прерывать рекурсию и выводить перестановку.

2) Как совершить все перестановки из **n** элементов, если, допустим, что для **n – 1** элемента мы умеем их совершать? Это будет рекурсивный переход, сведение задачи к меньшей размерности:

2.1 Оставляем **n**-й элемент на месте и переставляем первые **n – 1** элементы.

2.2 Для каждого элемента **i** от **1** до **n – 1** повторять:

а) меняем местами $a_n$ и $a_i$;

б) переставляем получившиеся первые **n – 1** элементы;

в) меняем обратно местами $a_n$ и $a_i$.

Получится, что на последнем месте в массиве по очереди побывают все элементы массива, а к оставшимся мы будем применять ту же самую рекурсивную подпрограмму **n** раз для меньшей размерности.

Напишем реализацию этого алгоритма.

```
def perestan(n):
    global k
    if n == 1:   # 1) конечная ветвь
        k += 1    # увеличиваем номер
        print('№', k, A) # вывод номера и списка
    else:      # 2) рекурсивная ветвь
        perestan(n - 1)                          # 2.1
        for i in range(n - 1):                   # 2.2
            A[i], A[n - 1] = A[n - 1], A[i]       # а
            perestan(n - 1)                       # б
            A[i], A[n - 1] = A[n - 1], A[i]       # в

k = 0 # - номер текущей перестановки
A = [1, 2, 3] # Задаем список
perestan(len(A)) # Вызов функции
```

**Вывод программы:**

**№ 1 [1, 2, 3]**

**№ 2 [2, 1, 3]**

**№ 3 [3, 2, 1]**

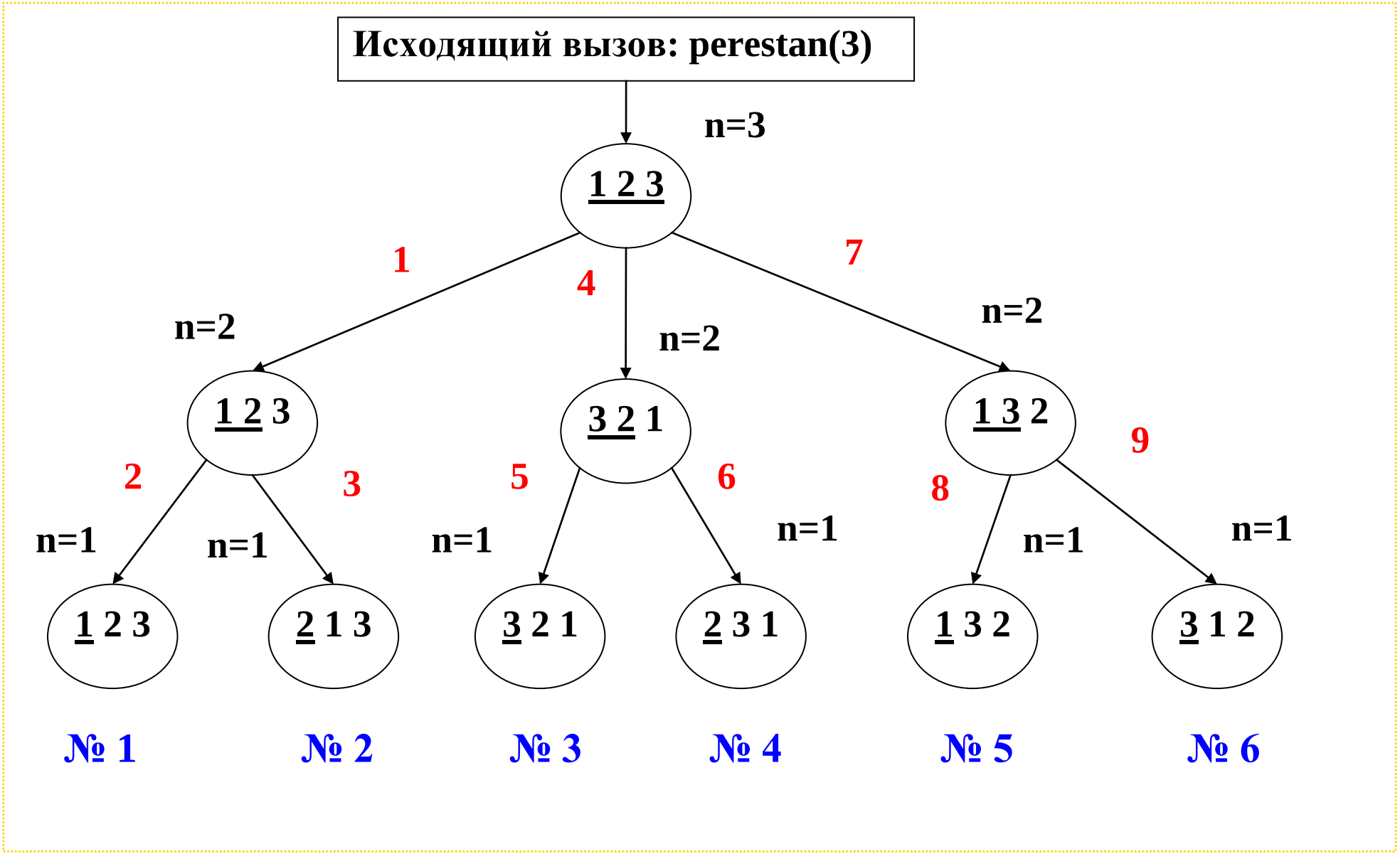
**№ 4 [2, 3, 1]**

**№ 5 [1, 3, 2]**

**№ 6 [3, 1, 2]**

## Разберем этот пример

Вначале список A заполняется подряд натуральными числами: 1, 2, 3. Запускается рекурсивная функция perestan с параметром n = 3. Из этой функции есть три вызова с параметром n = 2, каждая из которых в свою очередь дважды рекурсивно вызывает себя с параметром n = 1 (см. рисунок ниже). Перед вызовом каждой функции происходит перестановка списка, а после вызова обратная перестановка. На рисунке ниже вызовы функции обозначены овалами, внутри которых приведена расстановка массива перед их вызовом и выделена часть списка, с которой эта функция работает. Красным цветом отмечен порядок вызова функций. Номерами внизу отмечен порядок их вывода на экран и номера, с которым они будут выведены.


Исходящий вызов: perestan(3)
n=3
1 2 3
1
4
7
n=2
n=2
n=2
1 2 3
3 2 1
1 3 2
2
3
5
6
8
9
n=1
n=1
n=1
n=1
n=1
n=1
1 2 3
2 1 3
3 2 1
2 3 1
1 3 2
3 1 2
№ 1
№ 2
№ 3
№ 4
№ 5
№ 6

## Задания

1. Что нужно изменить в программе, чтобы она работала с большим количеством элементов в списке? Не с целыми числами, а, например, с символами (‘а’, ‘б’, ‘в’)?
2. Как вывести только те перестановки, в которых элементы чередуются: четный/нечетный?
3. Порядок вывода элементов не совпадает с приведенным порядком в начале лекции. Как изменить программу так, чтобы порядок был таким же?
4*. Решить эту задачу не используя рекурсию.

**Замечание 1.** Вообще теоретически любую рекурсивную программу можно переписать нерекурсивно. Для этого надо заменить работу со стеком внутри процедуры своей динамической структурой (своим стеком) и как-то организовать перебор тех же самых случа-

ев, которые рассматриваются внутри подпрограмм. Программа будет гораздо больше и сложнее.

**Замечание 2.** На самом деле эту задачу можно решить с помощью итерации не имитируя стек. Идея этого алгоритма состоит в том, чтобы установить порядок для всех перестановок и способ перехода от текущей перестановки к следующей. Структура программы будет такой:

получить и вывести первую перестановку;
**while** не дошли до конца **do**
  получить и вывести следующую перестановку;

Подробнее смотрите в книге С.М. Окулов Дискретная математика. Теория и практика решения задач по информатике  и ссылки в ней.

**5. Решить похожую задачу. Даны n различных объектов. Вывести на экран все подмножества из этих элементов.**

**Например, при n = 3 и элементах 1, 2, 3 все подмножества это:**

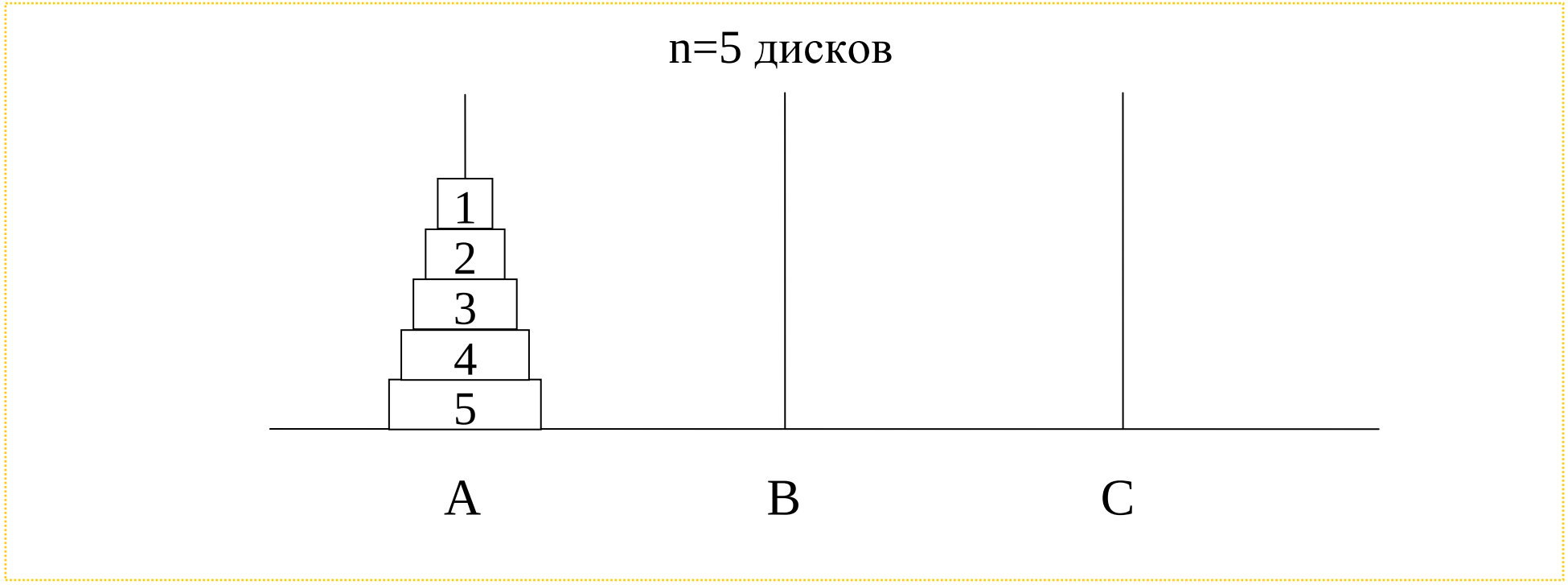
| | |
|---|---|
| **№ 1** | **пустое** |
| **№ 2** | **1** |
| **№ 3** | **2** |
| **№ 4** | **3** |
| **№ 5** | **1, 2** |
| **№ 6** | **1, 3** |
| **№ 7** | **2, 3** |
| **№ 8** | **1, 2, 3** |

**Всего 8 подмножеств, в общем случае для n объектов получится** $2^n$ **подмножеств.**

Рассмотрим еще один известный пример.

**Пример 2. «Ханойские башни».**

Даны три башни (иглы): A, B, C. Сначала на башню A нанизано несколько дисков разных размеров. Больший диск находится внизу, каждый следующий над ним меньше предыдущего.

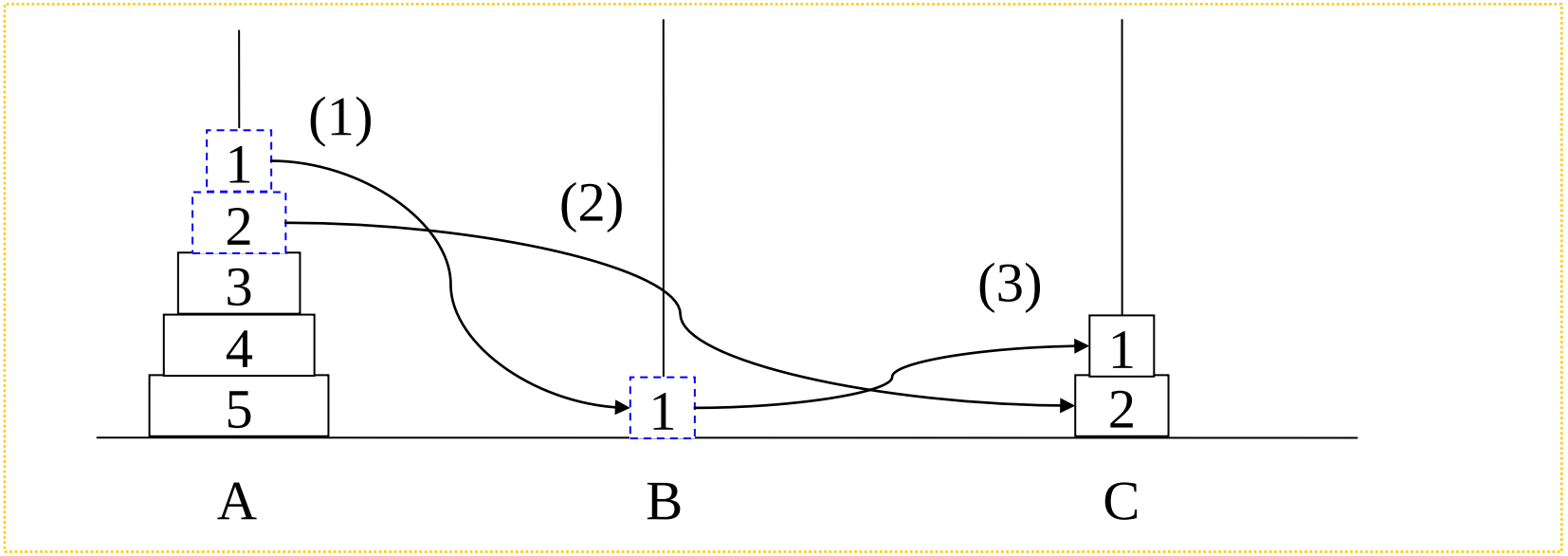


Требуется переместить все диски с башни A на башню C, башню B можно использовать в качестве промежуточной, соблюдая следующие правила:

1. За один ход перемещать можно только по одному верхнему диску с башни на башню.
2. Больший диск нельзя помещать над меньшим.

Например, если выполнить следующие три хода:

1) A → B (переместить верхний диск с A на B);
2) A → C;
3) B → C.




То за три хода переложили два диска с A на C.

Требуется решить эту задачу для произвольного n.

Здесь также будем переходить от задачи с размерностью n (n дисков) к задаче с размерностью n – 1 (n – 1 дисков).

**Алгоритм**

1) Если надо переложить один диск (n = 1), например, с A на B. Эта задача решается легко, просто его перемещаем. Будем выводить на экран информацию об этом ходе

**print('A –> B')**

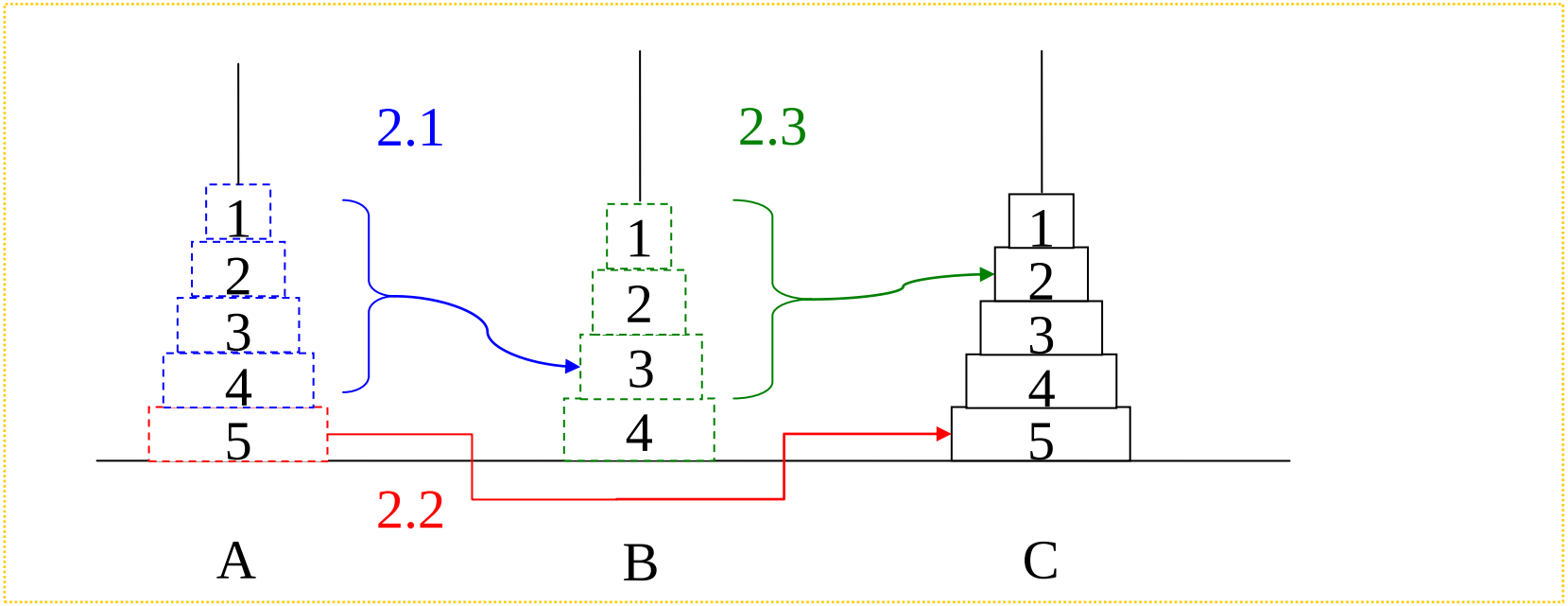
Эта запись означает, что перемещаем верхний диск с башни A наверх башни B. Здесь правда возникает вопрос: можно ли переместить с A на B? Может быть, наверху B находится меньший диск и будет нарушено условие задачи. Мы решим эту проблему при рекурсивном переходе.

2) Как переместить n дисков, например, с башни A на башню C, если меньшее количество дисков мы уже умеем перемещать? Для этого нужно сделать следующее:

2.1. Переместить (n – 1) диск с A на B;

2.2. Переместить один диск с A на C;

2.3. Переместить (n – 1) диск с B на C.



**Замечание 1.** То есть мы переходим от размерности n к размерности (n – 1). Можем делать рекурсивный переход.

**Замечание 2.** Все три хода (2.1, 2.2, 2.3) не нарушают правила перестановки:

2.1 – при перестановке (n – 1) диска можно использовать в качестве промежуточных все три столбца, так как самый крупный диск остается на месте на начальной башне, а две оставшиеся башни пустые. Это в начале перемещения. В средине процесса перемещения будет получаться, что крупные диски находятся на месте, а перемещаются более мелкие, то есть тоже все три башни могут использоваться в качестве промежуточных.

2.2 – мы перемещаем самый большой диск на пустую башню. В средине рекурсии она не пустая, но там могут быть крупные диски.

2.3 – то же самое, что и в 2.1.

То есть больший диск никогда не попадет сверху меньшего.

**Замечание 3.** Ошибочная рекурсия:

2.1. Переместить один диск с A на B;

2.2. Переместить (n – 1) диск с A на C;

2.3. Переместить один диск с B на C.

Вроде бы тоже переходим от n дисков к перемещению (n – 1)-го диска. Но здесь меньший диск может оказаться под верхним.

## Напишем реализацию этого алгоритма

```
def tower(FromT, AuxT, ToT, n):
    '''
Первые три параметра – это названия башен:
FromT – с которой переставляем;
AuxT – промежуточная башня;
ToT – на которую переставляем.
Параметр n хранит количество дисков для переста-
новки. '''
# Если остался один диск, то переместить его:
    if n == 1: print(FromT,' -> ',ToT)
    else:
        # 2.1. (n – 1) диск FromT -> AuxT
        tower(FromT, ToT, AuxT, n - 1);
        # 2.2. 1 диск FromT -> ToT
        print(FromT,' -> ',ToT)
```

```
        # 2.3. (n - 1) диск AuxT -> ToT
        tower(AuxT, FromT, ToT, n - 1);

# Начальный вызов. Переместить 5 дисков с A на C:
tower('A', 'B', 'C', 5);
```

**Для 3 дисков эта программа выведет:**

**A -> C**

**A -> B**

**C -> B**

**A -> C**

**B -> A**

**B -> C**

**A -> C**

## Задания

**1. Перепишите рекурсивную функцию, заменив условный оператор на “if n == 0 …”.**

**2. Добавьте в вывод номер перемещаемого диска, например, так: “переместить диск № 1 с A на B”.**

**3. Добавьте вывод на экран промежуточных состояний после каждого перемещения, например, так:**

**2**

**3**

**4**

**5 1**

**A B C**

**4*. Написать, не используя рекурсии.**

## Пример 3. Построение дерева игры и выбор хода

**Два игрока, Петя и Ваня, играют в следующую игру. Перед игроками лежит куча камней. Игроки ходят по очереди, первый ход делает Петя. За один ход игрок может добавить в кучу один или два камня или увеличить количество камней в куче в два раза. Игра завершается в тот момент, когда количество камней в куче становится не менее 20. Победителем считается игрок, сделавший последний ход, то есть первым получивший кучу, в которой будет 20 или больше камней.**

**В начальный момент в куче было S камней, $1 \leq S \leq 19$.**

**Возможные ходы:**

**За один ход игрок может**

**1) добавить в кучу один**

**2) или два камня**

**3) или увеличить количество камней в куче в два раза.**

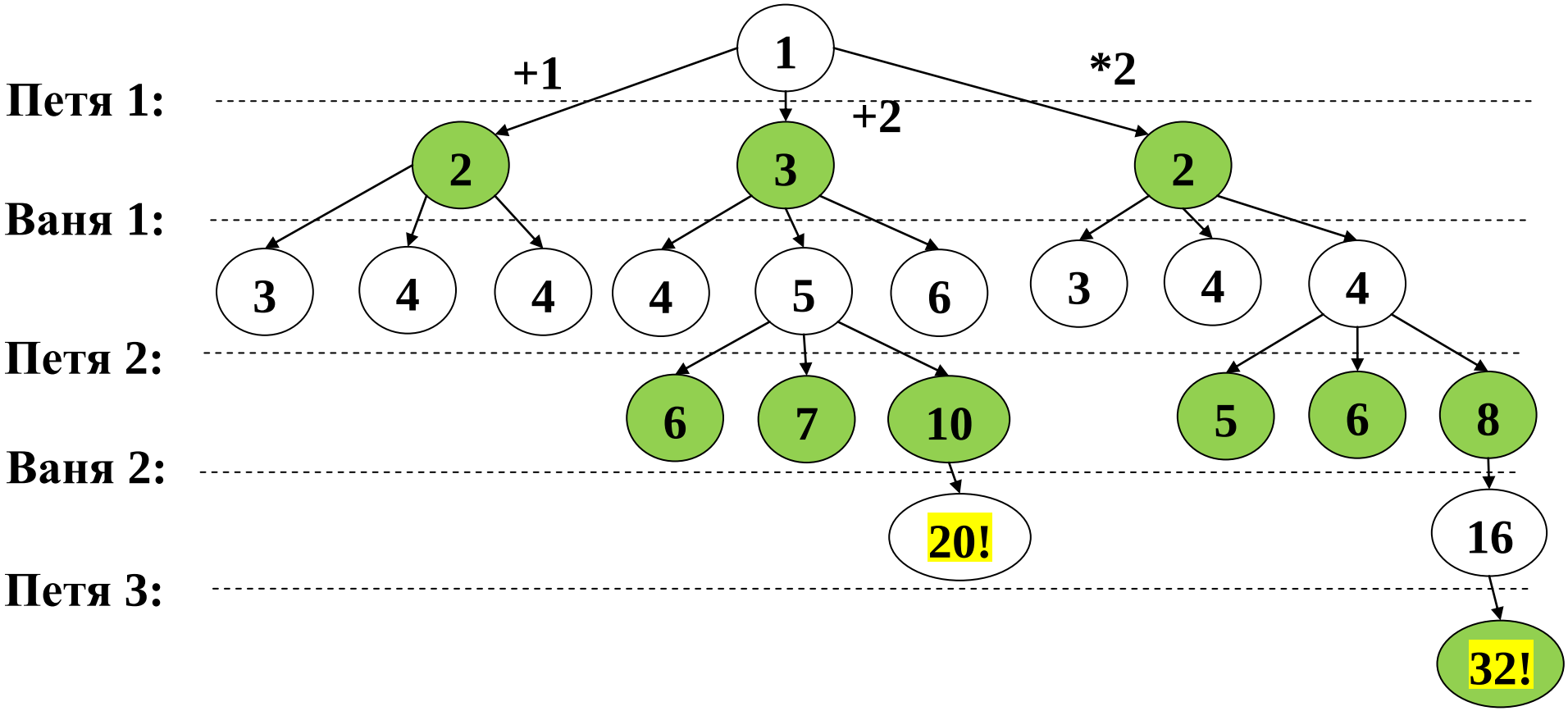
**Напишем вспомогательную функцию для возможных ходов:**

```
def hod(x):
    #  возможные ходы:
    return [x + 1, x + 2, x * 2]
```

**Будем использовать две глобальные переменные:**

```
S_Start = 1    #  – начальное количество камней
S_Win = 20     #  – конечное количество камней
```

# Дерево игры


Петя 1:
Ваня 1:
Петя 2:
Ваня 2:
Петя 3:
1
+1
+2
*2
2
3
2
3
4
4
4
5
6
3
4
4
6
7
10
5
6
8
20!
16
32!

**В функцию будем передавать начальное значение камней s. Функция строит дерево игры и возвращает: 1, если есть выигрышная стратегия или (–1) в случае поражения.**

**Рекурсивный переход от всей задачи к подзадаче**

**1. Если игрок выигрывает первым ходом, т.е. у него есть выигрышная стратегия, функция возвращает 1 (конечная ветвь).**

**2. Если нет выигрыша первым ходом, то делаем все возможные ходы и передаем ход сопернику. У результата, возвращаемого соперником, меняем знак: если соперник возвращает 1 (его выигрыш), то меняем знак –1 (наш проигрыш). После ищем максимальное значение из всех возващенных им и выдаем результат (рекурсивная ветвь).**

## Основная рекурсивная функция

```
# s - текущее количество камней
def next_step(s):
# Делаем все возможные ходы игрока из s:
    st = hod(s)
# Если хотя бы один ход выигрывает, то
    if max(st) >= S_Win:
        return 1 # возвращаем (1) - победа
# Иначе, ход соперника:
# Продолжаем дерево для каждого случая камней:
    res = [ - next_step(i) for i in st]
# Минус победу соперника 1 переводит в -1
# Возвращаем лучший для игрока результат:
    return max(res)
```

**Вся программа:**

```
S_Start = 1
S_Win = 20
def hod(x): return [x + 1, x + 2, x*2]

def next_step(s):
    # Ход игрока:
    st = hod(s)
    if max(st) >= S_Win:
        return 1
    # Ход соперника (res = -1, 0 или 1):
    res = [ - next_step(i) for i in st]
    return max(res)

# Начальный вызов:
print(next_step(S_Start))
```

**Замечание 1.** **Функция делает полный перебор и возвращает 1 или –1. Т. е. пока только сообщает, есть ли выигрышная стратегия или нет. К ней можно добавить и возврат лучшего хода (т. е. st, для которого res максимально). Ее тогда можно использовать для написания игры, в которой она будет давать оптимальные ходы для игрока.**

**Замечание 2.** **В дереве часто повторяются числа. Данная рекурсивная функция каждый раз будет считать одно и то же в разных ветвях. Можно сохранять результаты для s и в других ветках рекурсии их использовать. Это сократит рекурсивный обход (см. следующий пример).**

**Замечание 3.** **Также, вместо рекурсивной функции, можно использовать динамическое программирование, и написать перебор s, начиная с 19 до 1.**

**Будем записывать результаты для разных камней s в таблицу. И в других ветках дерева будем использовать эти данные:**

```
tableS = [0] * S_Win   # заполнили таблицу 0-ми
def next_step(s):
    # Ход игрока:
    st = hod(s)
    if max(st) >= S_Win:
        tableS[s] = 1      # - заполнение таблицы
        return 1
    # Ход соперника (используем таблицу):
    res = [ - tableS[i] if tableS[i] else
            - next_step(i) for i in st]
    if tableS[s] == 0: tableS[s] = max(res)
    return max(res)
```

**[0, 1, –1, 1, 1, 1, –1, 1, 1, –1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0]**

# Игра

```
import random # Подключаем модуль random
random.seed( )
# Возможные ходы игроков:
def hod(x):
    return [x + 1, x + 2, x*2]

def next_step(s):
    # Ход игрока:
    st = hod(s)
    if max(st) >= S_Win:
        tableS[s] = 1
        return 1, st.index(max(st))
    # Ход соперника:
    res = [ - tableS[i] if tableS[i] else
```

```
                - next_step(i)[0] for i in st]
    if tableS[s] == 0: tableS[s] = max(res)
    return max(res), res.index(max(res))

# Игра:
S_Start = random.randint(1, 9)
S_Win = random.randint(20, 30)
print('Начальное количество камней %d,' % S_Start,
      'победа при s >=%d.' % S_Win)
# В цикле игроки ходят по очереди:
while True:
    # Ход игрока 1:
    # (1) Выбор хода игроком:
    print('Ход игрока 1: (1) +1; (2) +2; (3) *2.')
    s = input('>')
```

```
n = 3
while s not in ['1', '2', '3'] and n > 0:
    print('Нет такого хода. Введите еще раз')
    s = input('>')
    n -= 1
if s not in ['1', '2', '3']:
    print('Конец игры')
    break
s = int(s)
# (2) Выполнить ход:
S_Start = hod(S_Start)[s - 1]
if S_Start >= S_Win:
    print('Победа игрока 1')
    break
print('Количество камней %d.' % S_Start)
```

```
# Ход игрока 2:
# (1) Выбор оптимального хода:
tableS = [0] * S_Win
res = next_step(S_Start)
# (2) Выполнить ход:
print('Ход игрока 2:',res[1]+1,'Оценка',res[0])
S_Start = hod(S_Start)[res[1]]
print('Количество камней %d.' % S_Start)
if S_Start >= S_Win:
    print('Победа игрока 2')
    break
```

## Пример 4. Алгоритм с возвратом

В этом примере наша программа будет искать решение поставленной задачи методом проб и ошибок. Процесс можно представить как поиск – исследование, в котором постепенно строится решение, перебираются все возможные варианты, отбрасываются неверные. Этот общий прием мы рассмотрим на примере следующей задачи.

**Пример 4. «Задача о путешествии шахматного коня».**

Пусть дана доска $n \times n$ с $n^2$ полями. Конь, который передвигается по шахматным правилам, вначале ставится на любую клетку $(x_0, y_0)$. Задача – обойти всю доску, если это возможно. Конь должен побывать на каждой клетке ровно один раз.

**Например, при n = 4 (размер доски $4 \times 4$).**

| x \ y | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | | | |
| 2 | | | 2 | |
| 3 | 3 | 2’ | | |
| 4 | | 3’ | | 3’’ |

**Нумеровать клетки будем как в матрице. Если вначале поставить коня в левый верхний угол $(x_0 = 1, y_0 = 1)$, то из него можно сделать ход (2) или (2’). После хода (2) есть три варианта продолжения, отмеченные на рисунке (3), (3’) и (3’’). И так далее можно перебрать все возможные ходы и выяснить приведет ли какая-нибудь их последовательность к полному заполнению доски или нет.**

**В этой задаче нам нужно обойти $n^2$ полей. Эту общую задачу мы сведем к подзадаче, которая состоит в том, чтобы либо выполнить какой-нибудь очередной ход, либо обнаружить, что очередной ход невозможен. Примерная схема функции будет такой:**

```
def TryNextMove( ): # попытка сделать ход
  if <доска заполнена>: <задача решена>
  else:
    <начать перебор возможных ходов
    и выбрать первый>
    while <есть очередной ход> and
      <сделать ход и посмотреть, приведет ли
      он к результату – если нет>:
       <выбрать следующий допустимый ход>
```

**Условие внутри цикла «сделать очередной ход и посмотреть, может быть, он в дальнейшем приведет к результату» означает следующее:**

1) **Записать очередной ход.**
2) **Посмотреть, приведут ли дальнейшие ходы за ним к нужному результату – заполнению всей доски.**
3) **Если да, то вернуть результат «да», если нет, то сделать отмену этого хода и вернуть результат «нет».**

**Это удобно оформить в виде функции с булевским результатом.**

**Примерная схема функции такая:**

```
# попытаться завершить текущий ход:
def CanBeDone( ):
begin
   <записать ход>
# приведут ли последующие ходы к результату:
   TryNextMove( )
   if <не удалось достроить>:
         <отменить запись хода>
   return <удалось достроить или нет>
```

Это общая схема алгоритма с возвратом. Вначале запускается функция TryNextMove для пустой доски, заполнена только одна начальная клетка $(x_0, y_0)$. Внутри нее запускаются по очереди все возможные первые ходы из точки $(x_0, y_0)$ – функция CanBeDone. Очередной первый ход будет запущен, если CanBeDone вернет False, т.е. предыдущий ход не привел в дальнейшем к заполнению всего поля.

Функция CanBeDone запоминает один из первых ходов и пытается его продолжить, сделать второй ход: запускает TryNextMove, чтобы эта процедура перебрала все возможные вторые ходы, после выбранного первого. Если какой-то из них приведет к заполнению всего поля, то TryNextMove вернет внутрь CanBeDone значение True, иначе – False. По этому ре-

зультату CanBeDone будет решать отменять ей первый ход или нет.

Здесь в общей схеме видно, что перебираться будут все возможные ходы до того момента, пока не найдется правильная последовательность ходов. Вначале строится одно решение: делаются первые попавшиеся ходы №1, №2, №3 и т.д. (TryNextMove выбирает первый попавшийся ход и вызывает CanBeDone, а та в свою очередь его фиксирует и вызывает обратно TryNextMove для следующего хода). Если первые попавшиеся ходы не приведут к результату, то в какой-то момент оказывается, что нельзя сделать очередной ход, и доска еще не заполнена. Тогда происходит откат рекурсии. Отмена ходов идет с последних: мы не можем сделать k-й ход, тогда отменяем (k–1)-й и делаем другой (k–1)-й. Так происходит перебор всех ходов.

**Замечание 1.** Здесь применяется косвенная рекурсия.

**Замечание 2.** Видно, что с ростом размера доски n резко возрастает количество возможных ходов ($e^{n^2}$). Для n = 5 и n = 6 за несколько секунд делает полный обход. Для n = 7 и n = 8 выдает результат примерно за минуту, но благадаря тому, что несколько первых ходов не отменяются.

**Для того чтобы написать программу полностью надо решить несколько задач:**

**1.Как хранить данные о сделанных ходах?**

**2. Какие параметры передавать в функциях?**

**3. Как определять, что все поля заполнены?**

**4. Как определить все возвожные ходы?**

**1. Данные о ходах можно хранить в двумерном списке n×n. Элемент списка h[x, y] будет хранить данные о клетке (x, y). Вначале список будет состоять только из нулей. Далее в клетку будем ставить номер хода ( > 0 ). В конце этот список будет хранить ответ с порядком обхода клеток.**

**2. Какие параметры передавать в функциях?**

**Функция TryNextMove должна иметь данные о сделанном текущем ходе: x, y – координаты; i – номер хода. В конце сообщить True/False об его успешности.**

**Функция CanBeDone аналогично: u, v – координаты; i – номер хода. В конце сообщить True/False об его успешности.**

**3. Как определять, что все поля заполнены?**

**Условие окончания тогда i == n ** 2 – сделали все ходы и следовательно заполнили доску.**

**4. Как определить все возвожные ходы?**

**Из текущей позиции конь может сделать максимум 8 ходов.**

| | | | | | |
|---|---|---|---|---|---|
| | 6 | | 7 | | |
| 5 | | | | 8 | |
| | | | | | |
| 4 | | | | 1 | |
| | 3 | | 2 | | |
| | | | | | |

**Их надо еще проверять на то, что они попадают внутрь доски: 0 ≤ u < n, 0 ≤ v < n.**

**И что они еще не заполнены: h[u, v] == 0.**

**Начальное заполнение данных и вызов рекурсии:**

```
# вспомогательные списки:
dx = [2, 1, -1, -2, -2, -1, 1, 2]
dy = [1, 2, 2, 1, -1, -2, -2, -1]
n = 8    # выбор размера доски
h = [ [0]*n for i in range(n)] # заполнение 0-ми
x0 = 0; y0 = 0    # начальная клетка
h[x0][y0] = 1 # отмечаем начальную клетку
# Старт рекурсии:
done = TryNextMove(x0, y0, 1)
# Вывод результата:
print(done)
if done:
    for i in range(n):
        for j in range(n):
            print('%2d' % h[i][j], end = ' ')
        print( )
```

## Функция «попытаться сделать следующий ход»

```
def TryNextMove(x, y, i):
    # Функция перебора возможных ходов:
    def next(k):
      k += 1  # k - номер хода от 1 до 8
      if k >= 8: return True, 0, 0, k
      u = x + dx[k]; v = y + dy[k]
  # Проверка - внутри доски и поле пустое:
      while not (0 <= u < n and 0 <= v < n
                 and h[u][v] == 0):
         k += 1
         if k >= 8: return True, 0, 0, k
         u = x + dx[k]; v = y + dy[k]
      # Если нашли возможный ход, возвращаем:
      # False; (u, v) - координаты; k - номер
      return False, u, v, k
```

```
# начало функции:
if i >= n ** 2: # если доска заполнена
    return True  # значит задача решена
else:
# выбрать первый ход:
    eos, u, v, k = next(-1)
# eos = True - невозможен следующий ход
# u, v - координаты следующего хода
# k - номер возможного из 8 ходов
while not eos and not CanBeDone(u,v,i+1):
    # выбрать следующий допустимый ход:
    eos, u, v, k = next(k)
# Результат True, если удалось достроить:
return not eos
```

## Функция «можно ли продолжить»

```
def CanBeDone(u, v, i):
    # записать ход:
    h[u][v] = i
    # делаем следующий ход:
    done = TryNextMove(u, v, i)
    # TryNextMove вернет True, если заполнит
    # всю доску после хода (u, v).
    # Если не удалось заполнить:
    if not done:
        h[u][v] = 0 # отменяем ход
    # Вернуть результат:
    return done
```

## Результаты работы программы:

```
n = 5
True
1   6 15 10 21
14  9 20  5 16
19  2  7 22 11
8  13 24 17  4
25 18  3 12 23

n = 6
True
 1 16  7 26 11 14
34 25 12 15  6 27
17  2 33  8 13 10
32 35 24 21 28  5
23 18  3 30  9 20
36 31 22 19  4 29
```

```
n = 7    True
 1 38 31  8 19 36 15
32 29 20 37 16  7 18
39  2 33 30  9 14 35
28 25 40 21 34 17  6
41 22  3 26 45 10 13
24 27 48 43 12  5 46
49 42 23 4  47 44 11

n = 8     True
 1 60 39 34 31 18  9 64
38 35 32 61 10 63 30 17
59  2 37 40 33 28 19  8
36 49 42 27 62 11 16 29
43 58  3 50 41 24  7 20
48 51 46 55 26 21 12 15
57 44 53  4 23 14 25  6
52 47 56 45 54  5 22 13
```

## Задания

**1. Как сделать так, чтобы программа выдавала все возможные решения, а не одно?**

**2*. Решить похожую задачу: расположить на шахматной доске 8 ферзей так, чтобы ни один из них не угрожал другому.**

**3**. На основе двух последних примеров написать игру «крестики – нолики».**