

# Рекурсия. Перебор

Лектор:  
Михно Марк

# Что такое рекурсия?

- Рекурсия — это прием программирования, при котором решение задачи сводится к некоторым действиям плюс решение такой же задачи, но в более простом случае. Под более простым подразумевается либо случай с меньшими входными данными, либо с меньшим их количеством.

# Пример

Нахождение факториала натурального числа  $n$ .

По определению :  $n! = 1 * 2 * \dots * n$

Видоизменим равенство :  $n! = (1 * 2 * \dots * n - 1) * n$

Таким образом,  $n! = (n - 1) ! * n$

В словесной формулировке результаты преобразований будут звучать так : чтобы найти факториал числа  $n$ , надо найти факториал числа  $n - 1$ , а затем домножить его на число  $n$ .

Для описания рекурсивного решения необходимы две четко определенные вещи :

- *Правило*, по которому решение задачи в сложном случае сводится к решению такой же задачи, но в более простом случае (рекурсивный переход).
- *Условие*, при котором дальнейшее упрощение нужно прекратить (терминальное условие). При отсутствии этого условия процесс упрощения будет продолжаться до бесконечности.

С точки зрения программирования рекурсивное решение записывается в виде функции, содержащей вызов самой себя. Для примера с факториалом возможна, к примеру, такая реализация:

```
int f(int n)
{
```

```
    if(n == 0 || n == 1)
        return 1;
```

Терминальное условие

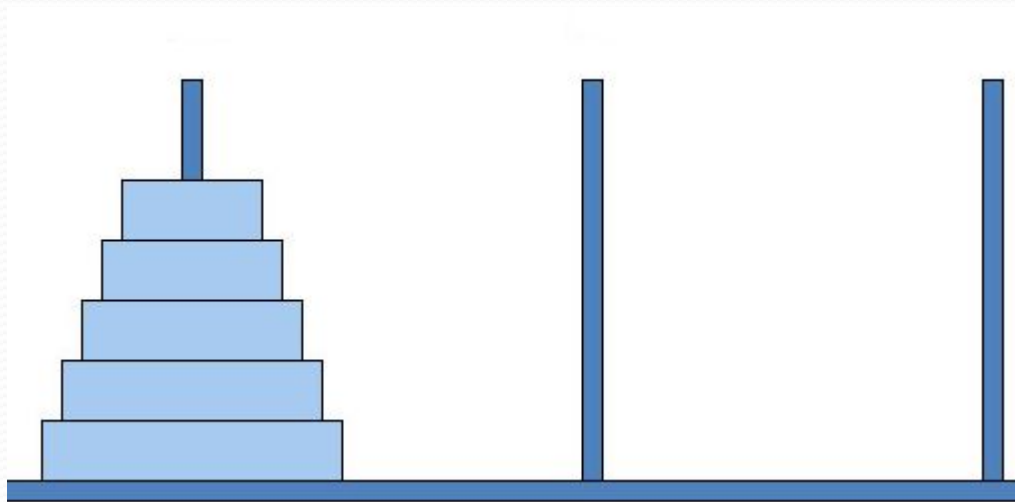
```
    return n * f(n - 1);
```

Рекурсивный переход.

```
}
```

# Ханойские башни

- Дано три стержня. В начальный момент времени на первый нанизано  $N$  колец различного диаметра так, что они образуют пирамидку :



Требуется перенести все кольца с первого стержня на третий за минимальное количество ходов, соблюдая два правила:

- Можно брать только свободное кольцо (то, на котором ничего не лежит).
- Взятое кольцо можно нанизывать на любой стержень, но нельзя класть большее кольцо на меньшее.

Для  $N$ , равного 1 или 2, решения очевидны.

При  $N = 3$  можно сделать следующее замечание :  
Пока верхние два кольца не будут перемещены на другой стержень, с нижним кольцом по правилам ничего сделать нельзя.

Поэтому можно временно про нижнее кольцо «позабыть» и решать только задачу переноса двух верхних колец на второй стержень. После этого «освободившееся» нижнее кольцо можно перенести на третий стержень, а затем опять вернуться к задаче о перемещении первых двух колец. Таким образом задача для  $N=3$  сводится к двум задачам для  $N=2$ .




Из этих рассуждений можно вывести схему упрощения задачи и сведения ее решения к более простому случаю :

- Чтобы перенести  $n$  колец с одного стержня на другой, необходимо для начала  $n - 1$  кольцо поместить на третий стержень, потом перенести самое большое кольцо на нужный стержень и снова переместить  $n - 1$  кольцо.

Можно написать процедуру `moveTo`, которая будет переносить `N` колец со стержня с номером **from** на стержень **to**.

```
void moveTo(int from, int to, int N)
{
    if(N == 1)
    {
        cout << "move from " << from << " to " << to << '\n';
    }else
    {
        int temp = 6 - from - to;
        moveTo(from, temp, N - 1);
        cout << "move from " << from << " to " << to << '\n';
        moveTo(temp, to, N - 1);
    }
}
```



Основной идеей рекурсивного решения является «вера» в то, что внутренняя функция успешно справится с решением своей более простой задачи. А это вполне возможно, так как внутренняя функция может быть либо последней в цепочке рекурсии (тогда она выдает простой ответ), либо от нее цепочка будет тянуться дальше, тогда все зависит от правильности рекурсивного соотношения.


# Рекурсивный перебор

Перебором мы будем называть в первую очередь перебор некоторых, скажем так, комбинаторных объектов.

Основная цель перебора—перебрать все объекты из некоторого множества, дабы что-то сделать с каждым.

Наиболее часто, встречаются три варианта :

- либо надо найти объект (любой), удовлетворяющий некоторому условию.
- либо посчитать количество таких объектов,.
- либо найти в некотором смысле оптимальный объект (дающий минимальную стоимость и т.п.)



Конечно, перебор можно писать по-разному. Но наиболее общим и в большинстве случаев довольно простым является рекурсивный перебор, также называемый перебором с возвратом. Помимо более простой реализации, он обладает рядом других достоинств: например, в нем возможно очень легко реализовывать различные отсечения и эвристики.

Давайте научимся решать следующую задачу :

Дан массив из  $N$  различных чисел. Мы хотим узнать количество различных способов выбрать некоторые числа из этого массива так, чтобы их сумма равнялась заданному числу  $K$ .

Задача имеет множество решений, но давайте попробуем применить технику перебора – то есть просто попробуем перебрать все возможные подмножества чисел, затем найдем их сумму и сравним с  $K$ .

Как перебирать все подмножества массива?

Заметим, что все подмножества делятся на два типа :

- Те, в которых есть элемент массива с номером 1.
- Те, в которых его нет.

Давайте зафиксируем состояние первого элемента (выберем, будет ли он в нашем подмножестве), и переберем все возможные подмножества оставшихся  $N - 1$  элементов массива.

Это и будет наш рекурсивный переход : вместо того, чтобы решать задачу о переборе всех подмножеств  $N$ -элементного множества, мы фиксируем первый элемент и перебираем все подмножества  $N - 1$  - элементного множества.

Логика процедуры **find** следующая:  
Пускай мы уже перебрали для первых  $m - 1$  элементов, будут ли они входить в наше подмножество. Тогда **find** переберет все возможные подмножества  $N - m + 1$  оставшихся элементов массива (здесь  $m$  обозначает номер первого неопределенного элемента).

Заведем вспомогательный массив **used**, который для каждого элемента хранит, будет ли он находиться в нашем подмножестве.

Фактически, задача состоит в переборе всех возможных состояний массива **used**.

```
void find(int m)
{
    if (m > n)
    {
        int sum = 0;

        for (int i = 1; i <= n; ++i)
            if (used[i])
                sum += a[i];

        if (k == sum)
            ++answer;

        return;
    }

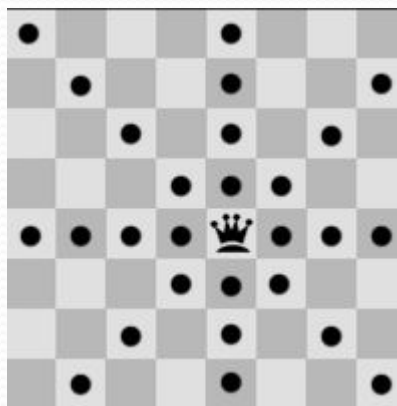
    used[m] = 0;
    find(m + 1);

    used[m] = 1;
    find(m + 1);
}
```



# Задача о расстановке ферзей

Возьмем обычную шахматную доску и попробуем расставить на ней 8 ферзей так, чтобы никакая пара ферзей не била друга. Оказывается, сделать это не так просто, так как ферзь «бьет» огромное количество клеточек:





Давайте попробуем перебрать все возможные расстановки восьми ферзей рекурсивным перебором.

Воспользуемся следующим переходом: выберем, в какую клеточку поставить первого ферзя, а потом переберем все возможные расстановки семи ферзей в оставшиеся клетки. Напишем следующую процедуру.

```
void find(int m)
{
    if(m > 8)
    {
        for(int i = 1; i < 8; ++i)
            for(int j = i + 1; j <= 8; ++j)
                if(bad(answer[i], answer[j]))
                    return;

        cout << "Answer is found!\n";

        return;
    }

    for(int i = 1; i <= 8; ++i)
        for(int j = 1; j <= 8; ++j)
            if(used[i][j] == false)
            {
                used[i][j] = true;
                answer[m] = make_pair(i, j);

                find(m + 1);

                used[i][j] = false; /// не забудьте "убрать" ферзя
            }
}
```

Попытаемся оценить время работы. Мы выбираем, куда поставим первого ферзя – 64 варианта, потом второго – 63 варианта, и так далее.

Получаем  $64 * 63 * .. * 58$  вариантов, что, конечно, слишком большое число. Решение необходимо оптимизировать.

Давайте, например, при переборе не пытаться ставить ферзя в клеточку, которая уже находится под боем.

Оказывается, этого небольшого отсеечения ненужных вариантов более чем хватает для быстрого решения задачи.

Вот так изменится процедура find:

```
for(int i = 1; i <= 8; ++i)
    for(int j = 1; j <= 8; ++j)
        if(used[i][j] == false)
        {
            answer[m] = make_pair(i, j);

            bool OK = true;

            for(int k = 1; k < m; ++k)
                if(bad(answer[k], answer[m]))
                {
                    OK = false;
                    break;
                }

            if(OK)
            {
                used[i][j] = true;
                find(m + 1);
                used[i][j] = false;
            }
        }
}
```

Прием, который позволяет уменьшить количество рассматриваемых вариантов в переборе, называется отсечением. Существует огромное количество разных по сложности и эффективности отсечений и эвристик.


Вот самые популярные из них:

- Отсечение по ответу (откидывание заведомо ненужных вариантов)
- Отсечение по времени.
- Оптимальный порядок перебора.
- Жадных поиск каких-то решений еще до запуска перебора.

Рекурсия очень требовательная к ресурсам компьютера, и писать её нужно аккуратно.

Все что можно закодить рекурсией, можно в теории закодить итеративно (и наоборот).

Если вы можете без особых проблем написать итеративное решение задачи, то, скорее всего, оно будет работать лучше рекурсивного.



Конец  
спасибо