

Севастопольский государственный университет  
Кафедра «Информационные системы»

Курс лекций по дисциплине

**«АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ»**  
(АиТ)

Лектор: Сметанина Татьяна Ивановна

# Лекция 3

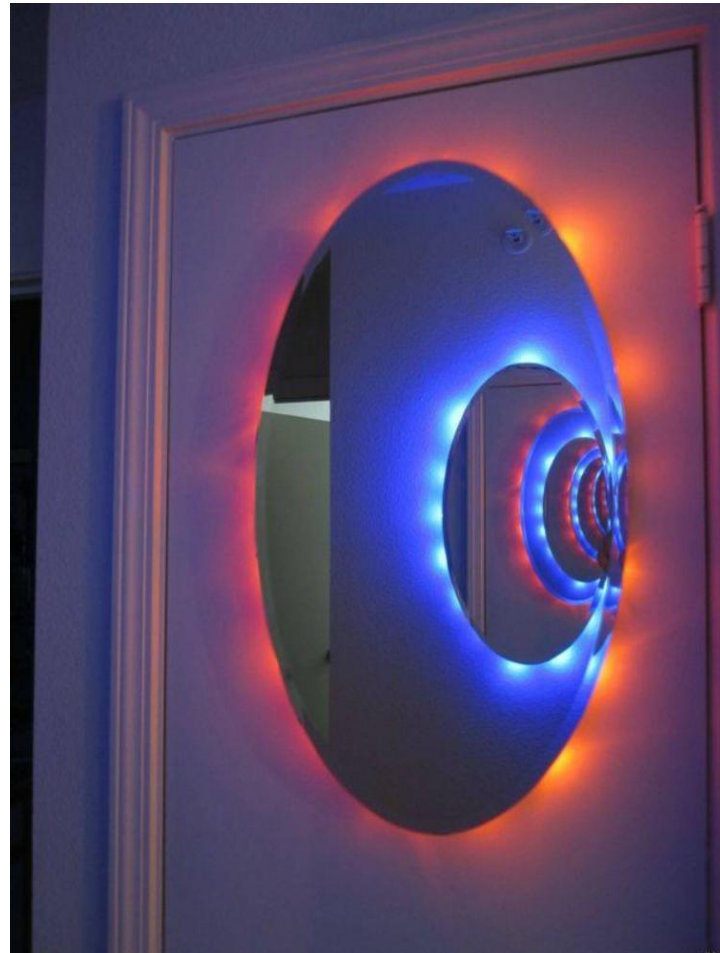
## Рекурсивные функции

# Рекурсивные функции

*Рекурсивной* называется функция, в тексте которой имеется обращение к самой себе.

Существует множество видов рекурсии, рассмотрим следующие:

- прямую;
- косвенную;
- параллельную.



# Рекурсивные функции

Рекурсию можно назвать *прямой*, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще *рекурсией первого порядка, простой или линейной*.

Рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

- *параллельная рекурсия* - если функция содержит несколько вызовов самой себя (без вложения);
- *взаимная рекурсия (косвенная)* – если две и более функции содержат вызов друг друга.
- *рекурсия более высокого порядка* – если в параметрах рекурсивной функции присутствуют вложенные вызовы самой себя.

# Рекурсивные функции

Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции. При повторном вызове этот процесс повторяется.

Для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

# Рекурсивные функции

Стандартом языка C++ рекурсивно может вызываться любая функция, кроме **main()**, а по стандарту языка C рекурсивно можно вызвать даже **main()**!

Пример прямой рекурсии функции **main()** на языке C++:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world!" << endl;
    main();
    return 0;
}
```

Так делать не нужно

# Рекурсивные функции

Пример косвенной рекурсии функции `main()` на языке C++:

```
#include <iostream>
using namespace std;
void f();
int main() {
    cout << "Hello world!" << endl;
    f();
    return 0;
}
void f() {
    main();
}
```

Так делать не нужно

В обоих примерах на экране монитора достаточно долго будет выводиться радостный текст

Hello world!

а затем произойдёт аварийное завершение программы из-за переполнения стека.

# Рекурсивные функции

Независимо от того, какой вид рекурсии используется, в рекурсивной функции ОБЯЗАТЕЛЬНО должна выполняться *проверка* какого-либо условия, в зависимости от которого осуществлялся бы *выход из рекурсивной функции*. Причём, необходимо гарантировать, что это условие обязательно будет выполнено через конечное количество рекурсивных вызовов, и функция в конце концов закончит работу. Иначе — аварийное завершение, смотри примеры выше.

Последовательный вызов функцией самой себя называют *рекурсивным спуском*, последовательный выход из многократного вызова — *рекурсивным подъёмом*.

При каждом рекурсивном вызове функции заново выделяется память под локальные переменные.



# Рекурсивные функции

Рассмотрим вычисление факториала целого числа

$$n! = 1 * 2 * 3 \dots * (n-1) * n$$

```
factorial=1;  
  for (i=2; i<n; i++)  
    factorial=factorial*i;
```

Здесь сначала вычисляется  $1! = 1$ , потом  $2! = 1! * 2$ , потом  $3! = 2! * 3$  и т.д. Таким образом вычисление факториала сводится к многократному применению формулы  $n! = (n-1)! * n$ .

Эта формула является рекурсивной, т.к. определяет факториал, через факториал. Однако способ вычислений не является рекурсивным и представлен **восходящим вычислительным процессом**, т.е. вычисляется  $1!$ , потом  $2!$  и т.д.

# Прямая рекурсия

Классическим примером рекурсивной функции является вычисление факториала (это не означает, что факториал следует вычислять именно так). Для того, чтобы получить значение факториала числа  $n$ , требуется умножить на  $n$  факториал числа  $(n-1)$ . Известно также, что  $0! = 1$  и  $1! = 1$ .

Рекурсивное вариант можно получить, используя **нисходящий вычислительный процесс**, т.е. «вычислим  $(n-1)!$  и полученное значение умножим на  $n$ ». При вычислении  $(n-1)!$  применим тоже правило «вычислим  $(n-2)!$  и полученное значение умножим на  $n-1$ » и т.д.

**Условие завершения рекурсии**  $n=0$  и факториал  $0! = 1$ .

# Прямая рекурсия

```
#include <iostream>
using namespace std;

long factorial (long n) {
    if (n==0) return 1;
    return (n * factorial(n-1)); // рекурсивный вызов ф-ции
}

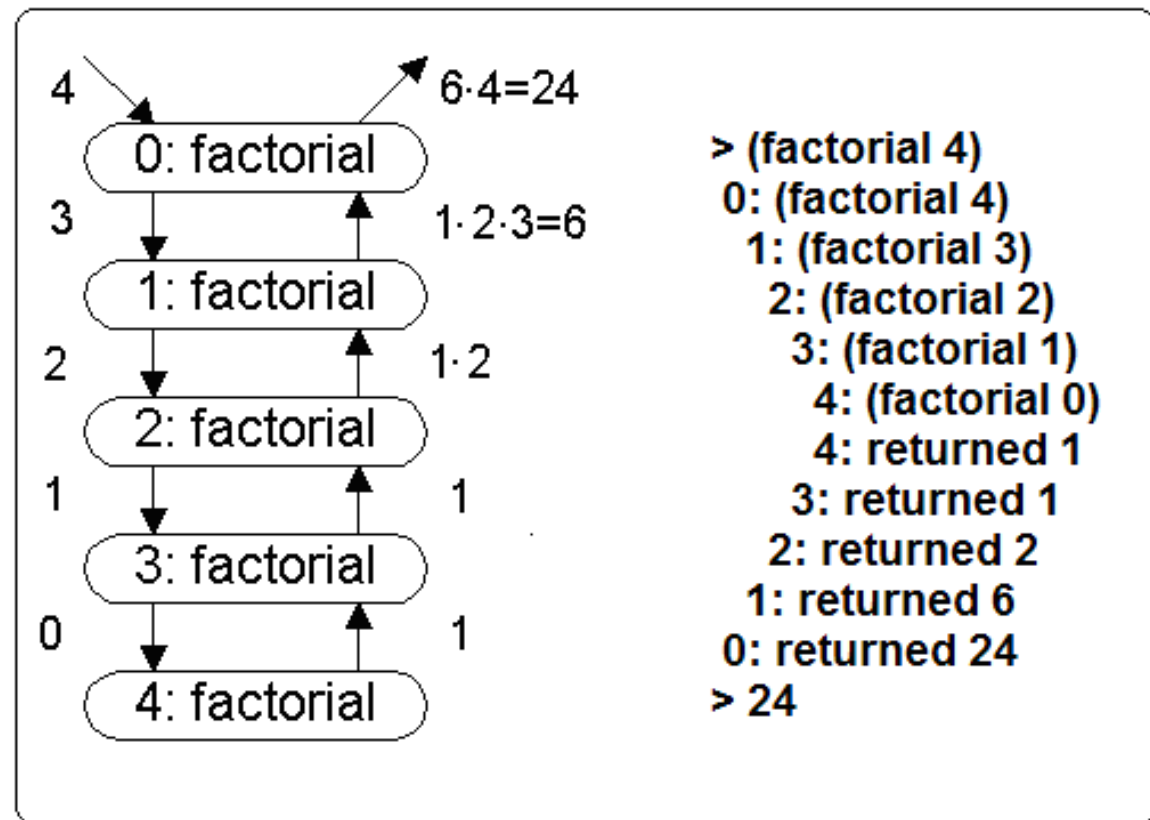
int main() {
    cout<<factorial(4);
    return 0;
}

/* long factorial (long n) { // версия № 2
    return (n>1) ? n * factorial(n-1) : 1;
}*/
```

Работу функции рассмотрим на примере вычисления **4!**

# Прямая рекурсия

```
long factorial(long n) {  
    if (n==0) return 1;  
    return (n * factorial(n-1)); // рекурсивный вызов ф-ции  
}  
  
int main() {  
    cout<<factorial(4);  
    return 0;  
}
```



# Прямая рекурсия

```
long factorial(long n) {  
    return (n>1) ? n * factorial(n-1) : 1;  
}
```

```
factorial(4)  
4 * factorial(3)  
4 * 3 * factorial(2)  
4 * 3 * 2 * factorial(1)  
4 * 3 * 2 * 1  
4 * 3 * 2  
4 * 6  
4 * 24
```

В рекурсивных программах можно выделить два процесса: **рекурсивное погружение** подпрограммы в себя; **рекурсивное возвращение** из подпрограммы. После вызова подпрограммы с аргументом **n** выполняется еще **n-1** вызов и общее количество незавершенных вызовов достигает **n**.

Величина, которая характеризует максимальное количество незавершенных вызовов, называется **глубиной рекурсии**.

# Прямая рекурсия

От глубины рекурсии зависит время выполнения программы и объём требуемой памяти.

Если в теле подпрограммы выполняется более одного рекурсивного вызова, то рекурсивный вычислительный процесс может оказаться **неэффективным**. В этом случае следует предпочесть решение задачи с помощью итерационных циклов. Пример неэффективной рекурсии – вычисление чисел Фибоначчи.

Рассмотрим еще один пример прямой рекурсии:

# Прямая рекурсия

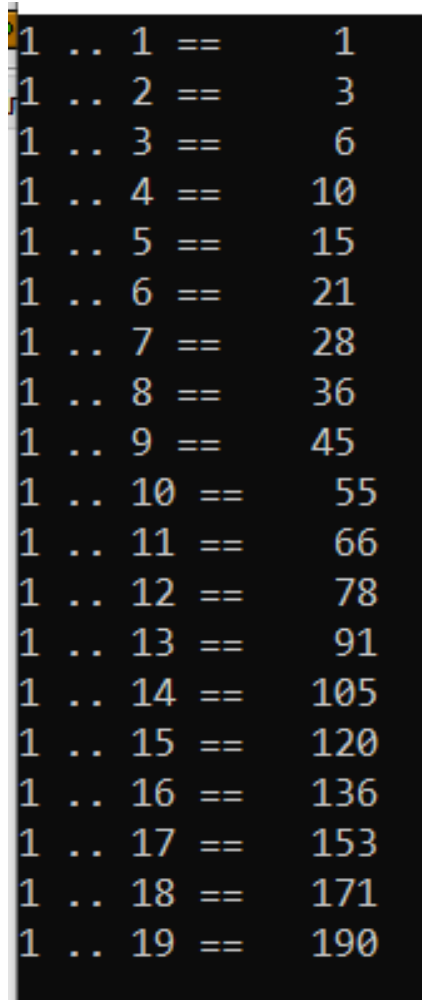
```
#include <stdio.h>
#include <conio.h>

void sum(int value, int result=0) {
    if (value) {
        result = value + result;
        sum(value-1, result); //рекурсивный вызов
    } else { //если достигнута контрольная точка,
        //выводим результат на экран
        printf(" %5d ", result);
    }
}

int main() {
    for (int i=1; i<20; i++) {
        printf("1 .. %d ==", i);
        sum(i); //вызов функции sum
        printf("\n");
    }
    getch();
} // Какие действия выполняет программа?
```

# Прямая рекурсия

Программа суммирует числа от 1 до N

A screenshot of a terminal window with a black background and yellow text. It displays a series of 19 lines, each representing a recursive call to a function that sums numbers from 1 to N. The format of each line is 'N .. 1 == 1', 'N .. 2 == 3', etc., where N increases from 1 to 19. The terminal has a vertical scrollbar on the left side.

1	..	1	==	1
1	..	2	==	3
1	..	3	==	6
1	..	4	==	10
1	..	5	==	15
1	..	6	==	21
1	..	7	==	28
1	..	8	==	36
1	..	9	==	45
1	..	10	==	55
1	..	11	==	66
1	..	12	==	78
1	..	13	==	91
1	..	14	==	105
1	..	15	==	120
1	..	16	==	136
1	..	17	==	153
1	..	18	==	171
1	..	19	==	190



# Косвенная рекурсия

*Косвенная рекурсия (взаимная)*, когда две или более функций вызывают друг друга. Рассмотрим на примере:

```
#include <stdio.h>
```

```
void two(int n);
```

```
void one(int n) {  
    if (n>0) {  
        printf("Hello");  
        two(n);  
    }  
}
```


```
void two(int n) {  
    printf(" world \n");  
    one(n-1);  
}
```

```
int main() {  
    one(4);  
    return 0;
```

```
} // Какие действия выполняет программа?
```

# Косвенная рекурсия

Результат работы программы:



```
hello world  
hello world  
hello world  
hello world
```

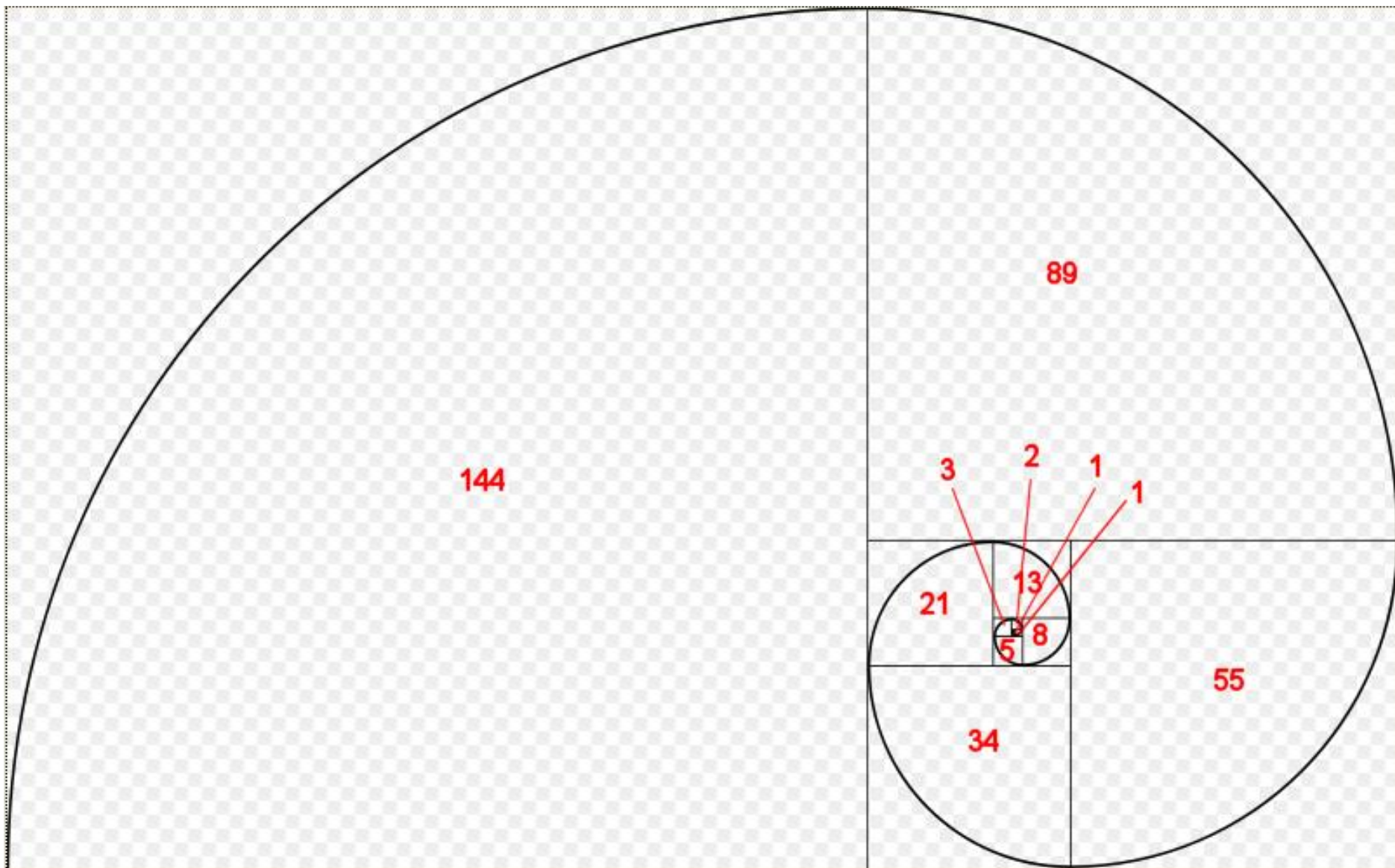
# Числа Фибоначчи

Одним из наиболее известных математических рекурсивных алгоритмов является последовательность Фибоначчи. Последовательность Фибоначчи можно увидеть даже в природе: ветвление деревьев, спираль ракушек, плоды ананаса, разворачивающийся папоротник и т.д.



# Числа Фибоначчи

Спираль Фибоначчи выглядит следующим образом:



# Числа Фибоначчи - параллельная рекурсия

Каждое из чисел Фибоначчи – это длина стороны квадрата, в которой находится данное число. Математически числа Фибоначчи определяются следующим образом:

Следовательно, довольно просто написать рекурсивную функцию для вычисления  $n$ -го числа Фибоначчи:

$$f(n) = \begin{cases} 0, & \text{если } n = 0 \\ 1, & \text{если } n = 1 \\ f(n-1) + f(n-2), & \text{если } n > 1 \end{cases}$$

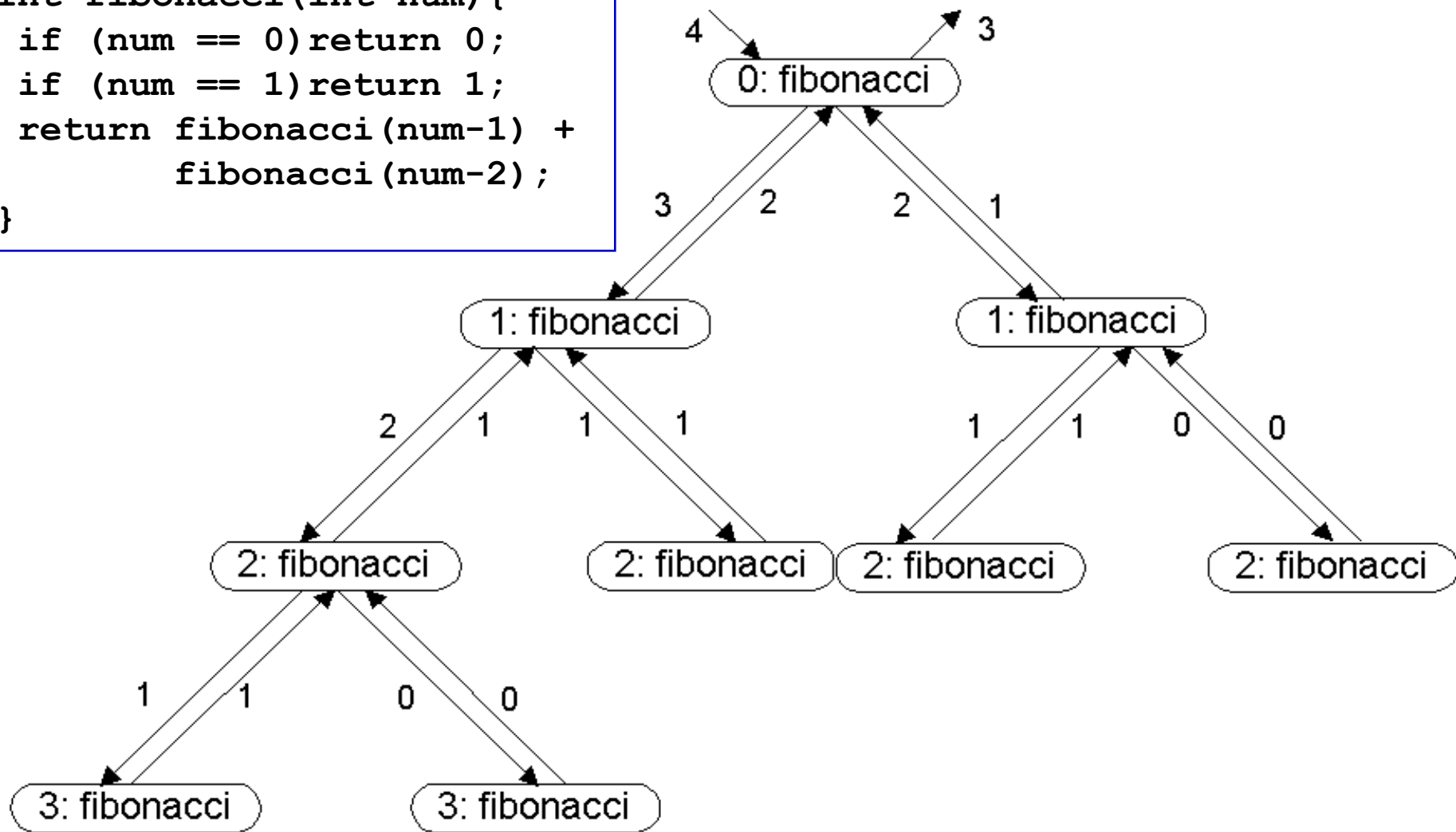
```
#include <iostream>
int fibonacci(int num) {
    if (num == 0) return 0; // условие завершения
    if (num == 1) return 1; // условие завершения
    return fibonacci(num-1) + fibonacci(num-2);
}
// Выводим первые 13 чисел Фибоначчи
int main() {
    for (int count=0; count < 13; ++count)
        std::cout << fibonacci(count) << " ";
    return 0;
}
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144
-----
```



# Параллельная рекурсия

```
int fibonacci(int num){  
    if (num == 0) return 0;  
    if (num == 1) return 1;  
    return fibonacci(num-1) +  
           fibonacci(num-2);  
}
```



Здесь каждый вызов порождает два отложенных вызова. И вызовы повторяются при одних и тех значениях аргументов.

# Сравнение рекурсии и циклов

Наиболее популярный вопрос, который задают о рекурсивных функциях: «Зачем использовать рекурсивную функцию, если задание можно выполнить и с помощью итераций (используя циклы **for** или **while**)?». Оказывается, всегда можно решить рекурсивную проблему итеративно. Однако, для нетривиальных случаев, рекурсивная версия часто бывает намного проще как для написания, так и для чтения. Например, функцию вычисления  $n$ -го числа Фибоначчи можно написать и с помощью итераций, но это будет сложнее! (Попробуйте!)

**Итеративные функции** (те, которые используют циклы **for** или **while**) почти всегда более эффективны, чем их рекурсивные аналоги. Это связано с тем, что каждый раз при вызове функции, расходуется определенное количество ресурсов, которое тратится на добавление и вытягивание фреймов из стека. Итеративные функции расходуют намного меньше этих ресурсов.

# Сравнение рекурсии и циклов

Это не значит, что итеративные функции всегда являются лучшим вариантом. Иногда рекурсивная реализация может быть чище и проще, а некоторые дополнительные расходы могут быть более чем оправданы, сведя к минимуму трудности при будущей поддержке кода, особенно, если алгоритм не требует слишком много времени для поиска решения.

В общем, рекурсия является хорошим выбором, если выполняется большинство из следующих утверждений:

- рекурсивный код намного проще реализовать;
- глубина рекурсии может быть ограничена;
- итеративная версия алгоритма требует управления стеком данных;
- это не критическая часть кода, которая напрямую влияет на производительность программы.



# Сравнение рекурсии и циклов

Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями. Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно. Достоинством рекурсии является компактная запись, а недостатками – расход времени и памяти на повторные вызовы функции, передачу ей копий параметров, создание локальных переменных и, главное, опасность переполнения стека.

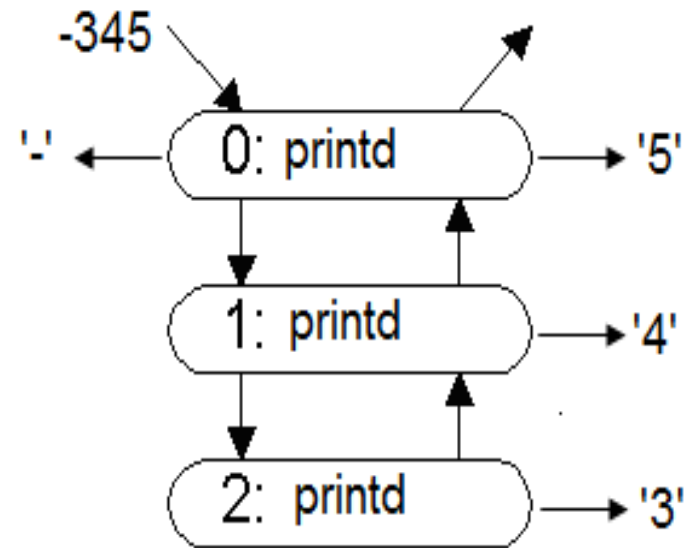
## **Доп. задание:**

Напишите рекурсивную функцию, которая принимает целое число в качестве входных данных и возвращает сумму всех чисел этого значения (например,  $482 = 4 + 8 + 2 = 14$ ). Протестируйте вашу программу, используя число 83569 (результатом должно быть 31).

# Прямая рекурсия

Рассмотрим печать числа в виде строки символов.

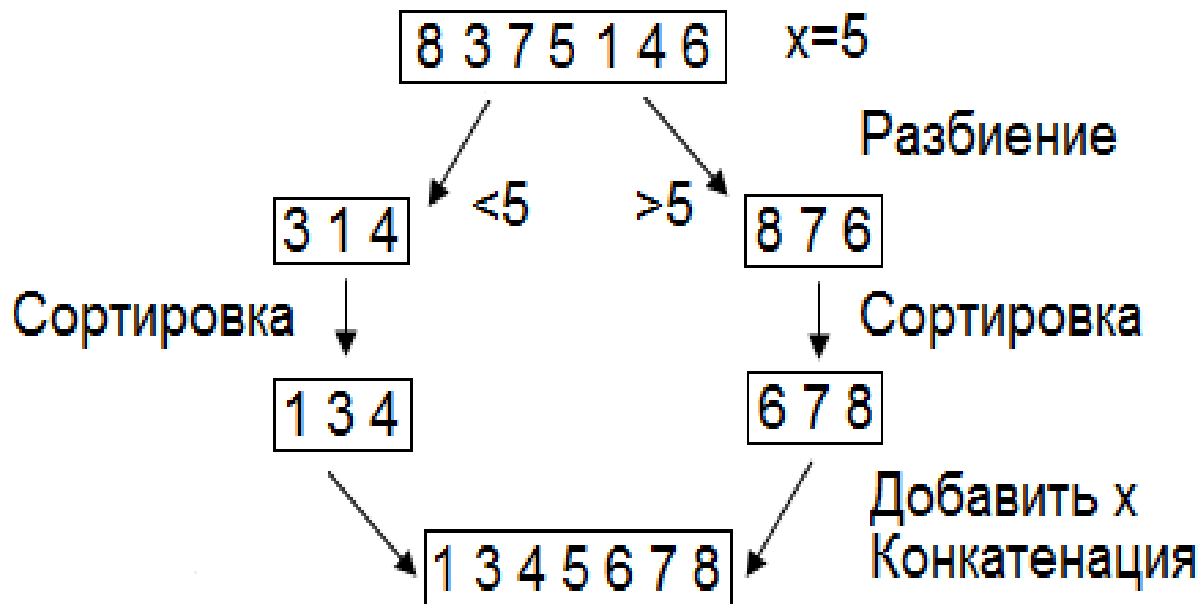
```
#include <stdio.h>
void printd(int n) {
    if (n < 0) {
        putchar('-'); n = -n;
    }
    if (n / 10) printd(n / 10)
    putchar(n % 10 + '0' );
}
```



В обращении **printd(-345)** при первом вызове аргумент **n = 345**, при втором - **34**, при третьем - **3**. Функция на третьем уровне вызова печатает **3** и возвращается на второй уровень; печатает **4** и возвращается на первый уровень, печатает **5** и заканчивает работу.

# Быстрая сортировка

Следующий хороший пример рекурсии – это **быстрая сортировка**, предложенная Ч. А. Р. Хоаром в 1962 г. Для заданного массива выбирается элемент, который разбивает массив на два подмножества - те, что меньше, и те, что больше него. Эта же процедура рекурсивно применяется и к двум полученным подмножествам. Если в подмножестве менее двух элементов, то сортировать нечего, и рекурсия завершается.



# Быстрая сортировка

```
/* qsort: сортирует v[left]...v[right] по возрастанию */  
void qsort(int v[], int left, int right) {  
    int i, last;  
    void swap(int v[], int i, int j);  
    if (left >= right)          /* ничего не делать, если */  
        return;                /* в массиве менее двух элементов */  
    swap(v, left, (left+right)/2); /*делящий эл-т перенести в v[0]*/  
    last = left;  
    for(i = left+1; i <= right; i++) /* деление на части */  
        if (v[i] < v[left]) swap(v, ++last, i);  
    swap(v, left, last);        /* возврат делящего элемента */  
    qsort(v, left, last-1);     /*сортировка левой части*/  
    qsort(v, last+1, right);    /*сортировка правой части*/  
}
```

# Быстрая сортировка

```
/* swap: поменять местами v[i] и v[j] */  
void swap(int v[], int i, int j) {  
    int temp;  
    temp = v[i];  
    v[i]=v[j];  
    v[j] = temp;  
}
```

Рекурсивная программа не обеспечивает ни экономии памяти, поскольку требуется где-то поддерживать стек значений, подлежащих обработке, ни быстродействия; но по сравнению со своим нерекурсивным эквивалентом она часто короче, и часто намного легче для написания и понимания.

# Рекурсивные функции: достоинства и недостатки

- Любую рекурсию теоретически можно заменить обычными циклами.
- Рекурсии требуют больше времени и памяти, чем требуют обычные циклы.
- Но иногда рекурсия способствует культуре кода, т. е. код становится читать легко и алгоритм работы понимается значительно быстрее, чем при использовании циклов для реализации того же самого алгоритма.
- Основное достоинство рекурсии в том, что с помощью конечной рекурсивной программы можно описать бесконечное вычисление, причём программа не будет содержать явных повторений.
- Основной недостаток рекурсии – издержки времени на создание экземпляров локальных переменных для функций и прожорливость к памяти.