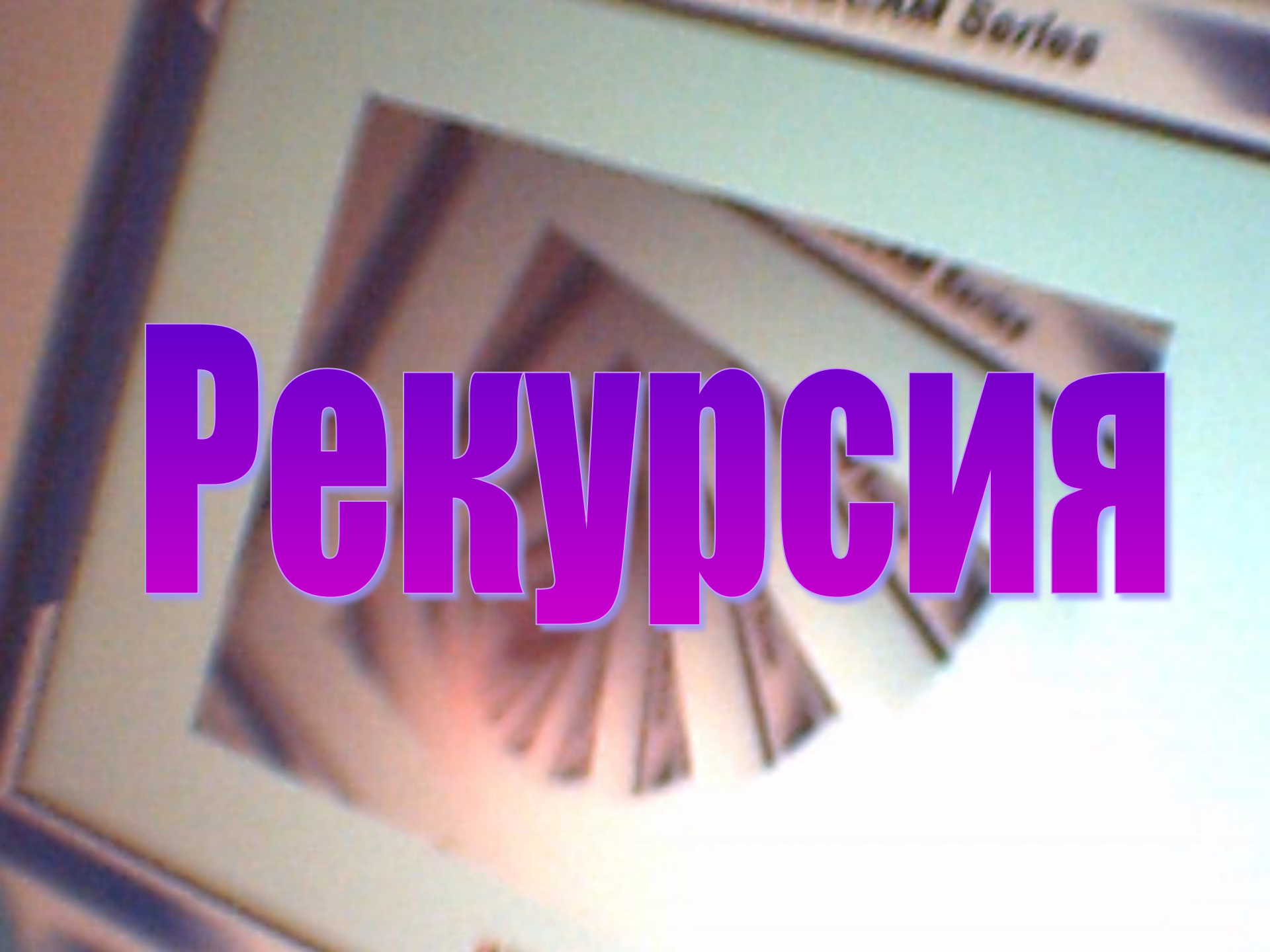
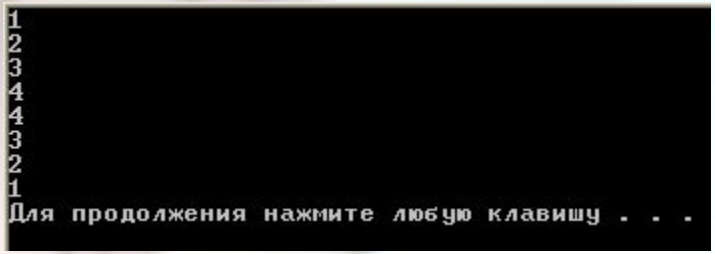

Рекурсия

Рекурсивная функция – это…

- Функция, которая вызывает саму себя

```
#include "iostream";
int rekurs(int count);
using namespace std;

void main()
{
	int a = 1;
	rekurs(a);
}
int rekurs (int count) //рекурсивная функция
{
	cout << count << endl;
	if (count <= 3)
		rekurs (count + 1);
	cout << count << endl;
	return count;
}
```

Итог работы программы



Рекурсия изнутри

Базис рекурсии - это предложение, определяющее некую начальную ситуацию или ситуацию в момент прекращения. Как правило, в этом предложении записывается некий простейший случай, при котором ответ получается сразу даже без использования рекурсии.

Шаг рекурсии - это правило, в теле которого обязательно содержится, в качестве подцели, вызов определяемого предиката.

Подпрограмма – Все, что находится внутри рекурсивной функции

Рекурсия изнутри (пример)

Рассмотрим части рекурсивной функции на основе примера, вычисляющей факториал числа

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
int factorial(int n)
{
      if (n == 0) // Базис рекурсии
      {
        return 1;
      }
      Else // recursive call
      {
        int  value = factorial(n - 1);
        return n * value;
      }
}
int main ()
{
      cout << factorial(5) << endl;
      return NULL;
}
```

А вот, собственно, и ее части:

```
if (n == 0)
{
      return 1;
}
```

Это базис рекурсии. При использовании данного алгоритма мы не нуждаемся в исплользовании рекурсии и поэтому далее не вызываем ее (для данного участка алгоритма)

---

n – 1 //Это шаг рекурсии. При последующем вызове функции мы передаем ей число на единицу большее, чем получили.

```
int factorial(int n)
if (n == 0)      {
                 return 1;
      }
      Else // recursive call
      {
                 int value = factorial(n - 1);
                 return n * value;
      }
```

Это подфункция. Данный алгоритм вызывается тогда, когда мы вызываем саму функция factorial (int n)

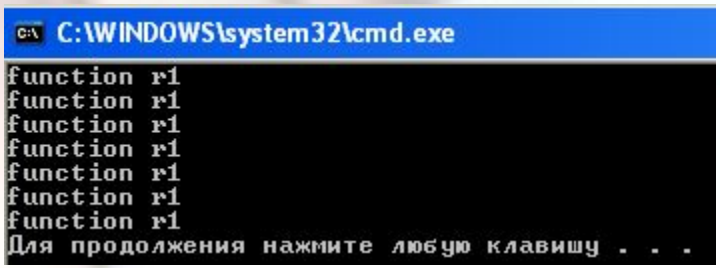
Типы рекурсий: прямая рекурсия

Прямая рекурсия – непосредственный вызов алгоритма (функции, процедуры, метода) из текста самого метода

```
#include <iostream>
using namespace std;
void r1 (int a);
void r2 (int a);
void r3 (int a);

void r1(int a)
{
        cout << "function r1" << endl;
        if (a < 6)
        r1(a+1);
}

int main ()
{
r1 (0);
return NULL;
}
```

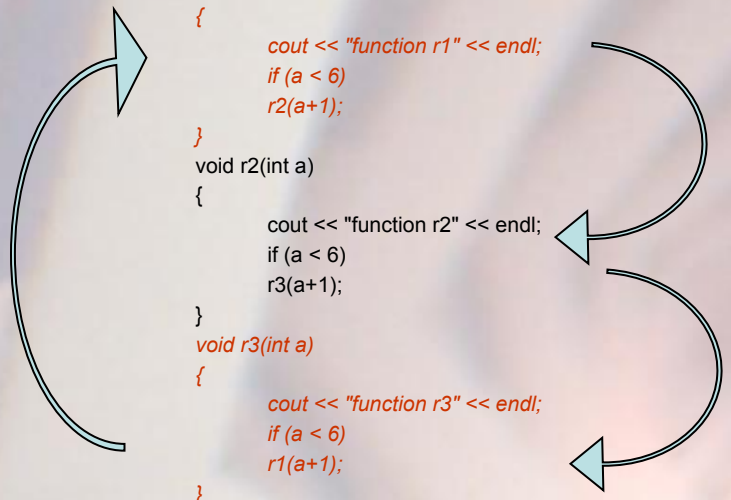
В данном случае функция r1() вызывает саму себя

Вот результат работы программы



Типы рекурсий: косвенная

Косвенная рекурсия – При косвенной рекурсии мы имеем циклическую последовательность вызовов нескольких алгоритмов.

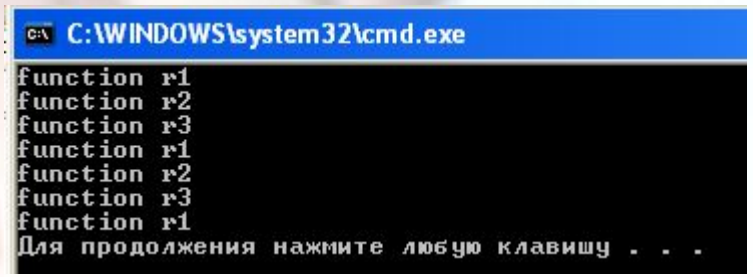


```
#include <iostream>
using namespace std;
void r1 (int a);
void r2 (int a);
void r3 (int a);

void r1(int a)
{
        cout << "function r1" << endl;
        if (a < 6)
        r2(a+1);
}
void r2(int a)
{
        cout << "function r2" << endl;
        if (a < 6)
        r3(a+1);
}
void r3(int a)
{
        cout << "function r3" << endl;
        if (a < 6)
        r1(a+1);
}
int main ()
{
        r1 (0);
        return NULL;
}
```

В данном случае функция r1() вызывает функцию r2(), которая вызывает r3().

Функция r3() в свою очередь снова вызывает r1()

Вот результат работы этой программы:



типы рекурсий: линейная

- *Линейная рекурсия* - Если исполнение подпрограммы приводит только к одному вызову этой же самой подпрограммы, то такая *рекурсия* называется ***линейной***.

```
#include <iostream>

using namespace std;
void function(int a);

void function (int a)
{
        if (a > 0)
        function(a-1);
}
int main ()
{
        function(3);
        return NULL;
}
```

типы рекурсий: ветвящаяся

```
#include <iostream>

using namespace std;
int function(int a);

int function (int a)
{
	if (a > 3)
	a = function (a-1) * function(a-2);
	return a;
}

int main ()
{
	cout << function(6) << endl;
	return NULL;
}
```

- *Ветвящаяся рекурсия* - Если каждый экземпляр подпрограммы может вызвать себя несколько раз, то *рекурсия* называется ***нелинейной*** или "***ветвящейся***".

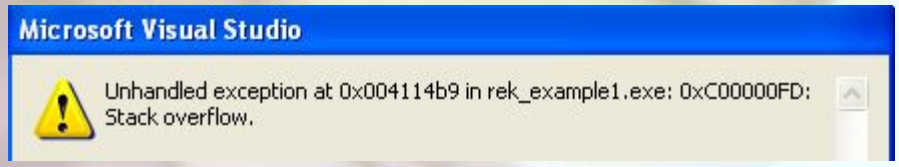
Бесконечная рекурсия*

Одна из самых больших опасностей рекурсии – бесконечный вызов функцией самой себя.

Причиной такой проблемы чаще всего является отсутствие Базиса, либо других точек останова. А так же неправильно заданные точки прерывания рекурсии.

Например:
```
void function ()
{ function(); }
```

Другой пример:
```
int Function (unsigned int n)
// Unsigned int – тип, содержащий
//неотрицательные значения
{ if (n > 0)
  {
      Function(n++); return n;
  }
      else return 0;
}
```



При использовании подобных алгоритмов может выскочить ошибка, предупреждающая о переполнении стека**

*На самом деле это условное обозначение так как при переполнении памяти компьютера программа выдаст ошибку и/или завершит ее в аварийном режиме

**Про стек будет расказано далее

Стековая организация рекурсии

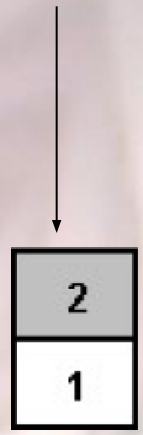
Во-первых: что такое стек?

Стек – это линейная организация данных, которая предоставляет доступ только к последнему помещенному элементу. Часто применяют абревиатуру LIFO – last in – first out (последний вошел – первый вышел).

1) Помещаем элемент 1.

Имеем доступ к элементу 1

2) Помещаем элемент 2



Имеем доступ к элементу 2, но не имеем доступа к элементу 1

3) Помещаем элемент 3

Имеем доступ к элементу 3, но не имеем доступа к элементам 1 и 2

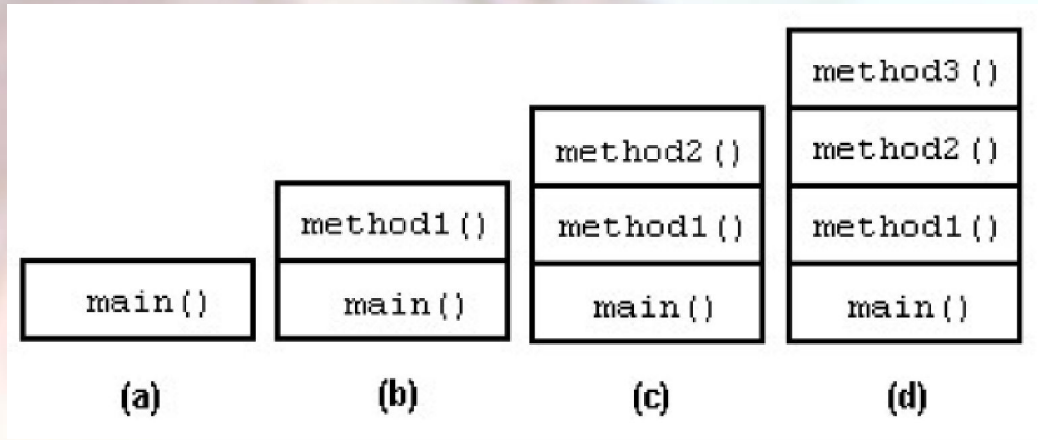
Стековая организация рекурсии

Проведем аналогию для рекурсии. В нашем примере элементами рекурсии служат вызываемые функции:

```
#include <iostream>

using namespace std;
void method3(void)
{
	cout << "Method 3" << endl;
}
void method2(void)
{
	method3();
}
void method1(void)
{
	method2();
}
int main()
{
	method1();
	return EXIT_SUCCESS;
}
```

Вызывая функции одну за другой мы «наращиваем» таким образом стек, состоящий из этих же функций

При выходе из метода, мы удаляем его из стека, и переходим к следующему.



Преимущества рекурсии

1) Часто это наиболее легкий метод написания алгоритма для задач, которые можно решить с помощью рекурсии(число Фиббоначи, Факториал)
2) В некоторых случаях можно выиграть затрачиваемое время для решения задачи. Например при использовании Быстрой сортировки трудоемкость алгоритмы составляет N*ln(N). Большинство остальных сортировок (сортировка вставками, сортировка минимума с обменом) использует N^2 времени (т.е. Затраты времени на сортировку пропорциональны «N квадрат») – где N = количество элементов в сортируемом списке.

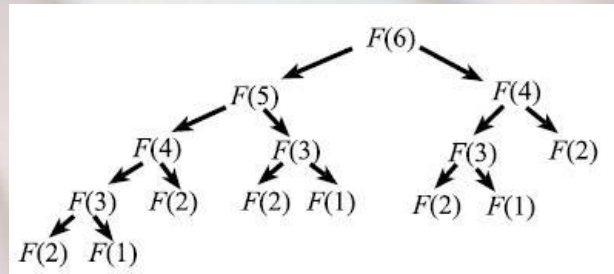
Недостатки рекурсии

1) Велика возможность войти в бесконечный цикл

2) При использовании некоторых формул слишком большие затраты памяти компьютера. Например если вычислять число Фибоначи или Факториал, нам приходится запоминать все значения чисел (в связи со стековой организацией рекурсии), и вычислять одни и те же значения по многу раз.

На рисунке приведен упрощенный пример работы нахождения числа Фибоначи

Можно заметить, что *F*(3) вычисляется три раза. Если рассмотреть вычисление *F*(*n*) при больших *n*, то повторных вычислений будет очень много. Это и есть основной недостаток рекурсии — повторные вычисления одних и тех же значений



3) В случае, если вызываемых функций будет очень много, может произойти переполнение стека.

Альтернатива рекурсии

Наиболее сильный аргумент против рекурсии, не зависящий от программиста, – быстрое расходование памяти компьютера. Для решения этой проблемы выделен отдельный предмет, называемый «Динамическое программирование»

Динамическое программирование работает почти так же, как и рекурсия, за исключением того, что «запоминает» лишь необходимые значения. Таким образом мы не используем стек, хранящий все данные, а храним несколько необходимых значений.

Обычно в динамическом программировании используют итеративный подход – Циклический перебор с запоминанием ТОЛЬКО последних нужных значений, которые далее используются как аргументы вызываемой функции.

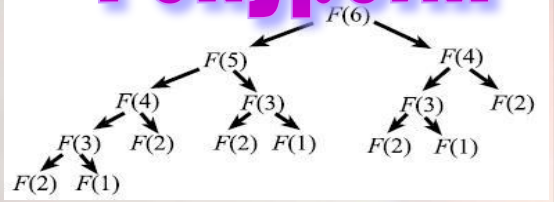
Что лучше: Итерация или рекурсия?

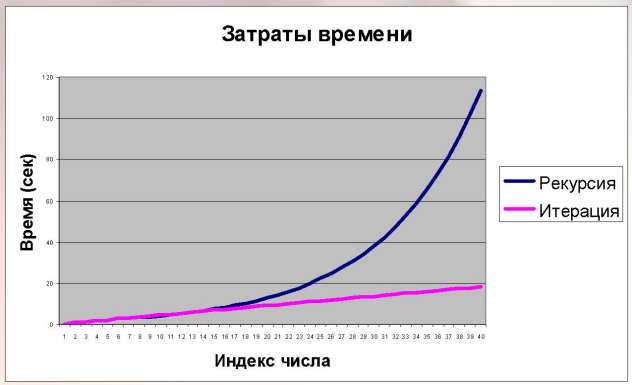
Для того чтобы превратить нелинейную *рекурсию* в итеративный алгоритм, придется приложить много усилий и, возможно, даже внести некоторые изменения в обрабатываемую структуру данных. В этом случае сложно сказать, каким алгоритмом лучше пользоваться.

Если же рекурсия линейная, то ее лучше всего реализовать с помощью итеративного алгоритма так как сложность самого алгоритма будет почти такая же, как и у рекурсивного алгоритма. Но трудоемкость чаще всего уменьшается.

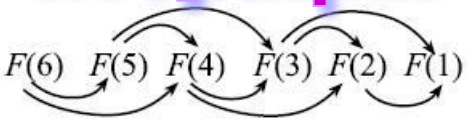
Для сравнения приведена работа двух программ, реализующих вычисление числа Фибоначчи по индексу, основанных на Рекурсии и Динамическом программировании.

Рекурсия





Итерация



Примеры переходов от рекурсии к итерации

Нахождение числа Фиббоначи по его индексу (число 0 имеет индекс 0)

Рекурсивный метод

```
#include <iostream>

using namespace std;
int n;
int Fsum;

int fibbonachi(int n)
{
	if (n>2)
	{
		Fsum = (fibbonachi(n-1)+ fibbonachi(n-2));
	return Fsum;
	}
	else return 1;
}

void main()
{
	cout << fibbonachi(7) << endl;
}
```

Итеративный

```
#include <iostream>

using namespace std;
int n, F1, F2, Fsum, temp;

int fibbonachi(int n)
{
	for (int i = 2; i <= n; i++)
	{
		Fsum = F1 + F2;
		F2=F1;
		F1 = Fsum;
	}
	return Fsum;
}

void main()
{
	F1 = 1;
	F2 = 0;
	Fsum = 0;
	cout << fibbonachi(7) << endl;
}
```

Примеры переходов от рекурсии к итерации

Нахождение факториала числа по его индексу

Рекурсивный метод

```
#include <iostream>

using namespace std;
int n, F;

int factorial(int n)
{
        if (n > 1)
        {
                F = factorial (n-1)*n;
                return F;
        }
        else return 1;
}

void main()
{
        F = 1;
        cout << factorial(6) << endl;
}
```

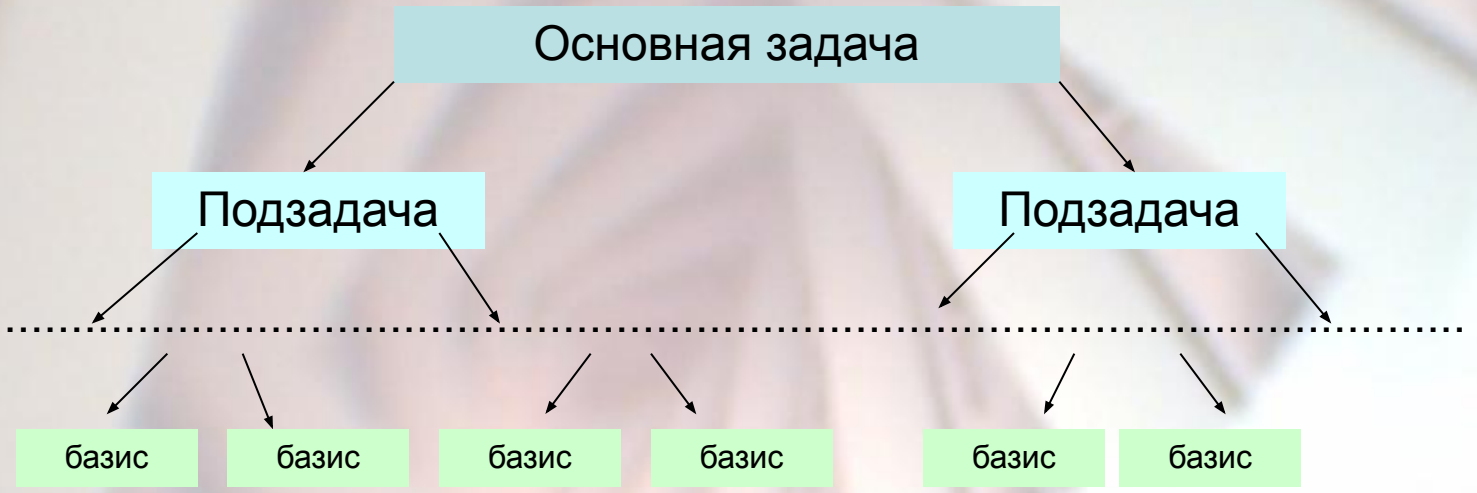
Итеративный

```
#include <iostream>

using namespace std;
int n, F;

int factorial(int n)
{
        for (int i = 1; i <=n; i++)
                F = F*i;
        return F;
}

void main()
{
        F = 1;
        cout << factorial(5) << endl;
}
```

Решаемые рекурсией проблемы

- Разделяй и властвуй

Это метод решения задачи с помощью разделения первоначальной задачи на более мелкие подзадачи, которые решаются аналогичным методом. Разделение происходит до тех пор, пока не будет достигнут базис рекурсии



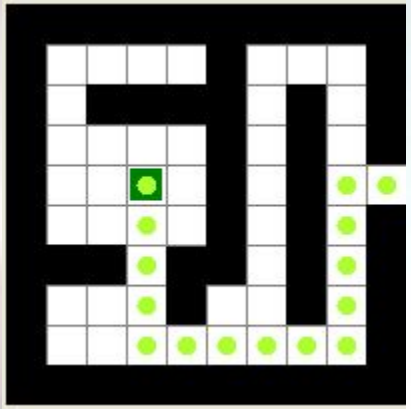
Решаемые рекурсией проблемы

- BackTracking

Суть метода заключается в поиске решения с помощью перебора. Удобно использовать, если, например, нужно найти выход из лабиринта.

Принцип решения задачки выхода из лабиринта таков, что если «испытуемый» упирается в стенку и «понимает» что зашел не туда, то может возвратиться обратно и если будет возможность – повернуть в другую сторону для дальнейшего поиска выхода.

Это и называется BackTracking



Рекурсивные алгоритмы

Вот наиболее распространенные задачи, которые часто решают с помощью Рекурсии:

- Число Фибоначчи
- Факториал числа
- Задача о ханойских башнях
- Функция Аккермана
- Задача о золотых горках
- Задача «Сделай палиндром»
- Задача коммивояжёра
- Нахождение выхода из лабиринта
- Задача о 8 королевах (шахматы)

Кратко о задачах...

Число Фибоначчи: Рассмотрим последовательность чисел в которой каждое число является суммой двух предыдущих. Это числа Фибоначчи. Формальное их определение таково:
$F(1) = 1 \quad ,F(2) = 1 \quad ,F(n) = F(n-2) + F(n-1)$если $\quad , n > 2$.
Функция $F(n)$ задана *рекурсивно*, то есть «через себя». База — значения функции F на аргументах 1 и 2, а шаг — формула $F(n) = F(n-2) + F(n-1)$.

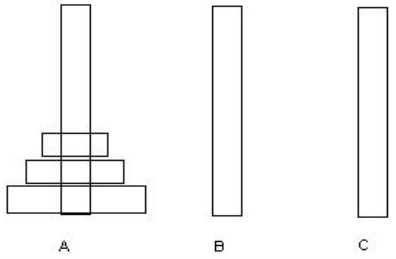
Факториал: Рассмотрим последовательность чисел, в котором каждое число является произведением всех предыдущих. Это Факториал числа. Можно задать с помощью формулы F(1) = 1, F(n) = F(n-1)*n (или !n = !(n-1)*n). База – F(1)=1; шаг – сама формула

Ханойские башни: Задача звучит следующим образом:

В одном из буддийских монастырей монахи уже тысячу лет занимаются перекладыванием колец. Они располагают тремя пирамидами, на которых надеты кольца разных размеров. В начальном состоянии 64 кольца были надеты на первую пирамиду и упорядочены по размеру. Монахи должны переложить все кольца с первой пирамиды на вторую, выполняя единственное условие — кольцо нельзя положить на кольцо меньшего размера. При перекладывании можно использовать все три пирамиды. Монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света…

Задача решается в 4 шага, повторяющихся друг за другом:

– Переместить N колец с A на C, используя B как промежуточный
– Переместить (N-1) кольцо с A на B, используя C как промежуточный
– Переместить кольцо с A на C
– Переместить (N-1) кольцо с B на C, используя A как промежуточный



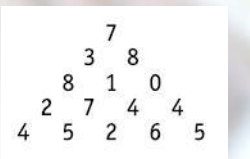
Кратко о задачах...

Функция Аккермана: определяется рекурсивно для целых чисел *m* и *n* следующим образом:

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0, n = 0 \\ A(m-1, A(m,n-1)) & m > 0, n > 0 \end{cases}$$

Может показаться неочевидным, что рекурсия всегда заканчивается. Это следует из того, что при рекурсивном вызове или уменьшается значение *m*, или значение *m* сохраняется, но уменьшается значение *n*. Это означает, что каждый раз пара (m; n) уменьшается с точки зрения лексикографического порядка, значит, значение *m* в итоге достигнет нуля: для одного значение *m* существует конечное число возможных значений *n* (так как первый вызов с данным *m* был произведён с каким-то определённым значением *n*, а в дальнейшем, если значение *m* сохраняется, значение *n* может только уменьшаться), а количество возможных значений *m*, в свою очередь, тоже конечно. Однако, при уменьшении *m* значение, на которое увеличивается *n*, неограничено и обычно очень велико.

Золотые горки: На рисунке показан пример треугольника из чисел. Написать программу, вычисляющую наибольшую сумму чисел, через которые проходит путь, начинающийся на вершине и заканчивающийся где-то на основании.



Пусть число $a(i,j)$ есть число в треугольнике, находящееся в i-й строчке на j-м месте, а число (i,j) есть максимальное значение суммы, которое можно получить спускаясь с горки, начиная с этого элемента. Понятно, что (i,j) есть число $a(i,j)$ плюс максимум из двух вариантов: $(i + 1,j)$ и $(i + 1,j + 1)$. Эти два варианта соответствуют тому, что мы можем спуститься вниз-вправо или вниз-влево. Получилось рекурсивное определение: S(i,j) = a(i; j) + max(S(i+1 ; j), S(i+1 ; j+1))

Кратко о задачах...

Задача о коммивояжере: Рекурсия с запоминанием работает не всегда. Рассмотрим пример задачи, для которой есть долго работающий рекурсивный алгоритм, который нельзя существенно ускорить с помощью запоминания вычисленных значений.

Задача коммивояжёра — побывать во всех городах (ровно по одному разу) и при этом потратить как можно меньше денег на проезд и вернуться обратно. При том, что:

- внутри города проезд ничего не стоит;
- проезд между двумя городами напрямую стоит одинаково в оба конца;
- стоимость — целое число от 1 до 10000;
- городов не более 100.

Сделай палиндром: Палиндром — это последовательность символов, которая слева-направо и справа-налево пишется одинаково. Например «АБА» или «АББ ББА». Дана последовательность символов. Какое минимальное количество символов нужно удалить из неё, чтобы получить палиндром?

Решение: Если строка имеет вид h α t , где *h* и *t* символы, а α — подстрока, возможно пустая. Пусть *S*(*x*) — вычисляет минимальное количество символов, которые нужно убрать из строки *x*, чтобы оставшаяся строка была палиндромом.

Базой рекурсии являются строки из одного символа и пустая строка — все они полиндромы по определению, и для них $S = 0$. Шаг рекурсии состоит из двух частей:

$$S(h\,\alpha\,t) = \begin{cases} S(\alpha), & h = t; \\ \min\left(S(\alpha t) + 1,\ S(h\alpha) + 1,\ S(\alpha) + 2\right), & h \neq t. \end{cases}$$

Заключение

Рекурсия, дополненная идеями динамического программирования, жадными алгоритмами и идеей отсечения, превращается в мощный инструмент для программистов. Но не следует забывать, что краткость записи рекурсивных функций не всегда означает высокую скорость их вычисления. И есть ряд задач, в которых рекурсия просто вредна (такова, например, задача вычисления кратчайшего пути в графе).