####

#### Содержание

[Введение 2](#_Toc262819110)

[1. Теория рекурсивных алгоритмов 3](#_Toc262819111)

[1.1. Понятие рекурсии и её виды 3](#_Toc262819112)

[1.2. Общие принципы программной реализации рекурсии. 9](#_Toc262819113)

[2. Программная реализация рекурсии 17](#_Toc262819114)

[2.1. Примеры решения задач с помощью рекурсии 17](#_Toc262819115)

[2.2. Решение экономической задачи с использованием рекурсивного алгоритма 20](#_Toc262819116)

[Заключение 22](#_Toc262819118)

[Список использованных источников 23](#_Toc262819119)

#### Введение

Не так давно человечество вступило в новый XXI век – век информации и компьютеров. Благодаря бурному развитию научных, технических и технологических исследований стало возможным хранить огромные объёмы данных и преобразовывать их со скоростями, которые ещё несколько десятилетий назад были немыслимы. При создании сложных информационных систем перед проектировщиками стали нетривиальные задачи, требующие разработки новых концепций программирования.

Для решения проблем такого рода, особенно при учёте человеческого фактора, возникает необходимость обеспечения понятности алгоритма, так называемой «читабельности» исходного кода программы, и как следствие модифицируемости и относительной лёгкости сопровождения конечного программного продукта. Часто этого можно достигнуть включением в реализацию приложения рекурсивных подпрограмм, механизмы использования которых предоставляются практически всеми современными компиляторами и средами разработки.

В настоящее время область практического применения рекурсии весьма широка. Она включает, в частности, сложные задачи численного анализа, алгоритмы трансляции, а также различные операции над списками, являющиеся необходимым аппаратом разработки современных автоматизированных систем управления. Поэтому аппарат рекурсии предусматривается практически во всех языках программирования, появляющихся после АЛГОЛа.

Полезность, важность и необходимость рекурсии, как одного из концептуальных методов решения практических задач подчеркивалась многими мэтрами информатики. Одни из них, это два лауреата премии Тьюринга: американский специалист по системному программированию Д.Кнут и английский теоретик информатики Ч.Хоар.

Значимость рекурсии определяется тем, что она является одним из наиболее мощных, а также самым общим методом научного познания. Она эффективно применяется во многих прикладных и теоретических естественнонаучных дисциплинах, и стала неотъемлемой их частью.

#### 1. [Теория рекурсивных алгоритмов](#_Toc105389918)

##### 1.1. Понятие рекурсии и её виды

Рекурсия – метод определения класса объектов или методов предварительным заданием одного или нескольких (обычно простых) его базовых случаев или методов, а затем заданием на их основе правила построения определяемого класса, ссылающегося прямо или косвенно на эти базовые случаи.

Другими словами, рекурсия – способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Рекурсия используется, когда можно выделить самоподобие задачи.

Рекурсивный алгоритм (процедура, функция):

* алгоритм называется рекурсивным, если в его определении содержится прямой или косвенный вызов этого же алгоритма;
* рекурсивная функция – одно из математических уточнений интуитивного понятия вычислимой функции.

Адаптивный рекурсивный алгоритм – алгоритм, который благодаря рекурсивности учитывает те или иные индивидуальные характеристики решаемой задачи из области своего определения.

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.

Основное правило рекурсии: до рекурсивного вызова должна стоять проверка на возврат из рекурсии. Существуют следующие виды рекурсии:

* прямая рекурсия – непосредственный вызов алгоритма (функции, процедуры, метода) из текста самого метода.

В данном случае функция r1( ) вызывает саму себя.

#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;

}

Результат работы программы представлен на рисунке 1.1.

Рис. 1.1. Прямая рекурсия

* косвенная рекурсия.

При косвенной рекурсии имеется циклическая последовательность вызовов нескольких алгоритмов.

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

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

#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;

}

Результат работы программы представлен на рисунке 1.2.

Рис. 1.2. Косвенная рекурсия

* линейная рекурсия – если исполнение подпрограммы приводит только к одному вызову этой же самой подпрограммы, то такая рекурсия называется линейной.

**Корень**

Рис. 1.3. Линейная рекурсия

#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;

}

* ветвящаяся рекурсия – если каждый экземпляр подпрограммы может вызвать себя несколько раз, то рекурсия называется нелинейной или "ветвящейся".

**Корень**

Рис. 1.4. Ветвящаяся рекурсия

#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;

}

* бесконечная рекурсия(на самом деле это условное обозначение так как при переполнении памяти компьютера программа выдаст ошибку и/или завершит ее в аварийном режиме).

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

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;

}

Результат работы программы представлен на рисунке 1.5.

Рис. 1.5. Сообщение о переполнении стека

При использовании подобных алгоритмов появляется ошибка, предупреждающая о переполнении стека. Причиной такой проблемы чаще всего является отсутствие базиса, либо других точек останова, так же неправильно заданные точки прерывания рекурсии.

#####

##### 1.2. Общие принципы программной реализации рекурсии.

В программировании рекурсия – вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная рекурсия), например, функция A вызывает функцию B, а функция B – функцию A. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии.

Мощь рекурсивного определения объекта в том, что такое конечное определение способно описывать бесконечно большое число объектов. С помощью рекурсивной программы же возможно описать бесконечное вычисление, причём без явных повторений частей программы.

Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов – адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за этот счёт работает корректно. Оборотной стороной этого довольно простого по структуре механизма является то, что рекурсивные вызовы не бесплатны – на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов. Вследствие этого обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии.

Имеется специальный тип рекурсии, называемый «хвостовой рекурсией». Интерпретаторы и компиляторы функциональных языков программирования, поддерживающие оптимизацию кода (исходного и/или исполняемого), автоматически преобразуют хвостовую рекурсию к итерации, благодаря чему обеспечивают выполнение алгоритмов с хвостовой рекурсией в ограниченном объёме памяти. Такие рекурсивные вычисления, даже если они формально бесконечны (например, когда с помощью рекурсии организуется работа командного интерпретатора, принимающего команды пользователя), никогда не приводят к исчерпанию памяти.

Нужно учитывать одну особенность, характерную для рекурсивных программ, подобных той, которую используют для генерации чисел Фибоначчи. Каждый уровень рекурсии в функции fibonacci удваивает количество вызовов, так что количество рекурсивных вызовов, которое должно быть выполнено для вычисления n – го числа Фибоначчи, оказывается порядка 2N.

Объем вычислений резко нарастает с увеличением n. Вычисление только 20 – го числа Фибоначчи потребовало бы порядка 2N или около миллиона вызовов, вычисление 30 – го числа Фибоначчи потребовало бы порядка 3N или около миллиарда вызовов и так далее. В методах вычислений это называется экспоненциальной сложностью.

Из этого следует, что по возможности нужно избегать рекурсивных программ, подобных программе для вычисления чисел Фибоначчи, которые приводят к экспоненциальному нарастанию количества вызовов.

Любые задачи, которые можно решить рекурсивно, могут быть решены также и

итеративно (нерекурсивно). Обычно рекурсивный подход предпочитают итеративному, если он более естественно отражает задачу и ее результаты, то есть более нагляден и легче отлаживается. Другая причина предпочтения рекурсивного решения состоит в том, что итеративное решение может не быть очевидным.

Рекурсивные алгоритмы в программировании реализованы в механизме так называемых рекурсивных подпрограмм. Рекурсивной считается подпрограмма, которая прямо или косвенно, через другие подпрограммы, обращается к себе, быть может с иными фактическими параметрами. В современных системах программирования корректное функционирование подпрограмм, особенно рекурсивных, обеспечивается с помощью стека.

Стек – связная структура данных, построенная на принципе «первый пришёл – первый вышел» (Fiгst In – Fiгst Out, FIFO). То есть вновь добавляемые объекты помещаются в начало, вершину стека, и выбираются они также только из вершины.

Стек является чрезвычайно удобной структурой данных для многих задач вычислительной техники. Наиболее типичной из таких задач является обеспечение вложенных вызовов процедур. Предположим, имеется процедура A, которая вызывает процедуру B, а та в свою очередь – процедуру C. Когда выполнение процедуры A дойдет до вызова B, процедура A приостанавливается и управление передается на входную точку процедуры B. Когда B доходит до вызова C, B приостанавливается и управление передается на процедуру C. Когда заканчивается выполнение процедуры C, управление должно быть возвращено в B, причем в точку, следующую за вызовом C. При завершении B управление должно возвращаться в A, в точку, следующую за вызовом B. Правильную последовательность возвратов легко обеспечить, если при каждом вызове процедуры записывать адрес возврата в стек. Так, когда процедура A вызывает процедуру B, в стек заносится адрес возврата в A; когда B вызывает C, в стек заносится адрес возврата в B. Когда C заканчивается, адрес возврата выбирается из вершины стека – а это адрес возврата в B. Когда заканчивается B, в вершине стека находится адрес возврата в A, и возврат из B произойдет в A.

Механизм вызова функции или процедуры в языке высокого уровня существенно зависит от архитектуры компьютера и операционной системы. В рамках IBM PC совместимых компьютеров, в микропроцессорах семейства Intel, как и в большинстве современных процессорных архитектур, поддерживается аппаратный стек. Аппаратный стек расположен в ОЗУ, указатель стека содержится в паре специальных регистров SS:SP доступных для программиста. Аппаратный стек расширяется в сторону уменьшения адресов, указатель его адресует первый свободный элемент. Схематично этот механизм иллюстрирован на рисунке 1.6.

Рис. 1.6. Механизм вызова функции в аппаратном стеке

Языки PASCAL, C, C++ используют стек для размещения в нем локальных переменных процедур и иных программных блоков. Стек разбит на фрагменты, представляющие собой блоки последовательных ячеек. Каждый вызов подпрограммы использует фрагмент стека, длина которого зависит от вызывающей подпрограммы. При каждой активизации процедуры память для ее локальных переменных выделяется в стеке; при завершении процедуры эта память освобождается. Поскольку при вызовах процедур всегда строго соблюдается вложенность, то в вершине стека всегда находится память, содержащая локальные переменные активной в данный момент процедуры.

Таким образом, в общем случае при вызове процедурой A процедуры B происходит следующее:

1. В вершину стека помещается фрагмент нужного размера. В него входят следующие данные:
	1. указатели фактических параметров вызова процедуры В;
	2. пустые ячейки для локальных переменных, определенных в процедуре В;
	3. адрес возврата, т.е. адрес команды в процедуре A, которую следует выполнить после того, как процедура B закончит свою работу.

Если B – функция, то во фрагмент стека для B помещается указатель ячейки во фрагменте стека для A, в которую надлежит поместить значение этой функции (адрес значения).

1. Управление передается первому оператору процедуры B.
2. При завершении работы процедуры B управление передается процедуре A с помощью следующей последовательности шагов:
3. адрес возврата извлекается из вершины стека;
4. если B – функция, то ее значение запоминается в ячейке, предписанной указателем на адрес значения;
5. фрагмент стека процедуры B извлекается из стека, в вершину ставится фрагмент процедуры A;
6. выполнение процедуры A возобновляется с команды, указанной в адресе возврата.

При вызове подпрограммой самой себя, т.е. в рекурсивном случае, выполняется та же самая последовательность действий.

Этот прием делает возможной легкую реализацию рекурсивных процедур. Когда процедура вызывает сама себя, то для всех ее локальных переменных выделяется новая память в стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая его переменными область памяти в стеке освобождается и актуальным становится представление локальных переменных предыдущего уровня.

Рекурсия использует стек в скрытом от программиста виде, но все рекурсивные процедуры могут быть реализованы и без рекурсии, но с явным использованием стека.

Рассмотрим пример для функции вычисления факториала: дерево рекурсии при вычислении 5! (рисунок 1.7).

Рис. 1.7. Дерево рекурсии при вычислении факториала числа 5

Дерево рекурсивных вызовов может иметь и более сложную структуру, если на каждом вызове порождается несколько обращений – фрагмент дерева рекурсий для чисел Фибоначчи представлен на рисунке 1.8.

Рис. 1.8. вычисление 5 – ого числа Фибоначчи (Fb(5)).

Упомянутый анализ практической сложности программ показывает, что часто асимптотически более сложные итеративные алгоритмы, на практике становятся более эффективными. Сравнивая скорость вычисления чисел Фибоначчи с помощью итеративной и рекурсивной функции можно заметить, что итеративная функция выполняется почти «мгновенно», не зависимо от значения n. При использовании же рекурсивной функции уже при n=40 заметна задержка при вычислении, а при больших n результат появляется весьма не скоро. Причина, как уже было сказано, кроется в том, что в теории не учтена зависимость времени работы программы от количества вызовов вложенных подпрограмм. Неэффективность рекурсии проявляется в том, что одни и те же вычисления производятся много раз. Особенно сильно это проявляется в методе, который был самым востребованным при построении теоретически быстрых рекурсивных алгоритмов, – методе бинарного разбиения на независимые подзадачи. Когда подзадачи независимы, это часто приводит к недопустимо большим затратам времени, так как одни и те же подзадачи решаются многократно.

Обходить подобные ситуации позволяет подход, известный как динамическое программирование. Этот подход для реализации рекурсивных программ дает возможность получать эффективные и элегантные решения для обширного класса задач.

Технология, называемая восходящим динамическим программированием (bottom – up dynamic pгogгamming) основана на том, что значение рекурсивной функции можно определить, вычисляя все значения этой функции, начиная с наименьшего, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения. Она применима к любому рекурсивному вычислению при условии, что мы можем позволить себе хранить все ранее вычисленные значения. Это в результате позволит уменьшить временную зависимость с экспоненциальной на линейную.

Нисходящее динамическое программирование (top – down dynamic pгogгamming). Оно позволяет выполнять рекурсивные функции при том же количестве итераций, что и восходящее динамическое программирование. Технология требует введения в рекурсивную программу неких средств, обеспечивающих сохранение каждого вычисленного значения и проверку сохраненных значений во избежание их повторного вычисления.

Таким образом, современные информационные технологии содержат достаточно средств, чтобы сделать теоретически эффективные рекурсивные алгоритмы, также эффективными и широко используемыми на практике. Преимущества рекурсиизаключаются в следующих аспектах:

* часто это наиболее легкий метод написания алгоритма для задач, которые можно решить с помощью рекурсии (число фибоначчи, факториал);
* рекурсивно реализованные алгоритмы, при правильных на то основаниях, имеют лаконичную запись и менее трудоёмки при последующей отладке и модификации, они сокращают временные затраты на разработку, отладку и модификацию программных средств;
* целый ряд структур данных и многие объекты современных язы­ков программирования рекурсивны по самой своей сути (фрактальные объекты, иерар­хия классов в объектно – ориентированном программировании, древовидные регулярные структуры данных) и программы для работы с такими структурами выглядят намного более естественно в рекурсивной реализации;
* рекурсия делает код более читабельным (позволяет читать код с любого места, не просматривая его весь, отслеживая все изменения переменной), что облегчает отладку;
* рекурсия защищает от ошибок типа: «действия выполнены в не верном порядке», «использована неинициализированная переменная» и других аналогичных.

Недостатки рекурсии заключаются в следующем:

* велика возможность войти в бесконечный цикл;
* при использовании некоторых формул слишком большие затраты памяти компьютера (к примеру, при вычислении числа Фибоначчи или факториалов, необходимо запоминать все значения чисел и вычислять одни и те же значения по многу раз);
* в случае, если вызываемых функций будет очень много, может произойти переполнение стека;
* следует избегать использования рекурсии в случаях, когда требуется высокая эффективность, так как они требуют времени и дополнительных затрат памяти и обладают худшей временной эффективностью.

Обслуживание рекурсивных вызовов влечет определенные накладные расходы, но при этом рекурсивные алгоритмы, разработанные методом декомпозиции, имеют лучшие асимптотические оценки. Это означает, что, начиная с некоторой длины входа, достаточно часто соответствующей области практического использования, программная реализация рекурсивного алгоритма будет иметь лучшие временные показатели.

#### 2. Программная реализация рекурсии

##### 2.1. Примеры решения задач с помощью рекурсии

**Задача 1. Числа Фибоначчи**

Рекурсия аналогична методу математической индукции. Базе индукции соответствует база рекурсии. Предположению индукции соответствует предположение о том, что нужная процедура уже написана. Шагу индукции соответствует вызов создаваемой рекурсивной процедуры. В любой рекурсии необходимо предусмотреть условие завершения процесса, т.е. когда вызова больше не происходит.

Рассмотрим – вычисление N – ого по счёту числа Фибоначчи. Числа Фибоначчи составляют последовательность, очередной элемент которой вычисляется по двум предыдущим значениям:

F­n=Fn – 1 + Fn – 2 .

#include <iostream>

int fib(int n) {

 if(n < 3)

 return 1;

 return fib(n – 2) + fib(n – 1);

}

int main() {

 int n = 0;

 std::cout << "Vvedite nomer chisla:\n";

 std::cin >> n;

 std::cout << "Result: chislo " << fib(n) << " eto " << n << "th po schetu chislo v ryade Fibonacci.\n";

 return 0;

}

Результат работы программы представлен на рисунке 2.1.

Рис. 2.1. Числа Фибоначчи

**Задача 2. Нахождение факториала**

#include <iostream>

int factorial(int n)

 {

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

 return n\* factoгial (n – 1);

 }

int main() {

 int n = 0;

 std::cout << "Vvedite chislo:\n";

 std::cin >> n;

 std::cout << "Factorial " << n << " raven " << factorial(n) << "\n";

 return 0;

}

Результат работы программы представлен на рисунке 2.2.

Рис. 2.2. Факториал числа

**Задача 3. Нахождение наибольшего общего делителя**

#include <iostream>

int nod(int m, int n)

{

if (n == 0) return m;

 else

return nod(n, m % n);

}

int main() {

 int n = 0;

 int m = 0;

 std::cout << "Vvedite pervoe chislo:\n";

 std::cin >> n;

 std::cout << "Vvedite vtoroe chislo:\n";

 std::cin >> m;

 std::cout << "NOD raven "<< nod(m, n) << "\n";

 return 0;

}

Результат работы программы представлен на рисунке 2.3.

Рис. 2.3. Нахождение наибольшего общего делителя

Рекурсия является в мощным инструментом 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).

#####

##### 2.2. Решение экономической задачи с использованием рекурсивного алгоритма

Одной и наиболее востребованной операцией в экономической сфере является расчет процентов по вкладу. Пример задачи: вкладчик положил в банк сумму в sum денежных единиц под pr процентов за один период времени (год, месяц, неделя и т.д.). Составить программу, возвращающую величину вклада по истечении m периодов времени (m = 24).

#include <iostream>

#include <windows.h>

using namespace std;

double rec\_fun(double sum, double pr, int m, int i)

{

if(m==i)

return sum;

sum+=(sum\*(pr/100));

 return rec\_fun(sum, pr, m, i+1);

}

int main ()

{

 int m;

 double sum, pr;

 cout<<"Vvedite nachalnuyu summu vklada: "<< endl;

 cin>>sum;

 cout<<"Vvedite ejemesyachniy procent po vkladu: "<< endl;

 cin>>pr;

 cout<<"Vvedite colichestvo mesyacev:"<< endl;

 cin>>m;

 cout<<"Za "<<m<<" mesyacev summa vklada sostavit: "<<rec\_fun(sum, pr, m, 0)<<endl;

 return 0;

}

Результат работы программы представлен на рисунке 2.4.

Рис. 2.4. Задача о сумме вклада

##### Проверка правильности решения задачи на Excel:

Рис. 2.5. Расчет задачи на Excel

Результаты обоих решений сходятся, следовательно ошибки в расчетах не возникло, программа работает правильно.

#### Заключение

На основании проведенного исследования можно сделать несколько выводов:

* во – первых, рекурсивные алгоритмы есть универсальное средство решения разнообразных алгоритмических проблем. Любая разрешимая задача такого рода имеет рекурсивное решение, которое при этом отличается изяществом и простотой для восприятия человеком;
* во – вторых, рекурсивные алгоритмы часто имеют более низкую асимптотическую сложность, чем эквивалентные им итерационные. То есть теоретически они быстрее;
* в – третьих, развитие современных программных средств сделало практическое использование рекурсии достаточно несложным делом, а новые концепции и технологии программирования преодолели проблему низкой эффективности рекурсивных программ, созданную необходимостью вызова большого количества подпроцедур.

Рекурсия отражает черту абстрактного мышления, проявляющуюся в самых различных приложениях (в математике, синтаксическом анализе и трансляции, древовидной сортировке и обработке данных с динамической структурой, шахматных задачах и т.д.). Именно в задачах такого рода лучше применять рекурсивные алгоритмы, так как они, избавляют от необходимости последовательного описания процессов, к тому же, практически все действующие языки программирования поддерживают рекурсию.

Но следует сказать, что всегда полезно подумать о замене рекурсии на циклические алгоритмы. Однако в некоторых случаях решение задачи без рекурсии может быть чрезвычайно сложным и прирост производительности не будет стоить потраченных усилий.

#### Список использованных источников

1. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб.: Питер, 2003. – 461 с: ил.
2. Баррон Д. Рекурсивные методы в программировании (Мир, 1974, 80 стр.).
3. Головешкин В.А., Ульянов М. В. Теория рекурсии для программистов. М.: ФИЗМАТЛИТ, 2006. 296 с.
4. Катленд Н. Вычислимость. Введение в теорию рекурсивных функций. – М., 1983.
5. Мальцев А.И. Алгоритмы и рекурсивные функции. – М., 1965.
6. Харви Дейтел и Пол Дейтел Как программировать на C++. Бином, 2006.
7. Громов Ю.Ю., Татаренко С.И. Языки СИ и С++ для решения инженерных и экономических задач. Тамбов: Издательство ТГТУ, 2001.
8. [www.intuit.ru](http://www.intuit.ru)
9. [www.cyberforum.ru](http://www.cyberforum.ru)
10. [www.tehnari.ru](http://www.tehnari.ru)