Ворожцов Артём Викторович
Кандидат физико-математических наук, преподаватель кафедры информатики Московского физико-технического института (МФТИ), тренер сборной команды МФТИ по программированию.

1000-е число Фибоначчи

Последовательность чисел Фибоначчи $F(0)$, $F(1)$, $F(2)$, ... образуется следующим образом:

- Первые два числа равны 1.
- Каждое следующее число равно сумме двух предыдущих.

Из этого определения получаем последовательность 1, 1, 2, 3, 5, 8 ...

Задача. Найти 1000-е число Фибоначчи.

Сформулируем общую задачу подробнее: напишите программу, которая на стандартный вход получает натуральное число n меньше либо равно 1000 и на стандартный поток вывода печатает n-е число Фибоначчи.

Есть два классических метода, которые можно использовать при решении этой задачи:

- метод итераций,
- метод рекурсии.

Метод итераций

Первый метод соответствует тому, что делал бы человек, если бы сам решил вычислить $F(n)$:

Псевдокод 1: Числа Фибоначчи, метод итераций

```
A=1; B=1; // два последних вычисленных числа Фибоначчи
          // A - последнее
          // B - предпоследнее
n-1 раз сделать:
    C = A+B; // вычислить следующее число Фибоначчи
    B = A;   // предпоследнее становится равно последнему
    A = C;   // последнее становится равным только что
             // вычисленному
конец
```

Данный псевдокод отображается в следующую программу на языке Си:

Программа на Си 1: Числа Фибоначчи, метод итераций

```
/* Аргумент: натуральное число n.
 * Возвращаемое значение: n-е число Фибоначчи.
 */
int fib(int n) {
    int i;
    int a = 0, b = 1;
    for (i = 0; i < n; i++) {
        int c = a + b; // вычисляем следующее
        a = b; // обновим пару последних чисел
        b = c; //    Фибоначчи
    }
    return b;
}
int main() {
    int n;
    scanf("%d", &n); // считаем n
    printf("%d\n", fib(n)); // вычислим и выведем fib(n)
    return 0;
}
```

Подобный код напишут многие школьники. Это естественный путь вычисления n-го числа Фибоначчи.

Есть другой путь вычисления чисел Фибоначчи, более естественный, но для некоторых, возможно, ещё непривычный: написать программу, непосредственно соответствующую определению чисел Фибоначчи. *Рекурентно* (то есть через себя) определённые последовательности можно вычислять, используя *метод рекурсии*.

Псевдокод 2: Числа Фибоначчи, метод рекурсии

```
функция Fib(n)
    если n < 2
        вернуть 1;
    иначе
        вернуть Fib(n-1)+Fib(n-1);
    конец
конец
```

Программа на Си 2: Числа Фибоначчи, метод рекурсий

```
/* Аргумент: натуральное число n
 * Возвращаемое значение: n-е число Фибоначчи.
 */
int fib(int n) {
    int i;
    if (n < 2)
        return 1;
    else
        return fib(n-1)+fib(n-1);
}
int main() {
    int n;
    scanf("%d", &n); // считаем n
    printf("%d\n", fib(n)); // вычислим и выведем fib(n)
    return 0;
}
```

В этом коде функция **fib** при вычислении своего значения вызывает сама себя!

В момент, когда вычисления доходят до выражения **fib(n-1)**, происходит временная приостановка вычисления текущей функции и запускается новый процесс вычисления функции **fib**, но уже для меньшего значения аргумента. После окончания вычисления этого значения процесс вычисления текущей функции возобновляется. Вызов функции из самой себя называется рекурсивным вызовом, а такой метод вычисления функций называется методом рекурсии.

Единственная причина, по которой не происходит «зависания» алгоритма, – это наличие проверки **«if (n < 2)» («если n<2»)**, при выполнении которой не происходит рекурсивного вызова.

Итак, мы написали уже две программы для вычисления чисел Фибоначчи, осталось вызвать и напечатать **fib(1000)** и можно отправиться на заслуженный отдых. Нет! Нельзя.

Разработка кода – это лишь малая часть деятельности программиста.

Большую часть времени программист должен тратить на обдумывание, анализ и тестирование алгоритмов. Несложный анализ и тестирование показывают следующее:

Приведённые выше программы *не работают правильно* при всех значениях n, меньше либо равных 1000.

Программа, основанная на методе рекурсии, работает существенно медленнее при больших n (например, при n=50).

Задача. Выведите значения первых 100 чисел Фибоначчи, которые получит приведённая выше программа, основанная на методе итераций. Просмотрите их и убедитесь, что всё нормально. Если обнаружите ошибку в последовательности, попробуйте её объяснить.

Ошибки в этой последовательности будут. Автор решил эту задачу на своём ноутбуке и обнаружил, что **fib(46)=-1323752223**, то есть отрицательное число, чего быть не должно. После этого числа начинается «полный бардак». Из определения чисел Фибоначчи очевидным образом следует, что это возрастающая последовательность положительных чисел. Проблема очевидным образом кроется в том, что для хранения чисел типа **int** система использует конечный набор байт. На большинстве современных архитектур и компиляторов размер типа **int** равен 4 байта. Для вычисления размера типа в Си есть специальный оператор **sizeof**, в частности, **sizeof(int)** даёт 4. Число различных состояний, которые могут принимать 4 байта, равно 2^{32}, то есть 4294967296. Именно столько натуральных чисел находится на промежутке [–2147483648; 2147483648) – это числа, представимые типом **int**.

Задача. Вычислите **fib(40)**, используя программу, основанную на методе рекурсии. Сколько времени это заняло? Объясните причину такого большого времени вычисления.

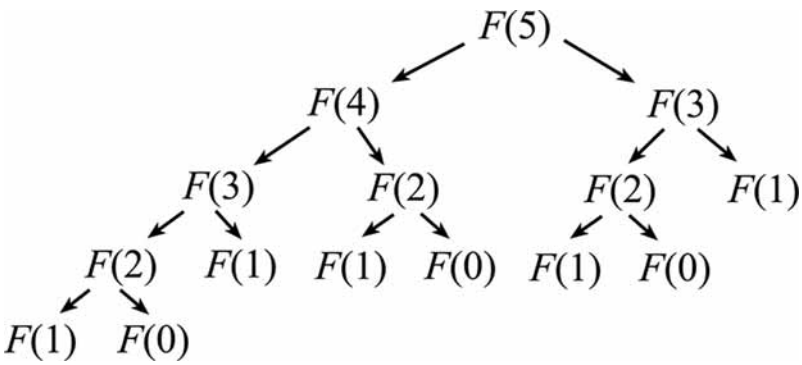

Рис. 1. Рекурсивное дерево вызовов при вычислении числа Фибоначчи F_5

Вычисление **fib(30)** займёт около 10 секунд, а вычисление **fib(40)** более чем в 120 раз больше. Это связано с тем, что алгоритм повторяет вычисления одних и тех же чисел по нескольку раз. Например, вычисление **fib(38)** будет запущено как при вычислении **fib(40)**, так и при вычислении **fib(39)**. На рис. 1 показано *дерево рекурсивных вызовов* вычисления числа **fib(5)**. Стрелочки, исходящие из числа Фибоначчи $F(n)$, показывают, какие два числа Фибоначчи нужно вычислить, чтобы вычислить данное число. Стрелочки не исходят только из чисел $F(1)$ и $F(0)$, так как они по определению равны 1. В нашем алгоритме непосредственно указано, что функция **fib** для $n < 2$ возвращает 1. При вычислении методом рекурсии вычисленное число, по сути, складывается из множества единичек, и из этого непосредственно следует, что время вычисления числа Фибоначчи пропорционально самому числу.

Решите следующие две задачи самостоятельно.

Задача. Покажите, что время работы программы, основанной на методе рекурсии, растёт экспоненциально, а именно пропорционально φ^n, где $\varphi = (1+\sqrt{5})/2$, то есть *при увеличении n на* 1 время работы тоже вырастет примерно в φ раз. Также растут и сами числа Фибоначчи.

Грубое и неполное решение заключается в том, что вы предполагаете, что время растёт примерно как a^n для некоторого a, затем составляете уравнение: «время вычисления числа **fib(n)** равно сумме времён вычисления **fib(n-1)** и **fib(n-2)**». Время вычисления числа Фибоначчи рекурсивным алгоритмом подчиняется тому же реккурентному соотношению, что и сами числа Фибоначчи.

Задача. Покажите, что время работы программы, основанной на методе итераций, растёт линейно, то есть *при увеличении n в два раза* время работы тоже вырастет примерно в два раза.

Число $\varphi = (1+\sqrt{5})/2 \approx 1{,}618$ называется *золотым сечением*. Оно является решением квадратного уравнения $x^2 = x + 1$. Отношение соседних чисел Фибоначчи $F(n) / F(n\text{-}1)$ с ростом становится все ближе и ближе к φ. Золотое сечение считается гармоничной пропорцией между частями и использовалось в архитектуре древней Греции.

Длинная арифметика

В языке Си, если возникает необходимость работать с большими по модулю числами, не представимыми типом **int**, приходится работать с одной из библиотек, реализующих *длинную арифметику*, или реализовывать длинную арифметику самому.

При реализации длинной арифметики каждое число обычно представляется как массив цифр плюс переменная, которая хранит знак числа. Приходится реализовывать алгоритмы сложения, умножения и деления (например, известные всем алгоритмы сложения и умножения столбиком и алгоритм деления уголком). И это уже серьёзная программистская задача. Но в нашем случае (вычисление 1000-го числа Фибоначчи), мы можем ограничиться только сложением и не заниматься отрицательными числами.

Посмотрите программу на языке Си 3.

Программа на Си 3: Числа Фибоначчи, метод итераций, длинная арифметика

```
#include <stdio.h>
// максимальное число цифр
#define max_digs 300
// основание системы счисления
#define q 10
void bigint_add(int *a, const int *b) {
    int i;
    for(i = 0; i < max_digs - 1; i++) {
        a[i] += b[i];
        a[i+1] += a[i]/q;
        a[i] %= q;
```

```
    }
}
// функция печати большого числа
void bigint_print(int *a) {
    int i = max_digs-1; // начинаем со старшего разряда
    // пропускаем лидирующие нули
    while ( a[i] == 0 && i > 0 ) i--;
    for (; i >= 0; i--)
        printf("%d", a[i]);
    printf("\n"); // печатаем символ новой строки
}
int main() {
    int n; // номер искомого числа Фибоначчи
    int i;
    // объявим статические массивы as и bs
    // они будут содержать десятичные цифры
    // двух соседних чисел Фибоначчи A и B;
    // заполним их нулями, кроме первой цифры числа B
    int as[max_digs] = {0,};     // неуказанные элементы
    int bs[max_digs] = {1,0,};   // инициализируются нулём
    // объявим два указателя на эти статические массивы
    int *a = as, *b = bs;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        bigint_add(a, b);
        int *tmp; tmp = b;
        b = a; a = tmp;
    }
    bigint_print(b);
    return 0;
}
```

Задача. Покажите, что число знаков в записи чисел Фибоначчи растёт примерно линейно, то есть длина записи числа $F(2n)$ примерно в 2 раза длиннее, чем запись числа $F(n)$. Старшеклассники, знакомые с логарифмом, смогут записать явное выражение для коэффициента пропорциональности между n и числом знаков в десятичной записи числа $F(n)$ (это не строгая пропорциональность, но с ростом n она становится всё точнее и точнее).

Динамическое типизирование и язык Ruby

Язык Си является высокоуровневым, то есть на нём можно определять свои новые функции, создавать свои библиотеки и переходить на сколь угодно высокий уровень абстракции. Высокий уровень абстракции означает следующее: вместо низкоуровневых команд типа «сложить два числа», «увеличить значение числа на 1», «напечатать символ», программист может мыслить более сложными составными действиями – «нари-

совать окно диалога», «загрузить данные из базы данных», «найти решение уравнения», определив для этих действий соответствующие функции.

Всё это хорошо, но нужно понимать, что язык Си при всём этом машино-ориентированный, то есть программисту предоставляется возможность работать непосредственно с адресным пространством программы (есть указатели) и можно получить доступ к каждому хранимому биту любой сущности (переменной, программного кода функции). Программисту в языке Си даётся полный контроль над хранимыми данными. Наличие высоких прав доступа (контроля) часто влечёт необходимость учитывать и заниматься множеством деталей.

Вычисление чисел Фибоначчи в языке Ruby (читается как «руби») отличается существенно меньшим количеством строчек:

Программа на языке Ruby 1: Вычисление первых 1000 чисел Фибоначчи

```
a,b=0,1
1000.times do
    a,b = b,a+b
    puts b
end
```

Приведённая программа на Ruby верно вычисляет 1000 чисел Фибоначчи от F_1 до F_{1000} включительно. Переменные в языке Ruby не требуют объявления. Первая строка «**a,b=0,1**» означает «создать переменные **a** и **b** и присвоить им значения **0** и **1** соответственно». В этой строке используется *оператор многозначного присваивания* (multiple assignment). Слева от знака «=» стоит пара имён переменных, разделённых запятой, а справа – пара значений, разделённых запятой. По сути, это можно было бы разбить на два действия: «**a=0**» и «**b=1**». Конструкция multiple assignment встречается в программе ещё один раз: «**a,b = b,a+b**». В этом случает разбить его на два действия уже нельзя. Дело в том, что если справа в выражениях есть переменные, которые стоят слева, то начинает играть роль последовательность вычислений. В конструкции «multiple assignment» сначала целиком вычисляется значение выражений, перечисленных справа через запятую, а потом уже происходит присвоение этих значений переменным, которые стоят слева. Это можно использовать, например, для операции обмена местами (swap) двух переменных **x** и **y**:

```
x,y=y,x
```

Идём далее. Строка «**1000.times do**» начинает арифметический цикл, тело которого ограничено строкой «**end**». Цикл повторится ровно 1000 раз. Если прочитать программу на Ruby вслух, получится вполне вразумительный текст на английском языке: «**1000.times do**» читается, как «1000 раз сделать ...». Но не это самое интересное в приведённой программе. Интересно, что программа выдает правильное значение 1000-го числа Фибоначчи. Вот вывод программы:

```
1
2
3
5
8
13
21
34
55
89
144
… (пропущено 987 строк)
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288757033036771142281582183525487718354977018126983635873274260490508715453711819693357974224949456261173348775044924176599108818636326545022364710601205337412127386733911119813937312559876769009190224524532340350 1
```

Сделаем так, чтобы программа получала на вход количество чисел Фибоначчи, которое следует вывести. Для этого нужно просто 1000 заменить на **gets.to_i**:

«**gets.to_i.times do … end**».

Что переводится на русский язык следующим образом:

«Считать строчку (**gets** – это сокращение от get string), преобразовать её в число (**to_i**) и столько раз выполнить следующий блок **do … end**».

В Ruby есть возможность узнать тип (а точнее, класс) любого выражения. Для этого используется метод **class**. Модифицируем нашу программу:

Программа на Ruby 1: Вычисление первых 1000 чисел Фибоначчи

```
a,b=0,1
gets.to_i.times do |i|
    a,b = b, a+b
    puts "#{i}: #{b.class} #{b}"
end
```

Блок «**do … end**» метода **times** может получать аргумент – номер итерации. Этот аргумент объявляется в вертикальных палочках сразу после «**do**» и в данном случае назван как **i**. Данная программа печатает следующий текст:

```
0: Fixnum 1
1: Fixnum 2
2: Fixnum 3
3: Fixnum 5
4: Fixnum 8
5: Fixnum 13
```

```
38: Fixnum 102334155
39: Fixnum 165580141
40: Fixnum 267914296
41: Fixnum 433494437
42: Fixnum 701408733
43: Bignum 1134903170
44: Bignum 1836311903
45: Bignum 2971215073
46: Bignum 4807526976
...
```

Эти данные показывают, что класс переменной **b** автоматически сменился с **Fixnum** на **Bignum**, когда вычисления дошли до итерации 43 (если считать с 0). Можно считать, что класс **Fixnum** соответствует типу **int** языка Си, а класс **Bignum** реализует целые числа сколь угодно большие по модулю.

Возможность менять переменным свой тип во время вычисления называется динамической типизацией.

Подробнее о языке Ruby можно прочитать на страницах Викиучебника:

http://ru.wikibooks.org/wiki/Ruby.

На сайте

http://acm.mipt.ru/bin/view/Ruby

дано множество простых примеров с пояснениями. Официальный сайт языка http://ruby-lang.org содержит ссылки на дистрибутивы под различные компьютерные платформы, в том числе под Windows. Полезно научиться пользоваться текстовым редактором SciTe, который позволяет создавать и редактировать программы на языке Ruby (программы должны иметь расширение **rb**). Запуск программ осуществляется по горячей клавише F5 (панель с выводом находится прямо в окне этого редактора справа, нужно просто её раскрыть, потянув мышкой за правую границу окна влево).

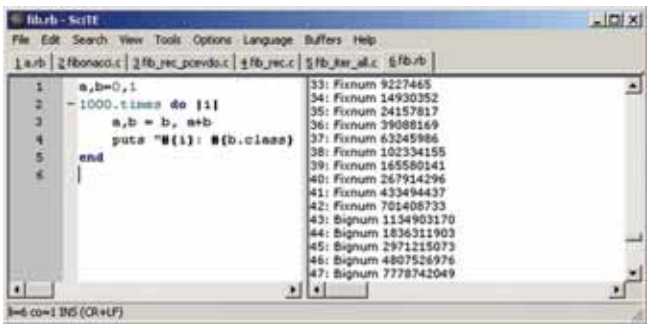

Рис 2. Скриншот программы SciTe.

В нашем журнале ожидается целая серия статей, посвящённая языку Ruby и объектно-ориентированному программированию (ООП). Это один самых интересных языков программирования, который создавался как язык, ориентированный на человека, а не на компьютерную архитектуру. Автор этого языка японец Юкихиро Мацумото (*Yukihiro Matsumoto*, по прозвищу «*Matz*») старался сделать целостный язык, основанный на объектно-ориентированной парадигме, который был бы максимально близок к человеческому образу мышления.