

ΜΑΤΗΜΑΤΗΧΕΣΚΟΕ ΜΟΔΕΛΗΡΟΒΑΝΗΕ

ΠΡΗΕΜΥ Ι ΜΕΤΟΔΥ ΠΡΟΓΡΑΜΜΗΡΟΒΑΝΗΑ

άγεωμέτρητοζ μηδείζ είσιτω

На практике при программировании особого внимания и аккуратности требуют те участки кода, которые описывают длинные повторяющиеся действия. Опишем основные схемы для решения такого рода задач, а также различные приёмы, которые, возможно, будут вообще полезны при реальном программировании.

1. Итерации. В математике *итерация* — результат повторного применения какой-либо математической операции. Так, если

$$y = f(x) \stackrel{\text{def}}{=} f_1(x),$$

то функции

$$f_2(x) = f(f_1(x)), f_3(x) = f(f_2(x)), \dots, f_n(x) = f(f_{n-1}(x)), \dots$$

называют соответственно второй, третьей, \dots , n -й итерациями. Переход от функции $f_1(x)$ к $f_2(x)$, $f_3(x)$, \dots — *итерированием*. Итерации широко применяются в самых различных алгоритмических процедурах.

Программно итерирование реализуется посредством *циклов* различных типов и состоит в том, что строится преобразование T , которое последовательно применяется, начиная с некоторого начального элемента x_0 , до тех пор, пока не будет получен элемент x_n с требуемыми свойствами:

$$x_1 = T(x_0), x_2 = T(x_1), \dots, x_n = T(x_{n-1}).$$



2. Рекурсия — способ задания функции или процедуры, при котором её значения от некоторых аргументов выражаются через значения этой функции от других аргументов.

В программировании рекурсии соответствует ситуация, когда программа вызывает сама себя, либо непосредственно, либо через другие программы. Применение рекурсии становится возможным, когда при анализе некоторую задачу удалось свести к точно такой же подзадаче, но с другими исходными данными. Рассмотрим конкретные примеры.

Пример 1. Вычисление факториала. По определению

$$n! = n(n - 1)!, \quad 0! = 1.$$

```
function Factorial( n: word ): LongInt;  
begin  
    if    ( n <= 1 )  
    then result := 1  
    else result := n*Factorial( n - 1 ); // рекурсивный вызов;  
end;
```



назад

закр.

Пример 2. Обращение строки символов. Пусть требуется введённую строку произвольных символов **записать задом наперёд**.

```
procedure Reverse;           // перед вызовом про-
var letter : Char;           // цедуры нужно
begin                         // ввести с консоли
    if ( not eoln ) then     // строку символов;
    begin
        read( letter );
        Reverse;              // рекурсивный вызов;
        write( letter );
    end
end;
```

Обычно в языках программирования высокого уровня нет никаких ограничений на рекурсию, но нужно принять во внимание, что любой рекурсивный вызов приводит к образованию новой копии локальных объектов подпрограммы, которые динамически размещаются в памяти, в программном стеке, который будет тем больше, чем длиннее цепочка вызовов. Рекурсивная форма записи обычно выглядит изящнее и компактнее, чем итерационная, но, как правило, работает медленнее и может вызвать переполнение стека, если не принять специальных мер предосторожности.

3. Проектирование цикла с помощью инварианта. Это — разновидность метода итераций, при котором связь между меняющимися в теле цикла параметрами выражается в виде неизменного условия (инварианта).

Практически необходимо:

- Придумать стратегию решения задачи.
- Чётко сформулировать условие окончания цикла.
- Описать взаимосвязи между всеми параметрами, изменяющимися в ходе выполнения цикла в виде инварианта.

Пример 3. Средневековая задача. В куче находятся 50 спичек. Два игрока по очереди могут брать из кучи от 1 до 6 спичек. Противники видят все ходы друг друга. Выигрывает тот, кто возьмёт последнюю спичку. Составить программу, которая будет всегда выигрывать (компьютер делает первый ход).

Программа в действии.

```

const Pile = 50;
var  comp, user, cash: Integer;
BEGIN
cash := Pile;
writeln( ' Pile  Computer User' );
  while ( cash > 0 )  do
    begin
      comp := 1;
      while ( ( cash - comp ) mod 7 <> 0 ) do      // ход
        inc( comp );                             // компьютера;
      write( ' ', cash, ' ', comp, ' ');
      if ( cash = comp ) then
        begin
          writeln;  write('I win!, hit <Enter> to exit');
          readln;  break;
        end;
      repeat                                     // ход
        read( user );                             // пользователя;
      until ( user <= 6 ) and ( user >= 1 );
      cash := cash - comp - user;
    end; // while ( cash > 0 )...
  readln;
END.

```

4. Вычисление инвариантной функции. Пусть для функции f существует преобразование T , такое, что

$$f(T(x)) = f(x), \text{ для любых } x,$$

тогда функцию f называют *инвариантной*.

Практически вычисление инвариантной функции сводится к нахождению преобразования T , которое переводит элемент x , для которого ищется $f(x)$, в некоторый другой элемент x_0 , для которого значение функции $f(x_0)$ легко находится.

Пример 4. Наибольший общий делитель (НОД) чисел a, b .

Здесь $x = (a, b)$. Очевидно, что

$$\text{НОД}(a, 0) = a, \quad \text{НОД}(0, b) = b,$$

и, таким образом, если a или b равно 0, то легко можно вычислить $\text{НОД}(a, b) = a + b$. Следовательно, осталось придумать преобразование T , «сохраняющее» НОД. Поскольку

$$\text{НОД}(a, b) = \text{НОД}(a - b, b) = \text{НОД}(a, b - a),$$

то искомое преобразование определим так

$$T(a, b) = \begin{cases} (a - b, b), & \text{если } a \geq b > 0; \\ (a, b - a), & \text{если } b \geq a > 0. \end{cases}$$

Теперь, вместо того, чтобы находить общие делители и определять среди них наибольший, можно значительно упростить программный код, к тому же, полностью избежав операций деления, весьма накладных по времени. Таким образом, помимо того, что программный код стал проще и яснее, он ещё и быстрее будет выполняться.

```
function GCD( a, b: word ): word;  
begin  
    while ( ( a > b ) AND ( b > 0 ) ) do  
        if ( a >= b )    then a := a - b  
                        else b := b - a;  
  
result := a + b;  
end;
```


5. Уменьшение объёма занимаемой программой памяти. Одним из самых эффективных способов экономии памяти является понижение размерности (в идеале — полное устранение) используемых в программе массивов.

Например, чтобы вычислить 50-е число Фибоначчи F_{50} , не обязательно заводить массив целых размерности 50, т. к. 49 предыдущих чисел не нужны и будут зря занимать память. Числа Фибоначчи задаются рекуррентным соотношением

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = F_2 = 1$$

и, в этом случае, легко можно обойтись вообще без массивов:

```
function Fibonachi( number: word ): Int64;  
var F, predF, predpredF : Int64;  
    i                     : word;  
begin  
predpredF := 1;  
predF      := 1;  
F          := 1;  
    for i := 3 to number do  
        begin  
            F := predpredF + predF;  
            predpredF := predF;  
            predF := F;  
        end;  
result := F;  
end;
```

5. Уменьшение времени выполнения программ. Для подавляющего большинства программ основная часть времени их исполнения расходуется на выполнение небольших участков кода, который называют *критичным* или *внутренним* циклом. Именно с этих программных блоков нужно начинать оптимизацию (и, в большинстве случаев, ими можно и ограничиться). Большая часть кода, обычно до 80–90%, не нуждается ни в какой оптимизации.

Как правило, этот код действительно является циклом в смысле синтаксиса языков программирования. Если программа содержит несколько вложенных друг в друга циклов, то внутренний — это цикл, принадлежащий всем циклам. Если имеется несколько циклов, не содержащих вложенных циклов, то нужно оценить время их работы, подсчитав количество повторений каждого из них.

Пример. Рассмотрим следующий блок операторов

```
for j := 1 to N do // цикл 1;
begin
  for k := 1 to M do
  begin
    // тело цикла 2;
  end;
  for l := N downto j do
  begin
    // тело цикла 3;
  end;
end;                // конец цикла 1;
```



Время работы программы, помимо всего прочего, зависит от характеристик компьютера («железа»), на котором программа выполняется. Поэтому заранее точно установить время работы того или иного алгоритма практически невозможно и обычно используют приближённые оценки.

Пусть время одного прогона тела циклов 2 и 3 приближённо равно, соответственно, T_2 и T_3 (если это время не является постоянным, например, в теле цикла имеется условный переход или вызов подпрограммы, то нужно взять среднее время). Операторы тела цикла 2 выполняются MN раз, а цикла 3 — всего $N - j + 1$ раз при каждом прогоне цикла 1. Следовательно, общее количество итераций цикла 3 равно

$$\sum_{j=1}^N (N - j + 1) = \frac{1}{2} N(N + 1).$$

А время работы всего блока

$$MN T_2 + \frac{1}{2} N(N + 1)T_3.$$

Какой из циклов в данном случае будет критичным зависит от конкретных значений величин M , N , T_2 , T_3 . В более сложных случаях можно воспользоваться специально предназначенными для этих целей программами — профайлерами (profiler).



назад

закр.

После того, как определены критические участки кода, можно приступать к оптимизации. Следующие советы могут помочь повысить быстродействие:

- ☞ По возможности, старайтесь исключить операции с плавающей точкой, так как они выполняются много медленнее, чем манипуляции с целыми величинами.
- ☞ Операции целочисленного умножения и деления лучше заменить сложением и вычитанием, или, ещё лучше, *сдвигами*. Например, вместо: $k := k * 11$; можно использовать $k := k \text{ shl } 3 + k \text{ shl } 1 + k$;, поскольку $11 = 2^3 + 2 + 1$.
- ☞ Так как перед каждым вызовом процедуры или функции её параметры помещаются в стек, а затем изымаются оттуда, то процедуры лучше спроектировать так, чтобы они имели как можно меньше параметров.
- ☞ Если есть возможность не выполнять какие-либо расчёты в реальном времени, можно их предварительно подготовить.
- ☞ В самых критических случаях можно использовать язык Assembler, программирование на котором более трудоёмко по сравнению с языками высокого уровня, но обеспечивает максимальную скорость выполнения программы.

Следует отметить, что самым лучшим способом оптимизации является *выбор более эффективного алгоритма* — пузырьковая сортировка будет всегда идти медленно, несмотря на все программистские ухищрения [4].

6. Тестирование и отладка программ.

Ошибки в программах могут быть трёх видов:

- Синтаксические ошибки, вызванные тем, что программист нарушил правила языка программирования.
- Ошибки периода выполнения программы, когда синтаксически правильная программа работает не верно (или вообще не работает), например, ошибка «деление на 0», или «бесконечный цикл».
- Логические ошибки, связанные с тем, что неправильно запрограммирован алгоритм, или сам алгоритм неверен.

Синтаксические ошибки *автоматически* обнаруживаются при трансляции, причём компилятор выдаёт соответствующее сообщение о том, где произошла ошибка и возможных её причинах.

Ошибки двух других типов более неприятны и коварны, т. к. они могут проявляться не при всех наборах входных данных, а только при некоторых, или при возникновении каких-то конкретных обстоятельств. Эти ошибки устраняются при помощи отладки и тестирования программ.

В настоящее время не существует теории тестирования и отладки, применение которой гарантировало бы выявление всех возможных ошибок. Отсюда следует, что любая более или менее сложная программа *почти неизбежно* содержит ошибки. Некоторые из них можно устранить посредством отладки, при этом, быть может, внося в программу новые ошибки. В связи с этим очевидна необходимость написания программного продукта с учётом последующего *сопровождения*.



назад

закр.

Отладка и тестирование программы — искусство, где нет строго определённых правил и где реально наиболее ярко проявляется квалификация программиста.

Следующие три простых совета могут значительно облегчить отладку:

- ☞ Лучше сначала отладить процедуры, функции, объекты и т. д. по отдельности, а затем уже проверить их взаимодействие в программе в целом.
- ☞ При отладке особое внимание нужно уделить наиболее потенциально уязвимым участкам кода, таким, как *циклы* и *ветвления*, *рекурсивные функции*, а также всем программным объектам, связанным с работой с динамической памятью, *указателями*, пользовательским вводом данных и т. п.
- ☞ При необходимости можно воспользоваться отладчиком (debugger), который сейчас имеется в составе почти любой IDE, и который позволяет пошагово выполнять программу, от оператора к оператору, расставлять точки прерывания и т. д. Причём при отладке дебаггером на каждом шаге выполнения будет доступна вся информация о текущем состоянии всех элементов программы.

Под *тестированием* понимается выполнение программы с набором таких входных данных, при которых результат работы программы известен заранее. Цель тестирования — определение логических ошибок в программе. Конечно, прохождение набора тестов *не гарантирует* логическую правильность в достаточно сложных программных проектах. С другой стороны, провал любого теста *всегда* означает, что программный код нуждается в исправлении. Логическую правильность программы обычно удаётся доказать только в относительно простых случаях.

При тестировании рекомендуются следующие правила:

- ☞ Составление тестов лучше проводить параллельно с разработкой программы.
- ☞ Сравнение эталонных и полученных значений предпочтительнее проводить в ходе самого теста.
- ☞ Программа должна не только правильно работать при корректных входных данных, но и уметь обрабатывать недопустимые для данной программы входные параметры.
- ☞ Все тесты нужно тщательно анализировать. При существенном изменении программного кода, скорее всего, понадобится модифицировать и набор тестов.



Рассмотрим в качестве простого примера вычисление выражения

$$\frac{e^x - 1}{x}.$$

Тестирование функции

```
function exp_div_x( x: double ): double;  
begin  
  result := ( exp( x ) - 1 )/x;  
end;
```

даст следующие результаты

№ теста	значение x	результат вызова функции
1	1.0	1.71828182845905 E+0000
2	0.0	Invalid floating point operation
3	-1.0	6.32120558828558 E-0001
4	1000.0	Floating point overflow
5	1.0 E-3	1.00050016670834 E+0000
6	1.0 E-1000	Invalid floating point operation

Причина неправильной работы функции в тесте №2 — деление на нуль. Поскольку

$$\lim_{x \rightarrow 0} \frac{e^x - 1}{x} = 1,$$

то эту логическую ошибку можно исправить так:

```
function exp_div_x( x: double ): double;  
begin  
    if ( x = 0.0 ) then result := 1.0  
    else result := ( exp( x ) - 1 )/x;  
end;
```

Сбои в тестах №4 и №6 вызваны выходом за допустимый диапазон изменения величины типа double, который в Delphi составляет

$$5,0 \times 10^{-324} \dots 1,7 \times 10^{308}.$$

Для решения таких проблем можно, например, воспользоваться механизмом *обработки исключительных ситуаций*, который применяется в Object Pascal, C++, Java и других языках [2].

```

function exp_div_x( x: double ): double;
begin
    if    ( x = 0.0 ) then result := 1.0
    else  begin
            try
            // если при выполнении
            result := ( exp( x ) - 1 )/x;
            // возникла исключительная ситуация, то
            except
                On EOverflow do
                begin
                // если используется GUI Windows, то
                // можно использовать процедуру
                // ShowMessage('result is too large !');
                // из модуля Dialogs,
                // а для консольных приложений:
                write(' result is too large! ');
                result := 0.0;
                end;
            end;
        end;
    end;
end;

```

Теперь в случае вещественного переполнения будет выдано сообщение: «result is too large!» («результат вычисления слишком велик!») и функция вернёт 0,0. Возвращаемое значение сознательно выбрано невозможным для данной функции (она обращается в нуль только при $x \rightarrow -\infty$ и этого, конечно, не может произойти при вычислениях), чтобы по этой величине можно было впоследствии в вызывающем функцию блоке соответствующим образом обработать эту ситуацию. Применение программных исключений особенно удобно для глубоко вложенных блоков.

✌ С приобретением определённого опыта у программиста появится возможность в простых и средней сложности программных блоках обходиться без тестирования или, по крайней мере, свести его к некоторому необходимому минимуму.

6. Работа с вещественными числами. Основная трудность при работе с вещественными числами является следствием того факта, что любая переменная в памяти компьютера может принимать только конечное число значений, тогда как даже на конечном отрезке вещественной прямой содержится бесконечно много действительных чисел.

Вещественные числа в компьютере представляются в т. н. *формате с плавающей точкой* — отдельно хранится *мантисса* M , $|M| \leq 1$, и *порядок* p . Число вычисляется по формуле

$$Ma^p.$$

Для записи мантиссы всегда используется фиксированное количество цифр, диапазон изменения порядка также ограничен. Поэтому машинное представление вещественных чисел имеет следующие, важные для практического программирования, особенности:

- a.** в компьютере невозможно представить очень большие и очень малые по абсолютной величине действительные числа;
- b.** вещественное число, даже и попадающее в допустимый диапазон, может быть записано с некоторой погрешностью.



назад

закр.

Из свойства **a** непосредственно следует, что существует величина, называемая *машинным нулём*, т. е. такое число ε , что в компьютерных расчётах для всех чисел x , таких что $0 < x < \varepsilon$, выполняется $1.0 + x = 1.0$. Другими словами, все вещественные числа, меньшие ε , компьютер будет «воспринимать» как нуль. Величина машинного нуля зависит от типа x .

Из свойства **b** вытекает неизбежность ошибок округления, что можно показать на простом **примере**:

```
program real_error;  
{ $APPTYPE CONSOLE }  
uses SysUtils;  
const x = 17.0;  
begin  
  writeln(' ERROR = ', x - sqrt(x)*sqrt(x) );  
  readln;  
end.
```

Кроме этого, нужно всегда помнить, что погрешность при суммировании чисел складывается из погрешностей слагаемых и погрешности выполнения операции сложения. Если вначале складывать большие по модулю числа, то можно получить неправильный результат. Рассмотрим пример суммирования первых N членов ряда, задающего ζ -функцию от 2

$$\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}.$$



При суммировании в различном порядке можно получить различные **результаты**.

```
program floatPrj_1;  
{ $APPTYPE CONSOLE }  
uses SysUtils;  
  
const N          = 100000;  
      dzeta2      = PI*PI/6;  // "точное" значение dzeta(2);  
var k            : integer;  
    S_to, S_downto : real;  
  
begin  
  S_to := 0;  
    for k := 1 to N do           // суммирование по возрастанию k;  
      S_to := S_to + 1/k/k;  
  S_downto := 1/N/N;  
    for k := N - 1 downto 1 do   // суммирование по убыванию k;  
      S_downto := S_downto + 1/k/k;  
  writeln( 'dzeta(2) = ', dzeta2 );  
  writeln( 'S_downto = ', S_downto );  
  writeln( 'S_to      = ', S_to );  
  readln;  
end.
```

Список литературы

1. В. И. Зенкин. Практический курс математического и компьютерного моделирования. Учеб. пособие. Калининград: изд. РГУ им. Канта, 2006.
2. В. В. Фаронов. Delphi 3. Учебный курс. М.: Нолидж, 1998.
3. Д. Тейлор, Дж. Мишель, Дж. Пенман, Т. Гоггин, Дж. Шемитц. Delphi 3: библиотека программиста. Спб, 1996.
4. Д. Кнут. Искусство программирования для ЭВМ. М., Т. 1, Т. 2: Получисленные алгоритмы, 1977; Т. 3: Сортировка и поиск, 1978.



назад

закр.