Порядок решения олимпиадной задачи.

(Материалы от Новожиловой Валентины Ивановны)

Всякая алгоритмическая задача решается в несколько этапов:

1. Следует очень внимательно прочитать условие задачи и постараться его правильно понять. К тексту условия задачи следует подходить формально, т. е. понимать его буквально, а не так, как покажется при его поверхностном чтении. Если с точки зрения формальной логики или, по вашему мнению, условие задачи можно трактовать неоднозначно, то следует задавать вопросы. На олимпиадах есть правило: вопрос задается в письменной форме и формулируется так, чтобы ответить на него можно было одним из слов: «да», «нет».
2. Выделить входные данные и четко определить выходные данные. Для входных данных задать область определения, для выходных данных теоретически оценить область значений. Это позволит вам правильно задать тип этих данных.
3. Решить задачу. При этом следует подумать над подходами к решению задачи, для чего ответить себе на ряд вопросов и проделать ряд действий.
    - Определить, относится ли данная задача к знакомому вам классу задач, можете ли вы выделить известные вам алгоритмы для ее решения или решение придется искать “с нуля”.
    - Попытаться найти на бумаге (именно на бумаге!) точное решение, возможно только для малых размерностей. Такой подход зачастую позволяет обнаружить закономерности, которые затем можно попытаться распространить и на общий случай. Отобразить на бумаге принципиально различные случаи, в том числе, и вырожденные. Это поможет при составлении тестов для самостоятельного тестирования написанной программы.
4. Записать алгоритм решения задачи в виде: математической модели, описания последовательности действий на русском языке (псевдокод), блок-схемы.
5. При записи алгоритма будут выявлены данные, которые возникают во время решения задачи, их называют рабочими данными, им тоже нужно правильно назначить тип с учетом теоретической оценки их области значений. Входные, выходные и рабочие данные вместе с их типами называются моделью данных задачи.
6. Анализ алгоритма решения задачи. Следует разделить алгоритм решения задачи на части, которые будут выполняться один или более раз. Каждая такая часть алгоритма должна делать только одну работу по решению задачи, иметь свои входные и выходные данные. Такая часть алгоритма оформляется в виде процедуры или функции.
7. Для каждой процедуры или функции написать «заглушку», которая включает заголовок с описанием входных и выходных данных процедуры (или функции), рабочие данные и ключевые слова begin end; Тело процедуры пусто, пока еще не написано.
8. Написать головную программу, состоящую из вызова «заглушек». Отладить ее в среде Турбо (или Borland) Паскаль. При этом будут выявлены и исправлены ошибки по взаимодействию процедур и функций и описанию данных. Такая программа называется функциональным описанием решения задачи (или функциональной спецификацией). Она правильна с точки зрения синтаксиса и семантики алгоритмического языка, но не делает никакой работы. Этапы решения задачи с 1 по 6 имеют общее название – проектирование программы.
9. При проектировании программы следует особое внимание обратить на инициализацию (задание начальных значений) переменных. Входные данные задаются в текстовом файле, имя этого файла стандартное «Input.txt». Имя выходного файла, в который записываются результаты решения задачи, также стандартное

«Output.txt», тип файла – текстовый. Следует помнить, что в среде Турбо Паскаль обнуляются при запуске программы только глобальные переменные, локальные данные процедур и функций не обнуляются. Незнание этого факта приводит к ошибкам при повторных запусках программы, т.к. тот «мусор» из нулей и единиц, который остался в памяти после предыдущего выполнения программы, интерпретируется компилятором, как значения локальных переменных. При этом в программе нужно предусмотреть процедуру ввода, вывода и инициализации данных.

10. В профессиональном программировании после ввода входных данных производится программная проверка их на корректность, т.е. на соответствие области определения каждого данного. Проверка на корректность в последнее время на олимпиадах практически не производится. Хотя на олимпиадах прошлых лет такая практика и существовала. Объясняется это тем, что количество предлагаемых на одном туре задач и их сложность возросли, и во время олимпиады важнее определить не степень профессиональной пригодности участника как программиста, а его способность решать те или иные задачи. Отмена проверки входных данных на корректность позволяет участникам больше времени потратить на обдумывание алгоритмов решения задач и их реализацию.
11. Написать тела процедур и функций, провести их отладку. В разделе `OPTIONS\enviroment\preferences` среды программирования Turbo Pascal полезно установить параметр `Auto save [X]Editor Files` (автоматическое сохранение редактируемых файлов). Это гарантирует сохранение текста программы на жестком диске при каждом ее запуске. Если программа зависнет и среду программирования придется запускать заново, то результат последнего редактирования будет сохранен. Во время тура не следует забывать время от времени запоминать сделанные изменения в тексте программы.
12. Вывести выходные данные в файл. Следует помнить, что выходной текстовый файл должен быть закрыт обязательно, иначе при проверке он окажется пуст. Это особенность текстовых файлов. Информация в него из буфера переносится только, если буфер полон, или при закрытии файла.

Если вы не придумали эффективного решения задачи, то запрограммируйте его по-простому: например, с помощью полного перебора или простой эвристики (приближенного решения, в ряде случаев дающего точный ответ). Если и это сложно, то упростите себе задачу, то есть отбросьте условия, которые вам мешают или добейтесь, чтобы программа проходила на самых простых, например, вырожденных тестах (большинство параметров равны 0 или 1). Аналогично следует поступить с задачами, на решение которых у вас не хватило времени.

Требования к оформлению решения олимпиадной задачи.

В самом начале полезно набить приведенную ниже универсальную заготовку для решения олимпиадной задачи (она представляет собой работающую программу). В образце приведены процедуры без параметров. Таких заготовок следует сделать столько, сколько задач предлагается для решения, чтобы больше не отвлекаться на подготовительную работу.

```
var
  {Описание глобальных данных}
procedure readdata;
begin
  assign(input,'Input.txt');
  reset(input);
  {ввод входных данных их файла}
end;
procedure outdata;
begin
  assign(output,'');
  rewrite(output);
  {вывод выходных данных в файл}
  close(output);
end;
procedure initial;
begin
  {инициализация глобальных переменных}
end;
procedure run;
begin
 {одна из процедур для решения задачи}
end;
begin {головная программа}
 readdata;
 initial;
 run;
 outdata;
end.
```

Директивы компилятора. В текст программы полезно вставлять директивы для компилятора, они размещаются перед описанием глобальных данных.

Для отладки программы наиболее значимыми являются ключи `D+` и `L+`. Благодаря этим ключам можно проводить пошаговую отладку программы из среды программирования.

Еще одна пара ключей — `E+` и `N+` необходима, если программа использует вещественные числовые типы данных, арифметические и логические операции над которыми производятся с помощью сопроцессора, а именно: `comp`, `single`, `double` и `extended`. Без упомянутых ключей программа, использующая эти типы, просто не будет компилироваться. Строго говоря, ключ `E+` был необходим лишь для компьютеров, сопроцессор в которых отсутствовал, и операции над вещественными числами осуществлялись с помощью так называемого эмулятора — программы, реализующей эти операции только через команды процессора. Однако все процессоры класса Pentium, а именно ими и оснащено большинство современных компьютеров, содержат встроенный сопроцессор, который будет использоваться при выполнении программы, написанной на языке Turbo Pascal, в случае установки ключа компилятора `N+`. То есть такая комбинация ключей делает программу “переносимой”, не зависящей от компьютера, на котором она исполняется.

Установка ключа I+ приводит к тому, что при работе программы строго контролируется соответствие значений, поступающих на вход программы, типам переменных, которым эти значения будут присвоены. В случае несоответствия типов, например, при вводе символа вместо числа или вещественного числа вместо целого, выполнение программы будет прервано.

Совершенно незаменима при отладке программы следующая установка ключей компилятора: R+ и Q+ (последний ключ появился лишь в версии Turbo Pascal 7.0). Они позволяют контролировать во время выполнения программы “выход за границу массивов” и “выход за границу допустимого диапазона значений” при операциях над целочисленными переменными. То есть при работе с массивом контролируется попытка обращения к несуществующему элементу. Если во время выполнения операции (арифметической или присваивания) над целыми числами результат, в том числе и промежуточный, не является допустимым для соответствующего типа, то выполнение программы прерывается. При этом ключ R+ отвечает за корректную работу с массивами и присваивание только допустимых значений переменным типа byte и shortint, а Q+ — за корректное выполнение арифметических операций над целыми числами в рамках соответствующих типов. При отсутствии такого контроля поиск ошибки может быть затруднен тем, что промежуточные вычисления чаще всего производятся в целом типе данных в разрядной сетке наибольшего размера (обычно 32-разрядном) и лишь при присваивании полученного значения переменной меньшего размера лишние старшие разряды отбрасываются. Как следствие, отладочная информация о значении арифметического выражения и его результат могут не совпадать.

Рассмотрим это на примере следующей простой программы:

```
{$Q-}
var a:integer;
begin
  a:=1*2*3*4*5*6*7;
  writeln('7!=',a);
  a:=a*8;
  writeln('8!=',a)
end.
```

Если после получения переменной **a** своего первого значения, равного 7!, мы посмотрим в отладчике значение выражения **a*8**, то оно будет равно 40320, а в результате второго присваивания значение **a** окажется равным –25216, что и будет выведено на экран.

Это очень распространенная ошибка. Часто встречается при решении графических задач, если нужно вычислить расстояние между двумя точками, координаты которых заданы целыми числами. Формула программируется точно так, как записывается в математике: r:=sqrt(sqr(x1-x2)+sqr(y1-y2)); Ошибка возникает при извлечении корня из отрицательного числа при входных данных x1=500; x2=0; Это наведенная ошибка, если переменные x1 и x2 описаны как integer, то под результат вычисления корня из разности координат будет отведено ровно столько места, как и под тип integer. При переносе значения из «быстрой» памяти в оперативную память старшие разряды будут отброшены, в знаковом бите может оказаться как 0, так и 1. Если окажется 1, то число интерпретируется как отрицательное, представленное в дополнительном коде.

После того как программа отлажена, то ряд ключей компилятора следует заменить. Сдавать программу на тестирование следует с ключами D-,I-,L-,R-,Q-. Объясняется это двумя причинами. Во-первых, при отмене ряда проверок и отсутствии отладочной информации программа будет выполняться быстрее. Во-вторых, если часть ошибок при отладке не устранена, но не является для работы программы фатальной (например, обращение к несуществующему элементу массива может не влиять на правильное формирование реальных его элементов), то программа может вполне успешно пройти процедуру тестирования. Если же проверка корректного обращения с данными в

исполняемом коде остается, то, скорее всего, на большинстве тестов выполнение программы будет прервано досрочно и результат ее работы просто не будет получен.

Ввод и вывод данных

В большинстве олимпиадных задач ввод данных предлагается производить из текстового файла. У ряда участников олимпиад такое техническое требование вызывает некоторое затруднение. Покажем, как можно быстро преодолеть все сложности работы с файлами и сделать при этом как можно меньше ошибок.

Как видно из примера, приведенного выше, для организации ввода данных из текстового файла наличие файловой переменной типа `text` в программе вовсе не обязательно. Более того, перенаправление стандартного потока ввода `input` и потока вывода `output` в файлы является более удобным при программировании и избавляет от ряда ошибок. После подобного перенаправления ввод данных из файла осуществляется с помощью обычных процедур `read` и `readln`, а вывод — с помощью `write` и `writeln` без указания в качестве первого параметра имени какой-либо файловой переменной. Такой подход избавляет от типичной ошибки при работе с текстовыми файлами, которая заключается в том, что в некоторых обращениях к процедурам ввода или вывода имя файловой переменной оказывается пропущенным. Вторая типичная ошибка при работе с файлом, открытым на запись — отсутствие в конце программы команды, закрывающей файл. В таком случае, создаваемый программой выходной файл скорее всего окажется пустым. Дело в том, что реальная запись данных на жесткий диск происходит или при выполнении уже упомянутой команды `close` или, если количество выводимой информации велико, в момент переполнения буфера оперативной памяти, предназначенного для ускорения работы с файлами. Но и от этой ошибки работа со стандартным потоком вывода спасает. Дело в том, что файл `output` закрывается при окончании работы программы автоматически, вне зависимости от наличия или отсутствия команды `close(output)`.

Рассмотрим теперь полезные приемы программирования ввода данных различных типов.

1) Ввод числовых данных.

Начнем с описания считывания из текстового файла или консоли (клавиатуры), которая с точки зрения программы также является текстовым файлом, числовых данных. В условии задачи ввод большого количества чисел может быть задан двумя способами.

В первом способе сначала предлагается ввести количество чисел, а уж затем сами эти числа. В данном случае при программировании сложности не возникает.

Во втором же случае количество чисел приходится определять в процессе их считывания. Пусть, например, для каждой строки входного файла требуется найти среднее арифметическое для чисел, расположенных в ней, количество чисел в каждой из строк и количество строк при этом неизвестно. Наиболее простым и правильным будет следующее решение такой задачи:

```
while not seekeof do begin {пока не конец файла}
 n:=0; {обнулить счетчик чисел в строке входного файла}
 s:=0; {обнулить переменную для хранения суммы чисел}
 while not seekeoln do begin {пока не конец строки}
  read(a); {читать число}
  s:=s+a;  {увеличить сумму}
  n:=n+1   {увеличить количество прочитанных чисел}
 end;
 {readln;}
 if n>0 then writeln(s/n:0:2) else writeln
end;
```

Обычно применяемые в таких случаях функции `eof` и `eoln` заменены функциями `seekeof` и `seekeoln` соответственно. Имя файловой переменной при этом опускается, что возможно для стандартного потока ввода, даже после перенаправления его в файл. Только при таком способе ввода чисел не возникают ошибки в работе программы, связанные с наличием пробелов в конце строк и пустых строк в конце файла. Для корректного использования стандартной функции `eof` требуется, чтобы признак конца файла стоял непосредственно после последнего числа в файле. То же требование относится к признаку конца строки при использовании функции `eoln`. Несмотря на то, что числа расположены в различных строках файла, процедуру `readln` при вводе именно чисел можно не использовать (в приведенном примере она взята в комментарий, снятие которого не изменит работу программы).

2) Ввод текста по строкам.

Если во входном файле находится текст, размер которого не известен, то поступать следует по-другому. Использование `seekeoln` может привести к ошибке, так как в тексте пробел уже является значимым символом. Служебные символы, обозначающие конец строки в файле и перевод на новую строку (их коды 13 и 10), не могут считаться частью текста и не должны анализироваться алгоритмом его обработки. Поэтому, если известно, что длина каждой строки текстового файла не превосходит 255 символов, то удобнее всего считывание производить с использованием переменной типа `string`:

```
while not eof do begin {пока не конец файла}
 readln(S); {читать строку}
 if s<>'' then {если строка не пуста, то обработать строку S}
end;
```

В этом примере использование `readln`, а не `read` является уже принципиальным.

3) Ввод текста по символам.

Если же ограничения на количество символов в одной строке нет, то считывание следует производить посимвольно. Причем на Всероссийской олимпиаде отсутствие такого ограничения означает, что при тестировании программы будут тесты, содержащие очень длинные строки, а на школьной или районной олимпиаде, — скорее всего, организаторы такое ограничение забыли включить в текст условия, а все тесты будут состоять все-таки из коротких строк. Пример посимвольного считывания текста из файла:

```
while not eof do begin {пока не конец файла}
 n:=0;{обнуление счетчика символов}
 while not eoln do begin {пока не конец строки}
  read(c);{чтение символа}
  {запись символа c в массив или его обработка}
  n:=n+1;{увеличить счетчик символов}
 end;
 readln;{переход на другую строку}
 if n>0 then {обработка строки} else {строка пустая}
end;
```

Именно использование оператора `readln` позволяет и в данном случае автоматически исключить из рассмотрения символы перевода строки.

Последний вариант считывания данных относится к случаю смешанной информации, то есть в файле присутствуют как числа, так и символы или последовательности символов. Формат такого файла обычно определен заранее, поэтому считывание можно организовать сразу в переменные соответствующих типов. Считывание информации в одну строковую переменную, а затем выделение из нее отдельных элементов и их преобразование, делает

программу более громоздкой и зачастую требует дополнительной отладки, а это отнимает время .

Пример. Пусть в каждой строке файла записана фамилия человека, затем через пробел его год рождения и через пробел — его пол, обозначенный одной буквой. Приведем фрагмент программы, считывающий данные описанного формата из файла сразу в переменные соответствующих типов:

```
while not seekeof do begin {пока не конец файла}
 read(c); {читать символ}
 S:='';   {задать пустую строку }
 while c<>' ' do begin {пока не пробел формируем строку с
фамилией}
  S:=S+c; {к строке присоединяем символ}
  read(c) {читаем следующий символ}
 end;
 read(n);{считываем число - год рождения}
 readln(c,c);{считываем пол}
 {обработка считанной информации}
end;
```

При считывание символа, обозначающего пол человека, предварительно следует пропустить пробел, который ему предшествует. Именно поэтому считываются два символа, а не один, и значение первого символа (пробела) теряется при считывании второго (значения пола).

4) Запись результатов в файл.

Во время записи результатов работы программы в файл обычно проблем не возникает. Ошибки в формате вывода могут быть связаны с отсутствием разделителей (пробелов или символов перевода строки) между выведенными в файл числами или с формой записи вещественного числа. Если вещественные типы данных используются для работы с целыми числами, то выводить результат следует так: `writeln(x:0:0)`. Такой прием используется для получения более точного результата. При этом количество значащих цифр более чем в два раза превосходит количество значащих цифр в максимальном целом типе. Если результатом работы программы является произвольное вещественное число, то формат его вывода обычно оговорен в условии задачи.

Пример: Если требуется получить в дробной части три цифры, то печать можно производить по формату `writeln(x:0:3`.

Инициализация данных.

Одна из типичных ошибок при программировании олимпиадных задач — неинициализация глобальных переменных. Нулевые значения всем статическим переменным в программе присвоить достаточно легко. Сделать это можно, например, так: `fillchar(i,ofs(Last)-ofs(i)+sizeof(Last),0)`

Здесь `i` — имя обязательно первой из описанных в программе переменных, Last— последней. Таким образом, данная стандартная процедура заполнит нулями все байты памяти, которые используют статические переменные. После выполнения этой операции все числовые переменные, в том числе и элементы массивов, получат нулевые значения, всем символьным будет присвоен символ с кодом 0, а всем строковым — пустые строки, так как в байт, отвечающий за длину строки, также будет занесен ноль и т.д. Если же количество глобальных переменных в программе невелико и не для всех из них ноль подходит в качестве начального значения, то инициализацию можно проводить для каждой переменной в отдельности. Для простых переменных это можно делать с помощью оператора присваивания или путем описания переменных как типизированных констант (в разделе описаний `const`, но с одновременным указанием и типа переменной и ее значения). Для массивов— с использованием все той же процедуры `fillchar`, но в пределах конкретного массива. Например:

```
var a:array[1..1000]of integer;
    c:array[1..10000]of char;
begin
 fillchar(a,sizeof(a),0);{заполняем массив a нулями}
 fillchar(c,sizeof(c),'+');{заполняем символом плюс массив c}
 end.
```

К сожалению, таким способом ненулевые значения можно присвоить лишь массивам, элементы которых по размеру не превосходят один байт (типы `byte, shortint, char, boolean`). Значения элементов массивов других типов задавать приходится в цикле. Однако, если два массива одного и того же типа требуется проинициализировать одинаково, то заполнить в цикле можно только один из них, а второму массиву просто присвоить первый (присваивание — единственная допустимая операция над составными переменными, такими как массив, как над целыми объектами). Иногда массивы удобно описывать и задавать в разделе констант как типизированную константу путем непосредственного перечисления значений всех элементов массивов.

Как уже говорилось выше, для размещения всех глобальных переменных программе отводится не более 64 килобайт оперативной памяти. Однако при решении задач иногда требуется завести несколько массивов, размер каждого из которых не менее 32 килобайт. Покажем, как достаточно просто решить подобную проблему:

```
const n=150;
type
 aa=array[1..n,1..n] of integer;
var a:aa; {a - массив}
 b:^aa;{b - указатель на массив}
 i,j:integer;
begin
 fillchar(a,sizeof(a),0);
 new(b);{создание динамического массива}
 b^:=a;{копирование массива a в динамический массив}
 for i:=1 to n do
  for j:=1 to n do
   b^[i,j]:=i+j;{обращение к элементам динамического массива}
end.
```

Из примера видно, что работа с динамическими массивами не намного отличается от работы со статическими. Причем использовать данный прием можно “по образцу”, не вдаваясь в механизм работы с указателями. Если же размер двумерного массива превосходит 64 килобайта, то создать его с помощью динамических переменных можно, например, следующим образом:

```
const
 n=500;
 m=100;
type
 aa=array[1..n] of integer;
var b:array[1..m] of ^aa;
 {b - массив указателей на одномерный массив}
 i,j:integer;
begin
 for i:=1 to m do
  new(b[i]);{создание m динамических массивов}
 for i:=1 to m do
  for j:=1 to n do
```

```
   b[i]^[j]:=i+j;{обращение к элементам двумерного массива}
end.
```

Таким образом, использование динамических переменных позволяет практически не изменять алгоритм решения задачи в случае, когда использование статических массивов уже невозможно. Использование же таких переменных требует лишь небольшой аккуратности в их создании и корректного обращения с уже созданными динамическими переменными.

Подсчет времени работы программы

На олимпиадах высокого уровня, к которым можно отнести и региональные соревнования, практически всегда в тексте условия задачи указано максимальное время работы программы на одном тесте. Поэтому писать программу, которая будет работать при каких-либо входных данных дольше этого времени, смысла нет. Если же выбранный участником алгоритм в отведенное для работы программы время не укладывается, то зачастую помогает прием, называемый — *отсечение по времени*. Применяется он в основном в задачах, решение которых производится с помощью перебора вариантов (а большинство олимпиадных задач в силу их дискретности можно решать с помощью полного перебора вариантов, правда лишь при небольших размерностях).

Пусть в условии задачи требуется найти любой допустимый или даже оптимальный вариант, а при отсутствии допустимого варианта выдать сообщение, что решение отсутствует. Тогда работу программы следует организовать так, чтобы по истечении отведенного на ее выполнение времени выполнение программы заканчивалось и печаталось лучшее из рассмотренных к этому моменту решений или сообщение, об отсутствии решения. Если перебор в такой программе организовать “с предпочтением”, то есть сначала рассматривать наиболее вероятные варианты, то такая программа может работать правильно почти на всех входных данных, несмотря на возможную неэффективность заложенного в нее алгоритма. Так как правильно организованный перебор зачастую быстро находит решение задачи, а рассмотрение остальных вариантов необходимо в нем лишь для того, чтобы доказать оптимальность уже найденного. Поэтому, если за отведенное время не найден никакой вариант, то будем считать, что решение в задаче отсутствует (конечно, это не всегда оказывается справедливым), а если какой-то вариант найден, но перебор еще не закончен, то опять же можно надеяться, что этот вариант и есть искомый. В любом случае такая программа всегда результативно заканчивает свою заботу за время ее тестирования и баллы, полученные за нее, могут быть весьма высокими (иногда такая программ получает полный балл за решение задачи).

Программная реализация приема отсечения рассматриваемых вариантов по времени. Опытные участники олимпиад делают это так:

```
const timetest=10;{время тестирования программы, задается в
условии}
var
 timer:longint absolute $40:$6c;
 timeold:longint;
begin
 timeold:=timer;
 while true do
  if timer-timeold>18.2*(timetest-0.5) then
  begin
   {запись текущего результата в файл
     или сообщение об отсутствии решения}
   halt
  end else {собственно работа программы}
end.
```

Данная программа использует тот факт, что к значению четырехбайтовой целой переменной, расположенной по абсолютному адресу `$40:$6C`, раз в 1/18.2 секунды аппаратно прибавляется единица. Поэтому, если мы опишем в нашей программе переменную, привязав ее к этому адресу, то легко сможем определить время работы программы. А именно, запомнив в самом начале программы значение этой переменной (в нашем примере это оператор `timeold:=timer`), в процессе работы определить время выполнения в секундах можно по формуле (timer-timeold)/18.2. Поэтому, если время тестирования известно, то прерывать поиск решения следует за некоторое время до его окончания (в нашем примере это 0,5 секунды), для того, чтобы успеть вывести результат.