С 1 по 30 апреля 2002 года в Московском городском доме учителя состоится

Московский педагогический марафон учебных предметов

• 1 апреля – День учителя русского языка • 2 апреля – День учителя литературы • 3 апреля – День учителей мировой художественной культуры, музыки и ИЗО • 4 апреля – День учителя истории • 5 апреля – День школьного библиотекаря • 8 апреля – День учителя географии • 9 апреля – День учителя биологии • 10 апреля – День учителя химии • 11 апреля – День учителя физики • 12 апреля – День учителя математики • 15 апреля – **ДЕНЬ УЧИТЕЛЯ ИНФОРМАТИКИ** • 16 апреля – День учителя английского языка • 17 апреля – День учителя немецкого языка • 18 апреля – День учителя французского языка • 19 апреля – День учителей технологии, профориентации и ОБЖ • 22 апреля – День учителя физкультуры • 23 апреля – День здоровья детей • 24 апреля – День дошкольного образования • 25 апреля – День учителя начальной школы • 26 апреля – День логопедов и коррекционных педагогов • 29 апреля – День школьного психолога • 30 апреля – День школьной администрации •

№ 12 (349) 23–31 марта 2002

ПОДПИСКА: (095) 249-47-58

Еженедельная газета Издательского дома «ПЕРВОЕ СЕНТЯБРЯ»

ИНФОРМАТИКА

Олимпиады

СОДЕРЖАНИЕ

Олимпиады по информатике. Пути к вершине

Лекции читает Е.В. Андреева

Лекция 10. Конечные автоматы. Разбор выражений

На многих олимпиадах по информатике встречаются так называемые "технические" задачи, решение которых на первый взгляд не требует знания никаких алгоритмов и зависит лишь от владения техникой программирования. К этому классу можно отнести задачи на обработку текстов и выделения с целью последующей обработки некоторых конструкций, формальное описание которых нам задано (например, адресов электронной почты). Однако существуют способы решения подобных задач, существенно облегчающие написание программы. Одним из таких способов является построение математической модели для так называемого *конечного автомата*.

Определение конечного автомата

Конечным автоматом называется реализация алгебраической структуры (S, Σ, m, s_0, F), где

- S — непустое множество *состояний*;
- Σ — конечное множество входных символов (*алфавит*);
- m — отображение $S \times \Sigma \to S$, или функция *переходов*, которая каждой паре (символ, состояние) ставит в соответствие состояние из множества S;
- s_0 — состояние из S, известное как *начальное* (стартовое);
- F — множество заключительных (*допускающих*) состояний, $F \subseteq S$ или просто соответствует окончанию просмотра текста.

Иногда сюда следует добавить для каждой пары (*символ*, *состояние*) процедуру обработки *символа* (например, его печать). Работа автомата заключается в том, что изначально он находится в состоянии s_0 и под действием первого входного символа переходит в следующее состояние, читая следующий символ, и т.д. Автомат заканчивает свою работу, если достигнуто одно из состояний множества F, или прочитан символ, не принадлежащий Σ, или входные данные исчерпаны.

Если отображение m однозначно, то есть каждой паре (символ, состояние) соответствует определенное состояние, то автомат называют *детерминированным*, в противном случае (одной и той же паре в соответствие ставится сразу несколько состояний, чаще всего в зависимости от предыдущих или последующих символов обрабатываемой входной строки) — недетерминированным. Интересно, что разница между двумя типами автоматов несущественна, так как доказано, что для любого недетерминированного конечного автомата можно построить соответствующий ему детерминированный. Последний легче реализовать, а в терминах первого проще записывать условия для большого числа задач.

На примере следующей задачи рассмотрим построение и программную реализацию конечного автомата.

Задача GoTo

Ученики, недавно начавшие программировать, употребляют слишком много операторов **GOTO**, что является почти недопустимым для структурированной программы. Помогите преподавателю информатики написать программу, которая будет оценивать степень структурированности отлаженной программы школьника на языке Паскаль, для начала просто подсчитывая количество операторов **GOTO** в ней.

В синтаксически верной программе ключевое слово оператора перехода **GOTO** может стоять или в начале строки, или после пробела, или после одного из символов — ";", ":", "}", а после него может стоять или пробел, или перевод строки, или символ "{" (табуляцию в качестве разделителя рассматривать не будем).

Напомним, что, кроме обозначения действительно оператора перехода, слово "GOTO" может встречаться в тексте программы в строковых константах, заключенных в апострофы, или в комментариях, причем для простоты будем считать, что комментарий всегда начинается с символа "{", а заканчивается первым встретившимся после этого символом "}", при этом символ "{" должен находиться не внутри строковой константы. В этих случаях слово "GOTO" подсчитывать не нужно. Строчные и прописные буквы в Паскале не различимы.

Во входном файле `goto.in` находится текст программы без синтаксических ошибок на языке Паскаль. Размер программы не превосходит 64 Кб. В выходном файле `goto.out` должно оказаться одно число — количество операторов **GOTO** в этой программе.

План публикаций лекций курса "Олимпиады по информатике. Пути к вершине" на "Страницах повышения квалификации"

Номер лекции	Номер газеты
1	38/2001
2	40/2001
3	42/2001
4	44/2001
5	46/2001
6	48/2001
7	6/2002
8	8/2002
9	10/2002
10	12/2002
11	14/2002
12	16/2002

Пример входного файла

```
label 1,2;
var I:byte;
begin
 1: I := I + 1;
    if I > 10 then goto 2 else
      writeln(' goto '); GoTo 1;
 2: if I < 10 then goto 1 {else { goto 2;}
end.
```

Выходной файл для приведенного примера

```
3
```

Р е ш е н и е. Множество состояний для конечного автомата, с помощью которого легко осуществить решение данной задачи, состоит из трех элементов — соответствующих комментарию, строковой программе и собственно программе (в программной реализации автомата они обозначаются константами перечислимого типа com, str, prog). Алфавит в данном случае совпадает со всеми символами кодовой таблицы. Функция переходов для большинства символов из этого алфавита оставляет состояние прежним. Исключение составляют лишь следующие пары:

(“}”, com) → prog;
(“'”, str) → prog;
(“{”,) → com;
(“'”, prog) → str.

Состояние	Символ	Операция	Новое состояние
com	}	Перейти к следующему символу	prog
	остальные символы	Перейти к следующему символу	com
str	'	Перейти к следующему символу	prog
	остальные символы	Перейти к следующему символу	str
prog	{	Перейти к следующему символу	com
	'	Перейти к следующему символу	str
	G g	Проверить наличие оператора GOTO. Перейти к следующему символу	prog
	остальные символы	Перейти к следующему символу	prog
	конец текста	Вывести результат подсчета и завершить работу	

Начальным является состояние prog. В этом же состоянии для символов “g”, “G” следует применить процедуру обработки, проверяющую наличие в данном месте программы оператора **GOTO**. Данную информацию удобно записать в виде таблицы, представленной ниже.

Теперь написать программу, реализующую работу описанного детерминированного автомата, не составит труда.

```
const sr : set of char=[' ',';',':','}'];
var cgoto : word;{счетчик операторов GOTO}
    s, s1 : string;
    i, j : byte;
    state : (com, str, prog);
begin
 assign(input,'goto.in');
 reset(input);
 cgoto := 0;
 state := prog;
 while not eof do
  begin
   readln(s);
   {s — очередная строка текста программы}
   for i := 1 to length(s) do
   case state of
   com: if s[i] = '}'  then state := prog;
   str: if s[i] = '''' then state := prog;
   prog:
     if s[i] = '''' then state := str else
     if s[i] = '{'  then state := com else
     if (s[i] in ['g','G']) and
        ((i = 1) or (s[i - 1] in sr)) then
      begin
       s1 := copy(s, i, 5);
       for j := 1 to length(s1) do
         s1[j] := upcase(s1[j]);
       if (s1 = 'GOTO'){это конец строки}
        or (s1 = 'GOTO ') or (s1 = 'GOTO{')
       then inc(cgoto)
      end
   end {case}
  end; {while}
 assign(output,'goto.out'); rewrite(output);
 writeln(cgoto)
end.
```

Проверка арифметического выражения на корректность

При рассмотрении различных вопросов, касающихся арифметических выражений, обычно оперируют следующими терминами. *Операнд* — число, переменная или выражение в скобках. *Бинарная операция* — операция, выполняющаяся над двумя операндами, например, сложение. *Унарная операция* — операция, выполняющаяся над одним операндом, знак этой операции обычно стоит непосредственно перед операндом, например, унарный минус или функция одной переменной.

Прежде чем проверять корректность того или иного выражения (не обязательно арифметического), его следует формально описать (по-другому говорят — ввести грамматику). Существует несколько способов для подобных описаний, например, синтаксические диаграммы, регулярные выражения, форма Бэкуса — Наура. Воспользуемся последним способом.

```
<выражение>::=<терм>|<терм>+<выражение>|<терм>-<выражение>
<терм>::=<множитель>|<множитель>*<терм>|<множитель>/<терм>
<множитель>::=(<выражение>)|<имя>|<натуральное число>
<имя>::=<буква>|<имя><буква>|<имя><цифра>
<натуральное число>::=<цифра>|<натуральное число><цифра>
<цифра>::=0|1|2|3|4|5|6|7|8|9
<буква>::=_|A|B|…|Z|a|b|…|z|
```

Здесь слева от метазнака "::=" стоит определяемое понятие, а справа — его определение. Знаки " | " обозначают логическую операцию ИЛИ в определении, остальные символы входят в определение того или иного понятия. В так введенной грамматике для арифметического выражения отсутствуют унарные операции.

Для проверки соответствия некоторого выражения данной грамматике можно построить конечный автомат, имеющий три допускающих состояния (0, 1 и 2 в программной реализации) плюс состояние обнаружения ошибки. В каждом из трех состояний допустимыми являются лишь некоторые символы. Наличие же на входе другого символа говорит об ошибке. Так стартовое нулевое состояние говорит о том, что выражение может начинаться с цифры, буквы или символа "(". В первых двух случаях следует пропустить число или имя целиком и перейти в состояние 1, открывающаяся же скобка оставляет автомат в том же состоянии. В первом состоянии допустимыми являются лишь знаки бинарных арифметических операций и закрывающаяся скобка. Знак операции переводит автомат в состояние 0, а закрывающаяся скобка оставляет его в состоянии 1. Если текущий символ — последний в анализируемой строке, то автомат переводится в состояние 2, допустимым для которого являются буквы, цифры и закрывающаяся скобка. Остается только проверить, что общее число открывающихся скобок равно числу закрывающихся скобок, а при встрече очередной закрывающейся скобки общее их количество к этому моменту не может превышать количество уже встретившихся открывающихся скобок. Для этого вводится целочисленная переменная — счетчик скобок (в программе k), первоначально равная 0. Если в выражении встретилась открывающаяся скобка (состояние 0), то к счетчику прибавляется единица, если закрывающаяся (состояние 1), то единица вычитается. Данная переменная по ходу вычислений не должна принимать отрицательных значений, а при завершении просмотра (состояние 2) должна быть в точности равна нулю.

Выпишем таблицу, задающую описанный автомат.

Приведем программу, реализующую работу этого автомата.

```
const digits: set of char = ['0'..'9'];
  letters: set of char = ['_','A'..'Z','a'..'z'];
       op: set of char = ['+','-','*','/'];
var     s: string;
    i, k: word;
   state: 0..3;
procedure error;
begin
  writeln('Выражение некорректно'); halt
end;
procedure Identifier; {пропустить имя}
begin
  while (i < length(s)) and
        (s[i + 1] in (letters + digits)) do
         i := i + 1
end;
procedure Number; {пропустить число}
begin
   while (i < length(s)) and
         (s[i + 1] in digits) do i := i + 1
end;
begin {Main}
  readln(s);
  i := 0;
  k := 0;
  state := 0;
  while state <> 3 do
  case state of
  0:if i < length(s) then
    begin
      i := i + 1;
      if s[i] = '(' then k := k + 1 else
      if not (s[i] in (letters + digits))
         then error else
      begin
        if s[i] in letters then Identifier
        else if s[i] in digits then Number;
        state := 1
      end
    end else state := 2;
  1:if i < length(s) then
    begin
      i := i + 1;
```

Состояние	Символ	Операция	Новое состояние
0	последний символ		2
	(	k := k + 1, перейти к следующему символу	0
	буква или цифра	пропустить имя или число, перейти к следующему символу	1
	остальные символы	ошибка, конец работы	
1	последний символ		2
	)	k := k − 1, проверить, что k ≥ 0, перейти к следующему символу	1
	+, −, *, /	перейти к следующему символу	0
	остальные символы	ошибка, конец работы	
2	), буква или цифра	если k = 0, то выражение корректно, в противном случае — ошибка, конец	
	остальные символы	ошибка, конец работы	

```
      if s[i] = ')' then
        begin
          k := k — 1;
          if k < 0 then error
        end
      else
      if s[i] in op then state := 0
      else error
    end else state := 2;
  2:if (s[i] in(letters + digits + [')'])) and
       (k = 0) then
     begin
       writeln('Выражение корректно');
       state := 3
     end
    else error
  end {case}
end.
```

Попробуйте изменить грамматику и построенный конечный автомат так, чтобы в арифметических выражениях допускался унарный минус перед числом, именем или выражением в скобках. Придется ли в этом случае увеличивать число состояний?

Стековый конечный автомат

Изменим грамматику арифметического выражения так, чтобы допустимыми в выражении являлись сразу три вида скобок: круглые, квадратные и фигурные. Для этого к определению множителя следует добавить следующее описание: |[выражение]|{выражение}. К сожалению, подсчет в отдельности сбалансированности скобок каждого вида не гарантирует в данном случае корректности выражения. Например, тогда выражение [1+2+{3—4]*5} будет считаться корректным, что неверно. В такой ситуации все встречающиеся открывающиеся скобки следует заносить в стек. Анализ же корректности закрывающейся скобки будет состоять в ее сравнении с верхним элементом стека (т.е. последней занесенной в него открывающейся скобкой). При соответствии скобок друг другу открывающаяся скобка из стека извлекается, в противном случае — выражение некорректно. При достижении конца входных данных следует проверить, что стек пуст.

Перейдем теперь к рассмотрению различных способов подсчета значений арифметических выражений.

"Палочный" способ разбора арифметических выражений

Данный способ подсчета значения арифметического выражения фактически моделирует действия человека, выполняемые при вычислении арифметических выражений. То есть сначала ищется операция, которую можно выполнить первой, она выполняется и в измененном выражении вновь ищется первая выполнимая операция. Причем в большинстве случаев человек при этом понятие "первой выполнимой операции" не формализует строго, а опирается на многолетний практический опыт работы с арифметическими выражениями, хотя в школьных учебниках по математике для начальной школы эти правила в том или ином виде сформулированы. Для компьютерной же реализации этого метода нужна четкая система правил. Прежде чем ее выписать, поставим в соответствие исходному арифметическому выражению строку, в которой каждый элементарный операнд из выражения (в случае подсчета такими операндами являются только числа) заменим на символ "|" ("палочку"), а знаки операций и скобки оставим неизменными. Например, выражению

32/(2*4)+10+(5—3—1)

соответствует строка |/(|*|)+|+(|—|—|). Теперь выпишем в терминах "палочек" действия, которые следует выполнять при подсчете значения арифметического выражения.

1	(|) → |	Скобки можно снять
2	|*| или |/| → |	Умножение или деление, встретившееся первым, можно выполнять
3	(|±|) → (|) или (|±|± → (|±	В данном контексте сумму или разность можно вычислять
4	|±| → |	Если 1—3 применить нельзя, то эти операции выполнить можно

Применять указанные правила следует так. В "палочной" строке ищется первое вхождение подстроки сначала из левой части правила 1, если такая не найдется вообще — то из правила 2 и т.д. Для найденной подстроки осуществляется замена согласно правилу, а над соответствующими этой подстроке элементами исходного выражения производятся те же арифметические действия, что и в найденной подстроке (для правила 1 — просто снимаются скобки). После этого то же самое правило пытаются применить еще раз, а если это невозможно, то снова переходят к поиску подстроки из правила 1 (а не из следующего правила, что весьма существенно). Можно доказать, что таким образом к поиску подстроки из левой части правила 4 мы приступим лишь тогда, когда в выражении уже не останется ни скобок, ни операций умножения или деления. Действия заканчиваются, когда у нас останется одна палочка, т.е. исходное выражение будет сведено к одному числу, являющемуся его значением. Покажем, как по этим правилам будет вычисляться значение выражения из примера.

По правилу 2	|/(|)+|+(|—|—|)	32/(8)+10+(5—3—1)
По правилу 1	|/|+|+(|—|—|)	32/8+10+(5—3—1)
По правилу 2	|+|+(|—|—|)	4+10+(5—3—1)
По правилу 3	|+|+(|—|)	4+10+(2—1)
По правилу 3	|+|+(|)	4+10+(1)
По правилу 1	|+|+|	4+10+1
По правилу 4	|+|	14+1
По правилу 4	|	15

На первый взгляд может показаться, что правила 3 являются лишними. Однако это не так. Например, выражение `2+3*(4–5+6)` без любого из правил 3 будет вычислено неверно.

Программу, реализующую описанный алгоритм подсчета, написать нетрудно. "Палочное" выражение легко представить с помощью переменной типа **`string`** (тогда поиск любой подстроки можно выполнять с помощью стандартной функции `Pos`), а исходное выражение — с помощью массива записей, одно поле которых символьное, а второе — числовое: если *i*-й элемент выражения символ скобки или арифметической операции, то у *i*-го элемента массива соответствующим символом будет заполнено только символьное поле, если же элемент — число, то в символьном поле можно поставить все ту же палочку, а само число поместить в числовое поле. При таком способе представления исходного арифметического выражения *i*-й символ строки будет соответствовать *i*-му элементу массива. При дальнейших преобразованиях это соответствие нужно сохранять (из строки удалять лишние символы и сдвигать массив влево на место удаляемых элементов).

Однако такой способ подсчета арифметических выражений весьма неэффективен и, например, в компиляторах не применяется. Ведь для выполнения очередной арифметической операции нам придется просмотреть в худшем случае всю строку из палочек, причем, возможно, не один раз (ведь даже нахождение операции умножения из правила 2 еще не означает, что ее можно выполнять, так как операция деления может встретиться раньше). Поэтому перейдем к рассмотрению более эффективных способов разбора, правда, не столь очевидных.

Подсчет арифметических выражений с помощью постфиксной нотации

Любое арифметическое выражение можно переписать в виде бесскобочного постфиксного выражения, в котором знак операции стоит после своих операндов, а обозначение функции — после всех ее аргументов. Такую запись еще называют *обратной польской нотацией* (*записью*). Преимущество этой записи заключается в том, что все встречающиеся в ней операции выполняются последовательно слева направо. Разделим алгоритм подсчета арифметического выражения на две части — перевод в "обратную польскую запись" и подсчет по ней значения выражения.

Перевод в "обратную польскую запись"

При выполнении следующих действий над исходной строкой с арифметическим выражением мы будем использовать вспомогательную структуру данных — стек и строку для записи результата. Просматриваем исходную строку с выражением один раз слева направо.

1. Если очередной элемент — число или имя переменной, то он сразу переносится в результирующую строку.
2. Открывающаяся скобка всегда помещается в стек.
3. Если встретилась закрывающаяся скобка, то из стека извлекаются все знаки операций до первой открывающейся скобки и в порядке извлечения переносятся в результирующую строку, а открывающаяся скобка в вершине стека уничтожается.
4. Если встречается знак операции, то
   a) если в вершине стека знак операции с более низким приоритетом или открывающаяся скобка или ничего нет, то встретившийся знак заносится в стек;
   b) в противном случае знаки операций из стека, пока приоритет их больше или равен приоритету данного знака, извлекаются из стека и заносятся в результирующую строку, после чего рассматриваемый знак записывается в стек.
5. Если исходная строка исчерпана, то оставшиеся в стеке знаки операций по порядку переносятся в результирующую строку.

Например, постфиксной записью для `9/(5+2*3-8)` будет `9523*+8-/`.

Несложно заметить, что эти действия легко описать и реализовать с помощью стекового конечного автомата. Интересно, что для данного алгоритма преобразования выражения неважно, какие именно операции в нем могут встречаться. Нужно только знать приоритеты всех допустимых операций. Укажем приоритеты наиболее типичных операций в арифметических выражениях (от высшего к низшему):

1) унарный минус, функция;
2) умножение, деление, в том числе целочисленные;
3) сложение, вычитание.

При преобразовании выражения в постфиксную нотацию унарный минус следует обозначать специальным знаком, например, символом подчеркивания, в противном случае вычисления по такой записи могут быть произведены неверно.

Рассмотрим теперь, как по построенной постфиксной записи подсчитать значение выражения.

Подсчет значения выражения, записанного в "польской нотации"

Строка с выражением в "обратной польской записи" просматривается один раз слева направо. В качестве вспомогательной структуры данных опять используется стек.

1. Если встретившийся элемент — операнд, то он помещается в стек.

2. Если очередной элемент — знак бинарной операции, то из стека извлекаются два верхних элемента и над ними выполняется операция, результат которой заносится в стек, унарная операция выполняется над верхним элементом.

В конце концов в стеке остается одно число — результат.

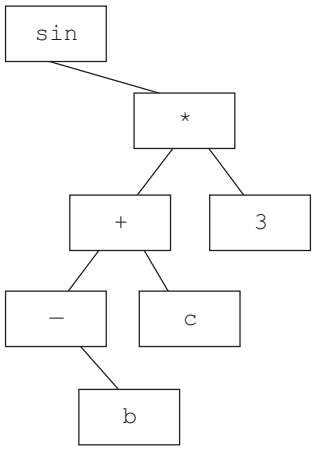
В нашем примере сначала в стек попадают операнды 9, 5, 2 и 3 (верхний элемент — 3). Покажем содержимое стека после выполнения каждого действия алгоритма подсчета.

Операция	Стек
*	9 5 6
+	9 11
8	9 11 8
—	9 3
/	3

Именно таким образом в большинстве компиляторов осуществляется разбор арифметических и логических выражений.

Метод рекурсивного спуска

Альтернативой стековому представлению выражения является так называемое *дерево разбора*. Для арифметического выражения его можно описать так. В корне бинарного дерева находится знак операции, которая должна быть выполнена последней. Левым поддеревом (сыном) является дерево, представляющее первый из операндов последней операции, правым — второй из операндов. Если эта операция унарная, то левое поддерево отсутствует. Для функции одной переменной правое поддерево представляет ее аргумент. Концевые узлы дерева (листья) представляют числа или имена переменных. Например, для выражения $\sin((-b + c) \cdot 3)$ дерево разбора выглядит так:



Подсчет значения арифметического выражения, основанный на таком способе представления, рекурсивен и фактически совпадает с рекурсивным определением самого выражения (см. с. 4). Так, выражение представляет собой сумму слагаемых (здесь под суммой подразумевается выполнение операций как сложения, так и вычитания). Каждое слагаемое представляет собой произведение множителей (сюда же отнесена операция деления). А множитель представляет собой либо число, либо переменную, либо выражение в скобках (при наличии унарного минуса множителем является также выражение со стоящим перед ним знаком "минус").

Далее приведены две реализации рекурсивного спуска. В первой из них производится только подсчет значения арифметического выражения. В качестве элементарных операндов используются только числа. Во второй — сначала строится дерево разбора, элементарными операндами в котором являются однозначные цифры или однобуквенные имена переменных. Унарный минус в обоих случаях допускается. Если в построенном с помощью правой программы дереве все листья — однозначные числа, то значение выражения также будет верно подсчитано. Бинарное дерево строится с использованием динамических переменных. Преимущество второй программы заключается в том, что с помощью построенного дерева можно решать и другие задачи, например, перебирать все возможные способы расстановки скобок или распараллеливать на насколько процессоров вычисление значения выражения.

```
var s : string;{исходное выражение}
    i : integer;{номер текущего символа}
function Mul:longint; forward;
function Factor:longint; forward;
function Add:longint;
{суммирует слагаемые}
var q, res : longint;
    c : char;
begin
 res := Mul;{первое слагаемое}
 while s[i] in ['+','—'] do
 begin
  c := s[i];
  i := i + 1;
  q := Mul;{очередное слагаемое}
  case c of
  '+' : res := res + q;
  '-' : res := res — q;
 end
 end;{while}
 Add := res
end;
function Mul:longint;
{перемножает множители}
var q, res : longint;
    c : char;
begin
 res := Factor;{первый множитель}
 while s[i] in ['*', '/'] do
 begin
  c := s[i];
  i := i + 1;
  q := Factor;{очередной множитель}
  case c of
  '*' : res := res * q;
  '/' : if q = 0 then
        begin
         writeln('деление на 0');
         halt
        end
        else res := res div q
 end {case}
 end; {while}
 Mul := res
end;
function Number : longint;
{выделяет число}
var res : longint;
begin
 res := 0;
 while (i <= length(s)) and
       (s[i] in ['0'..'9']) do
```

```
 begin
 res := res * 10 + (ord(s[i]) – ord('0'));
 i := i + 1
 end;
 Number := res
end;
function Factor : longint;
{выделяет множитель}
var q : longint;
    c : char;
begin
 case s[i] of
 '0'..'9' : Factor := Number;
 '(' : begin i := i + 1; Factor := Add;
     i := i + 1;{пропустили ')'}end;
 '-':begin i := i + 1;
       Factor := –Factor;
     end
  else begin writeln('ошибка');
        halt
       end
  end {case}
end;
begin {основная программа}
 readln(s); i := 1;
 writeln(Add)
end.
```

А теперь программа, строящая дерево разбора.

```
type ptr = ^element;
element = record c : char; left,right : ptr
end;
var s : string; i : integer; res : ptr;
Function Mul : ptr; forward;
Function Factor : ptr; forward;
Function Add : ptr;
var p, q : ptr;
begin
 p := Mul;{левый операнд}
 while s[i] in ['+', '–'] do
  begin new(q);
   q^.c := s[i]; i := i + 1;
   q^.left := p;{левый операнд}
   q^.right := Mul;{правый операнд}
   p := q
  end;{while}
 Add := p{в корне последняя операция}
end;
function Mul : ptr;
var p, q : ptr;
begin
 p := Factor;{левый операнд}
 while s[i] in ['*', '/'] do
  begin new(q);
   q^.c := s[i]; i := i + 1;
   q^.left := p;{левый операнд}
   q^.right := Factor;{правый операнд}
   p := q;
  end; {while}
 Mul := p{в корне последняя операция}
end;
function Factor:ptr;
var q : ptr;
begin
 case s[i] of
 '0'..'9','A'..'Z':{это лист}
   begin new(q);
    q^.c := s[i]; i := i + 1;
    q^.left := nil; q^.right := nil;
    Factor := q
   end;
 '(':begin i := i + 1;
      Factor := Add;
      i := i + 1{пропускаем ')'}
     end;
 '-':begin {унарный минус} new(q);
      q^.c := s[i]; i := i + 1;
      q^.right := Factor;
      q^.left := nil; {левого сына нет}
      Factor := q
     end
 else begin writeln('ошибка'); halt end
 end {case}
end;
function calc(p : ptr) : real;
{подсчитывает по дереву значение выражения}
var r : real;
begin
 if p <> nil then
  case p^.c of
  '+' : calc := calc(p^.left) + calc(p^.right);
  '-' : calc := calc(p^.left) – calc(p^.right);
  '*' : calc := calc(p^.left) * calc(p^.right);
   '/' : begin r := calc(p^.right);
         if r = 0 then
          begin writeln('деление на 0');
                halt
          end
         else calc := calc(p^.left)/r
        end;
  '0'..'9' : calc := ord(p^.c) – ord('0');
  end {case}
 else {p = nil} calc := 0
end;
begin {основная программа}
 readln(s); i := 1;
 res := Add; {построили дерево}
 writeln(calc(res):0:3)
end.
```

В данной лекции мы рассмотрели лишь применение конечных автоматов и некоторые способы разбора выражений, не доказывая строго возможность их использования в тех или иных задачах. О теории конечных автоматов, формальных способах описания выражений и методах их разбора более подробно можно прочитать в [1—4].

Литература

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2001.

2. *Бауэр Ф.Л., Гоос Г.* Информатика. Вводный курс. Часть 2. М.: Мир, 1990.

3. *Кук Д., Бейз Г.* Компьютерная математика. М.: Наука, 1990.

4. *Шень А.* Программирование: теоремы и задачи. М.: МЦНМО, 1995.

Введение в профессию "учитель информатики"

Лекции читает А.Г. Гейн

Лекция 4. В каком порядке излагать материал

Где начало того конца, которым оканчивается начало?
Козьма Прутков. "Мысли и афоризмы"

Практически любая научная дисциплина устроена так, что, казалось бы, самые далекие друг от друга ее разделы оказываются тем не менее связанными. И полноценное изучение одного из разделов невозможно без основательной предварительной проработки другого. Переплетение различных "ниточек" курса иногда бывает настолько тесным, что их изучение вообще возможно только в едином комплексе.

Но человек не способен воспринимать эту "многомерную ткань", поэтому обучение выстраивается линейно — тема за темой. Над тем, как это сделать, ломают голову дидактики и методисты, авторы учебных программ и учебников. На помощь призвана идея концентризма — ранее изученная тема некоторое время спустя вновь выходит на сцену и рассматривается с позиций уже большего знания, которое к этому времени получено из других тем[1].

В каком-то смысле противоположной идеей является идея модульного построения курса. Модули предполагаются самодостаточными и независимыми друг от друга. Поэтому их можно комбинировать если не вообще в произвольном порядке, то по крайней мере достаточно свободно. Эта свобода выглядит весьма привлекательно: учитель получает возможность создавать из модулей, как из кубиков, курс произвольной конфигурации, и, разумеется, прежде всего той, которая ему лично по вкусу. Авторы же подобных курсов тоже ощущают облегчение: объявив курс модульным, они как бы снимают с себя обязанность продумывать связи между модулями — теперь это забота учителя, взявшегося за реализацию такого курса.

Все сказанное выше отнюдь не чуждо информатике. Более того, складывается впечатление, что именно на курсе информатики апробируются различные идеи модульности. Оно и понятно: в изложении других дисциплин существуют вековые традиции, дидактические и методические связи уже давно выстроены и устоялись, поэтому они кажутся естественными и незыблемыми. В информатике все ново, все зыбко.

В том, что это за связи, мы и попытаемся разобраться в данной лекции.

План публикации лекций курса "Введение в специальность "учитель информатики" на "Страницах повышения квалификации".

Номер лекции	Номер газеты
1	6/2002
2	8/2002
3	10/2002
4	12/2002
5	14/2002
6	16/2002

[1] Конечно, это не единственная и, быть может, не самая главная причина концентрического построения процесса изучения того или иного предмета. Важную роль играют возрастные особенности учащегося, взаимодействие с другими дисциплинами и т.д. Но и той причины, которая указана нами, более чем достаточно для возведения концентрического построения материала в некий методический принцип.

Если задать вопрос, какие связи должны неукоснительно соблюдаться, то, как показывает практика, на первое место, как правило, выходит следующий ответ: связи логического следования. И этот ответ кажется вполне очевидным. Кому придет в голову рассматривать свойство адекватности модели до того, как введено само понятие модели, или изучать понятие циклического алгоритма до того, как введено само понятие алгоритма? Ведь циклический алгоритм — это частный вид алгоритма вообще.

Но если мы посмотрим на математику — эту, без сомнения, самую логичную дисциплину среди всех школьных дисциплин, — то обнаружим нечто странное. Скажем, понятие прямоугольника вводится на несколько лет раньше, чем понятие многоугольника и даже параллелограмма. Введение понятия "отрезок" предшествует понятию "прямая", хотя по определению отрезок — это часть прямой, заключенная между двумя различными точками. Едва ли надо говорить, что понятие производной, составляющее основу курса алгебры и анализа в 10—11-х классах, полностью опирается на понятие предела функции, о котором услышат только те, кто поступит в технические вузы. Не так все просто, как кажется на первый взгляд. Мы здесь снова сталкиваемся с проявлением принципа концентризма — сначала осваиваются более простые (наглядные, легче объяснимые, чаще встречающиеся на практике и т.п.) понятия, а потом, по мере накопления знаний, рассматриваются более общие понятия.

Вот достаточно распространенный пример подобной ситуации в информатике. Изучение алгоритмической конструкции "цикл **пока**" или "цикл **для**" предваряется рассмотрением цикла "*n* **раз**", хотя эта разновидность цикла с очевидностью является частным случаем любого из двух других. Именно так обстоит дело в учебнике [6]. И это методически оправданно, поскольку простое повторение заданное количество раз воспринимается учащимися намного легче (хотя бы за счет того, что совершенно прозрачным является условие окончания цикла), нежели более сложная для понимания формулировка "пока"[2].

[2] Сложность восприятия конструкции "пока" вызвана, по-видимому, еще и неоднозначной семантикой этого слова в русском языке. Нередко оно означает какой-то интервал времени, в течение которого что-то происходило. Вот фразы, в которых слово "пока" употребляется именно в таком смысле: "Пока мы тут разговаривали, поезд ушел"; "Пока жена его не бросила, он был вполне порядочным человеком" — и т.п. Вообще выбор на роль термина слова, имеющего неоднозначную семантику, всегда создает трудности в освоении того понятия, для обозначения которого данный термин выбирается. Нередко, однако, подобные методические ошибки совершают и авторы учебников. А расплачиваться за них приходится ученикам и студентам. В качестве вопиющего примера приведем всем хорошо известный математический термин "неправильная дробь". Что должен думать ученик, услышав от учителя фразу: "У тебя получилась неправильная дробь"?

Можно сделать такой вывод: предварять изучение общего понятия каким-либо частным целесообразно при выполнении двух условий:

— это частное понятие должно допускать определение, независимое от общего понятия;

— данное частное понятие должно играть значительную роль в практике использования общего понятия (например, более часто применяться).

Легко убедиться, что, скажем, и понятие прямоугольника[3], и понятие цикла "*n* **раз**" удовлетворяют обоим указанным условиям. Отметим, что речь идет именно о частном случае понятия; то, что введение *любого* понятия весьма полезно сначала проиллюстрировать подходящим частным примером, дискуссии не вызывает.

Вернемся теперь к рассмотрению вопроса, что определяет порядок изучения понятий (и, более общо, дидактических блоков и модулей), когда они логически относительно независимы. Данное обсуждение начнем с примера.

При изучении алгоритмизации, после того как введены понятия алгоритма и его построения в виде линейной последовательности действий, рассматриваются основные алгоритмические конструкции: ветвление, цикл и вспомогательный алгоритм. Возникает естественный вопрос: какую из указанных конструкций изучать первой, какую второй, а какую в последнюю очередь?

Обратимся к существующим учебникам.

В [6] порядок следующий: вспомогательный алгоритм, циклы *n* **раз** и **пока**, ветвление, цикл **для**.

В [1] и [2] сначала рассказывается о ветвлении, затем о цикле **пока**, затем вводится понятие вспомогательного алгоритма и, наконец, цикл **для**.

В [3] порядок таков: цикл **пока**, ветвление, вспомогательный алгоритм, цикл **для**.

В [5]: ветвление, цикл **пока**, команда безусловного перехода, цикл **для**, подпрограмма.

В [4]: цикл **повтори** *n* и процедура.

Налицо весьма значительное разнообразие применяемых вариантов. Проанализируем 3 из них и попытаемся ответить на вопрос, какое методическое обоснование можно дать каждому из рассматриваемых вариантов.

1-й вариант: *вспомогательный алгоритм, цикл* ***пока****, ветвление.*

Введение вспомогательного алгоритма позволяет использовать для решения задачи механизм нисходящего проектирования, согласно которому исходная задача разбивается на ряд более простых задач, в свою очередь, эти задачи могут быть разбиты на еще более простые и т.д. Более того, на этом пути некоторые из подзадач могут оказаться уже ранее решенными и потому имеющими уже готовый алгоритм. Решение каждой подзадачи оформляется отдельным вспомогательным алгоритмом, а главный алгоритм просто осуществляет их сборку в единое целое.

Таким образом, ознакомление школьников со вспомогательными алгоритмами позволяет сразу расширить круг обсуждаемых задач, привлекая интересные задачи. А это, в свою очередь, повышает мотивацию к изучению предмета.

Нередко оказывается, что в основном алгоритме требуется несколько раз применить один и тот же вспомогательный алгоритм для разных значений аргумента вспомогательного алгоритма. Если множество этих значений можно как-либо "регуляризировать", например, представить как множество значений некоторого вычисляемого выражения, то вспомогательный алгоритм можно заменить конструкцией цикла: телом цикла фактически будет тело вспомогательного алгоритма, а вместо неоднократных вызовов вспомогательного алгоритма при разных значениях аргумента ставится проверяемое условие продолжения (или окончания) цикла. Особенно это важно, когда заранее неизвестно, сколько раз пришлось бы вызывать вспомогательный алгоритм. В этом случае как раз и возникает цикл в форме **пока**.

Итак, как мы видим, переход от вспомогательных алгоритмов к циклам довольно прост: понятие тела цикла фактически сформировано при рассмотрении вспомогательных алгоритмов, и остается только дополнительно ввести понятие условия продолжения или окончания цикла.

Условие продолжения или окончания цикла сигнализирует о том, следует ли выполнять следующую за ним серию действий. Но такую же роль играет условие и в конструкции "ветвление". Поэтому после изучения цикла в форме **пока** учащиеся легко переходят к освоению ветвлений в алгоритмах.

Вот как, по нашему представлению, могут быть высказаны те методические аргументы в пользу данного варианта последовательности изучения.

2-й вариант: *ветвление, цикл* ***пока****, вспомогательный алгоритм.*

После того как введено понятие алгоритма и рассмотрена линейная форма организации действий, перед учащимися естественно поставить проблему, что в реальной жизни линейные алгоритмы имеют весьма ограниченную сферу применения — ведь почти всегда до выполнения действия требуется принять решение, надо ли данное действие выполнять. Вот как такая постановка делается в [1].

"Легко и просто было бы жить (и даже неинтересно), если бы удалось раз и навсегда расписать, какие поступки и в какой последовательности совершать. На самом деле нам постоянно приходится принимать решения в зависимости от создавшейся ситуации. Если идет дождь, то мы надеваем плащ. Если жарко, то идем купаться. Разумеется, встречаются и более сложные положения, когда надо сделать выбор. Рассмотрим два примера.

Пример 1. Допустим, вы собрались пойти в кинотеатр на сеанс 12.00. Алгоритм покупки билетов может выглядеть так:

```
Подойти к кассе.
Если билеты на сеанс 12.00 есть, то
   купить билеты.
Отойти от кассы.
```

Пример 2. Рассматривается жизненная ситуация, когда используется ветвление в полной форме: "если ... то ... иначе ...".

[3] Прямоугольник вводится еще в начальной школе как четырехугольник, у которого все углы прямые. Дети еще не знакомы ни с понятием параллельности (и, значит, не знают понятие параллелограмма), ни с понятием многоугольника вообще.

Необходимость изучения конструкции ветвления теперь у школьников сомнений не вызывает; конструкция эта весьма естественна и знакома им как из повседневной жизни, так и из их занятий математикой. Так что в этом контексте вполне естественно начинать именно с этой конструкции. Кроме того, специальной отработки здесь требует только умение выделять условие, в соответствии с которым исполнитель будет принимать решение, выполнять ему данную последовательность действий или нет. Формирование самой этой последовательности редко вызывает у учащихся какие-либо затруднения.

После освоения ветвления совершенно естественно перейти к понятию цикла **пока**. Действительно, основное отличие состоит в том, что выделяемая последовательность действий должна быть исполнена не один раз, а несколько, пока остается выполненным условие продолжения цикла. Конечно, здесь потребуется более тщательная отработка умения выделять тело цикла, но она разделена с отработкой выделения условия, которая проведена раньше.

Наконец, введение вспомогательных алгоритмов полностью подготовлено рассмотрением циклов. Ведь прибегать к вспомогательному алгоритму приходится тогда, когда невозможно единообразно сформулировать условие цикла, а каждый раз принимать решение об исполнении нужной последовательности действий, т.е. об обращении к вспомогательному алгоритму, приходится в контексте той обстановки, которая к этому моменту сформирована исполнителем. Вспомогательный алгоритм — конструкция более сложная, чем предыдущие, поскольку требует введения понятий формальные и фактические параметры для своих аргументов, а также понятий локальной и глобальной переменных. Кроме того, вспомогательный алгоритм — это средство для мощного, но весьма тонкого инструмента алгоритмизации, каким является рекурсия. Фактически если не пользоваться инструментом рекурсии, то можно вообще обойтись без вспомогательных алгоритмов, встраивая всякий раз в основной алгоритм вместо вызова тело вспомогательного алгоритма с соответствующими значениями входных параметров. Все это показывает обоснованность последнего места понятия "вспомогательный алгоритм" в ряду указанных алгоритмических конструкций.

3-й вариант: *цикл* ***пока****, ветвление, вспомогательный алгоритм.*

Спросите любого учителя информатики, и почти каждый из них скажет, что начинать надо с циклов — ведь именно конструкция цикла позволяет продемонстрировать высокую эффективность компьютера: в весьма коротком вычислительном алгоритме могут спрятаться тысячи и даже миллионы действий, на исполнение которых у человека ушло бы не только несколько месяцев, но, возможно, и несколько лет. В этом смысле ветвления всегда вызывают у школьников некоторое неудовольствие. Ведь реально будет исполняться только одна последовательность действий из тех двух, которые приходится прописывать "на всякий случай" в ветвлении. Получается, что при составлении ветвящегося алгоритма работать приходится больше, чем компьютеру при исполнении данного алгоритма. А кому же охота делать лишнюю работу? Так что мотивационно циклы ветвлениям дадут 100 очков вперед.

Перейти от циклов к ветвлениям можно уже так, как это сделано в варианте 1, и мы на этом останавливаться не будем. Можно, конечно, от изучения циклов переходить и ко вспомогательным алгоритмам, но напомним, что введение вспомогательных алгоритмов требует изучения еще целого ряда понятий, весьма непростых для школьников. Так что вспомогательные алгоритмы и здесь лучше отправить на третье место.

Остановимся на этом. Нам представляется, что приведенный разбор методических мотивов, согласно которым может выстраиваться та или иная последовательность изучения материала, убедительно показал отсутствие однозначности в решении данного вопроса. У каждого учителя могут оказаться свои предпочтения, но мы хотим призвать к тому, чтобы выбор был обдуманным, а не случайным.

Вопросы и задания

1. Изучение конструкций "ветвление", "цикл **пока**" и "вспомогательный алгоритм" можно упорядочить шестью вариантами. Три из них мы рассмотрели в тексте лекции и показали, что приемлемость каждого из них имеет определенное методическое обоснование. Проанализируйте с этой точки зрения три оставшихся варианта.
2. В курсе школьной информатики обычно изучаются 4 информационных технологии: текстовый редактор, графический редактор, электронная таблица, база данных (или информационно-поисковая система). Теоретически можно составить 24 варианта последовательности, в которой будут изучаться эти технологии. Для каждого из вариантов попытайтесь найти методическое обоснование для именно такой последовательности изучения технологий. Укажите наиболее методически оправданную, на ваш взгляд, последовательность изучения указанных технологий.

Литература

1. *Гейн А.Г., Линецкий Е.В., Сапир М.В., Шолохович В.Ф.* Информатика: Учебник для 8—9-х классов общеобразовательных учреждений. М.: Просвещение, 1994, 256 с.

2. *Гейн А.Г., Сенокосов А.И., Шолохович В.Ф.* Информатика: Учебник для 7—9-х классов общеобразовательных учебных заведений. М.: Дрофа, 1997, 224 с.

3. *Гейн А.Г., Сенокосов А.И., Юнерман Н.А.* Информатика: Учебное пособие для 10—11-х классов общеобразовательных учреждений. М.: Просвещение, 2000, 255 с.

4. Информатика: Учебник для 6—7-х классов средней школы / Под ред. Н.В. Макаровой. СПб.: ПитерКом, 1998, 256 с.

5. *Кузнецов А.А., Апатова Н.В.* Основы информатики. 8—9-е классы: Учебник для общеобразовательных учебных заведений. М.: Дрофа, 1999, 176 с.

6. *Кушниренко А.Г., Лебедев Г.В., Сворень Р.А.* Основы информатики и вычислительной техники: Пробный учебник для средних учебных заведений. М.: Просвещение, 1990, 224 с.

II Всероссийская командная олимпиада школьников по программированию

А.С. Станкевич,
Санкт-Петербург

Задача A

Две стены

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Максимальное время работы на одном тесте:	10 секунд

У Пети есть набор из N кирпичиков. Каждый кирпичик полностью окрашен в один из K цветов, i-й кирпичик имеет размер $1 \times 1 \times L_i$.

Петя знает, что он может построить из кирпичиков прямоугольную стену толщиной 1 и высотой K, причем первый горизонтальный слой кирпичиков в стене будет первого цвета, второй — второго и т.д. Теперь Петя хочет узнать, может ли он из своего набора построить две прямоугольные стены, обладающие тем же свойством. Помогите ему выяснить это.

Формат входных данных

На первой строке входного файла находятся числа N и K ($1 \le N \le 5000$, $1 \le K \le 100$). Следующие N строк содержат описание Петиных кирпичиков: сначала длина L_i, затем номер цвета C_i ($1 \le L_i \le 100$, $1 \le C_i \le K$). Известно, что у Пети не более 50 кирпичиков каждого цвета.

Формат выходных данных

Выведите на первой строке выходного файла YES, если Петя сможет построить из своих кирпичиков две прямоугольные стены высотой K, j-й слой кирпичиков в каждой из которых будет j-го цвета, и NO в противном случае. В случае положительного ответа выведите на второй строке в произвольном порядке номера кирпичиков, из которых следует построить первую стену (кирпичики нумеруются в том порядке, в котором они заданы во входном файле, начиная с 1). Если решений несколько, можно выдать любое из них.

Примеры

input.txt	output.txt
5 2 2 1 2 1 2 1 3 2 3 2	NO
2 1 1 1 3 1	YES 1
9 3 7 3 1 1 3 2 2 1 3 2 3 1 4 2 4 1 3 3	YES 2 3 4 9

Решение

Для решения этой задачи, оказавшейся одной из самых сложных (а самые простые задачи — C, F, G, H), применим динамическое программирование. Пусть сумма длин всех кирпичиков каждого из K цветов равна W (для всех $j = 1, 2, \ldots, K$ выполняется $\sum_{i:C_i=j} L_i = W$). Поскольку одну прямоугольную стену Петя построить может, эта сумма для всех цветов одинакова и наше определение имеет смысл. Обозначим для удобства множество номеров кирпичиков цвета j за Z_j.

Заметим, что, для того чтобы Петя мог построить две стены, необходимо и достаточно, чтобы существовало $1 \le V < W$, такое, что для каждого цвета j существовал набор кирпичиков j-го цвета, таких, что сумма их длин была равна V, т.е. для всех $j = 1, 2, \ldots, K$ существовало множество $U_j \subset Z_j$, такое, что $\sum_{i \in U_j} L_i = V$.

Если зафиксировать какое-либо V, то задача о существовании такого множества для конкретного множества Z_j эквивалентна одной из формулировок известной задачи о рюкзаке (можно ли набрать вес V с помощью камней с весами w_i, можно использовать не все камни). Это одна из известных задач, для которой в случае произвольных (неограниченных сверху) весов w_i эффективное решение неизвестно. Однако, когда веса ограничены (как в нашем случае, когда $L_i \le 100$), эта задача решается с помощью динамического программирования.

Рассмотрим сначала кирпичики только 1-го цвета. Условно перенумеруем их заново от 1 до m_1, где m_1 — количество кирпичиков 1-го цвета. Заполним вспомогательный двухмерный массив B так. $B[i]\ [j]$ содержит 1, если можно, используя некоторое подмножество первых i кирпичиков, построить стену длиной j, и 0, если этого сделать нельзя. Тогда рекуррентная формула для $B[i]\ [j]$ имеет вид:

$$B[i]\ [j] = \begin{cases} 1, & \text{если } i = 0,\ j = 0 \\ 0, & \text{если } i = 0,\ j > 0 \\ B[i-1]\ [j], & \text{если } i > 0,\ j < L_i \\ \max(B[i-1]\ [j], B[i-1]\ [j-L_i]), & \text{если } i > 0,\ j \ge L_i. \end{cases}$$

Если заполнять массив B в порядке возрастания первого индекса (по строкам), то для определения значения очередного элемента используется только элемент предыдущей строки, причем стоящий в ряду не больше текущего, поэтому можно сэкономить память, храня только одну текущую строку и перезаполняя ее с конца. Предполагая, что данные прочитаны в массив L и $L\,[i]\ [j]$ — длина j-го кирпичика цвета i, получаем процедуру заполнения массива B (сделаем его булевским) для i-го цвета:

```
Procedure FillB(i: integer);
var j, t: integer;
begin
  fillchar(b, sizeof(b), false);
  b[0] := true;
  for j := 1 to m[i] do
    for t := w downto l[i][j] do
      if b[t - l[i][j]] then b[t] := true
end;
```

Отметим, что если бы мы решали задачу о рюкзаке, т.е. для конкретного V хотели бы проверить, что ряд такой длины можно получить из кирпичиков, то ответом было бы $B[m_i]$ [V] или, поскольку в результате выполнения нашей процедуры в b хранится именно последняя строка, b[v]. Но так как значение V мы не знаем и оно должно быть одним для всех K цветов, то логично сохранить все возможные значения длины ряда каждого цвета для последующего сравнения с результатами расчетов для других рядов. Более того, поскольку нам надо найти длину ряда, которая бы подошла сразу для всех цветов, то достаточно иметь всего один одномерный массив *Can*, который будет содержать значение true для тех V, для которых можно построить ряд кирпичиков длиной V для всех обработанных цветов. Тогда если после окончания обработки всех цветов найдется $1 \le V < W$, такое, что $Can[V] = true$, то задача имеет решение, то есть Петя сможет построить две стены: одну размером $K \times V$ и другую — $K \cdot (W - V)$.

Приведем фрагмент программы, который реализует заполнение массива can:

```
fillchar(can, sizeof(can), true);
for i := 1 to k do
  begin
    FillB(i);
    for j := 1 to w do
      can[j] := can[j] and b[j]
  end;
```

Таким образом, мы можем ответить на первый вопрос задачи — можно ли построить две стены:

```
f := false;
for i := 1 to w - 1 do
  if can[i] then
    begin
      f := true;
      v := i;
      break
    end;
if f then writeln('YES')
     else writeln('NO');
```

Теперь не представляет труда восстановить, из каких кирпичиков надо построить первую из двух стен. Для этого следует использовать так называемые "ссылки назад", т.е. при заполнении массива B для каждого возможного размера слоя j хранить, какой кирпич надо положить последним, чтобы получить такую длину слоя. Оставим конкретную реализацию в качестве упражнения.

Задача B

Блокада

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Максимальное время работы на одном тесте:	2 секунды

Государство Флатландия представляет собой прямоугольник размером $M \times N$, состоящий из единичных квадратиков. Флатландия разделена на K провинций ($2 \le K \le 100$). Каждая провинция представляет собой связное множество квадратиков, т.е. из каждой точки провинции можно дойти до любой другой ее точки, при этом разрешается переходить с квадратика на квадратик, только если они имеют общую сторону (общей вершины недостаточно). Во Флатландии нет точки, которая граничила бы более чем с тремя провинциями (т.е. четыре квадратика, имеющие общую вершину, не могут принадлежать четырем разным провинциям).

Каждая провинция имеет свой символ. Столица Флатландии находится в провинции, которой принадлежит символ A (заглавная латинская буква "A"). Провинция называется пограничной, если она содержит граничные квадратики. Провинция, в которой находится столица Флатландии, не является пограничной.

Король соседнего с Флатландией королевства Ректилания решил завоевать Флатландию. Для этого он хочет захватить столицу Флатландии. Однако он знает, что сил его армии недостаточно, чтобы сделать это сразу. Поэтому сначала он хочет окружить столичную провинцию, чтобы ослабить силы противника долгой блокадой, а потом захватить столицу.

Чтобы окружить провинцию, требуется захватить все провинции, с которыми она граничит. Две провинции граничат, если существуют два квадратика, имеющие общую сторону, один из которых принадлежит первой из них, а другой — второй. Чтобы захватить провинцию, надо чтобы выполнялось одно из двух условий: либо она пограничная, либо граничит с какой-либо уже захваченной провинцией.

Чтобы сберечь силы своей армии, король Ректилании хочет установить блокаду столичной провинции, захватив как можно меньше провинций. Помогите ему выяснить, сколько провинций требуется захватить. Захватывать столичную провинцию нельзя, поскольку для этого сил армии Ректилании пока недостаточно.

Формат входных данных

Первая строка содержит M и N ($3 \le M, N \le 200$). Следующие M строк содержат N символов каждая и задают карту Флатландии. Символ, находящийся в $(i + 1)$-й строке входного файла на j-м месте, представляет собой символ провинции, которой принадлежит квадратик (i, j). Все символы имеют ASCII-код больше 32 (пробела).

Формат выходных данных

Выведите в выходной файл единственное число — количество провинций, которые требуется захватить. Если установить блокаду невозможно, выведите "—1".

Примеры

input.txt	output.txt
5 6 BBBBBZ BCCCBZ BCAbbZ BDDDbZ 33333Z	4

Решение

Для решения представим Флатландию в виде неориентированного графа $G = <V, E>$, вершинами которого будут провинции, и две провинции соединим ребром, если они граничат (определение графа и некоторые связанные с этой структурой определения можно найти в решении задачи *F*).

Теперь наша задача — найти наименьшее по мощности (т.е. по количеству элементов) связное множество вершин, такое, что в него входят все вершины, соседние с вершиной, соответствующей столичной провинции, и хотя бы одна вершина, отвечающая граничной провинции, но не будет входить столичная вершина. Напомним, что множество вершин *V* называется связным, если существует путь из любой вершины $u \in V$ в любую другую его вершину $v \in V$, проходящий только по вершинам этого множества. Максимальное по включению связное множество графа (то есть такое, в которое уже нельзя добавить вершин без нарушения связности) называется компонентой связности.

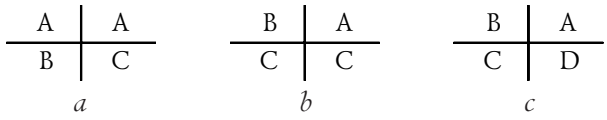
Для начала рассмотрим случай, когда этого нельзя сделать. Обозначим вершину, отвечающую столичной провинции, v_0. Удалим из графа эту вершину и рассмотрим получившийся граф $G' = <V' = V\backslash\{v_0\}, E'>$, где $E' = E|_{V' \times V'}$.

Предложение. Если граф G' несвязен, то построить искомое множество нельзя.

Доказательство. Действительно, исходный граф связен. Поскольку столичная провинция не является граничной, то все граничные вершины лежат в одной компоненте связности графа G' (мы можем добраться из одной в другую, следуя по граничным квадратикам). Значит, существует некоторое множество вершин, до которых нельзя добраться от граничных, не проходя через столичную вершину. Назовем такие вершины плохими. Тогда обязательно найдется хотя бы одна плохая вершина, соседняя со столичной в исходном графе. Последнее утверждение можно доказать так. Возьмем вершину v_{bad} из множества плохих вершин и какую-нибудь граничную вершину v_{border}. В исходном графе между ними существовал путь $v_{border}, v_1, v_2, ..., v_0, v_k, ..., v_{bad}$, проходящий через столичную вершину. Тогда следующая за последним появлением v_0 в этом пути вершина (v_k в нашем случае) и будет плохой и соседней со столичной. Значит, искомое множество построить нельзя.

Теперь перейдем ко второму случаю, когда граф G' связен (будем это полагать в дальнейшем тексте). В этом случае хотя бы одно множество, отвечающее нашим требованиям, существует (все множество V').

Отметим следующее свойство вершин, соседних со столичной вершиной: оно связно. Не приводя полностью формального доказательства этого факта, чтобы не запутывать изложение, рассмотрим рисунок, поясняющий его:



В случае, если переход от одной соседней провинции к другой осуществляется, как в случаях a) и b), связность не нарушается, а случай c) невозможен по условию.

Итак, чтобы множество вершин удовлетворяло нашим условиям, достаточно, чтобы оно содержало все соседние со столичной провинции, граничную провинцию и в нем существовал путь из граничной провинции в какую-нибудь соседнюю со столичной. Значит, ответом в задаче является $\min_{x \in N,\ y \in B} dist(x, y) + |N|$, где N — множество вершин, соседних со столичной, B — множество граничных вершин и $dist(x, y)$ — длина кратчайшего пути от вершины x до вершины y.

Таким образом, осталось привести способ нахождения длины кратчайшего пути от вершины x до вершины y для всех вершин $x \in N$ и $y \in B$. Для этого будем использовать алгоритм Флойда — Уоршелла, который находит длины кратчайших путей между всеми парами вершин. Алгоритм работает следующим образом: в двухмерном массиве a элемент $a[i][j]$ после k шагов алгоритма содержит длину кратчайшего пути из i-й вершины в j-ю, проходящего только по вершинам из множества $M_k = \{1, 2, ..., k\}$. Соответственно $(k + 1)$-й шаг алгоритма состоит в добавлении в это множество вершины $k + 1$ и пересчета матрицы a. Ниже приведена реализация алгоритма для графа с n вершинами, изначально a инициализируется матрицей смежности.

```
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      if ((a[i][k] <> 0) and
          (a[k][j] <> 0)) and
          ((a[i][j] = 0) or
          (a[i][k] + a[k][j] < a[i][j]))
      then
        a[i][j] := a[i][k] + a[k][j];
```

Приведем ключевые фрагменты программы (матрица смежности хранится в массиве can):

```
readln(m, n);
for i := 1 to m do
  for j := 1 to n do
    begin
      read(a[i][j]);
      {Массив u(u[char] of boolean) содержит true
       для символов, соответствующих какой-либо
        провинции}
```

```
      u[a[i][j]] := true
    end;
k := 0;
{Перенумеровываем вершины}
for c := #33 to #255 do
  if u[c] then
    begin
      inc(k);
      ind[c] := k
    end;
{Заполняем матрицу смежности, если ребра нет, храним -1}
for i := 1 to k do
  for j := 1 to k do
    can[i][j] := -1;
for i := 1 to m do
  for j := 1 to n do
    begin
      if (i < m) then
        begin
         can[ind[a[i][j]]][ind[a[i+1][j]]] := 1;
         can[ind[a[i+1][j]]][ind[a[i][j]]] := 1
        end;
      if (j < n) then
        begin
         can[ind[a[i][j]]][ind[a[i][j+1]]] := 1;
         can[ind[a[i][j+1]]][ind[a[i][j]]] := 1
        end
    end;
{Путь из себя в себя имеет длину 0}
for i := 1 to k do can[i][i] := 0;
{Находим граничные вершины}
for i := 1 to m do
  begin
    border[ind[a[i][1]]] := true;
    border[ind[a[i][n]]] := true
  end;
for j := 1 to n do
  begin
    border[ind[a[1][j]]] := true;
    border[ind[a[m][j]]] := true
  end;
{Находим столичную вершину}
main := ind['A'];
{Находим соседей столичной вершины}
answer := 0;
for i := 1 to k do
  if can[i][main] = 1 then
    begin
      neighbor[i] := true;
      inc(answer)
    end;
```

Алгоритм Флойда — Уоршелла с исключением из рассмотрения столичной вершины:

```
for p := 1 to k do
  if p <> main then
    for i := 1 to k do
      for j := 1 to k do
      if (can[i][p] <> -1) and (can[p][j] <> -1) and
        ((can[i][p] + can[p][j] < can[i][j]) or
         (can[i][j] = -1)) then
        can[i][j] := can[i][p] + can[p][j];
```

Находим, есть ли изолированные вершины, т.е. вершины, до которых нельзя добраться, не проходя через столичную:

```
for i := 1 to k do isolated[i] := true;
for i := 1 to k do
  for j := 1 to k do
    if border[i] and (can[i][j] <> -1) then
      isolated[j] := false;
```

Если такие нашлись, выводим —1, иначе находим и выводим ответ.

```
no := false;
for i := 1 to k do
  if isolated[i] then
    begin
      writeln(-1);
      no := true;
      break
    end;
if not no then
  begin
    best := maxlongint;
    for i := 1 to k do
      for j := 1 to k do
        if border[i] and neighbor[j] and
           (can[i][j] < best) then
          best := can[i][j];
    writeln(best + answer)
  end;
```

Задача C

Электронные часы

Имена входных файлов: `digits.txt, input.txt`
Имя выходного файла: `output.txt`
Максимальное время работы
на одном тесте: 2 секунды

Циферблат новых электронных часов, установленных на главном здании офиса фирмы Macrohard, состоит из 4 прямоугольных панелей, каждая из которых состоит из 6 рядов по 5 лампочек в каждом. Первые две панели отображают цифры, из которых складываются часы, а следующие две — минуты. (Если сейчас меньше 10 часов, первая панель отображает 0.)

К сожалению, лампочки, установленные на панелях, были произведены компанией Sveta.Net, которая известна своим принципом "*раньше перегорит — больше спрос*", вследствие чего на следующий день люди, проходя мимо офиса компании, видели лишь некоторое подобие цифр, поскольку некоторые лампочки больше не горели.

Петя живет в доме, стоящем прямо напротив офиса компании Macrohard. В первый день после установки часов он зарисовал у себя в блокноте, как выглядят все цифры на панелях (панели однотипные, поэтому одна и та же цифра на различных панелях выглядит одинаково). Теперь Петя хочет узнать, можно ли по текущему изображению на часах однозначно определить, сколько сейчас времени. Помогите ему.

Формат входных данных

При тестировании этой задачи в каталоге, который будет текущим, когда будет запущена ваша программа, будут находиться два файла. Файл `digits.txt` содержит 6 строк по 50 символов в каждой. Он будет одина-

ковым для всех тестов и будет совпадать с приведенным в примере. Содержимое файла digits.txt задает правильное написание цифр на панелях (первый прямоугольник 6 × 5 символов задает число 0, следующий — 1, и т.д. до 9). Не горящая лампочка обозначается символом "." (точка), а горящая — "#" (диез).

Входной файл input.txt содержит 6 строк по 20 символов в каждой — текущее изображение на часах. Первый прямоугольник 6 × 5 задает первую панель, следующий — вторую, следующий — третью и последний — четвертую.

Формат выходных данных

Если можно точно определить время, которое сейчас отображается на часах, выведите это время в формате hh:mm. Если время нельзя определить однозначно, выведите AMBIGUITY. Если же в часах точно сломалось еще что-то, например, центральный процессор, который управляет лампочками, выведите ERROR.

Примеры

digits.txt

```
..##.....#..##..####.#..#.####..###.####..##...##.
.#..#...##.#..#....#.#..#.#....#.......#.#..#.#..#
.#..#..#.#....#...#..#..#.###..###....#...##..#..#
.#..#....#...#.....#.####....#.#..#..#...#..#..###
.#..#....#..#......#....#.#..#.#..#..#...#..#....#
..##.....#.####.###.....#..##...##...#....##..###.
```

input.txt	output.txt
..##..####.#..#.#### ....#....#......... ....#...#..#..#.###. ...#.....#.##......# ..#......#....#.#..# .####..##.....#..#..	23:45
#.##.....#..##..#### .#..#...##.#..#....# .#.....#......#...#. .#..#..............# .#..#.......#......# ..##.....#.####.....	ERROR
..##........##..#### .#..#....#.#..#....# .#............#...#. .#..#..............# .#..#.......#......# ..##.......####.....	AMBIGUITY

Решение

Введем понятие "символ" как массив размера 6 × 5, в котором 1 соответствует горящей лампочке, а 0 — не горящей. Для двух символов A и B введем понятие их пересечения как массив $C = A \cdot B$, где $c[i][j] = a[i][j] \cdot b[i][j]$. Тогда верно следующее утверждение: чтобы символ A мог быть цифрой i, должно выполняться $A \cdot D_i = A$, где D_i — символ, соответствующий каноническому написанию цифры i.

Помня также, что время лежит в диапазоне 00:00—23:59, получаем следующий алгоритм решения задачи: для каждого возможного времени проверить, может ли оно быть отображено сейчас на табло (для этого каждую цифру на табло надо сравнить с соответствующей канонической цифрой). После этого: если ровно одно время может быть отображено на табло, то это и есть ответ, если более чем одно, то ответ AMBIGUITY, а если ни одно время не подошло, то ответ ERROR.

Приведем процедуру проверки того, что символ a может быть цифрой, содержащейся в символе b (в реализации не будем заменять "." на 0, а "#" на 1, а оставим, как есть):

```
type tdigit = array [1..6, 1..5] of char;
function check(a, b: tdigit): boolean;
var i, j: longint;
begin
  check := false;
  for i := 1 to 6 do
    for j := 1 to 5 do
      if (a[i][j] = '.') and
         (b[i][j] = '#') then exit;
  check := true
end;
```

Теперь проверяем, какое время может быть на табло: часы и минуты, и выводим ответ:

```
ch := 0;
for i := 0 to 23 do
  if can[1][i div 10] and can[2][i mod 10] then
    begin
      inc(ch);
      hh := i
    end;
cm := 0;
for i := 0 to 59 do
  if can[3][i div 10] and can[4][i mod 10] then
    begin
      inc(cm);
      mm := i
    end;
if (ch = 0) or (cm = 0) then writeln('ERROR')
else if (ch = 1) and (cm = 1) then
       writeln(hh div 10, hh mod 10, ':',
               mm div 10, mm mod 10)
     else writeln('AMBIGUITY');
```

Задача D

Сравнение URL

Имя входного файла: input.txt
Имя выходного файла: output.txt
Максимальное время работы на одном тесте: 2 секунды

Для идентификации ресурсов в сети Интернет используется URL (***U**niform **R**esource **L**ocator*). URL состоит из нескольких элементов: *протокол*, *хост*, *порт*, *путь*, *файл* и *секция*. Некоторые элементы URL могут быть опущены. Рассмотрим упрощенный формат URL:

[протокол://]хост[:порт][путь/[файл[#секция]]]

Заключенные в квадратные скобки элементы могут быть опущены, т.е., например, можно не указать протокол или секцию. Но, например, если указан файл, то обязательно должен быть указан путь. Регистр букв в элементах URL не важен.

Рассмотрим кратко все элементы URL:

Протокол — это способ доступа к файлу, URL с разными протоколами и одинаковыми остальными элементами могут указывать на различные ресурсы.

Хост и *порт* — это имена некоторого сервера в сети и способ доступа к нему (порт — натуральное число, не превосходящее 65 535).

Путь представляет собой путь к файлу, содержащему запрашиваемый ресурс от некоторого каталога на сервере, который называется *корневым*. При этом для разделения имен каталогов используется символ "/". Путь, если он не пуст, всегда начинается с символа "/". Специальное обозначение `'.'` соответствует самому каталогу, `'..'` — родительскому каталогу.

Файл — это файл, содержащий запрашиваемый ресурс.

Наконец, файл может быть разбит на *секции* каким-либо способом, и можно указать, к какой именно секции вы хотите обратиться.

Различные символы в URL могут быть заменены своими шестнадцатеричными ASCII-кодами с помощью символа %, например, `a = %41, Z = %5A`. В коде всегда используются ровно две шестнадцатеричные цифры.

Некоторые символы могут встречаться в элементах URL только как шестнадцатеричные коды — все символы, кроме букв латинского алфавита, цифр и символов "." и "-", а некоторые не могут встречаться вообще: "\", "#", "*", "@", "%", "?", ":", ",", — а также символы с ASCII-кодом меньше %20. Символ "/" может встречаться в элементах URL только в пути для разделения входящих в него каталогов. Имя файла не может состоять только из точек.

Рассмотрим примеры URL:

```
http://neerc.ifmo.ru/school
ftp://somewhere.net:1234/pub/files/coolgame.zip
nobody.nowhere.net/some%20dir/
                        some%20file#some%20info
```

Ваша цель в этой задаче — помочь разработчикам web-сервера. Для web-сервера отсутствующие части URL имеют следующие значения по умолчанию:

Протокол	http
Порт	80
Путь	*пустая строка*
Файл	index.html
Секция	*пустая строка*

Различные как строки URL могут указывать на один и тот же ресурс, например, следующие три URL:

```
neerc.ifmo.ru
http://neerc.ifmo.ru:80/index.html#
Http://NEERC.IFMO.Ru/Dir/.././../
```

Для разграничения доступа к ресурсам необходимо уметь определять, указывают ли два различных URL на один и тот же ресурс. Помогите разработчикам написать соответствующую проверку.

Формат входных данных

Входной файл состоит из двух строк, каждая из них содержит URL. Оба URL удовлетворяют формату, приведенному в условии этой задачи. Длина каждого URL не превосходит 200 символов. Гарантируется, что ни один из промежуточных каталогов на пути к ресурсу не лежит выше корневого каталога (т.е. не может встретиться, например, URL `http://somewhere.com/../dir/index.html`), а также что имена всех каталогов состоят по крайней мере из одного символа (два символа "/" не могут идти подряд в любом месте, кроме как непосредственно после двоеточия после имени протокола).

Формат выходных данных

Выведите YES в выходной файл, если оба URL, приведенные во входном файле, указывают на один и тот же ресурс, и NO в противном случае.

Примеры

input.txt	output.txt
http://neerc.ifmo.ru:80/index.html# neerc.ifmo.ru	YES
neerc.ifmo.ru/dir/../school NEERC.IFMO.RU/./SCHOOL	YES
neerc.ifmo.ru %6E%65%65%72%63%2E%69%66%6D%6F%2E%72%75	YES
nowhere.com somewhere.com	NO
http:// neerc.ifmo.ru ftp:// neerc.ifmo.ru	NO
neerc.ifmo.ru/index.htm neerc.ifmo.ru/index.html	NO
neerc.ifmo.ru:80 neerc.ifmo.ru:8080/index.html	NO
http:// somewhere.com somewhere.com/index.html#a	NO

Решение

Решение этой задачи представляет собой исключительно техническую проблему. Постепенно разделив заданные URL на части, составим из них затем URL в канонической форме (т.е. такой, где имеются все части, все латинские буквы находятся в верхнем регистре, и не встречается каталогов .. и .) и затем сравним их как строки.

Приведем функцию, приводящую URL к канонической форме:

```
function norm(x: string): string;
var z: integer;
    i, j: longint;
    tmp, protocol, port, host, path,
    filename, section, element: string;
begin
  if pos('://', x) <> 0 then
    begin
     protocol := copy(x, 1, pos('://', x) – 1);
     delete(x, 1, pos('://', x) + 2)
    end
  else protocol := 'http' end;
  if pos(':', x) <> 0 then
```

```
    begin
      i := pos(':', x);
      j := i;
      while (j <= length(x)) and
        (x[j] in ['0'..'9', ':']) do inc(j);
      port := copy(x, i + 1, j - i - 1);
      delete(x, i, j - i)
    end
  else port := '80' end;
  if pos('/', x) <> 0 then
    begin
        host := copy(x, 1, pos('/', x) - 1);
        delete(x, 1, pos('/', x))
    end
    else begin
           host := x;
           x := ''
         end;
  path := '';
  while (pos('/', x) <> 0) do
    begin
      element := copy(x, 1, pos('/', x) - 1);
      delete(x, 1, pos('/', x));
      if element = '..' then
        begin
          i := length(path);
          while path[i] <> '/' do dec(i);
         delete(path, i, length(path) - i + 1);
        end
      else if element <> '.' then
             path := path + '/' + element
    end;
  path := path + '/';
  if (x = '') then x := 'index.html';
  if pos('#', x) = 0 then x := x + '#';
  filename := copy(x, 1, pos('#', x) - 1);
  delete(x, 1, pos('#', x));
  section := x;
  x := protocol + '://' + host + ':' + port +
       path + filename + '#' + section;
  i := 1;
  while i < length(x) do
    begin
      if (x[i] = '%') then
        begin
          tmp := x[i + 1] + x[i + 2];
          val('$' + tmp, j, z);
          delete(x, i, 3);
          insert(char(j), x, i)
        end;
      inc(i)
    end;
  for i := 1 to length(x) do
    x[i] := upcase(x[i]);
  norm := x
end;
```

Задача E

Триангуляция Делоне

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Максимальное время работы на одном тесте:	10 секунд

Триангуляцией некоторого набора точек на плоскости называется набор невырожденных треугольников, удовлетворяющий следующим свойствам:

1. Вершинами треугольников являются только точки исходного набора. Каждая точка исходного набора является вершиной хотя бы одного треугольника.

2. Два различных треугольника либо не имеют общих точек, либо имеют общую вершину, либо имеют общую сторону (но площадь их пересечения всегда равна 0).

3. Любая точка, лежащая внутри выпуклой оболочки исходного набора точек, принадлежит хотя бы одному треугольнику (она может принадлежать нескольким треугольникам, если является их общей вершиной или принадлежит их общей стороне). (Выпуклой оболочкой некоторого набора точек называется наименьший выпуклый многоугольник, содержащий все эти точки.)

Триангуляция называется триангуляцией Делоне, если, кроме того, для нее выполняется следующее условие:

4. Внутри окружности, описанной около любого треугольника из триангуляции, не лежит ни одна из исходных точек (точки могут лежать на окружности, в частности, на ней, очевидно, лежат вершины рассматриваемого треугольника).

Для заданного набора точек найдите количество его триангуляций Делоне (две триангуляции считаются различными, если они отличаются хотя бы одним треугольником).

Формат входных данных

На первой строке входного файла находится число N — количество точек ($3 \le N \le 30$) исходного набора. Следующие N строк содержат по одной паре вещественных чисел — координаты соответствующей точки. Никакие три точки не лежат на одной прямой.

Формат выходных данных

Выведите в выходной файл количество различных триангуляций Делоне указанного набора точек.

Пример

input.txt	output.txt
4 0.0 0.0 1.0 0.0 0.0 1.0 1.0 1.0	2

Решение

Эта задача — самая сложная из предлагавшихся на олимпиаде. Для ее решения рассмотрим структуру, двойственную триангуляции Делоне: диаграмму Вороного. Пусть дан набор точек $P = \{p_i = (x_i, y_i)\}_{i=1}^{N}$.

Определение. Областью Вороного точки $p_i \in P$ относительно набора P называется множество точек p' плоскости, для которых расстояние

$$dist(p', p_i) = \min_{j=1,..,N} dist(p', p_j)$$

(т.е. множество точек, для которых она является ближайшей точкой из данного набора).

Разбиение плоскости на области Вороного для точек заданного набора называется "диаграммой Вороного данного набора". Отметим, что некоторые точки могут

принадлежать одновременно двум и более областям Вороного.

Диаграмма Вороного единственна для данного набора точек. Введем теперь понятие разбиения Делоне.

Определение. "Разбиением Делоне" назовем разбиение выпуклой оболочки заданного набора точек на выпуклые вписанные многоугольники, которое удовлетворяет свойствам 1—4 из условия с заменой слова "треугольник" на "многоугольник", и, кроме того, на окружности, описанной около каждого из многоугольников, лежат только вершины этого многоугольника.

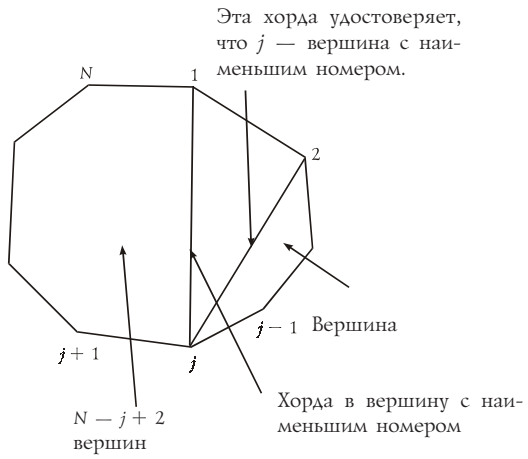
Тогда любая триангуляция Делоне является триангуляцией какого-либо разбиения Делоне. Оказывается, что разбиение Делоне единственно и однозначно строится по диаграмме Вороного. В дальнейшем тексте зафиксируем набор точек.

Предложение. Чтобы точка была центром окружности, описанной около какого-нибудь многоугольника разбиения Делоне, необходимо и достаточно, чтобы она принадлежала более чем двум областям Вороного.

Доказательство. Необходимость очевидна — центр описанной окружности равноудален от вершин многоугольника, около которого эта окружность описана, поскольку другие точки не лежат внутри окружности, расстояние до них больше, значит, эта точка принадлежит диаграммам Вороного сразу всех вершин многоугольника, т.е. более чем двум. Для доказательства достаточности построим соответствующее разбиение Делоне. Для каждой точки, которая принадлежит более чем двум областям Вороного, построим многоугольник, вершинами которого будут точки, диаграммам Вороного которого она принадлежит. Легко видеть, что полученное разбиение удовлетворяет свойствам 1—4 (доказательство этого утверждения оставим в качестве упражнения).

Заметим, что мы попутно доказали, что разбиение Делоне единственно (мы однозначно построили его по диаграмме Вороного, и любой многоугольник любого разбиения принадлежит нашему за счет необходимости). Итак, осталось подсчитать количество способов триангуляции разбиения Делоне. Для того чтобы триангулировать разбиение Делоне, необходимо триангулировать каждый входящий в него многоугольник. Рассмотрим метод подсчета количества способов триангулировать выпуклый многоугольник.

Используем динамическое программирование: пусть мы знаем количество способов, которыми можно триангулировать многоугольники с количеством вершин меньшим N, $C[i]$, для многоугольника с количеством вершин i, найдем количество способов триангулировать многоугольник с количеством вершин N. Рассмотрим вершину 1. Либо она принадлежит ровно одному треугольнику, в таком случае количество способов есть $C[N-1]$. В противном случае рассмотрим положение хорды, идущей в вершину с наименьшим номером, пусть это j. Если $j = 3$, то снова надо триангулировать многоугольник с количеством вершин $N-1$, иначе имеем количество способов триангулировать многоугольник с количеством вершин $j-1$ умножить на количество способов триангулировать многоугольник с количеством вершин $N-j+2$.



Итого,

$$C[N] = C[N-1] + C[N-1] + \sum_{j=4}^{N-1} C[j-1]C[N-j+2] =$$

$$= \sum_{j=3}^{N} C[j-1]C[N-j+2],$$

предполагая, что $C[2] = 1$.

Эти числа хорошо известны в комбинаторике, например в связи с правильными скобочными последовательностями, и называются "числами Каталана". Так, в наших обозначениях $C[i]$ есть $(i-2)$-е число Каталана.

Итак, решение задачи выглядит следующим образом: найдем все наборы точек, которые являются вершинами многоугольника из разбиения Делоне (для этого рассмотрим все тройки точек, для каждой тройки найдем множество точек, которые лежат на окружности, описанной около треугольника, вершинами которого являются эти точки; все такие наборы и будут искомыми множествами), и перемножим числа $C[|V_i|]$ для всех таких наборов V_i. Полученное число есть ответ. Отметим также, что, из-за того что числа Каталана растут чрезвычайно быстро, существуют наборы, для которых ответ не помещается в стандартные целые типы данных, соответственно необходимо реализовать длинную арифметику.

Задача F

Яблоко от яблони...

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Максимальное время работы на одном тесте:	2 секунды

У Пети в саду растет яблоня. Воодушевленный историей об Исааке Ньютоне, который, как известно, открыл закон всемирного тяготения после того, как ему на голову упало яблоко, Петя с целью повысить свою успеваемость по физике часто сидит под яблоней.

Однако поскольку по физике у Пети твердая "тройка", яблоки с его яблони падают следующим образом. В какой-то момент одно из яблок отрывается от ветки, на

которой оно висит, и начинает падать строго вниз. Если в некоторый момент оно задевает другое яблоко, то то тоже отрывается от своей ветки и начинает падать вниз, при этом первое яблоко не меняет направление своего падения. Вообще если любое падающее яблоко заденет другое яблоко на своем пути, то оно также начнет падать.

Таким образом, в любой момент каждое яблоко либо висит на ветке, либо падает строго вниз, причем все яблоки, кроме первого, чтобы начать падать, должны сначала соприкоснуться с каким-либо другим падающим яблоком.

Выясните, какие яблоки упадут с Петиной яблони.

Формат входных данных

Первая строка входного файла содержит N — количество яблок на Петиной яблоне ($1 \le N \le 200$). Следующие N строк содержат описания яблок. Будем считать все яблоки шарами. Каждое яблоко задается координатами своей самой верхней точки (той, где оно исходно прикреплено к дереву, длиной черенка пренебрежем) x_i, y_i и z_i и радиусом r_i ($-10\,000 \le x_i, y_i, z_i \le 10\,000$, $1 \le r_i \le 10\,000$, все числа целые). Гарантируется, что изначально никакие яблоки не пересекаются (даже не соприкасаются). Ось OZ направлена вверх.

Формат выходных данных

Выведите на первой строке выходного файла количество яблок, которые упадут с яблони, если начнет падать первое яблоко. На следующей строке выведите номера упавших яблок. Яблоки нумеруются, начиная с 1, в том порядке, в котором они заданы во входном файле.

Примеры

input.txt	output.txt
4 0 0 10 4 5 0 3 1 -7 4 7 1 0 1 2 6	3 1 2 4

Решение

Эта задача эквивалентна задаче о нахождении достижимого множества вершин в ориентированном графе. Рассмотрим поэтому последнюю задачу, а потом покажем, как наша задача сводится к ней.

"Графом" называется пара $G = \langle V, E \rangle$, где V — некоторое множество, которое называют множеством вершин графа, а E — отношение на V ($E \subset V \times V$) — множество ребер графа. Если отношение E симметричное (т.е. $(u, v) \in E \Leftrightarrow (v, u) \in E$), то граф называют неориентированным, в противном случае граф называют ориентированным.

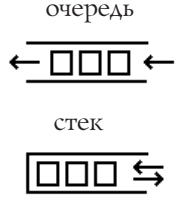
Обычно ориентированный граф изображают графически следующим образом: вершины графа (элементы множества V) изображают в виде точек или маленьких кружков, а ребра в виде стрелок — если $(u, v) \in E$ (т.е. вершины u и v соединены ребром), то из точки, изображающей вершину u, проводят стрелку в вершину v.

При программировании вершины графа обычно сопоставляют числам от 1 до N, где $N = |V|$ — количество вершин графа, и рассматривают $V = \{1, 2, ..., N\}$. Для хранения графа в программе можно применить различные методы. Самым простым является хранение матрицы смежности, т.е. двухмерного массива, скажем, A, где $A[i][j] = true$, если $(i, j) \in E$, и $A[i][j] = false$ в противном случае. Существуют также другие способы хранения графа, такие, как список ребер или матрица инцидентности, однако в нашей задаче будет достаточно матрицы смежности.

Путем в графе называют последовательность ребер графа $U = <(u_1, v_1), (u_2, v_2), ..., (u_l, v_l)>$, такую, что $v_1 = u_2, v_2 = u_3, ..., v_{l-1} = u_l$. Интуитивно понятие пути соответствует последовательности стрелок, в которой каждая следующая начинается из конца предыдущей. Часто также путем называют последовательность вершин $< u_1, u_2, ..., u_l, v_l >$. Начальная вершина u_1 называется началом пути и обозначается $\mathrm{beg}(U)$, конечная вершина v_l называется концом пути и обозначается $\mathrm{end}(U)$, l называется длиной пути и обозначается $\mathrm{len}(U)$. Также рассматривают пустой путь (путь, не содержащий ни одной вершины). Будем считать, что в графе N пустых путей, причем для каждой вершины $v \in V$ есть ровно один пустой путь U_v, для которого $\mathrm{beg}(U_v) = \mathrm{end}(U_v) = v$.

Рассмотрим некоторый ориентированный граф $G = \langle V, E \rangle$. Рассмотрим некоторую его вершину $v_0 \in V$. Тогда множеством вершин графа, достижимых из v_0, называется множество вершин, для которых есть путь из v_0 в эту вершину: $R\nu_0 = \{\nu \in V : \exists \text{ путь } U : \mathrm{beg}(U) = \nu_0, \mathrm{end}(U) = \nu\}$. Отметим, что благодаря нашему определению $v_0 \in R\nu_0$, поскольку существует пустой путь из v_0 в v_0. Например, для первой вершины графа, изображенного на рисунке, $R_1 = \{1, 2, 4\}$.

Рассмотрим задачу нахождения множества вершин, достижимых из данной. Для этого можно применить процедуру поиска в ширину или в глубину. Поиск в ширину осуществляется следующим образом: заводится очередь, в которую сначала помещается вершина v_0, при этом вершина v_0 помечается как обработанная. Затем, пока очередь непуста, производится следующая операция: из головы очереди извлекается вершина, все непомеченные вершины, в которые из нее ведет ребро, помечаются и помещаются в очередь. После окончания поиска в ширину множество помеченных вершин совпадает со множеством

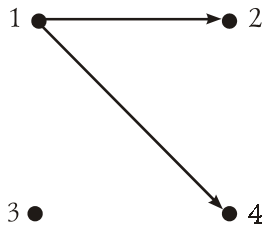


вершин, достижимых из данной. Поиск в глубину осуществляется аналогично, за тем исключением, что вместо очереди используется стек, т.е. вершины извлекаются из того же конца, куда и добавляются. Отметим, что реализация поиска в глубину в программе обычно проще, поскольку в качестве стека можно использовать встроенный стек языка программирования, реализуя поиск в виде рекурсивной процедуры:

```
procedure find(x: longint);
var i: longint;
begin
  u[x] := true;
  for i := 1 to n do
    if a[x][i] and not u[i] then
      find(i)
end;
```

Здесь x — номер вершины, которая сейчас обрабатывается (находится на вершине стека), a — матрица смежности, u — массив отметок. Весь поиск в глубину осуществляется тогда с помощью вызова `find(v0)`, где `v0` — номер начальной вершины.

Теперь рассмотрим, как наша задача сводится к задаче поиска множества достижимых вершин. Рассмотрим граф, в котором вершинами будут яблоки, из i-й вершины в j-ю имеется ребро, если i-е яблоко задевает j-е при своем падении. Тогда множество вершин, достижимых из первой, и будет множеством упавших яблок. Граф, соответствующий примеру, приведен на рисунке.



Осталось только выяснить, как проверить, что i-е яблоко $<(X_i, Y_i, Z_i), R_i>$ задевает при падении j-е $<(X_j, Y_j, Z_j), R_j>$. Для этого сначала сделаем так, чтобы координаты X, Y и Z соответствовали не верхней точке яблока, а его центру. Для этого отнимем R от Z у всех яблок. Теперь, предполагая, что Z_i и Z_j — координаты центра соответствующих яблок, для того чтобы проверить, что i-е яблоко задевает j-е, достаточно проверить, что

1) $Z_i > Z_j$;

2) проекции яблок на плоскость, параллельную OXY, пересекаются; а чтобы проверить, что два круга пересекаются, достаточно проверить, что

$$\sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2} \le R_i + R_j .$$

Приведем фрагмент программы, который вводит данные и преобразует их в матрицу смежности графа:

```
read(n);
for i := 1 to n do
  begin
    read(x[i], y[i], z[i], r[i]);
    z[i] := z[i] - r[i]
  end;
for j := 1 to n do
  for k := 1 to n do
    if (sqr(x[j] - x[k]) + sqr(y[j] - y[k]) <=
        sqr(r[j] + r[k])) and (z[k] < z[j])
      then a[j][k] := true;
```

Задача G

Контрольный блок

Имя входного файла: `input.txt`
Имя выходного файла: `output.txt`
Максимальное время работы на одном тесте: 2 секунды

Фирма Macrohard разработала новый протокол обмена данными по сети. Каждый блок данных при этом обмене состоит из N чисел в диапазоне от 0 до $M - 1$ включительно. Чтобы повысить надежность передачи, вместе с блоком данных пересылается контрольный блок такой же длины.

Предположим, что исходный блок состоит из чисел $a_1, a_2, \ldots, a_N$. Тогда контрольный блок состоит из чисел $b_1, b_2, \ldots, b_N$, из диапазона от 0 до $M - 1$ включительно, таких, что выполняются следующие равенства:

$$b_1 = (a_N + b_N) \bmod M,\ b_2 = (a_1 + b_1) \bmod M,$$

$$b_3 = (a_2 + b_2) \bmod M,\ \ldots,\ b_N = (a_{N-1} + b_{N-1}) \bmod M$$

(выражение $X \bmod M$ обозначает остаток от деления X на M, например, $7 \bmod 4 = 3$, $6 \bmod 2 = 0$).

Блоки данных, для которых нельзя построить контрольный блок, удовлетворяющий указанному свойству, считаются подозрительными и их передача по сети не разрешается.

Ваня хочет поступить на работу программистом в фирму Macrohard, и в качестве вступительного задания ему поручили написать процедуру построения контрольного блока для заданного блока данных. Помогите ему.

Формат входных данных

Первая строка входного файла содержит числа N и M ($1 \le N \le 1000$, $2 \le M \le 10^9$). Следующая строка содержит блок данных, для которого следует построить контрольный блок, числа разделены пробелами.

Формат выходных данных

На первой строке выходного файла выведите YES, если для данного блока данных можно построить контрольный блок, и NO, если нельзя. В случае, если контрольный блок построить можно, на второй строке выведите контрольный блок. Числа разделяйте пробелами. Если решений несколько, можно выдать любое из них.

Примеры

input.txt	output.txt
4 3 1 0 0 2	YES 1 2 2 2
4 3 1 1 1 1	NO

Решение

Прежде чем приступать к решению этой задачи, отметим одно свойство сложения по модулю: если

$A = B \bmod M$, то $(A + C) \bmod M = (B + C) \bmod M \quad (*)$.

Предположим, что для данного блока данных $a_1, a_2, ..., a_N$ существует какой-либо контрольный блок $b_1, b_2, ..., b_N$. Прибавим к каждому b_i число $(M - b_1)$ по модулю M. Получится новый блок данных той же длины

$$c_1 = (b_1 + (M - b_1)) \bmod M,$$

$$c_2 = (b_2 + (M - b_1)) \bmod M, ...,$$

$$c_N = (b_N + (M - b_1)) \bmod M.$$

Отметим, что $c_1 = 0$.

Благодаря свойству $(*)$ набор $c_1, c_2, ..., c_N$ также оказывается контрольным блоком для исходного блока данных $a_1, a_2, ..., a_N$. Таким образом, получаем следующее утверждение: для того чтобы для некоторого блока данных $a_1, a_2, ..., a_N$ существовал контрольный блок данных $b_1, b_2, ..., b_N$, необходимо и достаточно, чтобы существовал контрольный блок с $b_1 = 0$.

Теперь решение задачи не представляет никакого труда: зная b_1, мы без труда из равенства $b_2 = (a_1 + b_1) \bmod M$ находим b_2, зная b_2 — b_3, и т.д. Осталось только после нахождения b_N проверить, что выполняется равенство $b_1 = (a_N + b_N) \bmod M$.

Отметим также факт, что, для того чтобы существовал контрольный блок, необходимо и достаточно, чтобы $\sum_{i=1}^{N} a_i \bmod M = 0$, в чем легко убедиться, сложив равенства

$b_1 = (a_N + b_N) \bmod M$, $b_2 = (a_1 + b_1) \bmod M$,

$b_3 = (a_2 + b_2) \bmod M$, ..., $b_N = (a_{N-1} + b_{N-1}) \bmod M$

по модулю *M*.

Приведем полный текст программы:

```
program control_block;
var i, r, n, m: longint;
  a: array [0..1001] of longint;
begin
  assign(input, 'input.txt'); reset(input);
  assign(output, 'output.txt');
  rewrite(output);
  read(n, m);
  r := 0;
  for i := 1 to n do
    begin
      read(a[i]);
      r := (r + a[i]) mod m
    end;
  if r = 0 then
    begin
      writeln('YES');
      for i := 1 to n do
        begin
          write(r, ' ');
          r := (r + a[i]) mod m
        end
    end else writeln('NO')
end.
```

Задача H

Куб

Имя входного файла: `input.txt`
Имя выходного файла: `output.txt`
Максимальное время
работы на одном тесте: 10 секунд

Петя склеил из N^3 единичных кубиков большой куб размером $N \times N \times N$. Устав от этой сложной работы, он отправился спать, а утром, проснувшись, с ужасом обнаружил, что его младший брат Ваня *K* раз проткнул куб спицей.

При этом Ваня действовал очень аккуратно: каждый раз, установив конец спицы точно в центр грани какого-нибудь граничного единичного кубика, он протыкал куб параллельно соответствующей оси координат, при этом целый ряд из *N* кубиков оказывался испорчен.

Немного успокоившись после этого тяжелого потрясения, Петя заинтересовался, сколько кубиков в его творении осталось неповрежденными. Помогите ему ответить на этот сложный вопрос.

Формат входных данных

На первой строке входного файла находятся числа *N* и *K* ($1 \le N \le 1000$, $0 \le K \le 150$). Следующие *K* строк описывают Ванины преступные действия. Каждая строка содержит три числа — два из них представляют собой соответствующие координаты всех кубиков, проткнутых спицей, а третье, соответствующее координате, в направлении которой был проткнут куб, равно 0. Например, если $N = 3$, тройка (1, 0, 3) означает, что спицей были проткнуты кубики (1, 1, 3), (1, 2, 3) и (1, 3, 3). Все координаты лежат в пределах от 1 до *N*. Известно, что Ваня никакое действие не выполнял два раза (т.е. никакая тройка не встретится во входном файле дважды).

Формат выходных данных

Выведите в выходной файл единственное число — количество неповрежденных кубиков.

Пример

input.txt	output.txt
3 2 2 2 0 2 0 1	22

Решение

Разделим все кубики на четыре категории: кубики, которые были проткнуты спицей три раза, кубики, которые были проткнуты спицей два раза, один раз и кубики, которые не были проткнуты спицей вообще. Будем называть эти категории соответственно третьей, второй, первой и нулевой. Обозначим количество кубиков этих категорий за Q_3, Q_2, Q_1 и Q_0 соответственно. Оче-

видно, выполняется равенство $Q_3 + Q_2 + Q_1 + Q_0 = N^3$. Наша задача — узнать Q_0. Если мы узнаем Q_3, Q_2, Q_1, то мы сразу найдем $Q_0 = N^3 - Q_1 - Q_2 - Q_3$.

Отметим, что количество граней кубиков, проткнутых спицами, не зависит от Q_1, Q_2 и Q_3 и равно $2NK$. Отметим также, что у кубиков третьей категории проткнуто спицей 6 граней, у кубиков второй категории — 4, а у кубиков первой — 2. Отсюда получаем равенство

$$6Q_3 + 4Q_2 + 2Q_1 = 2NK$$

или, сокращая на 2,

$$3Q_3 + 2Q_2 + Q_1 = NK.$$

Теперь рассмотрим способ найти количество кубиков третьей категории. Для этого надо найти количество троек проколов, которые пересекаются в одной точке. Будем называть тройку чисел, которые задают прокол, координатами прокола. Отметим, что проколы с координатами (X_1, Y_1, Z_1), (X_2, Y_2, Z_2) и (X_3, Y_3, Z_3) пересекаются в одной точке, если выполняются следующие условия:

1. Нули в этих тройках должны стоять на разных позициях. Это можно записать как

$$X_1X_2X_3 + Y_1Y_2Y_3 + Z_1Z_2Z_3 = 0$$

(действительно — в каждой тройке ровно один 0, остальные числа — положительные, чтобы сумма трех неотрицательных чисел была равна 0, они все должны быть равны 0).

2. В каждой из трех ненулевых координат ненулевые координаты должны совпадать. Это можно записать в виде

$$\sum_{V=X,Y,Z} \sum_{i=1}^{3} \sum_{j=3}^{3} \left|V_i - V_j\right| V_i V_j = 0$$

(несложно убедиться, что если хотя бы одна пара ненулевых соответствующих координат не совпадает, соответствующее слагаемое не равно 0, в противном случае все слагаемые равны 0).

Таким образом, перебрав все тройки проколов и проверив для них выполнение указанных двух свойств, мы найдем Q_3.

Теперь найдем количество попарных пересечений проколов. Проколы с координатами (X_1, Y_1, Z_1) и (X_2, Y_2, Z_2) пересекаются в одной точке, если выполняются следующие условия:

1. Нулевые координаты в них не совпадают

$$\prod_{V=X,\,Y,\,Z} (V_1 + V_2) \neq 0.$$

2. Ненулевые координаты у них равны

$$\sum_{V=X,\,Y,\,Z} \left|V_1 - V_2\right| V_1 V_2 = 0.$$

Отметим, что на кубик третьей категории приходится три пары пересекающихся проколов, а на кубик второй категории — одна. Таким образом, если количество пар пересекающихся проколов P, получаем $3Q_3 + Q_2 = P$. Таким образом, получаем

$$Q_2 = P - 3Q_3,$$

$$Q_1 = NK - 3Q_3 - 2Q_2,$$

$$Q_0 = N^3 - Q_1 - Q_2 - Q_3.$$

Описанное решение не единственно, однако является одним из наиболее простых в реализации. Как альтернативу приведенным формулам можно рассматривать разбиение проколов на три категории в соответствии с направлением и дальнейший анализ троек и пар пересечений. Отметим лишь, что при указанных ограничениях ($N \le 1000$, $K \le 150$) альтернативные методы, например запоминающие уже проткнутые кубики или эмулирующие процесс протыкания с дальнейшим подсчетом оставшихся кубиков, неэффективны, так как требуют до $O(N^3)$ действий и от $O(K^3)$ до $O(N^3)$ памяти, ни то ни другое неприемлемо. Эффективность же описанного метода не зависит от N и есть $O(K^3)$.

Приведем полный текст программы:

```
program cube;
const maxm = 300;
var i, j, k, m, n, q1, q2, q3: longint;
   x, y, z: array [0..maxm] of longint;
function pair(x1, y1, z1, x2, y2, z2: longint):
boolean;
begin
  pair := ((x1 + x2) * (y1 + y2) * (z1 + z2) <> 0)
        and (abs(x1 - x2) * x1 * x2 +
             abs(y1 - y2) * y1 * y2 +
             abs(z1 - z2) * z1 * z2 = 0)
end;
function triple(x1, y1, z1, x2, y2, z2,
                x3, y3, z3: longint): boolean;
begin
    triple := (x1 * x2 * x3 + y1 * y2 *
               y3 + z1 * z2 * z3 = 0)
            and (abs(x1 - x2) * x1 * x2 +
                 abs(x1 - x3) * x1 * x3 +
                 abs(x2 - x3) * x2 * x3 +
                 abs(y1 - y2) * y1 * y2 +
                 abs(y1 - y3) * y1 * y3 +
                 abs(y2 - y3) * y2 * y3 +
                 abs(z1 - z2) * z1 * z2 +
                 abs(z1 - z3) * z1 * z3 +
                 abs(z2 - z3) * z2 * z3 = 0)
end;
begin
  assign(input, 'input.txt'); reset(input);
  assign(output, 'output.txt');
  rewrite(output);
  read(n, m);
  for i := 1 to m do
    read(x[i], y[i], z[i]);
    q3 := 0;
    for i := 1 to m do
      for j := i + 1 to m do
        for k := j + 1 to m do
          if triple(x[i], y[i], z[i],
                    x[j], y[j], z[j],
                    x[k], y[k], z[k])
          then inc(q3)
    for i := 1 to m do
      for j := i + 1 to m do
      if pair(x[i], y[i], z[i], x[j], y[j], z[j])
        then inc(q2);
    q2 := q2 - 3 * q3;
    q1 := m * n - 2 * q2 - 3 * q3;
    writeln(n * n * n - q1 - q2 - q3)
  end.
```

Рыбинская городская олимпиада школьников по информатике

В.Н. Пинаев,
г. Рыбинск

От редакции

В статье на примере рыбинской городской олимпиады по информатике показано, как провести городскую олимпиаду так, чтобы в ней смогли принять участие все желающие, по крайней мере в первом туре. Так в 2002 г. в рыбинской олимпиаде участвовали 118 школьников (для сравнения — в московской городской олимпиаде по информатике смогли принять участие всего около 40 школьников). Обеспечивается это в основном наличием теоретического тура и продуманным набором задач для него. Задания этого тура сформулированы так, чтобы в полной мере можно было проявить все свои знания и способности решать задачи из различных разделов информатики, причем не всегда напрямую связанных с программированием, а сложность проверки при этом относительно невелика. Задачи же практического тура позволяют отобрать достойных кандидатов для участия в областной олимпиаде.

Байт пишем — два в уме!

Читателю может показаться, что некоторые из описанных ниже задач ему уже знакомы. Что ж, он будет прав, идеи многих задач давно стали классикой и переходят из олимпиады в олимпиаду. Оригинальной в приведенных ниже задачах может быть либо основная идея, либо ее интерпретация, либо новая трактовка вопроса задания. При подборе и формулировке заданий неоценимую помощь оказал мне Алексей Филатов — студент Рыбинской государственной авиационной технологической академии.

Цель этой статьи — обсуждение задач и их решений, а также попытка обосновать методики проведения олимпиады и подбора самих задач.

Выделим некоторые методические особенности формулировки условий и организации олимпиады.

1. Во многих задачах выделены подзадачи. Такое разбиение задания на отдельные этапы призвано подсказать возможные пути решения и позволяет пройти участнику если не весь этот путь, то хотя бы его часть.
2. В текстах заданий часто приводятся конкретные алгоритмы. Предполагается, что этот прием повышает познавательный аспект олимпиады, так как, даже если участник не справится с заданием, он узнает новый для себя материал.
3. Во многих заданиях требуется обосновать то или иное свойство алгоритма. В такой трактовке задания становятся проще, чем если бы этот алгоритм требовалось написать самим участникам.
4. Предполагается, что во всех заданиях, если иное не оговорено, входные данные всегда корректны.
5. Если требуется найти рациональное решение, то это явно указывается в условии, причем обязательно оговаривается критерий оптимальности.
6. По положению о проведении городской олимпиады участник имеет право сдать на проверку решения не более четырех заданий. Это, с одной стороны, позволяет выбрать каждому задания себе по силам, с другой стороны, воспитывает у учащихся полезное качество оценивания сложности задания.

Ниже приводятся тексты заданий вместе с памяткой, которую получал каждый участник. После каждого задания приводится краткий разбор идей решений.

Теоретический тур городской школьной олимпиады по информатике

Памятка участнику

Время решения задач теоретического тура — 3 часа. На теоретическом туре разрешается пользоваться только письменными принадлежностями. В зачет идут результаты по не более чем четырем задачам, номера которых указывает сам участник. Максимальные баллы выставляются за полное, рациональное и обоснованное решение. Алгоритмы следует описывать с комментариями, не заменяя их текстами программ!

1. Архивация данных (1 + 2 + 4 = 7 баллов)

Как известно, на практике широко применяются специальные программы архивации данных. Такая программа-архиватор по заданному файлу создает новый файл меньшего размера. При разархивации из этого нового файла восстанавливается исходный файл.

① За счет чего удается "сжимать" файлы?

② Верно ли утверждение, что любой непустой файл может быть "сжат" подходящим архиватором? Ответ обоснуйте.

③ Предположим, у нас имеются десятичные записи числа π и дроби 22/7 со ста тысячами цифр каждая. Каждое из этих чисел записано в свой файл. Поясните, какой файл можно сжать лучше и почему.

Решение

① "Сжатие" файла выполняется за счет устранения избыточности в способе представления информации. Это становится возможным, если использовать более "экономную" кодировку.

② Если предположить, что любой файл может быть сжат, то тогда можно провести следующий эксперимент. Берем произвольный большой файл F_1, сжимаем его до файла F_2, этот файл вновь сжимаем до файла F_3 и т.д.

Понятно, что в силу конечности размера начального файла F_1 процесс сжатия рано или поздно закончится файлом нулевого размера. Если утверждение из условия задачи верно, то такой эксперимент можно проделать по отношению к любому исходному файлу. Следовательно, все файлы могут быть сжаты до нулевого размера. Так как в пустом файле вообще отсутствует какая-либо информация, то становится невозможным восстановить из него все исходные файлы. Поэтому утверждение неверно[1]. Утверждение было бы верным, если бы не требовалось восстанавливать исходные файлы, но такие архиваторы никому не нужны!

③ В первом пункте задания мы уже определились, что архивация данных — это устранение избыточности в способе их записи. Число π является иррациональным. В противоположность этому дробь $22/7$ — число рациональное и представимо в виде десятичной периодической дроби $3,(142857)$. Поэтому можно предположить, что используемые сегодня архиваторы[2] гораздо лучше "упакуют" число $22/7$, чем число π.

2. Схема Горнера[3] (6 баллов)

Известно, что значение многочлена

$$P = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

можно вычислить по так называемой "схеме Горнера":

1) `P := a[n]`
2) Для каждого i от $n - 1$ до 0 выполнить:

```
P := P * x + a[i]
```

Здесь через `a[i]` обозначается i-й коэффициент.

Докажите, что приведенный алгоритм действительно вычисляет значение многочлена при заданных коэффициентах и известному значению x.

Решение

Проще всего доказать правильность алгоритма, используя метод математической индукции.

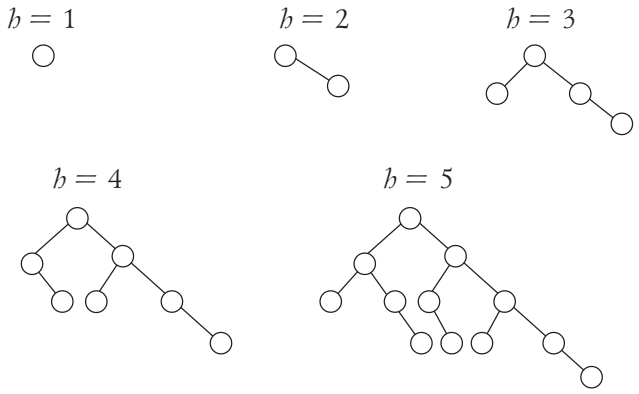
База индукции при $n = 0$ очевидна, так как в этом случае цикл ни разу не выполняется. Далее, делаем индукционное предположение, что при $n = k$ алгоритм вычисляет значение многочлена степени k правильно.

Остается только показать, что при $n = k + 1$ дополнительное повторение цикла переводит значение многочлена степени k (вычисленное по индукционному предположению) в значение многочлена степени $k + 1$.

Другой возможный подход — это выписывание вычисляемого значения с вложенными скобками, а потом их раскрытие. Но в конечном итоге аккуратное раскрытие скобок — это та же индукция.

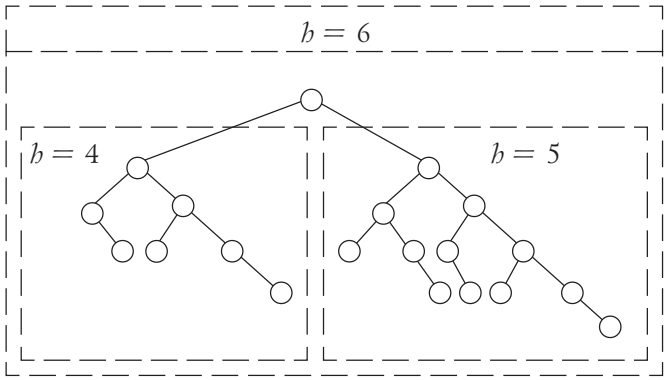
3. Деревья (2 + 3 = 5 баллов)

В информатике широко используются данные, объединенные в различные структуры. Рассмотрим так называемую "древовидную бинарную структуру". У такой структуры, если она не пуста, выделяется специальный узел, называемый корнем (на рисунке корень изображен вверху). К корню могут прикрепляться слева и справа такие же древовидные структуры (они называются левым и правым поддеревьями). На рисунке ниже изображены древовидные бинарные структуры при h, изменяющемся от 1 до 5. Они построены по определенному алгоритму.



① Догадайтесь, как построены эти структуры, и приведите древовидную структуру при $h = 6$.

② Опишите алгоритм построения таких структур по заданному параметру h.



Решение

Легко догадаться, что параметр h соответствует высоте дерева. Закономерность в построении деревьев очень проста: каждое следующее дерево получается из двух предыдущих приписыванием нового узла вверху (то есть просто строится новый корень дерева). На рисунке изображено дерево при $h = 6$. Благодаря принципу построения такие деревья называются "деревьями Фибоначчи". Алгоритм построения дерева Фибоначчи наиболее просто выписывается рекурсивно.

---

[1] Вообще говоря, любой исходный файл действительно может быть сжат до нулевого размера, но это возможно только тогда, когда программа-архиватор "знает" этот файл и рассчитана для распаковки только данного файла. Сравните, стоит ли вам конспектировать (то есть сжимать) текст, если вы его знаете наизусть!

[2] Можно повторить предыдущее примечание, что если архиватор "знает наизусть" требуемое количество цифр в записи каждого из чисел, то оба числа могут быть сжаты до их названий.

[3] Эта задача заранее была "обречена" на критику. В школьном, да и в вузовском курсах информатики очень мало уделяется внимания обоснованию правильности алгоритма. А сколько раз мы становились свидетелями, когда учащийся подходил к педагогу и торжествующе объявлял, что он написал требуемую программу. На вопрос: "А ты проверил?" — следовал обескураживающий ответ: "А зачем?".

Алгоритм "Построение дерева с параметром h"

1. Если $h < 1$, то дерево пустое.
2. Иначе
   2.1. Построить корневой узел.
   2.2. Вычислить $h1 = h - 1$, $h2 = h - 2$.
   2.3. Выполнить алгоритм "Построение дерева с параметром $h1$".
   2.4. Прикрепить построенное дерево к корню справа.
   2.5. Выполнить алгоритм "Построение дерева с параметром $h2$".
   2.6. Прикрепить построенное дерево к корню слева.

Второе возможное решение — итерационное. Для построения дерева высоты h воспользуемся предыдущим деревом высоты $h - 1$. Для этого переберем все узлы и для каждого из них выполним описанную ниже операцию (при этом возможны три варианта):

— от узла вниз направлены две дуги — ничего делать не нужно;

— от узла направлена вниз одна правая дуга — добавим дугу влево и прикрепим на ее конце новый узел;

— узел является концевым (нет выходящих из него дуг) — добавим дугу вправо и прикрепим на ее конце новый узел.

4. Проблема для жюри (9 + 1 = 10 баллов)

В городе N решили провести олимпиаду школьников по информатике. Для этого были выделены две большие аудитории в местном вузе. Жюри олимпиады стало известно, что школьники завидуют друг другу. Полная информация о том, кто кому завидует, доступна жюри. Причем, если школьник A завидует школьнику $Б$, то школьник $Б$ обязательно завидует школьнику A. Всего на олимпиаду собираются прийти 200 учеников, и каждый из них завидует ровно 101 будущему участнику олимпиады. Но самое страшное заключается в том, что если в одном кабинете окажутся участник и больше 50 учеников, которым он завидует, то этот участник лопнет от зависти.

Жюри решает эту проблему следующим образом.

1. Пусть каждый участник выберет себе аудиторию произвольно.
2. Пока есть участники, которые могут лопнуть от зависти, выполняем следующее действие 2.1, иначе переходим к пункту 3.
   2.1. Выберем любого из тех, кому грозит опасность, и пересадим его в другую аудиторию.
3. Начинаем олимпиаду.

① Докажите, что либо сформулированный выше алгоритм рано или поздно закончится, либо опровергните это и предложите свой вариант алгоритма рассаживания участников.

② Каким образом удобнее хранить информацию о том, кто кому завидует?

Решение

Первоначально предполагалось не формулировать в условии задачи алгоритма в явном виде. Но было решено, во-первых, упростить задачу, во-вторых, дополнительно обратить внимание участников на необходимость обоснования конечности алгоритма.

Будем считать, что наши аудитории отличаются цветом. Одна аудитория будет красной, другая — синей. Определимся с представлением информации о том, кто кому завидует. Для этого каждому участнику поставим в соответствие вершину графа. Дуги в этом графе будут "связывать" завидующих учеников. Среди всех дуг нас в первую очередь будут интересовать дуги, связывающие вершины одного цвета. Такие дуги будем называть ***опасными***.

Итак, участники рассажены произвольным образом. Пусть сейчас в красной аудитории есть участник X, которого требуется пересадить. Обратим внимание, что пересаживание одного из участников может привести к увеличению общего числа тех участников, кому грозит опасность! Пусть для пересаживаемого участника число опасных дуг равно T. Заметим, что $T \geq 51$, или, что то же самое, $2T \geq 102$. После пересаживания у участника X количество опасных дуг будет равно $101 - T$. В следующей таблице показано, как меняется количество опасных дуг.

	Число опасных дуг в красной аудитории	**Число опасных дуг в синей аудитории**	**Общее число опасных дуг**
До пересаживания участника X	R	B	$R + B$
После пересаживания участника X	$R - T$	$B + 101 - T$	$(R - T) + (B + 101 - T) = R + B + 101 - 2T < R + B$

Отсюда следует, что общее число опасных дуг на каждом шаге алгоритма уменьшается, оставаясь целым положительным числом. Следовательно, алгоритм конечен.

5. Шаг вперед, два шага назад (2 + 2 = 4 балла)

Робота поставили на прямую дорогу, протяженность которой в обе стороны бесконечна, и сообщили ему две целые положительные константы M и N. Первоначально положение робота задается числом 0. За 1 минуту робот может выполнить одну из следующих двух команд:

а) передвинуться вперед на M метров;

б) отойти назад на N метров.

Науке известно, что программа робота несовершенна, и он передвигается, чередуя команды A и B. Причем первой выполняется команда A.

Программа робота имеет вид (в скобках указаны комментарии):

1. $i := 1$ (задание начального номера команды).
2. Если расстояние от точки начала равно T метров, то "стоп".

3. Если i — нечетно, то выполнить команду "вперед на M метров", иначе "назад на N метров".

4. $i := i + 1$ (увеличение номера хода).

5. Перейти к пункту 2.

① Составьте рациональный по времени алгоритм определения, может ли робот когда-либо оказаться на расстоянии T ($T \geq 0$) от своего первоначального положения. Результатом выполнения алгоритма является наименьшее количество команд, необходимых роботу для этого, либо сообщение "это невозможно".

② Приведите в виде таблицы несколько примеров N, M и T с различными ответами.

№	Входные данные			Результат
	N	M	T	
1				
...				

Решение

Задание б) специально сформулировано для того, чтобы обратить внимание участников на необходимость заранее продуманного тестирования. Предполагалось, что, отвечая на этот пункт, участники попутно выявят возможные недостатки своего решения.

Рассмотрим варианты решения. Очевидно, что робот удалится на расстояние T метров либо за нечетное, либо за четное число шагов. Обозначим это число шагов через P. В первом случае должно выполняться равенство

$$|P(M - N)/2| = T,$$

во втором —

$$|(P - 1)(M - N)/2 + M| = T.$$

Решим эти уравнения. В первом случае получаем:

$$P_1 = 2T/|M - N|.$$

Второй случай при раскрытии модуля распадается на два подслучая:

$$P_{2,3} = 2(T \pm M)/(M - N) + 1.$$

Рациональный алгоритм задает вычисление трех возможных значений P_1, P_2, P_3, проверку найденных решений и выбор минимального из них.

Внимание: проверка решений означает, что из P_1, P_2, P_3 должны быть оставлены только натуральные числа, которые, будучи подставленными в исходные уравнения, превращают их в верные равенства!

6. Турнир (5 баллов)

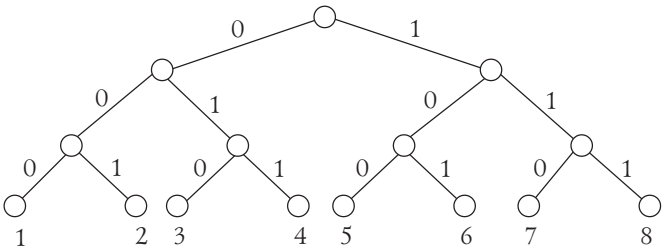
Хвастун Дима рассказывает всем знакомым, что участвовал в межгалактической олимпиаде по информатике. По его словам, на эту олимпиаду приехали 2^{30} участников со всех концов Галактики. Всем участникам "присвоили" номера от 1 до 2^{30}. Соревнование проходило по олимпийской системе: проигравший в споре двух участников "вылетал" из чемпионата. На первом этапе первый участник соревновался со вторым, третий с четвертым и так далее. В следующий круг выходили только победители. Олимпиада продолжалась до тех пор, пока не остался один участник, который и был объявлен чемпионом Галактики.

① Опишите алгоритм, определяющий, мог ли Дима, получивший номер i, встретиться с участником под номером k на j-м этапе турнира?

② Входные данные: i, j, k.

Решение

Представим себе схему проведения турнира в виде древовидной структуры. На нижнем уровне этой структуры в порядке слева направо представлены все участники межгалактической олимпиады по возрастанию их номеров. На вершине дерева обозначено имя победителя. Обозначим левые ветви дерева нулями, правые — единицами.



На рисунке показана схема турнира для восьми участников.

Легко увидеть, что если путь, ведущий от корня дерева к концевой вершине, рассматривать как двоичное число, то это число ровно на единицу меньше номера концевой вершины. (Было бы немного проще, если бы участникам межгалактической олимпиады присваивались номера, начиная с нуля.)

Заметим, что если два участника встретились на j-м этапе турнира, то далее двоичные разложения (уменьшенные на 1) их номеров совпадают.

В нашем задании количество участников таково, что для определения победителя потребуется 30 этапов. Отсюда следует искомый алгоритм.

1. Вычислить для значений $(i - 1)$ и $(k - 1)$ первые $(30 - j)$ разрядов.

2. Если эти разряды попарно не совпадают, то сообщить "встреча была невозможна".

3. Иначе вычислить $(30 - j + 1)$-е разряды.

   3.1. Если разряды совпадают, то сообщить "встреча была невозможна", иначе сообщить "встреча была возможна".

Алгоритм вычисления двоичных разрядов здесь опущен.

7. Загадочные числа (2 + 3 = 5 баллов)

Петя загадал 7 целых чисел: от —4 до 4 каждое. Хитрый Коля предложил Пете составить сумму по следующему алгоритму. Первое число умножают на 1, второе — на 10, третье — на 100, четвертое — на 1000 и так далее. А затем все эти числа складывают.

① Помогите Коле, зная результат сложения

$$S = -1\,417\,169,$$

найти все эти числа.

② Опишите рациональный по времени алгоритм восстановления исходных чисел по известной сумме.

Решение

В задании а) правильный ответ: 1, 3, —2, 3, —2, —4, —1.

Это задание напрямую связано с системами счисления. Запишем выполняемые Петей операции[4] в общем виде: $S = d_7 10^6 + d_6 10^5 + d_5 10^4 + d_4 10^3 + d_3 10^2 + + d_2 10 + d_1$. Здесь d_i — i-е загаданное число.

Авторское решение основано на стандартном алгоритме перевода десятичного числа в позиционную систему счисления, заключающемся в последовательном делении с остатком на основание системы (в Паскале для этого применяются операции **mod** и **div**). Однако в нашем случае есть одна особенность, связанная с наличием отрицательных значений у цифр в используемой системе. Можно заранее предположить, что немногие знают, как именно выполняются (и выполняются ли вообще!) названные операции в его любимом языке программирования при отрицательных операндах.

Например, в Турбо Паскале принято следующее решение. Сначала выполняется целочисленное деление для операндов без знака, потом частному приписывается знак по правилам деления, а остаток определяется как дополнение произведения частного на делитель до делимого. В таблице приведены примеры выполнения операций **mod** и **div**.

x	y	x div y	x mod y
23	5	4	3
—23	5	—4	—3
23	—5	—4	3
—23	—5	4	—3

Теперь, когда мы определились с правилами вычисления операций **mod** и **div**, перейдем непосредственно к алгоритму.

Понятно, что при выполнении операции **mod** мы можем получить значение, которое не вписывается в диапазон —4 .. +4, тогда потребуется скорректировать это значение. Важно, что частное от деления мы должны вычислять только после того, как вычтем из суммы S скорректированный остаток!

Ниже приводится соответствующий фрагмент программы.

```
for i := 1 to 7 do
 begin
   p := S mod 10;
   if p > 4 then p := p − 10
   else if p < -4 then p := p + 10;
   d[i] = p;
   S := (S − p) div 10
 end;
```

Можно пытаться двигаться по исходной сумме слева направо, восстанавливая исходные значения методом выделения максимальной степени 10 в записи числа. Но этот вариант реализовать не очень просто.

[4] Вообще-то можно было бы умножать исходные числа на степени девятки, так как всего существует девять значений: от —4 до +4. Однако было решено не усложнять задачу.

8. Шифровка (3 + 2 + 3 = 8 баллов)

В городе N разработана новейшая система шифрования упорядоченной по невозрастанию последовательности натуральных чисел, каждое из которых принадлежит диапазону от 1 до maxN. Для этого используется процедура, шифрующая массив Mas из maxN элементов. Закодированная последовательность размещается в массиве Temp. Соответствующий тип и заголовок этой процедуры имеют вид:

```
Type numbers = array[1..maxN] of integer;
Procedure Shifr(mas: numbers;
                var temp: numbers);
```

Основу процедуры кодирования составляет следующий алгоритм.

```
for i := 1 to maxN do Temp[i] := 0;
for i := 1 to maxN do
 if mas[i] <> 0 then
   temp[Mas[i]] := temp[Mas[i]] + 1;
for i := maxN downto 2 do
    temp[i − 1] := temp[i] + temp[i − 1];
```

Указание: упорядочивание по невозрастанию означает, что каждый следующий элемент не больше (то есть меньше либо равен) предыдущего.

① Сформулируйте алгоритм восстановления закодированной таким образом последовательности.

② Раскодируйте следующий массив:

temp = (10, 7, 6, 5, 5, 2, 2, 2, 0, 0), MaxN = 10.

③ Догадайтесь, почему в алгоритме предусмотрена проверка (выделенная жирным шрифтом) во втором цикле?

Решение

Можно аккуратно выписать все действия в обратном порядке, так как они обратимы. Такие решения оценивались как верные, но тогда участник не имел шансов заработать дополнительные баллы за пункт ③.

А действительно, почему предусмотрена проверка во втором цикле? Казалось бы, в этом нет никакой необходимости, так как все элементы входного массива отличны от нуля.

Дело в том, что декодировать массив можно, применяя повторно тот же самый алгоритм! Но если для восстановления применять процедуру кодирования, то нужно позаботиться о корректной обработке нулевых значений закодированного массива, которые там вполне могут быть! Удивительно, но два участника действительно догадались об этом. Обоснование этого "феномена" оставим для читателя.

Восстановленный массив в пункте ② имеет вид: (8, 8, 5, 5, 5, 3, 2, 1, 1, 1).

9. Произведения (7 + 1 = 8 баллов)

Даны две последовательности натуральных чисел. Из элементов последовательностей получаются всевозможные произведения пар чисел. Эти произведения выписываются в порядке возрастания.

Если одно и то же произведение может быть получено несколькими способами, то оно записывается только один раз.

① Опишите рациональный по времени алгоритм порождения по возрастанию всех возможных произведений, полученных при умножении пар сомножителей, взятых по одному из каждой последовательности.

② Приведите пример исходных и полученной последовательностей.

Решение

Это задание получило наименьший рейтинг. Отчасти это объясняется некоторой нечеткостью условия и плохим разбором[5].

Автором предполагалось, что решение не должно использовать дополнительную память размера $n \times m$, где n и m — количество различных элементов в исходных последовательностях. Однако об этом ограничении ничего не было сказано в условии.

Большинство участников, решавших это задание, предложили вычислить результаты всех произведений, а потом упорядочить их. Но даже и в этом варианте следовало предварительно упорядочить исходные последовательности (попутно удалив элементы-двойники).

Далее, если заполнить произведениями прямоугольную таблицу, то элементы в ней по строкам и столбцам будут упорядочены, чем следовало воспользоваться, так как по условию требуется рациональный по времени алгоритм. Поэтому задача сводилась к описанию алгоритма сортировки для упорядоченной по столбцам и строкам матрицы. Алгоритм сортировки можно сравнить с волной, которая накатывается от левого верхнего угла к правому нижнему. Для этого необходимо отслеживать фронт волны. Например, этот фронт можно представить себе ломаной линией, разбивающей таблицу на уже выведенные в выходную последовательность и оставшиеся элементы. На очередном шаге определяется минимальный из элементов фронта волны, размещенных в начале каждой "ступеньки".

Ниже приводится пример матрицы с фронтом волны после вывода элементов: 12, 15, 28, 35, 36, 48, 56.

		Первая последовательность				
		3	**7**	**12**	**14**	**27**
Вторая последовательность	**4**	12	28	48	56	108
	5	15	35	60	70	135
	12	36	84	144	168	324
	23	69	161	276	322	621

Примечание: при исключении очередного элемента из фронта волны необходимо также удалить все совпадающие с ним элементы.

[5] Разбор заданий проводился после олимпиады раздельно с педагогами и учащимися и длился около двух часов. Это задание было последним, и на разборе с учащимися я небрежно сказал: "А в этой задаче мы напутали с условием ...". Задача, имевшая и до этого невысокий рейтинг, стала "обречена". Для педагогов разбор этого задания проводил мой помощник из числа студентов, что, по-видимому, сказалось на понимании ими решения. Данный эпизод лишний раз доказывает важность и необходимость разбора решений олимпиадных задач.

Практический тур городской школьной олимпиады по информатике

Памятка участнику

- На практическом туре разрешается пользоваться только письменными принадлежностями и предоставленным компьютером.
- В зачет идут результаты тестирования только одного решения и только по одной задаче, номер которой указывает сам участник. Апелляция по результатам практического тура не предусмотрена, но участник имеет право присутствовать при тестировании сданного им решения.
- Время решения задач практического тура — 4 часа.
- Решение задач практического тура предусматривает ввод исходных данных из файла и вывод результатов в файл. Все файлы являются текстовыми.
- Время работы программы на любом тесте не может превышать 5 секунд.
- Файл исходных данных находится в текущем каталоге. Выходной файл также должен находиться в текущем каталоге.
- Формат входного и выходного файлов определен в условии задач. Жюри может принять от участника решение, в котором использован другой вариант ввода-вывода, но в этом случае участнику будут назначены штрафные баллы.
- Решение сдается в виде файла с исходным текстом на языке Turbo Pascal 7.0 и будет компилироваться пакетным компилятором TPC с опциями по умолчанию.
- Периодически сохраняйте свои файлы!

1. Спички (18 баллов)

Математик Петя играет в следующую игру. Первоначально он разложил некоторое количество спичек (n штук) по кучкам. Далее на каждом ходе игры Петя берет из каждой кучки по одной спичке и из них образует новую кучку. При этом какие-то кучки могут исчезнуть, так как в них не останется спичек. Так как Петя очень хороший математик, то он проанализировал игру и доказал, что при некоторых n (независимо от распределения по кучкам) гарантированно возникает **определенная, постоянно повторяющаяся комбинация кучек**. Эта комбинация кучек в последующем уже не может измениться, то есть следующий ход вновь порождает эту же комбинацию. Такую комбинацию мы будем называть заключительной.

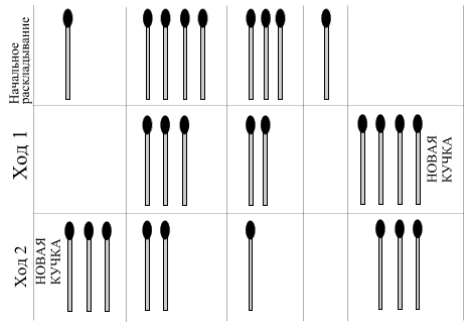
Ваша задача — написать программу, которая анализирует начальные данные и определяет:

1) минимальный (возможно, равный нулю) размер одной кучки, при добавлении которой к уже имеющимся кучкам гарантированно достижима заключительная комбинация;

2) число ходов, после выполнения которых возникает заключительная комбинация.

Пример

Ниже приведен пример первых двух ходов игры для 9 спичек и начальном раскладывании: первая кучка — 1 спичка, вторая — 4 спички, третья — 3 спички, четвертая — 1 спичка (нумерация кучек здесь условна).



Задание

Составьте программу, которая по входным данным — начальному количеству кучек (m) и перечислению количества спичек в кучках (S_1, S_2, ..., S_m) — вычисляет два неотрицательных числа:

1) минимальный размер кучки (R), при добавлении которой к уже имеющимся кучкам достижима заключительная комбинация;

2) число ходов (T), после выполнения которых возникает заключительная комбинация.

Ограничения

Входные данные всегда корректны и удовлетворяют ограничениям: $1 \le m \le 1000$;
$1 \le S_1 + S_2 + ... + S_m \le 1000$;
$S_i > 0$ для любого i.

Выходные данные — два неотрицательных числа.

Формат входного файла input.txt

Первая строка содержит количество кучек в начальном распределении (m). Последующие m строк содержат по одному положительному числу S_i на строке.

Формат выходного файла output.txt и строк, выводимых на экран

В выходной файл необходимо вывести два целых неотрицательных числа R и T, разделенных пробелом.

Пример входного файла input.txt

```
3
1
1
1
```

Пример выходного файла output.txt для предыдущего входного файла

```
0 2
```

Решение

Решение задачи основано на том факте, что заключительная комбинация кучек возможна только в том случае, если общее количество спичек представимо в виде $N(N+1)/2$. Сама комбинация имеет вид: 1, 2, 3, ..., N. Догадаться, что именно только такое распределение соответствует заключительной комбинации, довольно легко.

Математически точное обоснование этого можно найти в книге "Заочные математические олимпиады"[6]. Знание этого факта позволяет нам определить минимальное количество спичек для добавления. После определения размера новой кучки остается промоделировать игру.

Удобно это сделать следующим образом. Распределение спичек по кучкам представим массивом, в i-м элементе которого будем хранить количество кучек из i спичек. Дополнительно будем запоминать общее количество кучек.

Например, в таблице показано, какой вид будет иметь массив для примера из условия.

Индексы элементов массива	1	2	3	4	5	...
Значения элементов массива в начале игры	2	0	1	1	0	...
Значения элементов массива после первого хода	0	1	1	1	0	...
Значения элементов массива после второго хода	1	1	2	0	0	...

После очередного хода количество кучек увеличивается на 1 (за счет появления новой кучки) и уменьшается, если имелись кучки из одной спички. Каждая кучка после хода потеряет спичку, следовательно, все значения нашего массива просто сместятся на 1 позицию влево.

Остается проделывать такие ходы, пока не будет достигнута заключительная позиция.

2. Сумма (18 баллов)

Пусть в магазине цена любого товара определяется целым числом рублей.

По каждому товару вам известны его цена и количество. Определите минимальную сумму, которую вы не сможете целиком истратить в этом магазине.

Пример

Пусть в магазине имеются товары, информация о которых представлена в таблице ниже.

Номер товара	1	2	3	4	5
Цена	20	15	3	1	2
Количество товара	1	3	2	4	1

В этом примере, подбирая товар, можно сделать покупку на любую сумму от 1 до 12 рублей. Минимальная сумма, на которую покупку без сдачи сделать не удастся, равна 13 рублям.

Задание

Во входном файле задано общее количество различных единиц товара. Для каждого товара в файле заданы его цена и количество. Вывести в выходной файл сумму, которую невозможно истратить без остатка.

Ограничения

Количество пар (n) удовлетворяет условию: $1 \le n \le 1000$.

---

[6] *Васильев Н.Б., Гутенмахер В.Л., Раббот Ж.М., Тоом А.Л.* Заочные математические олимпиады. М.: Наука, 1985.

Цена (C_i) и количество (T_i) удовлетворяют условиям: $1 \le C_i \le 1000$, $1 \le T_i \le 1000$ при $1 \le i \le n$.

Все значения — натуральные числа.

Формат входного файла input.txt

Первая строка содержит количество пар (n). Последующие n строк содержат пары S_i, T_i, разделенные пробелами.

Формат выходного файла output.txt и строк, выводимых на экран

В выходной файл необходимо вывести одно целое положительное число.

Пример входного файла input.txt

```
5
20 1
15 3
3 2
1 4
2 1
```

Пример выходного файла output.txt для предыдущего входного файла

```
13
```

Решение

Представим товары, продающиеся в магазине, в виде массива записей. Каждая запись будет содержать сведения о стоимости и количестве данного товара. Упорядочим массив по возрастанию стоимости товаров (товары одной стоимости объединим). Очевидно, что если в магазине не оказалось товара стоимостью 1 рубль, то сумма в 1 рубль не может быть потрачена в магазине.

Пусть после анализа стоимости первых K товаров мы определили, что, покупая только эти товары, мы можем потратить полностью любую сумму от 1 до S рублей.

Рассмотрим стоимость следующего по цене $(K+1)$-го товара. Если этот товар стоит дороже $(S+1)$ рубля, то сумму в $(S+1)$ рубль мы не сможем полностью израсходовать. Это очевидно, потому что остальные товары суммарно стоят не меньше, чем $(K+1)$-й товар.

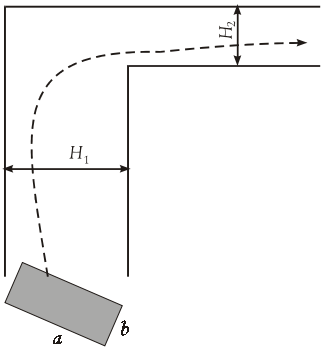
Если же стоимость $(K+1)$-го товара не превышает $(S+1)$ рубль, то, используя первые $(K+1)$ товаров, мы можем потратить любую сумму от 1 рубля до суммарной стоимости $(K+1)$ товара.

Из этого факта следует алгоритм решения задачи.

Будем продвигаться по нашему упорядоченному массиву товаров, каждый раз учитывая S — суммарную стоимость товара. Как только величина $(S+1)$ станет больше стоимости следующего товара или массив исчерпан, то печатаем ответ $(S+1)$.

3. Шкаф (30 баллов)

В свою новую квартиру вы решили доставить новый шкаф. Но вот беда: попасть домой можно только через Г-образный коридор, через который пройдет не любой шкаф! Один размер шкафа (a) вы заранее определили. Второй размер (b), который должен быть не больше первого, вы желаете выбрать максимально большим, но так, чтобы шкаф можно было пронести через коридор. Шкаф можно двигать и поворачивать как угодно, но запрещается отрывать его от пола.



Задание

Проанализируйте размеры коридора (H_1 и H_2) и размер длинной стороны шкафа (a) и выведите сообщение либо о максимально возможном втором размере (b) шкафа, либо выведите число "—1", если через коридор шкаф пронести невозможно.

Ограничения

Размеры коридора и длинной стороны шкафа заданы вещественными положительными числами с двумя знаками после десятичной точки и удовлетворяют условию: $1 \le H_1, H_2, a \le 100$.

Формат входного файла input.txt

В единственной строке файла через пробелы записаны три числа: H_1, H_2, a.

Формат выходного файла output.txt и строк, выводимых на экран

В выходной файл и на экран необходимо вывести одно число: максимально возможную ширину шкафа (с округлением до двух знаков после десятичной точки), либо число "—1", если шкаф пронести невозможно. Незначащие нули в записи числа можно не выводить.

Пример входного файла input.txt

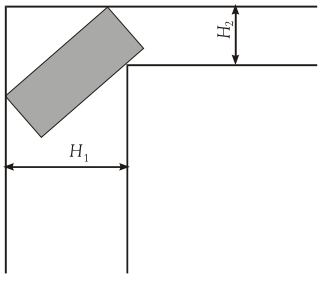
```
3 3 5
```

Пример выходного файла output.txt для предыдущего входного файла

```
1.74
```

Решение

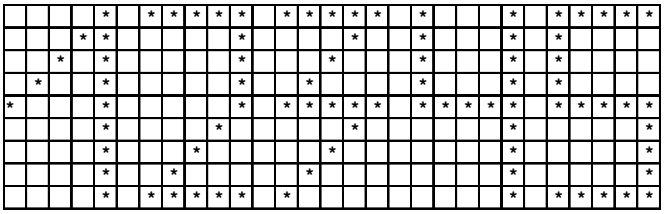
Авторское решение основано на таком моделировании перемещения шкафа, что два его угла с общей длинной стороной примыкают к внешней стене коридора (см. рисунок). При этом один из упомянутых углов смещается вдоль стены с достаточно малым приращением, например, $d = 0{,}001$. На каждом шаге вычисляется расстояние от угла внутренней стены до передвигаемой стенки шкафа и запоминается минимальное из полученных значений. При этом следует проверять, не пересекает ли угол внутренней стены коридора длинную стенку шкафа. Начальное значение ширины шкафа определяется равным величине H_1.

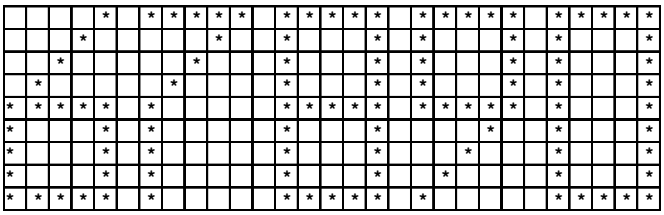


Особо следует рассмотреть случай, при котором $H_2 \ge a$, тогда шкаф не потребуется поворачивать, и результат b определяется равным минимуму из (H_1, a).

4. Почтовая арифметика (9 баллов)

Все мы умеем писать на конверте цифры почтового индекса. Правила записи этих цифр приведены ниже. Цифры в записи числа отделяются друг от друга одним столбиком пробелов. Ваша задача — распознать число, записанное в исходном файле звездочками по "почтовым правилам", и вывести в выходной файл максимальную цифру этого числа.





Задание

Проанализируйте число во входном файле и выведите в выходной файл максимальную цифру этого числа.

Ограничения

Во входном файле записано не более десяти цифр по почтовым правилам. Запись всех цифр корректна.

Формат входного файла input.txt

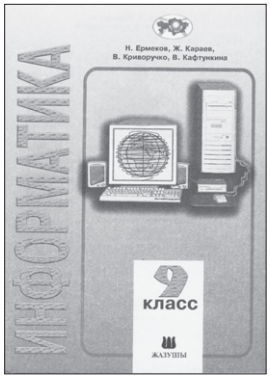
В первой строке входного файла записано количество (n) цифр в записи анализируемого числа: $1 \le n \le 10$. Каждая цифра занимает 5 столбиков, цифры отделяются друг от друга столбиком из пробелов. Последним символом в каждой строке записана точка. Таким образом, число в файле занимает 9 равных по длине строк, каждая из которых содержит звездочки, пробелы и точку в конце. Размер каждой строки равен $6n$.

Формат выходного файла output.txt и строк, выводимых на экран

В выходной файл и на экран необходимо вывести одну обычную десятичную цифру.

Решение

Эта задача рассматривалась как утешительная. Основная сложность ее решения состоит в отделении цифр, записанных подряд, друг от друга. Когда этот этап пройден, каждую цифру сравниваем с шаблоном и узнаем ее значение. Под шаблоны цифр можно завести отдельный массив констант. Однако можно поступить при определении значения цифры и по-другому. Например, цифру можно определить по общему количеству звездочек, из которых она состоит, и количеству звездочек в первом столбце. Более интересный способ состоит в выделении определенного числа клеток, по которым можно однозначно "опознать" цифру.



В № 9, 10, 11 была опубликована глава "Моделирование" из нового учебника по информатике для 9-го класса, который вышел в Казахстане. К сожалению, фамилия автора этих публикаций, одного из соавторов указанного учебника, была указана неверно. Мы приносим извинения ЕРМЕКОВУ Нурмухаммету Турлымовичу.

Гл. редактор
С.Л. Островский
Зам. гл. редактора
А.И. Сенокосов
Редакция:
Е.В. Андреева
Н.Л. Беленькая
Л.Н. Картвелишвили
Н.П. Медведева
Дизайн и верстка:
Н.И. Пронская
Корректоры:
Е.Л. Володина,
С.М. Подберезина

Адрес редакции и издателя:
121165, Киевская, 24
тел. 249-48-96
Отдел рекламы
тел. 249-98-70

ИНДЕКС ПОДПИСКИ
для индивидуальных подписчиков 32291
комплекта изданий 32744

Тел.: (095)249-31-38, 249-33-86. Факс (095)249-31-84

Учредитель: ООО "Чистые пруды"

Зарегистрировано в Министерстве РФ по делам печати. ПИ № 77-7230 от 12.04.2001.
Отпечатано в ОИД "Медиа-Пресса",
125993, ГСП-3, Москва, А-40, ул. "Правды", 24.
Тираж 7000 экз.
Срок подписания в печать по графику 13.03.2002.
Номер подписан 13.03.2002.
Заказ №
Цена свободная

Internet: inf@1september.ru
WWW: http://www.1september.ru

ИЗДАТЕЛЬСКИЙ ДОМ «ПЕРВОЕ СЕНТЯБРЯ», ГЛАВНЫЙ РЕДАКТОР — А.СОЛОВЕЙЧИК

Газеты ИЗДАТЕЛЬСКОГО ДОМА: **Первое сентября** — гл. ред. Е.Бирюкова, **Английский язык** — гл. ред. А.Громушкина, **Библиотека в школе** — гл. ред. О.Громова, **Биология** — гл. ред. Н.Иванова, **Воскресная школа** — гл. ред. монах Киприан (Ященко),**География** — гл. ред. О.Коротова, **Дошкольное образование** — гл. ред. М.Аромштам, **Здоровье детей** — гл. ред. А.Лекманов, **Информатика** — гл. ред. С.Островский, **Искусство** — гл. ред. Н.Исмаилова, **История** — гл. ред. А.Головатенко, **Литература** — гл. ред. Г.Красухин, **Математика** — гл. ред. И.Соловейчик, **Начальная школа** — гл. ред. М.Соловейчик, **Немецкий язык** — гл. ред. М.Бузоева, **Русский язык** — гл. ред. Л.Гончар, **Спорт в школе** — гл. ред. Н.Школьникова, **Управление школой** — гл. ред. А.Адамский, **Физика** — гл. ред. Н.Козлова, **Французский язык** — гл. ред. Г.Чесновицкая, **Химия** — гл. ред. О.Блохина, **Чудесная газета** — гл. ред. М.Аромштам, **Школьный психолог** — гл. ред. М.Сартан.