Вояковская Наталия Николаевна
Шафиров Максим Геннадьевич

ЗАДАЧИ МЕЖДУНАРОДНЫХ СТУДЕНЧЕСКИХ ОЛИМПИАД

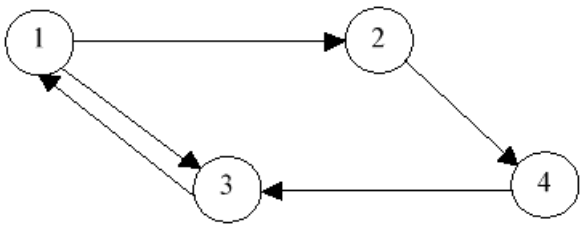
Как говорят сами участники олимпиад, задачи бывают следующих типов: технические задачи, задачи на знание алгоритмов, задачи на метод динамического программирования, переборные задачи и нормальные задачи, под которыми понимаются такие задачи, для решения которых необходимо основательное знание математики. Мы продемонстрируем решения некоторых задач, предлагавшихся на финале 24-й международной студенческой олимпиады по программированию ICPC. Тем читателям, которые хотят попробовать свои силы в решении подобного рода задач, мы рекомендуем обратиться к сайту htpp://www.ifmo.ru/neerc, на котором можно найти большое количество архивов с задачами.

В статье приводятся решения студентов математико-механического факультета Санкт-Петербургского государственного университета, которая в 2000 году стала чемпионом мира. Решение задачи 1 принадлежит Андрею Лопатину. Решения задач 2 и 3 предложил Николай Дуров. Все программы написаны на языке Pascal. Задачу 4 мы предлагаем вам решить самостоятельно. Вы можете прислать решения этих задач по электронной почте по адресу olymp@tepkom.ru. Желаем Вам успехов!

Задача 1. Page Hopping (Со страницы на страницу).

Недавно сообщалось, что в среднем необходимо только 19 нажатий мышки, чтобы попасть с любой страницы World Wide Web на любую другую. Это означает, что если страницы представить себе как узлы графа, то средняя длина пути между произвольными парами узлов графа равна 19.

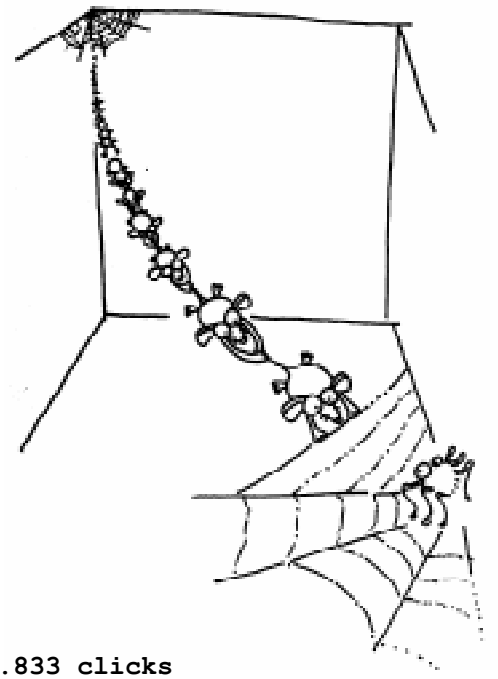
Дан граф, в котором все узлы доступны из любого начального узла. Ваша задача найти среднюю длину кратчайшего пути между произвольными парами узлов. В качестве примера рассмотрим следующий граф. Заметим, что связи между узлами изображаются как направленные ребра, поскольку связь между страницами *a* и *b* не означает, что страницы *b* и *a* также связаны.



Длины кратчайших путей из узла 1 в узлы 2, 3 и 4 равны 1, 1 и 2 соответственно. Кратчайшие пути, ведущие из узла 2 в узлы 1, 3 и 4, имеют длины 3, 2 и 1 соответственно. Кратчайшие пути, ведущие из узла 3 в узлы 1, 2 и 4, имеют длины 1, 2 и 3. И, наконец, длины кратчайших путей из узла 4 в узлы 1, 2 и 3 равны 2, 3 и 1. Сумма длин этих путей равна 1+1+2+3+2+1+1+2+3+2+3+1= 22. Поскольку имеется 12 возможных пар узлов, мы получаем среднюю длину пути 22/12 или 1.833 (с точностью до трех знаков после десятичной точки).

Вход.

Входные данные содержат несколько вариантов тестов. Каждый тест состоит из некоторого количества пар целых чисел, пара a, b означает, что страница a непосредственно связана со страницей b. Номера страниц заключены между 1 и 100. Каждый тест заканчивается парой нулей, которые не могут быть номерами страниц. Дополнительная пара нулей завершает входной поток. Граф не имеет петель, то есть никакой узел не связан сам с собой, и существует, по крайней мере, один путь между любыми двумя узлами графа.



Выход.

Для каждого теста выходной файл определяет среднюю длину кратчайшего пути между двумя произвольными узлами графа. Результат должен содержать три цифры после десятичной точки. Выходной файл состоит из номера теста (тесты нумеруются последовательно, начиная с 1) и средней длины в формате, показанном в примере.

Пример.

Вход.

```
1 2  2 4  1 3  3 1  4 3  0 0
1 2  1 4  4 2  2 7  7 1  0 0
0 0
```

Выход.

```
Case 1: average length between pages = 1.833 clicks
Case 2: average length between pages = 1.750 clicks
```

Решение.

Идея решения задачи весьма проста: нужно всего лишь найти сумму длин кратчайших путей между любыми двумя узлами графа и их количество, а в качестве ответа выдать частное от деления первого числа на второе. Проблема заключается в том, как искать кратчайшие пути. Можно поступить следующим образом: перебрать все узлы графа и для каждого узла найти длины кратчайших путей, ведущих из него во все остальные узлы, воспользовавшись для этой цели, например, алгоритмом Дейкстры. Наша команда решила воспользоваться другим алгоритмом, а именно, алгоритмом Флойда-Уоршелла (R.W.Floyd, S.Warshall). Основное отличие этого алгоритма от других алгоритмов, которые ищут пути из одного узла, заключается в том, что для него важно, какие узлы используются в качестве промежуточных узлов пути, и характеристикой пути является максимальный номер в нем промежуточной вершины.*

---

* Приведенное в статье описание алгоритма Флойда-Уоршолла может показаться трудным из-за использования мало знакомой учащимся школ терминологии теории графов (начальные сведения по теории графов изложены в журнале «Компьютерные инструменты в образовании» № 5, 6 за 1999 г.) В то же время алгоритм Флойда-Уоршелла достаточно прост, чтобы его можно было объяснить «на пальцах». Далее мы приводим это объяснение.

Алгоритм Флойда-Уоршелла решает задачу нахождения длин кратчайших путей между всеми парами узлов (вершин) графа. Идея алгоритма состоит в последовательном решении множества узлов, которые могут быть промежуточными в искомых путях. Для этого сначала рассматриваются пути, не имеющие промежуточных узлов – это ребра (дуги) графа. Поскольку два узла соединены в

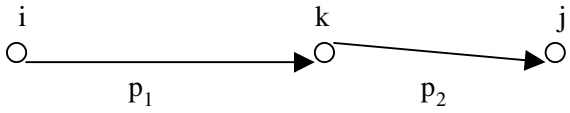
Будем рассматривать ориентированный взвешенный граф G = (V, E), то есть ориентированный граф с множеством вершин V и ребер E, каждому ребру которого приписано некоторое число, называемое весом ребра. Определим матрицу $W=(w_{ij})$ следующим образом:

$$w_{ij} = \begin{cases} 0, & \text{если } i = j \\ \text{вес (ориентированного) ребра } (i, j), & \text{если } i \neq j \text{ и } (i, j) \in E \\ \infty, & \text{если } i \neq j \text{ и } (i, j) \notin E \end{cases}$$

Под промежуточными узлами мы понимаем все узлы пути, которые не являются конечными, то есть промежуточными узлами простого пути $p = v_1, v_2, \ldots v_n$ являются все узлы $v_2, \ldots v_{n-1}$. Пусть узлы графа пронумерованы числами 1,2,...,*n*. Рассмотрим произвольное $1 \le k \le n$.

Для данной пары узлов i, j рассмотрим все пути из i в j, у которых все промежуточные узлы принадлежат множеству {1,2,...k}. Пусть p – путь минимального веса среди всех таких путей. Заметим, что этот путь будет простым, то есть в нем не будет повторяющихся вершин. Наша задача найти вес этого пути, зная веса всех таких путей для всех пар узлов при меньших значениях k. Имеются две возможности.

1. Если узел k не является промежуточным в p, то все промежуточные узлы пути p содержатся во множестве {1,2,...,k-1}. Значит, путь p является кратчайшим путем из i в j, промежуточные узлы которого принадлежат множеству {1,2,...k-1}.
2. Если k является промежуточным узлом пути p, он разбивает его на два участка p_1 и p_2.



Можно доказать, что путь p_1 является кратчайшим путем из i в k с промежуточными вершинами из множества {1, ..., k-1}, а путь p_2 является кратчайшим путем из k в j с промежуточными вершинами из множества {1, ..., k-1}.

Отсюда следует, что можно написать следующую рекуррентную формулу для длин кратчайших путей. Обозначим через $d_{ij}^{(k)}$ вес кратчайшего пути из вершины i в вершину j с промежуточными вершинами из множества {1, ..., k}.

графе ровно одним ребром – ребра и являются кратчайшими путями среди путей, не имеющих промежуточных вершин, поэтому в начале работы алгоритма на расстоянии d{i; j} берется длина ребра, соединяющего узлы i и j, а для пар узлов, не соединенных ребром, расстояние считается равным бесконечности.

Далее первый узел графа рассматривается в роли промежуточного (k=1) и перебираются все пары (i; j) узлов графа. Расстояния между ними модифицируются так: если длина пути через промежуточный узел короче расстояния, вычисленного ранее, то d{i; j} заменяется на d{i; k}+d{k; j}. После окончания этого цикла смысл значений d{i; j} изменится – это будут кратчайшие расстояния для всех путей, у котоорых промежуточным может быть первый узел.

Если проделать аналогичный цикл и модификацию для второго узла, то после окончания цикла значения d{i; j} будут равны кратчайшим расстояниям по путям, у которых промежуточными могут быть только первая и/или вторая вершина.

Повторяем процесс подключения новых промежуточных вершин до тех пор, пока все вершины не будут исчерпаны.

Очевидно, что по завершении этого процесса (цикла по k) d{i; j} и будут искомыми кратчайшими расстояниями между вершинами i и j. (*Примечание редакции*)

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{если } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{если } k \geq 1 \end{cases}$$

В нашем случае вес каждого ребра равен единице. Построим матрицу, размер которой равен n×n, где n – количество узлов графа, и заполним ее следующим образом:

- если узлы i и j соединены, то элемент матрицы с индексами i, j установим равным единице
- если узлы i и j не соединены, то положим соответствующий элемент матрицы равным «бесконечности», то есть некоторому большому числу, которое никогда не может быть равным длине пути

Следующий фрагмент программы устанавливает элемент матрицы с индексами i, j равным длине кратчайшего пути, ведущего из узла i в узел j:

```
for k:=1 to 100 do                {внешний цикл по вершине k}
   for i:=1 to 100 do             {пункт отправления}
      for j:=1 to 100 do          {пункт назначения}
          d[i, j] := Min (d[i, j], d[i, k]+d[k, j])
```

Собственно говоря, это и есть реализация алгоритма Флойда-Уоршолла. Полный текст программы содержится на дискете.

Задача 2. According to Bartjens (По словам Бартенса).

Широкое распространение калькуляторов и компьютеров имеет некоторые недостатки. Студенты теряют способность к вычислениям. Привыкшие использовать калькуляторы и компьютеры они не могут вычислить в уме, например, 7*8, или с помощью карандаша и бумаги 13*17. Мы все это знаем, но кого это беспокоит?

Профессора Бартенса беспокоит. Профессор Бартенс немного старомоден. Он решил немного потренировать своих студентов в вычислениях без использования электронного оборудования, для этого он создал набор вычислительных примеров (типа такого 2100-100=...). Для того, чтобы упростить процесс создания подобного рода примеров, он конструировал их таким образом, чтобы результаты почти всех из них были равны 2000. Но, конечно, не все примеры таковы. Его студенты достаточно умны, чтобы обнаружить закономерность и проставить 2000 в качестве ответа везде, даже не подумав.

К сожалению, драйвер печатающего устройства, который имел профессор Бартенс был еще более старомоден, чем сам профессор, и он не мог корректно работать с новым печатающим устройством. Изучив проблему печатающего устройства, профессор понял, что знаки операций не печатаются. То есть пример 2100-100= печатается как 2100100=. К счастью, все цифры и знак равенства печатались правильно.

Беда не приходит одна. Исходный файл профессора исчез. Таким образом, профессор столкнулся с более серьезной проблемой: какими были его исходные примеры. Известный факт, что результат вычислений (почти всегда) должен быть равен 2000, приводит к тому, что строка 2100100= соответствует одной из следующих строк:

2100-100=

2*100*10+0=

2+100*10-0=
2*10*0100=
2*-100*-10+0=

Профессор Бартенс запомнил некоторые вещи, которые он имел в виду, когда писал свои примеры:

- Он уверен, что никогда не писал числа (отличные от 0) с незначащими нулями, поэтому 2*10*0100 не мог быть одним из его примеров.
- Он также знает, что всегда писал число нуль просто как 0. Таким образом, пример 2*1000+000 неверен.
- Он использовал только бинарные операции, т.е. унарные + и – встретиться не могут, поэтому 2*-100*-10+0 неправильный пример.
- Он использовал только операции +, - и * и не использовал операцию /.
- Кроме того, он знал правила старшинства операций и ассоциативности.

Вы должны помочь профессору восстановить его примеры, написав программу, которая размещает в ряду цифр по крайней мере один знак операций +, -, * таким образом, чтобы значение получившегося выражения было равно 2000.

Вход.

Входной файл содержит один или более тестов. Один тест представляет собой последовательность из n цифр (‘0’, .., ‘9’), 1<=n<=9, за цифрами следует знак равенства. Между цифрами и между последней цифрой и знаком равенства не может быть пробелов, однако, некоторое количество пробелов может располагаться после знака равенства.

После последнего теста следует строка, состоящая из одного знака равенства. Эта строка не должна обрабатываться вашей программой.

Выход.

Для каждого теста вы должны напечатать слово Problem, номер теста и все возможные выражения, значения которых равны 2000. Используйте формат, показанный ниже. Если существует несколько выражений, то вы можете выводить их в любом порядке, однако никакой пример не может встретиться более одного раза. Каждый пример должен быть выведен в новой строке после двух пробелов. Если решения не существует, то вы должны напечатать слово IMPOSSIBLE, перед которым следует вывести два пробела.

Пример.

Вход	Выход
2100100=	**Problem 1**
77=	**2100-100=**
=	**2*100*10+0=**
	2*100*10-0=
	Problem 2
	IMPOSSIBLE

Решение.

Задача сводится к двум более мелким задачам, первая из которых заключается в разбиении заданной последовательности цифр на правильные подпоследовательности, а вторая – это вычисление значения формулы.

Начнем с решения второй задачи. Сформулируем ее следующим образом. Пусть функция getValue возвращает очередной операнд, из этих операндов мы должны с помощью операций +, -, * сконструировать формулу и проверить, равно ли ее значение некоторому числу (в нашем случае двум тысячам). Мы считаем, что функция

getValue настолько “умная”, что умеет перебирать все возможные значения, которые можно построить из заданной строки цифр.

Предположим вначале, что в формуле может использоваться только сложение, тогда ее значение вычисляется очень просто, а именно, мы каждый раз к уже вычисленному значению суммы должны добавлять новое число, то самое, которое нам дает процедура getValue. Таким образом, если мы заведем целочисленную переменную sum и обнулим ее вначале, то вычисление значения формулы сводится к исполнению следующего цикла

```
while числа не закончились do
               sum +:= getValue;
```

Однако, наша задача построить всевозможные формулы, значение которых было бы равно 2000. Поэтому мы не сможем обойтись только одной переменной. Нам потребуется как бы много переменных, а если говорить проще, то нам не остается ничего, кроме как воспользоваться рекурсией. Таким образом, в нашем упрощенном варианте мы можем описать такую рекурсивную процедуру:

```
procedure ComputeValue1 (T : integer);
          {параметр T используется для накапливания суммы}
var Value: integer;
begin
      if числа закончились
      then if T = 2000
           then вывод очередной формулы
           else begin
                 while числа не закончились do
                 begin
                       Value := getValue; {получили очередное число
                        {Теперь мы будем пытаться построить формулу,
                                          используя это значение}
                       ComputeValue1 (T+Value)
                 end
                   end
end;
```

Теперь мы усложним задачу, разрешив использовать еще и операцию умножения. Вначале вновь попробуем решить эту задачу для вычисления значения одной формулы, содержащей операции + и *. Понятно, что в данном случае нам не хватит одной переменной sum, а придется завести еще одну переменную prod и проинициализировать ее значением 1. Тогда мы сможем написать следующий цикл:

```
while числа не закончились do
      if очередная операция +
      then sum +:= prod * getValue; prod := 1
       else { очередная операция * }
           prod := prod * getValue
```

Поскольку мы должны получить всевозможные формулы, использующие операции + и *, то нам придется вновь описать рекурсивную процедуру.

```
procedure ComputeValue2 (T, M : integer);
       {параметр T используется для накапливания суммы,
        параметр M используется для накапливания произведения}
var Value: integer;
begin
      if числа закончилисль
```

```
      then if T = 2000
           then вывод очередной формулы
      else begin
            while числа не закончилисль do
            begin
                  Value := getValue;
                  {Получили очередное число.
                   Теперь мы будем пытаться построить формулу,
                   используя это значение.}
                 {Используем сложение для построения формулы}
                  ComputeValue2 (T+M*Value, 1);
                 {Используем умножение для построения формулы}
                  ComputeValue2 (T, M*Value)
            end;
      end
end;
```

Наконец, опишем процедуру ComputeValue, которая полностью решает задачу, то есть она позволяет использовать операции +, -, *.

```
procedure ComputeValue (T, M : integer);
          {Параметр T используется для накапливания суммы,
            параметр M – для накапливания произведения.}
var Value: integer;
begin
      if числа закончилисль
      then if T = 2000
           then вывод очередной формулы
 else begin
            while числа не закончились do
            begin
                  Value := getValue;
                  {Получили очередное число.
                   Теперь будем пытаться построить формулу,
                  используя это значение.}
                  {Используем сложение для построение формулы}
                   ComputeValue (T+M*Value, 1);

                  {Используем вычитание для построения формулы}
                   ComputeValue (T+M*Value, -1);

                 {Используем умножение для построения формулы}
                   ComputeValue (T, M*Value);
                 end;
      end
end;
```

Теперь перейдем к решению первой задачи, необходимому для полного решения задачи 2.

Наша цель – научиться генерировать всевозможные «правильные» числа по заданной последовательности цифр. Пусть нам задана последовательность цифр $S_1 ... S_n$

Определим матрицу для хранения чисел, которые в принципе могут быть операн-

$$A_{ij} = \begin{cases} 0, & \text{если } i \le j \text{ и } 0 \le i, j \le n \\ -1, & \text{если } i > j \text{ и } 0 \le i, j \le n \text{ и } S_{i+1} \ldots S_{i+(j-i)} - \text{неправильное число} \\ A_{ij} \times 10 + S_j, & \text{если } i > j \text{ и } 0 \le i, j \le n \text{ и } S_{i+1} \ldots S_{i+(j-i)} - \text{правильное число} \end{cases}$$

дами формул, следующим образом:

Значением –1 помечаются элементы матрицы, которые соответствуют либо строке, содержащей литеру, отличную от цифры, либо строке, начинающейся с последовательности 00.

Для примера, приведенного в условии задачи, мы получим матрицу:

0	2	21	210	2100	21001	210010	2100100
0	0	1	10	100	1001	10010	100100
0	0	0	0	–1	–1	–1	–1
0	0	0	0	0	–1	–1	–1
0	0	0	0	0	1	10	100
0	0	0	0	0	0	0	–1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0

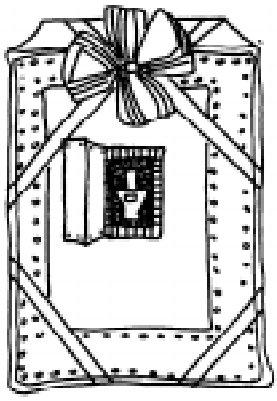
Для построения матрицы A опишем процедуру CreateMatrix.

```
procedure CreateMatrix;
var i, j: integer;
begin
      for i := 0 to n do A [i, j] := 0;{диагональные элементы
                                        устанавливаются в 0}

      for i := 0 to n do
          for j := 1 to n do
              if not (S[i] in ["0", "9"])
              then A [i, j] := -1
              else A [i, j] := A [i, j-1]*10+
                               (ord (S [i])-ord ("0"));

          {Этот цикл устанавли вает в –1 все элементы матрицы,
           которые соответствуют "неправильным" числам}
          for i := 0 to n-1 do
              if S [i] = "0"
              then for j := i+2 to n do
                       A [i, j] := -1;
          {Элемент с индексом 0 и n устанавливается в –1,
           поскольку формула должна содержать хотя бы один
            знак операции.}
          A [0, n] := -1
end;
```

Теперь нам осталось научиться обходить эту матрицу так, чтобы получать требуемые операнды. На дискете к журналу прилагается полный текст программы.

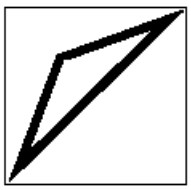
Задача 3. Gifts Large and Small (Подарки большие и маленькие).

Компания WrapIt.com специализируется на упаковке подарков. Начав несколько лет назад, как услуга, предлагаемая местным магазинам, сегодня WrapIt обслуживает клиентов по всему миру и утверждает, что может упаковать любую вещь от драгоценности в полкарата до целой квартиры.



WrapIt считает, что некоторые пользователи предпочитают упаковывать свои подарки в очень маленькие упаковки, а другие, напротив, хотят, чтобы их подарки выглядели значительно большими, чем они есть в действительности. Компании необходима программа для вычисления наибольшей и наименьшей прямоугольной коробки, в которую подарок может быть упакован «плотно». Поскольку это непростая проблема, компания решила вначале остановиться на двумерной версии программы.

Каждый подарок представляет собой простой многоугольник, а все упаковки – прямоугольники. Будем говорить, что подарок упакован плотно, если он касается всех четырех сторон упаковки. На картинках показано, каким образом треугольный подарок может быть плотно упакован в две упаковки различных размеров. Для каждого подарка ваша программа должна вычислить область наименьшего и наибольшего размера, в которую подарок может быть упакован плотно.



Вход.

Входной файл содержит описание нескольких подарков. Каждое описание начинается строкой, содержащей целое число n (3<=n<=100), которое задает количество углов многоугольника, представляющего подарок. Следующие n строк содержит пары целых чисел, которые определяют координаты углов, причем считается, что углы обходятся по часовой стрелке. Каждый многоугольник имеет ненулевую площадь. Ломаная, ограничивающая многоугольник, не имеет самопересечений.

Входной файл, заканчивается строкой, содержащей число 0.

Выход.

Для каждого подарка вначале выводится его номер, а затем на отдельных строках минимальная и максимальная площади его «плотной» упаковки. Вычисленные площади должны иметь три цифры после десятичной точки.

Пример.

Вход	Выход
3	**Gift 1**
-3 5	**Minimum area = 80.000**
7 9	**Maximum area = 200.000**
17 5	

```
4                                   Gift 2
10 10                               Minimum area = 100.000
10 20                               Maximum area = 200.000
20 20
20 10
0
```

Решение.

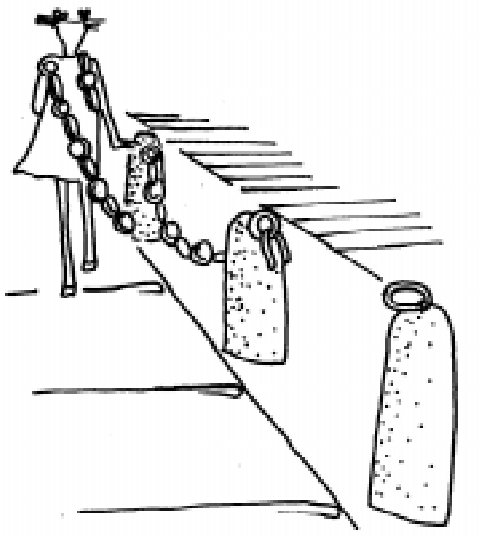
Это самая сложная задача из предлагавшихся на соревнованиях. Подробное объяснение решения займет слишком много места, поэтому мы изложим краткое описание алгоритма и приведем текст программы (на дискете), в котором читателю предлагается разобраться самому.

Легко понять, что угол α наклона одной из сторон («нижней» – к оси абцисс) прямоугольника однозначно задает его (с учетом необходимого условия касания всех его сторон многоугольником). Обозначим $f(\alpha)$ функцию, равную площади такого прямоугольника. Нам надо найти ее минимум и максимум на отрезке от нуля до 2π. Рассматриваемая функция – кусочно-гладкая и выпуклая вниз на каждом интервале монотонности. Исходя из этого, получаем решение задачи:

- вначале нужно найти все точки нарушения гладкости, отвечающие качественным скачкам в поведении прямоугольника при его вращении
- максимум функции равен ее максимуму на множестве точек разрыва, поскольку функция выпуклая
- минимум находится так: рассматривается множество точек нарушения гладкости, а также множество корней производной функции f, находящихся внутри интервалов гладкости. Минимум выбирается на этом конечном множестве.

Задача 4. Cutting Chains (Разорванные цепочки).

Что за находка! Анна Лок купила несколько звеньев цепочки, некоторые из которых могут быть соединены. Они сделаны из зоркиума – материала, который часто использовался для производства драгоценностей в прошлом веке, однако больше не используется для этих целей. Он имеет свой собственный, неповторимый блеск, который невозможно сравнить с блеском золота или серебра. Невозможно описать это тому, кто сам его не видел.



Анна хочет соединить звенья в одну нить. Она отнесла звенья ювелиру, который сказал ей, что стоимость соединения будет кардинально зависеть от того, сколько звеньев придется разъединить и соединить. Пытаясь минимизировать цену Анна тщательно вычисляет минимальное число звеньев, требующих разъединения или соединения. Оказалось, что это несколько более сложная задача, чем ей казалось сначала. Вы должны помочь ей решить эту проблему.

Вход.

Входной файл состоит из описаний наборов звеньев цепи, один набор на строке. Каждый набор представляет собой список целых чисел, разделенных одним или более пробелом. Каждое описание начинается целым числом n, которое обозначает количество звеньев в наборе ($1 \le n \le 15$). Мы будем обозначать звенья 1, 2, ..., n. Целые числа,

следующие за n описывают, какие звенья цепи связаны друг с другом. Каждое соединение задано парой целых чисел i, j, где 1≤i, j≤n и i≠j, в знак того, что звенья i и j связаны, то есть одно пропущено через другое. Описание каждого набора заканчивается парой –1 –1, которая не должна обрабатываться.

Входной файл заканчивается описанием, которое начинается с n=0. Это описание не должно обрабатываться и не будет содержать данных, описывающих соединения.

Выход.

Для каждого набора звеньев цепи выход должен содержать единственную строку в следующем формате

```
Set N: Minimum links to open is M
```

Где N – номер набора, а M – минимальное число связей звеньев, которые придется открыть и закрыть для того, чтобы получилась целая нить.

Пример.

Вход	Выход
`5 1 2 2 3 4 5 -1 -1`	`Set 1: Minimum links to open is 1`
`7 1 2 2 3 3 1 4 5 5 6 6 7 7 4 -1`	`Set 2: Minimum links to open is 2`
`4 1 2 1 3 1 4 -1 -1`	`Set 3: Minimum links to open is 1`
`3 1 2 2 3 3 1 -1 -1`	`Set 4: Minimum links to open is 1`
`3 1 2 2 1 -1 -1`	`Set 5: Minimum links to open is 1`
`0`	

Вояковская Наталия Николаевна, старший преподаватель кафедры системного программирования мат.-мех. факультета СПбГУ.

Шафиров Максим Геннадьевич, аспирант кафедры системного программирования мат.-мех. факультета СПбГУ.