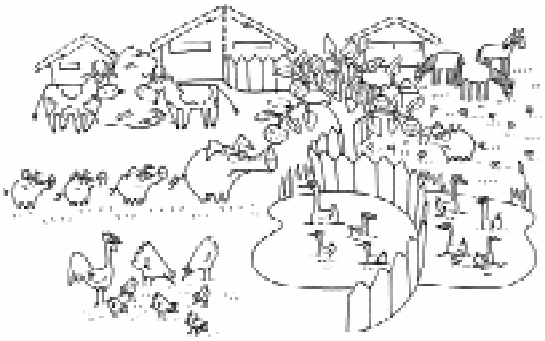
Станкевич Андрей Сергеевич

РАЗБОР ЗАДАЧ ПЯТОЙ ВСЕРОССИЙСКОЙ КОМАНДНОЙ ОЛИМПИАДЫ ШКОЛЬНИКОВ ПО ПРОГРАММИРОВАНИЮ

Окончание. Начало смотри в № 6 за 2004 год.

ЗАДАЧА F. ФЕРМЕРСКИЕ БУДНИ



Автор: *Сергей Оршанский, СПбГУ ИТМО.*

Год назад Флатландию потряс серьезный экономический кризис, после которого, с целью снижения издержек, фермер Джон и фермер Билл решили объединить свои фермы. За год совместной продуктивной работы на объединенной территории было построено четыре сарая.

Однако последствия кризиса в последнее время ощущаются все меньше, да и совместное ведение хозяйства – дело непростое. Поэтому фермеры решили снова разделить свои фермы.

Для раздела ферм решено было построить забор. Разумеется, прежде чем приступить к строительству, фермеры взяли карту совместного хозяйства и стали обсуждать возможное положение забора. Забор должен представлять собой прямую линию. Поскольку по закону границы участков должны быть направлены либо с севера на юг, либо с запада на восток, забор на карте должен представлять собой прямую, параллельную краю карты – вертикальную либо горизонтальную.

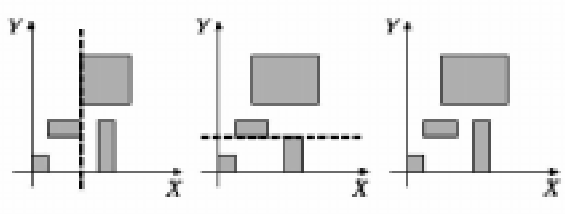


Рисунок 1.

Единственная проблема – четыре построенных за год совместной работы сарая. Разумеется, каждому фермеру в результате раздела должно достаться по два сарая. Поэтому после постройки забора два сарая должны оказаться с одной стороны от него, а два других – с другой.

Помогите фермерам найти такое положение для забора. Забор может проходить непосредственно вдоль стены сарая.

Приведенные на рисунке 1 варианты соответствуют примерам входных данных.

Формат входного файла

Входной файл содержит четыре четверки целых чисел. Каждая четверка описывает один сарай. Первые два числа в описании сарая – это координаты на карте его левого нижнего угла, следующие два числа – координаты правого верхнего угла.

Система координат размещена таким образом, что ось *OX* направлена слева направо, а ось *OY* – снизу вверх. Оси координат параллельны краям карты. Стороны сараев также параллельны краям карты. Сараи не имеют общих точек. Каждый сарай имеет ненулевую площадь. Все координаты неотрицательны и не превышают 10^9.

Формат выходного файла

Если построить забор, удовлетворяющий всем условиям, невозможно, выведите на первой строке выходного файла слово «Impossible».

В противном случае на первой строке выведите слово «Vertical», если забор следует построить параллельно вертикальному краю карты, или слово «Horizontal», если забор следует построить параллельно горизонтальному краю карты.

На следующей строке выведите координату x всех точек забора, если он должен быть вертикальным, либо координату y всех точек забора, если он должен быть горизонтальным. Выведенная координата должна быть целым числом (несложно показать, что если забор можно построить, то его можно построить так, чтобы искомая координата была целой).

Примеры

Входной файл	Выходной файл
0 0 1 1 1 2 3 3 4 0 5 3 3 4 6 7	Vertical 3
0 0 1 1 1 2 3 3 4 0 5 2 2 4 6 7	Horizontal 2
0 0 1 1 1 2 3 3 4 0 5 3 2 4 6 7	Impossible

Решение и программа

Одна из самых простых задач на олимпиаде задача «Фермерские будни» тем не менее требует некоторой аккуратности при написании. Хорошее владение техникой программирования позволяет существенно упростить реализацию.

Заметим, что если искомый забор вертикален, то коорината сараев по оси y не играют никакой роли. Аналогично, если забор горизонтален, то не играют роли координаты x.

Отсортируем сараи по x-координате левого нижнего угла. Теперь возможность разделить сараи вертикальным забором соответствует тому, что x-координата правого верхнего угла второго сарая не превышает x-координаты левого нижнего угла третьего сарая. В качестве координаты забора можно вывести, например, x-координату левого нижнего угла третьего сарая.

Аналогично можно проверить, что сараи можно разделить горизонтальным забором, отсортировав их по y-координате левого нижнего угла.

Приведем текст программы-решения полностью.

```
program farmers;
var
  x1, y1, x2, y2: array [1..4] of longint;
  i, j, t: longint;
begin
  reset(input, 'farm.in');
  rewrite(output, 'farm.out');
  for i := 1 to 4 do
    read(x1[i], y1[i], x2[i], y2[i]);
  for i := 1 to 4 do
    for j := i + 1 to 4 do
      if x1[i] > x1[j] then begin
        t := x1[i]; x1[i] := x1[j]; x1[j] := t;
        t := x2[i]; x2[i] := x2[j]; x2[j] := t;
      end;
  for i := 1 to 4 do
    for j := i + 1 to 4 do
      if y1[i] > y1[j] then begin
        t := y1[i]; y1[i] := y1[j]; y1[j] := t;
        t := y2[i]; y2[i] := y2[j]; y2[j] := t;
      end;
  if (x2[1] <= x1[3])
      and (x2[2] <= x1[3]) then begin
    writeln('Vertical');
    writeln(x1[3]);
  end else if (y2[1] <= y1[3])
      and (y2[2] <= y1[3]) then begin
    writeln('Horizontal');
    writeln(y1[3]);
  end else begin
    writeln('Impossible');
  end;
end.
```

ЗАДАЧА G.
НА ЛИФТАХ ЧЕРЕЗ ПРОПАСТЬ

Автор: *Сергей Оршанский, СПбГУ ИТМО.*

Сережа очень любит старые игры. Недавно он нашел у себя на компьютере

одну старую приключенческую игру. Управляя героем, надо перемещаться по карте и собирать различные предметы.

На определенном этапе игры Сережа столкнулся с неожиданной проблемой. Для продолжения приключений герою надо перебраться через пропасть. Для этого можно использовать последовательно расположенные лифты, которые имеют вид горизонтальных платформ. Каждый лифт вертикально перемещается туда-сюда между некоторыми уровнями. Герой может переходить между соседними платформами, однако это можно сделать только в тот момент, когда они находятся на одном уровне. Аналогично с края пропасти на лифт и обратно можно перейти лишь в тот момент, когда лифт окажется на уровне края.

Каждый лифт имеет ширину, равную четырем метрам. Вначале герой находится на расстоянии два метра от края пропасти. Он должен закончить свое путешествие в двух метрах от противоположного края пропасти. Герой перемещается со скоростью два метра в секунду. Таким образом, если герой находится в начальном положении или в центре лифта и хочет перейти на соседний лифт (или сойти с последнего лифта на противоположный край пропасти), он должен начать движение ровно за одну секунду до того момента, когда они окажутся на одном уровне. Через две секунды герой оказывается в центре соседнего лифта (или в конечном положении).

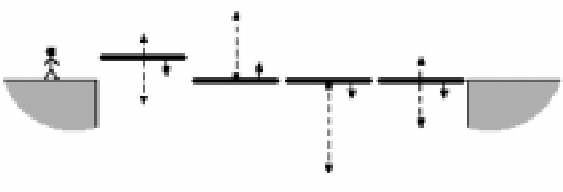


Рисунок 2.

Края пропасти находятся на одном уровне. Для каждого лифта задан диапазон высот, между которыми он перемещается, начальное положение и направление движения в начальный момент (рисунок 2). Все лифты перемещаются со скоростью один метр в секунду. Выясните, сможет ли герой перебраться на противоположный край пропасти, и если да, то какое минимальное время ему для этого понадобится.

Формат входного файла

Первая строка входного файла содержит целое число n – количество лифтов ($1 \le n \le 100$). Следующие n строк содержат описания лифтов.

Каждое описание состоит из четырех целых чисел: l, u, s – самое нижнее, самое верхнее и начальное положение лифта относительно края пропасти в метрах ($-100 \le l \le s \le u \le 100$, $l < u$), и d – направление движения лифта в начальный момент ($d = 1$ означает, что лифт двигается вверх, $d = -1$ – вниз).

Формат выходного файла

Выведите в выходной файл минимальное время в секундах, необходимое, чтобы перебраться на противоположный край пропасти. Если перебраться на противоположный край пропасти невозможно, выведите в выходной файл –1.

Пример

Входной файл	Выходной файл
4 -1 2 1 -1 0 3 0 1 -4 0 0 -1 -2 1 0 -1	29

Решение

В этой задаче требовалось аккуратно провести эмуляцию процесса, применив одну ключевую оптимизацию.

Для единообразия будем предполагать, что левый и правый края пропасти также

являются лифтами, двигающимися со скоростью 0 и находящимися на нулевом уровне. Будем, соответственно, считать их нулевым и $n + 1$-ым лифтом.

Сначала заметим, что всегда, когда можно перейти с платформы на следующую платформу, надо это сделать, а также никогда не требуется перемещаться назад. Поскольку лифты могут встречаться только спустя целое или полуцелое число секунд с начала процесса, удобно ввести новую единицу времени, равную половине секунды, назовем ее *полусекундой*. Если измерять время в полусекундах, то все интересные события будут происходить в целые моменты времени. Аналогично, положение лифтов относительного базового уровня удобно измерять в *полуметрах*.

В начале игры за две полусекунды переместимся вперед, встав ровно у края платформы. Каждый раз, зайдя на лифт, будем за четыре полусекунды перемещаться к его противоположному краю, таким образом, сразу, как только платформа соседнего лифта окажется рядом с нашей, мы сможем на нее переместиться.

Будем проводить посекундную эмуляцию процесса, перемещаясь вперед каждый раз, как только это возможно. Оказавшись на последней платформе (которая, напомним, соответствует правому краю пропасти), закончим процесс и выведем текущее время.

Осталось две проблемы, которые следует решить. Первая состоит в том, что бывают ситуации, когда добраться до противоположного края платформы нельзя. Такой случай возникает, например, когда диапазоны перемещения соседних платформ не пересекаются или пересекаются, но лифты, тем не менее, никогда не встречаются. Чтобы отследить такой случай, возможны два подхода. Первый состоит в том, что можно попытаться, зная диапазоны, начальные положения и направления лифтов, определить, могут ли они встретиться. Однако попытка выписать условия возможности встречи наталкивается на существенные сложности. Другой вариант состоит в том, чтобы вычислить максимальное возможное время пребывания на платформе перед переходом на следующую, и, если это время прошло, сделать вывод, что платформы никогда не встретятся.

Используем второй способ. Максимальное время между последовательными встречами двух соседних лифтов возникает, если их диапазоны движения пересекаются по одной точке, и равно наименьшему общему кратному периодов их движения. Заметим, что сумма длин диапазонов не превышает 400 полуметров (нижняя и верхняя точки диапазонов движения лифтов по условию не превышают по модулю 100 метров) и длины диапазонов четны. Следовательно, максимум наименьшего общего кратного периодов достигается для диапазонов длиной 198 и 202 и равен $2 \cdot 2 \cdot 99 \cdot 101 = 39996$ полусекундам (вторая двойка возникла из-за того, что период в полусекундах равен удвоенной длине диапазона в полуметрах). Итак, если герой пребывает на лифте дольше 39996 полусекунд, то он не сможет добраться до противоположного края пропасти.

Вторая проблема состоит в том, что при непосредственной эмуляции всех лифтов ограничения по времени, отведенного на время работы программы, может не хватить. Действительно, как было показано выше, ожидание на одном лифте может достигать 39996 полусекунд, что для 100 лифтов составлет порядка $4 \cdot 10^6$ полусекунд. Эмулируя четыре миллиона перемещений для 100 лифтов, нам придется совершить порядка 400 миллионов операций, при этом операции являются не самыми элементарными (нам надо еще следить, не достиг ли лифт крайней точки своей траектории, в этом случае следует сменить текущее направление его движения).

Решение состоит в том, чтобы эмулировать в каждый момент только два лифта: тот, на котором находится герой, и следующий, на который он хочет перейти. Такое решение позволяет существенно (в крайнем случае – до 50 раз) сократить объем вычислений и уложиться в отведенные временные рамки. При переходе на очередной лифт следует найти текущее положение следующего лифта, эмуляция перемещений кото-

рого ранее не производилась. Чтобы это сделать, заметим, что перемещения лифта периодичны, с периодом $2(u - l)$, где u и l – верхняя и нижняя точки его траектории. Пусть нам надо найти положение лифта через t полусекунд. Для этого достаточно проэмулировать $t \bmod 2(u - l)$ шагов лифта от его начального положения.

Программа

Перейдем к описанию реализации. Прочитаем входные данные и удвоим большинство входных параметров, переходя к полуметрам и полусекундам. Инициализируем также фиктивные нулевой и $n + 1$-ый лифты.

```
read(n);
for i := 1 to n do begin
  read(l[i], u[i], s[i], d[i]);
  u[i] := u[i] * 2;
  l[i] := l[i] * 2;
  s[i] := s[i] * 2;
end;
l[0] := 0;
u[0] := 0;
s[0] := 0;
d[0] := 0;
l[n + 1] := 0;
u[n + 1] := 0;
s[n + 1] := 0;
d[n + 1] := 0;
```

В массиве `tl` будем хранить количество времени, которое уже проэмулировано для каждого лифта. Исходно это ноль для всех лифтов.

```
for i := 1 to n do
  tl[i] := 0;
```

Напишем процедуру, которая находит позицию лифта i в момент времени t.

```
procedure golift(i, t: integer);
var
  j, dt: integer;
begin
  dt := t - tl[i];
  tl[i] := t;
  dt := dt mod ((u[i] - l[i]) * 2);
  for j := 1 to dt do begin
    s[i] := s[i] + d[i];
    if (s[i] = l[i]) or (s[i] = u[i])
                  then d[i] := -d[i];
  end;
end;
```

Теперь перейдем к непосредственной эмуляции. Константа `maxcount` хранит максимальное время, которое можно провести на одном лифте. Начальное значение переменной `t`, которая хранит текущее время в полусекундах, соответствует подходу к краю пропасти. Увеличение `t` на 4 при переходе на соседний лифт соответствует времени, которое надо потратить на то, чтобы пройти от одного края платформы до другого.

Поскольку ответ посчитан в полусекундах, его надо поделить пополам. Вычитание единицы из ответа соответствует тому, что нам надо отойти на два, а не на четыре метра от края пропасти.

```
t := 2;
for i := 1 to n + 1 do begin
  golift(i - 1, t);
  golift(i, t);
  count := 0;
  while (s[i - 1] <> s[i])
      and (count < maxcount) do begin
    inc(t);
    golift(i - 1, t);
    golift(i, t);
    inc(count);
  end;
  if s[i - 1] = s[i] then begin
    inc(t, 4);
end else begin
    writeln(-1);
    exit;
  end;
end;
writeln(t div 2 - 1);
```

ЗАДАЧА H. НАНОКРИСТАЛЛ

Автор: *Петр Митричев, МГУ.*

Сверхсекретный завод, расположенный высоко в горах, занимается изготовлением новейших систем контроля торсион-

ных полей – нанокристаллов. Нанокристалл состоит из нескольких атомов, некоторые из них попарно связаны сверхпрочными торсионными связями.

Нанокристалл стабилен, если между любыми двумя его атомами можно построить соединяющую их цепочку связей, возможно, с использованием других атомов. Например, нанокристалл X из четырех атомов A, B, C и D, в котором между собой связаны пары $A - B$, $A - C$, $B - C$ и $B - D$, стабилен. Если же, например, в нанокристалле из данных четырех атомов связаны только пары $A - B$ и $C - D$, то кристалл нестабилен, поскольку, например, A и C не соединены никакой цепочкой связей.

Для любой пары атомов стабильного нанокристалла определена их взаимная удаленность – минимальная длина цепочки из связей, которая их соединяет. Например, рассмотрим описанный выше нанокристалл X. Взаимная удаленность атомов A и B равна единице (они соединены напрямую), а взаимная удаленность C и D равна двум (они соединены цепочками $C - B - D$ и $C - A - B - D$, длина кратчайшей цепочки равна двум).

Важнейшей характерикой стабильного нанокристалла является его емкость. Емкость нанокристалла равна сумме взаимных удаленностей всех пар его атомов. Например, емкость нанокристалла X равна 8.

Недавно на завод поступил заказ – разработать стабильный нанокристалл заданной емкости c. При этом как число атомов в нанокристалле, так и число связей может быть произвольным. Помогите ученым разработать такой кристалл!

Формат входного файла

Входной файл содержит целое число c ($1 \le c \le 10\ 000$).

Формат выходного файла

На первой строке выходного файла выведите два целых числа n и m – количество атомов и связей в разработанном нанокристалле, соответственно. Будем считать, что атомы нанокристалла пронумерованы от 1 до n. Следующие m строк должны содержать по два целых числа – пары атомов, которые следует соединить торсионными связями. Если решений несколько, выведите любое.

Если искомого нанокристалла не существует, выведите на первой строке выходного файла два нуля.

Пример

Входной файл	Выходной файл
8	4 4 1 2 1 3 2 3 2 4
2	0 0

Решение

Это одна из двух задач на творческий поиск, предложенных на олимпиаде. Постараемся рассмотреть возможные подходы к ее решению.

Сначала переформулируем задачу в терминах теории графов. Разобравшись в определениях, несложно заметить в определении нанокристалла определение графа, где атом соответствует вершине, торсионная связь – ребру, а условие стабильности – связности графа. Емкость нанокристалла есть сумма кратчайших расстояний между всеми парами вершин графа. Будем для краткости, следуя легенде задачи, называть эту сумму *емкостью* графа, хотя этот термин и не является общепринятым.

Сперва отметим, что мы уже сталкивались с кратчайшими расстояниями в графе в задачах A и C. Однако, если там требовалось в заданном графе найти кратчайшее расстояние между парой вершин, то здесь нам наоборот требуется по заданной сумме кратчайших расстояний между всеми парами вершин построить соответствующий граф.

Сумма кратчайших расстояний между всеми парами вершин – достаточно странная характеристика графа. Она накладывает весьма слабые ограничения на его структуру. Поэтому возникает естественное желание искать решения в достаточно узком классе графов, где легко оценить искомую сумму.

Наиболее «простым» в некотором смысле графом является полный граф. В нем кратчайшее расстояние между любой парой вершин равно единице, следовательно у полного графа с n вершинами емкость равна $\frac{n(n-1)}{2}$.

Заметим, что, если удалить из полного графа одно ребро, длина пути между вершинами, которые оно соединяло, увеличится на один и станет равна двум. Остальные пути сохранятся, таким образом емкость графа увеличится на единицу. Посмотрим, можно ли увеличить емкость графа еще на единицу. Можно! Для этого нужно удалить любое другое ребро. Правда при дальнейшем удалении ребер из графа его емкость может меняться и на большие значения, если длины кратчайших путей начинают превышать два. Однако этого можно избежать, если, например, сохранять ребра, инцидентные первой вершине. Таким образом мы можем удалить до $\left(\frac{n(n-1)}{2}-(n-1)\right)$ ребер из графа, увеличив его емкость на эту величину.

Итак, мы научились получать графы с емкостью от $l(n) = \frac{n(n-1)}{2}$ до $r(n) = (n-1)^2$ для всех n. Посмотрим, может быть, любая возможная емкость лежит в одном из таких интервалов, и задача решена. К сожалению, нет. Для $n \geq 4$ выполнено $l(n+1) \leq r(n)+1$, поэтому все емкости, начиная с 6, разобраны. Но емкости от 1 до 5 требуют дополнительного исследования.

Емкости 1, 3 и 4 покрываются общим случаем для $n = 2$ и $n = 3$ соответственно. Однако для емкостей 2 и 5 придется искать альтернативное решение. Впрочем, заметим, что связные графы с двумя и тремя вершинами имеют емкости 1, 3 и 4. Эти графы приведены на рисунке 3.

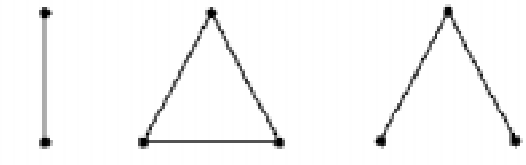


Рисунок 3.

А значит, граф-решение должен иметь как минимум четыре вершины. Но емкость графа не может быть меньше количества пар вершин в нем, которое для графа с четырьмя вершинами равно 6. Следовательно, для емкостей 2 и 5 решений не существует. Итак, задача решена.

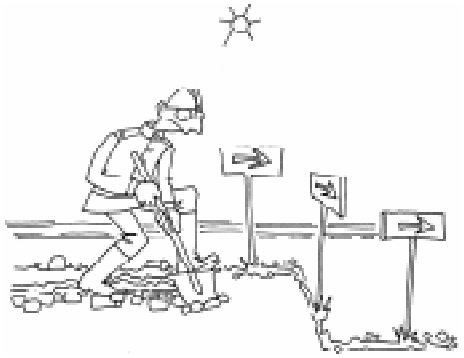
Отметим, что описанный способ решения задачи не является единственным. Например, не обязательно начинать рассуждение с полного графа. Решение можно получить, начиная, к примеру, с графа-звезды (одна вершина, с которой соединены все остальные), поочередно добавляя к нему ребра (этот процесс в некотором смысле обратен рассмотренному выше). Возможны и другие варианты.

Программа

Приведем текст программы целиком. Программа сначала находит n, для которого заданная емкость попадает в диапазон $\{l(n), ..., r(n)\}$, и затем выводит искомый граф.

```
program nano;
var
  n, s, k, i, j: integer;
begin
  reset(input, 'nano.in');
  rewrite(output, 'nano.out');
  read(s);
  if (s = 2) or (s = 5) then begin
    writeln('0 0');
    exit;
end;
n := 1;
while (n * (n - 1) div 2) <= s do inc(n);
dec(n);
k := s - (n * (n - 1) div 2);
writeln(n, ' ', n * (n - 1) div 2 - k);
for i := 1 to n do
  for j := i + 1 to n do begin
    if (i = 1) or (k = 0) then
      writeln(i, ' ', j);
    else
      dec(k);
  end;
end.
```

ЗАДАЧА I. ВОССТАНОВЛЕНИЕ ПРОГРАММЫ



Автор: *Павел Маврин, СПбГУ ИТМО.*

Выполняя очередное задание Министерства образования и сельского хозяйства, программисты Флатландского государственного университета информационных технологий, удобрений и ядохимикатов создали специального робота по прокладке оросительных каналов.

Программа для робота представляет собой последовательность команд. Команды робота приведены в таблице 1.

Любая программа начинается и заканчивается командами перемещения и не содержит двух команд перемещения или двух команд поворота подряд.

Исходно робот размещается в некоторой точке поля, на котором требуется создать оросительную систему. После запуска робот последовательно выполняет команды своей программы. Программа считается *корректной*, если полученный в результате ее выполнения канал не имеет самопересечений или самокасаний.

Программисты университета написали программу и собрались отправить ее по электронной почте в министерство. Однако в результате поражения сети университета вирусом программа оказалась испорчена. А именно, из нее исчезли все команды перемещения. Теперь программистам требуется восстановить программу. Поскольку времени очень мало, решено было оставить сохранившиеся команды поворота, вставив между ними команды перемещения так, чтобы получившаяся программа была корректной.

Формат входного файла

Входной файл содержит команды поворота исходной программы в том порядке, в котором они в ней следовали. Каждая команда представляет собой символ «L» либо «R», команды друг от друга не отделяются. Количество команд не превышает 30 000.

Формат выходного файла

Выведите в выходной файл любую корректную программу, последовательность команд поворота в которой совпадает с последовательностью, заданной во входном файле. Параметр X каждой команды перемещения должен быть положительным целым числом и не должен превышать 10^9. Все команды выведите в одной строке и друг от друга не отделяйте.

Пример (рисунок 4)

Входной файл	Выходной файл
LLLRRR	(4)L(3)L(2)L(2)R(2)R(1)R(1)

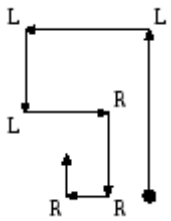


Рисунок 4.

Таблица 1.

Команда	**Обозначение**	**Действия робота**
Перемещение	(*X*)	Переместиться вперед на *X* километров, прокладывая за собой оросительный канал
Поворот налево	L	Повернуть налево на 90°
Поворот направо	R	Повернуть направо на 90°

Решение и программа

Еще одна задача на творческий поиск допускает множество решений. Мы опишем два из них: первое получается при попытке оптимизировать непосредственное решение, которое сразу приходит в голову, а второе носит эвристический характер.

Рассмотрим сначала непосредственный подход. Будем каждый раз делать в текущем направлении шаг на один километр. Если это приводит к тому, что робот оказывается в точке, в которой он уже был, то скорректируем предыдущие шаги робота. А именно, увеличим на единицу длину всех отрезков, параллельных последнему шагу робота и пересекающих перпендикулярную прямую, проходящую через его текущее положение. Тогда робот переместится на один километр назад, и его траектория перестанет быть самопересекающейся.

Результат применения этого алгоритма к программе робота в примере приведен на рисунке 5.

Посмотрим на возможную реализацию алгоритма. Если напрямую проверять для текущего положения робота, не лежит ли оно на его траектории, а также каждый раз искать отрезки траектории, которые следует увеличить, и производить это увеличение, то время работы алгоритма будет составлять $O(n^2)$, где n – длина программы. Такая оценка сложности не позволяет программе уложиться в отведенный предел времени.

Посмотрим, как можно оптимизировать это решение. Заметим сначала, что проводить увеличение на единицу длины всех отрезков, параллельных данному и пересекающих прямую, проходящую через конечное положение робота, перпендикулярную последнему отрезку траектории, можно всегда, а не только в тех случаях, когда робот оказался в точке, где он уже был. Это избавляет нас от необходимости проверять, попадает ли робот на свою траекторию. Применение модифицированного таким образом алгоритма к программе из примера приводит к последовательности траекторий, изображенной н рисунке 6.

Заметим, что полученная траектория обладает следующим свойством: все горизонтальные отрезки заканчиваются в точках с различными абсциссами, причем отличными от абсциссы начальной точки, а все вертикальные отрезки – в точках с различными ординатами, причем отличными от ординаты начальной точки. Причем когда проводится горизонтальный отрезок, происходит следующая операция: точки «раздвигаются» и между соседними абсциссами вставляется новое значение. Аналогичное действие с ординатами происходит при добавлении нового вертикального отрезка.

Это позволяет реализовать рассматриваемый алгоритм таким образом, что время его работы окажется $O(n)$. Организуем потенциальные абсциссы и ординаты точек поворота траектории в виде связных списков. При этом конкретные значения координатам присваивать не будем, а порядок элементов в списке будет соответствовать порядку их следования на соответствующей оси.

Для каждой вершины ломаной (точки поворота траектории) будем хранить ссылку на элемент списка абсцисс, соответствующий ее абсциссе, а также на элемент списка ординат, соответствующий ее ординате, Каждый раз, когда нам надо добавить очередную вершину траектории, будем создавать для нее новый элемент в списке абсцисс, если в нее ведет горизонтальный отрезок, или в списке ординат, если в нее ведет вертикальный отрезок. Вновь созданный элемент будет добавляться в соответствующий список после или перед эле-

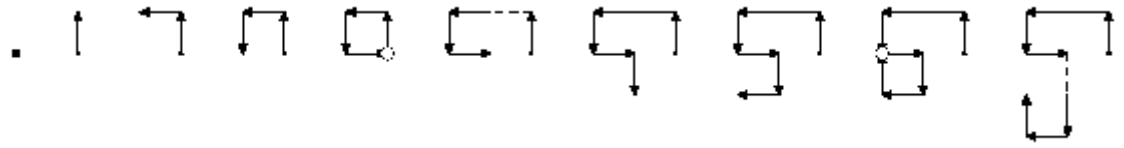


Рисунок 5.

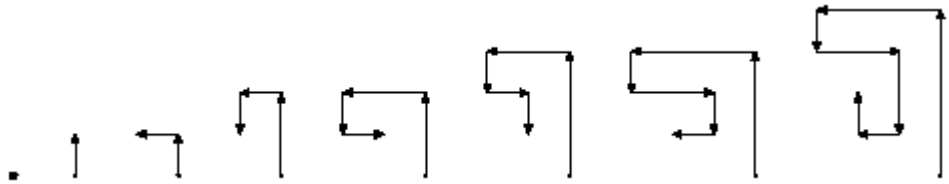


Рисунок 6.

ментом, соответствующим предыдущей вершине, в зависимости от направления отрезка (вправо или влево, вверх или вниз).

После окончания построения траектории пробежим по каждому из списков и присвоим координатам конкретные значения (каждая следующая получит значение на единицу больше предыдущего). Теперь осталось пройти по траектории и, пользуясь ссылками на значения в списках, вывести длину каждого звена.

Рассмотрим программную реализацию алгоритма.

Обратим сначала внимание на списки. Два двусвязных списка будут организованы в паре массивов `next` и `prev`. Голова списка абсцисс будет соответствовать первым элементам массивов, а голова списка ординат – вторым. Массив `val` будет использоваться после построения списков, `val[i]` будет соответствовать координате, присвоенной элементу списка, хранящемуся в *i*-ой ячейке.

Следующие процедуры реализуют работу со списками. Фиктивный нулевой элемент позволяет не рассматривать отдельно операции с головой и хвостом списка. Значения `next[0]` и `prev[0]` могут быть произвольными, они никак не используются. Переменная `l` содержит количество уже созданных элементов списков.

```
var
  l: integer;
  val, next, prev: array[0..maxlength]
                              of integer;
procedure addafter(x: integer);
begin
  inc(l);
  next[l] := next[x];
  prev[next[x]] := l;
  prev[l] := x;
  next[x] := l;
end;
procedure addbefore(x: integer);
begin
  inc(l);
  prev[l] := prev[x];
  next[prev[x]] := l;
  next[l] := x;
  prev[x] := l;
end;
```

Прочитав входную программу, проводим инициализацию. Переменные `x` и `y` содержат указатели на элементы списков абсцисс и ординат, соответственно, отвечающих текущей вершине. Массивы `xx` и `yy` содержат ссылки на элементы списков для вершин ломаной. Концы *i*-го отрезка ломаной соответствуют вершинам $i - 1$ и i.

Переменная `d` содержит текущее направление: 0 – вправо, 1 – вниз, 2 – влево, 3 – вверх.

```
readln(s);
x := 1;
y := 2;
xx[0] := x;
yy[0] := y;
d := 3;
l := 2;
```

Теперь последовательно проходим траекторию.

```
for i := 1 to length(s) + 1 do begin
  case d of
    0: begin
      addafter(x);
      xx[i] := l;
      yy[i] := y;
    end;
    1: begin
      addbefore(y);
      xx[i] := x;
      yy[i] := l;
    end;
    2: begin
      addbefore(x);
      xx[i] := l;
```

```
      yy[i] := y;
    end;
    3: begin
      addafter(y);
      xx[i] := x;
      yy[i] := l;
    end;
  end;
  x := xx[i];
  y := yy[i];
  if i <= length(s) then begin
    if s[i] = 'L' then
      d := (d + 3) mod 4
    else
      d := (d + 1) mod 4;
  end;
end;
```

Списки построены, теперь присваиваем конкретные значения координатам. Заметим, что исходные головы списков в настоящий момент не обязаны быть первыми элементами!

```
for i := 1 to 2 do begin
  j := i;
  while prev[j] <> 0 do j := prev[j];
  val[j] := 0;
  while next[j] <> 0 do begin
    val[next[j]] := val[j] + 1;
    j := next[j];
  end;
end;
```

Теперь можно вывести ответ.

```
for i := 1 to length(s) + 1 do begin
  j := abs(val[xx[i]] - val[xx[i - 1]]) +
       abs(val[yy[i]] - val[yy[i - 1]]);
  write('(', j, ')');
  if i <= length(s) then write(s[i]);
end;
```

Однако, оказывается, приведенное решение задачи не является единственным, у задачи есть другое, более простое решение. Расскажем теперь о нем.

Постараемся строить траекторию таким образом, чтобы постоянно поддерживалось следующее свойство: после каждого перемещения робота и справа, и слева от текущей его позиции не было точек, в которых он уже был. Для этого попытаемся обеспечить выполнение следующего дополнительного условия: робот всегда находится в вершине минимального прямоугольника со сторонами, параллельными осям координат, содержащего всю траекторию. При этом одна из сторон прямоугольника, смежных с этой вершиной, содержит последний проведенный отрезок траектории, а другая не содержит точек траектории.

Посмотрим, как сохранить это условие. Если после поворота робот должен выйти из прямоугольника, сделаем шаг на один километр. Условие сохранилось. Если же робот должен проследовать вдоль стороны прямоугольника, пройдем до противоположного угла и выйдем наружу на один километр. Условие снова сохранилось.

Посмотрим на траекторию, которая получается для робота при использовании этого алгоритма для решения тестового примера (рисунок 7).

Отметим интересную особенность этого алгоритма. Если предыдущий рассмотренный алгоритм «углублялся» внутрь заполненной территории, постоянно «раздвигая» мешающие ему отрезки, то этот, наоборот, постоянно держит конечную вершину «на свободе», разворачивая траекторию наружу. Примечательно также, что алгоритм является «потоковым» – он обрабатывает программу последовательно и может выводить ответ прямо в процессе обработки – уже проведенные отрезки при проведении нового не меняются.

Приведем программную реализацию и этого алгоритма. Переменные `minx`, `maxx`, `miny` и `maxy` задают текущий прямоуголь-

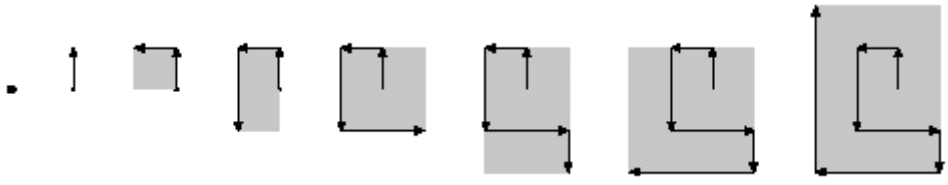


Рисунок 7.

ник, содержащий траекторию. Переменные `x` и `y` – текущие координаты робота, а `d` – направление последнего отрезка (как и прежде, значения соответствуют направлениям следующим образом: 0 – вправо, 1 – вниз, 2 – влево, 3 – вверх).

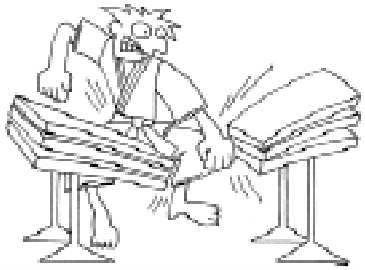
```
minx := 0;
maxx := 0;
miny := 0;
maxy := 1;
d := 3;
write('(1)');
x := 0;
y := 1;
```

Теперь пробегаем по заданному шаблону программы и применяем алгоритм. Новое направление движения задает увеличение одной из сторон охватывающего прямоугольника на единицу в соответствующем направлении, после чего мы перемещаем робота в соответствующий угол прямоугольника.

```
for i := 1 to length(s) do begin
  if s[i] = 'L' then
    d := (d + 3) mod 4
  else
    d := (d + 1) mod 4;
  case d of
    0: begin
      inc(maxx);
      l := maxx - x;
      x := maxx;
    end;
    1: begin
      dec(miny);
      l := y - miny;
      y := miny;
    end;
    2: begin
      dec(minx);
      l := x - minx;
      x := minx;
    end;
    3: begin
      inc(maxy);
      l := maxy - y;
      y := maxy;
    end;
  end;
  write(s[i], '(', l, ')');
end
```

Напоследок отметим еще один интересный (для жюри) момент, касающийся этой задачи – отдельную трудность составляет написание проверяющей программы для нее. В качестве интересного упражнения можно предложить всем желающим написать программу, проверяющую корректность ответа для этой задачи, с временем работы $O(n \log n)$.

ЗАДАЧА J. ЛЕСОПИЛКА



Автор: *Игорь Синев, СПбГУ ИТМО.*

Недавно на лесопилку, где работает Вася, поступил новый заказ. Для постройки нового дома мэру соседнего города требуется a досок длины x футов и b досок длины y футов.

Поскольку на лесопилке имеется только неограниченный запас досок длины z футов, Васе поручили исполнить заказ клиента, распилив имеющиеся доски на меньшие. Вася хочет закончить работу как можно быстрее, поэтому он хочет выполнить заказ, сделав как можно меньше распилов. При этом количество использованных досок длины z роли не играет, кроме того, часть досок, образовавшихся в результате распила, может не требоваться для заказа и остаться на лесопилке.

Например, если на лесопилке имеются доски длины 80, а клиенту требуется две доски длины 30 и семь досок длины 20, то достаточно сделать семь распилов: одну доску распилить двумя распилами на доски длины 20, 30 и 30, одну тремя распилами на четыре доски длины 20 и одну двумя распилами на доски длины 20, 20 и 40. Доска длины 40 клиенту не нужна, она останется на лесопилке, остальные доски будут отправлены клиенту.

Формат входного файла

Входной файл содержит целые числа a, x, b, y и z. Все числа положительны и не превышают 300, $x \le z$, $y \le z$, $x \ne y$.

Формат выходного файла

Выведите в выходной файл минимальное количество распилов, которые требуется сделать для того, чтобы выполнить заказ.

Примеры

Входной файл	Выходной файл
2 30 7 20 80	7

Решение и программа

Для решения этой задачи следует применить динамическое программирование. Сначала мы изложим простой алгоритм, который позволяет решить задачу за $O(abz)$, а затем укажем способ ускорить алгоритм, что позволит довести время работы до $O(ab)$. Заметим, что и первый вариант алгоритма достаточно эффективен при приведенных в условии задачи ограничениях, однако второй алгоритм оказывается на удивление элегантным.

Прежде чем решать задачу, изменим немного целевую функцию. Пусть для изготовления нужных клиенту досок истрачено несколько досок длины z, причем k из них распилены полностью (без остатка). Тогда, очевидно, всего проведено $a + b - k$ распилов. Таким образом, для минимизации числа распилов достаточно максимизировать число досок, которые будут полностью распилены.

Итак, мы получили следующую задачу: требуется распилить максимальное число досок длины z без остатка на доски длины x и y, чтобы досок первого типа было не больше a и досок второго – не больше b.

Если мы фиксируем число досок каждого типа в решении, то из этого однозначно будет следовать количество распиленных досок длины z – если получилось i досок длины x и j досок длины y, то распилено $(ix + jy)/z$ досок длины z. Поэтому достаточно для каждой пары $(i; j)$ где $0 \le i \le a$, $0 \le j \le b$ выяснить, можно ли напилить i досок длины x и j досок длины y.

Используем динамическое программирование. Обозначим как $f_{i,j}$ возможность решить задачу для пары $(i; j)$. Мы будем для краткости обозначать истиное значение логической переменной как 1, а ложное как 0.

Пусть мы решили задачу для всех пар, где одна из компонент меньше, чем в рассматриваемой задаче. Укажем, как найти значение $f_{i,j}$. Рассмотрим способ, которым была распилена последняя доска. Для этого надо найти все неотрицательные целочисленные решения уравнения $px + qy = z$. Это можно сделать, просто перебрав все значения p от 1 до z. Получаем:

$$f_{i,j} = \bigvee_{\substack{px+qy=z \\ 0 \le p \le i \\ 0 \le q \le j}} f_{i-p,\,j-q}$$

Здесь «$\vee$» означает логическое *или* соответствующих значений. В качестве инициализации используем $f_{i,j} = 1$.

Остается среди всех пар $(i; j)$, для которых $f_{i,j} = 1$, найти ту, для которой значение $(ix + jy)/z$ максимально, и вывести в качестве ответа $a + b - (ix + jy)/z$.

Реализация описанного алгоритма не представляет труда, оставим ее в качестве упражнения. Мы же опишем способ ускорить приведенный метод.

Заметим, что вообще говоря алгоритм работает не очень эффективно из-за того, что «заглядывает» слишком далеко назад, чтобы обновить информацию в динамическом массиве. Проверка всех возможных способов распилить целую доску на маленькие занимает слишком много времени.

Если бы нам удалось проводить обновление за $O(1)$, то время работы алгоритма автоматически сократилось бы до $O(ab)$. Это наводит на мысль, что удобно добавлять маленькие доски по одной, а не целыми наборами, получающимися при распиливании большой доски. А в динамическом массиве следует хранить информацию не только о «завершившихся» распилах, когда распилено целое число досок длины z, а также и о промежуточных состояниях, когда распилено некоторое целое число больших досок, а также часть еще одной доски.

Итак, введем вместо $f_{i,j}$ значение $g_{i,j}$, которое означает возможность получить i

досок длины x и j досок длины y, распилив некоторое число досок длины z целиком и не более одной доски – частично. Пусть $l = (ix + jy) \bmod z$ – длина доски, распиленной не целиком. Тогда $g_{i,j} = 1$, если выполнено хотя бы одно из условий:

$l \geq x$ и $g_{i-1,j} = 1$;

$l \geq y$ и $g_{i-1,j} = 1$;

$l = 0$ и/или $g_{i-1,j} = 1$, или $g_{i,j-1} = 1$.

Ответом же будет максимум по всем парам $(i; j)$ где $g_{i,j} = 1$ значения $\lfloor (ix + jy)/z \rfloor$, которое как раз равно числу полностью распиленных досок.

Заметим, что при реализации удобно вместо описанного динамического программирования «назад» использовать динамическое программирование «вперед», а именно, если значение $g_{i,j} = 1$, то попытаемся «отпилить» от последней, частично распиленной доски еще одну доску каждого типа. Пусть по прежнему $l = (ix + jy) \bmod z$, если $l + x \leq z$, то устанавливаем $g_{i+1,j} = 1$, аналогично, если $l + y \leq z$, то устанавливаем $g_{i,j+1} = 1$.

Приведем текст программы целиком.

```
program sawmill;
var
  max, a, x, b, y, z, i, j, q, l: integer;
  d: array[0..1000, 0..1000] of boolean;
begin
  reset(input, 'sawmill.in');
  rewrite(output, 'sawmill.out');
  read(a, x, b, y, z);
  d[0, 0] := true;
  max := 0;
  for i := 0 to a do
    for j := 0 to b do
      if d[i, j] then begin
        l := i * x + j * y;
        q := l div z;
        if q > max then max := q;
        l := l mod z;
        if l + x <= z then d[i + 1, j] := true;
        if l + y <= z then d[i, j + 1] := true;
      end;
  writeln(a + b - max);
end.
```

Замечание: для всех задач ограничение по времени – 2 секунды, ограничение по памяти – 64 мегабайта.

Литература

1. *Андреева Е.В., Егоров Ю.Е.* Вычислительная геометрия на плоскости // Информатика, 2002. С. 39.

2. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999.

Станкевич Андрей Сергеевич, преподаватель кафедры Компьютерных технологий СПбГУ ИТМО, лауреат премии Президента Российской Федерации.