1 The problem

This is a problem 66 from Project Euler, and it goes like this:

Consider quadratic Diophantine equations of the form:

$$x^2 - Dy^2 = 1$$

For example, when $D = 13$, the minimal solution in x is $649^2 - 13 * 180^2 = 1$.

It can be assumed that there are no solutions in positive integers when D is square.

By finding minimal solutions in x for $D = \{2, 3, 5, 6, 7\}$, we obtain the following:

$$3^2 - 2 * 2^2 = 1$$
$$2^2 - 3 * 1^2 = 1$$
$$9^2 - 5 * 4^2 = 1$$
$$5^2 - 6 * 2^2 = 1$$
$$8^2 - 7 * 3^2 = 1$$

Hence, by considering minimal solutions in x for $D \leqslant 7$, the largest x is obtained when $D = 5$.

Find the value of $D \leqslant 1000$ in minimal solutions of x for which the largest value of x is obtained.

2 Definitions

```
import Data.Maybe (isJust)
import Data.Ord (comparing)
import Data.List (maximumBy)
```

Let me begin with main definitions.

The equation for a certain D will be stored as an *Equation*:

```
type Equation = Integer
```

We can test if a certain pair of x and y is a solution for the *Equation*:

```
test :: Equation → Integer → Integer → Bool
test d x y = (x ↑ 2 − d ∗ y ↑ 2 ≡ 1)
```

Let's check an example solution (from now on, every $check_*$ function should return $True$: a simple unit test):

```
check_1 = test 13 649 180
```

I can't make up a quick test if a certain x is a minimal solution.

3 Research

This problem appears to be well-known. The equation $x^2 - ny^2 = 1$ is named ***Pell's equation***. A pair of $x = 1$ and $y = 1$ is a trivial solution; for any natural n there are x and $y > 0$ that satisfy the equation.

4 Iteration

Lagrange proved that:

$$\forall n \neq m^2 : \exists x > 0, y > 0 : x^2 - ny^2 = 1 \tag{1}$$

So we're interested in any *Equation* d there d is not a perfect square:

```
isqrt :: Integer → Integer
isqrt n = floor $ sqrt $ (fromIntegral n :: Double)


is_perfect_square :: Integer → Bool
is_perfect_square n = sq * sq ≡ n
  where sq = isqrt n


equations :: Integer → [Equation]
equations max_d = filter (¬ ∘ is_perfect_square) [1 .. max_d]
```

Since there does not seem to be a way in which *Equations* for different ds are related, we can just solve every *Equation* independently:

```
minimal_solutions :: (Equation → (Integer, Integer)) → Integer → [(Equation, (Integer, Integer))]
minimal_solutions solve max_d = zip eqs (map solve eqs)
  where eqs = equations max_d
```

We can even parallelize it.

5 Solving an equation

We have to implement a *solve* function now:

```
solve :: Equation → (Integer, Integer)
```

Let's start with a simple brute-force approach: iterate over $x \in N$ and try to calculate an y. If it succeeds, then we have a solution. Due to (1) this process will eventually end.

$$y^2 = \frac{x^2 - 1}{d}$$

```
divides :: Integer → Integer → Bool
divides a b = a `mod` b ≡ 0


get_y :: Equation → Integer → Maybe Integer
get_y d x = if dy2 `divides` d ∧ y * y ≡ y2
  then Just y
  else Nothing
  where dy2 = x * x − 1
    y2 = dy2 `div` d
    y = isqrt y2


solve_brute :: Equation → (Integer, Integer)
solve_brute d = head solutions
  where solutions = map (λ(x, Just y) → (x, y)) $ filter (isJust ∘ snd) variants
    variants = zip [2 . .] $ map (get_y d) [2 . .]
```

Let's try it out on the example data set:

```
minimal_solutions solve_brute 7
```

$[(2,(3,2)),(3,(2,1)),(5,(9,4)),(6,(5,2)),(7,(8,3))]$

With this naive approach calculating a minimal x for $d \leqslant 60$ works well, but $d = 61$ takes too long to solve. Let's further investigate the problem.

6 Chakravala method

The ***Chakravala method*** is a method to solve equations such as ours:

$$x^2 - dy^2 = 1$$

Let's start with observation that the following equation is true:

$$(x_1^2 - dy_1^2)(x_2^2 - dy_2^2) = (x_1x_2 + dy_1y_2)^2 - d(x_1y_2 + y_1x_2)^2 \tag{2}$$

That means that

$$P(x_1, y_1, k_1), P(x_2, y_2, k_2) \implies P(x_1x_2 + dy_1y_2, x_1y_2 + y_1x_2, k_1k_2)$$

Where $P(x, y, k)$ means $x^2 - ny^2 = k$. We can use this fact to generate new solutions from known ones.

```
data Triple = Triple { d :: Equation
  , x :: Integer
  , y :: Integer
  , k :: Integer }
  deriving (Show)
```

$$
\begin{aligned}
&\mathit{combine} :: \mathit{Triple} \to \mathit{Triple} \to \mathit{Triple} \\
&\mathit{combine}\ t1\ t2 = \mathit{Triple}\ (d\ t1) \\
&\quad ((x\ t1) * (x\ t2) + (d\ t1) * (y\ t1) * (y\ t2)) \\
&\quad ((x\ t1) * (y\ t2) + (y\ t1) * (x\ t2)) \\
&\quad ((k\ t1) * (k\ t2))
\end{aligned}
$$

We are going to combine triples with each other to get eventually a triple with $x, y \in N; k = 1$. Where to get initial triples? We can use a trivial one such as $(m, 1, m^2 - d)$:

$$
\begin{aligned}
&\mathit{trivial} :: \mathit{Equation} \to \mathit{Integer} \to \mathit{Triple} \\
&\mathit{trivial}\ d\ m = \mathit{Triple}\ d\ m\ 1\ (m \uparrow 2 - d)
\end{aligned}
$$

To test if a triple is genuine, we just place its values into the equation:

$$
\begin{aligned}
&\mathit{test\_triple} :: \mathit{Triple} \to \mathit{Bool} \\
&\mathit{test\_triple}\ t = (x\ t) \uparrow 2 - (d\ t) * (y\ t) \uparrow 2 \equiv (k\ t)
\end{aligned}
$$

We need to be sure if our *trivial* function works as expected:

$$
\begin{aligned}
&\mathit{check\_2} :: \mathit{Bool} \\
&\mathit{check\_2'}\ d = \mathit{and}\ \$\ \mathit{map}\ \mathit{test\_triple}\ \$\ \mathit{map}\ (\mathit{trivial}\ d)\ [1\,..\,10] \\
&\mathit{check\_2} = \mathit{and}\ \$\ \mathit{map}\ \mathit{check\_2'}\ [1\,..\,10]
\end{aligned}
$$

If we have a triple (x, y, k) such that $gcd(x, y) = 1$, and a trivial triple $(m, 1, m^2 - d)$, we can combine them and scale down the result by k:

$$P(x, y, k) \implies P(\frac{xm + dy}{k}, \frac{x + ym}{k}, \frac{m^2 - d}{k})$$

And, since first two elements of a triple are inside the square:

$$P(x, y, k) \implies P(\frac{xm + dy}{|k|}, \frac{x + ym}{|k|}, \frac{m^2 - d}{k})$$

If $\frac{x+ym}{k} \in Z$, then two other elements are integers, too. So the requirement on m is that $x + ym$ should divide k; also we need $\frac{m^2-d}{k}$ to be as low as possible (we want it to scale down 1 eventually); that is implied by minimization of $m^2 - d$.

$$x + ym \mid k \iff ym \equiv -x \pmod{k}$$

To get m, we need to implement a modular division algorithm. I stolen this implementation somewhere.

$$
\begin{aligned}
&\mathit{extended\_gcd} :: \mathit{Integer} \to \mathit{Integer} \to [\mathit{Integer}] \\
&\mathit{extended\_gcd}\ a\ b \mid \mathit{mod}\ a\ b \equiv 0 = [0, 1, b] \\
&\quad \mid \mathit{otherwise} = [y, x - y * (\mathit{div}\ a\ b), z] \\
&\qquad \textbf{where}\ [x, y, z] = \mathit{extended\_gcd}\ b\ (\mathit{mod}\ a\ b)
\end{aligned}
$$

```
inverse_in :: Integer → Integer → Integer
inverse_in m a = if ans < 0 then m + ans else ans
  where ans = head $ tail $ extended_gcd m a


solve_ax_b :: Integer → Integer → Integer → Integer
solve_ax_b m a b = if m < 0
  then solve_ax_b (−m) a b
  else if ¬ $ divides b d
    then error "omg"
    else let m1 = m `div` d
      a1 = a `div` d
      b1 = b `div` d
      c = inverse_in m1 a1 in (c ∗ b1) `mod` m1
  where d = abs $ gcd a m


select_m :: Triple → Integer
select_m t = if (k t) < 0
  then select_m t {k = −(k t)}
  else ((m_want − mod_diff) `min` (m_want + mod_diff2))
  where m_mod = solve_ax_b (k t) (y t) (−(x t))
    m_want = isqrt (d t)
    mod_diff = (m_want `mod` (abs $ k t)) − m_mod
    mod_diff2 = (abs $ k t) − mod_diff
```

Okay, now we can generate a trivial triple to combine with from any triple. We are going to use a tuple with $y = 1$:

```
initial_tuple :: Equation → Triple
initial_tuple d = Triple d x 1 k
  where x = 1 + isqrt d
    k = x ↑ 2 − d
```

A step is generating an m and combining it with a tuple, effectively scaling it down:

```
scaledown :: Integer → Triple → Triple
scaledown q t = if divides (x t) q ∧ divides (y t) q
  then Triple (d t) (abs $ (x t) `div` q) (abs $ (y t) `div` q) (k t `div` (q ∗ q))
  else error $ "does not divide?! " ++ show q ++ "   " ++ show t


step :: Triple → Triple
step t = if divides (x com) (k t)
  then scaledown (k t) com
  else error $ "Look " ++ show m ++ "  " ++ show triv ++ "  " ++ show t ++ "  " ++ show com
  where m = select_m t
```

```
triv = trivial (d t) m
com = combine t triv
```

We need to do such steps until $k \notin \{\pm 1\}$.

```
make_steps :: Triple → Triple
make_steps t = if (k t) ≡ 1
  then t
  else make_steps $ step t
```

Now we can combine the triple with itself, and scale it down to get k = 1.

```
get_minimal_solution :: Equation → Triple
get_minimal_solution d = make_steps $ initial_tuple d


minimal_pair :: Integer → (Integer, Integer)
minimal_pair d = (x sol, y sol)
  where sol = get_minimal_solution d
```

7 Solving a problem

```
minimal_solutions minimal_pair 1000
```

Works pretty fast, so all we need to do is to get maximum among x's:

```
main :: IO ()
main = print $ fst $ maximumBy (comparing (fst ∘ snd)) $ minimal_solutions minimal_pair 1000
```

Problem solved, answer is 661.