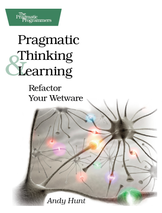
Problems problems problems: Solving Project Euler with Scala

Tim Day

Back in 2009...

- Influenced by the book "Refactor Your Wetware", I was thinking it was finally time to learn another programming language besides the C++ and Python I knew.
- Techniques from the functional programming world were being advocated as a solution to the "software development crisis" and parallel programming difficulties generally. I had to try this for myself.
- Scala was enjoying a period of massive hype.
- I had been aware of Project Euler for some time, but holding it in reserve for learning a new programming language.



Scala http://www.scala-lang.org

- A hybrid of functional and object-oriented paradigms. Pragmatic rather than pure.
- Intended to be a Scalable Language, which grows with the demands of its users.
- Notable adopters: Twitter (switched from Ruby), Foursquare (use Lift, Scala's equivalent of Rails), guardian.co.uk.
- Combines the convenience of a dynamic language with the assurance of compiler type checking by use of type inference. It's rarely *necessary* to specify explicit types anywhere other than function declarations. This also makes Scala very convenient to use interactively.
- The Scala standard library is enormous, and deeply intertwined with the language.
- Things which look like language features may not be what they first appear. This:

```
1 to 10 by 2   // Returns scala.collection.immutable.Range = Range(1,3,5,7,9)
```

is actually method invocations:

```
1.to(10)       // Returns scala.collection.immutable.Range.Inclusive = Range(1,2,3,4,5,6,7,8,9,10)
1.to(10).by(2) // Returns scala.collection.immutable.Range = Range(1,3,5,7,9)
```

- No primitive types (c.f Java's `int` / `Int` dichotomy). Everything inherits from `Any`.
- Strong emphasis on immutability: immutable container types, "use `val`s not `var`s".
- Rich variety of container types: lists, arrays/vectors, maps, sets, multimaps (trees and hashed), queues, lazy streams...
- Incredible variety of methods per class. For example, `List` has:

```
 ++ ++:  +:  -  -- /:  /:\:+ ::  :::  :\  addString aggregate andThen apply asInstanceOf break canEqual collect collectFirst combinations companion compose contains containsSlice copyToArray
copyToBuffer corresponds count diff distinct drop dropRight dropWhile elements endsWith equalsWith exists filter filterNot find findIndexOf findLastIndexOf first firstOption flatMap
flatten fold foldLeft foldRight forall foreach genericBuilder groupBy grouped hasDefiniteSize head headOption indexOf indexOfSlice indexWhere indices init inits intersect isDefinedAt
isEmpty isInstanceOf isTraversableAgain iterator last lastIndexOf lastIndexOfSlice lastIndexWhere lastOption length lengthCompare lift map mapConserve max maxBy min minBy mkString nonEmpty
orElse padTo par partition patch permutations prefixLength product productArity productElement productElements productIterator productPrefix projection reduce reduceLeft reduceLeftOption
reduceOption reduceRight reduceRightOption remove removeDuplicates repr reverse reverseIterator reverseMap reverse_::: reversedElements sameElements scan scanLeft scanRight segmentLength
seq size slice sliding sort sortBy sortWith sorted span splitAt startsWith stringPrefix sum tail tails take takeRight takeWhile toArray toBuffer toIndexedSeq toIterable toIterator toList
toMap toSeq toSet toStream toString toTraversable transpose union unzip unzip3 updated view withFilter zip zipAll zipWithIndex
```

- Inheritance and mix-in traits used extensively. For example, `HashSet` inherits traits from around 32 supertypes.
- Extensive support for and use of generics, designed into the language from the start.
- Multi-paradigm support for parallelism: parallel collections, Futures and Erlang-inspired Actors.
- Standard Java classes wrapped into much more usable forms by implicits (`BigInt` particularly useful for Project Euler).
- Many nice touches e.g `(1 until N)` is equivalent to `(1 to N-1)`; linebreaks obviate semicolons; interchangeable parentheses; largely optional "." and "( )" on method invocations; `object` (c.f `class`) keyword to declare singletons; flexible `for`-loop syntax, constructor syntax,...
- Excellent comprehensive Scaladoc online documentation.

Arcane/quirky language features

Example: method names ending in ":" are right associative, enabling "traditional" notation for list "cons":

```
0 :: List(1,2,3)          // Returns List(0,1,2,3)
```

instead of something like

```
List(1,2,3).prepend(0)
```

Warning: There's a *lot* of this sort of fine-detail; e.g `unary` methods, so-called "case classes", a class named "`::`", Scala's peculiar equivalent of the switch statement (`match` and extractors), remarkable XML support...

Operator overloading, implicit conversions

Java and C# dropped their C++ predecessor's operator overloading and implicit conversion features, regarding them as a failed experiment. In Scala these ideas return, taken to extremes.

- Example: Adding a `containsDuplicates` method to Scala's existing `List` class without subclassing

```
implicit def enhanceWithContainsDuplicates[T](s:List[T]) = new {
  def containsDuplicates = (s.distinct.size != s.size)  // .distinct returns a new List without any
    duplicates
}
assert(List(1,2,3,2).containsDuplicates)
assert(!List("a","b","c").containsDuplicates)
```

- Few restrictions, so well suited to DSL creation. For example "<-->" or "#!?*" would be valid method/function names.

Performance

- Small changes can have surprisingly large effects.
  Equivalent codes below were timed (steady state JVM) for $n=2^{25}$ on a quad-core i7 (with HT):

```
val factors=List(101,103,107,109,113)
def hit(i:Int) = factors.exists(i%_==0)       // Moderately expensive test function

(1 to n).count(hit)                           // 1.3s - baseline serial case
var c=0;for (i<-1 to n) if (hit(i)) c=c+1;c   // 1.1s - imperative style, but then why use Scala?
(1 to n).par.count(hit)                       // 0.37s - x 3.5 performance improvement
(0 until n).map(_+1).count(hit)               // 4.7s - massive new object creation in map kills performance!
(0 until n).par.map(_+1).count(hit)           // 7.3s - ...and it breaks parallelism too!
(0 until n).iterator.map(_+1).count(hit)      // 1.6s - but an iterator tames object "churn"

// Iterator doesn't support .par, but a hybrid approach is possible:
val B=(1<<20)  // Batch size
(0 until n by B).par.map(
  i=>(i until i+B).iterator.map(_+1).count(hit)
).sum                                         // 0.44s - .par benefit returns
```

- By comparison, equivalent C++ code runs in 0.78s, or 0.19s with OpenMP parallelism ($> \times 4$ scaling).
- For a few problems C++ solutions were also implemented; the general observation being that C++ is around $\times 3$ faster than Scala, *but the Scala solution will only require* 1/3 *the number of lines of code.*

"wordle" of Scala solutions to Project Euler problems 1-207

Scala books

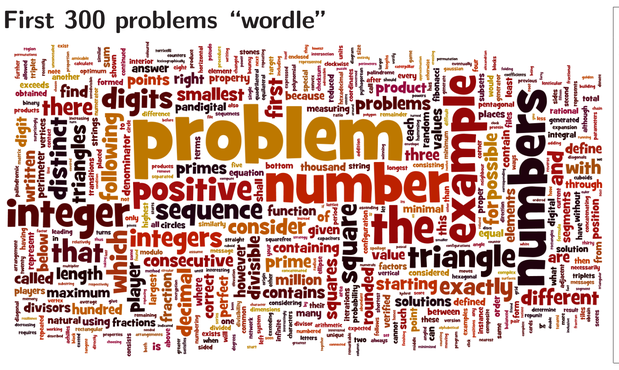
This poster

Created on a Debian GNU/Linux system with LaTeX: TeXLive release, beamer and beamerposter packages, EulerVM fonts.

Project Euler http://projecteuler.net

- A series of problems in "computational mathematics".
- Originated in 2001; currently $>$ 360 problems. New problems added every few weeks.
- All problems soluble in less than a minute runtime on a modest computer *with an efficient algorithm*.
- Submitting a correct solution (typically a large integer) grants access to the problem's forum.
- Pedagogical sequencing: earlier problems introduce techniques needed to solve later ones.
- $>$ 200000 registered users worldwide (though average solutions-per-user is only 16.2).
- 790 users have solved at least 200 problems, 24 have solved at least 350.
- Leaderboards by country and programming language. Also a "Eulerian" category for the fastest solvers.

First 300 problems "wordle"



Named after Leonhard Euler

Swiss mathematician and physicist, 1707-1783.
One of the greatest mathematicians of all time.

- Analysis: Euler's identity: $e^{i\pi} + 1 = 0$.
- Number theory: In 1772, proved $2^{31} - 1$ is prime; largest known prime until 1867.
- Graph theory: Seven Bridges of Königsberg, $V - E + F = 2$.
- Applied mathematics: Analytic solutions to real world problems; numerical approximation of integrals.
- Physics: Fluid dynamics, optics and astronomy.
- Notation: First to write e, Σ, i, functions as $f(x)$, popularised π.

Selected problems

Problem 31

In England the currency is made up of pound, £, and pence, p, and there are eight coins in general circulation: 1p, 2p, 5p, 10p, 20p, 50p, £1 (100p) and £2 (200p).

It is possible to make £2 in the following way:
1×£1 + 1×50p + 2×20p + 1×5p + 1×2p + 3×1p

How many different ways can £2 be made using any number of coins?

Actually my first encounter with Project Euler, although I didn't know it at the time, when a colleague posted this on a company newsgroup years ago. Easy.

Problem 152

There are several ways to write the number $\frac{1}{2}$ as a sum of inverse squares using distinct integers. For instance, the numbers 2, 3, 4, 5, 7, 12, 15, 20, 28, 35 can be used:
$\frac{1}{2} = \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \frac{1}{7^2} + \frac{1}{12^2} + \frac{1}{15^2} + \frac{1}{20^2} + \frac{1}{28^2} + \frac{1}{35^2}$

In fact, only using integers between 2 and 45 inclusive, there are exactly three ways to do it, the remaining two being:
2, 3, 4, 6, 7, 9, 10, 20, 28, 35, 36, 45 and
2, 3, 4, 6, 7, 9, 12, 15, 28, 30, 35, 36, 45.

How many ways are there to write the number $\frac{1}{2}$ as a sum of inverse squares using distinct integers between 2 and 80 inclusive?

The most difficult yet, when I reached it. Was relieved to find I wasn't the only one to think so in the forum once I'd solved it and gained access. At first sight, an intractable search tree; the trick is in the pruning.

Problem 160

For any N, let f(N) be the last five digits before the trailing zeroes in N!. For example,

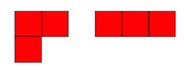
9! = 362880 so f(9) = 36288
10! = 3628800 so f(10) = 36288
20! = 2432902008176640000 so f(20) = 17664

Find f(1,000,000,000,000)

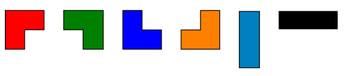
Was stuck on this for a long time; modular arithmetic... my nemesis!

Problem 161

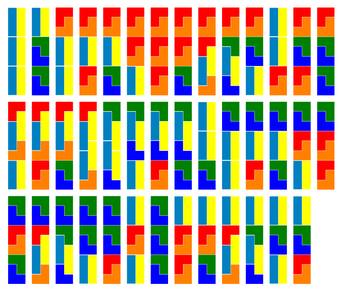
A triomino is a shape consisting of three squares joined via the edges. There are two basic forms:



If all possible orientations are taken into account there are six:



Any n by m grid for which $n \times m$ is divisible by 3 can be tiled with triominoes. If we consider tilings that can be obtained by reflection or rotation from another tiling as different there are 41 ways a 2 by 9 grid can be tiled with triominoes:



In how many ways can a 9 by 12 grid be tiled in this way by triominoes?

Classic Dynamic Programming problem. These are usually fairly easy and always have a high "solved by" number.

Problem 195

Let's call an integer sided triangle with exactly one angle of 60 degrees a 60-degree triangle. Let r be the radius of the inscribed circle of such a 60-degree triangle.

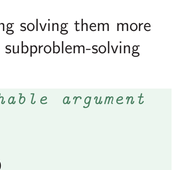
There are 1234 60-degree triangles for which $r \leqslant 100$. Let T(n) be the number of 60-degree triangles for which $r \leqslant n$, so T(100) = 1234, T(1000) = 22767, and T(10000) = 359912.

Find T(1053779).

Found this very hard. Took me a long time to even reproduce the given results, let alone devise a solution which would scale.

Problem 201

Three mirrors are arranged in the shape of an equilateral triangle, with their reflective surfaces pointing inwards. There is an infinitesimal gap at each vertex of the triangle through which a laser beam may pass.



Label the vertices A, B and C. There are 2 ways in which a laser beam may enter vertex C, bounce off 11 surfaces, then exit through the same vertex. One way is shown in the illustration; the other is the reverse of that.

There are 80840 ways in which a laser beam may enter vertex C, bounce off 1000001 surfaces, then exit through the same vertex.

In how many ways can a laser beam enter at vertex C, bounce off 12017639147 surfaces, then exit through the same vertex?

Becomes remarkably simple, once you make a crucial conceptual leap.

Recurring themes; essential tools

"Bignum" arithmetic

32-bit integers alone won't get you very far in Project Euler. 64-bit will go a bit further, but for many problems an "unlimited" integer class is indispensable. Scala provides an excellent wrapping of Java's `BigInt` class with fully overloaded arithmetic operators and useful implicit conversions.

Less than 1/2 of my Scala solutions get away with using `Int` only, and around 1/3 use `BigInt`

Rational arithmetic

A few (usually easy) problems can be tackled with conventional double precision floating-point numerics, but one of the key applications of "bignum" integers is to implement a rational number class with an unlimited precision numerator and denominator. This will be a `fundamental component` in any Project Euler toolkit (and Scala's implicits/overloading mean it's possible to implement a very usable one).

A `Rational` class is used in around 1/13 of my Scala solutions.

Euclidean GCD algorithm

A function equivalent to

```
def gcd(x:Int,y:Int):Int =
  if (x==0) y else gcd(y%x,x)
```

will be a crucial helper for any Rational class implementation (for reduction of numerator and denominator by their greatest common divisor), and will also find use in problems where the mathematics is specific to coprime integers.

A GCD function appears in 1/8 of my Scala solutions.

Prime numbers

So far, two tools have sufficed for all problems:

- Sieve of Eratosthenes - quickly generates the first 50 million or so primes in around 10s.
- Miller-Rabin primality test - conveniently provided by Scala's `BigInt.isProbablePrime`.

Prime sieving is used in around 1/6 of my Scala solutions; Miller-Rabin in only 1/40.

Combinatorics

Taking a set of elements and efficiently enumerating or searching combinations/permutations, subject to certain constraints, is a recurring theme in problems.

This field was completely unknown to me before I started this programme.

Dynamic Programming and Memoisation

Dynamic programming refers to identifying overlapping subproblems and avoiding solving them more than once. The implementation technique is generally to memoise a (recursive) subproblem-solving function; Scala makes this easy:

```
// Generic memoiser: memoise any function with a hashable argument
class Memo[K,V](f:(K=>V)) extends
  scala.collection.mutable.HashMap[K,V] {
    override def apply(k:K) = getOrElseUpdate(k,f(k))
}
```

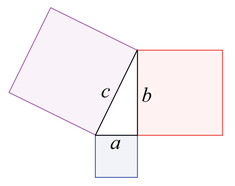
Usage example:

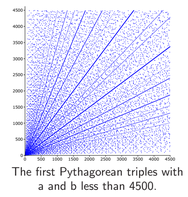
```
def g(x:Int)={...}  // Some expensive function
val h=new Memo(g)   // Memoised version of g
h(1);h(2);h(1);h(2) // Only invokes g twice
```

Memoisation is used in around 1/8 of my solutions.

Pythagorean triples

Pythagoran triples are integer solutions of $a^2 + b^2 = c^2$. Many problems require efficient generation of such triples; fortunately there are various well-known algorithms for doing this.





The first Pythagorean triples with a and b less than 4500.

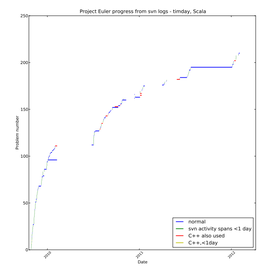
Continued fractions

Many problems require solution of Diophantine Equations (polynomial equations with integer solutions) and continued fractions are an important tool for solving the linear $ax + by = c$ case and also Pell's Equation $x^2 - ny^2 = 1$. They also have various other interesting properties; for example, the continued fraction representation $a_0, a_1, a_2, \cdots$ of any square root eventually repeats.

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \ddots}}}} \qquad \sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \ddots}}} \qquad \sqrt{11} = 3 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{3 + \cfrac{1}{6 + \ddots}}}}$$

...now in 2012

- Problems 1 to 218 solved in-order, avoiding the temptation to "cherry pick" easier problems. I've not yet attempted to compete for speed on newer problems.
- I've learned a lot of new-to-me mathematics, and enjoyed revisiting old ground.
- My Scala knowledge remains little developed beyond the basics needed to solve problems with it. Project Euler is far more about mathematics than programming.
- In Python, I find myself using list comprehensions and map/filter/reduce/lambda much more than I did previously. I'm also impressed by the power of immutability to tame complexity, and the Scala parallel collections library is a showcase for how easy parallelism should be.



For more information contact timday@bottlenose.demon.co.uk

"Each problem that I solved became a rule which served afterwards to solve other problems." - René Descartes