4. Euclid, Fibonacci, Lamé & Lucas

All the problems of the world could be settled easily, if men were only willing to *think*.

Thomas Watson Sr.

Please read §31.2 Greatest Common Divisor in the textbook (Corman et al., 2009).

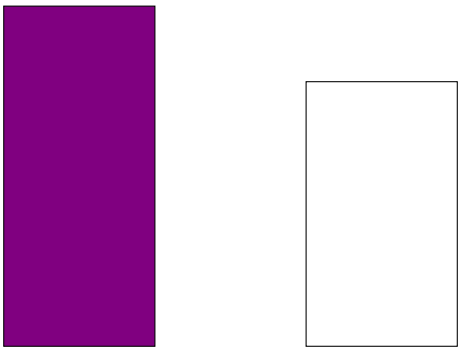
Wikipedia states one flagon is about 1.1 liters. My exchange range is made up for its simple arithmetic properties.

This experiment may be more fun if you drink the wine while performing it!

Imagine riding a time machine back to 457 BC. Traveling around this long-ago world you'll need to exchange money. Perhaps you'll need to convert lengths, volumes, and weights from modern to ancient units. For illustration, consider calculating the conversion factor from liters to flagons. Pretend 9 liters is equivalent to 7 flagons, but we don't know this yet! We have to calculate it. The Euclidean algorithm provides an efficient way to calculate conversion ratios. It finds a common measure (the greatest common divisor) for liters and flagons and uses this common unit to measure both.

$$\begin{aligned} 9 \text{ liters} &= 7 \text{ flagons} \\ 1 \text{ liter} &= 0.7777\cdots \text{ flagons} \\ 1.285714\cdots \text{ liters} &= 1 \text{ flagon} \end{aligned}$$

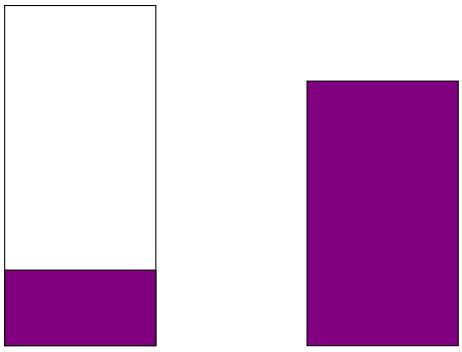
View the stylized drawing below: It show a flagon full of wine on the left and an empty liter on the right.



Full flagon Empty liter

When the wine is poured into the liter, a residue remains in the flagon. We'd like to know what fraction of a liter is left.

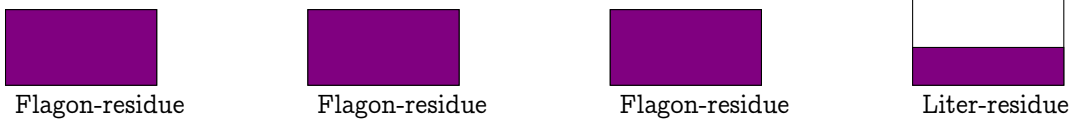
A flagon is one liter plus a small residue.



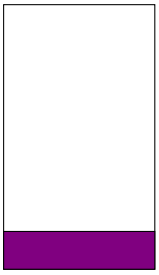
Flagon-residue Full liter

So, pour the full liter into containers each holding the amount remaining in the flagon. This takes three containers and leaves a smaller residue.

A liter is three flagon residues plus a smaller residue.



Flagon-residue Flagon-residue Flagon-residue



Liter-residue

A flagon residue is two liter residues with nothing remaining.

Now, we'd like to know the fraction of the flagon-residue is this liter-residue. So, we pour the flagon-residue into a containers each equal to the amount of the liter-residue. This takes two containers and leaves no residue.

Empty-flagon-residue Liter-residue Liter-residue

To recapitulate, a liter residue is a common measure for liters and flagons. A liter is 7 liter residues; A flagon is 9 liter residues;

$$\begin{aligned} 1 \text{ flagon} &= 1 \text{ liter} + \text{ flagon-residue} & 9 &= 7 + 2 \\ 1 \text{ liter} &= 2 \text{ flagon-residue} + \text{ liter-residue} & 7 &= 2 \cdot 3 + 1 \\ 1 \text{ flagon-residue} &= 2 \text{ liter-residue} & 2 &= 1 \cdot 2 + 0 \end{aligned}$$

Now, share the wine with your friends as you do the math. To compute the ratio $9 : 7$, run the above equations backwards. From the last equation,

$$1 \text{ flagon-residue} = 2 \text{ liter-residue}.$$

and

$$1 \text{ liter-residue} = \tfrac{1}{2} \text{ flagon-residues}$$

Therefore,

$$1 \text{ liter} = 3 \text{ flagon-residue plus } 1 \text{ liter-residue} = \tfrac{7}{2} \text{ flagon-residues}$$

and

$$1 \text{ flagon-residue} = \tfrac{2}{7} \text{ liters}$$

Finally,

$$1 \text{ flagon} = 1 \text{ liter plus } 1 \text{ flagon-residue} = \tfrac{9}{7} \text{ liters}$$

A theorem helps to explain the algorithm.

Theorem 4: Euclidean division

Given two integers a and m, with $m \neq 0$, there exist unique integers q and r such that

$$a = mq + r \quad \text{and} \quad 0 \leq r < |m|.$$

The dividend a equals the divisor m times the quotient q plus the remainder r.

Here's how the Euclidean algorithm computes $\gcd(34, 21)$ occurs.

In example , see how the divisor m and remainder r values shift down and left (southwest) at each step.

Example: Compute $\gcd(34, 21)$

Dividend a	$=$	*Divisor* m	$\cdot$	*Quotient* q	$+$	*Remainder* r
34	$=$	21	$\cdot$	1	$+$	13

The last divisor, where the remainder is 0, is the greatest common divisor. In this case, $\gcd(34, 21) = 1$.

Definition 6: Greatest Common Divisor

The greatest common divisor of two integers a and m is the

largest integer that divides them both.

$$\gcd(a, m) = \max\{k \mid k \backslash a \text{ and } k \backslash m\}$$

The Euclidean algorithm iterates the Euclidean division equation using the recursion: Let $r_0 = a$ and $r_1 = m > 0$, and assume $m \leq a$. Euclid's algorithm computes

$$\begin{aligned}
r_0 &= r_1 q_1 + r_2 && 0 \leq r_2 < r_1 \\
r_1 &= r_2 q_2 + r_3 && 0 \leq r_3 < r_2 \\
&\vdots = \vdots \\
r_{n-2} &= r_{n-1} q_{n-1} + r_n && 0 \leq r_n < r_{n-1} \\
r_{n-1} &= r_n q_n
\end{aligned}$$

The iteration halts when $r_{n+1} = 0$, and the last divisor (non-zero remainder) r_n is the greatest common divisor of a and m

Coding the Euclidean algorithm

The code for the Euclidean algorithm is often based on the identity in theorem 5.

Theorem 5: Greatest Common Divisor Recurrence

Let $0 \leq m < a$. Then,

$$\begin{aligned}
\gcd(a, 0) &= a \\
\gcd(a, m) &= \gcd(m, a \bmod m), \quad \text{for } m > 0
\end{aligned}$$

In C, the code might look something like this:

Listing 7: Imperative GCD algorithm

42a ⟨*Imperative GCD algorithm* 42a⟩≡

```
int gcd(int a, int m) {
  ⟨GCD local state 42b⟩
  while (⟨The divisor m is not 0 43b⟩) {
    ⟨Move m to a and a mod m to m 43a⟩
  }
  ⟨Return the absolute value of a 43c⟩
}
```

To change the values: m goes to a, and $a \bmod m$ goes to m, a local temporary value `t` is used.

42b ⟨*GCD local state* 42b⟩≡

```
int t;
```

43a ⟨*Move* `m` *to* `a` *and* `a mod m` *to* `m` 43a⟩≡
```
t = a;
a = m;
m = t % a;
```

C is not type safe. Instead of a Boolean test `(m == 0)` you can use the (wrong type) integer `m` itself.

43b ⟨*The divisor* `m` *is not 0* 43b⟩≡
```
m
```

The greatest common divisor is a positive integer. So, just in case the negative value was computed, change the answer to the absolute value of a before returning it.

43c ⟨*Return the absolute value of* `a` 43c⟩≡
```
return a < 0 ? -a : a;
```

Function GCD algorithm

Here's a functional implementation written in Haskell. It is from the standard Prelude for Haskell. It uses the idea that there is no largest integer that divides 0: All integers divide 0. Therefore, gcd(0, 0) is undefined and raises an error.

HASKELL supports elegant methods for handling errors, but here we just raise our hands and give up.

Listing 8: Functional GCD algorithm

44 ⟨*Functional GCD algorithm* 44⟩≡

```
gcd     :: (Integral a) => a -> a -> a
gcd 0 0 =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y =  gcd' (abs x) (abs y) where
  gcd' a 0  =  a
  gcd' a m  =  gcd' m (a `rem` m)
```

Analyzing the Euclidean algorithm

It is fitting that this early algorithm was also one of the first to have its complexity analyzed (Shallit, 1994). The result is called Lame's theorem, which incorporates the Fibonacci sequence, later widely popularized by Édouard Lucas.

The complexity of the Euclidean algorithm can be measured by the number of quotient–remainder steps taken. Seven steps are taken when computing gcd(34, 21), (see example).

The time complexity of the Euclidean algorithm is reduced least when each quotient is 1, except the last. For instance, when the greatest common divisor is 1, the last quotient is 2, and all other quotients are 1, terms in the Fibonacci sequence is produced. Running the Euclidean algorithm equations backwards, see Fibonacci sequence:

Recall, the Fibonacci sequence is

$$\vec{F} = \langle 0, 1, 1, 2, 3, 5, 8, 13, \ldots \rangle$$

and indexed from 0, that is $f_0 = 0$.

$$\begin{aligned} 2 &= 1 \cdot 2 + 0 \\ 3 &= 2 + 1 \\ 5 &= 3 + 2 \\ 8 &= 5 + 3 \\ 13 &= 8 + 5 \\ &\vdots \quad \vdots \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for } n \geq 3. \end{aligned}$$

Here's another Fibonacci-like example that show the worst-case time complexity of the Euclidean algorithm. In this case, the greatest com-

mon divisor is 2 and the last quotient is 5.

$$\begin{aligned} 10 &= 2 \cdot 5 + 0 \\ 12 &= 10 + 2 \\ 22 &= 12 + 10 \\ 34 &= 22 + 12 \\ &\vdots \\ 10F_n + 2F_{n-1} &\text{ for } n \geq 3. \end{aligned}$$

In the general case, when g is the greatest common divisor and q is the last quotient and all other quotients are 1, the steps come from a Fibonacci-like sequence.

$$\begin{aligned} gq &= g \cdot q + 0 \\ gq + g &= gq + g \\ 2gq + g &= (gq + g) + gq \\ 3gq + 2g &= (2gq + g) + (gq + g) \\ 5gq + 3g &= (3gq + 2g) + (2gq + q) \\ gqF_n + gF_{n-1} &\text{ for } n \geq 3. \end{aligned} \qquad \vdots \qquad \vdots$$

The asymptotic growth of F_n is $O(\varphi^n)$ where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio. Therefore, when m and a are consecutive terms in a Fibonacci-like sequence the Euclidean algorithm takes n steps where

$$n = O(\log_\varphi(a))$$

Theorem 6: Lamé's Theorem

Let $a, m \in \mathbb{Z}^+$ with $a \geq m > 0$. Let n be the number of quotient–remainder steps in Euclidean algorithm. Then

$$n \leq 1 + 3 \lg m$$

Proof: Lamé's Theorem

Let $r_0 = a$ *and* $r_1 = m$. *Euclid's algorithm computes*

$$\begin{aligned} r_0 &= r_1 q_1 + r_2 \qquad 0 \le r_2 < r_1 \\ r_1 &= r_2 q_2 + r_3 \qquad 0 \le r_3 < r_2 \\ &\vdots \\ r_{n-2} &= r_{n-1} q_{n-1} + r_n \qquad 0 \le r_n < r_{n-1} \\ r_{n-1} &= r_n q_n \end{aligned}$$

using n *divisions to compute* $r_n = \gcd(a, m)$. *Note that*

- $q_i \ge 1$, $i = 1, 2, \ldots, (n-1)$
- $(r_n < r_{n-1}) \Rightarrow (q_n \ge 2)$

Let F_i *denote the* i^{th} *Fibonacci number. Then*

$$\begin{aligned} r_n &\ge 1 = F_2 \\ r_{n-1} &= r_n q_n \ge 2 r_n \ge 2 = F_3 \\ r_{n-2} &\ge r_{n-1} + r_n \ge F_3 + F_2 = F_4 \\ &\vdots \\ r_2 &\ge r_3 + r_4 \ge F_{n-1} + F_{n-2} = F_n \\ r_1 &\ge r_2 + r_3 \ge F_n + F_{n-1} = F_{n+1} \end{aligned}$$

Using the growth rate of the Fibonacci numbers $F_{n+1} \approx \varphi^{n-1}$, *we find*

$$m = r_1 \ge F_{n+1} > \varphi^{n-1}$$

Take the logarithm base φ *of both sides and use the change of base formula for logarithms to derive the inequality*

$$\log_\varphi m = \frac{\lg m}{\lg \varphi} > n - 1$$

Since $(\lg \varphi)^{-1} < 3$ *we have*

$$3 \lg m > \frac{\lg m}{\lg \varphi} > n - 1$$

Another way to state the result is that if m can be represented in k bits, then the number of divisions in Euclid's algorithm is less than 3 times the number of bits in m's binary representation.

Application

The least common multiple (lcm) is a number related to the greatest common divisor (gcd). You've learned of it, if not by name, when learning to add fractions: To add $5/3$ and $8/5$ use a common denominator. In this case $3 \cdot 5 = 15$.

Note the cross multiplication and addition of numerators and denominators: $5 \cdot 5 + 3 \cdot 8$.

$$\frac{5}{3} + \frac{8}{5} = \frac{25}{15} + \frac{24}{15} = \frac{49}{15}.$$

Coding the extended Euclidean algorithm

Bézout's identity provides the link between the greatest common divisor and solving linear congruence equations.

Theorem 7: Bezout's identity

Let $0 < m \leq a$. Then, there exists integers s and t such that

$$\gcd(a, m) = at + ms$$

That is, the greatest common divisor $\gcd(a, m)$ can be written as a *linear* combination of a and m.

Proof: Bezout's identity

Let $\mathbb{L} = \{ax + my > 0 : x, y \in \mathbb{Z}\}$ *be the set of all positive* linear combinations *of* a *and* m*, and let*

$$d = \min\{ax + my > 0 : x, y \in \mathbb{Z}\}$$

be the minimum value in $\mathbb{L}$*, Let* t *and* s *be values of* x *and* y *that give the* minimum *value* d*. That is,*

$$d = at + ms > 0$$

Let a *divided by* d *give quotient* q *and remainder* r*.*

$$a = dq + r, \quad 0 \leq r < d$$

Then

$$\begin{aligned} r &= a - dq \\ &= a(1 - tq) + m(sq) \\ &\in \{ax + my \geq 0 : x, y \in \mathbb{Z}\} \text{ and } 0 \leq r < d. \end{aligned}$$

But since d *is the smallest positive linear combination,* r *must be* 0 *and* d *divides* a*. A similar argument shows* d *divides* m*. That is,* d *is a common divisor of* a *and* m*. Finally, if* c *is any common divisor of* a *and* m*, then* c *divides* $d = at + ms$*. That is,* d *is the greatest common divisor of* a *and* m*.*

I found the code for the extended Euclidean algorithm here. The original is by by Trevor Dixon.

Listing 9: Haskell Extended Euclidean Algorithm

48 ⟨*Haskell Extended Euclidean algorithm* 48⟩≡

```
extendedEu :: Integer -> Integer -> (Integer, Integer)
extendedEu a 0 = (1, 0)
extendedEu a m = (t, s - q * t)
  where (q, r) = quotRem a m
        (s, t) = extendedEu m r
```

Exercises

1. Prove theorem 4.
2. What is the time complexity of the algorithm in listing 9.
3. Show that $\text{lcm}(a, m)\,\text{gcd}(a, m) = am$.
4. Write an least common multiple (lcm) algorithm in HASKELL and C.

Bibliography

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff. [page 120]

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15. [page 119]

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition.* Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language.* Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley, ACM Press. [page 147]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley. [page 193], [page 195], [page 205]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach.* Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It.* Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach.* Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules.* Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355. [page 119]

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857. [page 119]

Sedgewick, R. (2004). *Algorithms in Java.* Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index

HASKELL programming language, 20, 31, 44, 116, 137
JAVA programming language, 31

Ackermann function, 174
Axioms
  Peano, 192

Bézout, Étienne, 47

C programming language, 30, 31, 42, 43, 55, 109, 152, 153
Calendar, 7
Cartesian product, 142
Catalan numbers, 156, 161
Computability, 191
Computable function, 28
Computer Language Benchmark Game, 30
Contradiction, 193

Dijkstra's algorithm, 183
DNA, 146

Empty
  String λ, 61

Fibonacci, 44, 46, 142
Floating point
  Machine epsilon, 50
Function
  Ackermamm, 174
Functional programming, 31

Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstader, Douglas, 25

Kleene, Stephen, 53
Knuth, Donald, 30

Lamé, Gabreil, 44
λ calculus, 26, 28
Linear congruence, 47
Listing
  C
    Matrix Chain Multiplication, 158
Listings
  C
    A Minimum Algorithm, 135
    Aho-Corasick, 89
    Boyer-Moore Pattern Matching, 72
    Brute-force Left-to-Right Pattern Matching, 56
    Bubble Sort, 109
    Build a Heap, 126
    Counting Sort, 128
    Direct Address Table Operations, 94
    Edit Distance, 152
    Find Set with Path Compression, 177
    Greatest Common Divisor, 42
    Hash Table with Chaining, 95
    Heap Sort, 127
    Heapify a List, 126
    Inner Product, 155
    Insertion Sort, 112
    Knuth–Morris–Pratt Pattern Matching, 67
    Maximum Subsequence Sum (Cubic), 34
    Maximum Subsequence Sum (Linear), 35
    Merge Sort, 118
    Morris–Pratt Pattern Matching, 58
    Partition about a pivot, 120
    Quick Sort, 120
    Radix Exchange Sort, 132
    Radix Sort, 129
    Randomized Partition, 136
    Randomized Selection, 138
    Rational Knapsack, 167
    Selection Sort, 114
    Shell Sort, 116
    Union by rank, 177
  Haskell
    Ackermann function, 174
    Apply a Function Repeatedly, 50
    Brute-force Left-to-Right Pattern Matching, 57
    Bubble Function, 108
    Bubble Sort, 109
    Extended Euclidean Algorithm, 48
    Fold from the left, 37
    Greatest Common Divisor, 44
    Inner Product, 155
    Insertion Sort, 112
    Matrix Multiplication, 155
    Maximum Subsequence Sum, 37
    Merge Sort, 117
    Merging Sorted Lists, 117
    Newton's Square Root Method, 51
    Newton's Square Root Recurrence, 50
    Quicksort, 120
    Randomized Partition, 137
    Relative Error Convergece Test, 50
    Sample Code Header, 31
    Second of a pair, 36
    Selection Sort, 114
    Sorted Decision Problem, 106

Splitting a List, 117
Swapping head with Another Element, 137
Pseudocode
Activity Selection, 170
Bucket Sort, 130
Dijkstra's Algorithm, 186
Kruskal Minimum Spanning Tree, 172
Prim Minimal Spanning Tree, 181
Lucas, Édouard, 44

Memorization, 141
Metric sace, 152

Newton's method, 13
Newton, Issac, 49
Noweb, 30, 55

Object-oriented programming, 31

Pattern Matching, 53, 54
Boyer-Moore, 71
Brute Force, 55
Knuth–Morris–Pratt, 67
Morris–Pratt, 58
Right-to-Left Brute Force, 70
φ, 142
Presburger arithmetic, 191

RAM, 26, 29

Sequence, 19
Lucas, 44
Sets
Cartesian product, 142
Simpson's rule, 51
Sorting Algorithms
Bubble Sort, 107
Bucket Sort, 129
Counting Sort, 128
Heap Sort, 124
Insertion Srt, 111
Merge Sort, 116
Quick Sort, 119
Radix Sort, 129
Selection Sort, 114
Shell Sort, 115
Time complexity, 127
String
Border, 61
Period, 61
Prefix, 61
Suffix, 61
Syllabus, 5

Theorems
Bezout's, 47
Euclidean division, 41
Lamé's, 45
Taylor's, 49, 51
Turing
Machine, 9, 26, 28, 192

Wikipedia, 201

xkcd
Algorithms by Complexity, 25
Calendar of meaningful dates, 7
Haskell, 30
Ineffective Sorts, 105
Traveling Salesman Problem, 141