Python and Coding Theory

Course Notes, Spring 2009-2010

Prof David Joyner, wdj@usna.edu

March 22, 2010

Draft Version - work in progress

Acknowledgement: There are `XKCD` comics scattered throughout (`http://xkcd.com/`), created by Randall Munroe. I thank Randall Munroe for licensing his comics with a a Creative Commons Attribution-NonCommercial 2.5 License, which allows them to be reproduced here. Commercial sale of his comics is prohibited. I also have made use of William's Stein's class notes [St] and John Perry's class notes, resp., on their Mathematical Computation courses.

Except for these, and occasional brief quotations (which are allowed under Fair Use guidelines), these notes are copyright David Joyner, 2009-2010, and licensed under the Creative Commons Attribution-ShareAlike License.

Python is a registered trademark
(`http://www.python.org/psf/trademarks/`)

There are some things which cannot be learned quickly,
and time, which is all we have,
must be paid heavily for their acquiring.
They are the very simplest things,
and because it takes a man's life to know them
the little new that each man gets from life
is very costly and the only heritage he has to leave.

- *Ernest Hemingway* (From A. E. Hotchner, **Papa Hemingway**, Random House, NY, 1966)

Contents

1 Motivation **8**

2 What is Python? **9**
2.1 Exercises . . . 11

3 I/O **12**
3.1 Python interface . . . 12
3.2 Sage input/output . . . 13
3.3 SymPy interface . . . 16
3.4 `IPython` interface . . . 16

4 Symbols used in Python **16**
4.1 period . . . 17
4.2 colon . . . 17
4.3 comma . . . 18
4.4 plus . . . 19
4.5 minus . . . 19
4.6 percent . . . 20
4.7 asterisk . . . 20
4.8 superscript . . . 20
4.9 underscore . . . 21
4.10 ampersand . . . 21

5 Data types **21**
5.1 Examples . . . 22
5.2 Unusual mathematical aspects of Python . . . 24

6 Algorithmic terminology **27**
6.1 Graph theory . . . 27
6.2 Complexity notation . . . 30

7 Keywords and reserved terms in Python **34**
7.1 Examples . . . 36
7.2 Basics on scopes and namespaces . . . 42
7.3 Lists and dictionaries . . . 43
7.4 Lists . . . 43
7.4.1 Dictionaries . . . 44

7.5 Tuples, strings . . . 47
7.5.1 Sets . . . 49

8 Iterations and recursion **50**
8.1 Repeated squaring algorithm . . . 50
8.2 The Tower of Hanoi . . . 51
8.3 Fibonacci numbers . . . 55
8.3.1 The recursive algorithm . . . 56
8.3.2 The matrix-theoretic algorithm . . . 58
8.3.3 Exercises . . . 59
8.4 Collatz conjecture . . . 59

9 Programming lessons **62**
9.1 Style . . . 62
9.2 Programming defensively . . . 63
9.3 Debugging . . . 64
9.4 Pseudocode . . . 71
9.5 Exercises . . . 75

10 Classes in Python **76**

11 What is a code? **78**
11.1 Basic definitions . . . 78

12 Gray codes **80**
12.1 Binary Gray codes . . . 80
12.2 Non-binary Gray codes . . . 85
12.3 An application of Gray codes to mathematics . . . 88

13 Huffman codes **90**
13.1 Exercises . . . 93

14 Error-correcting, linear, block codes **93**
14.1 The communication model . . . 94
14.2 Basic definitions . . . 94
14.3 Decoding . . . 96
14.4 The covering radius . . . 98
14.5 Finite fields . . . 98
14.5.1 A simple Python class for a prime finite fields . . . 101

14.6 Repetition codes 108
14.7 Hamming codes 109
14.7.1 Binary Hamming codes 109
14.7.2 Decoding Hamming codes 110
14.7.3 Non-binary Hamming codes 111
14.8 The Singleton bound 112
14.9 Dual codes 113
14.10 Reed-Muller codes 115

15 Cryptography **116**
15.1 Linear feedback shift register sequences 117
15.1.1 Linear recurrence equations 118
15.1.2 Golumb's conditions 119
15.1.3 Exercises 123
15.2 RSA 123
15.3 Diffie-Hellman 125

16 Matroids **127**
16.1 Matroids from graphs 128
16.2 Matroids from linear codes 130

17 Class projects **131**

18 Labs and tests **134**
18.1 Computer Lab 1 134
18.2 Computer Lab 2 136
18.3 Computer Lab 3 141
18.4 Computer Lab 4 143
18.5 Take-home Test 1 143

These are lecture notes for a course on Python and coding theory designed for students who have little or no programmig experience. The text is [B],

> N. Biggs, **Codes: An introduction to information, communication, and cryptography**, Springer, 2008.

No text for Python is officially assigned. There are many excelnt ones, some free (in pdf form), some not. One of my personal favorites is David Beazley's [Be], but I know people who prefer Mark Lutz and David Ascher's [LA]. Neither are free. There are also excellent books which are are free, such as [TP] and [DIP]. Please see the references at the end of these notes. I have really tried to include *good* refereences (at least, references on Python that I realy liked), not just throw in ones that are related. It just happens that there are a lot of good free references for learning Python. The MIT Python programming course [GG] also does not use a text. They do however, list as an optional reference

> Zelle, John. **Python Programming: An Introduction to Computer Science,** Wilsonville, OR: Franklin, Beedle & Associates, 2003.

(Now I *do* mention this text for completeness.) For a cryptography reference, I recommend the Handbook of Applied Cryptography [MvOV]. For a more complete coding theory reference, I recommend the excellent book by Cary Huffman and Vera Pless [HP].

You will learn some of the Python computer programming language and selected topics in "coding theory". The material presented in the actual lectures will probably not follow the same linear ordering o these notes, as I will probably bring in various examples from the later (mathematical) sections when discussing the earlier sections (on programming and Python).

I wish I could teach you all about Python, but there are some limits to how much information can be communicated in one semester! We broadly interprete "coding theory" to mean error-correcting codes, communication codes (such as Gray codes), cryptography, and data compression codes. We will introduce these topics and discuss some related algorithms implemented in the Python programs.

A programming language is a language which allows us to create programs which perform data manipulations and/or computations on a computer. The basic notions of a programming language are "data", "operators", and "statements." Some basic examples are included in the following table.

Data	Operators	Statements
numbers	+, - , *, ...	assignment
strings	+ (or concatenation)	input/output
Booleans	and, or	conditionals, loops

Our goal is to try to understand how basic data types are represented, what types of operations or manipulations Python allows to be performed on them, and how one can combine these into statements or Python commands. The focus of the examples will be on mathematics, especially coding theory.

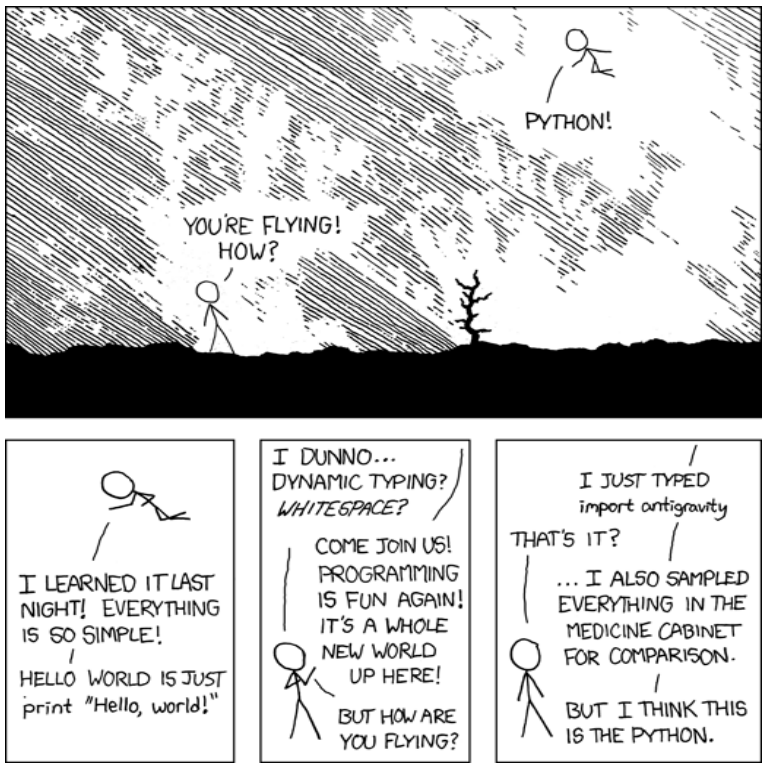


Figure 1: Python.

`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

1 Motivation

Python is a powerful and widely used programming language.

- "Python is fast enough for our site and allows us to **produce maintainable features in record times**, with a minimum of developers," said Cuong Do, Software Architect, **YouTube.com.**

- "Google has made no secret of the fact they use Python a lot for a number of internal projects. Even knowing that, once **I was an employee, I was amazed at how much Python code there actually is in the Google source code system**.", said Guido van Rossum, **Google**, creator of Python.

  Speaking of **Google**, Peter Norvig, the Director of Research at Google, is a fan of Python and an expert in both management and computers. See his very interesting article [N] on learning computer programming. *Please* read this short essay.

- "Python plays a key role in our production pipeline. Without it a project the size of **Star Wars: Episode II** would have been very difficult to pull off. From crowd rendering to batch processing to compositing, Python **binds all things together**," said Tommy Burnette, Senior Technical Director, **Industrial Light & Magic.**

Python is often used as a *scripting language* (i.e., a programming language that is used to control software applications). Javascript embedded in a webpage can be used to control how a web browser such as Firefox displays web content, so javascript is a good example of a scripting language. Python can be used as a scripting language for various applications (such as Sage [S]), and is ranked in the top 5-10 worldwide in terms of popularity.

Python is fun to use. In fact, the origin of the name comes from the television comedy series Monty Python's Flying Circus and it is a common practice to use Monty Python references in example code. It's okay to laugh while programming in Python (Figure 1).

According to the Wikipedia page on Python, Python has seen extensive use in the information security industry, and has been used in a number of commercial software products, including 3D animation packages such as Maya and Blender, and 2D imaging programs like GIMP and Inkscape.

Please see the bibliography for a good selection of Python references. For example, to install Python, see the video [YTPT] or go to the official Python website `http://www.python.org` and follow the links. (I also recommend installing `IPython` `http://ipython.scipy.org/moin/`.)

2 What is Python?

Confucius said something like the following: "If your terms are not used carefully then your words can be misinterpreted. If your words are misinterpreted then events can go wrong." I am probably misquoting him, but this was the idea which struck me when I heard this some time ago. That idea resonates in both mathematics and in computer programming. Statements must be constructed from carefully defined terms with a clear and unambiguous meaning, or things can go wrong.

Python is a computer programming language designed for readability and functionality. One of Python's design goals is that the meaning of the code is easily understood because of the very clear syntax of the language. The Python programming language has a specific syntax (form) and semantics (meaning) which enables it to express computations and data manipulations which can be performed by a computer.

Python's implementation was started in 1989 by Guido van Rossum at CWI (a national research institute in the Netherlands) as a successor to the ABC programming language (an obscure language made more popular by the fact that it motivated Python!). Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, *Benevolent Dictator for Life* (BDFL).

Python is an interpreted language, i.e., a programming language whose programs are not directly executed by the host cpu but rather executed (or"interpreted") by a program known as an interpreter. The source code of a Python program is translated or (partially) compiled to a "bytecode" form of a Python "process virtual machine" language. This is in distinction to C code which is compiled to cpu-machine code before runtime.

Python is a "dynamically typed" programming language. A programming language is said to be **dynamically typed**, when the majority of its type checking is performed at run-time as opposed to at compile-time. Dynamically typed languages include JavaScript, Lisp, Lua, Objective-C, Python, Ruby, and Tcl.

The data which a Python program deals with must be described precisely. This description is referred to as the **data type**. In the case of Python, the fact that Python is dynamically typed basically means that the interpreter or compiler will figure out for you what type a variable is at run-time, so you don't have to declare variable types yourself. The fact that Python is

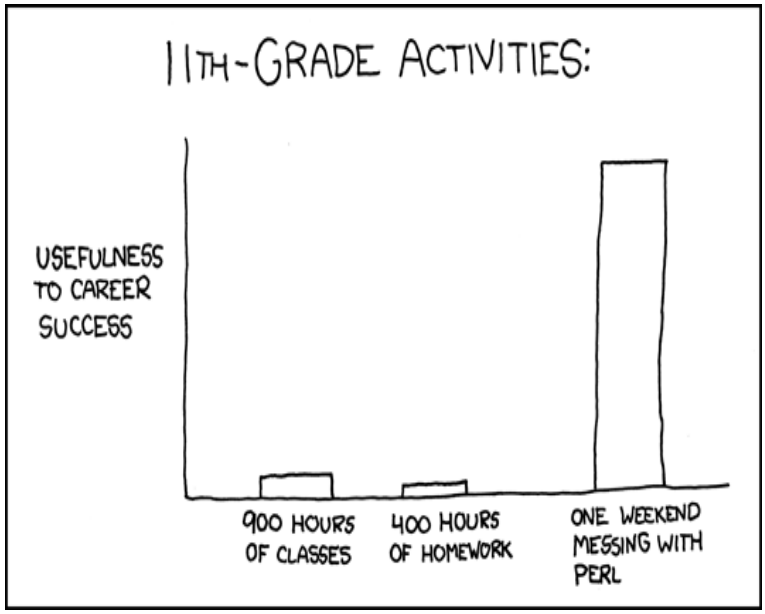


Figure 2: **11th grade.** (You may replace Perl by Python if you wish:-)
`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

"strongly typed" means[1] that it will actually raise a run-time type error when you have violated a Python grammar/syntax rule as to how types can be used together in a statement.

Of course, just because Python is dynamically and strongly typed does not mean you can neglect "type discipline", that is carelessly mixing types in your statements, hoping Python to figure out things.

Here is an example showing how Python can figure out the type from the command at run-time.

Python

```
>>> a = 2012
>>> type(a)
<type 'int'>
>>> b = 2.011
```

[1]A caveat: This terminology is not universal. Some computer scientists say that a strongly typed language must also be statically typed. A staticaly typed language is one in which the variables themselves, and not just the values, have a fixed type associated to them. Python is not statically typed.

```
>>> type(b)
<type 'float'>
```

The Python compiler can also "coerce" types as needed. In this example below, the interpreter coerces at runtime the integer `a` into a float so that it can compute `a+b`:

Python

```
>>> c = a+b
>>> c
2014.011
>>> type(c)
<type 'float'>
```

However, if you try to so something illegal, it will raise a type error.

Python

```
>>> 3+"3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Also, Python is an object-oriented language. Object-oriented programming (OOP) uses "objects" - data structures consisting of datafields and methods - to design computer programs. For example, a matrix could be the "object" you want to write programs to deal with. You could define a `class` of matrices and, for example, a method for that class might be addition (representing ordinary addition of matrices). We will return to this example in more detail later in the course.

2.1 Exercises

Exercise 2.1. *Install* Python *[Py] or* SymPy *[C] or* Sage *[S] (which contains them both, and more), or better yet, all three. (Don't worry they will not conflict with each other).*

Create a "hello world!" program. Print out it and your output and hand both in.

3 I/O

This section is on very basic I/O (input-output), so skip if you know all you need already.

How do you interface with

- Python,
- Sage (a great mathematical software system that includes Python and has its own great interface),
- SymPy (another great mathematical software system that includes Python and has its own great interface),
- `IPython` (a Python interface)?

This section tries to address these questions.

Another option is codenode which also runs Python in a nice graphical interface (`http://codenode.org/`) or `IDLE` (another Python command-line interface or CLI). Another way to learn about interfaces is to watch (for example) J. Unpingco's videos [Un] this.

3.1 Python interface

Python is available at `hht://www.python.org/` and works equally well on all computer platforms (MS Windows, Macs, Linux, etc.) Documentation for Python can be found at that website but see the references in the bibliography at the end as well.

The input prompt is `>>>`. Python does not print lines which are assignments as output. If it does print an output, the output will appear on a line without a `>>>`, as in the following example.

Python

```
>>> a = 3.1415
>>> print a
3.1415
>>> type(a)
<type 'float'>
```

Python has several ways to read in files which are filled with legal Python commands. One is the `import` command. This is really designed for Python "modules" which have been placed in specific places in the Python directory structure. Another is to "execute" the commands in the file, say `myfile.py`, using the Python command: `python myfile.py`.

To have Python read in a file of data, or to write data to a file, you can use the `open` command, which has both `read` and `write` methods. See the Python tutorial, `http://docs.python.org/tutorial/inputoutput.html` , for more details. Since Sage has a more convenient mechanism for this (see below), we shall not go into more details now.

3.2 Sage input/output

Sage is built on Python, so includes Python, but is designed for general purpose mathematical computation (the lead developer of Sage is a number-theorist). The interface to Sage is `IPython`, though it has been configured in a customized way to that the prompt says `sage:` as opposed to `In` or `>>>`. Other than this change in prompt, the command line interface to Sage is similar to that if Python and SymPy.

Sage

```
sage: a = 3.1415
sage: print a
3.14150000000000
sage: type(a)
<type 'sage.rings.real_mpfr.RealLiteral'>
```

Sage also include SymPy and a nice graphical interface (`http://www.sagenb.org/`), called the `Sage notebook`. The graphical interface to Sage works via a web browser (`firefox` is recommended, but most others should also work).

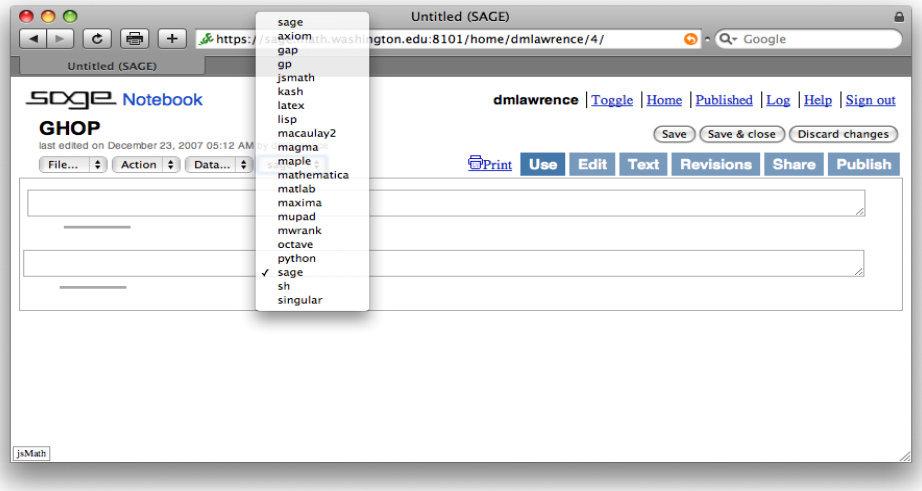


Figure 3: Sage **notebook interface .** The default interface is Sage but you can also select Python for example.

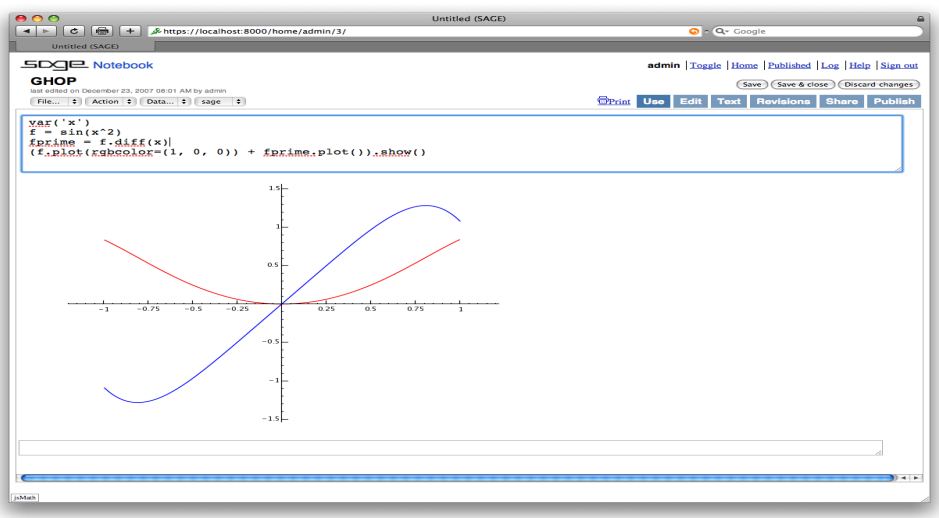


Figure 4: Sage **notebook interface .** You can plot two curves, each with their own color, on the same graph by simply "adding" them.

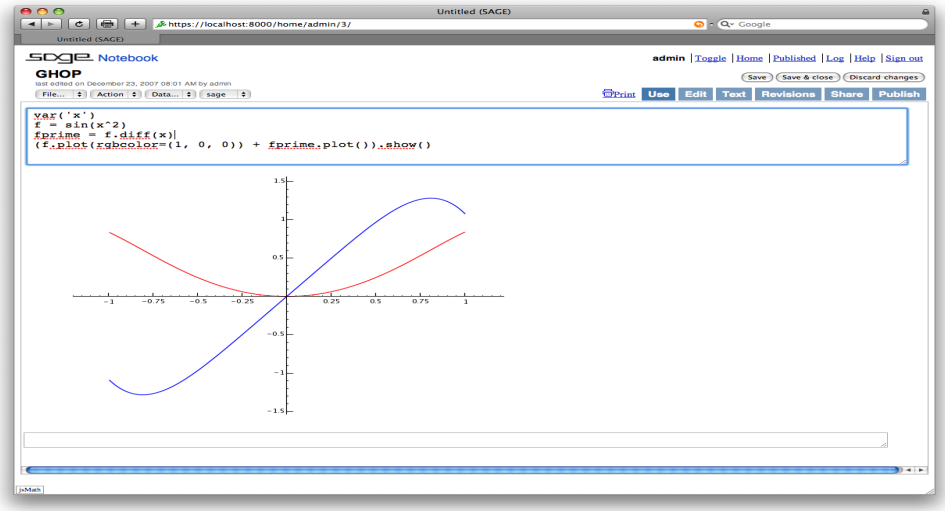


Figure 5: Sage **notebook interface .** Plots in 3 dimensions are also possible in Sage (3d-curves, surfaces and parametric plots). Sage creates this plot of the Rubik's cube, "under the hood", by "adding" lots of colored cubes.

See `http://www.flickr.com/photos/sagescreenshots/` or the Sage website for more screenshots.

You can try it out at `http://www.sagenb.org/`, but there are thousands of other users around the world also using that system, so you might prefer to install it yourself on your own computer.

Sage has a great way to read in files which are filled with legal Sage commands - it's called the `attach` command. Just type `attach 'myfilename'` in either the command-line version or the notebook version of Sage.

Sage also has a great way to communicate your worksheets with a friend (or any other Sage user):

- First, you can "publish" the worksheets on a webserver running Sage and send your friend the link to your worksheet. (Go to `http://www.sagenb.org/`, log in, and click on the "published" link for lots of examples.) If your friend has an account on the same Sage server, then all you need to do is "share" your saved worksheet with them (after clicking "share" you will go to another screen at which you type your friends account name into the box provided and click "invite").

- Second, you can download your worksheet to a file `myworksheet.sws` (they always end in `sws`) and email that file to someone else. They can either open it using a copy of Sage they have on their own computer, or go to a public Sage server like `http://www.sagenb.org/`, log in, and upload your file and open it that way.

3.3 SymPy interface

SymPy is also available for all platforms.

SymPy is built on Python, so includes Python, but is designed for people who are mostly interested in applied mathematical computation (the lead developer of SymPy is a geophysicist). The interface to SymPy is `IPython`, which is a convenient and very popular Python shell/interface which has a different (default) prompt for input. Each input prompt looks like `In [n]:` as opposed to `>>>`.

SymPy

```
In [1]: a = 3.1415

In [2]: print a
------> print(a)
3.1415

In [3]: type(a)
Out[3]: <type 'float'>
```

More information about SymPy is available form its website `http://www.sympy.org/`.

3.4 `IPython` interface

`IPython` is an excellent interface but it is visually the same as SymPy's interface, so there is nothing new to add. See `http://www.ipython.org/` (or `http://ipython.scipy.org/moin/`) for more information about `IPython`.

4 Symbols used in Python

What are symbols such as `.`, `:`, `,`, `+`, `-`, `%`, `^`, `*`,`\_`, and `&`, used for in Python?

4.1 period

The *period* . This symbol is used by Python is several different ways.

- It can be used as a separator in an `import` statement.

  Python
  ```
  >>> import math
  >>> math.sqrt(2)
  1.4142135623730951
  ```

  Here `math` is a Python module (i.e., a file named `math.py`) somewhere in your Python directory and `sqrt` is a function defined in that file.

- It can be used to separate a Python object from a method which applies to that object. For example, `sort` is a method which applies to a list; `L.sort()` (as opposed to the functional notation `sort(L)`) is the Python-ic, or object-oriented, notation for the `sort` command. In other words, we often times (but not always, as the above `sqrt` example showed) put the function *behind* the argument in Python.

  Python
  ```
  >>> L = [2,1,4,3]
  >>> type(L)
  <type 'list'>
  >>> L.sort()
  >>> L
  [1, 2, 3, 4]
  ```

4.2 colon

The *colon* : is used in several ways. First, it appears at the end of each `def` statement, `for` statement, `if` statement, and `while` statement, and signals that an indentation must be used in the next block of statements. It is also in the `lambda` statement. The colon is also used for manipulating lists. It comprises the so-called *slice* notation for constructing sublists.

Python
```
>>> L = [1,2,3,4,5,6]
>>> L[2:5]
[3, 4, 5]
>>> L[:-1]
```

```
[1, 2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
>>> L[2:]
[3, 4, 5, 6]
```

By the way, slicing also works for tuples and strings.

Python

```
>>> s = "123456"
>>> s[2:]
'3456'
>>> a = 1,2,3,4
>>> a[:2]
(1, 2)
```

I tried to think of a joke with "slicing", "dicing", "Veg-O-Matic" , and "Python" in it but failed. If you figure one out, let me know! (I give a link in case you are too young to remember the ads: remember the `http://en.wikipedia.org/wiki/Veg-O-Matic`.)

4.3 comma

The *comma* **,** is used in ways you expect. However, there is one nice and perhaps unexpected feature.

Python

```
>>> a = 1,2,3,4
>>> a
(1, 2, 3, 4)
>>> a[-1]
4
>>> r,s,u,v = 5,6,7,8
>>> u
7
>>> r,s,u,v = (5,6,7,8)
>>> v
8
>>> (r,s,u,v) = (5,6,7,8)
>>> r
5
```

You can finally forget parentheses and not get yelled at by your mathematics professor! In fact, if you actually do forget them, other programmers will

think you are realy cool since they think that means you know about Python *tuple packing*! Python adds parentheses in for you automatically, so don't forget to drop parentheses next time you are using tuples.
`http://docs.python.org/tutorial/datastructures.html`

4.4 plus

The *plus* + symbol is used of course in mathematical expressions. However, you can also add lists, tuples and strings. For those objects, + acts by concatenation.

Python

```
>>> words1 = "Don't"
>>> words2 = "skip class tomorrow!"
>>> words1+" "+words2
"Don't skip class tomorrow!"
```

Notice that the nested quote symbol in `words1` doesn't bother Python. You can either use single quote symbols, `'`, or double quote symbols `"` to define a string, and nesting is allowed.

Concatenation works on tuples and lists as well.

Python

```
>>> a = 1,2,3,4
>>> a[2:]
(3, 4)
>>> a[:2]
(1, 2)
>>> a[2:]+a[:2]
(3, 4, 1, 2)
>>> a[:2]+a[2:]
(1, 2, 3, 4)
```

4.5 minus

The *minus* - sign is used of course in mathematical expressions. It is (unlike +) also used for `set` objects. It is not used for lists, strings or tuples.

Python

```
>>> s1 = set([1,2,3])
>>> s2 = set([2,3,4])
>>> s1-s2
set([1])
```

```
>>> s2-s1
set([4])
```

4.6 percent

The *percent* % symbol is used for modular arithmetic operations in Python. If m and n are positive integers (say $n > m$) then `n%m` means the remainder after dividing m into n. For example, dividing 5 into 12 leaves 2 as the remainder. The remainder is an integer r satisfying $0 \leq r < m$.

Python

```
>>> 12%5
2
>>> 10%5
0
```

4.7 asterisk

The *asterisk* * is the symbol Python uses for multiplication of numbers. When applied to lists or tuples or strings, it has another meaning.

Python

```
>>> L = [1,2,3]
>>> L*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 2*L
[1, 2, 3, 1, 2, 3]
>>> s = "abc"
>>> s*4
'abcabcabcabc'
>>> a = (0)
>>> 10*a
0
>>> a = (0,)
>>> 10*a
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

4.8 superscript

The *superscript* ^ in Python is not used for mathematical exponentiation! It is used as the Boolean operator "exclusive or" (which can get confusing at times ...). Mathematically, it is used as the union of the set-theoretic differences, i.e., the elements in exactly one set but not the other.

Python

```
>>> s1 = set([1,2,3])
>>> s2 = set([2,3,4])
>>> s1-s2
set([1])
>>> s2-s1
set([4])
>>> s1^s2
set([1, 4])
```

Python does mathematical exponentiation using the double asterisk.

Python

```
>>> 2**3
8
>>> (-1)**2009
-1
```

4.9 underscore

The *underscore* _ is only used for variable, function, or module names. It does not act as an operator.

4.10 ampersand

The *ampersand* `&` sign is used for intersection of `set` objects. It is not used for lists, strings or tuples.

Python

```
>>> s1 = set([1,2,3])
>>> s2 = set([2,3,4])
>>> s1&s2
set([2, 3])
```

5 Data types

> the lyf so short, the craft so long to lerne
> - *Chaucer (1340-1400)*

Python data types are described in `http://docs.python.org/library/datatypes.html`. Besides numerical data types, such as `int` (for integers) and `float` (for reals), there are other types such as `tuple` and `list`. A more complete list, with examples, is given below.

Type	*Description*	*Syntax example*
`str`	An immutable sequence of Unicode characters	`"string", """\python is great""", '2012'`
bytes	An immutable sequence of bytes	`b'Some ASCII'`
`list`	Mutable, can contain mixed types	`[1.0, 'list', True]`
`tuple`	Immutable, can contain mixed types	`(-1.0, 'tuple', False)`
set, frozenset	Unordered, contains no duplicates	`set([1.2, 'xyz', True]), frozenset([4.0, 'abc', True])`
`dict`	A mutable group of key and value pairs	`{'key1': 1.0, 'key2': False}`
`int`	An immutable fixed precision number of unlimited magnitude	`42`
`float`	An immutable floating point number (system-defined precision)	`2.71828`
`complex`	An immutable complex number with real and imaginary parts	`-3 + 1.4j`
`bool`	An immutable Boolean value	`True`, `False`

5.1 Examples

Some examples illustrating some Python types.

Python

```
>>> type("123") ==str
True
>>> type(123) ==str
False

>>> type("123") ==int
False
>>> type(123) ==int
True

>>> type(123.1) == float
True
>>> type("123") == float
False
>>> type(123) == float
False
```

The next examples illustrate syntax for Python tuples, lists and dictionaries.

Python

```
>>> type((1,2,3))==tuple
True
>>> type([1,2,3])==tuple
False
>>> type([1,2,3])==list
True
>>> type({1,2,3})==tuple  # set-theoretic notation is not allowed
  File "<stdin>", line 1
    type({1,2,3})==tuple
           ^
SyntaxError: invalid syntax
>>> type({1:"a",2:"b",3:"c"})==tuple
False
>>> type({1:"a",2:"b",3:"c"})
<type 'dict'>
>>> type({1:"a",2:"b",3:"c"})==dict
True
```

Note you get a syntax error when you try to enter illegal syntax (such as set-theoretic notation to describe a set) into Python.

However, you can enter sets in Python, and you can efficiently test for membership using the `in` operator.

Python

```
>>> S = set()
>>> S.add(1)
>>> S.add(2)
>>> S
set([1, 2])
>>> S.add(1)
>>> S
set([1, 2])
>>> 1 in S
True
>>> 2 in S
True
>>> 3 in S
False
```

Of course, you can perform typical set theoretic operations (e.g., `union`, `intersection`, `issubset`, ...) as well.

5.2 Unusual mathematical aspects of Python

Print the floating point version of 1/10.

Python

```
>>> 0.1
0.10000000000000001
```

There is an interesting story behind this "extra" trailing 1 displayed above. Python is not trying to annoy you. It follows the IEEE 754 Floating-Point standard (`http://en.wikipedia.org/wiki/IEEE_754-2008`): each (finite) number is described by three integers: a *sign* (zero or one), s, a *significand* (or 'mantissa'), c, and an *exponent*, q. The numerical value of a finite number is $(-1)^s \times c \times b^q$, where b is the base (2 or 10). Python stores numbers internally in base 2, where $1 \leq c < 2$ (recorded to only a certain amount of accuracy) and, for 64-bit operating systems, $-1022 \leq q \leq 1023$. When you write 1/10 in base 2 and print the rounded off approximation, you get the funny decimal expression above.

If that didn't amuse you much, try the following.

Python

```
>>> x = 0.1
>>> x
0.10000000000000001
>>> s = 0
>>> print x
0.1
>>> for i in range(10): s+=x
...
>>> s
0.99999999999999989
>>> print s
1.0
```

The addition of errors creates a bigger error, though in the other direction! However, print does rounding, so the output of floats can have this schizophrenic appearance.

This is one reason why using SymPy or Sage (both of which are based on Python) is better because they replace Python's built-in mathematical functions with much better libraries. If you are unconvinced, look at the following example.

Python

```
>>> a = sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> a = math.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> import math
>>> a = math.sqrt(2)
>>> a*a
2.0000000000000004
>>> a*a == 2
False
>>> from math import sqrt
>>> a = sqrt(2)
>>> a
1.4142135623730951
```

Note the `NameError` exception raised form the command on the first line. This is because the Python`math` library (which contains the definition of the `sqrt` function, among others) is not automatically loaded. You can `import` the `math` library in several ways. If you use `import math` (which imports all the mathematical functions defined in `math`), then you have to remember to type `math.sqrt` instead of just `sqrt`. You can also only import the function which you want to use (this is the recommended thing to do), using `from math import sqrt`. However, this issue is is not a problem with SymPy or Sage.

Sage

```
sage: a = sqrt(2)
sage: a
sqrt(2)
sage: RR(a)
1.41421356237310
```

SymPy

```
In [1]: a = sqrt(2)

In [2]: a
Out[2]:
  ___
\/ 2

In [3]: a.n()
```

```
Out[3]: 1.41421356237310
```

And if you are not yet confused by Python's handling of floats, look at the "long" (`L`) representation of "large" integers (where "large" depends on your computer architecture, or more precisely your operating system, probably near 2^{64} for most computers sold in 2009). The following example shows that once you are an `L`, you stay in `L` (there is no getting out of `L`), even if you are number 1!

Python

```
>>> 2**62
4611686018427387904
>>> 2**63
9223372036854775808L
>>> 2**63/2**63
1L
```

Note also that the syntax in the above example did not use `^`, but rather `**`, for exponentiation. That is because in Python `^` is reserved for the Boolean `and` operator. Sage "preparses" `^` to mean exponentiation.

The Zen of Python, I

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

6 Algorithmic terminology

Since we will be talking about programs implementing mathematical procedures, it is natural that we will need some technical terms to abstractly describe features of those programs. For this reason, some really basic terms of graph theory and complexity theory will be helpful.

6.1 Graph theory

Graph theory is a huge and interesting field in its own, and a lifetime of courses could be taught on its various aspects and applications, so what we introduce here will not even amount to an introduction.

Definition 1. A *graph* $G = (V, E)$ is an ordered pair of sets, where V is a set of *vertices* (possibly with weights attached) and $E \subseteq V \times V$ is a set of *edges* (possibly with weights attached). We refer to $V = V(G)$ as the vertex set of G, and $E = E(G)$ the edge set. The cardinality of V is called the *order* of G, and $|E|$ is called the *size* of G.

If $e \in E$ is an edge and $v \in V$ is a vertex on either "end" of e then we say v is *incident* to e (or that e is *incident* to v). If u, v are vertices and $(u, v) \in E$ is an edge then u and v are called *adjacent* edges.

A *loop* is an edge of the form (v, v), for some $v \in V$. If the set E of edges is allowed to be a *multi-set* and if multiple edges are allowed then the graph is called a *multi-graph*. A graph with no multiple edges or loops is called a *simple* graph.

There are various ways to describe a graph. Suppose you want into a room with 9 other people. Some you shake hands with and some you don't. Construct a graph with 10 vertices, one for each person in the room, and draw and edge between two vertices if the associated people have shaken hands. Is there a "best" way to describe this graph? One way to describe the graph is to list (i.e., order) the people in the room and (separately) record the set of pairs of people who have shaken hands. This is equivalent to labeling the people 1, 2, ..., 10 and then constructing the 10×10 matrix $A = (a_{ij})$, where $a_{Ij} = 1$ if person i shook hands with person j, and $a_{ij} = 0$ otherwise. (This matrix A is called the "adjacency matrix' of the graph.) Another way to descibe the graph is to list the people in the room, but this time, attached to each person, add the set of all people that person shook hands with. This

way of describing a graph is related to the idea of a Python dictionary, and is called the "dictionary description."

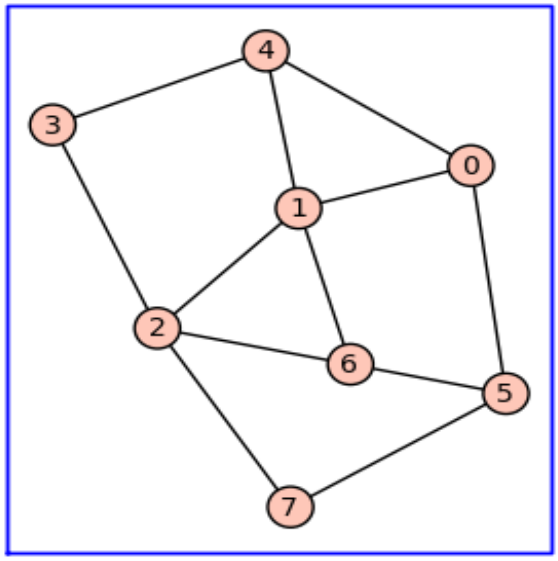


Figure 6: **A graph created using Sage.**

If no weights on the vertices or edges are specified, we usually assume all the weights are implicitly 1 and call the graph *unweighted*. A graph with weights attached, especially with edge weights, is called a *weighted graph*.

One can label a graph by attaching labels to its vertices. If $(v_1, v_2) \in E$ is an edge of a graph $G = (V, E)$, we say that v_1 and v_2 are *adjacent* vertices. For ease of notation, we write the edge (v_1, v_2) as v_1v_2. The edge v_1v_2 is also said to be *incident* with the vertices v_1 and v_2.

Definition 2. A *directed edge* is an edge such that one vertex incident with it is designated as the head vertex and the other incident vertex is designated as the tail vertex. A directed edge is said to be directed from its tail to its head. A *directed graph* or *digraph* is a graph such that each of whose edges is directed.

If u and v are two vertices in a graph G, a *u-v walk* is an alternating sequence of vertices and edges starting with u and ending at v. Consecutive

vertices and edges are incident. Notice that consecutive vertices in a walk are adjacent to each other. One can think of vertices as destinations and edges as footpaths, say. We are allowed to have repeated vertices and edges in a walk. The number of edges in a walk is called its *length*.

A graph is *connected* if, for any distinct $u, v \in V$, there is a walk connecting u to v.

A *trail* is a walk with no repeating edges. Nothing in the definition of a trail restricts a trail from having repeated vertices. Where the start and end vertices of a trail are the same, we say that the trail is a *circuit*, otherwise known as a *closed* trail.

A walk with no repeating vertices is called a *path*. Without any repeating vertices, a path cannot have repeating edges, hence a path is also a trail. A path whose start and end vertices are the same is called a *cycle*.

A graph with no cycles is called a *forest*. A connected graph with no cycles is called a *tree*. In other words, a tree is a connected forest.

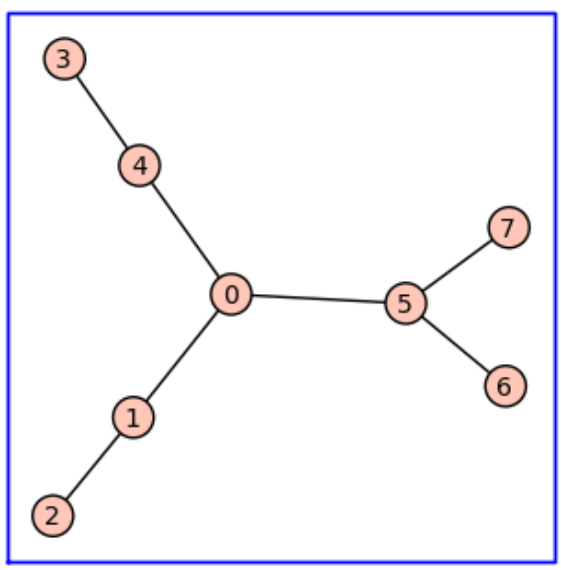


Figure 7: **A tree created using Sage.**

6.2 Complexity notation

There are many interesting (and very large) texts on complexity theory in theoretical computer science. However, here we merely introduce some new terms and notation to allow us to discuss how "complex" and algorithm or computer program is.

There are many ways to model complexity and the discussion can easily get diverted into technical issues in theoretical computer science. Our purpose in this section is not to be complete, or really even to be rigorously accurate, but merely to explain some notation and ideas that will help us discuss abstract features of an algorithm to help us decide which algorithm is better than another.

The first idea is simply a bit of technical notation which helps us compare the rate of growth (or lack of it) of two functions.

Let f and g be two functions of the natural numbers to the positive reals. We say f *is big-O of* g, written[2]

$$f(n) = O(g(n)), \qquad n \to \infty,$$

provided there are constant $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c \cdot g(n),$$

for all $n > n0$. We say f *is little-o of* g, written

$$f(n) = o(g(n)), \qquad n \to \infty,$$

provided for *every* constant $\epsilon > 0$ there is an $n_0 = n_0(\epsilon) > 0$ (possibly depending on ϵ) such that

$$f(n) \leq \epsilon \cdot g(n),$$

for all $n > n_0$. This condition is also expressed by saying

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

[2]This notation is due to Edmund Landau a great German number theorists. This notation can also be written using the Vinogradov notation $f(n) \ll g(n)$, though the "big-O" notation is much more common in computer science.

We say f *is big-theta of* g, written[3]

$$f(n) = \Theta(g(n)), \qquad n \to \infty,$$

provided both $f(n) = O(g(n))$ and $g(n) = O(f(n))$ hold.

Example 3. We have

$$n \ln(n) = O(3n^2 + 2n + 10),$$
$$3n^2 + 2n + 10 = \Theta(n^2),$$

and

$$3n^2 + 2n + 10 = o(n^3).$$

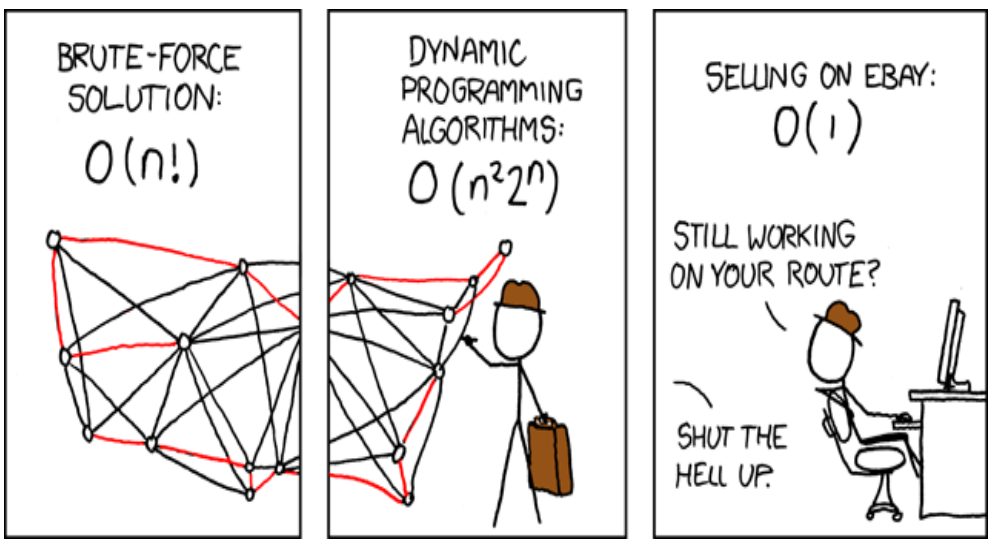


Figure 8: **Travelling Salesman Problem .**
`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

Here is a simple example of how this terminology could be used.

[3]This notation can also be written using the Vinogradov notation $f(n) \equiv g(n)$ or $f(n) \approx g(n)$, though the "big-theta" notation is much more common in computer science.

Suppose that an algorithm takes as input an n-bit integer. We say that algorithm has *complexity* $f(n)$ if, for all inputs of size n, the worst-case number of computations required to return the output is $f(n)$.

Some algorithms have really terrible worst-case complexity estimates but excellent "average-case complexity" estimates. This topic goes well beyond this course, but the (excellent) lectures of the video-taped course [DL] are a great place to learn more about these deeper aspects of the theory of algorithms (see, for example, the lectures on sorting).

Example 4. Consider the extended Euclidean algorithm. This is an algorithm for finding the greatest common divisor (GCD) of integers a and b which also finds integers x and y satisfying

$$ax + by = \gcd(a, b).$$

For example, $\gcd(12, 15) = 3$. Obviously, $15 - 12 = 3$, so with $a = 12$ and $b = 15$, we have $x = -1$ and $y = 1$. How do you compute these systematically and quickly?

Python

```
def extended_gcd(a, b):
    """

    EXAMPLES:
        >>> extended_gcd(12,15)
        (-1, 1)

    """
    if a%b == 0:
        return (0, 1)
    else:
        (x, y) = extended_gcd(b, a%b)
        return (y, x-y*int(a/b))
```

Python

```
def extended_gcd(a, b):
    """

    EXAMPLES:
        >>> extended_gcd(12,15)
        (-1, 1, 3)

    """
    x = 0
    lastx = 1
    y = 1
    lasty = 0
```

```
    while b <> 0:
        quotient = int(a/b)
        temp = b
        b = a%b
        a = temp
        temp = x
        x = lastx - quotient*x
        lastx = temp
        temp = y
        y  = lasty - quotient*y
        lasty = temp
    return (lastx, lasty, a)
```

Let us analyze the complexity of the second one. How many steps does this take in the worst-case situation?

Suppose that $a > b$ and that a is an n-bit integer (i.e., $a \leq 2^n$). The first four statements are "initializations", which are done just one time. However, the nine statements inside the while loop are repeated over and over, as long as b (which gets re-assigned each step of the loop) stays strictly positive.

Some notation will help us understand the steps better. Call (a_0, b_0) the original values of (a, b). After the first step of the while loop, the values of a and b get re-assigned. Call these updated values (a_1, b_1). After the second step of the while loop, the values of a and b get re-assigned again. Call these updated values (a_2, b_2). Similarly, after the k-th step, denote the updated values of (a, b), by (a_k, b_k). After the first step, $(a_0, b_0) = (a, b)$ is replaced by $(a_1, b_1) = (b, a \pmod{b})$. Note that $b > a/2$ implies $a \pmod{b} < a/2$, therefore we must have either $0 \leq a_1 \leq a_0/2$ or $0 \leq b_1 \leq a_0/2$ (or both). If we repeat this while loop step again, then we see that $0 \leq a_2 \leq a_0/2$ *and* $0 \leq b_2 \leq a_0/2$. Every 2 steps of the while loop, we decrease the value of b by a factor of 2. Therefore, this algorithm has complexity $T(n)$ where

$$T(n) \leq 4 + 18n = O(n).$$

Such an algorithm is called a *linear time* algorithm, since it complexity is bounded by a polynomial in n of degree 1.

> Excellence in any department can be attained only by the labor of a lifetime; it is not to be purchased at a lesser price.
> - *Samuel Johnson (1709-1784)*

7 Keywords and reserved terms in Python

Three basic types of Python statements are

- conditionals (such as an "if-then" statement),
- assignments, and
- iteration (such as a `for` or `while` loop).

Python has set aside many commands to help you create such statements. Python also protects you from accidentally over-writing these commands by "reserving" these commands.

When you make an assignment in Python, such as `a = 1`, you add the *name* (or "identifier" or "variable") `a` to the Python *namespace*. You can think of a namespace as a mapping from identifiers (i.e., a variable name such as `a`) to Python objects (e.g., an integer such as `1`). A name can be

- "local" (such as `a` in `a = 1`),
- "global" (such as the complex constant `j` representing $\sqrt{-1}$),
- "built-in" (such as `abs`, the absolute value function), or
- "reserved", or a "keyword" (such as `and` - see the table below).

The terms below are reserved and cannot be re-assigned. For example, trying to set `and` equal to `1` will result in a syntax error:

Python

```
>>> and = 1
  File "<stdin>", line 1
    and = 1
      ^
SyntaxError: invalid syntax
```

Also, `None` cannot be re-assigned, though it is not considered a keyword. Note: the Boolean values `True` and `False` are not keywords and in fact can be re-assigned (though you probably should not do so).

Keyword	*meaning*
`and`	boolean operator
`as`	used with `import` and `with`
`assert`	used for debugging
`break`	used in a `for/while` loop
`class`	creates a class
`continue`	used in `for/while` loops
`def`	defines a function or method
`del`	deletes a reference to a object instance
`elif`	used in `if ... then` statements
`else`	used in `if ... then` statements
`except`	used in `if ... then` statements
`exec`	executes a system command
`finally`	used in `if ... then` statements
`for`	used in a `for` loop
`from`	used in a `for` loop
`global`	this is a (constant) data type
`if`	used in `if ... then` statements
`import`	loads a file of data or Python commands
`in`	boolean operator on a set
`is`	boolean operator
`lambda`	defined a simple "one-liner" function
`not`	boolean operator
`or`	boolean operator
`pass`	allows and `if-then-elif` statement to skip a case
`print`	*duh:-)*
`raise`	used for error messages
`return`	output of a function
`try`	allows you to test for an error
`while`	used in a `while` loop
`with`	used for ???
`yield`	used for iterators and generators

The names in the table above are reserved for your protection. Even though type names such as `int`, `float`, `str`, are not reserved variables that does not mean you should reuse them.

Also, you cannot use operators (for example, `-`, `+`, \, or `^`) in a variable assignment. For example, `my-variable = 1` is illegal.

The `keyword` module:

Python

```
>>> import keyword
>>> keyword.kwlist()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not callable
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print',
'raise',
'return', 'try', 'while', 'with', 'yield']
>>>
```

7.1 Examples

`and`:

Python

```
>>> 0==1
False
>>> 0==1 and (1+1 == 2)
False
>>> 0+1==1 and (13%4 == 1)
True
```

Here `n%m` means "the remainder of n modulo m", where m and n are integers and $m \neq 0$.

`as`:

Python

```
>>> import numpy as np
```

The `as` keyword is used in `import` statements. The `import` statement adds new commands to Python whcih were not loaded by default. Not loading "espoteric" commands into Python has some advantages, such as making various aspects of Python more efficient.

I probably don't need to tell you that, in spite of what the `xkcd` cartoon Figure 1 says, `import antigravity` will probably not make you fly!

`break`

An example of `break` will appear after the `for` loop examples below.

A `class` examples ("borrowed" from Kirby Urber [U], a Python +mathematics educator from Portland Oregon):

`class`:

Python

```
thesuits = ['Hearts','Diamonds','Clubs','Spades']
theranks = ['Ace'] + [str(v) for v in range(2,11)] + ['Jack','Queen','King']
rank_values = list(zip(theranks, range(1,14)))

class Card:
    """
    This class models a card from a standard deck of cards.
    thesuits, theranks, rank_values are local constants

    From an email of kirby urner <kirby.urner@gmail.com>
    to edu-sig@python.org on Sun, Nov 1, 2009.

    """
    def __init__(self, suit, rank_value ):
        self.suit = suit
        self.rank = rank_value[0]
        self.value = rank_value[1]
    def __lt__(self, other):
        if self.value < other.value:
            return True
        else:
            return False
    def __gt__(self, other):
        if self.value > other.value:
            return True
        else:
           return False
    def __eq__(self, other):
        if self.value == other.value:
            return True
        else:
            return False
    def __repr__(self):
        return "Card(%s, %s)"%(self.suit, (self.rank, self.value))
    def __str__(self):
        return "%s of %s"%(self.rank, self.suit)
```

Once read into Python, here is an example of its usage.

Python

```
>>> c1 = Card("Hearts", "Ace")
>>> c2 = Card("Spades", "King")
```

```
>>> c1<c2
True
>>> c1; c2
Card(Hearts, ('A', 'c'))
Card(Spades, ('K', 'i'))
>>> print c1; print c2
A of Hearts
K of Spades
```

`def`:

Python

```
>>> def fcn(x):
...   return x**2
...
>>> fcn(10)
100
```

The next simple example gives an interactive example requiring user input.

Python

```
>>> def hello():
...          name = raw_input('What is your name?\n')
...          print "Hello World! My name is %s"%name
...
>>> hello()
What is your name?
David
Hello World! My name is David
>>>
```

The examples above of `def` and `class` bring up an issue of how variables are recalled in Python. This is briefly discussed in the next subsection.

The `for` loop construction is useful if you have a static (unchanging) list you want to run through. The most common list used in `for` loops uses the `range` construction. The Python expression

`range(a,b)`

returns the list of integers a, $a+1$, $\ldots$, $b-1$. The Python expression

`range(b)`

returns the list of integers 0, 1, ..., $b-1$.

`for/while`:

Python

```
>>> for n in range(10,20):
...         if not(n%4 == 2):
...             print n
...
11
12
13
15
16
17
19
>>> [n for n in range(10,20) if not(n%4==2)]
[11, 12, 13, 15, 16, 17, 19]
```

The second example above is an illustration of *list comprehension*. List comprehension is a syntax for list construction which mimics how a mathematician might define a set.

The `break` command is used to break out of a `for` loop.

`break`:

Python

```
>>> for i in range(10):
...  if i>5:
...   break
...  else:
...   print i
...
0
1
2
3
4
5
```

`for/while`:

Python

```
>>> L = range(10)
>>> counter = 1
>>> while 7 in L:
...     if counter in L:
...         L.remove(counter)
...         print L
...         counter = counter + 1
...
[0, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 3, 4, 5, 6, 7, 8, 9]
[0, 4, 5, 6, 7, 8, 9]
[0, 5, 6, 7, 8, 9]
[0, 6, 7, 8, 9]
[0, 7, 8, 9]
[0, 8, 9]
```

`if/elif`:

Python

```
>>> def f(x):
...       if x>2 and x<5:
...             return x
...       elif x>5 and x<8:
...             return 100+x
...       else:
...             return 1000+x
...
>>> f(0)
1000
>>> f(1)
1001
>>> f(3)
3
>>> f(5)
1005
>>> f(6)
106
```

When using `while` be very careful that you actually do have a terminating condition in the loop!

`lambda`:

Python

```
>>> f = lambda x,y: x+y
>>> f(1,2)
3
```

The command `lambda` allows you to create a small simple function which does not have any local variables except those used to define the function.

`raise`:

Python

```
>>> def modulo10(n):
...     if type(n)<>int:
...          raise TypeError, 'Input must be an integer!'
...     return n%10
...
>>> modulo10(2009)
9
>>> modulo10(2009.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in modulo10
TypeError: Input must be an integer!
```

`yield`:

Python

```
>>> def pi_series():
...     sum = 0
...     i = 1.0; j = 1
...     while(1):
...             sum = sum + j/i
...             yield 4*sum
...             i = i + 2; j = j * -1
...
>>> pi_approx = pi_series()
>>> pi_approx.next()
4.0
>>> pi_approx.next()
2.666666666666667
>>> pi_approx.next()
3.4666666666666668
>>> pi_approx.next()
2.8952380952380956
>>> pi_approx.next()
3.3396825396825403
>>> pi_approx.next()
2.9760461760461765
>>> pi_approx.next()
3.2837384837384844
>>> pi_approx.next()
3.0170718170718178
```

This function generates a series of approximations to $\pi = 3.14159265\ldots$. For more examples, see for example the article [PG].

7.2 Basics on scopes and namespaces

We talked about namespaces in §7. Recall a namespace is a mapping from variable names to objects. For example, `a = 123` places the name `a` in the namespace and "maps it" to the integer object `123` of type `int`.

The namespace containing the built-in names, such as the absolute value function `abs`, is created when the Python interpreter starts up, and is never deleted.

The local namespace for a function is created when the function is called. For example, the following commands show that the name `b` is "local" to the function `f`.

Python

```
>>> a = 1
>>> def f():
...         a = 2
...         b = 3
...         print a,b
...
>>> f()
2 3
>>> a
1
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

In other words, the value of `a` assigned in the command `a = 1` is not changed by calling the function `f`. The assignment `a = 2` inside the function definition cannot be accessed outside the function. This is an example of a "scoping rule" – a process the Python interpreter follows to try to determine the value of a variable name assignment.

Scoping rules for Python classes are similar to functions. That is to say, variable names declared inside a class are local to that class. The Python tutorial has more on the subtle issues of scoping rules and namespaces.

7.3 Lists and dictionaries

These are similar data types in some ways, so we clump them together into one section.

7.4 Lists

Lists are one of the most important data types. Lists are "mutable" in the sense that you can change their values (as is illustrated below by the command `B[0] = 1`). Python has a lot of functions for manipulating and computing with lists.

Python

```
sage: A = [2, 3, 5, 7, 11]
sage: B = A
sage: C = copy(A)
sage: B[0] = 1
sage: A; B; C
[1, 3, 5, 7, 11]
[1, 3, 5, 7, 11]
[2, 3, 5, 7, 11]
```

Note `C`, the copy, was left alone in the reassignment.

Python

```
sage: A = [2, 3, [5, 7], 11, 13]
sage: B = A
sage: C = copy(A)
sage: C[2] = 1
sage: A; B; C
[2, 3, [5, 7], 11, 13]
[2, 3, [5, 7], 11, 13]
[2, 3, 1, 11, 13]
```

Here again, `C`, the copy, was the only odd man out in the reassignment.

An analogy: `A` is a list of houses on a block, represented by their street addresses. `B` is a copy of these addresses. `C` is a snapshot of the houses. If you change one of the addresses on the block `B`, you change that in `A` but not `C`. If you use `GIMP` or `Photoshop` to modify one of the houses depicted in `C`, you of course do not change what is actually on the block in `A` or `B`. Does this seem like a reasonable analogy?

It is not a correct analogy! The example below suggests a deeper behaviour, indicating that this analogy is wrong!

Python

```
sage: A = [2, 3, [5, 7], 11, 13]
sage: B = A
sage: C = copy(A)
sage: C[2][1] = 1
sage: A; B; C
[2, 3, [5, 1], 11, 13]
[2, 3, [5, 1], 11, 13]
[2, 3, [5, 1], 11, 13]
```

Here C's reassignment changes everything!

This indicates that the "snapshot" analogy is missing the key facts. In fact, the `copy C` of a list `A` is not really a snapshop but a recording of some memory address information which points to data at those locations in `A`. If you change the addresses in `C`, you will not change what is actually stored in `A`. Accessing a sublist of a list is looking at the data stored at the location represented by that entry in the list. Therefore, changing a sublist entry of the copy changes the entries of the originals too. If you represent each house as its list of family members, so `A` is a list of lists, then the copy command will accurately copy family member, and so if you change elements in one copy of the sublist, you change those elements in all sublists.

7.4.1 Dictionaries

Dictionaries, like lists, are mutable. A Python *dictionary* is an unordered set of `key:value` pairs, where the keys are unique. A pair of braces {} creates an empty dictionary; placing a comma-separated list of `key:value` pairs initializes the dictionary.

Python

```
>>> d = {1:"a", 2:"b"}
>>> d
{1: 'a', 2: 'b'}
>>> print d
{1: 'a', 2: 'b'}
>>> d[1]
'a'
>>> d[1] = 3
>>> d
{1: 3, 2: 'b'}
```

```
>>> d.keys()
[1, 2]
>>> d.values()
[3, 'b']
```

One difference with lists is that dictionaries do not have an ordering. They are indexed by the "keys" (as opposed to the integers 0, 1, ..., $m-1$, for a list of length m). In fact, tere is not much difference between the dictionary `d1` and the list `d2` below.

Python

```
>>> d1 = {0:"a", 1:"b", 2:"c"}
>>> d2 = ["a", "b", "c"]
```

Dictionaries can be much more useful than lists. For example, suppose you wanted to store all your friends' cell-phone numbers in a file. You could create a list of pairs, `(name of friend, phone number)`, but once this list becomes long enough searching this list for a specific phone number will get time-consuming. Better would be if you could index the list by your friend's name. This is precisely what a dictionary does.

The following examples illustrate how to create a dictionary in Sage, get access to entries, get a list of the keys and values, etc.

Sage

```
sage: d = {'sage':'math', 1:[1,2,3]}; d
{1: [1, 2, 3], 'sage': 'math'}
sage: d['sage']
'math'
sage: d[1]
[1, 2, 3]
sage: d.keys()
[1, 'sage']
sage: d.values()
[[1, 2, 3], 'math']
sage: d.has_key('sage')
True
sage: 'sage' in d
True
```

You can delete entries from the dictionary using the `del` keyword.

Sage

```
sage: del d[1]
sage: d
{'sage': 'math'}
```

You can also create a dictionary by typing `dict(v)` where `v` is a list of pairs:

Sage

```
sage: dict( [(1, [1,2,3]), ('sage', 'math')])
{1: [1, 2, 3], 'sage': 'math'}
sage: dict( [(x, x^2) for x in [1..5]] )
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

You can also make a dictionary from a "generator expression" (we have not discussed these yet).

Sage

```
sage: dict( (x, x^2) for x in [1..5] )
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

In truth, a dictionary is very much like a list inside the Python interpreter on your computer. However, dictionaries are "hashed" objects which allow for fast searching.

Warning: Dictionary keys must be hashable The keys `k` of a dictionary must be *hashable*, which means that calling `hash(k)` doesn't result in an error. Some Python objects are hashable and some are not. Usually objects that can't be changed are hashable, whereas objects that can be changed are not hashable, since the hash of the object would change, which would totally devastate most algorithms that use hashes. In particular, numbers and strings are hashable, as are tuples of hashable objects, but lists are never hashable.

We hash the string 'sage', which works since one cannot change strings.

Sage

```
sage: hash('sage')
-596024308
```

The list `v = [1,2]` is not hashable, since `v` can be changed by deleting, appending, or modifying an entry. Because `[1,2]` is not hashable it can't be used as a key for a dictionary.

Sage

```
sage: hash([1,2])
Traceback (most recent call last):
...
TypeError: list objects are unhashable
sage: d = {[1,2]: 5}
Traceback (most recent call last):
...
TypeError: list objects are unhashable
\end{verbatim}
However the tuple {\tt (1,2)} is hashable and can hence be used as a
dictionary key.
\begin{verbatim}
sage: hash( (1,2) )
1299869600
sage: d = {(1,2): 5}
```

Hashing goes well beyong the subject of this course, but see the course [DL] for more details if you are interested.

7.5 Tuples, strings

Both of these are non-mutable, which makes them faster to store and manipulate in Python.

Lists and dictionaries are useful, but they are "mutable" which means their values can be changed. There are circumstances where you do not want the user to be allowed to change values.

For example, a linear error-correcting code is simply a finite dimensional vector space over a finite field with a fixed basis. Since the basis is fixed, we may want to use tuples instead of lists for them, as tuples are immutable objects.

Tuples, like lists, can be "added": the `+` symbol represents concatenation. Also, like lists, tuples can be multiplied by a natural number for iterated concatenation. However, as stated above, an entry (or "item") in a tuple cannot be re-assigned.

Python

```
>>> a = (1,2,3)
>>> b = (0,)*3
>>> b
```

```
(0, 0, 0)
>>> a+b
(1, 2, 3, 0, 0, 0)
>>> a[0]
1
>>> a[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Strings are similar to tuples in many ways.

Python

```
>>> a = "123"
>>> b = "hello world! "
>>> a[1]
'2'
>>> b*2
'hello world! hello world! '
>>> b[0] = "H"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> b+a
'hello world! 123'
>>> a+b
'123hello world! '
```

Note that addition is "non-commutative": `a+b` $\neq$ `b+a`.

There are lots of very useful string-manipulation functions in Python. For example, you can replace any substring using the `replace` method. You can find the location of (the first occurrence of) any substring using the `index` method.

Python

```
>>> a = "123"
>>> b = "hello world! "
>>> b.replace("h","H")
'Hello world! '
>>> b
'hello world! '
>>> b.index("o")
4
>>> b.index("w")
6
>>> b.replace("! ","")
```

```
'hello world'
>>> b.replace("! ","").capitalize().replace("w","W")
'Hello World'
```

Since strings are very important data objects, they are covered much more extensively in other places. Please see any textbook on Python for more examples.

7.5.1 Sets

Python has a set datatype, which behaves much like the keys of a dictionary. A *set* is an unordered collection of unique hashable objects. Sets are incredibly useful when you want to quickly eliminate duplicates, do set theoretic operations (union, intersection, etc.), and tell whether or not an objects belongs to some collection.

You create sets *from the other Python data structures* such as lists, tuples, and strings. For example:

Python

```
>>> set( (1,2,1,5,1,1) )
set([1, 2, 5])
>>> a = set('abracadabra'); b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> b
set(['a', 'c', 'z', 'm', 'l'])
```

There are also many handy operations on sets.

Python

```
>>> a - b    # letters in a but not in b
set(['r', 'b', 'd'])
>>> a | b    # letters in either a or b
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
>>> a & b    # letters in both a and b
set(['a', 'c'])
```

If you have a big list `v` and want to repeatedly check whether various elements `x` are in `v`, you *could* write `x in v`. This would work. Unfortunately, it would be really slow, since every command `x in v` requires *linearly* searching through for x. A much better option is to create `w = set(v)` and type `x in w`, which is very fast. We use Sage's time function to check this.

Sage

```
sage: v = range(10^6)
sage: time 10^5 in v
True
CPU time: 0.16 s,  Wall time: 0.18 s
sage: time w = set(v)
CPU time: 0.12 s,  Wall time: 0.12 s
sage: time 10^5 in w
True
CPU time: 0.00 s,  Wall time: 0.00 s
```

You see searching a list of length 1 million takes some time, but searching a (hashable) set is done essentially instantly.

> *The Zen of Python, II*
> In the face of ambiguity, refuse the temptation to guess.
> There should be one - and preferably only one - obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than right now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea - let's do more of those!
>
> - *Tim Peters* (Long time Pythoneer)

8 Iterations and recursion

Neither of these are data types but they are closely connected with some useful Python constructions. Also, they "codify" very common constructions in mathematics.

8.1 Repeated squaring algorithm

The basic idea is very simple. For input you have a number x and an integer $n > 0$. Assume x is fixed, so we are really only interested in an efficient algorithm as a function of n.

We start with an example.

Example 5. Compute x^{13}.

First compute x (0 steps), x^4 (2 steps, namely $x^2 = x \cdot x$ and $x^4 = x^2 \cdot x^2$), and x^8 (2 steps, namely x^4 and $x^8 = x^4 \cdot x^4$). Now (3 more steps)

$$x^{13} = x \cdot x \cdot x^4 \cdot x^8.$$

In general, we can compute x^n in about $O(\log n)$ steps. Here is an implementation in Python.

Python

```
def power(x,n):
    ""
    INPUT:
        x - a number
        n - an integer > 0

    OUTPUT:
        x^n

    EXAMPLES:
        >>> power(3,13)
        1594323
        >>> 3**(13)
        1594323
    ""
    if n == 1:
        return x
    if n%2 == 0:
       return power(x, int(n/2))**2
    if n%2 == 1:
       return x*power(x, int((n-1)/2))**2
```

Very efficient! You can see that we are, at each step, roughly speaking, dividing the exponent by 2. So the algorithm roughly has worst-case complexity $2\log_2(n)$.

For more variations on this idea, see for example `http://en.wikipedia.org/wiki/Exponentiation_by_squaring`.

8.2 The Tower of Hanoi

The "classic" Tower of Hanoi consists of $p = 3$ posts or pegs, and a number d of disks of different sizes which can slide onto any post. The puzzle starts

with the disks in a neat stack in ascending order of size on one post, the smallest at the top, thus making a conical shape[4] This can be generalized to any number of pegs greater than 2, if desired.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the posts and sliding it onto another one, on top of the other disks that may already be present on that post.
- No disk may be placed on top of a smaller disk.

The *Tower of Hanoi Problem* is the problem of designing a general algorithm which describes how to move d discs from one post to another. We may also ask how many steps are needed for the shorted possible solution. We many also ask for an algorithm to compute which disc should be moved at a given step in a shortest possible algorithm (without demanding to know which post to place it on).

The following procedure demonstrates a recursive approach to solving the classic 3-post problem.

- label the pegs A, B, C (we may want to relabel these to affect the recursive procedure)
- let d be the total number of discs, and label the discs from 1 (smallest) to d (largest).

To move d discs from peg A to peg C:

(1) move $d-1$ discs from A to B. This leaves disc d alone on peg A.

(2) move disc d from A to C

(3) move $d-1$ discs from B to C so they sit on disc d.

[4]For example, see the Wikipedia page `http://en.wikipedia.org/wiki/Tower_of_Hanoi` for more details and references.

The above is a recursive algorithm: to carry out steps (1) and (3), apply the same algorithm again for $d-1$ discs. The entire procedure is a finite number of steps, since at some point the algorithm will be required for $d=1$. This step, moving a single disc from one peg to another, is trivial.

Here is Python code implementing this algorithm.

Python

```
def Hanoi(n, A, C, B):
    if n != 0:
        Hanoi(n - 1, A, B, C)
        print 'Move the plate from', A, 'to', C
        Hanoi(n - 1, B, C, A)
```

There are many other ways to approach this problem.

Exercise 8.1. *Let T_n denote the number of step it takes to solve the 3-post Tower of Hanoi, if you make the best move possibly each time.*

- *Explain why $T_n = 2T_{n-1} + 1$ using only the definition of the Tower of Hanoi puzzle.*
- *Use this and mathematical induction to show $T_n = 2^n - 1$.*

If there are m posts and d discs, we label the posts 0, 1, ..., $m-1$ in some fixed manner, and we label the discs 1, 2, ..., d in order of decreasing radius. It is hopefully self-evident that you can uniquely represent a given "state" of the puzzle by a d-tuple of the form $(p_1, p_2, \ldots, p_d)$, where p_i is the post number that disc i is on (where $0 \leq p_i \leq m-1$, for all i). Indeed, since the discs have a fixed ordering (smallest to biggest, top to bottom) on each post, this d-tuple uniquely specifies a puzzle state. In particular, there are m^d different possible puzzle states.

Define a graph Γ to have vertices consisting of all m^d such puzzle states. These vertices can be represented by an element in the Cartesian product $V = (\mathbb{Z}/m\mathbb{Z})^d$. We connect two vertices v, w in V by an edge if and ony if it is possible to go from the state represented by v to the state represented by w using a legal disc move. (in this case, we say that v is a *neighbor* of w.) It is not hard to see that the only way two elements of $V = (\mathbb{Z}/m\mathbb{Z})^d$ can be connected by an edge is if the d-tuple v is the same as the d-tuple w in every coordinate except one.

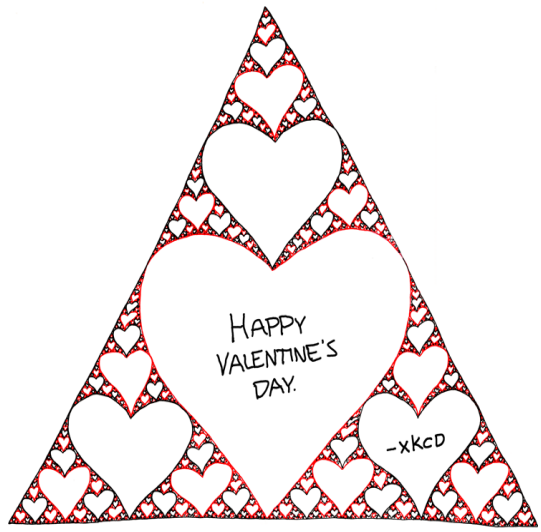


Figure 9: **Sierpinski Valentine .**
`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

Example 6. For instance, if $m = 3$ and $d = 2$ then $(2, 0)$ simply means that the biggest disc is on post 2 and the other (smaller) disc is on post 0.

Here is one possible solution in this case. Suppose we start with $(2, 2)$ (both discs are on post 2).

- First move: place the smaller disc to post 1 (this gives us $(2, 1)$).
- Second move: place the bigger disc on post 0 (giving us $(0, 1)$).
- Third and final move: place the smaller disc on post 0 (this gives us $(0, 0)$).

See the "bottom side" of the triangle in Figure 10, (made using a graph-theoretic construction implemented by Robert Beezer in Sage).

In fact, the above `Hanoi` program gives this output:

Python

```
>>> Hanoi(2, "2", "0", "1")
Move the plate from 2 to 1
Move the plate from 2 to 0
Move the plate from 1 to 0
```

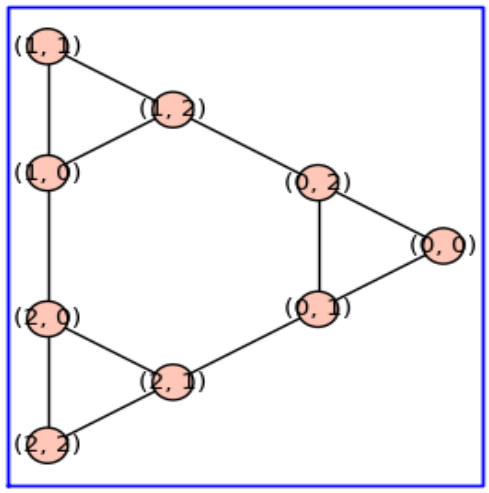


Figure 10: Tower of Hanoi graph for 3 posts and 2 discs.

Example 7. For instance, if $m = d = 3$ then $(2, 2, 2)$ simply means that all three discs are on the same post (of course, the smallest one being on top), namely on the post labeled as 2. See Figure 11, which used Sage as in the example above, for the possible solutions to this puzzle.

See Figure 12 for the example of the unlabeled graph representing the states of the Tower of Hanoi puzzle with 3 posts and 6 discs. Notice the similarity to the Sierpinski triangle (see for example, `http://en.wikipedia.org/wiki/Sierpinski_triangle`)!

See Figure 13 for the example of the unlabeled graph representing the states of the Tower of Hanoi puzzle with 5 posts and 3 discs.

8.3 Fibonacci numbers

The Fibonacci sequence is named after Leonardo of Pisa, known as Fibonacci, who mentioned them in a book he wrote in the 1200's. Apparently they were known to Indian mathematicians centuries before.

He considers the growth of a rabbit population,where

- In the 0-th month, there is one pair of rabbits.
- In the first month, the first pair gives birth to another pair.

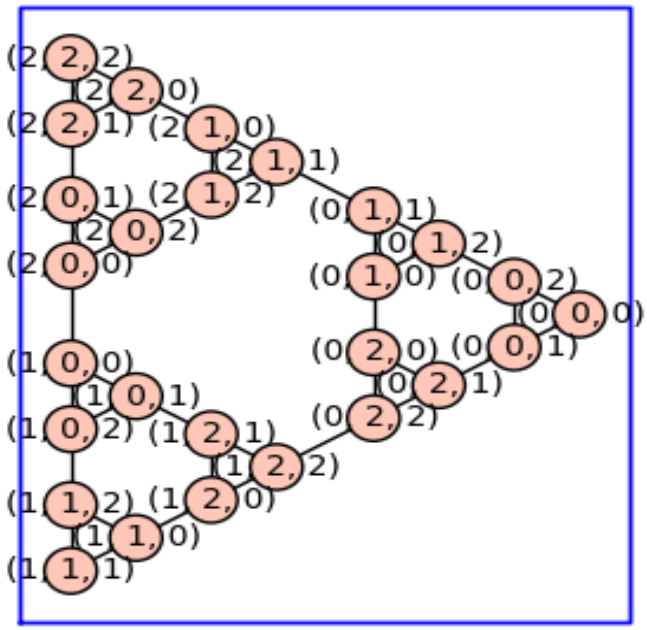


Figure 11: Tower of Hanoi graph for 3 posts and 3 discs.

- In the second month, both pairs of rabbits have another pair, and the first pair dies.
- In general, each pair of rabbits has 2 pairs in its lifetime, and dies.

Let the population at month n be f_n. At this time, only rabbits who were alive at month $n-2$ are fertile and produce offspring, so f_{n-2} pairs are added to the current population of f_{n-1}. Thus the total is $f_n = f_{n-1} + f_{n-2}$. The recursion equation

$$f_n = f_{n-1} + f_{n-2}, \quad n > 1, \quad f_1 = 1, \; f_0 = 0,$$

defined the *Fibonacci sequence*. The terms of the sequence are *Fibonacci numbers*. (See also Example 37 below.)

8.3.1 The recursive algorithm

There is an exponential time algorithm to compute the Fibonacci numbers.

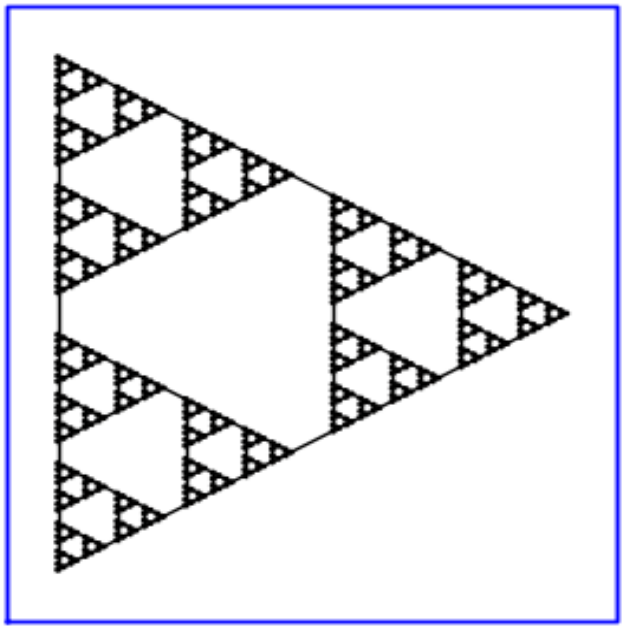


Figure 12: Unlabeled Tower of Hanoi graph for 3 posts and 6 discs.

Python

```
def my_fibonacci(n):
    """
    This is really really slow.
    """
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return my_fibonacci(n-1)+my_fibonacci(n-2)
```

How many steps does `my_fibonacci(n)` take?

In fact, the "complexity" of this algorithm to compute f_n is about equal to f_n (which is about ϕ^n, where $\phi = \frac{1+\sqrt{5}+1}{2}$ is the golden ratio.). The reason why is that the number of steps can be computed as being the number of "f_1"s and "f_2"s which occur in the ultimate decomposition of f_n obtained by re-iterating the recurrence $f_n = f_{n-1} + f_{n-2}$. Since $f_1 = 1$ and $f_2 = 1$,

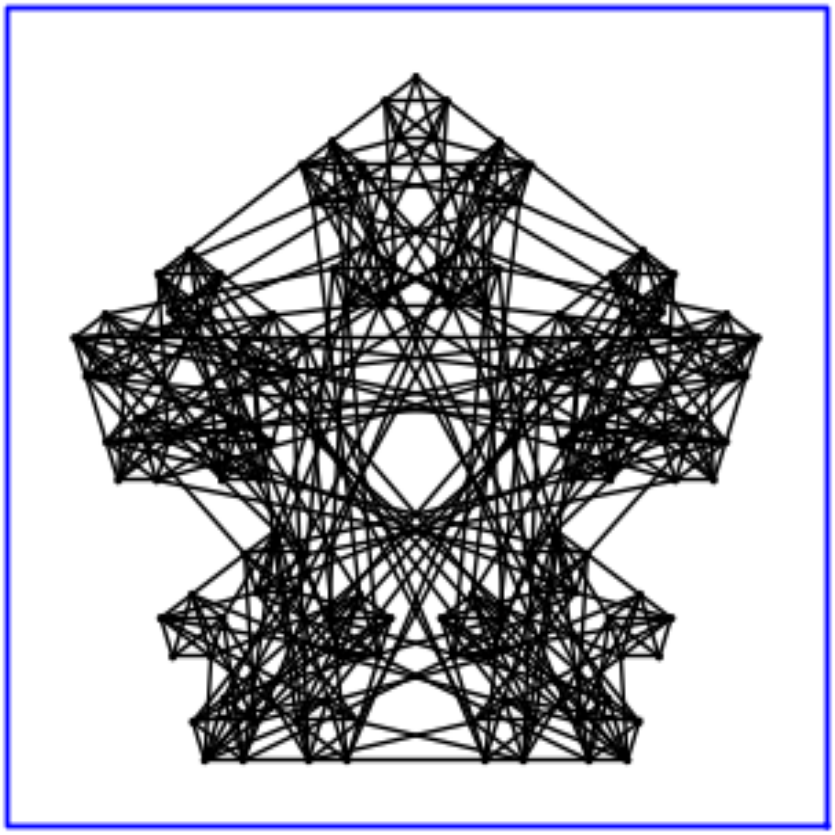


Figure 13: Unlabeled Tower of Hanoi graph for 5 posts and 3 discs.

this number is equal to simply f_n itself.

8.3.2 The matrix-theoretic algorithm

There is a sublinear algorithm to replace this exponential algorithm.

Consider the matrix

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Lemma 8. For each $n > 0$, we have $F^n = \begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix}$.

proof: The case $n = 1$ follows from the definition. Assume that $F^k = \begin{pmatrix} f_{k-1} & f_k \\ f_k & f_{k+1} \end{pmatrix}$, for some $k > 1$. We have

$$F^{k+1} = \begin{pmatrix} f_{k-1} & f_k \\ f_k & f_{k+1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} f_{k-1} & f_{k-1} + f_k \\ f_{k+1} & f_k + f_{k+1} \end{pmatrix} = \begin{pmatrix} f_{k-1} & f_{k+1} \\ f_{k+1} & f_{k+2} \end{pmatrix}.$$

The claim follows by induction. □

We can use the repeated squaring algorithm (§8.1) to compute F^n. Since this has complexity, $O(\log n)$, this algorithm for computing f_n has complexity $O(\log n)$.

8.3.3 Exercises

The sequence of Lucas numbers $\{L_n\}$ begins:

$$2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, \ldots,$$

and in general are defined by $L_n = L_{n-1} + L_{n-2}$, for $n > 1$ ($L_0 = 2$, $L_1 = 1$). This sequence is named after the mathematician Francois Édouard Anatole Lucas (1842-1891), A *Lucas prime* is a Lucas number that is prime. The first few Lucas primes are

$$2, 3, 7, 11, 29, 47, \ldots.$$

It is known that L_n is prime implies n is prime, except for the cases $n = 0$, 4, 8, 16.. The converse is false, however. (I've read the paper at one point many years ago but have forgotten the details now.)

Exercise 8.2. *Modify one of the Fibonacci programs above and create programs to generate the Lucas numbers. Remember to comment your program and put it in the format given in §9.4.*

8.4 Collatz conjecture

The Collatz conjecture is an unsolved conjecture in mathematics, named after Lothar Collatz. The conjecture is also known as the $3n + 1$ conjecture,

or as the Syracuse problem, among others. Start with any integer n greater than 1. If n is even, we halve it $n/2$, else we "triple it plus one" $(3n + 1)$. According to the conjecture, for all positive numbers this process eventually converges to 1. For details, see for example `http://en.wikipedia.org/wiki/Collatz_conjecture`.

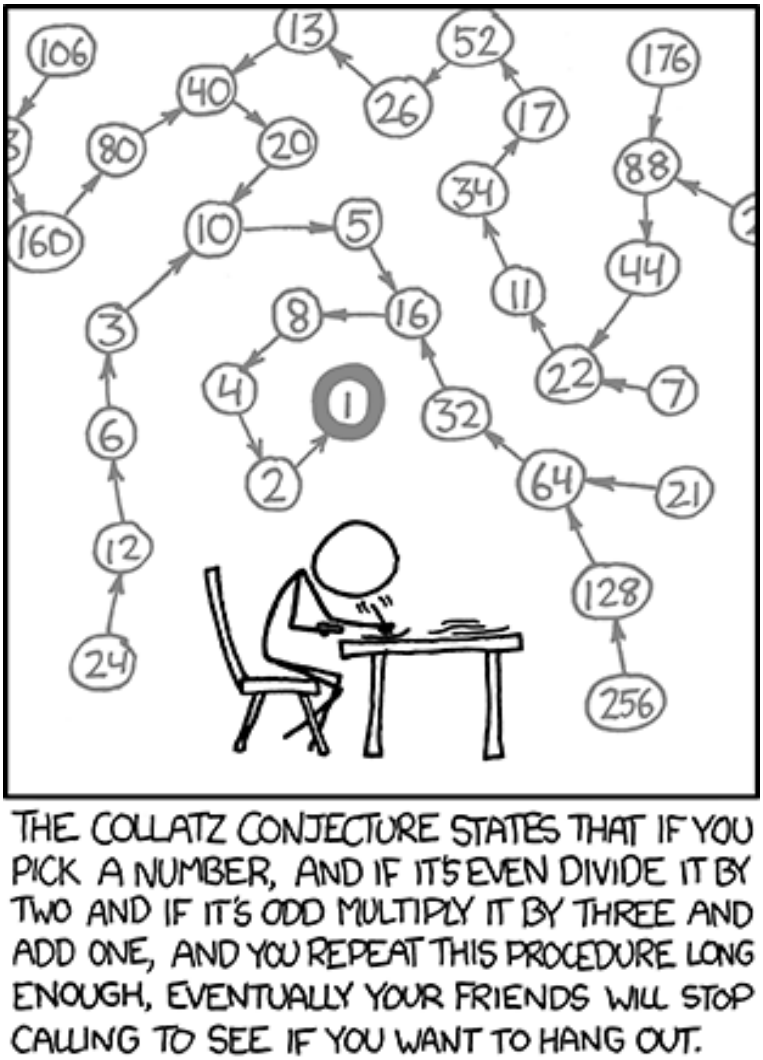


Figure 14: **The Collatz Conjecture.**
`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

Exercise 8.3. *Write a Python program which tests the Collatz conjecture for all numbers* $n < 100$. *You program should have input* n *and output the number of steps the program takes to "converge" to* 1.

9 Programming lessons

Try this in a Python interactive interpreter:
`>>> import this`

Programming is hard. You cannot fool a computer with faulty logic. You cannot hide missing details hoping your teacher is too tired of grading to notice. This time your teacher is the computer and it never tires. Ever. If your program does not work, you know it because your computer returns something unexpected.

An important aspect of programming is the ability to "abstract" and "modularize" your programs. By "abstract', I mean to determine what the essential aspects of your program are and possibly to see a pattern in something you or someone else has already done. This helps you avoid "reinventing the wheel." By "modularize", i.e., "decomposibility", I mean you should see what elements in your program are general and transportable to other programs then then separating those out as separate entities and writing them as separate subprograms[5].

Another part (very important, in my opinion) of programming is style conventions. **Please** read and **follow** the style conventions of Python programming described in `http://www.python.org/dev/peps/pep-0008/` (for the actual Python code) and `http://www.python.org/dev/peps/pep-0257/` (for the comments and docstrings).

9.1 Style

In general, you should read the *Style Guide for* Python *Code* `http://www.python.org/dev/peps/pep-0008/`, but here are some starter suggestions.

Whitespace usage:

- 4 spaces per indentation level.
- No tabs. In particular, never mix tabs and spaces.

[5]Note: In Python, the word "module" has a specific technical meaning which is separate (though closely related) to what I am talking about here.

- One blank line between functions.
- Two blank lines between classes.
- Add a space after "," in dicts, lists, tuples, and argument lists, and after ":" in dicts, but not before.
- Put spaces around assignments and comparisons (except in argument lists).
- No spaces just inside parentheses or just before argument lists.

Naming conventions:

- joined_lower for functions, methods, attributes.
- joined_lower or ALL_CAPS for constants (local, resp., global).
- StudlyCaps for classes.
- camelCase only to conform to pre-existing conventions.
- Attributes: interface, _internal, __private

9.2 Programming defensively

"Program defensively" (see MIT lecture 3 [GG]):

- If you write a program, expect your users to enter input other than what you want. For example, if you expect an integer input, assume they enter a float or string and anticipate that (check for input type, for example).
- Assume your program contains mistakes. Include enough tests to catch those mistakes before they catch you.
- Generally, assume people make mistakes (you the programmer, your users) and try to build in error-checking ingredients into your program. Spend time on type-checking and testing "corner cases" now so you don't waste time later.

- Add tests in the docstrings in several cases where you know the input and output. Add tests for the different types of options allowed for any optional keywords you have.

If it helps, think of how angry you will be at yourself if you write a poorly documented program which has a mistake (a "bug", as Grace Hopper phrased it[6] ; see also Figure 15 for a story behind this terminology) which you can't figure out. Trust me, someone else who wants to use your code and notices the bug, then tries reading your undocumented code to "debug" it will be even angrier. Please try to spend time and care and thought into carefully writing and commenting/documenting your code.

There is an article *Docstring Conventions*, `http://www.python.org/dev/peps/pep-0257/`, with helpful suggestions and conventions (see also `http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html`). Here are some starter suggestions.

Docstrings explain how to use code, and are for the users of your code. Explain the purpose of the function. Describe the parameters expected and the return values.

For example, see the docstring to the `inverse_image` function in Example 10.

Comments explain why your function does what it does. It is for the maintainers of your code (and, yes, you must always write code with the assumption that it will be maintained by someone else).

For example, `# !!! FIX: This is a hack` is a comment[7].

9.3 Debugging

> When you have eliminated the impossible, whatever remains, however improbable, must be the truth.
>
> *A. Conan Doyle*, **The Sign of Four**

[6]See `http://en.wikipedia.org/wiki/Grace_Hopper` for details on her interesting life.)

[7]By the way, a "hack", or "kludge", refers to a programming trick which does not follow expected style or method. Typically it involves a clever or quick fix to a computer programming problem which is perceived to be a clumsy solution.

There are several tools available for Python debugging. Presumably you can find them by "googling" but the simplest tools, in my opinion, are also the best tools:

- Use the `print` statement liberally to print out what you think a particular step in your program should produce.
- Use basic logic and read your code line-by-line to try to isolate the issue. Try to reduce the "search space" you need to test using `print` statements by isolating where you think the bug most likely will be.
- Read the Python error message (i.e., the "traceback"), if one is produced, and use it to further isolate the bug.
- Be systematic. Never search for the bug in your program by randomly selecting a line and checking that line, then randomly selecting another line ....
- Apply the "scientific method":
  - Study the available data (output of tests, `print` statements, and reading your program.
  - Think up a hypothesis consistent with all your data. (For example, you might hypothesize that the bug is in a certain section of your program.)
  - Design an experiment which tests and can possibly refute your hypothesis. Think about the expected result of your experiment.
  - If your hypothesis leads to the location of the bug, next move to fixing your bug. If not, then you should modify suitably your hypothesis or experiment, or both, and repeat the process.

If you use the Sage command line, there is a built-in debugger `pdb` which you can "turn on" if desired. For more on the `pdb` commands, see the Sage tutorial, `http://www.sagemath.org/doc/tutorial/interactive_shell.html`. For pure Python, see for example, the blog post [F] or the section of William Stein's mathematical computation course [St] on debugging. In fact, this is what William Stein says about using the `print` statement for debugging.

1. Put `print 0`, `print 1`, `print 2`, etc., at various points in your code. This will show you were something crashes or some other weird behavior happens. Sprinkle in more print statements until you narrow down exactly where the problem occurs.
2. Print the values of variables at key spots in your code.
3. Print other state information about Sage at key spots in your code, e.g., `cputime`, `walltime`, `get_memory_usage`, etc.

The main key to using the above is to think deductively and carefully about what you are doing, and hopefully isolate the problem. Also, with experience you'll recognize which problems are best tracked down using print statements, and which are not.

These suggestions can also be useful to simply tell when certain parts of your code are taking up more time than you expected (so-called "bottlenecks").

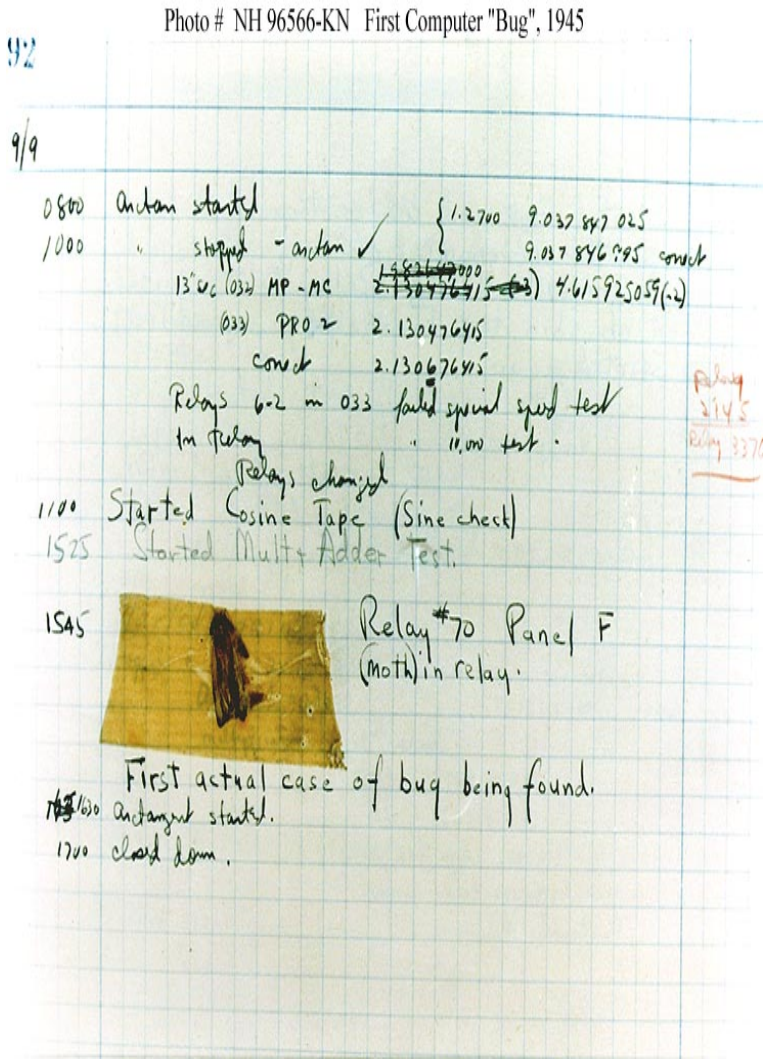


Figure 15: First computer "bug" (a moth jamming a relay switch). This was a page in the logbook of Grace Hopper describing a program running on the Mark II computer at Harvard University computing arc tangents, probably to be used for ballistic tables for WWII. (Incidentally, 1945 is a typo for 1947 according to some historians.)

Example 9. In the hope that it may help someone who has not every debugged anything before, here is a very simple example.

Suppose you are trying to write a program to multiply two matrices.

Python

```
def mat_mult(A, B):
    """
    Multiplies two 2x2 matrices in the usual way

     INPUT:
       A - the 1st 2x2 matrix
       B - the 2nd 2x2 matrix

    OUTPUT:
       the 2x2 matrix AB

    EXAMPLES:
        >>> my_function(1,2) # for a Python program
        <the output>

    AUTHOR(S):
        <your name>

    TODO:
        Implement Strassen's algorithm [1] since it
        uses 7 multiplications instaead of 8!

    REFERENCES:
        [1] http://en.wikipedia.org/wiki/Strassen_algorithm

    """
    a1 = A[0][0]
    b1 = A[0][1]
    c1 = A[1][0]
    d1 = A[1][1]
    a2 = B[0][0]
    b2 = B[0][1]
    c2 = B[1][0]
    d2 = B[1][1]
    a3 = a1*a2+b1*c2
    b3 = a1*b2+b1*d2
    c3 = c1*a2-d1*c2
    d3 = c1*b2+d1*d2
    return [[a3,b3],[c3,d3]]
```

This is actually wrong. In fact, if you read this into the Python interpreter and try an exampl, you get the following output.

Python

```
>>> A = [[1,2],[3,4]]; B = [[5,6],[7,8]]
>>> mat_mult(A, B)
[[19, 22], [-13, 50]]
```

This is clearly nonsense, since the product of matrices having positive entries must again be positive. Besides, an easy computation by hand tells us that

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}.$$

(I'm sure you see that in this extremely example there is an error in the computation of `c3`, but suppose for now you don't see that.)

To debug this, let us enter print statements in some key lines. In this example, lets see if the mistake occurs in the computation of `a3`, `b3`, `c3`, or `d3`.

Python

```
def mat_mult(A, B):
    """
    Multiplies two 2x2 matrices in the usual way

     INPUT:
       A - the 1st 2x2 matrix
       B - the 2nd 2x2 matrix

    OUTPUT:
       the 2x2 matrix AB

    EXAMPLES:
        >>> my_function(1,2) # for a Python program
        <the output>

    AUTHOR(S):
        <your name>

    TODO:
        Implement Strassen's algorithm [1] since it
        uses 7 multiplications instaead of 8!

    REFERENCES:
        [1] http://en.wikipedia.org/wiki/Strassen_algorithm

    """
    a1 = A[0][0]
    b1 = A[0][1]
    c1 = A[1][0]
    d1 = A[1][1]
    a2 = B[0][0]
    b2 = B[0][1]
    c2 = B[1][0]
    d2 = B[1][1]
    a3 = a1*a2+b1*c2
    print 'a3 = ', a3
    b3 = a1*b2+b1*d2
    print 'b3 = ',b3
    c3 = c1*a2-d1*c2
    print 'c3 = ',c3
```

```
    d3 = c1*b2+d1*d2
    print 'd3 =', d3
    return [[a3,b3],[c3,d3]]
```

Read this into Python again. The same input this time yields the following output.

Python

```
>>> A = [[1,2],[3,4]]; B = [[5,6],[7,8]]
>>> mat_mult(A, B)
a3 =  19
b3 =  22
c3 =  -13
d3 = 50
[[19, 22], [-13, 50]]
```

Now you see that the line computing `c3` has a bug. Opps - there is a `-` instead of a `+` there! We've located our bug. The correct program, with a correct example, is the following one.

Python

```
def mat_mult(A, B):
    """
    Multiplies two 2x2 matrices in the usual way

     INPUT:
       A - the 1st 2x2 matrix
       B - the 2nd 2x2 matrix

    OUTPUT:
       the 2x2 matrix AB

    EXAMPLES:
        >>> A = [[1,2],[3,4]]; B = [[5,6],[7,8]]
        >>> mat_mult(A, B)
        [[19, 22], [43, 50]]
        >>> A = [[2,0],[0,3]]; B = [[4,0],[0,5]]
        >>> mat_mult(A, B)
        [[8, 0], [0, 15]]


    AUTHOR(S):
        <your name>

    TODO:
        Implement Strassen's algorithm [1] since it
        uses 7 multiplications instaead of 8!
```

```
    REFERENCES:
        [1] http://en.wikipedia.org/wiki/Strassen_algorithm

    """
    a1 = A[0][0]
    b1 = A[0][1]
    c1 = A[1][0]
    d1 = A[1][1]
    a2 = B[0][0]
    b2 = B[0][1]
    c2 = B[1][0]
    d2 = B[1][1]
    a3 = a1*a2+b1*c2
    b3 = a1*b2+b1*d2
    c3 = c1*a2+d1*c2
    d3 = c1*b2+d1*d2
    return [[a3,b3],[c3,d3]]
```

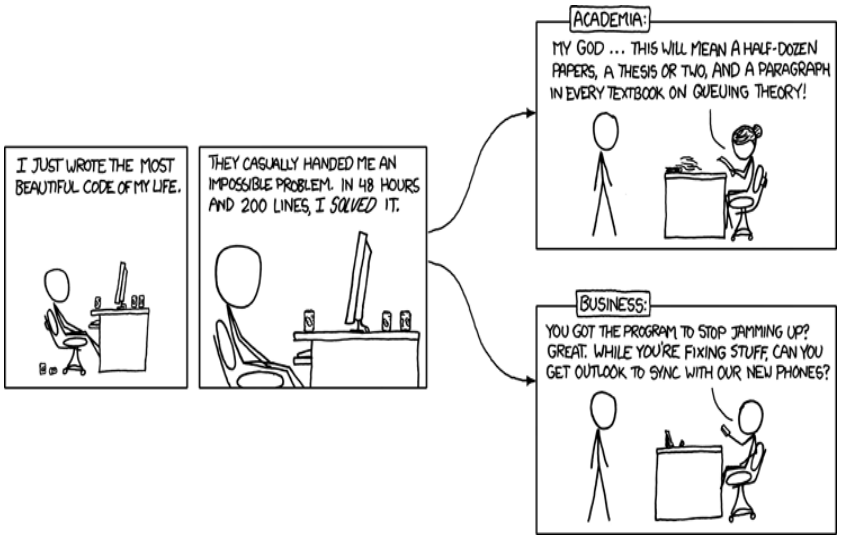


Figure 16: **Academia vs Business .**
`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

9.4 Pseudocode

Etymology

- *pseudo*: From the Ancient Greek $\phi\epsilon\nu\delta\eta\zeta$ (pseudes), meaning "false, lying"
- *code*: From the Old French (meaning "system of law") and Latin codex (meaning "book"), a later form of caudex ("a tablet of wood smeared over with wax, on which the ancients originally wrote").

This does not mean that your pseudocode can be false!
Example template of Python pseudocode.

Python

```
<variable> = <expression>

if <condition>:
    do stuff
else:
    do other stuff

while <condition>:
    do stuff

for <variable> in <sequence>:
    do stuff with variable

def <function name>(<arguments>):
    do stuff with arguments
    return something

<function name>(<arguments>)    # Function call
```

Here is a more detailed template of a Python function.

Python

```
def my_function(my_input1, my_input2 = my_default_value2):
    """
    Description.

     INPUT:
       my_input1 - the type of the 1st input
       my_input2 - the type of the 2nd input

    OUTPUT:
       the type of the output

    EXAMPLES:
        sage: my_function(1,2) # for a Sage program
        <the output>
        >>> my_function(1,2) # for a Python program
        <the output>
```

```
    AUTHOR(S):
        <your name>

    REFERENCES:
        [1] <A Wikipedia article describing the algorithm used>, <url>
        [2] <A book on algorithms describing the algorithm used>,
        <page numbers>
    """
    command1
    command2
    return output
```

Please remember these:

- Always indent using 4 spaces (no tabs).
- Comment, comment, comment. Even if your comment is longer than your program, still comment. (Please re-read §9.2 if you are unclear why that is important.)

Example 10. To illustratethe above-mentioned template, let's do an example of the so-called bisection method.

Suppose we have an integer-valued monotonicaly increasing function

$$f : \{0, 1, \ldots, M\} \to \mathbb{Z},$$

for some given integer M. Suppose that we are given n and we want to find m such that $f(m) = n$.

If the range of f is so large that we cannot enumerate the choices and search (the "brute force" way), then the following method might help.

Pseudocode:

pseudo–Python

```
low = 0
high = M
guess = (low + high)/2

while not(f(guess) == n):
    if f(guess) < n:
        low = guess
    else:
        high = guess
    guess = (low + high)/2

return guess
```

This is okay, except that if n is not in the range of f then it will run forever. We need to add another few statements to ensure that it will not run forever. We will also print out the number of steps the program takes to gives us better intuition as to how fast it runs.

Python

```
def inverse_image(fcn, val, max_domain):
    ""
    Description.

     INPUT:
       fcn - a monotonically increasing integer-valued function
       val - a value of that function
       max_domain - an integer M>0 defining the domain of fcn [0,1,..,M]

    OUTPUT:
       an integer m such that f(m) = val

    EXAMPLES:
        sage: f = lambda x: x^2
        sage: val = 11103^2
        sage: max_domain = 12500
        sage: inverse_image(f, val, max_domain); val
         (11103, 14)
         123276609

    Not bad - 14 steps to take the square-root of a 9 digit number!

    AUTHOR(S):
        John Q. Public

    REFERENCES:
        [1] Wikipedia article, http://en.wikipedia.org/wiki/Bisection_method
        [2] ''Introduction to Computer Science and Programming'',
        course taught by Prof. Eric Grimsom, Prof. John Guttag,
        MIT Fall 2008
        http://academicearth.org/courses/introduction-to-computer-science-and-programming

    ""
    counter = 1
    low = 0
    high = M
    guess = (low + high)/2

    while not(f(guess) == n) and counter<1000:
        if f(guess) < n:
            low = guess
        else:
            high = guess
        guess = (low + high)/2
        counter += 1

    assert counter <= 1000, 'Too many iterations'
    return guess, counter
```

Ars longa, vita brevis, occasio praeceps, experimentum periculosum, iudicium difficile (Life is short, [the] craft long, opportunity fleeting, experiment treacherous, judgment difficult.)
- *Hippocrates (c. 400BC)*

9.5 Exercises

Several of the exercises below will help you develop skills in algorithm design. The idea is to write a program in Sage or Python to solve the problem and to describe in pseudocode the algorithm you devised. Comment your program with detailed docstrings.

1. Explain and properly comment the following program.

Python

```
>>> def silly(y, x=3):
...          z = x
...          while(z>0):
...               y = y+x
...               z = z-1
...          return y
...
>>> silly(0,3)
9
>>> silly(0,5)
25
```

   Also, create a table of values for each step of the iteration.

2. Create a table of values of all the key variables for the extended Euclidean algorithm (see §4) for the case $a = 24$, $b = 15$.

3. A bowl of marbles in your math classroom contains 2009 green marbles and 2010 red ones. Every time you go to class, you must pick 2 marbles. If you pick 2 marbles of the same color, your math professor generously adds a red marble to the bowl. If you pick 2 marbles of different colora, your math professor generously adds a green marble to the bowl. What is the color of the last marble (hypotheticaly assuming you go to class for as many times as needed to answer the question)?

   Describe in pseudocode the algorithm you designed to solve this problem.

4. (`http://projecteuler.net/index.php?section=problems&id=24`, one of the easiest of the Project Euler problems) A permutation is an ordered arrangement of objects. For example, 3124 is one possible permutation of the digits 1, 2, 3 and 4. If all of the permutations are listed numerically or alphabetically, we say they are in *lexicographic order*. The lexicographically ordered permutations of 0, 1 and 2 are:

   $$012 \quad 021 \quad 102 \quad 120 \quad 201 \quad 210\ .$$

   What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9?

   Describe in pseudocode the algorithm you designed to solve this problem.

5. Take any 4-digit number with distinct digits. Permuting the digits gives $4! = 24$ different numbers. Let N be the maximum and n the minimum. Compute $N - n$. Repeat. Eventually you reach 6417 find the maximum number of repetitions to get to 6174.

   Describe in pseudocode the algorithm you designed to solve this problem.

6. (`http://projecteuler.net/index.php?section=problems&id=268`, the most difficult of the Project Euler problems as of Dec 15, 2009) It can be verified that there are 23 positive integers less than 1000 that are divisible by at least four distinct primes less than 100.

   Find how many positive integers less than 10^{16} are divisible by at least four distinct primes less than 100.

   Describe in pseudocode the algorithm you designed to solve this problem. Test it!

10 Classes in Python

A Python class can, for example, correspond to the mathematical object you are working with, e.g., a Matrix class for matrices, a DifferentialEquations class for differential equations, etc. This works very nicely for expressing mathematics, and is much different and conceptually superior to what you get in Mathematica and Matlab.

The Python class construction allows you to define your own new data types and methods for those data types. For example, you can define addition for instances of your Matrix class and also addition for instances of your DifferentialEquations class. You can use + for both operations (this is called operator overloading) and Python knows how to keep these different operations separate. Though modeled on C++ classes, Python classes are simpler and easier to use. They support both single and multiple inheritance and one can derive from builtin classes.

A `class` example ("borrowed" from Kirby Urber [U], a Python +mathematics educator from Portland Oregon).

Python

```
class Dog():
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'Dog(%s)'%self.name
   def __str__(self):
        return 'Dog named %s'%self.name
    def bark(self, loudness=1):
        if loudness == 1:
            print 'woof!'
        elif loudness == 2:
            print 'bark!'
        elif loudness == 3:
            print 'BARK!'
        else:
            print 'yipe-yipe-yipe!'
    def dogs_name(self):
        return self.name
```

Exercise 10.1. *Add docstrings to this code following the outline in* §**??**.

Once this class is read into Python, here is an example of its usage.

Python

```
>>> good_dog = Dog("zeus")
>>> type(good_dog)
<type 'instance'>
>>> type(Dog)
<type 'classobj'>
>>> good_dog
Dog named zeus
>>> good_dog.dogs_name()
'zeus'
```

```
>>> good_dog.bark(2)
bark!
```

The functions `bark` and `dogs_name` are examples of *methods* of the `Dog` class.

11 What is a code?

A *code* is a rule for converting data in one format, or well-defined tangible representation, into sequences of symbols in another format (and the finite set of symbols used is called the *alphabet*). We shall identify a code as a finite set of symbols which are the image of the alphabet under this conversion rule. The elements of this set are referred to as *codewords*. For example, using the ASCII code, the letters in the English alphabet get converted into numbers $\{0, 1, \ldots, 255\}$. If these numbers are written in binary then each codeword of a letter has length 8. In this way, we can reformat, or encode, a "string" into a sequence of binary symbols (i.e., 0's and 1's). *Encoding* is the conversion process one way. *Decoding* is the reverse process, converting these sequences of code-symbols back into information in the original format.

Some codes are used for *secure* communication (cryptography). Some codes are used for *reliable* communication (error-correcting codes). Some codes are used for *efficient* storage and communication (compression codes, hashes, Gray codes). We shall briefly study some of these later.

Other codes are merely simpler ways to communicate information (flag semaphores, color codes, genetic codes, braille codes, musical scores, chess notation, football diagrams, and so on), and have little or no mathematical structure. We shall not study them.

11.1 Basic definitions

If every word in the code has the same length, the code is called a *block code*. If a code is not a block code then it is called a *variable-length* code. A *prefix-free* code is a code (typically one of variable-length) with the property that there is no valid codeword in the code that is a prefix (start) of any other codeword[8]. This is the *prefix-free condition*.

[8]In other words, a codeword $s = s_1 \ldots s_m$ is a *prefix* of a codeword $t = t_1 \ldots t_n$ if and only if $m \leq n$ and $s_1 = t_1, \ldots, s_m = t_m$. Codes which are prefix-free are easier to decode

An example is the ASCII code. See for example, Michael Goerz' ASCII reference card at `http://users.physik.fu-berlin.de/~mgoerz/blog/refcards/`. (There is also a Python 2.5 reference card there too!)

Another example is

$$00, 01, 100.$$

A non-example is the code

$$00, 01, 010, 100$$

since the second codeword is a prefix of the third one. Another non-example is Morse code

a	·–	n	–·
b	–···	o	–––
c	–·–·	p	·––·
d	–··	q	––·–
e	·	r	·–·
f	··–·	s	···
g	––·	t	–
h	····	u	··–
i	··	v	···–
j	·–––	w	·––
k	–·–	x	–··–
l	·–··	y	–·––
m	––	z	––··

Table 1: Morse code

For example, look at the Morse code for `a` and the Morse code for `w`. These codewords violate the prefix-free condition.

than codes which are not prefix-free.

12 Gray codes

History[9] : Frank Gray[10] wrote about the so-called Gray codes in a 1951 paper published in the Bell System Technical Journal, and then patented a device (used for television sets) based on it in 1953. However, the idea of a binary Gray code appeared earlier. In fact, it appeared in an earlier patent (one by Stibitz in 1943). It was also used in E. Baudot's (a French engineer) telegraph machine of 1878 and in a French booklet by L. Gros on the solution to the "Chinese ring puzzle" published in 1872. The Gray code appearing in Frank Gray's 1953 patent, is a binary numeral system often used in electronics, but with many applications in mathematics.

Really, "the Gray code" is a misnomer, as that term encompasses a large class of related codes. We shall survey some of the constructions and applications of this very interesting class of "codes".

12.1 Binary Gray codes

A *binary Gray code* of length n is a sequence of 2^n n-tuples of 0's and 1's, where two successive terms of the sequence differ in exactly one coordinate.

Example 11. A binary Gray code of length 3:

$$000, 001, 011, 010, 110, 100, 101, 111$$

Another one:

$$000, 001, 011, 010, 110, 111, 101, 100$$

The coordinates in each term of a Gray code need not be taken only form the set $\{0, 1\}$. Let $m > 1$ be an integer. An *m-ary Gray code* of length n is a sequence of 2^n n-tuples elements taken from $\{0, 1, \ldots, m-1\}$, where two successive terms of the sequence differ in exactly one coordinate.

[9]This history comes from an unpublished section 7.2.1.1 ("Generating all n-tuples") in volume 4 of Donald Knuth's **The Art of Computer Programming**.

[10]Frank Gray (1887-1969) was a physicist and researcher at Bell Labs who made numerous innovations in television. He got his B.S. from Purdue University in 1911 and his PhD from the University of Wisconsin in 1916. He started worked at Bell Labs in 1925. He applied for the patent in 1947 but the patent was not awarded until 1953 for some reason.

Example 12. A 3-ary Gray code of length 2:

$$00, 10, 20, 21, 11, 01, 02, 12, 22.$$

Example 13. A 3-ary Gray code of length 3:

$$000, 100, 200, 210, 110, 010, 020, 120, 220, 221, 121, 021, 011, 111,$$

$$211, 201, 101, 001, 002, 102, 202, 212, 112, 012, 022, 122, 222.$$

Gray codes can be very useful in mathematics as they give a fast way of generating vectors in a vector space over a finite field. They also can be generalized to certain types of finite groups called Coxeter reflection groups.

Geometrically, a binary Gray code of length n can be visualized as a path along the edges of a unit hypercube in $\mathbb{R}^n$. A 3-ary Gray code can be visualized using a Sierpinski triangle (see for example, `http://en.wikipedia.org/wiki/Sierpinski_triangle` and §8.2 above).

Consider the so-called n-hypercube graph Q_n. This can be envisioned as the graph whose vertices are the vertices of a cube in n-space

$$\{(x_1, \ldots, x_n) \mid 0 \leq x_i \leq 1\},$$

and whose edges are those line segments in $\mathbb{R}^n$ connecting two "neighboring" vertices (namely, two vertices which differ in exactly one coordinate). A binary Gray code of length n can be regarded as a path on the hypercube graph Q_n which visits each vertex of the cube exactly once. In other words, a binary Gray code of length n may be identified with a Hamiltonian cycle on the graph Q_n (see Figure 17 for an example).

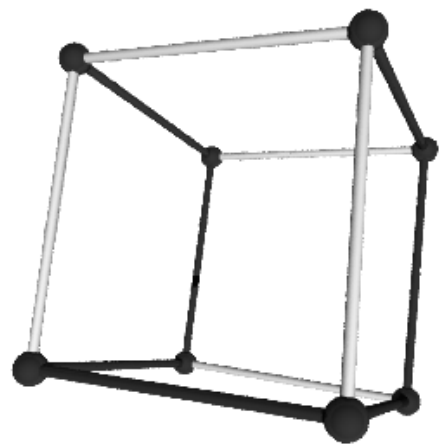


Figure 17: **Plot of Γ_3 viewed as a Hamiltonian path on Q_3 .**

How do you efficiently compute a Gray code?

Perhaps the simplest way to state the idea of quickly constructing the *reflected binary Gray code* Γ_n of length n is as follows:

$$\Gamma_0 = [], \qquad \Gamma_n = [0, \Gamma_{n-1}], [1, \Gamma_{n-1}^{rev}],$$

where Γ_m^{rev} means the Gray code in reverse order. For instance, we have

$$\Gamma_0 = [],$$
$$\Gamma_1 = [0], [1],$$
$$\Gamma_2 = [[0,0], [0,1], [1,1], [1,0],$$

and so on. This is a nice procedure if you want to create the entire list at once (which, by the way, gets very long very fast).

An implementation of the reflected Gray code using Python is given below.

Python 3.0

```
def graycode(length,modulus):
    """
    Returns the n-tuple reflected Gray code mod m.
```

```
    EXAMPLES:
        sage: graycode(2,4)

        [[0, 0],
         [1, 0],
         [2, 0],
         [3, 0],
         [3, 1],
         [2, 1],
         [1, 1],
         [0, 1],
         [0, 2],
         [1, 2],
         [2, 2],
         [3, 2],
         [3, 3],
         [2, 3],
         [1, 3],
         [0, 3]]

    """
    n,m = length,modulus
    F = range(m)
    if n == 1:
        return [[i] for i in F]
    L = graycode(n-1, m)
    M = []
    for j in F:
        M = M+[ll+[j] for ll in L]
    k = len(M)
    Mr = [0]*m
    for i in range(m-1):
        i1 = i*int(k/m)       # this requires Python 3.0 or Sage
        i2 = (i+1)*int(k/m)
        Mr[i] = M[i1:i2]
    Mr[m-1] = M[(m-1)*int(k/m):]
    for i in range(m):
        if is_odd(i):
            Mr[i].reverse()
    M0 = []
    for i in range(m):
        M0 = M0+Mr[i]
    return M0
```

Consider the reflected binary code of length 8, Γ_8. This has $2^8 = 256$ codewords. Sage can easily create the list plot of the coordinates (x, y), where x is an integer $j \in \mathbb{Z}_{256}$ which indexes the codewords in Γ_8 and the corresponding y is the j-th codeword in Γ_8 converted to decimal. This will give us some idea of how the Gray code "looks" in some sense. The plot is

given in Figure 18.

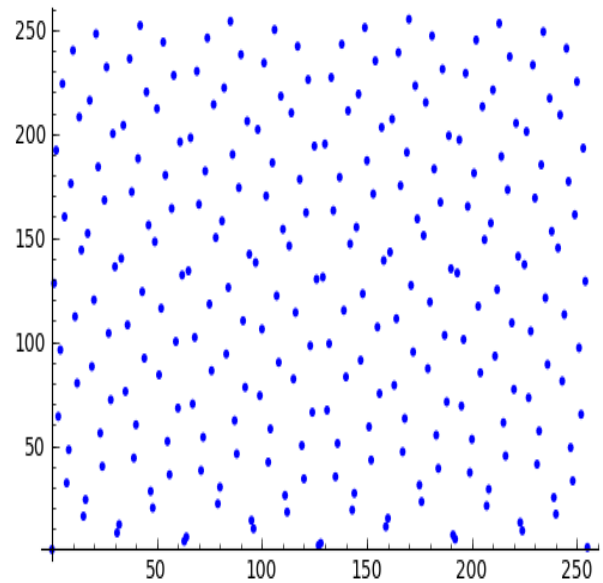


Figure 18: **List plot of Γ_8 created using Sage.**

What if you only want to compute the i-th Gray codeword in the Gray code of length n? Can it be computed quickly as well without computing the entire list? At least in the case of the reflected binary Gray code, there is a very simple way to do this. The k-th element in the above-described reflected binary Gray code of length n is obtained by simply adding the binary representation of k to the binary representation of the integer part of $k/2$.

An example using Sage is given below.

Sage

```
def int2binary(m, n):
    '''
    returns GF(2) vector of length n obtained
    from the binary repr of m, padded by 0's
    (on the left) to length n.

    EXAMPLES:
        sage: for j in range(8):
        ....:     print int2binary(j,3)+int2binary(int(j/2),3)
        ....:
```

```
        (0, 0, 0)
        (0, 0, 1)
        (0, 1, 1)
        (0, 1, 0)
        (1, 1, 0)
        (1, 1, 1)
        (1, 0, 1)
        (1, 0, 0)
    '''
    s = bin(m)
    k = len(s)
    F = GF(2)
    b = [F(0)]*n
    for i in range(2,k):
        b[n-k+i] = F(int(s[i]))
    return vector(b)

def binary2int(b):
    """
    inverts int2binary

    """
    k = len(b)
    n = sum([int(b[i])*2**(k-1-i) for i in range(k)])
    return n

def graycodeword(m, n):
    '''
    returns the mth codeword in the reflected binary Gray code
    of length n.

    EXAMPLES:
        sage: graycodeword(3,3)
        (0, 1, 0)
    '''
    return int2binary(m,n)+int2binary(int(m/2),n)
```

Exercise 12.1. *Convert the above function* `graycodeword` *into a pure* Python *function.*

12.2 Non-binary Gray codes

The term "Gray code" is ambiguous. It is actually a large family of sequences of n-tuples. Let $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. More precisely, an *$m$-ary Gray code of length n* (called a *binary Gray code* when $m = 2$) is a sequence of all possible (namely, $N = m^n$) n-tuples

$$g_1, g_2, \ldots, g_N,$$

where

- each $g_i \in \mathbb{Z}_m^n$,
- g_i and g_{i+1} differ by 1 in exactly one coordinate.

In other words, an m-ary Gray code of length n is a particular way to order the set of all m^n n-tuples whose coordinates are taken from $\mathbb{Z}_m$. From the transmission/communication perspective, this sequence has two advantages:

- It is easy and fast to produce the sequence, since successive entries differ in only one coordinate.
- An error is relatively easy to detect, since you can compare an n-tuple with the previous one. If they difer in more than one coordinate, you know an error was made.

Example 14. Here is a 3-ary Gray code of length 2:

$$[0,0],[1,0],[2,0],[2,1],[1,1],[0,1],[0,2],[1,2],[2,2]$$

and here is a binary Gray code of length 3:

$$[0,0,0],[1,0,0],[1,1,0],[0,1,0],[0,1,1],[1,1,1],[1,0,1],[0,0,1].$$

Gray codes have applications to engineering, recreational mathematics (solving the Tower of Hanoi puzzle, "The Brain" puzzle, the "Chinese ring puzzle", and others), and to mathematics (for example, aspects of combinatorics, computational group theory and the computational aspects of linear codes).

Next, let's try creating a decimal (i.e., 10-ary) Gray code of length 3. How far will the usual process of counting get us? We start

$$(0,0,0),(0,0,1),(0,0,2),\ldots,(0,0,9),$$

but the next natural choice, namely $(0,1,0)$, won't work since it changes 2 coordinates. Instead, let's pick $(0,1,9)$ and count in reverse order,

$$(0,1,9),(0,1,8),\ldots,(0,1,0).$$

Note that $(0,1,9)$ really was the smallest vector which had not yet been choosen after $(0,0,9)$ and which had the key property that it differed in

exactly one coordinate. After selecting that, we "filled in gaps" in the only way possible. Now we have

$$(0,0,0),(0,0,1),(0,0,2),\ldots,(0,0,9),(0,1,9),(0,1,8),\ldots,(0,1,0),$$

we choose the smallest vector which has not yet been choosen. This is $(0,2,0)$, so we start counting again,

$$(0,2,0),(0,2,1),\ldots,(0,2,9),$$

but we again have to stop. Pick the smallest legal one $(0,3,9)$ and count in reverse order,

$$(0,3,9),(0,3,8),\ldots,(0,3,0).$$

This type of construction is an example of a "greedy algorithm[11]" In any case, it is clear that this procedure will produce a decimal Gray code of length three.

In general, this algorithm generalizes to one which does not even require one with the same "radix" for each coordinate. Suppose you want to compute a *mixed-radix Gray code* which is a sequence of $N = \prod_{i=0}^{n-1} m_i$ codewords (for a fixed list of "radixes" $m_0, m_1, \ldots, m_{n-1}$, each of which is > 1),

$$(a_0, a_1, \ldots, a_{n-1}),$$

where $0 \leq a_i \leq m_i$ for all i, and each element of the sequence differs from a neighboring element by ± 1 in exactly one coordinate.
Algorithm: **Input**: A length n and a list of radixes $m_0, m_1, \ldots, m_{n-1}$.
Output: A mixed-radix Gray code of length n.

- Start with the all 0 tuple of length n, $(0,\ldots,0)$.
- Find the lexicographically smallest element which is "legally" a Gray codeword and append it to the current lsit of codewords.

[11] Wikipedia, which more-or-less follows the NIST definition in `http://www.itl.nist.gov/div897/sqg/dads/HTML/greedyalgo.html`, has a great definition: "A *greedy algorithm* is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.".

- Repeat until all a codewords are obtained.

Example 15. Here is an example of a mixed-radix Gray code with entries in $\mathbb{Z}_3 \times \mathbb{Z}_2 \times \mathbb{Z}_4$. For brevity, we write a codeword (a, b, c) as abc.

First, construct the entries with a 0 in the first coordinate:

$$000, 001, 002, 003, 013, 012, 011, 010. \tag{1}$$

Note the last four codewords can be obtained from the first four by "reflection" and substituting 1 for 0 in the second coordinate. Now, reflect all these and substitute 1 for 0 in the first coordinate:

$$110, 111, 112, 113, 103, 102, 101, 100. \tag{2}$$

Now, reflect all these and substitute 2 for 1 in the first coordinate:

$$200, 201, 202, 203, 213, 212, 211, 210. \tag{3}$$

Concatenating (1), (2), (3) together gives the $24 = 3 \cdot 2 \cdot 4$ elements of the Gray code.

12.3 An application of Gray codes to mathematics

There are many applications of Gray codes to mathematics. For example, the construction of fractals and space-filling curves can be accomplished using Gray codes. In this section, we focus on an application to linear codes in a particular example.

Gray codes and linear codes

In the computational aspects of error-correcting codes, it is very important to be able to compute, or at least find a good approximation for, the so-called minimum distance of the code. The only *general* method of doing this is to search over all codewords and compute the ones of minimum Hamming weight. The fastest way (known to me at this time) to implement this search uses Gray codes.

The idea easily is illustrated using an example.

Example 16. Consider the binary Hamming code C with parameters $[7, 4, 3]$. We shall discuss error-correcting codes in general later. For now, we simply define C to be the subset of vectors of $GF(2)^7$ of the form

$$E(m) = (m_1, m_2, m_3, m_4, m_1 + m_3 + m_4, m_1 + m_2 + m_4, m_1 + m_2 + m_3 + m_4),$$

where $m = (m_1, m_2, m_3, m_4)$ run over all possible elements in $GF(2)^4$. (Think of m as the "information" you want to transmit over a noisy channel and $E(m)$ as the message you send. The message contains the information plus some redundancy. Hopefully there is enough redundancy for the receiver to recover the information if an error was made during transmission.) Gray codes arise in the attempt to generate this set as quickly as possible.

Let

$$b_1 = (1, 0, 0, 0, 1, 1, 1),\ \ b_2 = (0, 1, 0, 0, 0, 1, 1),\ \ b_3 = (0, 0, 1, 0, 1, 0, 1),\ \ b_4 = (0, 0, 0, 1, 1, 1, 0).$$

Then we can write $E(m)$ as

$$E(m) = m_1 \cdot (1, 0, 0, 0, 1, 1, 1) + m_2 \cdot (0, 1, 0, 0, 0, 1, 1) + m_3 \cdot (0, 0, 1, 0, 1, 0, 1) + m_4 \cdot (0, 0, 0, 1, 1, 1, 0) = m_1$$

(Think of the b_i's as basis vectors spanning a vector space.) Let Γ_4 denote the reflected binary Gray code of length 4. This is the set $GF(2)^4$ ordered in such a way that successive elements differ in exactly one bit:

$$\begin{aligned}
&[0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 0, 0], [0, 1, 0, 0], [0, 1, 1, 0],\\
&[1, 1, 1, 0], [1, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 1], [1, 0, 1, 1],\\
&[1, 1, 1, 1], [0, 1, 1, 1], [0, 1, 0, 1], [1, 1, 0, 1], [1, 0, 0, 1], [0, 0, 0, 1].
\end{aligned}$$

Here is a short algorithm to generate C from Γ_4. Write

$$\Gamma_4 = \{g_0 = (0, 0, 0, 0), g_1, g_2, \ldots, g_{15}, g_{16} = g_0\}.$$

Initialize: $C = \{(0, 0, 0, 0, 0, 0, 0)\}$. $c = (0, 0, 0, 0, 0, 0, 0)$. (Think of c as the last element you added to the set C.)
for i in $\{1, \ldots, 2^4 = 16\}$:

- if g_i and g_{i-1} only differ in the k-th coordinate ($1 \leq k \leq 4$) then let

$$c = c + b_k.$$

- Add c to C.

At the end of this for loop, you will have constructed all possible elements of C.

Sage

```
G4 = graycode(4,2)
G4.append([0,0,0,0])
c = vector(GF(2), [0,0,0,0,0,0,0])
C = [c]
b1 = vector(GF(2), [1,0,0,0,1,1,1])
b2 = vector(GF(2), [0,1,0,0,0,1,1])
b3 = vector(GF(2), [0,0,1,0,1,0,1])
b4 = vector(GF(2), [0,0,0,1,1,1,0])
b = [b1,b2,b3,b4]
for i in range(1,16):
    k = add_vectors_mod_m(G4[i],G4[i-1],2).index(1)
   # this picks on where the vectors differ by 1
    c = c + b[k]
    C.append(c)
```

This generates the set

$$\{(0,0,0,0,0,0,0),(1,0,0,0,1,1,1),(1,1,0,0,1,0,0),(0,1,0,0,0,1,1),(0,1,1,0,1,1,0),(1,1,1,0,0,$$
$$(1,0,1,0,0,1,0),(0,0,1,0,1,0,1),(0,0,1,1,0,1,1),(1,0,1,1,1,0,0),(1,1,1,1,1,1,1),(0,1,1,1,0,$$
$$(0,0,0,1,1,1,0)\}.$$

13 Huffman codes

According to the September 1991 issue of **Scientific American** (see [HSA], [HW]):

> In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the

idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, the student outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code (the suboptimal Shannon-Fano coding scheme).

Here is the informal description of the problem that Prof. Fano gave his students:
Given: A set of symbols, say $A = \{a_1, a_2, \ldots, a_n\}$, and their weights, say $W = \{w_1, w_2, \ldots, w_n\}$ (usually proportional to probabilities of occurences). We shall assume thoughout that each $w_i > 0$.
Find: A prefix-free binary code (a set of codewords) with minimum expected codeword length.

In other words, if $C = C_{A,W} = \{c_1, c_2, \ldots, c_n\}$ is the code (the encoder simply being the map $a_i \longmapsto c_i$) then each c_i is a binary vector, say of length ℓ_i, and the expected codeword length

$$L(C) = \sum_{i=1}^{n} w_i \ell_i,$$

is minimal among all such prefix-free codes.

The algorithms for constructing a Huffman code are relatively sophisticated. We refer to Biggs [B], §3.6. However, there are several implementations of Huffman coding written in Python available free on the internet.

Example 17. We shall use the following program which can be found on the Python wiki.

Python

```
def huffman(freqtable):
    """
    Generate Huffman codes
    http://wiki.python.org/moin/ProblemSets
             /99%20Prolog%20Problems%20Solutions#Problem50.3AGenerateHuffmancodes

    License: Python License
                  http://www.python.org/psf/license/

    Return a dictionary mapping keys to huffman codes
    for a frequency table mapping keys to frequencies.

    >>> freqtable = dict(a=45, b=13, c=12, d=16, e=9, f=5)
    >>> sorted(huffman(freqtable).items())
    [('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'),
     ('f', '1100')]
```

```
    """
    from collections import defaultdict
    from heapq import heappush, heappop, heapify
    # mapping of letters to codes
    code = defaultdict(list)
    # Using a heap makes it easy to pull items with lowest frequency.
    # Items in the heap are tuples containing a list of letters and the
    # combined frequencies of the letters in the list.
    heap = [ ( freq, [ ltr ] ) for ltr,freq in freqtable.iteritems() ]
    heapify(heap)
    # Reduce the heap to a single item by combining the two items
    # with the lowest frequencies.
    while len(heap) > 1:
        freq0,letters0 = heappop(heap)
        for ltr in letters0:
            code[ltr].insert(0,'0')
        freq1,letters1 = heappop(heap)
        for ltr in letters1:
            code[ltr].insert(0,'1')
        heappush(heap, ( freq0+freq1, letters0+letters1))
    for k,v in code.iteritems():
        code[k] = ''.join(code[k])
    return code
```

Let us use it to find the Huffman code for the statement

"I like huffman codes more than brussels sprouts",

with apologies to all those Brussels sprouts lovers out there.

Python

```
>> s = "I like huffman codes more than brussels sprouts"
>> A = ("","a","b","c","d","e","f","g","h","i","j","k","l","m","n","o",
       "p","q","r","s","t","u","v","w","x","y","z")
>> freq = {}
>>> for a in A:
...     if a in s:
...         freq[a] = s.count(a)
...     else:
...         freq[a] = 0
...
>>> freq
{' ': 7, 'a': 2, 'c': 1, 'b': 1, 'e': 4, 'd': 1, 'g': 0, 'f': 2, 'i': 1,
'h': 2, 'k': 1, 'j': 0, 'm': 2, 'l': 2, 'o': 3, 'n': 2, 'q': 0, 'p': 1,
's': 6, 'r': 3, 'u': 3, 't': 2, 'w': 0, 'v': 0, 'y': 0, 'x': 0,
'z': 0}
>>> Freq = [(x,y) for (y,x) in freq.items()]
>>> sorted(Freq)
[(0, 'g'), (0, 'j'), (0, 'q'), (0, 'v'), (0, 'w'), (0, 'x'), (0, 'y'),
(0, 'z'), (1, 'b'), (1, 'c'), (1, 'd'), (1, 'i'), (1, 'k'), (1, 'p'),
(2, 'a'), (2, 'f'), (2, 'h'), (2, 'l'), (2, 'm'), (2, 'n'), (2, 't'),
(3, 'o'), (3, 'r'), (3, 'u'), (4, 'e'), (6, 's'), (7, ' ')]
```

Now we run the above program on this dictionary and sort the output:

Python

```
>>> sorted(huffman(freq).items())
[(' ', '101'), ('a', '11010'), ('b', '1111101'), ('c', '110110'),
('d','110111'), ('e', '1110'), ('f', '11110'), ('g', '11111000000000'),
('h', '0000'), ('i', '111111'), ('j', '11111000000001'), ('k', '00010'),
('l', '0010'), ('m', '0011'), ('n', '0100'), ('o', '0110'), ('p', '00011'),
('q', '1111100000001'), ('r', '0111'), ('s', '100'), ('t', '0101'),
('u', '1100'), ('v', '111110000001'), ('w', '11111000001'),
('x','1111100001'), ('y', '111110001'), ('z', '11111001')]
```

As you can see, the most common character symbols get assigned to the shortest codewords in the Huffman code for our statement above.

13.1 Exercises

Exercise 13.1. Verify this for your own statement. (Make one up or use your favorite quotation.)

Hand in the code, frequence table and the Python programming you did to produce them.

14 Error-correcting, linear, block codes

Error-correcting codes are used to facilitate reliable communication of digital information. Basically, you add redundancy in a clever way to allow the receiver to recover the message even if there were lots of errors in the transmission due to "noise" in the communication channel. Cell-phones, computers, DVDs, and many other devices use error-correcting codes. Postal codes (the little stripes at the bottom fo an envelope), ISBN codes, and product bar-codes are other examples. Different devices have different noise characteristics, and so use different types of codes. As with shoes, no one size fits all. The noise in a cell-phone is more variable (for example, if you are talking while driving in your car and moving away from a cell-phone tower), and also requires less fidelity than say a music CD player. Indeed, the error-correcting codes used by cell-phones today is much different than that used by CDs and DVDs. The type of error-coorecting code used by CDs and DVDs is called a "block" code. This means that you break up the digital data to be transmitted into blocks of a fixed size, say k bits, encodes that

block by adding $n-k$ redundancy bits, and transmits that n-bit block to the receiver. For example, NASA's Mariner spacecraft (between 1969 and 1977) used a Reed-Muller code. We shall discuss Reed-Muller codes briefly below.

14.1 The communication model

Consider a source sending messages through a noisy channel. The message sent will be regarded as a vector of length n whose entries are taken from a given finite field F (typically, $F = GF(2)$).

For simplicity, assume that the message being sent is a sequence of 0's and 1's. Assume that, due to noise, when a 0 is sent, the probability that a 0 is (correctly) received is p and the probability that a 1 is (incorrectly) received is $1-p$. Assume also that the noise of the channel is not dependent on the symbol sent: when a 1 is sent, the probability that a 1 is (correctly) received is p and the probability that a 0 is (incorrectly) received is $1-p$. Here p is a fixed probability which depends on the noise on the channel, $0 < p < 1/2$.

14.2 Basic definitions

The basic definition explains how the theory of linear codes relies heavily on basic linear algebra.

Definition 18. A *linear error-correcting block code*, or *linear code* for short, finite dimensional vector space with a fixed basis.

We shall typically think of a linear code as a subspace of $\mathbb{F}^n$ with a fixed basis, where $\mathbb{F}$ is a finite field and $n > 0$ is an integer called the *length* of the code. Moreover, the basis for the *whole space code* $\mathbb{F}^n$ will typically be the standard basis,

$$e_1 = (1, 0, \ldots, 0), e_2 = (0, 1, 0, \ldots, 0), \ldots, e_n = (0, \ldots, 0, 1). \tag{4}$$

There are two common ways to specify a linear code C.

- You can give C as a vector subspace of $\mathbb{F}^n$ by specifying a set of basis vectors for C. This set of basis vectors is, by convention, placed as the rows of a matrix called a *generator matrix* of C. Obviously, the order in which the rows are presented does not affect the code itself.

  If $g_1, \ldots g_k$ are the rows of G then

$$C = \{c = m_1 g_1 + \cdots + m_k g_k \mid somem_i \in GF(q)\},$$

is the set of linear combinations of the row vectors g_i. The vector of coefficients, $m = (m_1, \ldots, m_k)$ is sometimes called the *message vector* or *information vector*. In other words, encoding of a message can be defined via the generator matrix:

$$\begin{array}{ll} m = (m_1, \ldots, m_k) & \mapsto c = m_1 g_1 + \cdots + m_k g_k, \\ \mathbb{F}^k & \qquad\qquad \to C. \end{array} \tag{5}$$

- You can give C as a vector subspace of $\mathbb{F}^n$ by specifying a matrix H for which C is the kernel of H, $C = \ker(C)$. This matrix is called a *check matrix* of C. Again, the order in which the rows are presented does not affect the code itself.

These two ways of defining a code are not unrelated.

Proposition 19. *If $G = (I_k \mid A)$ is the generating matrix for C then $H = (-A^t \mid I_{n-k})$ is a parity check matrix.*

The proof of this is not too hard if you know how block matrix multiplication works and can verify that $H \cdot {}^tG = 0 = (-A^t \mid I_{n-k}) \cdot {}^t(I_k \mid A) = 0$.

A code with symbols taken from $GF(p)$ is sometimes called a *p-ary* code, though when $p = 2$ you usually simply say *binary* and for $p = 3$ you say *ternary*.

Geometrically, two codewords are "far" from each other if there are "a lot" of coordinates where they differ.

Definition 20. If $v, w \in \mathbb{F}^n$ are vectors then we define

$$d(v, w) = |\{i \mid v_i \neq w_i,\ 1 \leq i \leq n\},$$

to be the *Hamming distance* between v and w. The function d is called the *Hamming metric*. The *weight* of a vector v (in the Hamming metric) is the Hamming distance between v and the 0 vector.

A *metric* on a set X is a function

$$d : X \times X \to \mathbb{R}$$

(where $\mathbb{R}$ is the set of real numbers). For all $x, y, z \in X$, this function is required to satisfy the following conditions:

- $d(x, y) \leq 0$ and $d(x, y) = 0$ if and only if $x = y$,
- $d(x, y) = d(y, x)$ (symmetry)
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

Lemma 21. The Hamming metric is a metric on the vector space $V = GF(q)^n$.

proof: We must show $d(u, w) \leq d(u, v) + d(v, w)$, for all vectors $u, v, w \in V$.

We have: if $u_i \neq w_i$ then

- $u_i = v_i$ and $v_i \neq w_i$, or
- $u_i \neq v_i$ and $v_i = w_i$, or
- $u_i \neq v_i$ and $v_i \neq w_i$.

Counting these conditions, we see

$$|\{i \mid u_i \neq w_i\}| \leq |\{i \mid u_i \neq v_i\}| + |\{i \mid v_i \neq w_i\}|,$$

since $|\{i \mid u_i \neq v_i\}|$ counts the last two conditions and $|\{i \mid v_i \neq w_i\}|$ counts the first two conditions. $\square$

14.3 Decoding

Suppose a codeword $c \in C$ is sent over a noisy channel. Let $v \in GF(q)^n$ denote the received vector. If no error was made in transmission (the most likely scenerio), then $v = c$. If a single error was made (the second most likely scenerio), then v and c differ in exactly one bit, and so on.

Here is the simplest method of decoding, or correcting, an error in transmission - in other words, determining c from v.

Nearest neighbor decoding:
INPUT: A code $C \subset GF(q)^n$ and a vector $v \in GF(q)^n$.
OUTPUT: A codeword $c \in C$ with $d(v, c)$ as small as possible.

- Initialize $c_0 = 0$.
- For all $c \in C$: if $d(c, v) < d(c_0, v)$ then $c_0 = c$.

- Return c_0.

Let v, w be any vectors in $GF(q)^m$. We say v is *equivalent* to w if there is a non-zero scalar $r \in GF(q)$ such that $v = r \cdot w$. Otherwise, we say that v, w are *inequivalent*.

Proposition 22. Let $C \subset GF(q)^n$ be a code with check matrix H. Let $v \in GF(q)^n$ be a vector which differs from some codeword in C in at most one coordinate. The nearest neighbor algorithm can compute the error coordinate and the codeword if all columns of H are inequivalent.

proof: Suppose $v = c + a \cdot e_i$, for some $c \in C$, some non-zero $a \in GF(q)$, and some i (where e_i is the i-th standard basis vector of $GF(q)^n$). Can we solve for c, a and i? Yes, here is how. Compute

$$Hv = H(c + a \cdot e_i) = Hc + a \cdot He_i = a \cdot He_i,$$

which is a times the i-th column vector of H. But all these column vectors are inequivalent, so knowing $a \cdot He_i$, we can determine i and a. This allows us to determine $c = v - a \cdot e_i$. $\square$

If x is a real number, let $[x]$ denote its integer part.

Proposition 23. If C is an $[n, k, d]$ code then the nearest neighbor algorithm can correct $\leq [(d-1)/2]$ errors.

proof: Let $v \in GF(q)^n$ be a received vector. Assume $\leq [(d-1)/2]$ errors have been made in transmission. This means that the Hamming distance from v to the sent codeword c is $\leq [(d-1)/2]$. Assume that the nearest neighbot algorithm returns a codeword c', so c' is the closest codeword to v. We have

$$d(c', v) \leq d(c, v) \leq [(d-1)/2].$$

By the triangle inequality

$$d(c, c') \leq d(c, v) + d(c', v) \leq d - 1 < d.$$

This means $c' - c$ is a codeword of weight $< d$, so $c' = c$. $\square$

14.4 The covering radius

Question: What is the smallest radius r such that the balls of radius r centered about all the codewords,

$$B(c, r) = \{v \in GF(q)^n \mid d(c, v) \leq r\}$$

are disjoint.

Answer: $[(d-1)/2]$. By the above proof, we see that the triangle inequality will not allow two balls centered at neighboring codewords are disjoint if and only if they have radius $\leq [(d-1)/2]$.

The union of all these disjoint balls of radius $[(d-1)/2]$ centered at the codewords in C usually does not equal the entire space $V = GF(q)^n$. (When it does, C is called *perfect*) How much larger do we have to make the radius so that the union of these balls does cover all of V? In other words, we want to increase the radius $r = [(d-1)/2]$ to some new radius rho so that

$$\cup_{c \in C} B(c, \rho) = V.$$

This new radius is called the *covering radius*. In general, it is hard to find good upper bounds on ρ.

We need some basic facts about finite fields before proceeding further into the theory of linear codes.

14.5 Finite fields

What is a finite field? As you probably know already, a *field* is an algebraic structure with two binary operations, usually denoted $+$ (called *addition*) and $\cdot$ (or simply juxtaposition, called *multiplication*. These operations satisfy certain axioms such as associativity and distributivity. They are listed for completeness below.

- *Closure* of $\mathbb{F}$ under addition and multiplication: For all $a, b \in \mathbb{F}$, both $a + b$ and $a \cdot b$ are in F.
- *Associativity* of addition and multiplication: For all $a, b, c \in \mathbb{F}$, the following equalities hold: $a+(b+c) = (a+b)+c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- *Commutativity* of addition and multiplication: For all $a, b \in \mathbb{F}$, the following equalities hold: $a + b = b + a$ and $a \cdot b = b \cdot a$.

- Additive and multiplicative *identity*: There exists an element of $\mathbb{F}$, called the additive identity and denoted by 0, such that for all $a \in \mathbb{F}$, $a + 0 = a$. Likewise, there is another element, called the multiplicative identity and denoted by 1, such that for all $a \in \mathbb{F}$, $a \cdot 1 = a$. (In particular, any field must contain at least 2 distinct elements, 0 and 1.)

- Additive and multiplicative *inverses*: For every $a \in \mathbb{F}$, there exists an element $-a \in \mathbb{F}$, such that $a + (-a) = 0$. Similarly, for any $a \in \mathbb{F} - \{0\}$, there exists an element $a^{-1} \in \mathbb{F}$, such that $a \cdot a^{-1} = 1$.

- *Distributivity* of multiplication over addition: For all $a, b, c \in \mathbb{F}$, the following equality holds: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Examples of finite fields are not hard to construct. For example, look at the set of integers modulo a prime p, denoted[12]

$$\mathbb{Z}/p\mathbb{Z} = GF(p) = \{0, 1, \ldots, p-1\},$$

with addition and multiplication performed modulo p. This is called "the" finite field of prime order p, or sometimes simply a *prime field*.

Modular arithmetic is defined as follows. Two integers a and b are said to be *congruent* modulo p, denoted $a \equiv b \pmod{p}$, if their difference $a - b$ is an integer multiple of p. Compared to familiar addition and multiplication of integers $\mathbb{Z}$, on the set $\mathbb{Z}/p\mathbb{Z}$,

- replace $=$ on $\mathbb{Z}$ by $\equiv \pmod{p}$,

- replace $+$ on $\mathbb{Z}$ by addition followed by reducing modulo p,

- replace $\cdot$ on $\mathbb{Z}$ by multiplication followed by reducing modulo p.

For example, if $p = 7$ then $4 + 5 = 9 \equiv 2 \pmod{7}$, so $4 + 5 = 2 \in \mathbb{Z}/7\mathbb{Z}$. Likewise, $4 \cdot 5 = 20 \equiv 6 \pmod{7}$, so $4 \cdot 5 = 6 \in \mathbb{Z}/7\mathbb{Z}$. Since, $4 + 3 = 7 \equiv 0 \pmod{7}$, so $-4 = 3 \in \mathbb{Z}/7\mathbb{Z}$ (and $-3 = 4 \in \mathbb{Z}/7\mathbb{Z}$). Since $3 \cdot 5 = 15 \equiv 1 \pmod{7}$, we see $3^{-1} = 5 \in \mathbb{Z}/7\mathbb{Z}$.

You can see that addition and multipliation is pretty easy. The hardest operation is division. How do you compute the inverse of an element? Use the extended Euclidean algorithm (Example 4 above). Suppose $a \in \mathbb{Z}/p\mathbb{Z}$

[12] Both $GF(p)$ and $\mathbb{Z}/p\mathbb{Z}$ are commonly used notations for this field.

is non-zero and you want to compute a^{-1}. Since a and p have no common factors (remember, p is a prime and $0 < a < p$), by the extended Euclidean algorithm, there are x, y such that $ax + py = 1$. It turns out that $a^{-1} = x$. Why? Write $ax+py = 1$ as $ax-1 = p\cdot(-y)$. This implies $a \cdot x \equiv 1 \pmod{p}$, or $ax = 1 \in \mathbb{Z}/p\mathbb{Z}$. Therefore, by definition, $x = a^{-1}$.

There is no finite field class in the version of Python you download from `python.org`. However Sage [S] has excellent functionality for finite fields built in.

Exercise 14.1. *Create a class structure for $\mathbb{Z}/p\mathbb{Z}$ with methods for addition, multiplication, subtraction, and division.*

There are other finite fields besides $GF(p)$. It turns out that all finite field $\mathbb{F}$ have the following interesting properties:

- The set $\mathbb{F}-\{0\}$ (denoted $\mathbb{F}^\times$) provided with the multiplicative operation of the field is a cyclic group.
- There is a unique prime p such that $p \cdot a = 0 \in \mathbb{F}$ for all $a \in \mathbb{F}$. Here $p \cdot a$ simply means $a + a + \cdots + a$ (p times). (This prime is called the *characteristic* of $\mathbb{F}$.) Moreover, $GF(p)$ is a subfield of $\mathbb{F}$ and $\mathbb{F}$ is a finite dimensional vector space over $GF(p)$.

If $\dim_{GF(p)} \mathbb{F} = k$ then $\mathbb{F}$ is sometimes denoted as $GF(p^k)$.

Example 24. The most commonly used finite field which is not a prime field is $GF(4)$. There are several ways to construct this. One is to specific the set of elements

$$GF(4) = \{0, 1, z, z + 1\},$$

and then to define $+$ and $\cdot$ as addition and multiplication modulo $z^2 + z + 1$ and modulo 2 (so, for example, $z^2 = -z - 1 = z + 1$).

The addition table for $GF(4)$:

$+$	0	1	z	$z+1$
0	0	1	z	$z+1$
1	1	0	$z+1$	z
z	z	$z+1$	0	1
$z+1$	$z+1$	z	1	0

The multiplication table for $GF(4)$:

$\cdot$	0	1	z	$z+1$
0	0	0	0	0
1	0	1	z	$z+1$
z	0	z	$z+1$	1
$z+1$	0	$z+1$	1	z

14.5.1 A simple Python class for a prime finite fields

```
"""
Finite fields in Python.

"""

#def FF(p):
#    return FF_prime(p)

class FF:
    """
    Implements "prime" finite fields.

    EXAMPLES:
        sage: F = FF(5)
        sage: print F
        Finite field with 5 elements
        sage: F
        FF(5)

    """
    def __init__(self, p):
        self.characteristic = p

    def __repr__(self):
        """
        Called to compute the "official" string representation of an object.
```

```
        If at all possible, this should look like a valid Python expression
        that could be used to recreate an object with the same value.

        EXAMPLES:
            sage: F = FF(5)
            sage: F
            FF(5)

        """
        return "FF(%s)"%self.characteristic

    def __str__(self):
        """
        Called to compute the "informal" string description of an object.

        EXAMPLES:
            sage: F = FF(5)
            sage: print F
            Finite field with 5 elements

        """
        return "Finite field with %s elements"%self.characteristic

    def __lt__(self, other):
        """
        Returns True of self < other, False otherwise.
        """
        return False

    def __gt__(self, other):
        """
        Returns True of self > other, False otherwise.
        """
        return False

    def char(self):
        return self.characteristic
```

```
def __eq__(self, other):
    """
    Returns True of self = other and False otherwise.

    EXAMPLES:
        sage: F1 = FF(5)
        sage: F2 = FF(7)
        sage: F1 == F2
        False
        sage: F2 = FF(5)
        sage: F1 == F2
        True
    """
    p = self.char()
    q = other.char()
    return p == q

def __call__(self, a):
    """
    Reduces a mod p.

    EXAMPLES:
        sage: F1(12)
        2
    """
    p = self.characteristic
    return FFElement(p, a)

def __contains__(self, a):
    """

    EXAMPLES:
        sage: F = FF(5)
        sage: 2 in F
        True
        sage: 6 in F
        False
```

```
        """
        p = self.characteristic
        if a>=0 and a<p:
            return True
        else:
            return False

class FFElement:
    def __init__(self, p, a):
        self.characteristic = p
        self.element = a%p
        self.base_field = FF(p)

    def __repr__(self):
        """
        Called to compute the "official" string representation of an object.
        If at all possible, this should look like a valid Python expression
        that could be used to recreate an object with the same value.

        EXAMPLES:

        """
        return "FFElement(%s, %s)"%(self.characteristic, self.element)

    def __str__(self):
        """
        Called to compute the "informal" string description of an object.

        EXAMPLES:

        """
        return "Finite field element %s in %s"%(self.element, self.base_field)

    def __add__(self, other):
        """
        Implements +. Assumes both self and other are instances of
        FFElement class.
```

```
    EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a+b
        FFElement(7, 4)
        FFElement(7, 5)
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        2
    """
    p = self.characteristic
    return (self.element+other.element)%p

def __sub__(self, other):
    """
    Implements -.

    EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a-b
        FFElement(7, 4)
        FFElement(7, 5)
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        6
    """
    p = self.characteristic
    return (self.element-other.element)%p

def __mul__(self, other):
    """
    Implements multiplication *.

    EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a*b
```

```
        FFElement(7, 4)
        FFElement(7, 5)
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        6
    """
    p = self.characteristic
    return (self.element*other.element)%p

def __div__(self, other):
    """
    Implements /.

    EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a/b
        FFElement(7, 4)
        FFElement(7, 5)
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        5
    """
    p = self.characteristic
    a = self.element
    b = other.element
    return (a*b.__pow__(-1))%p

def __pow__(self, n):
    """
    Implements ^ or **.

    EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a**(-1); b^2
        FFElement(7, 4)
        FFElement(7, 5)
```

```
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        2
        4
    """
    p = self.characteristic
    a = self.element
    n = int(n)
    #print "computing %s ^ %s mod %s"%(a,n,p)
    if a%p == 0 and not(n<0):
        return 0
    if p == 2 and n == -1:
        return a%p
    if n == 0:
        return 1
    if n == 1:
        return a%p
    if n>1:
        if n%2 == 0:
            return ((a.__pow__(int(n/2)))**2)%p
        if n%2 == 1:
            return (a*(a.__pow__(int(n/2)))**2)%p
    if n == -1:
        return (a.__pow__(p-2))%p
    if n<-1:
        return ((a.__pow__(-1))**(-n))%p
    return 0 # should never happen

def inverse(self):
    """
    Implements the inverse.

    EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a.inverse(); b.inverse()
        FFElement(7, 4)
        FFElement(7, 5)
```

```
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        2
        3
    """
    p = self.characteristic
    a = self.element
    if a%p == 0:
        raise ValueError, "Element must be non-zero."
    if p == 2:
        return a%p
    return (a.__pow__(p-2))%p
```

14.6 Repetition codes

Example 25. You: "Good morning."

Me: "What?"

You: "Good Morning!" (louder).

Me: "What?"

You: "GOOD MORNING!" (even louder).

Me: "Yes. Why didn't you say that the first time?"

This illustrates a "repetition code". More precisely, the **p-ary repetition code** of length n is the set of all n-tuples of the form $(x, x, ..., x)$, for $x \in GF(p)$. (We leave it as an exercise to verify that this is a vector space over $GF(q)$.) We think of x as representing information you want to send. It could be the "greyness" of a pixel in a picture or a letter (represented in ASCII code) in a word, for example. Since the channel might contain noise, we send $(x, x, ..., x)$ instead, with the understanding that the receiver should perform a "majority vote" to decode the vector. (For example, if $(0, 1, 0, ..., 0)$ was received then 0 "wins the vote").

This wasn't a very efficient example. Let's try again.

14.7 Hamming codes

Richard Hamming, while at Bell Labs in New Jersey, was a pioneer of coding theory, virtually creating the theory in a seminal paper published in 1949, Hamming codes were discovered by Hamming in the 1940's, in the days when an computer error would crash the computer and force the programmer to retype his punch cards. Out of frustration, he tried to design a system whereby the computer could automatically correct certain errors. The family of codes named after him can easily correct one error, as we will see.

14.7.1 Binary Hamming codes

For each integer $r > 2$ the *binary Hamming code* H_r is a code with $2^r - r - 1$ information bits and r redundancy bits. The Hamming code is a code of length $n = 2^r - r - 1$ which is a subspace of $GF(2)^n$ defined to be the kernel of the $r \times n$ $GF(2)$-matrix H whose columns consist of all non-zero vectors of length r. In other words, we define $C = H_r$ by specifying the check matrix of C.

There are various ways to write such a check matrix of a Hamming code, depending on how you decide to order the column vectors. Different orderings can lead to different vector spaces. If two codes differ only in the ordering of the columns of their check matrix or generator matrix then they are called *permutation equivalent codes*, or sometimes simply *equivalent codes*. If C is a Hamming code, we call any code equivalent to C a Hamming code as well.

Example 26. The binary Hamming code of length $n = 7$ has check matrix

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} = (-\,{}^tA\ I),$$

and generator matrix

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} = (I\ A),$$

for a 3×4 matrix A, where I denotes the identity matrix of the appropriate dimension.

While linear codes are not built into the standard version of Python you download from `python.org`, *many* linear codes, including all the Hamming codes, are implemented in Sage.

Sage

```
sage: C = HammingCode(3,GF(2))
sage: C.check_mat()
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
```

14.7.2 Decoding Hamming codes

Let $C = H_r$ be our Hamming code, $r > 2$. Let $\mathbb{F} = GF(2)$.

For decoding, we make the assumption that for each message sent by the sender over the noisy channel, the transmission received by the receiver contains at most one error. Mathematically, this means that if the sender transmits the codeword $c \in C$ then the receiver either received c or $c + e_i$, for some i. Here e_i is the i-th standard basis element (see (4)).

Decoding algorithm: Assume that for each message sent by the sender over the noisy channel, the transmission received by the receiver contains at most one error.

INPUT: The received vector $v \in \mathbb{F}^n$.

OUTPUT: The codeword $c \in C$ closest to v in the Hamming metric.

ALGORITHM:

- Order the columns of the check matrix H of C in some fixed way.
- Compute $s = Hv$ (this is called the *syndrome* of v).
- If $s = 0$ then v is a codeword. Let $c = v$.

  If $s \neq 0$ then $v = c + e_i$ for some codeword c and some e_i. In this case,

  $$s = Hv = H(c + e_i) = Hc + He_i = 0 + He_i = He_i$$

  is the i-th column of H. This tells us what i is. Also, this means that there was an error in the i-th coordinate of c. Let $c = v + e_i$.

- Return c.

Example 27. If the code is simply

$$C = H_2 = \{(0,0,0),(1,1,1)\} = \ker \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

and the received vector is $v = (1,1,0)$ then $H \cdot {}^t v = {}^t(0,1)$, so the error is in the 3-rd coordinate. Therefore $c = (1,1,1) \in C$ is the closest codeword.

Example 28. Let $C = H_3$, so the check matrix is given as in Example 26. If $v = (1,1,1,0,0,0,0)$ then $Hv = (1,1,1)$, which is the 4-th column of H. Thus, $c = (1,1,1,1,0,0,0)$ is the closest codeword and is the decoded version of v.

14.7.3 Non-binary Hamming codes

It actually wasn't Hamming who first constructed the non-binary generalization of his codes but M. Golay, in another very influential paper on coding theory of the late 1940's.

There is a family of Hamming codes for every finite field $\mathbb{F}$, analogous to the family constructed above for $\mathbb{F} = GF(2)$. We shall construct them for the prime fields $\mathbb{F} = GF(p)$.

Let $V = \mathbb{F}^r$ and let $V^\times$ denote the set of all vectors in V except for the 0-vector. Define the map $s : V^\times \to V^\times$ as follows.

- If $v = (v_1, \ldots, v_r) \in V^\times$ satisfies $v_1 \neq 0$ then define $s(v) = \frac{1}{v_1} v$.
- Otherwise, let $i > 1$ denote the smallest coordinate index for which $v_i \neq 0$ (so $v_{i-1} = 0$ and $0 < i \leq r$). Define $s(v) = \frac{1}{v_i} v$.

Let $S = s(V^\times)$ denote the image of this map s.

Exercise 14.2. Show that $|S| = \frac{p^r - 1}{p-1}$.

The first step to constructing the family of Hamming codes for $\mathbb{F} = GF(p)$ is to compute the set S and order it is some fixed way, writing each element as a column vector of length r,

$$S = \{s_1, s_2, \ldots, s_n\},$$

where $n = \frac{p^r - 1}{p-1}$.

The next step is to construct a matrix $r \times n$ H with entries in $\mathbb{F}$ whose columns are the elements of the set S constructed above.

Finally, let $H_r = H_r(\mathbb{F})$ denote the code whose check matrix is H:

$$H_r = \ker(H).$$

This is "the" r-th *Hamming code* over $\mathbb{F}$. (The ordering of the coordinates is not well-defined by the conditions above.)

Example 29. If $\mathbb{F} = GF(3)$ and

$$H = \begin{pmatrix} 2 & 2 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix}$$

and

$$G = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix}$$

then H is a check matrix and G is a generator matrix of

$$\begin{aligned} H_2(GF(3)) &= \{(0,0,0,0),(1,0,2,2),(2,0,1,1),(0,1,2,1),(1,1,1,0), \\ &\quad (2,1,0,2),(0,2,1,2),(1,2,0,1),(2,2,2,0)\}. \end{aligned}$$

This is implemented in Sage.

Sage

```
sage: C = HammingCode(2,GF(3))
sage: C.check_mat()
[1 0 2 2]
[0 1 2 1]
sage: C.list()
[(0, 0, 0, 0), (1, 0, 2, 2), (2, 0, 1, 1), (0, 1, 2, 1), (1, 1, 1, 0),
(2, 1, 0, 2), (0, 2, 1, 2), (1, 2, 0, 1), (2, 2, 2, 0)]
```

14.8 The Singleton bound

Let H be a check matrix for an $[n, k, d]$ code C. Let

$$H = (h_1, h_2, ..., h_n),$$

where each h_i is a column vector in $GF(q)^{n-k}$. Each $c \in C$ gives rise to a dependency relation

$$c_1 h_1 + ... + c_n h_n = 0.$$

The dependency relation with the smallest number of non-zero terms is determined from a non-zero codeword having smallest possible weight, d.

We have proven the following

Lemma 30. The positive integer d is the minimum distance of C if and only if there is some set of d columns of H which are linearly dependent but no set of $d-1$ or fewer columns are linearly dependent.

What is the largest d can be? Let X be a maximal subset of the column vectors of H which are linearly independent. Since H is full rank, $|X| = \text{rank}(H) = n-k$. The largest d can be is if d is the cardinality of some set X' of columns for which each proper subset is independent. This means, X' is at most $n-k+1$. We have proven the following

Theorem 31. (Singleton bound): $d + k \leq n + 1$.

14.9 Dual codes

Just as the row span of the generator matrix G gives rise to a code, the row span of the check matrix H should also give rise to a code. How are these two row spans related? One is the "dual" of the other.

If $C \subset GF(q)^n$ is any linear code, define the *dual code* $C^\perp$ by

$$C^\perp = \{v \in GF(q)^n \mid v \cdot c = 0 \text{ for all } c \in C\}.$$

Do some examples ...

the weight enumerator polynomial of a binary linear code specifies the number of words of each possible Hamming weight. Let be a binary linear code length n. The weight distribution is the sequence of numbers

$$A_i = \{c \in C \mid \text{wt}(c) = i\},$$

giving the number of codewords c in C having weight i as i ranges from 0 to n. The weight enumerator is the bivariate polynomial

$$W_C(x, y) = \sum_{w=0}^{n} A_w x^w y^{n-w}.$$

The MacWilliams identity states that

$$W_{C^\perp}(x,y) = \frac{1}{\mid C \mid} W_C(y-x, y+x).$$

A *Vandermonde matrix*, named after Alexandre-Théophile Vandermonde[13], is an $m \times n$ matrix

$$V = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \dots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & amp;\vdots \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{pmatrix}$$

The determinant of a square Vandermonde matrix (where $m = n$) can be expressed as:

$$\det(V) = \prod_{1 \le i < j \le n} (\alpha_j - \alpha_i). \tag{6}$$

This is called the *Vandermonde determinant*. This is a widely used result in mathematics, which we shall not prove here.

Define the *Reed-Solomon code* of order k over $GF(p)$ by

$$RS_f(p) = \{(f(1), \dots, f(p-1)) \mid f \in GF(p)[x]_k\},$$

where

$$GF(p)[x]_k = \{f \in GF(p)[x] \mid \deg(f) \le k\}.$$

Lemma 32. If $f \in GF(p)[x]_k$ has more than k distinct zeroes then $f = 0$.

proof: Let $f(x) = a_k x^k + \dots a_1 x + a_0$. If $f(r_i) = 0$ for $1 \le i \le k+1$ then we have the set of $k+1$ equations in $k+1$ unknowns

$$a_k r_i^k + \dots a_1 r_i + a_0 = 0,$$

for $1 \le i \le k+1$. This can be converted into a matrix equation

[13] A French chemist from the 1700's; see, for example, `http://en.wikipedia.org/wiki/Alexandre-Théophile_Vandermonde`.

$$\begin{pmatrix} 1 & r_1 & r_1^2 & \dots & r_1^{n-1} \\ 1 & r_2 & r_2^2 & \dots & r_2^{n-1} \\ 1 & r_3 & r_3^2 & \dots & r_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & amp;\vdots \\ 1 & r_{k+1} & r_{k+1}^2 & \dots & r_{k+1}^{n-1} \end{pmatrix} \begin{pmatrix} r_1 \\ \vdots r_{k+1} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots 0 \end{pmatrix}.$$

If the rows are distinct the the Vandermonde determinant identity (6) implies that ll the coefficients a_i must be zero. $\square$

Theorem 33. If $p+1 > k$ then the minimum distance of $C = RS_k(p)$ is greater than or equal to $n+1-k$.

proof: Let $\vec{f_1}$ denote the codeword $(f_1(1), \dots, f_1(p-1))$ and let $\vec{f_2}$ denote the codeword $(f_2(1), \dots, f_2(p-1))$, for $f_1, f_2 \in GFp)[x]_k$. Suppose $d(\vec{f_1}, \vec{f_2}) < n+1-k$, in order to get a contradition. In this case, there are at least $k+1$ "points" i for which $f_1(i) = f_2(i)$. Therefore, the polynomial $f_2 - f_1$ has $> k$ zeros. The previous lemma implies $f_1 = f_2$ $\square$

Corollary 34. *If $p+1 > k$ then $C = RS_k(p)$ is an MDS code.*

14.10 Reed-Muller codes

Let $m > 1$ be an integer and let P_1, P_2, ..., P_n denote all the points in the set $\mathbb{F}^m$. For any integer r, $1 \le r \le m(p-1)$, let

$$\mathbb{F}[x_1, \dots, x_m]_r$$

denote the vector space over $\mathbb{F}$ of polynomials in the x_i of total degree $\le r$.

Definition 35. The *r-th order generalized Reed-Muller code* $RM_{\mathbb{F}}(r, m)$ of length $n = p^m$ is the vector space of all vectors of the form $(f(P_1), f(P_2), \dots, f(P_n))$, where $f \in \mathbb{F}[x_1, \dots, x_m]_r$.

In other words, $RM_{\mathbb{F}}(r, m)$ is the image of the evaluation map

$$\text{eval} : \mathbb{F}[x_1, \dots, x_m]_r \to \mathbb{F}^n,$$

defined by

$$\text{eval}(f) = (f(P_1), f(P_2), \dots, f(P_n)).$$

This is implemented in Sage but only in the binary case.

Sage

```
sage: C = BinaryReedMullerCode(2,4); C
Linear code of length 16, dimension 11 over Finite Field of size 2
sag: C.check_mat()
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

15 Cryptography

Cryptography is the study and practice of methods of secure communication. Though in the days of Cesear, secret communication amounted to very simple methods, modern cryptography required knowledge of extremely advanced and sophisticated mathematical techniques. In this section, only a few of the simplest (but relatively common) crytposystems will be discussed.

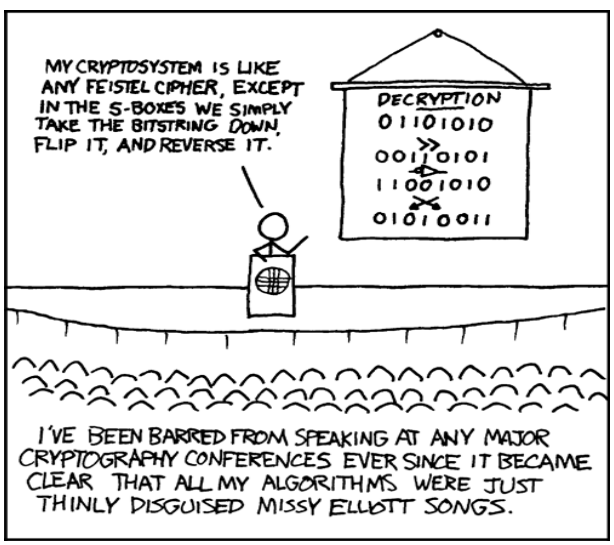


Figure 19: **Cryptography .**
`xkcd` license: Creative Commons Attribution-NonCommercial 2.5 License, `http://creativecommons.org/licenses/by-nc/2.5/`

Let A be a finite set, which we call the *alphabet* (typically, $A = \mathbb{F}$ is a

finite field, such as $GF(p)$ for some prime p), and let M be the set of all finite sequences of elements of A, which we call the *message space*. A *cipher* is a mapping

$$E : M \to M$$

called *encryption*, and an inverse mapping $D : M \to M$ called a *decryption*, which satisfy $D(E(m)) = m$ for all $m \in M$. The messages in the range of E are called the *cipher text* and the domain of E is called the *message text*.

15.1 Linear feedback shift register sequences

One type of cipher is the following. Suppose that your alphabet is $GF(2) = \{0, 1\}$ and that the message space M is as above. Let $r = (r_1, r_2, \dots)$ be an infinite sequence of random elements of A. Define the encryption map $E : M \to M$ by $E(m) = m + r$, where addition is componentwise modulo 2. Since r is a random sequence, any eavesdropper would think the received message is random as well. Define the decryption map $D : M \to E$ by $D(m) = m + r$, where again addition is componentwise modulo 2. This is called a *one time key pad cipher* and r is called the *key*.

This is a wonderful cryptosystem. There is just one problem. How do we construct a random sequence in a practical way that both the sender (for encoding) and the receiver (for decoding) have a copy?

Linear feedback shift registers are one way to try to solve that problem.

Definition 36. Let p be a prime, $k > 1$ be an integer, and let $a_1, ..., a_k$ are given elements of $GF(p)$. A *linear feedback shift register sequence* (LFSR) modulo p of *length* k is a sequence $s_1, s_2, ...$ such that $s_1, ..., s_k$ are given and

$$s_{k+i} = a_1 s_i + a_2 s_{i+1} + ... + a_k s_{k+i-1} \ , \ i > 0,$$

where addition and multiplication is performed over $GF(p)$.

An equation such as this is called a *recursion equation* of length k modulo p.

Example 37. The Fibonacci sequence is an example of a recursion equation of length 2 over the integers. However, you can also reduce each of the elements in the series modulo p, or simply compute the recursive equations modulo p, to get a LFSR of length 2 modulo p.

The sequence

$$f_{n+1} = f_n + f_{n-1}, \qquad f_0 = 0,\ f_1 = 1,$$

over $GF(3)$ is

$$0, 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2, 0, 2, 2, 1, \ldots\ .$$

Notice that this Fibonacci sequence mod 3 seems to be periodic with period 8. This will be explained below.

Though LFSR ciphers are rather easy to break (see for example T. Brock [Br]), they are still used today in bluetooth devices (`http://en.wikipedia.org/wiki/E0_(cipher)`), among other things.

15.1.1 Linear recurrence equations

Suppose that a_0, a_1, $\ldots$, a_{k-1} are given integers. The general method for solving a recurrence equation of the form

$$s_{k+i} = a_0 s_i + a_1 s_{i+1} + ... + a_{k-1} s_{k+i-1}, \qquad i > 1,$$

with $s_1, s_2, ..., s_k$ given, is rather simple to describe (in principle - in practice it may be quite hard).

First, "guess" $s_n = cr^n$, where c and r are constants. Substituting into the recursion relation and simplifying, we find that c can be arbitrary but r must satisfy

$$a_0 + a_1 r + ... + a_{k-1} r^{k-1} - r^k = 0.$$

Let $r_1, ..., r_k$ be the roots of this polynomial. We shall *assume that these roots are distinct*. Under these conditions, let s_n be an arbitrary linear combination of all your "guesses",

$$s_n = c_1 r_1^n + c_2 r_2^n + ...c_k r_k^n.$$

Recall that $s_1, ..., s_k$ are known, so we have k equations in the k unknown $c_1, ..., c_k$. This completely determines s_n.

Example 38. Let s_n satisfy $s_n = s_{n-1} + s_{n-2}$ and let $s_1 = 1, s_2 = 1$.

We must solve $r^2 - r - 1 = 0$, whose roots are $r_1 = \frac{1+\sqrt{5}}{2}$ and $r_2 = \frac{1-\sqrt{5}}{2}$. Therefore,

$$s_n = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n, \qquad n > 0.$$

Since $s_1 = 1$ and $s_2 = 1$, we have

$$s_n = 5^{-1/2} r_1^n - 5^{-1/2} r_2^n.$$

The recurrence equation

$$s_{k+n} = a_0 s_n + a_1 s_{n+1} + ... + a_{k-1} s_{k+n-1}, \qquad n > 1, \tag{7}$$

is equivalent to the matrix equation

$$\begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 1 \\ \vdots & & \dots & & \\ 0 & 0 & \dots & 0 & 1 \\ a_0 & a_1 & \dots & & a_{k-1} \end{pmatrix} \begin{pmatrix} s_n \\ s_{n+1} \\ \vdots \\ s_{n+k-1} \end{pmatrix} = \begin{pmatrix} s_{n+1} \\ s_{n+2} \\ \vdots \\ s_{n+k} \end{pmatrix},$$

where s_{n+k} is given as above.

15.1.2 Golumb's conditions

S. Golomb indtroduced a list of three statistical properties a sequence of numbers $A = \{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$, should display for it to be considered "random". Define the *autocorrelation* of A to be

$$C(k) = C(k, A) = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} (-1)^{a_n + a_{n+k}}.$$

In the case where A is periodic with period P then this reduces to

$$C(k) = \frac{1}{P} \sum_{n=1}^{P} (-1)^{a_n + a_{n+k}}.$$

Assume A is periodic with period P.

- *balance*: $|\sum_{n=1}^{P} (-1)^{a_n}| \leq 1$.
- *low autocorrelation*: For some "small" constant $\epsilon > 0$, the autocorrelation[14] satisfies, for $0 \leq \ell \leq P - 1$,

[14]Not everyone defined the autocorrelation this way, but this definition is useful for sequences of elements in $GF(2)$.

$$C(\ell) = \begin{cases} 1, & \ell = 0, \\ \epsilon, & \ell \neq 0. \end{cases}$$

(For sequences satisfying these first two properties, it is known that $\epsilon = -1/P$ must hold.)

- *proportional runs property*: In each period, about half the runs have length 1, one-fourth have length 2, and so on. Moveover, there are about as many runs of 1's as there are of 0's.

Example 39. The $GF(2)$-version of the Fibonacci sequence is

$$\{f_n\} = \{0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, \dots\}.$$

The period is $P = 3$, and so autocorrelation is

$$C(0) = \frac{1}{3}[(-1)^{f_1+f_1} + (-1)^{f_2+f_2} + (-1)^{f_3+f_3}] = \frac{1}{3}[(-1)^0 + (-1)^0 + (-1)^0] = 1,$$

$$C(1) = \frac{1}{3}[(-1)^{f_1+f_2} + (-1)^{f_2+f_3} + (-1)^{f_3+f_4}] = \frac{1}{3}[(-1)^0 + (-1)^1 + (-1)^1] = -1/3,$$

$$C(2) = \frac{1}{3}[(-1)^{f_1+f_3} + (-1)^{f_2+f_4} + (-1)^{f_3+f_5}] = \frac{1}{3}[(-1)^1 + (-1)^0 + (-1)^1] = -1/3.$$

Therefore, it has "low autocorrelation." It is "balanced":

$$|\sum_{n=1}^{3} (-1)^{f_n}| = |(-1)^1 + (-1)^1 + (-1)^0| = 1 \leq 1.$$

In a period, $\{0, 1, 1\}$, we have 1 run of length 1 and one run of length 2. For period 3, this is the best we can do to try to satisfy the "proportional runs property."

This verifies Golomb's statistical conditions in this example.

This can also be partially done in Sage.

Sage

```
sage: F = GF(2); l = F(1); o = F(0)
sage: fill = [o,l]; key = [1,l]; n = 20
sage: lfsr_sequence(key, fill, n)
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1]
```

The theorem's below, due to Golomb, tell us how easy it is to construct such random-looking sequences.

Theorem 40. Let $S = \{s_i\}$ be defined as above, (7). The period of S is at most $p^k - 1$. It's period is exactly $P = p^k - 1$ if and only if the characteristic polynomial of

$$A = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 1 \\ \vdots & & \dots & & \\ 0 & 0 & \dots & 0 & 1 \\ a_0 & a_1 & \dots & & a_{k-1} \end{pmatrix},$$

is irreducible and primitive[15] over $GF(p)$.

The notion of a primitive polynomial goes beyond this course, but examples will be given below.

A related result is the following fact, though it is only stated in the binary case.

Theorem 41. *If $C = \{c_n\}_{n=1}^{\infty}$ are the coefficients of $f(x)/g(x)$, where $f, g \in GF(2)[x]$ and $g(x)$ is irreducible and primitive. Then C is periodic with period $P = 2^d - 1$ (where d is the degree of $g(x)$) and satisfies Golomb's randomness conditions.*

Example 42. Consider the $GF(2)$ polynomial $f(x) = x^{16}+x^{14}+x^{13}+x^{11}+1$, which is the characteristic polynomials of the matrix

[15] A polynomial $f(x)$ of degree m with coefficients in $GF(p)$ is a *primitive polynomial* if it has a root α in $GF(p^m)$ such that $\{0, 1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^{p^m-2}\}$ is the entire field $GF(p^m)$, and moreover, $f(x)$ is the smallest degree polynomial having α as root. Roughly speaking, think of primitive as being a "nice" irreducible polynomial.

$$A = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1
\end{pmatrix}.$$

This polynomial $f(x)$ is, according to Sage, irreducible and primitive.

Sage

```
sage: R.<x> = PolynomialRing(GF(2),"x")
sage: R
Univariate Polynomial Ring in x over Finite Field of size 2 (using NTL)
sage: f = x^16 + x^14 + x^13 + x^11 + 1
sage: f.is_irreducible()
True
sage: f.is_primitive()
True
```

Remark 1. For polynomials of such relatively high degree, using an open-source mathematical software system like Sage can be very useful. Since Sage is open-source, you can check the `is_primitive` algorithm yourself if there is any doubt that it is correct. In fact, since the source code for the Sage implementation of the `is_primitive` algorithm is available for anyone to read, it is likely that many *others* already have checked it over. Though these two facts may give you greater confidence that Sage's `is_primitive` is

correct, it is a general principle that *all* software has bugs. Therefore, it is a healthy attitude to be skeptical of *all* computer programs. They are written by humans and humans make mistakes.

15.1.3 Exercises

Exercise 15.1. *Verify all the conditions of Golomb's tests for the degree* 16 *polynomial in Example 42.*

Exercise 15.2. *Is the Fibonacci sequence mod p periodic for other values of p? If so, find the periods for $p = 5$ and $p = 7$. Do you see a pattern?*

Exercise 15.3. *Find the characteristic polynomial associated to the Fibonacci sequence modulo* 2*. Is it irreducible and primitive?*

Exercise 15.4. *Think about how to generalize Golomb's statistical conditions to a LFSR over $GF(p)$. What would your conditions be?*

15.2 RSA

RSA was publicly described in 1978 by Ron Rivest, Adi Shamir, and Leonard Adleman, though it was discovered many years earlier by a researcher at GCHQ named Clifford Cocks as part of classified work (declassified in 1997). is one of the most popular cryptosystems used today. It has a small key-size given the data that it can encrypt, and appears to be fairly secure. There is a company, RSA Labs, which issued several challenge problems worth up to $200000. However, in 2007 the challenge was ended and the prizes were retracted for the remaining unsolved problems (`http://en.wikipedia.org/wiki/Rsa_challenge`).

RSA involves a public key and a private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. Even though the public key and the private key are mathematicaly related, the security of the RSA cryptosystem relies on that belief that it is computationally infeasible to compute the private key from the public key.

- Choose two distinct prime numbers p and q.
- Compute $n = pq$.

  n is used as the modulus for both the public and private keys

- Compute $\phi(pq) = (p-1)(q-1)$, where ϕ is Euler's totient function.
- Choose an integer e such that $1 < e < \phi(pq)$, and e and $\phi(pq)$ are relatively prime).

  e is released as the *public key* exponent.
- Determine d (using modular arithmetic) which satisfies the congruence relation $de \equiv 1 \pmod{\phi)(pq}$.

  This is often computed using the extended Euclidean algorithm.

  d is kept as the *private key* exponent.

The *public key* consists of the modulus n and the public (or encryption) exponent e. The *private key* consists of the modulus n and the private (or decryption) exponent d which must be kept secret.

Encryption algorithm: Alice and Bob want to send messages to each other. We assume the existence of Eve, an evil-hearted evesdropper, who knows RSA and the public key.

Alice transmits her public key (n, e) to Bob and keeps the private key secret. Bob wishes to send a message m to Alice, an integer $0 < m < n$. He then computes the ciphertext c corresponding to: $m^e \equiv c \pmod{n}$. Bob transmits c to Alice.

Decryption algorithm: Alice can recover m from c by using her private key exponent d by the following computation:

$$c^d \equiv (m^e)^d \equiv m^{ed} = m^{1+k\phi(n)} = m \cdot (m^k)^{\phi(n)} \equiv m \pmod{n},$$

by Euler's Theorem (`http://en.wikipedia.org/wiki/Euler's_theorem`).

Example 43. Let $p = 1009$ and $q = 1013$, so $n = pq = 1022117$. Therefore $\phi(n) = 1020096$. Select $e = 123451$, so we compute $d = 300019$. If the message is $m = 46577$ then we transmit the ciphertext $c = 622474$.

This can be done using Sage as well.

Sage

```
sage: p = next_prime(1000)
sage: q = next_prime(1010)
sage: n = p*q
sage: n
1022117
sage: k = euler_phi(n)
```

```
sage: e = 123451 # a random integer in [1,k-1]
sage: k; xgcd(k, e)
1020096
(1, -36308, 300019)
sage: x = xgcd(k, e)[1]
sage: y = xgcd(k, e)[2]
sage: d = y%k
sage: y*e%k; d*e%k
1
1
sage: m = randint(100, k); m
46577
sage: c = power_mod(m,e,n) # faster than m^e%n
622474
sage: power_mod(c,d,n) # the same as m, collaborating m was correctly decrypted
46577
```

15.3 Diffie-Hellman

We've looked at RSA, which seems to be a good method of sending messages secretly. However, RSA requires that a private key be transmitted secretly. How is that to be accomplished in a practical way? One method for solving this problem was suggested by Whitfield Diffie and Martin Hellman in 1976.

Here's a description of their protocol.

- Alice and Bob agree on a finite cyclic group G and a generating element $g \in G$. (This is usually done long before the rest of the protocol; g is assumed to be known by all attackers.) We will write the group G multiplicatively. Assume G has order n.
- Alice picks a random natural number a, $1 < a < n$, and sends g^a to Bob.
- Bob picks a random natural number b, $1 < b < n$, and sends g^b to Alice.
- Alice computes $(g^b)^a$.
- Bob computes $(g^a)^b$.
- Both Alice and Bob are now in possession of the group element g^{ab}, which can serve as the *shared secret key*.

Example 44. Let $G = (\mathbb{Z}/101\mathbb{Z})^\times$, $g = 3$, an element of order $n = |G| = 100$. Alice picks $a = 35$ and Bob picks $b = 36$. Alice computes $g^a = 44$ and Bob computes $g^b = 31$. The commonly shared key is $g^{ab} = 36$.

This can be done using Sage as well.

Sage

```
sage: G = IntegerModRing(101)
sage: g = G.random_element()
sage: g; g.multiplicative_order()
3
100
sage: a = randint(1,50)
sage: b = randint(1,50)
sage: a; b
35
36
sage: ga = g^a
sage: gb = g^b
sage: ga; gb
44
31
sage: ga^b; ga^b == gb^a
36
True
```

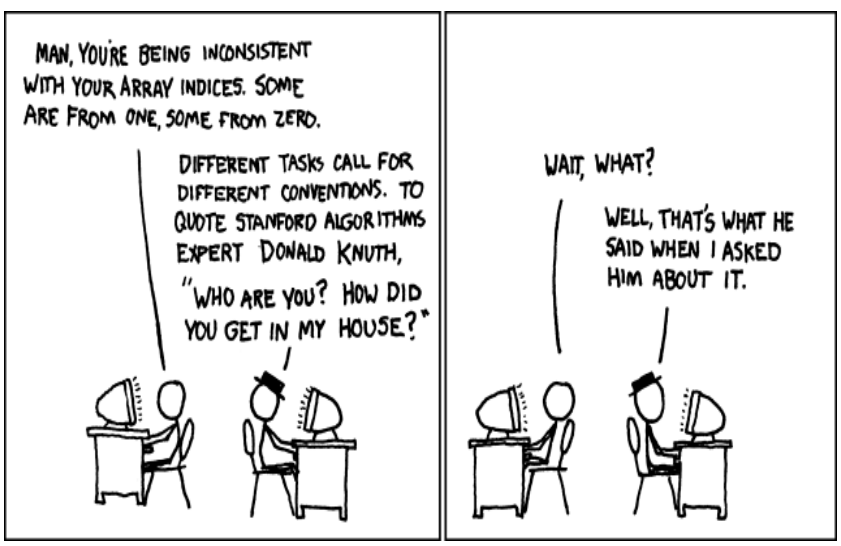


Figure 20: **Donald Knuth .**
xkcd license: Creative Commons Attribution-NonCommercial 2.5 License, http://creativecommons.org/licenses/by-nc/2.5/

16 Matroids

Matroid theory generalizes ideas of linear algebra and graph theory. A good reference is Oxley's fine book [O]. These "discrete" objects are excellent examples of what can be implemented using Python's class structure. They also generalize linear codes so fit nicely into this topic.

First, what is a matroid?

Definition 45. A finite *matroid* M is a pair (E, J), where E is a non-empty finite set and J is a collection of subsets of E (called the *independent sets*) with the following properties:

- The empty set is independent, i.e., $\emptyset \in J$.
- (the *hereditary property*) Every subset of an independent set is independent, i.e., for each $E' \subset E$, $E \in J$ *implies* $E' \in J$.
- (the *augmentation property* or the *independent set exchange property*) If A and B are two independent sets in J and A has more elements than B, then there exists an element in A which is not in B that when added to B still gives an independent set.

It can be shown that if $M_1 = (E, J_1)$ is a matroid on the set E and $M_2 = (E, J_2)$ is also a matroid on E then $|J_1| = |J_2|$. This cardinality is called the *rank* of the matroid.

If $M = (E, J)$ is a matroid then any element of J that has maximal possible cardinality is called a *base* of M.

If matroids generalize graphs, can you draw them? If so, what do they look like? A related question: How do you construct them? If we know how to construct them, perhaps we can "picture" that construction somehow.

- If E is any finite subset of a vector space V, then we can define a matroid M on E by taking the independent sets of M to be the linearly independent elements in E. We say the set E represents M.

  Matroids of this kind are called *vector matroids*.

  A matroid that is equivalent to a vector matroid, although it may be presented differently, is called *representable*. If M is equivalent to a vector matroid over a field F, then we say M is *representable over* F.

- Every finite graph (or multigraph) G gives rise to a matroid as follows: take as E the set of all edges in G and consider a set of edges independent if and only if it does not contain a simple cycle. This is called the *graphic matroid* of G.

16.1 Matroids from graphs

Let $\Gamma = (V, E)$ denote a graph. The matroid $M = (E, J)$ associated to Γ is obtained by taking the matroid E to be the same set as the graph E (i.e., the edges of the graph), and taking as a base for J the set of spanning forests[16] of Γ. An element of J, the set of independent elements of the matroid, is simply a forest in Γ. In the case then Γ is connected, this means that the base for the matroid associated to Γ is the set of all spanning trees of Γ.

Example 46. First, consider the graph in Figure 6 in §6.1. The matroid $M = (E, J)$ is fairly large. Indeed, merely the base for J has nearly 300 elements!

Sage

```
sage: graph_dict = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,2], 4: [0,1],
  5: [7, 6], 6: [2], 7: [2]}
sage: G = Graph(graph_dict); G
Graph on 8 vertices
sage: G.spanning_trees_count()
290
```

Let us consider a much smaller example.

Example 47. Consider the cycle on 3 vertices.

[16] Recall a forest in a graph is simply a subgraph which contains no cycles.

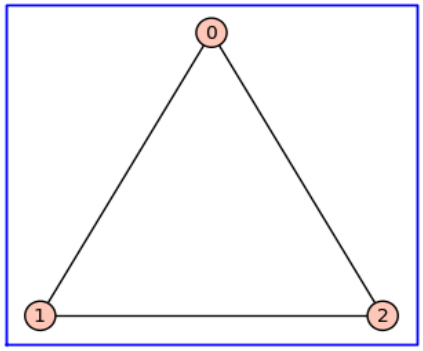
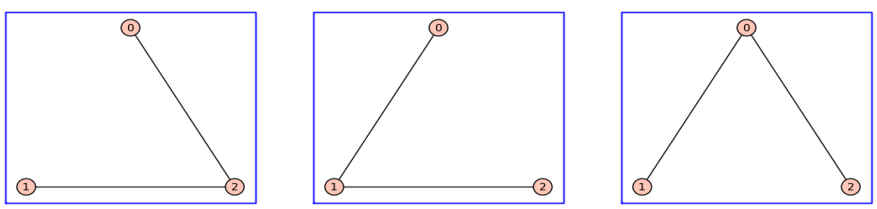


Figure 21: **A cycle on** 3 **vertices .**

What is the matroid associated to this graph? Here are the spanning trees in the graph:

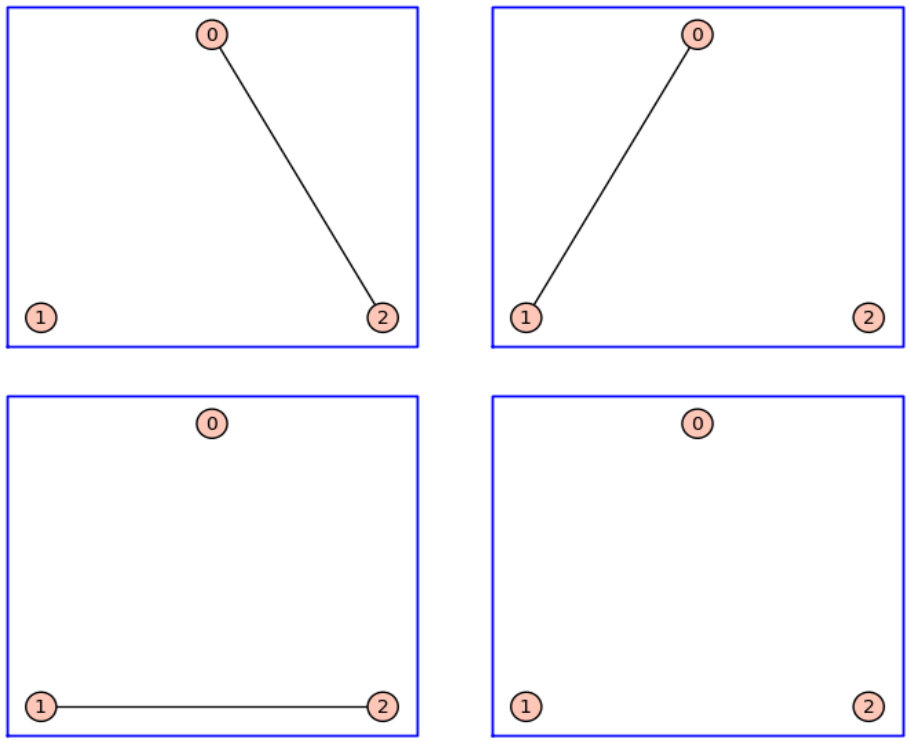


These form a base for the independent sets J. This count agrees with what Sage says as well:

Sage

```
sage: graph_dict = {0: [1,2], 1: [0,2], 2: [0,1]}
sage: G = Graph(graph_dict); G
Graph on 3 vertices
sage: G.spanning_trees_count()
3
```

The rest of the elements of J are the four graphs listed below.



16.2 Matroids from linear codes

Let C be a linear code over a finite field $\mathbb{F}$ and G a generator matrix. Let E be the set of all columns of G. This defines a matroid M representable over $\mathbb{F}$.

Example 48. If C is the binary linear code having generator matrix

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

then the set of subsets of the column indices which correspond to independent columns are

$$\begin{aligned} J = \quad & \{\{\}, \{0\}, \{0,1\}, \{0,1,2\}, \{0,1,4\}, \{0,2\}, \{0,2,3\}, \{0,3\}, \{0,3,4\}, \{0,4\}, \\ & \{1\}, \{1,2\}, \{1,2,3\}, \{1,2,4\}, \{1,3\}, \{1,3,4\}, \{1,4\}, \{2\}, \{2,3\}, \{2,3,4\}, \\ & \{2,4\}, \{3\}, \{3,4\}, \{4\}\}, \end{aligned}$$

according to Sage. This set J is the set of independent sets of M and the subset of J consisting of the 3-tuples is the set of bases of M. Here is the program used to compute J.

Python

```
def independent_sets(mat):
    F = mat.base_ring()
    n = len(mat.columns())
    k = len(mat.rows())
    J = Combinations(n,k)
    indE = []
    for x in J:
        M = matrix([mat.column(x[0]),mat.column(x[1]),mat.column(x[2])])
        if k == M.rank():     # all indep sets of max size
            indE.append(x)
            for y in powerset(x): # all smaller indep sets
                if not(y in indE):
                    indE.append(y)
    return indE
```

Of course, if S is an element of J then any subset of S is also independent.

Question: Is this matroid the matroid of a graph? If so, can you construct it?

17 Class projects

These are just suggestions. Just ask if you have strong interest in working on something different. You can also look for ideas in the course textbook Biggs [B].

All programs submitted must be released under an open-source license. If you write all the programs yourself, with no resources used, then they are in the public domain, since you are a U.S. government employee and this is part of your official duties. If you use or modify someone elses code then you must use code with an open-source GPL-compatible license. (For example, MIT license, GPLv2+, Python license, and many others.) A copyright and license statement must be included with your submitted code.

1. Gray codes. Cite and explain connections with/applications to campanology, Hilbert space curves, Hamiltonian paths in a graph, the

Tower of Hanoi, and electrical engineering. Implement versions versions and analyze them and test them for speed.

References:

- Steve Witham *Hilbert Curves in More (or fewer) than Two Dimensions*
  `http://www.tiac.net/~sw/2008/10/Hilbert/`
- Gray codes Wikipedia
  `http://en.wikipedia.org/wiki/Gray_code`
- David Joyner and Jim McShea *Gray codes*
  `http://www.usna.edu/Users/math/wdj/gray.html`
- *Application: Bell ringing*, a section in **Applied Abstract Algebra**, D. Joyner, R. Kreminski, J. Turisco, Johns Hopkins Univ. Press, 2002.
  `http://www.usna.edu/Users/math/wdj/book/node158.html`
- J. H. Conway, N. J. A. Sloane and Allan R. Wilks, *Gray Codes for Reflection Groups*
  `http://www2.research.att.com/~njas/doc/wilks.html`

2. Reed-Muller codes. Implement them as generally as possible. Discuss history and applications.

3. Implement the Tanner graph of an error-correcting code
   `http://www.usna.edu/Users/math/wdj/book/node204.html`
   `http://en.wikipedia.org/wiki/Tanner_graph`

4. Huffman codes.

   Implement Huffman codes in Sage. Discuss connection with information theory and other compression codes. Is there a relationship with efficiency of google computer searches?

   Note this: `http://en.wikipedia.org/wiki/Huffman_codes#History`

5. Cryptography. Some possible examples.

   - (Hard?) Implement a feedback with carry shift register stream cipher.
     `http://www.math.ias.edu/~goresky/EngPubl.html`
     `http://www.cs.uky.edu/~klapper/algebraic.html`

- (Hard?) The Biggs cryptosystem using graph theory, chip firing games and Diffie-Hellman.

  Reference:

  [B1] Simon R. Blackburn, *Cryptanalysing the critical group: efficiently solving Biggs's discrete logarithm problem*,
  `http://eprint.iacr.org/2008/170`

  [B2] ——, *Group Theory and Cryptography*
  `http://personal.rhul.ac.uk/uhah/058/talks/bath2009.pdf`

  [S] F. Shokrieh, *Discrete logarithms on the Jacobian of finite graphs*, pdf version avaialble on the internet, arXiv:0907.4764v1

6. Tower of Hanoi. Can you think of a Python class structure which would help model this puzzle? See the slides by S. Dorée.

   S. Dorée, *The graphs of Hanoi*, Portland Area Lecture Series (PALS), November 19, 2009.

7. Social network analysis and graph theory.

   - Implement the Havel-Hakimi algorithm in Sage. (More prescisely, write an interface to the implementation in `NetworkX`; please ask me for details and help.)
   - Look at a specific model, such as `http://en.wikipedia.org/wiki/Watts_and_Strogatz_model`, and implement it in Sage. Others:

     BarabásiAlbert model
     `http://en.wikipedia.org/wiki/BA_model`

     ErdösRényi model
     `http://en.wikipedia.org/wiki/Erdos-Renyi_model`

8. Crowd dynamics. Implement a simulated bomb evaculation of a rectangular room using Python, graphs, and Markoff processes. (For specific suggestions, see me. A vaguely similar project is discussed in lectures 17-19 in [GG].)

18 Labs and tests

18.1 Computer Lab 1

Exercises for lab 1.

To be handed in!

1. Create a hello world program using string concatenation

   ```
   c = "hello World!!"
   print c

   def hello():
       a = "Hello"
       b = " World!"
       c = a+b
       return c

   hello()
   ```

   Note `c` is different "inside" the program than "outside."

2. What is wrong with this statement?

   ```
   as = 5
   ```

3. ```
   def hello(name = 'Mom'):
       a = "Hello"
       b = name
       c = a+b
       return c
   ```

   How do you get this program to return 'Hello World!'?

   (Hint: Look at # 5 at `http://wiki.python.org/moin/SimplePrograms`.)

4. Type

   ```
   1/3
   ```
   ```

into python 2.5 or 2.6, then type

```
1/3
```

into python 3.1. What is the difference?

5. Type

```
a = 1/10
print a
```

and

```
a = 1/10
print(a)
```

into both python 2.6 and python 3.1. What is the difference?

6. Type

```
a = 0.1
a
print a
```

into python 2.6 and 3.1. What is the difference?

7. Type

```
range(10)
s = 0
for i in range(101):
    s = s+i
s
```

What is `s`?

8. Type

```
s = sum([i for i in range(101)])
s
```

What is `s`?

9. Type

```
4%2
5%2
4%3
5%3
s = sum([i for i in range(101) if i%2 == 0])
s
```

What does `%` means? What is `s`?

10. Type

```
s = sum([i for i in range(101) if i%2 == 0 and i%3 == 0])
s
```

What is `s`?

11. Sign up at `http://projecteuler.net/` and create an account. Write a program that solves problem 1.

18.2 Computer Lab 2

1. Create a companion matrix program in Sage by building the matrix row-by-row using the list append command.

```
def companion_mat(L):
    k = len(L)
    rows = []
    for i in range(k-1):
        r = [0]*k
        r[i+1] = 1
        rows.append(r)
    rows.append(L)
    return matrix(rows)
```

What is the companion matrix C of [1,1]? Of $[2, 3, 4]$?

2. What is the characteristic polynomials of the companion matrix of $[1, 1]$? Of $[2, 3, 4]$? (if `C = companion_mat(L)`, compute `C.charpoly()` in Sage.)

3. What are the roots of the characteristic polynomials of the companion matrix of $[1, 1]$? (If `f = C.charpoly()`, use Sage's `f.real_roots()` or `f.complex_roots()`.) Do you recognize them?

4. What is C^{10}? What is the 10-th Fibonacci number?

5. Let $s_0, s_1, ...$ an infinite sequence and a, b be fixed. Show that

$$s_{n+1} = bs_n + as_{n-1}$$

if and only if

$$\begin{pmatrix} s_n \\ s_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix} \begin{pmatrix} s_{n-1} \\ s_n \end{pmatrix}.$$

6. Use this recursion relation to compute

$$\begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^{16} \begin{pmatrix} s_0 \\ s_1 \end{pmatrix}.$$

7. Now take $a = b = 1$ and assume $s_0 = 0$, $s_1 = 1$. (This infinite sequence $\{s_n\}$ is now the Fibonacci sequence.)

   How many computations does it take to compute f_{1024}?

   Each matrix multiplication takes 8 scalar multiplications (actually only 7, thanks to an extremely clever algorithm due to Strassen, but we omit

the complicated details - use google to search "Strassen algorithm" if you are interested in details).

This gives 1024×8 (or 1024×7) multiplications. Right?

Wrong!

First, compute $\begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^2$. Next compute $(\begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^2)^2 = \begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^4$. Next compute $(\begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^4)^2 = \begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^8$. ... Finally, compute $(\begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^{512})^2 = \begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix}^{1024}$. What is the total number of multiplications needed to compute the last quantity?

Answer:

8. In general, the **Repeated Squaring Algorithm** says that to compute a^n you perform the following procedure.

$$a^n = \begin{cases} 1, & \text{if } n = 0 \\ a^{n-1}a, & \text{if } n \text{ is odd} \\ (a^{n/2})^2, & \text{if } n \text{ is even} \end{cases}$$

Here is another version.

- Compute the binary representation of n:

$$n = b_0 \times 2^0 + b_1 \times 2^1 + \cdots + b_m \times 2^m,$$

  where $m = [\log_2(n)]$. Here $b = [b_m, \ldots, b_0]$ (or $[b_0, \ldots, b_m]$, depending on how you write it) is the binary representation of n.
- Compute the m numbers a, a^2, a^4, a^8, $\ldots$, a^{2^m}.
- Compute $a^n = \prod_{i, \text{ such that } b_i \neq 0} a^{2^i}$.

9. Use the `bin` command in Python to convert the following numbers to binary: 4, 5, 32, 33, 2048, 2049.

How many steps (muliplications, including repeated squarings) does it take to compute 3^{2049} using this algorithm?

10. Can you implement the repeated squaring algorithm in Python (Sage has this implemented automatically already)?

    Use the template below and fill it in with Python or Sage commands.

```
def power(a,n):
    """
    your docstring...

    """
    p = 1
    #define p = a^n using above algorithm...









    return p
```

11. Multiplying $n \times (n-1)$, using "long multiplication" from elementary school, takes about $\log_2(n)^2 = O(\ln(n)^2$ multiplications. (There are faster algorithms, but this is the one you are most used to.)

    Write 37 and 75 as "binary polynomials":

    $$37 = ____ \times 1 + ____ \times 2 + ____ \times 2^2 + \cdots + ____ \times 2^5$$

    and

    $$75 = ____ \times 1 + ____ \times 2 + ____ \times 2^2 + \cdots + ____ \times 2^6.$$

    How many multiplications are needed to compute 37×75 this way?

12. Can other recursive procedures be computed quickly as well?

    Suppose that $s_0 = 1$ and $s_n = ns_{n-1}$ for $n > 0$. What is s_{10}? s_n?

18.3 Computer Lab 3

1. Use the following (or using Python's `xor`) programs in the assignment below.

```
def int2binary(m, n):
    """
    returns "binary list" of length n obtained
    from the binary repr of m, padded by 0's
    (on the left) to length n.
    """
    s = bin(m)
    k = len(s)
    b = [0]*n
    for i in range(2,k):
        b[n-k+i] = int(s[i])
    return b

def binary2int(b):
    """
    inverts int2binary

    """
    k = len(b)
    n = sum([int(b[i])*2**(k-1-i) for i in range(k)])
    return n
```

Write the following program:

```
def add_vectors_mod_m(L1, L2, m):
    """
    Adds two lists of the same length modulo m,
    using componentwise addition.

    INPUT:
        L1 - integer list of length n
        L2 - integer list of length n
        m  - integer >1.
```

```
    OUTPUT
        L1+L2 mod m
    """
    #write your program here.
```

Now use the program below to compute the 5th Gray codeword in the reflected Gray code of length 4: _____

```
def graycodeword(m, n):
    '''
    returns the mth codeword in the reflected binary Gray code
    of length n.

    '''
    return add_vectors_mod_m(int2binary(m,n), int2binary(int(m/2),n), 2)
```

2. You can XOR two positive integers a, b ($1 \leq a, b \leq 2^n - 1$) using

```
binary2int(add_vectors_mod_m(int2binary(a,n), int2binary(int(b),n), 2))
```

or simply

```
import operator
operator.xor(a,b)
```

Find 76 XOR 89 = _____ .

3. Try to decrypt this message:

$$[3, 11, 68, 10, 5, 18, 29, 69].$$

This is encoded by first converting to ASCII (using Python `chr` and `ord`) then XORing with a single lower-case character (called the *key*). (This is like Project Euler 59 but just XORs with a single character, rather than a 3-letter word.)

4. Now see if you can decode the message in Project Euler problem 59

   `http://projecteuler.net/index.php?section=problems&id=59`

18.4 Computer Lab 4

The Python class for finite fields $GF(p)$, p prime, is given in §14.5.1 above. Make your own class that implements the class `FFVectorSpace` and `FFVectors`. The vector space class must be able to take a prime p (for the characteristic) and an integer n (for the dimension) as arguments. The vectors class must be able to take a prime p, an integer n and a list of length n of integers (for the coordinates of the vector) as arguments. You must implement vector addition and subtraction. However, scalar multiplication is extra credit. Document your code with standard Python docstrings.

18.5 Take-home Test 1

You may use class notes, class text, Python books or the internet, but please reference your use with appropriate detail. Work on your own and no *serious* discussion (questions like "Did you finish Problem 2 yet?" are okay) of the exam with others until they are all handed in.

1. Write a Python program to convert fahrenheit to celcius. Document your code with examples and references as in §9.2 and §9.4 of the notes (using Wikipedia is okay).

2. To start, in Python, define `A = [2, 3, [4, 5], 6]`, then define `B = A` and `C = copy(A)`. (You may need to import the `copy` command using `from copy import *` or `import copy`.)

   - Set `C[2] = 1`. What is `A, B, C`?
   - Start over[17]. Set `B[2] = 1`. What are `A, B, C`?
   - Start over. Set `A[2] = 1`. What are `A, B, C`?
   - Start over. Set `C[2][1] = 1`. What are `A, B, C`?

3. Using the ideas in §8.3.2, write a program to compute the 12-th Lucas number, as defined in §8.3.3.

[17] This means, redefine `A<B,C` as above, not to "manually reset values."

4. Write a program `collatz` which has

   `INPUT:` n - integer ≥ 1

   `OUTPUT:` An integer given by

$$\begin{array}{ll} 1, & \text{if } n = 1, \\ n/2, & \text{if } n \text{ is even}, \\ 3n+1, & \text{if } n \text{ is odd}, \end{array}$$

   (See the Wikipedia entry on the *Collatz conjecture* if you are interested in the underlying question.) How many times you have to iterate your program starting at $n = 100$ ($n_1 = 100$, $n_2 = \texttt{collatz}(n_1)$, $n_3 = \texttt{collatz}(n_2)$, ...) before you get to 1? In your program, be careful of what type you are returning.

5. Explain and properly comment the following program.

```
def silly(y, x=3):
    z=x
    while(z>0):
        y = y+x
        z = z-1
    return y
```

   In other words, add docstrings, formatted as in §9.2 and §9.4 . In particular, explain what `x=3` does.

6. Consider the extended Euclidean algorithm as implemented in the second program listed in §6.2, Example 4, of the notes. Create a table of values of all the key variables for for each step of the `while` loop for the case $a = 24$, $b = 15$.

7. A bowl of marbles in your math classroom contains 2009 green marbles and 2010 red ones. Every time you go to class, you must pick 2 marbles. If you pick 2 marbles of the same color, your math professor generously adds a red marble to the bowl. If you pick 2 marbles of different colors, your math professor generously adds a green marble to the bowl. What is the color of the last marble and how many times (in a worst case scenario) do you have to go to class before the bowl is empty?

   (Okay, this can be solved with no programming, but if you can program this, you will get extra credit.)

References

[Be] D. Beazley, Python: **essential reference**, 3rd edition, Sams, 2006.

[B] N. Biggs, **Codes: An introduction to information, communication, and cryptography**, Springer, 2008.

[BG] *Beginner's Guide to Python* webpage
`http://wiki.python.org/moin/BeginnersGuide`

[BoP] **A Byte of Python** by Swaroop C H
`http://www.swaroopch.com/byteofpython/`
A Python book for inexperienced programmers, free electronic versions.

[Br] T. Brock, *Linear Feedback Shift Registers and Cyclic Codes in* Sage, Rose-Hulman Undergraduate Mathematics Journal, vol. 7, 2006.
`http://www.rose-hulman.edu/mathjournal/v7n2.php`
`http://www.usna.edu/Users/math/wdj/brock/`

[C] Ondrej Certik and others, SymPy,
`http://www.sympy.org/`

[DL] Erik Demaine, Charles Leiserson *Introduction to Algorithms*
`http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/`

This course teaches techniques for the design and analysis of efficient algorithms, emphasizing methods useful in practice. Topics covered include: sorting; search trees, heaps, and hashing; divide-and-conquer; dynamic programming; amortized analysis; graph algorithms; shortest paths; network flow; computational geometry; number-theoretic algorithms; polynomial and matrix calculations; caching; and parallel computing.

Textbook: **Introduction to Algorithms**, Second Edition, by Cormen, Leiserson, Rivest, and Stein.

Extra credit: If you watch all these lectures and turn in your lecture notes you will get extra credit for sm450.

[DIP] **Dive Into Python** by Mark Pilgrim
`http://www.diveintopython.org/`
A Python book for experienced programmers, free electronic versions.

[F] Stephen Ferg, *Debugging in* Python,
`http://pythonconquerstheuniverse.wordpress.com/category/the-python-debugger/`

[GG] Eric Grimson, John Guttag, *Introduction to Computer Science and Programming*, Fall 2008 course taught at MIT, which were videotaped and available at
`http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-00Fall-2008/CourseHome/index.htm`

Description: This course is aimed at students with little or no programming experience. It aims to provide students with an understanding of the role computation can play in solving problems. It also aims to help students, regardless of their major, to feel justifiably confident of their ability to write small programs that allow them to accomplish useful goals. The class will use the Python programming language.

This course uses [TP], [PP], and [PT].

Extra credit: If you watch all these lectures and turn in your lecture notes you will get extra credit for sm450.

[H] The photo of Grace Hopper's logbook was obtained from the U.S. Navy's history webpage:
`http://www.history.navy.mil/photos/pers-us/uspers-h/g-hoppr.htm`
See also
`http://en.wikipedia.org/wiki/Grace_Hopper`

[HSA] Profile: David A. Huffman, **Scientific American**, Sep. 1991, p. 54, p. 58.
`http://www.huffmancoding.com/david-huffman/scientific-american`

[HW] *Huffman codes*, Wikipedia
`http://en.wikipedia.org/wiki/Huffman_codes`

[HP] W. C. Huffman, V. Pless, **Fundamentals of error-correcting codes**, Cambridge Univ. Press, 2003.

[L] **Building Skills in Python** by Steven F. Lott
`http://homepage.mac.com/s_lott/books/python.html`
A Python book for experienced programmers, free electronic versions.

[LtP] Alan Gauld, *Learning to Program* (in Javascript and Python) webpages
`http://www.freenetpages.co.uk/hp/alan.gauld/`

[Lu] P. Lutus, *Learning Sage: The first steps* webpage
`http://www.arachnoid.com/sage/learning_sage.html`

[LA] M. Lutz, D. Ascher, **Learning Python**, 2nd edition, O'Reilly, 2004.

[MvOV] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone , **Handbook of Applied Cryptography**, CRC Press, 1996.
`http://www.cacr.math.uwaterloo.ca/hac/`
(All chapters are free online.)

[N] Peter Norvig, *Teach Yourself Programming in Ten Years*, at
`http://norvig.com/21-days.html`

[O] J. Oxley, **Matroid theory**, Oxford Univ. Press, 1992.

[PE] Project Euler website
`http://projecteuler.net/index.php?section=problems`

Extra credit: If you use Python or Sage to do a lot of "easy problems" or some "hard problems", you will get extra credit for sm450. (Turn in your programs print outs, the problem page, and the "congratulations page" for each one.)

[PG] Pramode C.E., *Python generator tricks*, **Linux Gazzette**, March 2004,
`http://linuxgazette.net/100/pramode.html`

[PI] Python idiom webpages

- *Code Like a Pythonista: Idiomatic Python* webpage, by David Goodger,
`http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html`
- *Python programming idioms* webpage, by Philip Guo
`http://www.stanford.edu/~pgbovine/python-idioms.htm`

- *Python Idioms and Efficiency* webpage, by Rob Knight
  http://jaynes.colorado.edu/PythonIdioms.html

[PMC] John Perry, lecture notes on a course titled *Mathematical Computing*
http://www.math.usm.edu/perry/mat305fa09/

[PP] Wikibooks **Python Programming**
http://en.wikibooks.org/wiki/Python_Programming

[PQR] *Python 2.5 Quick Reference*
http://rgruet.free.fr/PQR25/PQR2.5.html
Also a free pdf is available for download.

[PT] **An Introduction to Python** Guido van Rossum (Fred L. Drake, Jr., editor)
http://www.network-theory.co.uk/docs/pytut/
Concisely written introduction by the "father" of Python. See also
http://docs.python.org/tutorial/index.html

[Py] *Python Programming Language* – Official Website, http://www.python.org

[S] W. Stein and others, Sage- *a mathematical software system*,
http://www.sagemath.org/

[St] W. Stein, lecture notes on a course titled *Algebraic, Scientic, and Statistical Computing, an Open Source Approach Using Sage*,
http://wiki.wstein.org/2008/480a

[TP] **How to Think Like a Computer Scientist - Learning with Python** (2nd Edition) by Jeffrey Elkner, Allen B. Downey, and Chris Meyers
http://openbookproject.net//thinkCSpy
A Python book for inexperienced programmers, free electronic versions.

[U] Kirby Urner's website on programming and teaching Python and mathematics
http://www.4dsolutions.net/ocn/index.html

[Un] J. Unpingco, IPython videos
http://ipython.scipy.org/moin/Documentation and Sage
http://sage.math.washington.edu/home/wdj/expository/unpingco/

[YTPT] YouTube Python *tutorials*,
http://www.youtube.com/watch?v=4Mf0h3HphEA *Python Programming Tutorial - 1 - Installing Python*

[WT] Wikibooks **Non-Programmer's Tutorial for Python**
http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.0